

The Supplemental Isabelle/HOL Library

January 18, 2026

Contents

1	Implementation of Association Lists	21
1.1	<i>update</i> and <i>updates</i>	21
1.2	<i>delete</i>	24
1.3	<i>update-with-aux</i> and <i>delete-aux</i>	25
1.4	<i>restrict</i>	27
1.5	<i>clearjunk</i>	28
1.6	<i>map-ran</i>	30
1.7	<i>merge</i>	31
1.8	<i>compose</i>	32
1.9	<i>map-entry</i>	36
1.10	<i>map-default</i>	36
2	Axiomatic Declaration of Bounded Natural Functors	37
3	Generalized Corecursor Sugar (<i>corec</i> and friends)	37
3.1	Coinduction	38
4	A general “while” combinator	41
4.1	<i>while-option</i>	42
4.2	<i>while</i>	45
4.3	Termination, <i>lfp</i> and <i>gfp</i>	46
4.4	<i>while-Some</i> and <i>while-saturate</i>	49
4.5	Reflexive, transitive closure	51
5	The Bourbaki-Witt tower construction for transfinite iteration	53
5.1	Connect with the while combinator for executability on chain-finite lattices.	58
6	Division with modulus centered towards zero.	61
7	Order on characters	65

8	A generic phantom type	66
9	Cardinality of types	67
9.1	Preliminary lemmas	67
9.2	Cardinalities of types	67
9.3	Classes with at least 1 and 2	70
9.4	A type class for deciding finiteness of types	70
9.5	A type class for computing the cardinality of types	71
9.6	Instantiations for <i>card-UNIV</i>	71
10	Code setup for sets with cardinality type information	75
11	Eliminating pattern matches	78
12	Lazy types in generated code	78
12.1	The type <i>lazy</i>	79
12.2	Implementation	81
13	Test infrastructure for the code generator	81
13.1	YXML encoding for <i>term</i>	81
13.2	Test engine and drivers	84
14	A combinator to build partial equivalence relations from a predicate and an equivalence relation	84
15	Formalisation of chain-complete partial orders, continuity and admissibility	85
15.1	Continuity	92
15.1.1	Theorem collection <i>cont-intro</i>	92
15.2	Admissibility	103
15.3	(=) as order	107
15.4	ccpo for products	108
15.5	Complete lattices as ccpo	114
15.6	Parallel fixpoint induction	119
16	Confluence	124
17	Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums	130
17.1	The datatype universe	130
17.2	Freeness: Distinctness of Constructors	132
17.3	Set Constructions	136

18 Bijections between natural numbers and other types	140
18.1 Type $\text{nat} \times \text{nat}$	140
18.2 Type $\text{nat} + \text{nat}$	142
18.3 Type int	143
18.4 Type nat list	144
18.5 Finite sets of naturals	145
18.5.1 Preliminaries	145
18.5.2 From sets to naturals	146
18.5.3 From naturals to sets	146
18.5.4 Proof of isomorphism	147
19 Encoding (almost) everything into natural numbers	148
19.1 The class of countable types	148
19.2 Conversion functions	149
19.3 Finite types are countable	149
19.4 Automatically proving countability of old-style datatypes	149
19.5 Automatically proving countability of datatypes	152
19.6 More Countable types	153
19.7 The rationals are countably infinite	154
20 Infinite Sets and Related Concepts	155
20.1 The set of natural numbers is infinite	155
20.2 The set of integers is also infinite	156
20.3 Infinitely Many and Almost All	157
20.4 Enumeration of an Infinite Set	160
20.5 Properties of <i>wellorder-class.enumerate</i> on finite sets	164
21 Countable sets	168
21.1 Predicate for countable sets	168
21.2 Enumerate a countable set	169
21.3 Closure properties of countability	172
21.4 Misc lemmas	176
21.5 Uncountable	177
22 Countable Complete Lattices	178
22.0.1 Instances of countable complete lattices	184
23 Type of (at Most) Countable Sets	185
23.1 Cardinal stuff	185
23.2 The type of countable sets	186
23.3 Additional lemmas	192
23.3.1 <i>empty</i>	192
23.3.2 <i>cinsert</i>	192
23.3.3 <i>cimage</i>	192

23.3.4	bounded quantification	193
23.3.5	<i>cUnion</i>	193
23.4	Setup for Lifting/Transfer	193
23.4.1	Relator and predicator properties	193
23.4.2	Transfer rules for the Transfer package	194
23.5	Registration as BNF	195
24	Debugging facilities for code generated towards Isabelle/ML	198
25	Sequence of Properties on Subsequences	198
26	Common discrete functions	201
26.1	Discrete logarithm	201
26.2	Discrete square root	204
27	Pi and Function Sets	210
27.1	Basic Properties of <i>Pi</i>	211
27.2	Composition With a Restricted Domain: <i>compose</i>	213
27.3	Bounded Abstraction: <i>restrict</i>	213
27.4	Bijections Between Sets	215
27.5	Extensionality	215
27.6	Cardinality	217
27.7	Extensional Function Spaces	217
27.7.1	Injective Extensional Function Spaces	222
27.7.2	Misc properties of functions, composition and restriction from HOL Light	223
27.7.3	Cardinality	224
27.8	The pigeonhole principle	226
27.9	Products of sums	227
28	Partitions and Disjoint Sets	228
28.1	Set of Disjoint Sets	228
28.1.1	Family of Disjoint Sets	229
28.2	Construct Disjoint Sequences	233
28.3	Partitions	234
28.4	Constructions of partitions	234
28.5	Finiteness of partitions	235
28.6	Equivalence of partitions and equivalence classes	235
28.7	Refinement of partitions	238
28.8	The coarsest common refinement of a set of partitions	239
29	Type of finite sets defined as a subtype of sets	241
29.1	Definition of the type	241
29.2	Basic operations and type class instantiations	242
29.3	Other operations	245

29.4	Transferred lemmas from Set.thy	247
29.5	Additional lemmas	263
29.5.1	<i>ffUnion</i>	263
29.5.2	<i>fbind</i>	264
29.5.3	<i>fsingleton</i>	264
29.5.4	<i>fempty</i>	264
29.5.5	<i>fset</i>	264
29.5.6	<i>ffilter</i>	264
29.5.7	<i>fset-of-list</i>	265
29.5.8	<i>finsert</i>	265
29.5.9	<i>fimage</i>	265
29.5.10	bounded quantification	266
29.5.11	<i>fcard</i>	267
29.5.12	<i>sorted-list-of-fset</i>	268
29.5.13	<i>ffold</i>	268
29.5.14	(\subset)	270
29.5.15	Group operations	270
29.5.16	Semilattice operations	270
29.6	Choice in fsets	273
29.7	Induction and Cases rules for fsets	273
29.8	Lemmas depending on induction	274
29.9	Setup for Lifting/Transfer	274
29.9.1	Relator and predicator properties	274
29.9.2	Transfer rules for the Transfer package	275
29.10	BNF setup	277
29.11	Size setup	279
29.12	Advanced relator customization	279
29.12.1	Countability	281
29.13	Quickcheck setup	281
29.14	Code Generation Setup	283
30	Type of finite maps defined as a subtype of maps	283
30.1	Auxiliary constants and lemmas over <i>map</i>	283
30.2	Abstract characterisation	286
30.3	Operations	286
30.4	BNF setup	301
30.5	<i>size</i> setup	306
30.6	Additional operations	306
30.7	Additional properties	308
30.8	Lifting/transfer setup	308
30.9	View as datatype	309
30.10	Code setup	310
30.11	Instances	312
30.12	Tests	314

31 Disjoint FSets	314
32 Lists with elements distinct as canonical example for datatype invariants	316
32.1 The type of distinct lists	316
32.2 Executable version obeying invariant	318
32.3 Induction principle and case distinction	319
32.4 Functorial structure	320
32.5 Quickcheck generators	320
32.6 BNF instance	320
33 Type of dual ordered lattices	322
33.1 Pointwise ordering	324
33.2 Binary infimum and supremum	325
33.3 Top and bottom elements	326
33.4 Complement	327
33.5 Complete lattice operations	328
34 Equipollence and Other Relations Connected with Cardinality	330
34.1 Eqpoll	330
34.2 The strict relation	334
34.3 Mapping by an injection	335
34.4 Inserting elements into sets	336
34.5 Binary sums and unions	337
34.6 Binary Cartesian products	338
34.7 General Unions	340
34.8 General Cartesian products (Pi)	341
34.9 Misc other resultd	346
35 Continuity and iterations	351
35.1 Continuity for complete lattices	351
35.1.1 Least fixed points in countable complete lattices . . .	359
36 Extended natural numbers (i.e. with infinity)	360
36.1 Type definition	361
36.2 Constructors and numbers	362
36.3 Addition	363
36.4 Multiplication	364
36.5 Numerals	365
36.6 Subtraction	366
36.7 Ordering	366
36.8 Cancellation simprocs	370
36.9 Well-ordering	372

36.10	Complete Lattice	372
36.11	Traditional theorem names	373
37	Liminf and Limsup on conditionally complete lattices	374
37.0.1	<i>Liminf</i> and <i>Limsup</i>	375
37.1	More Limits	386
38	Extended real number line	387
38.1	Definition and basic properties	392
38.1.1	Addition	394
38.1.2	Linear order on <i>ereal</i>	397
38.1.3	Multiplication	404
38.1.4	Power	412
38.1.5	Subtraction	412
38.1.6	Division	416
38.2	Complete lattice	421
38.3	Extended real intervals	423
38.4	Topological space	427
38.5	Relation to <i>enat</i>	437
38.6	Limits on <i>ereal</i>	439
38.6.1	Convergent sequences	442
38.6.2	Sums	452
38.6.3	Continuity	463
38.6.4	liminf and limsup	466
38.6.5	Tests for code generator	471
39	Indicator Function	471
40	The type of non-negative extended real numbers	476
40.1	Defining the extended non-negative reals	479
40.2	Cancellation simprocs	483
40.3	Order with top	485
40.4	Arithmetic	488
40.5	Coercion from <i>real</i> to <i>ennreal</i>	492
40.6	Coercion from <i>ennreal</i> to <i>real</i>	497
40.7	Coercion from <i>enat</i> to <i>ennreal</i>	498
40.8	Topology on <i>ennreal</i>	500
40.9	Approximation lemmas	512
40.10	<i>ennreal</i> theorems	514
41	Logarithm of Natural Numbers	519
41.1	Preliminaries	519
41.2	Floorlog	519
41.3	523

41.4 Bitlen	527
42 Various algebraic structures combined with a lattice	528
42.1 Positive Part, Negative Part, Absolute Value	531
43 Floating-Point Numbers	539
43.1 Real operations preserving the representation as floating point number	540
43.2 Arithmetic operations on floating point numbers	543
43.3 Quickcheck	545
43.4 Represent floats as unique mantissa and exponent	546
43.5 Compute arithmetic operations	549
43.6 Lemmas for types <i>real</i> , <i>nat</i> , <i>int</i>	552
43.7 Rounding Real Numbers	552
43.8 Rounding Floats	554
43.9 Truncating Real Numbers	558
43.10 Truncating Floats	560
43.11 Approximation of positive rationals	565
43.12 Division	569
43.13 Approximate Addition	570
43.14 Approximate Multiplication	578
43.15 Approximate Power	579
43.16 Lemmas needed by Approximate	584
44 Pointwise instantiation of functions to algebra type classes	589
45 Pointwise instantiation of functions to division	593
45.1 Syntactic with division	594
46 Lexicographic order on functions	595
47 The <i>going-to</i> filter	597
48 Big sum and product over function bodies	599
48.1 Abstract product	599
48.2 Concrete sum	603
48.3 Concrete product	604
49 Infinite Type Class	605
50 Algebraic operations on sets	607
51 Interval Type	614
51.1 Membership	618
51.2 Quickcheck	630

52 Approximate Operations on Intervals of Floating Point Numbers	632
52.1 Intervals with Floating Point Bounds	633
52.2 intros for <i>real-interval</i>	634
52.3 bounds for lists	635
52.4 constants for code generation	639
53 Immutable Arrays with Code Generation	639
53.1 Fundamental operations	640
53.2 Generic code equations	640
53.3 Auxiliary operations for code generation	641
53.4 Code Generation for SML	643
53.5 Code Generation for Haskell	643
54 Definition of Landau symbols	644
54.1 Definition of Landau symbols	644
54.2 Landau symbols and limits	667
54.3 Flatness of real functions	680
54.4 Asymptotic Equivalence	681
55 Values extended by a bottom element	693
55.1 Values extended by a top element	696
55.2 Values extended by a top and a bottom element	698
56 Infinite Streams	703
56.1 prepend list to stream	704
56.2 set of streams with elements in some fixed set	705
56.3 nth, take, drop for streams	706
56.4 unary predicates lifted to streams	709
56.5 recurring stream out of a list	709
56.6 iterated application of a function	711
56.7 stream repeating a single element	711
56.8 stream of natural numbers	712
56.9 flatten a stream of lists	712
56.10 merge a stream of streams	713
56.11 product of two streams	714
56.12 interleave two streams	714
56.13 zip	715
56.14 zip via function	716
57 List prefixes, suffixes, and homeomorphic embedding	716
57.1 Prefix order on lists	717
57.2 Basic properties of prefixes	718
57.3 Prefixes	722

57.4 Longest Common Prefix	723
57.5 Parallel lists	726
57.6 Suffix order on lists	727
57.7 Suffixes	732
57.8 Homeomorphic embedding on lists	734
57.9 Subsequences (special case of homeomorphic embedding) . . .	737
57.10 Appending elements	739
57.11 Relation to standard list operations	741
57.12 Contiguous sublists	742
57.12.1 <i>sublist</i>	742
57.12.2 <i>sublists</i>	746
57.13 Parametricity	746
58 Linear Temporal Logic on Streams	748
59 Preliminaries	748
60 Linear temporal logic	748
61 Weak vs. strong until (contributed by Michael Foster, University of Sheffield)	764
62 Lists as vectors	766
62.1 $+$ and $-$	766
62.2 Inner product	768
63 Definitions of Least Upper Bounds and Greatest Lower Bounds	769
63.1 Rules for the Relations $* \leq$ and $\leq *$	769
63.2 Rules about the Operators <i>leastP</i> , <i>ub</i> and <i>lub</i>	770
63.3 Rules about the Operators <i>greatestP</i> , <i>isLb</i> and <i>isGlb</i>	771
64 An abstract view on maps for code generation.	774
64.1 Parametricity transfer rules	774
64.2 Type definition and primitive operations	777
64.3 Functorial structure	778
64.4 Derived operations	778
64.5 Properties	779
64.5.1 <i>entries</i> , <i>ordered-entries</i> , and <i>fold</i>	789
64.6 Code generator setup	795
65 Monad notation for arbitrary types	795
66 Less common functions on lists	797

67 (Finite) Multisets	807
67.1 The type of multisets	807
67.2 Representing multisets	807
67.3 Basic operations	809
67.3.1 Conversion to set and membership	809
67.3.2 Union	812
67.3.3 Difference	812
67.3.4 Min and Max	815
67.3.5 Equality of multisets	815
67.3.6 Pointwise ordering induced by count	817
67.3.7 Intersection and bounded union	821
67.3.8 Additional intersection facts	822
67.3.9 Additional bounded union facts	824
67.4 Replicate and repeat operations	825
67.4.1 Simprocs	827
67.4.2 Conditionally complete lattice	828
67.4.3 Filter (with comprehension syntax)	833
67.4.4 Size	835
67.5 Induction and case splits	837
67.5.1 Strong induction and subset induction for multisets	839
67.6 Least and greatest elements	840
67.7 The fold combinator	841
67.8 Image	842
67.9 Further conversions	848
67.10 More properties of the replicate, repeat, and image operations	856
67.11 Big operators	862
67.12 Multiset as order-ignorant lists	871
67.13 The multiset order	875
67.13.1 Well-foundedness	876
67.13.2 Closure-free presentation	879
67.13.3 Monotonicity	881
67.13.4 The multiset extension is cancellative for multiset union	884
67.13.5 Strict partial-order properties	886
67.13.6 Strict total-order properties	888
67.14 Quasi-executable version of the multiset extension	890
67.14.1 Monotonicity of multiset union	892
67.14.2 Termination proofs with multiset orders	893
67.15 Legacy theorem bindings	896
67.16 Naive implementation using lists	897
67.17 BNF setup	901
67.18 Size setup	907
67.19 Lemmas about Size	908
67.20 The set of multisets of a given size	909

68 More Theorems about the Multiset Order	913
68.1 Alternative Characterizations	913
68.1.1 The Dershowitz–Manna Ordering	913
68.1.2 The Huet–Oppen Ordering	913
68.1.3 Monotonicity	917
68.1.4 Properties of Orders	917
68.1.5 Simplifications	927
68.2 Simprocs	928
68.3 Additional facts and instantiations	929
69 Fixed Length Lists	932
70 Non-negative, non-positive integers and reals	934
70.1 Non-positive integers	934
70.2 Non-negative reals	937
70.3 Non-positive reals	939
71 Numeral Syntax for Types	942
71.1 Numeral Types	943
71.2 <i>num1</i>	943
71.3 Locales for modular arithmetic subtypes	945
71.4 Ring class instances	947
71.5 Order instances	949
71.6 Code setup and type classes for code generation	950
71.7 Syntax	953
71.8 Examples	954
72 ω-words	954
72.1 Type declaration and elementary operations	955
72.2 Subsequence, Prefix, and Suffix	956
72.3 Prepending	960
72.4 The limit set of an ω -word	962
72.5 Index sequences and piecewise definitions	969
73 Combinator syntax for generic, open state monads (single-threaded monads)	973
73.1 Motivation	973
73.2 State transformations and combinators	974
73.3 Monad laws	975
73.4 Do-syntax	975
74 Canonical order on option type	976

75 Futures and parallel lists for code generated towards Isabelle/ML	986
75.1 Futures	986
75.2 Parallel lists	986
76 Input syntax for pattern aliases (or “as-patterns” in Haskell)	987
76.1 Definition	988
76.2 Usage	991
77 Periodic Functions	991
78 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view	995
78.1 Preliminary: auxiliary operations for <i>almost everywhere zero</i> .	995
78.2 Type definition	999
78.3 Additive structure	1000
78.4 Multiplicative structure	1003
78.5 Single-point mappings	1008
78.6 Integral domains	1010
78.7 Mapping order	1012
78.8 Fundamental mapping notions	1014
78.9 Degree	1016
78.10 Inductive structure	1018
78.11 Quasi-functorial structure	1019
78.12 Canonical dense representation of $nat \Rightarrow_0 'a$	1021
78.13 Canonical sparse representation of $'a \Rightarrow_0 'b$	1023
78.14 Size estimation	1025
78.15 Further mapping operations and properties	1027
78.16 Free Abelian Groups Over a Type	1027
79 Exponentiation by Squaring	1032
80 Preorders with explicit equivalence relation	1033
81 Additive group operations on product types	1035
81.1 Operations	1035
81.2 Class instances	1037
82 Roots of real quadratics	1038
83 Pretty syntax for Quotient operations	1042
84 Quotient infrastructure for the set type	1042
84.1 Contravariant set map (vimage) and set relator, rules for the Quotient package	1042

85 Quotient infrastructure for the product type	1044
85.1 Rules for the Quotient package	1044
86 Quotient infrastructure for the option type	1047
86.1 Rules for the Quotient package	1047
87 Quotient infrastructure for the list type	1048
87.1 Rules for the Quotient package	1048
88 Quotient infrastructure for the sum type	1052
88.1 Rules for the Quotient package	1052
89 Quotient types	1054
89.1 Equivalence relations and quotient types	1054
89.2 Equality on quotients	1056
89.3 Picking representing elements	1057
90 Ramsey’s Theorem	1058
90.1 Preliminary definitions	1058
90.1.1 The n -element subsets of a set A	1058
90.1.2 Further properties, involving equipollence	1062
90.1.3 Partition predicates	1064
90.2 Finite versions of Ramsey’s theorem	1066
90.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers	1066
90.2.2 Trivial cases	1067
90.2.3 Ramsey’s theorem with TWO colours and arbitrary exponents (hypergraph version)	1068
90.2.4 Full Ramsey’s theorem with multiple colours and ar- bitrary exponents	1073
90.2.5 Simple graph version	1075
90.3 Preliminaries for the infinitary version	1076
90.3.1 “Axiom” of Dependent Choice	1076
90.3.2 Partition functions	1077
90.4 Ramsey’s Theorem: Infinitary Version	1077
90.5 Disjunctive Well-Foundedness	1081
91 Modulo and congruence on the reals	1083
92 Generic reflection and reification	1089
93 Assigning lengths to types by type classes	1091

94 Saturated arithmetic	1093
94.1 The type of saturated naturals	1093
94.2 Enumeration	1098
95 Set Idioms	1099
95.1 Idioms for being a suitable union/intersection of something .	1099
95.2 The “Relative to” operator	1106
96 Signed division: negative results rounded towards zero rather than minus infinity.	1114
97 State monad	1119
98 Comparators on linear quasi-orders	1124
98.1 Basic properties	1124
98.2 Fundamental comparator combinators	1128
98.3 Direct implementations for linear orders on selected types . .	1130
99 Stably sorted lists	1131
100 Alternative sorting algorithms	1139
100.1 Quicksort	1139
100.2 Mergesort	1141
100.3 Lexicographic products	1145
101A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming	1147
102 Time functions for various standard library operations. Also defines <i>itrev</i>.	1148
103A table-based implementation of the reflexive transitive closure	1150
104 Binary Tree	1155
104.1 <i>map-tree</i>	1157
104.2 <i>size</i>	1157
104.3 <i>set-tree</i>	1158
104.4 <i>subtrees</i>	1158
104.5 <i>height</i> and <i>min-height</i>	1158
104.6 <i>complete</i>	1159
104.7 <i>acomplete</i>	1161
104.8 <i>wbalanced</i>	1162
104.9 <i>ipl</i>	1162

104.10	List of entries	1162
104.11	Binary Search Tree	1163
104.12	Heap	1163
104.13	Mirror	1163
105	Multiset of Elements of Binary Tree	1164
106	Unordered pairs	1167
107A	type of finite bit strings	1172
107.1	Preliminaries	1172
107.2	Fundamentals	1173
107.2.1	Type definition	1173
107.2.2	Basic arithmetic	1173
107.2.3	Basic tool setup	1175
107.2.4	Basic code generation setup	1175
107.2.5	Basic conversions	1176
107.3	Elementary case distinctions	1183
107.3.1	Basic ordering	1183
107.4	Enumeration	1186
107.5	Bit-wise operations	1186
107.6	Conversions including casts	1197
107.6.1	Generic unsigned conversion	1197
107.6.2	Generic signed conversion	1199
107.6.3	More	1201
107.7	Arithmetic operations	1205
107.8	Ordering	1208
107.9	Bit-wise operations	1210
107.10	More shift operations	1213
107.11	Single-bit operations	1214
107.12	Rotation	1214
107.13	Split and cat operations	1217
107.14	More on conversions	1217
107.15	Testing bits	1221
107.16	Word Arithmetic	1225
107.17	Transferring goals from words to ints	1229
107.18	Order on fixed-length words	1231
107.19	Conditions for the addition (etc) of two words to overflow	1233
107.20	Some proof tool support	1236
107.21	More on overflows and monotonicity	1238
107.22	Arithmetic type class instantiations	1242
107.23	Word and nat	1243
107.24	Cardinality, finiteness of set of words	1247
107.25	Bitwise Operations on Words	1247

107.25.1	Shift functions in terms of lists of bools	1252
107.25.2	Mask	1254
107.25.3	Slices	1256
107.25.4	Revcast	1258
107.26	Split and cat	1258
107.26.1	Split and slice	1259
107.27	Rotation	1260
107.27.1	Word rotation commutes with bit-wise operations . .	1262
107.28	Maximum machine word	1263
107.29	Recursion combinator for words	1267
107.30	Some more naive computations rules	1268
107.31	Executable intervals	1270
107.32	Tool support	1270
108	The Field of Integers mod 2	1270
109	Pointwise order on product types	1275
109.1	Pointwise ordering	1275
109.2	Binary infimum and supremum	1276
109.3	Top and bottom elements	1277
109.4	Complete lattice operations	1278
109.5	Complete distributive lattices	1279
109.6	Bekic's Theorem	1280
110	Finite Lattices	1281
110.1	Finite Complete Lattices	1281
110.2	Finite Distributive Lattices	1284
110.3	Linear Orders	1285
110.4	Finite Linear Orders	1286
111	Lexicographic order on lists	1287
112	Lexicographic order on lists	1289
113	Prefix order on lists as order class instance	1291
114	Lexicographic order on product types	1292
115	Subsequence Ordering	1294
115.1	Definitions and basic lemmas	1294
116	Records based on BNF/datatype machinery	1296
117	Implementation of mappings with Association Lists	1298

118	Avoidance of pattern matching on natural numbers	1305
118.1	Case analysis	1305
118.2	Preprocessors	1305
118.3	Candidates which need special treatment	1307
119	Implementation of natural numbers as binary numerals	1307
119.1	Representation	1307
119.2	Basic arithmetic	1308
119.3	Conversions	1310
120	Code generation of prolog programs	1310
121	Setup for Numerals	1310
122	Implementation of integer numbers by target-language integers	1310
123	Implementation of natural numbers by target-language integers	1318
123.1	Implementation for <i>nat</i>	1318
124	Implementation of natural and integer numbers by target-language integers	1323
125	Preprocessor setup for floats implemented by target language numerals	1323
126	Abstract type of association lists with unique keys	1324
126.1	Preliminaries	1325
126.2	Type (<i>'key, 'value</i>) <i>alist</i>	1325
126.3	Primitive operations	1325
126.4	Abstract operation properties	1326
126.5	Further operations	1326
126.5.1	Equality	1326
126.5.2	Size	1327
126.6	Quickcheck generators	1327
127	alist is a BNF	1329
128	Multisets partially implemented by association lists	1329
129	Implementation of Red-Black Trees	1339
129.1	Datatype of RB trees	1339
129.2	Tree properties	1339
129.2.1	Content of a tree	1339
129.2.2	Search tree properties	1340

129.2.3 Tree lookup	1341
129.2.4 Red-black properties	1345
129.3 Insertion	1345
129.4 Deletion	1350
129.5 Modifying existing entries	1360
129.6 Mapping all entries	1361
129.7 Folding over entries	1362
129.8 Bulkloading a tree	1362
129.9 Building a RBT from a sorted list	1363
129.10 Union and intersection of sorted associative lists	1376
129.1 Code generator setup	1404
130 Abstract type of RBT trees	1406
130.1 Type definition	1406
130.2 Primitive operations	1407
130.3 Derived operations	1408
130.4 Abstract lookup properties	1408
130.5 Quickcheck generators	1411
130.6 Hide implementation details	1411
131 Implementation of mappings with Red-Black Trees	1412
131.1 Data type and invariant	1412
131.2 Operations	1412
131.3 Invariant preservation	1413
131.4 Map Semantics	1413
132 Implementation of sets using RBT trees	1414
133 Definition of code datatype constructors	1414
134 Lemmas	1414
134.1 Auxiliary lemmas	1414
134.2 fold and filter	1414
134.3 foldi and Ball	1415
134.4 foldi and Bex	1415
134.5 folding over non empty trees and selecting the minimal and maximal element	1416
134.5.1 concrete	1416
134.5.2 abstract	1420
135 Code equations	1422
136 Introduction	1430
137 Termination	1431

138	Partial Functions	1432
139	Higher-Order Functions	1432
139.1	Limitations	1433
140	Predefined Functions	1434
141	Locales	1434
142	Fine Points	1435
143	Common constants	1437
144	Pairs	1438
145	Filters	1438
146	Bounded quantifiers	1438
147	Operations on Predicates	1438
148	Setup for Numerals	1438
149	Arithmetic operations	1439
149.1	Arithmetic on naturals and integers	1439
149.2	Inductive definitions for ordering on naturals	1440
150	Alternative list definitions	1441
150.1	Alternative rules for <i>length</i>	1441
150.2	Alternative rules for <i>list-all2</i>	1441
150.3	Alternative rules for membership in lists	1442
151	Setup for String.literal	1442
152	Simplification rules for optimisation	1442
153	A Prototype of Quickcheck based on the Predicate Com- piler	1443
154	TFL: recursive function definitions	1443
154.1	Lemmas for TFL	1443
154.2	Rule setup	1444
155	Program extraction from proofs involving datatypes and in- ductive predicates	1445
156	Refute	1445

1 Implementation of Association Lists

```
theory AList
  imports Main
begin
```

```
context
begin
```

The operations preserve distinctness of keys and function *clearjunk* distributes over them. Since *clearjunk* enforces distinctness of keys it can be used to establish the invariant, e.g. for inductive proofs.

1.1 *update* and *updates*

```
qualified primrec update :: 'key  $\Rightarrow$  'val  $\Rightarrow$  ('key  $\times$  'val) list  $\Rightarrow$  ('key  $\times$  'val) list
  where
    update k v [] = [(k, v)]
    | update k v (p # ps) = (if fst p = k then (k, v) # ps else p # update k v ps)
```

```
lemma update-conv': map-of (update k v al) = (map-of al)( $k \mapsto v$ )
  by (induct al) (auto simp add: fun-eq-iff)
```

```
corollary update-conv: map-of (update k v al) k' = ((map-of al)( $k \mapsto v$ )) k'
  by (simp add: update-conv')
```

```
lemma dom-update: fst ` set (update k v al) = {k}  $\cup$  fst ` set al
  by (induct al) auto
```

```
lemma update-keys:
  map fst (update k v al) =
    (if k  $\in$  set (map fst al) then map fst al else map fst al @ [k])
  by (induct al) simp-all
```

```
lemma distinct-update:
  assumes distinct (map fst al)
  shows distinct (map fst (update k v al))
  using assms by (simp add: update-keys)
```

```
lemma update-filter:
  a  $\neq$  k  $\implies$  update k v [q  $\leftarrow$  ps. fst q  $\neq$  a] = [q  $\leftarrow$  update k v ps. fst q  $\neq$  a]
  by (induct ps) auto
```

```
lemma update-triv: map-of al k = Some v  $\implies$  update k v al = al
  by (induct al) auto
```

```
lemma update-nonempty [simp]: update k v al  $\neq$  []
  by (induct al) auto
```

lemma *update-eqD*: $\text{update } k \ v \ al = \text{update } k \ v' \ al' \implies v = v'$

proof (*induct al arbitrary: al'*)

case *Nil*

then show ?*case*

by (*cases al'*) (*auto split: if-split-asm*)

next

case *Cons*

then show ?*case*

by (*cases al'*) (*auto split: if-split-asm*)

qed

lemma *update-last* [*simp*]: $\text{update } k \ v \ (\text{update } k \ v' \ al) = \text{update } k \ v \ al$

by (*induct al*) *auto*

Note that the lists are not necessarily the same: $\text{update } k \ v \ (\text{update } k' \ v' \ []) = [(k', v'), (k, v)]$ and $\text{update } k' \ v' \ (\text{update } k \ v \ []) = [(k, v), (k', v')]$.

lemma *update-swap*:

$k \neq k' \implies \text{map-of } (\text{update } k \ v \ (\text{update } k' \ v' \ al)) = \text{map-of } (\text{update } k' \ v' \ (\text{update } k \ v \ al))$

by (*simp add: update-conv' fun-eq-iff*)

lemma *update-Some-unfold*:

$\text{map-of } (\text{update } k \ v \ al) \ x = \text{Some } y \iff$

$x = k \wedge v = y \vee x \neq k \wedge \text{map-of } al \ x = \text{Some } y$

by (*simp add: update-conv' map-upd-Some-unfold*)

lemma *image-update* [*simp*]: $x \notin A \implies \text{map-of } (\text{update } x \ y \ al) \ `A = \text{map-of } al \ `A$

by (*auto simp add: update-conv'*)

qualified definition *updates* ::

$'key \ list \Rightarrow 'val \ list \Rightarrow ('key \times 'val) \ list \Rightarrow ('key \times 'val) \ list$

where $\text{updates } ks \ vs = \text{fold } (\text{case-prod } \text{update}) \ (\text{zip } ks \ vs)$

lemma *updates-simps* [*simp*]:

$\text{updates } [] \ vs \ ps = ps$

$\text{updates } ks \ [] \ ps = ps$

$\text{updates } (k \# ks) \ (v \# vs) \ ps = \text{updates } ks \ vs \ (\text{update } k \ v \ ps)$

by (*simp-all add: updates-def*)

lemma *updates-key-simp* [*simp*]:

$\text{updates } (k \# ks) \ vs \ ps =$

$(\text{case } vs \text{ of } [] \Rightarrow ps \mid v \# vs \Rightarrow \text{updates } ks \ vs \ (\text{update } k \ v \ ps))$

by (*cases vs*) *simp-all*

lemma *updates-conv'*: $\text{map-of } (\text{updates } ks \ vs \ al) = (\text{map-of } al)(ks[\mapsto]vs)$

proof –

have $\text{map-of} \circ \text{fold } (\text{case-prod } \text{update}) \ (\text{zip } ks \ vs) =$

$\text{fold } (\lambda(k, v) f. f(k \mapsto v)) \ (\text{zip } ks \ vs) \circ \text{map-of}$

```

  by (rule fold-commute) (auto simp add: fun-eq-iff update-conv')
  then show ?thesis
  by (auto simp add: updates-def fun-eq-iff map-upds-fold-map-upd foldl-conv-fold
split-def)
qed

```

```

lemma updates-conv: map-of (updates ks vs al) k = ((map-of al)(ks[↦]vs)) k
  by (simp add: updates-conv')

```

```

lemma distinct-updates:
  assumes distinct (map fst al)
  shows distinct (map fst (updates ks vs al))
proof -
  have distinct (fold
    (λ(k, v) al. if k ∈ set al then al else al @ [k])
    (zip ks vs) (map fst al))
  by (rule fold-invariant [of zip ks vs λ-. True]) (auto intro: assms)
  moreover have map fst ∘ fold (case-prod update) (zip ks vs) =
    fold (λ(k, v) al. if k ∈ set al then al else al @ [k]) (zip ks vs) ∘ map fst
  by (rule fold-commute) (simp add: update-keys split-def case-prod-beta comp-def)
  ultimately show ?thesis
  by (simp add: updates-def fun-eq-iff)
qed

```

```

lemma updates-append1[simp]: size ks < size vs ⟹
  updates (ks@[k]) vs al = update k (vs!size ks) (updates ks vs al)
  by (induct ks arbitrary: vs al) (auto split: list.splits)

```

```

lemma updates-list-update-drop[simp]:
  size ks ≤ i ⟹ i < size vs ⟹
  updates ks (vs[i:=v]) al = updates ks vs al
  by (induct ks arbitrary: al vs i) (auto split: list.splits nat.splits)

```

```

lemma update-updates-conv-if:
  map-of (updates xs ys (update x y al)) =
  map-of
    (if x ∈ set (take (length ys) xs)
     then updates xs ys al
     else (update x y (updates xs ys al)))
  by (simp add: updates-conv' update-conv' map-upd-upds-conv-if)

```

```

lemma updates-twist [simp]:
  k ∉ set ks ⟹
  map-of (updates ks vs (update k v al)) = map-of (update k v (updates ks vs al))
  by (simp add: updates-conv' update-conv')

```

```

lemma updates-apply-notin [simp]:
  k ∉ set ks ⟹ map-of (updates ks vs al) k = map-of al k
  by (simp add: updates-conv)

```

lemma *updates-append-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies \text{updates } (xs @ zs) \text{ } ys \text{ } al = \text{updates } xs \text{ } ys \text{ } al$
by (induct xs arbitrary: ys al) (auto split: list.splits)

lemma *updates-append2-drop* [simp]:
 $\text{size } xs = \text{size } ys \implies \text{updates } xs \text{ } (ys @ zs) \text{ } al = \text{updates } xs \text{ } ys \text{ } al$
by (induct xs arbitrary: ys al) (auto split: list.splits)

1.2 delete

qualified definition *delete* :: 'key \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list
where *delete-eq*: *delete* k = filter ($\lambda(k', -). k \neq k'$)

lemma *delete-simps* [simp]:
 $\text{delete } k \text{ } [] = []$
 $\text{delete } k \text{ } (p \# ps) = (\text{if } \text{fst } p = k \text{ then } \text{delete } k \text{ } ps \text{ else } p \# \text{delete } k \text{ } ps)$
by (auto simp add: delete-eq)

lemma *delete-conv'*: $\text{map-of } (\text{delete } k \text{ } al) = (\text{map-of } al)(k := \text{None})$
by (induct al) (auto simp add: fun-eq-iff)

corollary *delete-conv*: $\text{map-of } (\text{delete } k \text{ } al) \text{ } k' = ((\text{map-of } al)(k := \text{None})) \text{ } k'$
by (simp add: delete-conv')

lemma *delete-keys*: $\text{map } \text{fst } (\text{delete } k \text{ } al) = \text{removeAll } k \text{ } (\text{map } \text{fst } al)$
by (simp add: delete-eq removeAll-filter-not-eq filter-map split-def comp-def)

lemma *distinct-delete*:
assumes *distinct* (map fst al)
shows *distinct* (map fst (delete k al))
using *assms* **by** (simp add: delete-keys distinct-removeAll)

lemma *delete-id* [simp]: $k \notin \text{fst } ' \text{ set } al \implies \text{delete } k \text{ } al = al$
by (auto simp add: image-iff delete-eq filter-id-conv)

lemma *delete-idem*: $\text{delete } k \text{ } (\text{delete } k \text{ } al) = \text{delete } k \text{ } al$
by (simp add: delete-eq)

lemma *map-of-delete* [simp]: $k' \neq k \implies \text{map-of } (\text{delete } k \text{ } al) \text{ } k' = \text{map-of } al \text{ } k'$
by (simp add: delete-conv')

lemma *delete-notin-dom*: $k \notin \text{fst } ' \text{ set } (\text{delete } k \text{ } al)$
by (auto simp add: delete-eq)

lemma *dom-delete-subset*: $\text{fst } ' \text{ set } (\text{delete } k \text{ } al) \subseteq \text{fst } ' \text{ set } al$
by (auto simp add: delete-eq)

lemma *delete-update-same*: $\text{delete } k \text{ } (\text{update } k \text{ } v \text{ } al) = \text{delete } k \text{ } al$

by (*induct al*) *simp-all*

lemma *delete-update*: $k \neq l \implies \text{delete } l (\text{update } k \ v \ al) = \text{update } k \ v (\text{delete } l \ al)$
by (*induct al*) *simp-all*

lemma *delete-twist*: $\text{delete } x (\text{delete } y \ al) = \text{delete } y (\text{delete } x \ al)$
by (*simp add: delete-eq conj-commute*)

lemma *length-delete-le*: $\text{length } (\text{delete } k \ al) \leq \text{length } al$
by (*simp add: delete-eq*)

1.3 *update-with-aux* and *delete-aux*

qualified primrec *update-with-aux* ::
 $'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$
where
 $\text{update-with-aux } v \ k \ f \ [] = [(k, f \ v)]$
 $| \text{update-with-aux } v \ k \ f \ (p \# \ ps) =$
 $(\text{if } (\text{fst } p = k) \text{ then } (k, f \ (\text{snd } p)) \# \ ps \text{ else } p \# \text{update-with-aux } v \ k \ f \ ps)$

The above *delete* traverses all the list even if it has found the key. This one does not have to keep going because it assumes the invariant that keys are distinct.

qualified fun *delete-aux* :: $'key \Rightarrow ('key \times 'val) \text{ list} \Rightarrow ('key \times 'val) \text{ list}$
where
 $\text{delete-aux } k \ [] = []$
 $| \text{delete-aux } k \ ((k', v) \# \ xs) = (\text{if } k = k' \text{ then } xs \text{ else } (k', v) \# \text{delete-aux } k \ xs)$

lemma *map-of-update-with-aux'*:
 $\text{map-of } (\text{update-with-aux } v \ k \ f \ ps) \ k' =$
 $((\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \ \text{of } \text{None} \Rightarrow f \ v \mid \text{Some } v \Rightarrow f \ v))) \ k'$
by (*induct ps*) *auto*

lemma *map-of-update-with-aux*:
 $\text{map-of } (\text{update-with-aux } v \ k \ f \ ps) =$
 $(\text{map-of } ps)(k \mapsto (\text{case map-of } ps \ k \ \text{of } \text{None} \Rightarrow f \ v \mid \text{Some } v \Rightarrow f \ v))$
by (*simp add: fun-eq-iff map-of-update-with-aux'*)

lemma *dom-update-with-aux*: $\text{fst} \, ' \text{ set } (\text{update-with-aux } v \ k \ f \ ps) = \{k\} \cup \text{fst} \, ' \text{ set } ps$
by (*induct ps*) *auto*

lemma *distinct-update-with-aux* [*simp*]:
 $\text{distinct } (\text{map } \text{fst } (\text{update-with-aux } v \ k \ f \ ps)) = \text{distinct } (\text{map } \text{fst } ps)$
by (*induct ps*) (*auto simp add: dom-update-with-aux*)

lemma *set-update-with-aux*:
 $\text{distinct } (\text{map } \text{fst } xs) \implies$
 $\text{set } (\text{update-with-aux } v \ k \ f \ xs) =$

(set xs - {k} × UNIV ∪ {(k, f (case map-of xs k of None ⇒ v | Some v ⇒ v))}))

by (induct xs) (auto intro: rev-image-eqI)

lemma set-delete-aux: distinct (map fst xs) ⇒ set (delete-aux k xs) = set xs - {k} × UNIV

apply (induct xs)

apply simp-all

apply clarsimp

apply (fastforce intro: rev-image-eqI)

done

lemma dom-delete-aux: distinct (map fst ps) ⇒ fst ‘ set (delete-aux k ps) = fst ‘ set ps - {k}

by (auto simp add: set-delete-aux)

lemma distinct-delete-aux [simp]: distinct (map fst ps) ⇒ distinct (map fst (delete-aux k ps))

proof (induct ps)

case Nil

then show ?case by simp

next

case (Cons a ps)

obtain k' v where a: a = (k', v)

by (cases a)

show ?case

proof (cases k' = k)

case True

with Cons a show ?thesis by simp

next

case False

with Cons a have k' ∉ fst ‘ set ps distinct (map fst ps)

by simp-all

with False a have k' ∉ fst ‘ set (delete-aux k ps)

by (auto dest!: dom-delete-aux[where k=k])

with Cons a show ?thesis

by simp

qed

qed

lemma map-of-delete-aux':

distinct (map fst xs) ⇒ map-of (delete-aux k xs) = (map-of xs)(k := None)

apply (induct xs)

apply (fastforce simp add: map-of-eq-None-iff fun-upd-twist)

apply (auto intro!: ext)

apply (simp add: map-of-eq-None-iff)

done

lemma map-of-delete-aux:

$distinct \ (map \ fst \ xs) \implies map\text{-}of \ (delete\text{-}aux \ k \ xs) \ k' = ((map\text{-}of \ xs)(k := None))$
 k'

by (*simp add: map-of-delete-aux'*)

lemma *delete-aux-eq-Nil-conv*: $delete\text{-}aux \ k \ ts = [] \longleftrightarrow ts = [] \vee (\exists v. ts = [(k, v)])$

by (*cases ts (auto split: if-split-asm)*)

1.4 restrict

qualified definition *restrict* :: $'key \ set \Rightarrow ('key \times 'val) \ list \Rightarrow ('key \times 'val) \ list$
where *restrict-eq*: $restrict \ A = filter \ (\lambda(k, v). k \in A)$

lemma *restr-simps* [*simp*]:

$restrict \ A \ [] = []$

$restrict \ A \ (p \# ps) = (if \ fst \ p \in A \ then \ p \ \# \ restrict \ A \ ps \ else \ restrict \ A \ ps)$

by (*auto simp add: restrict-eq*)

lemma *restr-conv'*: $map\text{-}of \ (restrict \ A \ al) = ((map\text{-}of \ al)|^{'A})$

proof

show $map\text{-}of \ (restrict \ A \ al) \ k = ((map\text{-}of \ al)|^{'A}) \ k$ **for** k

apply (*induct al*)

apply *simp*

apply (*cases k ∈ A*)

apply *auto*

done

qed

corollary *restr-conv*: $map\text{-}of \ (restrict \ A \ al) \ k = ((map\text{-}of \ al)|^{'A}) \ k$

by (*simp add: restr-conv'*)

lemma *distinct-restr*: $distinct \ (map \ fst \ al) \implies distinct \ (map \ fst \ (restrict \ A \ al))$

by (*induct al (auto simp add: restrict-eq)*)

lemma *restr-empty* [*simp*]:

$restrict \ \{\} \ al = []$

$restrict \ A \ [] = []$

by (*induct al (auto simp add: restrict-eq)*)

lemma *restr-in* [*simp*]: $x \in A \implies map\text{-}of \ (restrict \ A \ al) \ x = map\text{-}of \ al \ x$

by (*simp add: restr-conv'*)

lemma *restr-out* [*simp*]: $x \notin A \implies map\text{-}of \ (restrict \ A \ al) \ x = None$

by (*simp add: restr-conv'*)

lemma *dom-restr* [*simp*]: $fst \ ^{'set} \ (restrict \ A \ al) = fst \ ^{'set} \ al \cap A$

by (*induct al (auto simp add: restrict-eq)*)

lemma *restr-upd-same* [*simp*]: $restrict \ (-\{x\}) \ (update \ x \ y \ al) = restrict \ (-\{x\}) \ al$

by (*induct al*) (*auto simp add: restrict-eq*)

lemma *restr-restr* [*simp*]: *restrict A (restrict B al) = restrict (A ∩ B) al*
by (*induct al*) (*auto simp add: restrict-eq*)

lemma *restr-update* [*simp*]:
map-of (restrict D (update x y al)) =
map-of ((if x ∈ D then (update x y (restrict (D - {x}) al)) else restrict D al))
by (*simp add: restr-conv' update-conv'*)

lemma *restr-delete* [*simp*]:
delete x (restrict D al) = (if x ∈ D then restrict (D - {x}) al else restrict D al)
apply (*simp add: delete-eq restrict-eq*)
apply (*auto simp add: split-def*)

proof –

have $y \neq x \longleftrightarrow x \neq y$ **for** y

by *auto*

then show $[p \leftarrow al. fst\ p \in D \wedge x \neq fst\ p] = [p \leftarrow al. fst\ p \in D \wedge fst\ p \neq x]$

by *simp*

assume $x \notin D$

then have $y \in D \longleftrightarrow y \in D \wedge x \neq y$ **for** y

by *auto*

then show $[p \leftarrow al. fst\ p \in D \wedge x \neq fst\ p] = [p \leftarrow al. fst\ p \in D]$

by *simp*

qed

lemma *update-restr*:
map-of (update x y (restrict D al)) = map-of (update x y (restrict (D - {x})
al))
by (*simp add: update-conv' restr-conv' (rule fun-upd-restrict)*)

lemma *update-restr-conv* [*simp*]:
 $x \in D \implies$
map-of (update x y (restrict D al)) = map-of (update x y (restrict (D - {x})
al))
by (*simp add: update-conv' restr-conv'*)

lemma *restr-updates* [*simp*]:
 $length\ xs = length\ ys \implies set\ xs \subseteq D \implies$
map-of (restrict D (updates xs ys al)) =
map-of (updates xs ys (restrict (D - set xs) al))
by (*simp add: updates-conv' restr-conv'*)

lemma *restr-delete-twist*: *(restrict A (delete a ps)) = delete a (restrict A ps)*
by (*induct ps*) *auto*

1.5 clearjunk

qualified function *clearjunk* :: $('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$

where
 $\text{clearjunk } [] = []$
 $| \text{clearjunk } (p \# ps) = p \# \text{clearjunk } (\text{delete } (\text{fst } p) \text{ } ps)$
by *pat-completeness auto*

termination
by *(relation measure length) (simp-all add: less-Suc-eq-le length-delete-le)*

lemma *map-of-clearjunk*: $\text{map-of } (\text{clearjunk } al) = \text{map-of } al$
by *(induct al rule: clearjunk.induct) (simp-all add: fun-eq-iff)*

lemma *clearjunk-keys-set*: $\text{set } (\text{map } \text{fst } (\text{clearjunk } al)) = \text{set } (\text{map } \text{fst } al)$
by *(induct al rule: clearjunk.induct) (simp-all add: delete-keys)*

lemma *dom-clearjunk*: $\text{fst } ' \text{ set } (\text{clearjunk } al) = \text{fst } ' \text{ set } al$
using *clearjunk-keys-set* **by** *simp*

lemma *distinct-clearjunk* [*simp*]: $\text{distinct } (\text{map } \text{fst } (\text{clearjunk } al))$
by *(induct al rule: clearjunk.induct) (simp-all del: set-map add: clearjunk-keys-set delete-keys)*

lemma *ran-clearjunk*: $\text{ran } (\text{map-of } (\text{clearjunk } al)) = \text{ran } (\text{map-of } al)$
by *(simp add: map-of-clearjunk)*

lemma *ran-map-of*: $\text{ran } (\text{map-of } al) = \text{snd } ' \text{ set } (\text{clearjunk } al)$
proof –
have $\text{ran } (\text{map-of } al) = \text{ran } (\text{map-of } (\text{clearjunk } al))$
by *(simp add: ran-clearjunk)*
also have $\dots = \text{snd } ' \text{ set } (\text{clearjunk } al)$
by *(simp add: ran-distinct)*
finally show *?thesis* .
qed

lemma *graph-map-of*: $\text{Map.graph } (\text{map-of } al) = \text{set } (\text{clearjunk } al)$
by *(metis distinct-clearjunk graph-map-of-if-distinct-dom map-of-clearjunk)*

lemma *clearjunk-update*: $\text{clearjunk } (\text{update } k \text{ } v \text{ } al) = \text{update } k \text{ } v \text{ } (\text{clearjunk } al)$
by *(induct al rule: clearjunk.induct) (simp-all add: delete-update)*

lemma *clearjunk-updates*: $\text{clearjunk } (\text{updates } ks \text{ } vs \text{ } al) = \text{updates } ks \text{ } vs \text{ } (\text{clearjunk } al)$
proof –
have $\text{clearjunk } \circ \text{fold } (\text{case-prod } \text{update}) (\text{zip } ks \text{ } vs) =$
 $\text{fold } (\text{case-prod } \text{update}) (\text{zip } ks \text{ } vs) \circ \text{clearjunk}$
by *(rule fold-commute) (simp add: clearjunk-update case-prod-beta o-def)*
then show *?thesis*
by *(simp add: updates-def fun-eq-iff)*
qed

lemma *clearjunk-delete*: $\text{clearjunk } (\text{delete } x \text{ } al) = \text{delete } x \text{ } (\text{clearjunk } al)$

by (induct al rule: clearjunk.induct) (auto simp add: delete-idem delete-twist)

lemma clearjunk-restrict: clearjunk (restrict A al) = restrict A (clearjunk al)
by (induct al rule: clearjunk.induct) (auto simp add: restr-delete-twist)

lemma distinct-clearjunk-id [simp]: distinct (map fst al) \implies clearjunk al = al
by (induct al rule: clearjunk.induct) auto

lemma clearjunk-idem: clearjunk (clearjunk al) = clearjunk al
by simp

lemma length-clearjunk: length (clearjunk al) \leq length al

proof (induct al rule: clearjunk.induct [case-names Nil Cons])

case Nil

then show ?case **by** simp

next

case (Cons kv al)

moreover have length (delete (fst kv) al) \leq length al

by (fact length-delete-le)

ultimately have length (clearjunk (delete (fst kv) al)) \leq length al

by (rule order-trans)

then show ?case

by simp

qed

lemma delete-map:

assumes $\bigwedge kv. \text{fst } (f \text{ kv}) = \text{fst } kv$

shows delete k (map f ps) = map f (delete k ps)

by (simp add: delete-eq filter-map comp-def split-def assms)

lemma clearjunk-map:

assumes $\bigwedge kv. \text{fst } (f \text{ kv}) = \text{fst } kv$

shows clearjunk (map f ps) = map f (clearjunk ps)

by (induct ps rule: clearjunk.induct [case-names Nil Cons])

(simp-all add: clearjunk-delete delete-map assms)

1.6 map-ran

definition map-ran :: ('key \Rightarrow 'val1 \Rightarrow 'val2) \Rightarrow ('key \times 'val1) list \Rightarrow ('key \times 'val2) list

where map-ran f = map ($\lambda(k, v). (k, f \text{ k } v)$)

lemma map-ran-simps [simp]:

map-ran f [] = []

map-ran f ((k, v) # ps) = (k, f k v) # map-ran f ps

by (simp-all add: map-ran-def)

lemma map-ran-Cons-sel: map-ran f (p # ps) = (fst p, f (fst p) (snd p)) # map-ran f ps

by (*simp add: map-ran-def case-prod-beta*)

lemma *length-map-ran[simp]*: $\text{length } (\text{map-ran } f \text{ al}) = \text{length } \text{al}$
by (*simp add: map-ran-def*)

lemma *map-fst-map-ran[simp]*: $\text{map } \text{fst } (\text{map-ran } f \text{ al}) = \text{map } \text{fst } \text{al}$
by (*simp add: map-ran-def case-prod-beta*)

lemma *dom-map-ran*: $\text{fst } \text{'set } (\text{map-ran } f \text{ al}) = \text{fst } \text{'set } \text{al}$
by (*simp add: map-ran-def image-image split-def*)

lemma *map-ran-conv*: $\text{map-of } (\text{map-ran } f \text{ al}) \text{ k} = \text{map-option } (f \text{ k}) (\text{map-of } \text{al } \text{k})$
by (*induct al*) *auto*

lemma *distinct-map-ran*: $\text{distinct } (\text{map } \text{fst } \text{al}) \implies \text{distinct } (\text{map } \text{fst } (\text{map-ran } f \text{ al}))$
by *simp*

lemma *map-ran-filter*: $\text{map-ran } f \text{ } [p \leftarrow \text{ps}. \text{fst } p \neq a] = [p \leftarrow \text{map-ran } f \text{ ps}. \text{fst } p \neq a]$
by (*simp add: map-ran-def filter-map split-def comp-def*)

lemma *clearjunk-map-ran*: $\text{clearjunk } (\text{map-ran } f \text{ al}) = \text{map-ran } f (\text{clearjunk } \text{al})$
by (*simp add: map-ran-def split-def clearjunk-map*)

1.7 merge

qualified definition *merge* :: $(\text{'key} \times \text{'val}) \text{ list} \Rightarrow (\text{'key} \times \text{'val}) \text{ list} \Rightarrow (\text{'key} \times \text{'val}) \text{ list}$
where $\text{merge } \text{qs } \text{ps} = \text{foldr } (\lambda(k, v). \text{update } k \text{ v}) \text{ ps } \text{qs}$

lemma *merge-simps [simp]*:
 $\text{merge } \text{qs } [] = \text{qs}$
 $\text{merge } \text{qs } (p \# \text{ps}) = \text{update } (\text{fst } p) (\text{snd } p) (\text{merge } \text{qs } \text{ps})$
by (*simp-all add: merge-def split-def*)

lemma *merge-updates*: $\text{merge } \text{qs } \text{ps} = \text{updates } (\text{rev } (\text{map } \text{fst } \text{ps})) (\text{rev } (\text{map } \text{snd } \text{ps})) \text{ qs}$
by (*simp add: merge-def updates-def foldr-conv-fold zip-rev zip-map-fst-snd*)

lemma *dom-merge*: $\text{fst } \text{'set } (\text{merge } \text{xs } \text{ys}) = \text{fst } \text{'set } \text{xs} \cup \text{fst } \text{'set } \text{ys}$
by (*induct ys arbitrary: xs*) (*auto simp add: dom-update*)

lemma *distinct-merge*: $\text{distinct } (\text{map } \text{fst } \text{xs}) \implies \text{distinct } (\text{map } \text{fst } (\text{merge } \text{xs } \text{ys}))$
by (*simp add: merge-updates distinct-updates*)

lemma *clearjunk-merge*: $\text{clearjunk } (\text{merge } \text{xs } \text{ys}) = \text{merge } (\text{clearjunk } \text{xs}) \text{ ys}$
by (*simp add: merge-updates clearjunk-updates*)

lemma *merge-conv'*: $\text{map-of } (\text{merge } xs \ ys) = \text{map-of } xs ++ \text{map-of } ys$

proof –

have $\text{map-of} \circ \text{fold } (\text{case-prod } \text{update}) (\text{rev } ys) =$
 $\text{fold } (\lambda(k, v) \ m. \ m(k \mapsto v)) (\text{rev } ys) \circ \text{map-of}$
 by (*rule fold-commute*) (*simp add: update-conv' case-prod-beta split-def fun-eq-iff*)
 then show *?thesis*
 by (*simp add: merge-def map-add-map-of-foldr foldr-conv-fold fun-eq-iff*)

qed

corollary *merge-conv*: $\text{map-of } (\text{merge } xs \ ys) \ k = (\text{map-of } xs ++ \text{map-of } ys) \ k$
by (*simp add: merge-conv'*)

lemma *merge-empty*: $\text{map-of } (\text{merge } [] \ ys) = \text{map-of } ys$
by (*simp add: merge-conv'*)

lemma *merge-assoc* [*simp*]: $\text{map-of } (\text{merge } m1 \ (\text{merge } m2 \ m3)) = \text{map-of } (\text{merge } (\text{merge } m1 \ m2) \ m3)$
by (*simp add: merge-conv'*)

lemma *merge-Some-iff*:

$\text{map-of } (\text{merge } m \ n) \ k = \text{Some } x \longleftrightarrow$
 $\text{map-of } n \ k = \text{Some } x \vee \text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{Some } x$
by (*simp add: merge-conv' map-add-Some-iff*)

lemmas *merge-SomeD* [*dest!*] = *merge-Some-iff* [*THEN iffD1*]

lemma *merge-find-right* [*simp*]: $\text{map-of } n \ k = \text{Some } v \implies \text{map-of } (\text{merge } m \ n) \ k = \text{Some } v$
by (*simp add: merge-conv'*)

lemma *merge-None* [*iff*]: $(\text{map-of } (\text{merge } m \ n) \ k = \text{None}) = (\text{map-of } n \ k = \text{None} \wedge \text{map-of } m \ k = \text{None})$
by (*simp add: merge-conv'*)

lemma *merge-upd* [*simp*]: $\text{map-of } (\text{merge } m \ (\text{update } k \ v \ n)) = \text{map-of } (\text{update } k \ v \ (\text{merge } m \ n))$
by (*simp add: update-conv' merge-conv'*)

lemma *merge-updatess* [*simp*]:

$\text{map-of } (\text{merge } m \ (\text{updates } xs \ ys \ n)) = \text{map-of } (\text{updates } xs \ ys \ (\text{merge } m \ n))$
by (*simp add: updates-conv' merge-conv'*)

lemma *merge-append*: $\text{map-of } (xs @ ys) = \text{map-of } (\text{merge } ys \ xs)$
by (*simp add: merge-conv'*)

1.8 compose

qualified function *compose* :: $('key \times 'a) \text{ list} \Rightarrow ('a \times 'b) \text{ list} \Rightarrow ('key \times 'b) \text{ list}$
where


```

    compose [] ys = []
  | compose (x # xs) ys =
    (case map-of ys (snd x) of
      None  $\Rightarrow$  compose (delete (fst x) xs) ys
    | Some v  $\Rightarrow$  (fst x, v) # compose xs ys)
  by pat-completeness auto
termination
  by (relation measure (length  $\circ$  fst)) (simp-all add: less-Suc-eq-le length-delete-le)

lemma compose-first-None [simp]: map-of xs k = None  $\implies$  map-of (compose xs
ys) k = None
  by (induct xs ys rule: compose.induct) (auto split: option.splits if-split-asm)

lemma compose-conv: map-of (compose xs ys) k = (map-of ys  $\circ_m$  map-of xs) k
proof (induct xs ys rule: compose.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 have hyp: map-of (compose (delete (fst x) xs) ys) k =
      (map-of ys  $\circ_m$  map-of (delete (fst x) xs)) k
    by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      from True delete-notin-dom [of k xs]
      have map-of (delete (fst x) xs) k = None
        by (simp add: map-of-eq-None-iff)
      with hyp show ?thesis
        using True None
        by simp
      case False
      from False have map-of (delete (fst x) xs) k = map-of xs k
        by simp
      with hyp show ?thesis
        using False None by (simp add: map-comp-def)
    qed
  next
  case (Some v)
  with 2
  have map-of (compose xs ys) k = (map-of ys  $\circ_m$  map-of xs) k
    by simp
  with Some show ?thesis
    by (auto simp add: map-comp-def)
  qed

```

qed

lemma *compose-conv'*: $\text{map-of } (\text{compose } xs \ ys) = (\text{map-of } ys \circ_m \text{map-of } xs)$
by (*rule ext*) (*rule compose-conv*)

lemma *compose-first-Some* [*simp*]: $\text{map-of } xs \ k = \text{Some } v \implies \text{map-of } (\text{compose } xs \ ys) \ k = \text{map-of } ys \ v$
by (*simp add: compose-conv*)

lemma *dom-compose*: $\text{fst } ' \text{ set } (\text{compose } xs \ ys) \subseteq \text{fst } ' \text{ set } xs$

proof (*induct xs ys rule: compose.induct*)

case 1

then show ?*case* **by** *simp*

next

case (2 *x xs ys*)

show ?*case*

proof (*cases map-of ys (snd x)*)

case *None*

with 2.*hyps* **have** $\text{fst } ' \text{ set } (\text{compose } (\text{delete } (\text{fst } x) \ xs) \ ys) \subseteq \text{fst } ' \text{ set } (\text{delete } (\text{fst } x) \ xs)$

by *simp*

also have $\dots \subseteq \text{fst } ' \text{ set } xs$

by (*rule dom-delete-subset*)

finally show ?*thesis*

using *None* **by** *auto*

next

case (*Some v*)

with 2.*hyps* **have** $\text{fst } ' \text{ set } (\text{compose } xs \ ys) \subseteq \text{fst } ' \text{ set } xs$

by *simp*

with *Some* **show** ?*thesis*

by *auto*

qed

qed

lemma *distinct-compose*:

assumes *distinct* (*map fst xs*)

shows *distinct* (*map fst (compose xs ys)*)

using *assms*

proof (*induct xs ys rule: compose.induct*)

case 1

then show ?*case* **by** *simp*

next

case (2 *x xs ys*)

show ?*case*

proof (*cases map-of ys (snd x)*)

case *None*

with 2 **show** ?*thesis* **by** *simp*

next

case (*Some v*)

```

    with 2 dom-compose [of xs ys] show ?thesis
    by auto
qed
qed

```

```

lemma compose-delete-twist: compose (delete k xs) ys = delete k (compose xs ys)
proof (induct xs ys rule: compose.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 have hyp: compose (delete k (delete (fst x) xs)) ys =
      delete k (compose (delete (fst x) xs) ys)
    by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      with None hyp show ?thesis
      by (simp add: delete-idem)
    next
      case False
      from None False hyp show ?thesis
      by (simp add: delete-twist)
    qed
  next
    case (Some v)
    with 2 have hyp: compose (delete k xs) ys = delete k (compose xs ys)
    by simp
    with Some show ?thesis
    by simp
  qed
qed

```

```

lemma compose-clearjunk: compose xs (clearjunk ys) = compose xs ys
by (induct xs ys rule: compose.induct)
(auto simp add: map-of-clearjunk split: option.splits)

```

```

lemma clearjunk-compose: clearjunk (compose xs ys) = compose (clearjunk xs) ys
by (induct xs rule: clearjunk.induct)
(auto split: option.splits simp add: clearjunk-delete delete-idem compose-delete-twist)

```

```

lemma compose-empty [simp]: compose xs [] = []
by (induct xs) (auto simp add: compose-delete-twist)

```

```

lemma compose-Some-iff:
  (map-of (compose xs ys) k = Some v)  $\longleftrightarrow$ 

```

($\exists k'. \text{map-of } xs \ k = \text{Some } k' \wedge \text{map-of } ys \ k' = \text{Some } v$)
by (*simp add: compose-conv map-comp-Some-iff*)

lemma *map-comp-None-iff*:

map-of (compose xs ys) k = None \longleftrightarrow
(map-of xs k = None \vee ($\exists k'. \text{map-of } xs \ k = \text{Some } k' \wedge \text{map-of } ys \ k' = \text{None}$))
by (*simp add: compose-conv map-comp-None-iff*)

1.9 map-entry

qualified fun *map-entry* :: 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list

where

map-entry k f [] = []
| *map-entry k f (p # ps)* =
(*if fst p = k then (k, f (snd p)) # ps else p # map-entry k f ps*)

lemma *map-of-map-entry*:

map-of (map-entry k f xs) =
(*map-of xs*)(*k* := *case map-of xs k of None* \Rightarrow *None* | *Some v'* \Rightarrow *Some (f v')*)
by (*induct xs*) *auto*

lemma *dom-map-entry*: *fst ' set (map-entry k f xs)* = *fst ' set xs*

by (*induct xs*) *auto*

lemma *distinct-map-entry*:

assumes *distinct (map fst xs)*
shows *distinct (map fst (map-entry k f xs))*
using *assms by (induct xs) (auto simp add: dom-map-entry)*

1.10 map-default

fun *map-default* :: 'key \Rightarrow 'val \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list

where

map-default k v f [] = [(*k*, *v*)]
| *map-default k v f (p # ps)* =
(*if fst p = k then (k, f (snd p)) # ps else p # map-default k v f ps*)

lemma *map-of-map-default*:

map-of (map-default k v f xs) =
(*map-of xs*)(*k* := *case map-of xs k of None* \Rightarrow *Some v* | *Some v'* \Rightarrow *Some (f v')*)
by (*induct xs*) *auto*

lemma *dom-map-default*: *fst ' set (map-default k v f xs)* = *insert k (fst ' set xs)*

by (*induct xs*) *auto*

lemma *distinct-map-default*:

assumes *distinct (map fst xs)*
shows *distinct (map fst (map-default k v f xs))*

```

using assms by (induct xs) (auto simp add: dom-map-default)

end

end

```

2 Axiomatic Declaration of Bounded Natural Functors

```

theory BNF-Axiomatization
imports Main
keywords
  bnf-axiomatization :: thy-decl
begin

ML-file <../Tools/BNF/bnf-axiomatization.ML>

end

```

3 Generalized Corecursor Sugar (corec and friends)

```

theory BNF-Corec
imports Main
keywords
  corec :: thy-defn and
  corecursive :: thy-goal-defn and
  friend-of-corec :: thy-goal-defn and
  coinduction-upto :: thy-decl
begin

lemma obj-distinct-prems:  $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$ 
  by auto

lemma inject-refine:  $g (f x) = x \Longrightarrow g (f y) = y \Longrightarrow f x = f y \longleftrightarrow x = y$ 
  by (metis (no-types))

lemma convol-apply:  $\text{BNF-Def.convol } f \ g \ x = (f \ x, \ g \ x)$ 
  unfolding convol-def ..

lemma Grp-UNIV-id:  $\text{BNF-Def.Grp UNIV id} = (=)$ 
  unfolding BNF-Def.Grp-def by auto

lemma sum-comp-cases:
  assumes  $f \circ \text{Inl} = g \circ \text{Inl}$  and  $f \circ \text{Inr} = g \circ \text{Inr}$ 
  shows  $f = g$ 
proof (rule ext)
  fix a show  $f \ a = g \ a$ 
  using assms unfolding comp-def fun-eq-iff by (cases a) auto

```

qed

lemma *case-sum-Inl-Inr-L*: $\text{case-sum } (f \circ \text{Inl}) (f \circ \text{Inr}) = f$
by (*metis case-sum-expand-Inr'*)

lemma *eq-o-InrI*: $\llbracket g \circ \text{Inl} = h; \text{case-sum } h \, f = g \rrbracket \implies f = g \circ \text{Inr}$
by (*auto simp: fun-eq-iff split: sum.splits*)

lemma *id-bnf-o*: $\text{BNF-Composition.id-bnf} \circ f = f$
unfolding *BNF-Composition.id-bnf-def* **by** (*rule o-def*)

lemma *o-id-bnf*: $f \circ \text{BNF-Composition.id-bnf} = f$
unfolding *BNF-Composition.id-bnf-def* **by** (*rule o-def*)

lemma *if-True-False*:
 $(\text{if } P \text{ then True else } Q) \longleftrightarrow P \vee Q$
 $(\text{if } P \text{ then False else } Q) \longleftrightarrow \neg P \wedge Q$
 $(\text{if } P \text{ then } Q \text{ else True}) \longleftrightarrow \neg P \vee Q$
 $(\text{if } P \text{ then } Q \text{ else False}) \longleftrightarrow P \wedge Q$
by *auto*

lemma *if-distrib-fun*: $(\text{if } c \text{ then } f \text{ else } g) \, x = (\text{if } c \text{ then } f \, x \text{ else } g \, x)$
by *simp*

3.1 Coinduction

lemma *eq-comp-compI*: $a \circ b = f \circ x \implies x \circ c = \text{id} \implies f = a \circ (b \circ c)$
unfolding *fun-eq-iff* **by** *simp*

lemma *self-bounded-weaken-left*: $(a :: 'a :: \text{semilattice-inf}) \leq \inf a \, b \implies a \leq b$
by (*erule le-infE*)

lemma *self-bounded-weaken-right*: $(a :: 'a :: \text{semilattice-inf}) \leq \inf b \, a \implies a \leq b$
by (*erule le-infE*)

lemma *symp-iff*: $\text{symp } R \longleftrightarrow R = R^{-1-1}$
by (*metis antisym conversep.cases conversep-le-swap predicate2I symp-def*)

lemma *equivp-inf*: $\llbracket \text{equivp } R; \text{equivp } S \rrbracket \implies \text{equivp } (\inf R \, S)$
unfolding *equivp-def inf-fun-def inf-bool-def* **by** *metis*

lemma *vimage2p-rel-prod*:
 $(\lambda x \, y. \text{rel-prod } R \, S \, (\text{BNF-Def.convolve } f1 \, g1 \, x) \, (\text{BNF-Def.convolve } f2 \, g2 \, y)) =$
 $(\inf (\text{BNF-Def.vimage2p } f1 \, f2 \, R) \, (\text{BNF-Def.vimage2p } g1 \, g2 \, S))$
unfolding *vimage2p-def rel-prod.simps convolve-def* **by** *auto*

lemma *predicate2I-obj*: $(\forall x \, y. P \, x \, y \longrightarrow Q \, x \, y) \implies P \leq Q$
by *auto*

lemma *predicate2D-obj*: $P \leq Q \implies P\ x\ y \longrightarrow Q\ x\ y$
by *auto*

locale *cong* =
fixes *rel* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'b \Rightarrow \text{bool})$
and *eval* :: $'b \Rightarrow 'a$
and *retr* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \Rightarrow \text{bool})$
assumes *rel-mono*: $\bigwedge R\ S. R \leq S \implies \text{rel}\ R \leq \text{rel}\ S$
and *equivp-retr*: $\bigwedge R. \text{equivp}\ R \implies \text{equivp}\ (\text{retr}\ R)$
and *retr-eval*: $\bigwedge R\ x\ y. \llbracket (\text{rel-fun}\ (\text{rel}\ R)\ R)\ \text{eval}\ \text{eval}; \text{rel}\ (\text{inf}\ R\ (\text{retr}\ R))\ x\ y \rrbracket$
 \implies
 $\text{retr}\ R\ (\text{eval}\ x)\ (\text{eval}\ y)$
begin

definition *cong* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{cong}\ R \equiv \text{equivp}\ R \wedge (\text{rel-fun}\ (\text{rel}\ R)\ R)\ \text{eval}\ \text{eval}$

lemma *cong-retr*: $\text{cong}\ R \implies \text{cong}\ (\text{inf}\ R\ (\text{retr}\ R))$
unfolding *cong-def*
by (*auto simp: rel-fun-def dest: predicate2D[OF rel-mono, rotated]*
intro: equivp-inf equivp-retr retr-eval)

lemma *cong-equivp*: $\text{cong}\ R \implies \text{equivp}\ R$
unfolding *cong-def* **by** *simp*

definition *gen-cong* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**
 $\text{gen-cong}\ R\ j1\ j2 \equiv \forall R'. R \leq R' \wedge \text{cong}\ R' \longrightarrow R'\ j1\ j2$

lemma *gen-cong-reflp*[*intro, simp*]: $x = y \implies \text{gen-cong}\ R\ x\ y$
unfolding *gen-cong-def* **by** (*auto dest: cong-equivp equivp-reflp*)

lemma *gen-cong-symp*[*intro*]: $\text{gen-cong}\ R\ x\ y \implies \text{gen-cong}\ R\ y\ x$
unfolding *gen-cong-def* **by** (*auto dest: cong-equivp equivp-symp*)

lemma *gen-cong-transp*[*intro*]: $\text{gen-cong}\ R\ x\ y \implies \text{gen-cong}\ R\ y\ z \implies \text{gen-cong}\ R\ x\ z$
unfolding *gen-cong-def* **by** (*auto dest: cong-equivp equivp-transp*)

lemma *equivp-gen-cong*: $\text{equivp}\ (\text{gen-cong}\ R)$
by (*intro equivpI reflpI sympI transpI*) *auto*

lemma *leq-gen-cong*: $R \leq \text{gen-cong}\ R$
unfolding *gen-cong-def*[*abs-def*] **by** *auto*

lemmas *imp-gen-cong*[*intro*] = *predicate2D*[*OF leq-gen-cong*]

lemma *gen-cong-minimal*: $\llbracket R \leq R'; \text{cong}\ R' \rrbracket \implies \text{gen-cong}\ R \leq R'$
unfolding *gen-cong-def*[*abs-def*] **by** (*rule predicate2I*) *metis*

lemma *congdd-base-gen-congdd-base-aux*:

$rel\ (gen-cong\ R)\ x\ y \implies R \leq R' \implies cong\ R' \implies R' (eval\ x)\ (eval\ y)$
by (*force simp: rel-fun-def gen-cong-def cong-def dest: spec[of - R'] predicate2D[OF rel-mono, rotated -1, of - - - R']*)

lemma *cong-gen-cong*: $cong\ (gen-cong\ R)$

proof –

have $rel\ (gen-cong\ R)\ x\ y \implies R \leq R' \implies cong\ R' \implies R' (eval\ x)\ (eval\ y)$ **for**
 $R' x y$

by (*force simp: rel-fun-def gen-cong-def cong-def dest: spec[of - R'] predicate2D[OF rel-mono, rotated -1, of - - - R']*)

then show $cong\ (gen-cong\ R)$ **by** (*auto simp: equivp-gen-cong rel-fun-def gen-cong-def cong-def*)

qed

lemma *gen-cong-eval-rel-fun*:

$(rel-fun\ (rel\ (gen-cong\ R))\ (gen-cong\ R))\ eval\ eval$
using *cong-gen-cong*[*of R*] **unfolding** *cong-def* **by** *simp*

lemma *gen-cong-eval*:

$rel\ (gen-cong\ R)\ x\ y \implies gen-cong\ R\ (eval\ x)\ (eval\ y)$
by (*erule rel-funD[OF gen-cong-eval-rel-fun]*)

lemma *gen-cong-idem*: $gen-cong\ (gen-cong\ R) = gen-cong\ R$

by (*simp add: antisym cong-gen-cong gen-cong-minimal leq-gen-cong*)

lemma *gen-cong-rho*:

$\varrho = eval \circ f \implies rel\ (gen-cong\ R)\ (f\ x)\ (f\ y) \implies gen-cong\ R\ (\varrho\ x)\ (\varrho\ y)$
by (*simp add: gen-cong-eval*)

lemma *coinduction*:

assumes *coind*: $\forall R. R \leq retr\ R \longrightarrow R \leq (=)$

assumes *cih*: $R \leq retr\ (gen-cong\ R)$

shows $R \leq (=)$

apply (*rule order-trans[OF leq-gen-cong mp[OF spec[OF coind]]]*)

apply (*rule self-bounded-weaken-left[OF gen-cong-minimal]*)

apply (*rule inf-greatest[OF leq-gen-cong cih]*)

apply (*rule cong-retr[OF cong-gen-cong]*)

done

end

lemma *rel-sum-case-sum*:

$rel-fun\ (rel-sum\ R\ S)\ T\ (case-sum\ f1\ g1)\ (case-sum\ f2\ g2) = (rel-fun\ R\ T\ f1\ f2) \wedge rel-fun\ S\ T\ g1\ g2)$

by (*auto simp: rel-fun-def rel-sum.simps split: sum.splits*)

context

fixes *rel eval rel' eval' retr emb*

assumes *base*: $cong\ rel\ eval\ retr$


```

and step: cong rel' eval' retr
and emb: eval'  $\circ$  emb = eval
and emb-transfer: rel-fun (rel R) (rel' R) emb emb
begin

interpretation base: cong rel eval retr by (rule base)
interpretation step: cong rel' eval' retr by (rule step)

lemma gen-cong-emb: base.gen-cong R  $\leq$  step.gen-cong R
proof (rule base.gen-cong-minimal[OF step.leq-gen-cong])
  note step.gen-cong-eval-rel-fun[transfer-rule] emb-transfer[transfer-rule]
  have (rel-fun (rel (step.gen-cong R)) (step.gen-cong R)) eval eval
  unfolding emb[symmetric] by transfer-prover
  then show base.cong (step.gen-cong R)
  by (auto simp: base.cong-def step.equivp-gen-cong)
qed

end

named-theorems friend-of-corec-simps

ML-file  $\langle \dots / \text{Tools} / \text{BNF} / \text{bnf-gfp-grec-tactics.ML} \rangle$ 
ML-file  $\langle \dots / \text{Tools} / \text{BNF} / \text{bnf-gfp-grec.ML} \rangle$ 
ML-file  $\langle \dots / \text{Tools} / \text{BNF} / \text{bnf-gfp-grec-sugar-util.ML} \rangle$ 
ML-file  $\langle \dots / \text{Tools} / \text{BNF} / \text{bnf-gfp-grec-sugar-tactics.ML} \rangle$ 
ML-file  $\langle \dots / \text{Tools} / \text{BNF} / \text{bnf-gfp-grec-sugar.ML} \rangle$ 
ML-file  $\langle \dots / \text{Tools} / \text{BNF} / \text{bnf-gfp-grec-unique-sugar.ML} \rangle$ 

method-setup transfer-prover-eq =  $\langle$ 
  Scan.succeed (SIMPLE-METHOD'  $\circ$  BNF-GFP-Grec-Tactics.transfer-prover-eq-tac)
 $\rangle$  apply transfer-prover after folding relator-eq

method-setup corec-unique =  $\langle$ 
  Scan.succeed (SIMPLE-METHOD'  $\circ$  BNF-GFP-Grec-Unique-Sugar.corec-unique-tac)
 $\rangle$  prove uniqueness of corecursive equation

end

```

4 A general “while” combinator

```

theory While-Combinator
imports Main
begin

```

Defining partial functions in HOL is tricky. This theory provides a while-combinator that facilitates the definition of (potentially) partial tail-recursive functions.

The theory provides the function *while-option* *b f s* that iterates *f* on *s* while *b* is true. If iteration terminates with *t*, *Some t* is returned, *None*

otherwise. Thus termination can be shown by proving that *Some* is always returned (for some subset of inputs).

Convenient variations include *while-Some* (for more efficient code) and *while-saturate* (for saturating a set).

4.1 while-option

definition *while-option* :: $(a \Rightarrow \text{bool}) \Rightarrow (a \Rightarrow a) \Rightarrow a \Rightarrow a$ **option where**
while-option $b\ c\ s = (\text{if } b\ s \text{ then } (\exists k. \neg b\ ((c \rightsquigarrow k)\ s))$
 then $\text{Some } ((c \rightsquigarrow (\text{LEAST } k. \neg b\ ((c \rightsquigarrow k)\ s)))\ s)$
 else None)

theorem *while-option-unfold*[code]:

while-option $b\ c\ s = (\text{if } b\ s \text{ then } \text{while-option } b\ c\ (c\ s) \text{ else } \text{Some } s)$

proof *cases*

assume $b\ s$

show *?thesis*

proof $(\text{cases } \exists k. \neg b\ ((c \rightsquigarrow k)\ s))$

case *True*

then obtain k **where** $1: \neg b\ ((c \rightsquigarrow k)\ s) \dots$

with $\langle b\ s \rangle$ **obtain** l **where** $k = \text{Suc } l$ **by** $(\text{cases } k)\ \text{auto}$

with 1 **have** $\neg b\ ((c \rightsquigarrow l)\ (c\ s))$ **by** $(\text{auto simp: funpow-swap1})$

then have $2: \exists l. \neg b\ ((c \rightsquigarrow l)\ (c\ s)) \dots$

from 1

have $(\text{LEAST } k. \neg b\ ((c \rightsquigarrow k)\ s)) = \text{Suc } (\text{LEAST } l. \neg b\ ((c \rightsquigarrow \text{Suc } l)\ s))$

by $(\text{rule Least-Suc})\ (\text{simp add: } \langle b\ s \rangle)$

also have $\dots = \text{Suc } (\text{LEAST } l. \neg b\ ((c \rightsquigarrow l)\ (c\ s)))$

by $(\text{simp add: funpow-swap1})$

finally

show *?thesis*

using *True 2* $\langle b\ s \rangle$ **by** $(\text{simp add: funpow-swap1 while-option-def})$

next

case *False*

then have $\neg (\exists l. \neg b\ ((c \rightsquigarrow \text{Suc } l)\ s))$ **by** *blast*

then have $\neg (\exists l. \neg b\ ((c \rightsquigarrow l)\ (c\ s)))$

by $(\text{simp add: funpow-swap1})$

with *False* $\langle b\ s \rangle$ **show** *?thesis* **by** $(\text{simp add: while-option-def})$

qed

next

assume $[\text{simp}]: \neg b\ s$

have *least*: $(\text{LEAST } k. \neg b\ ((c \rightsquigarrow k)\ s)) = 0$

by $(\text{rule Least-equality})\ \text{auto}$

moreover

have $\exists k. \neg b\ ((c \rightsquigarrow k)\ s)$ **by** $(\text{rule exI}[of - 0::\text{nat}])\ \text{auto}$

ultimately show *?thesis* **unfolding** *while-option-def* **by** *auto*

qed

lemma *while-option-stop2*:

while-option $b\ c\ s = \text{Some } t \implies \exists k. t = (c \rightsquigarrow k)\ s \wedge \neg b\ t$

apply(*simp add: while-option-def split: if-splits*)
by (*metis (lifting) LeastI-ex*)

lemma *while-option-stop*: *while-option b c s = Some t $\implies \neg b t$*
by(*metis while-option-stop2*)

theorem *while-option-rule*:
assumes *step*: $\bigwedge s. P s \implies b s \implies P (c s)$
and *result*: *while-option b c s = Some t*
and *init*: *P s*
shows *P t*
proof –
define *k* **where** *k* = (*LEAST k. $\neg b ((c \rightsquigarrow k) s)$*)
from *assms* **have** *t*: *t* = (*c \rightsquigarrow k*) *s*
by (*simp add: while-option-def k-def split: if-splits*)
have *1*: $\forall i < k. b ((c \rightsquigarrow i) s)$
by (*auto simp: k-def dest: not-less-Least*)
have $i \leq k \implies P ((c \rightsquigarrow i) s)$ **for** *i*
by (*induct i (auto simp: init step 1)*)
thus *P t* **by** (*auto simp: t*)
qed

lemma *funpow-commute*:
 $\llbracket \forall k' < k. f (c ((c \rightsquigarrow k') s)) = c' (f ((c \rightsquigarrow k') s)) \rrbracket \implies f ((c \rightsquigarrow k) s) = (c' \rightsquigarrow k) (f s)$
by (*induct k arbitrary: s*) *auto*

lemma *while-option-commute-invariant*:
assumes *Invariant*: $\bigwedge s. P s \implies b s \implies P (c s)$
assumes *TestCommute*: $\bigwedge s. P s \implies b s = b' (f s)$
assumes *BodyCommute*: $\bigwedge s. P s \implies b s \implies f (c s) = c' (f s)$
assumes *Initial*: *P s*
shows *map-option f (while-option b c s) = while-option b' c' (f s)*
unfolding *while-option-def*
proof (*rule trans[OF if-distrib if-cong], safe, unfold option.inject*)
fix *k*
assume $\neg b ((c \rightsquigarrow k) s)$
with *Initial* **show** $\exists k. \neg b' ((c' \rightsquigarrow k) (f s))$
proof (*induction k arbitrary: s*)
case *0* **thus** ?*case* **by** (*auto simp: TestCommute intro: exI[of - 0]*)
next
case (*Suc k*) **thus** ?*case*
proof (*cases b s*)
assume *b s*
with *Suc.IH*[*of c s*] *Suc.prem*s **show** ?*thesis*
by (*metis BodyCommute Invariant comp-apply funpow.simps(2) funpow-swap1*)
next
assume $\neg b s$
with *Suc* **show** ?*thesis* **by** (*auto simp: TestCommute intro: exI [of - 0]*)
qed

```

qed
next
  fix k
  assume  $\neg b' ((c' \rightsquigarrow k) (f s))$ 
  with Initial show  $\exists k. \neg b ((c \rightsquigarrow k) s)$ 
  proof (induction k arbitrary: s)
    case 0 thus ?case by (auto simp: TestCommute intro: exI[of - 0])
  next
    case (Suc k) thus ?case
    proof (cases b s)
      assume b s
      with Suc.IH[of c s] Suc.prems show ?thesis
      by (metis BodyCommute Invariant comp-apply funpow.simps(2) funpow-swap1)
    next
      assume  $\neg b s$ 
      with Suc show ?thesis by (auto simp: TestCommute intro: exI [of - 0])
    qed
  qed
next
  fix k
  assume  $k: \neg b' ((c' \rightsquigarrow k) (f s))$ 
  have *: (LEAST k.  $\neg b' ((c' \rightsquigarrow k) (f s))$ ) = (LEAST k.  $\neg b ((c \rightsquigarrow k) s)$ )
    (is ?k' = ?k)
  proof (cases ?k')
    case 0
    have  $\neg b' ((c' \rightsquigarrow 0) (f s))$ 
    unfolding 0[symmetric] by (rule LeastI[of - k]) (rule k)
    hence  $\neg b s$  by (auto simp: TestCommute Initial)
    hence ?k = 0 by (intro Least-equality) auto
    with 0 show ?thesis by auto
  next
    case (Suc k')
    have  $\neg b' ((c' \rightsquigarrow \text{Suc } k') (f s))$ 
    unfolding Suc[symmetric] by (rule LeastI) (rule k)
    moreover
    have  $b': b' ((c' \rightsquigarrow k) (f s))$  if asm:  $k \leq k'$  for k
    proof -
      from asm have  $k < k'$  unfolding Suc by simp
      thus ?thesis by (rule iffD1[OF not-not, OF not-less-Least])
    qed
  qed
  have b:  $b ((c \rightsquigarrow k) s)$ 
  and body:  $f ((c \rightsquigarrow k) s) = (c' \rightsquigarrow k) (f s)$ 
  and inv:  $P ((c \rightsquigarrow k) s)$ 
  if asm:  $k \leq k'$  for k
  proof -
    from asm have  $f ((c \rightsquigarrow k) s) = (c' \rightsquigarrow k) (f s)$ 
    and  $b ((c \rightsquigarrow k) s) = b' ((c' \rightsquigarrow k) (f s))$ 
    and  $P ((c \rightsquigarrow k) s)$ 
    by (induct k) (auto simp: b' assms)
  qed

```

```

  with  $\langle k \leq k' \rangle$ 
  show  $b \ ((c \rightsquigarrow k) \ s)$ 
    and  $f \ ((c \rightsquigarrow k) \ s) = (c' \rightsquigarrow k) \ (f \ s)$ 
    and  $P \ ((c \rightsquigarrow k) \ s)$ 
    by (auto simp:  $b'$ )
qed
hence  $k': f \ ((c \rightsquigarrow k') \ s) = (c' \rightsquigarrow k') \ (f \ s)$  by auto
ultimately show ?thesis unfolding Suc using b
proof (intro Least-equality[symmetric], goal-cases)
  case 1
  hence Test:  $\neg b' \ (f \ ((c \rightsquigarrow \text{Suc } k') \ s))$ 
    by (auto simp: BodyCommute inv b)
  have  $P \ ((c \rightsquigarrow \text{Suc } k') \ s)$  by (auto simp: Invariant inv b)
  with Test show ?case by (auto simp: TestCommute)
next
  case 2
  thus ?case by (metis not-less-eq-eq)
qed
qed
have  $f \ ((c \rightsquigarrow ?k) \ s) = (c' \rightsquigarrow ?k') \ (f \ s)$  unfolding *
proof (rule funpow-commute, clarify)
  fix k assume  $k < ?k$ 
  hence TestTrue:  $b \ ((c \rightsquigarrow k) \ s)$  by (auto dest: not-less-Least)
  from  $\langle k < ?k \rangle$  have  $P \ ((c \rightsquigarrow k) \ s)$ 
  proof (induct k)
    case 0 thus ?case by (auto simp: assms)
  next
    case (Suc h)
    hence  $P \ ((c \rightsquigarrow h) \ s)$  by auto
    with Suc show ?case
      by (auto, metis (lifting, no-types) Invariant Suc-lessD not-less-Least)
  qed
  with TestTrue show  $f \ (c \ ((c \rightsquigarrow k) \ s)) = c' \ (f \ ((c \rightsquigarrow k) \ s))$ 
    by (metis BodyCommute)
qed
thus  $\exists z. (c \rightsquigarrow ?k) \ s = z \wedge f \ z = (c' \rightsquigarrow ?k') \ (f \ s)$  by blast
qed

```

lemma *while-option-commute*:

```

  assumes  $\bigwedge s. b \ s = b' \ (f \ s) \wedge s. \llbracket b \ s \rrbracket \implies f \ (c \ s) = c' \ (f \ s)$ 
  shows  $\text{map-option } f \ (\text{while-option } b \ c \ s) = \text{while-option } b' \ c' \ (f \ s)$ 
  by (rule while-option-commute-invariant[where  $P = \lambda -. \text{True}$ ])
    (auto simp add: assms)

```

4.2 while

definition *while* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
 where *while* $b \ c \ s = \text{the } (\text{while-option } b \ c \ s)$

lemma *while-unfold* [code]:
 while $b \ c \ s = (\text{if } b \ s \text{ then } \text{while } b \ c \ (c \ s) \text{ else } s)$
unfolding *while-def* **by** (*subst while-option-unfold*) *simp*

lemma *def-while-unfold*:
 assumes *fdef*: $f == \text{while test do}$
 shows $f \ x = (\text{if test } x \text{ then } f(\text{do } x) \text{ else } x)$
unfolding *fdef* **by** (*fact while-unfold*)

The proof rule for *while*, where P is the invariant.

theorem *while-rule-lemma*:
 assumes *invariant*: $\bigwedge s. P \ s \implies b \ s \implies P \ (c \ s)$
 and *terminate*: $\bigwedge s. P \ s \implies \neg b \ s \implies Q \ s$
 and *wf*: $wf \ \{(t, s). P \ s \wedge b \ s \wedge t = c \ s\}$
shows $P \ s \implies Q \ (\text{while } b \ c \ s)$
using *wf*
apply (*induct s*)
apply *simp*
apply (*subst while-unfold*)
apply (*simp add: invariant terminate*)
done

theorem *while-rule*:
 $\llbracket P \ s;$
 $\bigwedge s. \llbracket P \ s; b \ s \rrbracket \implies P \ (c \ s);$
 $\bigwedge s. \llbracket P \ s; \neg b \ s \rrbracket \implies Q \ s;$
 $wf \ r;$
 $\bigwedge s. \llbracket P \ s; b \ s \rrbracket \implies (c \ s, s) \in r \rrbracket \implies$
 $Q \ (\text{while } b \ c \ s)$
apply (*rule while-rule-lemma*)
prefer 4 **apply** *assumption*
apply *blast*
apply *blast*
apply (*erule wf-subset*)
apply *blast*
done

Combine invariant preservation and variant decrease in one goal:

theorem *while-rule2*:
 $\llbracket P \ s;$
 $\bigwedge s. \llbracket P \ s; b \ s \rrbracket \implies P \ (c \ s) \wedge (c \ s, s) \in r;$
 $\bigwedge s. \llbracket P \ s; \neg b \ s \rrbracket \implies Q \ s;$
 $wf \ r \rrbracket \implies$
 $Q \ (\text{while } b \ c \ s)$
using *while-rule*[of P] **by** *metis*

4.3 Termination, *lfp* and *gfp*

theorem *wf-while-option-Some*:
 assumes *wf* $\{(t, s). (P \ s \wedge b \ s) \wedge t = c \ s\}$

and $\bigwedge s. P\ s \implies b\ s \implies P(c\ s)$ **and** $P\ s$
shows $\exists t. \text{while-option } b\ c\ s = \text{Some } t$
using *assms(1,3)*
proof (*induction s*)
 case less thus *?case using assms(2)*
 by (*subst while-option-unfold*) *simp*
qed

lemma *wf-rel-while-option-Some*:
assumes *wf*: *wf R*
assumes *smaller*: $\bigwedge s. P\ s \wedge b\ s \implies (c\ s, s) \in R$
assumes *inv*: $\bigwedge s. P\ s \wedge b\ s \implies P(c\ s)$
assumes *init*: $P\ s$
shows $\exists t. \text{while-option } b\ c\ s = \text{Some } t$
proof –
 from *smaller* **have** $\{(t, s). P\ s \wedge b\ s \wedge t = c\ s\} \subseteq R$ **by** *auto*
 with *wf* **have** *wf* $\{(t, s). P\ s \wedge b\ s \wedge t = c\ s\}$ **by** (*auto simp: wf-subset*)
 with *inv init* **show** *?thesis* **by** (*auto simp: wf-while-option-Some*)
qed

theorem *measure-while-option-Some*: **fixes** $f :: 's \Rightarrow \text{nat}$
shows $(\bigwedge s. P\ s \implies b\ s \implies P(c\ s) \wedge f(c\ s) < f\ s)$
 $\implies P\ s \implies \exists t. \text{while-option } b\ c\ s = \text{Some } t$
by(*blast intro: wf-while-option-Some[OF wf-if-measure, of P b f]*)

Kleene iteration starting from the empty set and assuming some finite bounding set:

lemma *while-option-finite-subset-Some*: **fixes** $C :: 'a\ \text{set}$
assumes *mono f* **and** $\bigwedge X. X \subseteq C \implies f\ X \subseteq C$ **and** *finite C*
shows $\exists P. \text{while-option } (\lambda A. f\ A \neq A)\ f\ \{\} = \text{Some } P$
proof(*rule measure-while-option-Some[where*
 $f = \%A::'a\ \text{set}. \text{card } C - \text{card } A$ **and** $P = \%A. A \subseteq C \wedge A \subseteq f\ A$ **and** $s = \{\}$)
 fix A **assume** $A: A \subseteq C \wedge A \subseteq f\ A$ $f\ A \neq A$
 show $(f\ A \subseteq C \wedge f\ A \subseteq f\ (f\ A)) \wedge \text{card } C - \text{card } (f\ A) < \text{card } C - \text{card } A$
 (*is ?L \wedge ?R*)
 proof
 show *?L* **by** (*metis A(1) assms(2) monoD[OF ‹mono f›]*)
 show *?R* **by** (*metis A assms(2,3) card-seteq diff-less-mono2 equalityI linorder-le-less-linear rev-finite-subset*)
 qed
qed *simp*

lemma *lfp-the-while-option*:
assumes *mono f* **and** $\bigwedge X. X \subseteq C \implies f\ X \subseteq C$ **and** *finite C*
shows $\text{lfp } f = \text{the}(\text{while-option } (\lambda A. f\ A \neq A)\ f\ \{\})$
proof –
 obtain P **where** $\text{while-option } (\lambda A. f\ A \neq A)\ f\ \{\} = \text{Some } P$
 using *while-option-finite-subset-Some[OF assms]* **by** *blast*

with *while-option-stop2*[*OF this*] *lfp-Kleene-iter*[*OF assms*(1)]
show *?thesis* **by** *auto*
qed

lemma *lfp-while*:
assumes *mono f* **and** $\bigwedge X. X \subseteq C \implies f X \subseteq C$ **and** *finite C*
shows $\text{lfp } f = \text{while } (\lambda A. f A \neq A) f \ \{\}$
unfolding *while-def* **using** *assms* **by** (rule *lfp-the-while-option*) *blast*

lemma *wf-finite-less*:
assumes *finite (C :: 'a::order set)*
shows $\text{wf } \{(x, y). \{x, y\} \subseteq C \wedge x < y\}$
by (rule *wf-measure*[**where** $f = \lambda b. \text{card } \{a. a \in C \wedge a < b\}$, *THEN wf-subset*])
(fastforce simp: less-eq assms intro: psubset-card-mono)

lemma *wf-finite-greater*:
assumes *finite (C :: 'a::order set)*
shows $\text{wf } \{(x, y). \{x, y\} \subseteq C \wedge y < x\}$
by (rule *wf-measure*[**where** $f = \lambda b. \text{card } \{a. a \in C \wedge b < a\}$, *THEN wf-subset*])
(fastforce simp: less-eq assms intro: psubset-card-mono)

lemma *while-option-finite-increasing-Some*:
fixes $f :: 'a::order \Rightarrow 'a$
assumes *mono f* **and** *finite (UNIV :: 'a set)* **and** $s \leq f s$
shows $\exists P. \text{while-option } (\lambda A. f A \neq A) f s = \text{Some } P$
by (rule *wf-rel-while-option-Some*[**where** $R = \{(x, y). y < x\}$ **and** $P = \lambda A. A \leq f A$ **and** $s = s$])
(auto simp: assms monoD intro: wf-finite-greater[where C=UNIV::'a set, simplified])

lemma *lfp-the-while-option-lattice*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono f* **and** *finite (UNIV :: 'a set)*
shows $\text{lfp } f = \text{the } (\text{while-option } (\lambda A. f A \neq A) f \text{ bot})$
proof –
obtain P **where** $\text{while-option } (\lambda A. f A \neq A) f \text{ bot} = \text{Some } P$
using *while-option-finite-increasing-Some*[*OF assms*, **where** $s = \text{bot}$] **by** *simp*
blast
with *while-option-stop2*[*OF this*] *lfp-Kleene-iter*[*OF assms*(1)]
show *?thesis* **by** *auto*
qed

lemma *lfp-while-lattice*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono f* **and** *finite (UNIV :: 'a set)*
shows $\text{lfp } f = \text{while } (\lambda A. f A \neq A) f \text{ bot}$
unfolding *while-def* **using** *assms* **by** (rule *lfp-the-while-option-lattice*)

lemma *while-option-finite-decreasing-Some*:

fixes $f :: 'a::order \Rightarrow 'a$
assumes $mono\ f$ **and** $finite\ (UNIV :: 'a\ set)$ **and** $f\ s \leq s$
shows $\exists P. while_option\ (\lambda A. f\ A \neq A)\ f\ s = Some\ P$
by (rule $wf_rel_while_option_Some$ [**where** $R=\{(x, y). x < y\}$ **and** $P=\lambda A. f\ A \leq A$
and $s=s$])
(auto simp add: assms monoD intro: wf-finite-less[**where** $C=UNIV::'a\ set$, simplified])

lemma *gfp-the-while-option-lattice*:

fixes $f :: 'a::complete_lattice \Rightarrow 'a$
assumes $mono\ f$ **and** $finite\ (UNIV :: 'a\ set)$
shows $gfp\ f = the(while_option\ (\lambda A. f\ A \neq A)\ f\ top)$
proof –
obtain P **where** $while_option\ (\lambda A. f\ A \neq A)\ f\ top = Some\ P$
using $while_option_finite_decreasing_Some$ [OF assms, **where** $s=top$] **by** simp
blast
with $while_option_stop2$ [OF this] gfp_Kleene_iter [OF assms(1)]
show ?thesis **by** auto
qed

lemma *gfp-while-lattice*:

fixes $f :: 'a::complete_lattice \Rightarrow 'a$
assumes $mono\ f$ **and** $finite\ (UNIV :: 'a\ set)$
shows $gfp\ f = while\ (\lambda A. f\ A \neq A)\ f\ top$
unfolding $while_def$ **using** assms **by** (rule *gfp-the-while-option-lattice*)

4.4 while-Some and while-saturate

A variation intended for efficient code. The problem with $while_option\ b\ c$: the computations of b and c may share subcomputations but they need to be performed twice.

definition $while_Some :: ('s \Rightarrow 's\ option) \Rightarrow 's \Rightarrow 's\ option$ **where**
 $while_Some\ f = while_option\ (\lambda s. f\ s \neq None)\ (the\ o\ f)$

lemma *while-Some-rec*[code]:

$while_Some\ f\ x = (case\ f\ x\ of\ None \Rightarrow Some\ x \mid Some\ y \Rightarrow while_Some\ f\ y)$
unfolding $while_Some_def$ $while_option_unfold$ [of - - x] **by** auto

A frequent special case: saturation of a set.

definition $while_saturate :: ('a\ set \Rightarrow 'a\ set) \Rightarrow 'a\ set \Rightarrow 'a\ set\ option$ **where**
 $while_saturate\ f = while_option\ (\lambda M. \neg f\ M \subseteq M)\ (\lambda M. M \cup f\ M)$

lemma *while-option-cong*: $(\bigwedge s. b\ s \Longrightarrow c\ s = c'\ s) \Longrightarrow while_option\ b\ c\ s = while_option\ b\ c'\ s$

using $while_option_commute$ [of $b\ b\ id\ c\ c'$]
by (simp add: option.map-id)

lemma *while-saturate-code*[code]: $while_saturate\ f\ M =$

$while_Some\ (\lambda M. let\ M' = f\ M\ in\ if\ M' \subseteq M\ then\ None\ else\ Some\ (M \cup M'))\ M$

unfolding *while-saturate-def* *Let-def* *while-Some-def*
by (*auto intro!*: *while-option-cong* *split*: *if-splits*)

Termination:

lemma *while-option-sat-finite-subset-Some*: **fixes** $C :: 'a \text{ set}$
assumes *mono* f **and** $\bigwedge X. X \subseteq C \implies f X \subseteq C$ **and** *finite* C **and** $M \subseteq C$
shows $\exists S. \text{while-option } (\lambda M. \neg f M \subseteq M) (\lambda M. M \cup f M) M = \text{Some } S$
proof(*rule measure-while-option-Some*[**where**
 $f = \%A::'a \text{ set. card } C - \text{card } A$ **and** $P = \%A. M \subseteq A \wedge A \subseteq C$ **and** $s = M$])
fix A **assume** $A: M \subseteq A \wedge A \subseteq C \neg f A \subseteq A$
show $(M \subseteq A \cup f A \wedge A \cup f A \subseteq C) \wedge \text{card } C - \text{card } (A \cup f A) < \text{card } C - \text{card } A$
(is ?L \wedge ?R)
proof
show ?L **by** (*metis* *assms*(2) $A(1)$ *sup.coboundedI1* *le-sup-iff*)
show ?R **using** A *assms*(2,3) *card-seteq* *finite-subset*
by (*metis* *diff-less-mono2* *finite-Un* *linorder-not-le* *sup-ge1* *sup-ge2*)
qed
next
show $M \subseteq M \wedge M \subseteq C$ **using** $\langle M \subseteq C \rangle$ **by** *blast*
qed

corollary *while-saturate-finite-subset-Some*:

assumes *mono* f **and** $\bigwedge X. X \subseteq C \implies f X \subseteq C$ **and** *finite* C **and** $M \subseteq C$
shows $\exists S. \text{while-saturate } f M = \text{Some } S$
unfolding *while-saturate-def*
using *while-option-sat-finite-subset-Some* *assms* **by** *blast*

Correctness: finds the least saturated/closed set above M

lemma *while-option-sat-prefix*: **assumes** *mono* f
and *while-option* $(\lambda M. \neg f M \subseteq M) (\lambda M. M \cup f M) M = \text{Some } S$
and $M \subseteq P$ **and** $f P \subseteq P$
shows $S \subseteq P$
proof –
have $(\lambda M. M \cup f M) \rightsquigarrow^k M \subseteq P$ **for** k
proof (*induction* k)
case 0 **thus** ?case **using** $\langle M \subseteq P \rangle$ **by** *simp*
next
case (*Suc* k) **thus** ?case
by *simp* (*meson* $\langle f P \subseteq P \rangle$ *monoD*[*OF* $\langle \text{mono } f \rangle$] *order.trans*)
qed
thus ?thesis **by** (*metis* *assms*(2) *while-option-stop2*)
qed

corollary *while-saturate-prefix*:

$\llbracket \text{mono } f; \text{while-saturate } f M = \text{Some } S; M \subseteq P; f P \subseteq P \rrbracket \implies S \subseteq P$
using *while-option-sat-prefix* **unfolding** *while-saturate-def* **by** *blast*

4.5 Reflexive, transitive closure

Computing the reflexive, transitive closure by iterating a successor function. Stops when an element is found that does not satisfy the test.

More refined (and hence more efficient) versions can be found in ITP 2011 paper by Nipkow (the theories are in the AFP entry Flyspeck by Nipkow) and the AFP article Executable Transitive Closures by René Thiemann.

context

fixes $p :: 'a \Rightarrow bool$
and $f :: 'a \Rightarrow 'a \text{ list}$
and $x :: 'a$

begin

qualified fun *rtrancl-while-test* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow bool$
where *rtrancl-while-test* ($ws, -$) = $(ws \neq [] \wedge p(hd\ ws))$

qualified fun *rtrancl-while-step* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow 'a \text{ list} \times 'a \text{ set}$
where *rtrancl-while-step* (ws, Z) =
 $(let\ x = hd\ ws; new = remdups\ (filter\ (\lambda y. y \notin Z)\ (f\ x))$
 $in\ (new\ @\ tl\ ws, set\ new\ \cup\ Z))$

definition *rtrancl-while* :: $('a \text{ list} * 'a \text{ set}) \text{ option}$

where *rtrancl-while* = *while-option* *rtrancl-while-test* *rtrancl-while-step* ($[x], \{x\}$)

qualified fun *rtrancl-while-invariant* :: $'a \text{ list} \times 'a \text{ set} \Rightarrow bool$

where *rtrancl-while-invariant* (ws, Z) =
 $(x \in Z \wedge set\ ws \subseteq Z \wedge distinct\ ws \wedge \{(x, y). y \in set(f\ x)\} \text{ “ } (Z - set\ ws) \subseteq Z$
 \wedge
 $Z \subseteq \{(x, y). y \in set(f\ x)\}^* \text{ “ } \{x\} \wedge (\forall z \in Z - set\ ws. p\ z))$

qualified lemma *rtrancl-while-invariant*:

assumes *inv*: *rtrancl-while-invariant* *st* **and** *test*: *rtrancl-while-test* *st*
shows *rtrancl-while-invariant* (*rtrancl-while-step* *st*)

proof (*cases* *st*)

fix $ws\ Z$

assume *st*: $st = (ws, Z)$

with *test* **obtain** $h\ t$ **where** $ws = h \# t\ p\ h$ **by** (*cases* ws) *auto*

with *inv* *st* **show** ?thesis **by** (*auto* *intro*: *rtrancl.rtrancl-into-rtrancl*)

qed

lemma *rtrancl-while-Some*:

assumes *rtrancl-while* = *Some*(ws, Z)

shows *if* $ws = []$

$then\ Z = \{(x, y). y \in set(f\ x)\}^* \text{ “ } \{x\} \wedge (\forall z \in Z. p\ z)$

$else\ \neg p(hd\ ws) \wedge hd\ ws \in \{(x, y). y \in set(f\ x)\}^* \text{ “ } \{x\}$

proof –

have *rtrancl-while-invariant* ($[x], \{x\}$) **by** *simp*

with *rtrancl-while-invariant* **have** *I*: *rtrancl-while-invariant* (ws, Z)

by (*rule* *while-option-rule*[*OF* - *assms*[*unfolded* *rtrancl-while-def*]])

```

show ?thesis
proof (cases ws = [])
  case True
    thus ?thesis using I
    by (auto simp del:Image-Collect-case-prod dest: Image-closed-trancl)
  next
    case False
    thus ?thesis using I while-option-stop[OF assms[unfolded rtrancl-while-def]]
    by (simp add: subset-iff)
qed
qed

lemma rtrancl-while-finite-Some:
  assumes finite ({(x, y). y ∈ set (f x)}* “{x}”) (is finite ?Cl)
  shows ∃y. rtrancl-while = Some y
proof –
  let ?R = (λ(-, Z). card (?Cl - Z)) < *mlex* > (λ(ws, -). length ws) < *mlex* >
  {}
  have wf ?R by (blast intro: wf-mlex)
  then show ?thesis unfolding rtrancl-while-def
  proof (rule wf-rel-while-option-Some[of ?R rtrancl-while-invariant])
    fix st
    assume *: rtrancl-while-invariant st ∧ rtrancl-while-test st
    hence I: rtrancl-while-invariant (rtrancl-while-step st)
    by (blast intro: rtrancl-while-invariant)
    show (rtrancl-while-step st, st) ∈ ?R
    proof (cases st)
      fix ws Z
      let ?ws = fst (rtrancl-while-step st)
      let ?Z = snd (rtrancl-while-step st)
      assume st: st = (ws, Z)
      with * obtain h t where ws: ws = h # t p h by (cases ws) auto
      show ?thesis
      proof (cases remdups (filter (λy. y ∉ Z) (f h)) = [])
        case False
          then obtain z where z ∈ set (remdups (filter (λy. y ∉ Z) (f h))) by
fastforce
          with st ws I have Z ⊂ ?Z Z ⊆ ?Cl ?Z ⊆ ?Cl by auto
          with assms have card (?Cl - ?Z) < card (?Cl - Z) by (blast intro:
psubset-card-mono)
          with st ws show ?thesis unfolding mlex-prod-def by simp
        next
          case True
          with st ws have ?Z = Z ?ws = t by (auto simp: filter-empty-conv)
          with st ws show ?thesis unfolding mlex-prod-def by simp
      qed
    qed
  qed (simp-all add: rtrancl-while-invariant)
qed

```

end

end

5 The Bourbaki-Witt tower construction for transfinite iteration

theory *Bourbaki-Witt-Fixpoint*
imports *While-Combinator*
begin

lemma *ChainsI* [intro?]:
 $(\bigwedge a\ b. \llbracket a \in Y; b \in Y \rrbracket \implies (a, b) \in r \vee (b, a) \in r) \implies Y \in \text{Chains } r$
unfolding *Chains-def* **by** *blast*

lemma *in-Chains-subset*: $\llbracket M \in \text{Chains } r; M' \subseteq M \rrbracket \implies M' \in \text{Chains } r$
by(*auto simp add: Chains-def*)

lemma *in-ChainsD*: $\llbracket M \in \text{Chains } r; x \in M; y \in M \rrbracket \implies (x, y) \in r \vee (y, x) \in r$
unfolding *Chains-def* **by** *fast*

lemma *Chains-FieldD*: $\llbracket M \in \text{Chains } r; x \in M \rrbracket \implies x \in \text{Field } r$
by(*auto simp add: Chains-def intro: FieldI1 FieldI2*)

lemma *in-Chains-conv-chain*: $M \in \text{Chains } r \longleftrightarrow \text{Complete-Partial-Order.chain}$
 $(\lambda x\ y. (x, y) \in r) \ M$
by(*simp add: Chains-def chain-def*)

lemma *partial-order-on-trans*:
 $\llbracket \text{partial-order-on } A\ r; (x, y) \in r; (y, z) \in r \rrbracket \implies (x, z) \in r$
by(*auto simp add: order-on-defs dest: transD*)

locale *bourbaki-witt-fixpoint* =
fixes *lub* :: $'a\ \text{set} \Rightarrow 'a$
and *leq* :: $('a \times 'a)\ \text{set}$
and *f* :: $'a \Rightarrow 'a$
assumes *po*: *Partial-order leq*
and *lub-least*: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\}; \bigwedge x. x \in M \implies (x, z) \in \text{leq} \rrbracket \implies (\text{lub } M, z) \in \text{leq}$
and *lub-upper*: $\llbracket M \in \text{Chains } \text{leq}; x \in M \rrbracket \implies (x, \text{lub } M) \in \text{leq}$
and *lub-in-Field*: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\} \rrbracket \implies \text{lub } M \in \text{Field } \text{leq}$
and *increasing*: $\bigwedge x. x \in \text{Field } \text{leq} \implies (x, f\ x) \in \text{leq}$
begin

lemma *leq-trans*: $\llbracket (x, y) \in \text{leq}; (y, z) \in \text{leq} \rrbracket \implies (x, z) \in \text{leq}$
by(*rule partial-order-on-trans[OF po]*)

lemma *leq-refl*: $x \in \text{Field } \text{leq} \implies (x, x) \in \text{leq}$
using *po* **by**(*simp add: order-on-defs refl-on-def*)

lemma *leq-antisym*: $\llbracket (x, y) \in \text{leq}; (y, x) \in \text{leq} \rrbracket \implies x = y$
using *po* **by**(*simp add: order-on-defs antisym-def*)

inductive-set *iterates-above* :: $'a \Rightarrow 'a \text{ set}$
for *a*
where
 base: $a \in \text{iterates-above } a$
 | *step*: $x \in \text{iterates-above } a \implies f x \in \text{iterates-above } a$
 | *Sup*: $\llbracket M \in \text{Chains } \text{leq}; M \neq \{\}; \bigwedge x. x \in M \implies x \in \text{iterates-above } a \rrbracket \implies \text{lub } M \in \text{iterates-above } a$

definition *fixp-above* :: $'a \Rightarrow 'a$
where *fixp-above* $a = (\text{if } a \in \text{Field } \text{leq} \text{ then } \text{lub } (\text{iterates-above } a) \text{ else } a)$

lemma *fixp-above-outside*: $a \notin \text{Field } \text{leq} \implies \text{fixp-above } a = a$
by(*simp add: fixp-above-def*)

lemma *fixp-above-inside*: $a \in \text{Field } \text{leq} \implies \text{fixp-above } a = \text{lub } (\text{iterates-above } a)$
by(*simp add: fixp-above-def*)

context
 notes *leq-refl* [*intro!*, *simp*]
 and *base* [*intro*]
 and *step* [*intro*]
 and *Sup* [*intro*]
 and *leq-trans* [*trans*]
begin

lemma *iterates-above-le-f*: $\llbracket x \in \text{iterates-above } a; a \in \text{Field } \text{leq} \rrbracket \implies (x, f x) \in \text{leq}$
by(*induction x rule: iterates-above.induct*)(*blast intro: increasing FieldI2 lub-in-Field*)⁺

lemma *iterates-above-Field*: $\llbracket x \in \text{iterates-above } a; a \in \text{Field } \text{leq} \rrbracket \implies x \in \text{Field } \text{leq}$
by(*drule (1) iterates-above-le-f*)(*rule FieldI1*)

lemma *iterates-above-ge*:
 assumes *y*: $y \in \text{iterates-above } a$
 and *a*: $a \in \text{Field } \text{leq}$
 shows $(a, y) \in \text{leq}$
using *y* **by**(*induction*)(*auto intro: a increasing iterates-above-le-f leq-trans leq-trans[OF - lub-upper]*)

lemma *iterates-above-lub*:
 assumes *M*: $M \in \text{Chains } \text{leq}$
 and *nempty*: $M \neq \{\}$
 and *upper*: $\bigwedge y. y \in M \implies \exists z \in M. (y, z) \in \text{leq} \wedge z \in \text{iterates-above } a$

shows $\text{lub } M \in \text{iterates-above } a$
proof –
 let $?M = M \cap \text{iterates-above } a$
 from M have $M': ?M \in \text{Chains } \text{leq}$ **by** (rule in-Chains-subset) simp
 have $?M \neq \{\}$ **using** nempty **by** (auto dest: upper)
 with M' have $\text{lub } ?M \in \text{iterates-above } a$ **by** (rule Sup) blast
 also have $\text{lub } ?M = \text{lub } M$ **using** nempty
by (intro leq-antisym) (blast intro!: lub-least[OF M] lub-least[OF M] intro: lub-upper[OF M] lub-upper[OF M] leq-trans dest: upper)+
 finally show $?thesis$.
qed

lemma iterates-above-successor:
 assumes $y: y \in \text{iterates-above } a$
 and $a: a \in \text{Field } \text{leq}$
 shows $y = a \vee y \in \text{iterates-above } (f a)$
using y
proof induction
 case base **thus** $?case$ **by** simp
next
 case (step x) **thus** $?case$ **by** auto
next
 case (Sup M)
 show $?case$
proof (cases $\exists x. M \subseteq \{x\}$)
 case True
 with $\langle M \neq \{\} \rangle$ obtain y where $M: M = \{y\}$ **by** auto
 have $\text{lub } M = y$
by (rule leq-antisym) (auto intro!: lub-upper Sup lub-least ChainsI simp add: $a \in M \text{ Sup.hyps}(3)[\text{of } y, \text{ THEN iterates-above-Field}]$ dest: iterates-above-Field)
 with $\text{Sup.IH}[\text{of } y] M$ show $?thesis$ **by** simp
next
 case False
 from $\text{Sup}(1-2)$ have $\text{lub } M \in \text{iterates-above } (f a)$
proof (rule iterates-above-lub)
 fix y
 assume $y: y \in M$
 from $\text{Sup.IH}[\text{OF this}]$ show $\exists z \in M. (y, z) \in \text{leq} \wedge z \in \text{iterates-above } (f a)$
proof
 assume $y = a$
 from y False obtain z where $z: z \in M$ and $\text{neg: } y \neq z$ **by** (metis insertI1 subsetI)
 with $\text{Sup.IH}[\text{OF } z] \langle y = a \rangle \text{ Sup.hyps}(3)[\text{OF } z]$
 show $?thesis$ **by** (auto dest: iterates-above-ge intro: a)
next
 assume $*: y \in \text{iterates-above } (f a)$
 with increasing[OF a] have $y \in \text{Field } \text{leq}$
by (auto dest!: iterates-above-Field intro: FieldI2)
 with $*$ show $?thesis$ **using** y **by** auto

qed
 qed
 thus ?thesis by simp
 qed
 qed

lemma *iterates-above-Sup-aux*:

assumes $M: M \in \text{Chains } \text{leq } M \neq \{\}$
 and $M': M' \in \text{Chains } \text{leq } M' \neq \{\}$
 and *comp*: $\bigwedge x. x \in M \implies x \in \text{iterates-above } (\text{lub } M') \vee \text{lub } M' \in \text{iterates-above } x$
 shows $(\text{lub } M, \text{lub } M') \in \text{leq} \vee \text{lub } M \in \text{iterates-above } (\text{lub } M')$
proof(cases $\exists x \in M. x \in \text{iterates-above } (\text{lub } M')$)
 case *True*
 then obtain x where $x: x \in M \ x \in \text{iterates-above } (\text{lub } M')$ by blast
 have $\text{lub-}M': \text{lub } M' \in \text{Field } \text{leq}$ using M' by(rule *lub-in-Field*)
 have $\text{lub } M \in \text{iterates-above } (\text{lub } M')$ using M
proof(rule *iterates-above-lub*)
 fix y
 assume $y: y \in M$
 from *comp*[*OF* y] show $\exists z \in M. (y, z) \in \text{leq} \wedge z \in \text{iterates-above } (\text{lub } M')$
proof
 assume $y \in \text{iterates-above } (\text{lub } M')$
 from *this iterates-above-Field*[*OF this*] y $\text{lub-}M'$ show ?thesis by blast
 next
 assume $\text{lub } M' \in \text{iterates-above } y$
 hence $(y, \text{lub } M') \in \text{leq}$ using *Chains-FieldD*[*OF* $M(1)$ y] by(rule *iterates-above-ge*)
 also have $(\text{lub } M', x) \in \text{leq}$ using $x(2)$ $\text{lub-}M'$ by(rule *iterates-above-ge*)
 finally show ?thesis using x by blast
 qed
 qed
 thus ?thesis ..
 next
 case *False*
 have $(\text{lub } M, \text{lub } M') \in \text{leq}$ using M
proof(rule *lub-least*)
 fix x
 assume $x: x \in M$
 from *comp*[*OF* x] x *False* have $\text{lub } M' \in \text{iterates-above } x$ by auto
 moreover from $M(1)$ x have $x \in \text{Field } \text{leq}$ by(rule *Chains-FieldD*)
 ultimately show $(x, \text{lub } M') \in \text{leq}$ by(rule *iterates-above-ge*)
 qed
 thus ?thesis ..
 qed

lemma *iterates-above-triangle*:

assumes $x: x \in \text{iterates-above } a$
 and $y: y \in \text{iterates-above } a$


```

and  $a: a \in \text{Field } \text{leq}$ 
shows  $x \in \text{iterates-above } y \vee y \in \text{iterates-above } x$ 
using  $x \ y$ 
proof(induction arbitrary: y)
  case base then show  $?case$  by simp
next
  case (step x) thus  $?case$  using  $a$ 
    by(auto dest: iterates-above-successor intro: iterates-above-Field)
next
  case  $x: (\text{Sup } M)$ 
    hence  $\text{lub } M \in \text{iterates-above } a$  by blast
    from  $\langle y \in \text{iterates-above } a \rangle$  show  $?case$ 
    proof(induction)
      case base show  $?case$  using  $\text{lub}$  by simp
    next
      case (step y) thus  $?case$  using  $a$ 
        by(auto dest: iterates-above-successor intro: iterates-above-Field)
    next
      case  $y: (\text{Sup } M')$ 
        hence  $\text{lub } M' \in \text{iterates-above } a$  by blast
        have  $*$ :  $x \in \text{iterates-above } (\text{lub } M') \vee \text{lub } M' \in \text{iterates-above } x$  if  $x \in M$  for  $x$ 
          using that lub' by(rule x.IH)
        with  $x(1-2) \ y(1-2)$  have  $(\text{lub } M, \text{lub } M') \in \text{leq} \vee \text{lub } M \in \text{iterates-above } (\text{lub } M')$ 
          by(rule iterates-above-Sup-aux)
        moreover from  $y(1-2) \ x(1-2)$  have  $(\text{lub } M', \text{lub } M) \in \text{leq} \vee \text{lub } M' \in \text{iterates-above } (\text{lub } M)$ 
          by(rule iterates-above-Sup-aux)(blast dest: y.IH)
        ultimately show  $?case$  by(auto 4 3 dest: leq-antisym)
      qed
    qed

```

lemma *chain-iterates-above*:

```

assumes  $a: a \in \text{Field } \text{leq}$ 
shows  $\text{iterates-above } a \in \text{Chains } \text{leq}$  (is  $?C \in -$ )
proof (rule ChainsI)
  fix  $x \ y$ 
  assume  $x \in ?C \ y \in ?C$ 
  hence  $x \in \text{iterates-above } y \vee y \in \text{iterates-above } x$  using  $a$  by(rule iterates-above-triangle)
  moreover from  $\langle x \in ?C \rangle \ a$  have  $x \in \text{Field } \text{leq}$  by(rule iterates-above-Field)
  moreover from  $\langle y \in ?C \rangle \ a$  have  $y \in \text{Field } \text{leq}$  by(rule iterates-above-Field)
  ultimately show  $(x, y) \in \text{leq} \vee (y, x) \in \text{leq}$  by(auto dest: iterates-above-ge)
qed

```

lemma *fixp-iterates-above*: $\text{fixp-above } a \in \text{iterates-above } a$

by(*auto intro: chain-iterates-above simp add: fixp-above-def*)

lemma *fixp-above-Field*: $a \in \text{Field } \text{leq} \implies \text{fixp-above } a \in \text{Field } \text{leq}$

using *fixp-iterates-above* **by**(*rule iterates-above-Field*)

```

lemma fixp-above-unfold:
  assumes a:  $a \in \text{Field } \text{leq}$ 
  shows  $\text{fixp-above } a = f (\text{fixp-above } a)$  (is  $?a = f ?a$ )
proof(rule leq-antisym)
  show  $(?a, f ?a) \in \text{leq}$  using fixp-above-Field[OF a] by(rule increasing)

  have  $f ?a \in \text{iterates-above } a$  using fixp-iterates-above by(rule iterates-above.step)
  with chain-iterates-above[OF a] show  $(f ?a, ?a) \in \text{leq}$ 
    by(simp add: fixp-above-inside assms lub-upper)
qed

end

lemma fixp-above-induct [case-names adm base step]:
  assumes adm:  $\text{ccpo.admissible } \text{lub } (\lambda x y. (x, y) \in \text{leq}) P$ 
  and base:  $P a$ 
  and step:  $\bigwedge x. P x \implies P (f x)$ 
  shows  $P (\text{fixp-above } a)$ 
proof(cases  $a \in \text{Field } \text{leq}$ )
  case True
  from adm chain-iterates-above[OF True]
  show ?thesis unfolding fixp-above-inside[OF True] in-Chains-conv-chain
  proof(rule ccpo.admissibleD)
    have  $a \in \text{iterates-above } a \dots$ 
    then show  $\text{iterates-above } a \neq \{\}$  by(auto)
    show  $P x$  if  $x \in \text{iterates-above } a$  for  $x$  using that
      by induction(auto intro: base step simp add: in-Chains-conv-chain dest:
ccpo.admissibleD[OF adm])
    qed
  qed(simp add: fixp-above-outside base)

end

```

5.1 Connect with the while combinator for executability on chain-finite lattices.

context bourbaki-witt-fixpoint **begin**

```

lemma in-Chains-finite: — Translation from  $\llbracket \text{Complete-Partial-Order.chain } (\leq) \rrbracket$ 
 $?A; \text{finite } ?A; ?A \neq \{\}$   $\implies \text{Sup } ?A \in ?A$ .
  assumes  $M \in \text{Chains } \text{leq}$ 
  and  $M \neq \{\}$ 
  and  $\text{finite } M$ 
  shows  $\text{lub } M \in M$ 
using assms(3,1,2)
proof induction
  case empty thus ?case by simp
next

```

```

case (insert x M)
note chain =  $\langle \text{insert } x \ M \in \text{Chains } \text{leq} \rangle$ 
show ?case
proof(cases M =  $\{\}$ )
  case True thus ?thesis
    using chain in-ChainsD leq-antisym lub-least lub-upper by fastforce
next
  case False
from chain have chain':  $M \in \text{Chains } \text{leq}$ 
  using in-Chains-subset subset-insertI by blast
hence  $\text{lub } M \in M$  using False by(rule insert.IH)
show ?thesis
proof(cases ( $x, \text{lub } M$ )  $\in \text{leq}$ )
  case True
have ( $\text{lub } (\text{insert } x \ M), \text{lub } M$ )  $\in \text{leq}$  using chain
  by (rule lub-least) (auto simp: True intro: lub-upper[OF chain'])
with False have  $\text{lub } (\text{insert } x \ M) = \text{lub } M$ 
  using lub-upper[OF chain] lub-least[OF chain'] by (blast intro: leq-antisym)
with  $\langle \text{lub } M \in M \rangle$  show ?thesis by simp
next
  case False
with in-ChainsD[OF chain, of x lub M]  $\langle \text{lub } M \in M \rangle$ 
have  $\text{lub } (\text{insert } x \ M) = x$ 
  by – (rule leq-antisym, (blast intro: FieldI2 chain chain' insert.premis(2)
leq-refl leq-trans lub-least lub-upper)+)
  thus ?thesis by simp
qed
qed
qed

```

lemma *fun-pow-iterates-above*: $(f \rightsquigarrow^k) a \in \text{iterates-above } a$
using *iterates-above.base* *iterates-above.step* **by** (*induct k*) *simp-all*

lemma *chfin-iterates-above-fun-pow*:
assumes $x \in \text{iterates-above } a$
assumes $\forall M \in \text{Chains } \text{leq}. \text{finite } M$
shows $\exists j. x = (f \rightsquigarrow^j) a$
using *assms(1)*
proof *induct*
case *base* **then show** ?*case* **by** (*simp add: exI[where x=0]*)
next
case (*step* *x*) **then obtain** *j* **where** $x = (f \rightsquigarrow^j) a$ **by** *blast*
with *step(1)* **show** ?*case* **by** (*simp add: exI[where x=Suc j]*)
next
case (*Sup* *M*) **with** *in-Chains-finite* *assms(2)* **show** ?*case* **by** *blast*
qed

lemma *Chain-finite-iterates-above-fun-pow-iff*:
assumes $\forall M \in \text{Chains } \text{leq}. \text{finite } M$

shows $x \in \text{iterates-above } a \iff (\exists j. x = (f \smallfrown j) a)$
using *chfin-iterates-above-fun-pow fun-pow-iterates-above assms* **by** *blast*

lemma *fixp-above-Kleene-iter-ex*:
assumes $(\forall M \in \text{Chains } \text{leq}. \text{finite } M)$
obtains k **where** $\text{fixp-above } a = (f \smallfrown k) a$
using *assms* **by** *atomize-elim (simp add: chfin-iterates-above-fun-pow fixp-iterates-above)*

context *fixes a* **assumes** $a \in \text{Field } \text{leq}$ **begin**

lemma *funpow-Field-leq*: $(f \smallfrown k) a \in \text{Field } \text{leq}$
using a **by** *(induct k) (auto intro: increasing FieldI2)*

lemma *funpow-prefix*: $j < k \implies ((f \smallfrown j) a, (f \smallfrown k) a) \in \text{leq}$
proof *(induct k)*
case *(Suc k)*
with *leq-trans[OF - increasing[OF funpow-Field-leq]] funpow-Field-leq increasing a*
show *?case* **by** *simp (metis less-antisym)*
qed *simp*

lemma *funpow-suffix*: $(f \smallfrown \text{Suc } k) a = (f \smallfrown k) a \implies ((f \smallfrown (j + k)) a, (f \smallfrown k) a) \in \text{leq}$
using *funpow-Field-leq*
by *(induct j) (simp-all del: funpow.simps add: funpow-Suc-right funpow-add leq-refl)*

lemma *funpow-stability*: $(f \smallfrown \text{Suc } k) a = (f \smallfrown k) a \implies ((f \smallfrown j) a, (f \smallfrown k) a) \in \text{leq}$
using *funpow-prefix funpow-suffix* **[where** $j=j - k$ **and** $k=k$ **]** **by** *(cases j < k) simp-all*

lemma *funpow-in-Chains*: $\{(f \smallfrown k) a \mid k. \text{True}\} \in \text{Chains } \text{leq}$
using *chain-iterates-above[OF a] fun-pow-iterates-above*
by *(blast intro: ChainsI dest: in-ChainsD)*

lemma *fixp-above-Kleene-iter*:
assumes $\forall M \in \text{Chains } \text{leq}. \text{finite } M$ — convenient but surely not necessary
assumes $(f \smallfrown \text{Suc } k) a = (f \smallfrown k) a$
shows $\text{fixp-above } a = (f \smallfrown k) a$
proof *(rule leq-antisym)*
show $(\text{fixp-above } a, (f \smallfrown k) a) \in \text{leq}$ **using** *assms a*
by *(auto simp add: fixp-above-def chain-iterates-above Chain-finite-iterates-above-fun-pow-iff funpow-stability[OF assms(2)] intro!: lub-least intro: iterates-above.base)*
show $((f \smallfrown k) a, \text{fixp-above } a) \in \text{leq}$ **using** a
by *(auto simp add: fixp-above-def chain-iterates-above fun-pow-iterates-above intro!: lub-upper)*
qed

context **assumes** *chfin*: $\forall M \in \text{Chains } \text{leq}. \text{finite } M$ **begin**

```

lemma Chain-finite-wf: wf {(f ((f  $\sim$  k) a), (f  $\sim$  k) a) | k. f ((f  $\sim$  k) a)  $\neq$  (f  $\sim$ 
k) a}
apply(rule wf-measure[where f= $\lambda$ b. card {(f  $\sim$  j) a | j. (b, (f  $\sim$  j) a)  $\in$  leq},
THEN wf-subset])
apply(auto simp: set-eq-iff intro!: psubset-card-mono[OF finite-subset[OF - bspec[OF
chfin funpow-in-Chains]]])
apply(metis funpow-Field-leq increasing leq-antisym leq-trans leq-refl)+
done

lemma while-option-finite-increasing:  $\exists P$ . while-option ( $\lambda A$ . f A  $\neq$  A) f a = Some
P
by(rule wf-rel-while-option-Some[OF Chain-finite-wf, where P= $\lambda A$ . ( $\exists k$ . A = (f
 $\sim$  k) a)  $\wedge$  (A, f A)  $\in$  leq and s=a])
(auto simp: a increasing chfin FieldI2 chfin-iterates-above-fun-pow fun-pow-iterates-above
iterates-above.step intro: exI[where x=0])

lemma fixp-above-the-while-option: fixp-above a = the (while-option ( $\lambda A$ . f A  $\neq$ 
A) f a)
proof –
  obtain P where while-option ( $\lambda A$ . f A  $\neq$  A) f a = Some P
  using while-option-finite-increasing by blast
  with while-option-stop2[OF this] fixp-above-Kleene-iter[OF chfin]
  show ?thesis by fastforce
qed

lemma fixp-above-conv-while: fixp-above a = while ( $\lambda A$ . f A  $\neq$  A) f a
unfolding while-def by (rule fixp-above-the-while-option)

end

end

end

lemma bourbaki-witt-fixpoint-complete-latticeI:
  fixes f :: 'a::complete-lattice  $\Rightarrow$  'a
  assumes  $\bigwedge x$ . x  $\leq$  f x
  shows bourbaki-witt-fixpoint Sup {(x, y). x  $\leq$  y} f
by unfold-locales (auto simp: assms Sup-upper order-on-defs Field-def intro: refl-onI
transI antisymI Sup-least)

end

```

6 Division with modulus centered towards zero.

```

theory Centered-Division
  imports Main
begin

```

lemma *off-iff-abs-mod-2-eq-one*:

$\langle \text{odd } l \longleftrightarrow |l| \bmod 2 = 1 \rangle$ **for** $l :: \text{int}$
by (*simp flip: odd-iff-mod-2-eq-one*)

The following specification of division on integers centers the modulus around zero. This is useful e.g. to define division on Gauss numbers. N.b.: This is not mentioned [2].

definition *centered-divide* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ (**infixl** $\langle \text{cdiv} \rangle$ 70)

where $\langle k \text{ cdiv } l = \text{sgn } l * ((k + |l| \text{ div } 2) \text{ div } |l|) \rangle$

definition *centered-modulo* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$ (**infixl** $\langle \text{cmod} \rangle$ 70)

where $\langle k \text{ cmod } l = (k + |l| \text{ div } 2) \bmod |l| - |l| \text{ div } 2 \rangle$

Example: $k \text{ cmod } 5 \in \{-2, -1, 0, 1, 2\}$

lemma *signed-take-bit-eq-cmod*:

$\langle \text{signed-take-bit } n \ k = k \text{ cmod } (2^{\text{Suc } n}) \rangle$

by (*simp only: centered-modulo-def power-abs abs-numeral flip: take-bit-eq-mod*)
(simp add: signed-take-bit-eq-take-bit-shift)

Property *signed-take-bit* $n \ k = k \text{ cmod } 2^{\text{Suc } n}$ is the key to generalize centered division to arbitrary structures satisfying *ring-bit-operations*, but so far it is not clear what practical relevance that would have.

lemma *cdiv-mult-cmod-eq*:

$\langle k \text{ cdiv } l * l + k \text{ cmod } l = k \rangle$

proof –

have *: $\langle l * (\text{sgn } l * j) = |l| * j \rangle$ **for** j

by (*simp add: ac-simps abs-sgn*)

show *?thesis*

by (*simp add: centered-divide-def centered-modulo-def algebra-simps **)

qed

lemma *mult-cdiv-cmod-eq*:

$\langle l * (k \text{ cdiv } l) + k \text{ cmod } l = k \rangle$

using *cdiv-mult-cmod-eq [of k l]* **by** (*simp add: ac-simps*)

lemma *cmod-cdiv-mult-eq*:

$\langle k \text{ cmod } l + k \text{ cdiv } l * l = k \rangle$

using *cdiv-mult-cmod-eq [of k l]* **by** (*simp add: ac-simps*)

lemma *cmod-mult-cdiv-eq*:

$\langle k \text{ cmod } l + l * (k \text{ cdiv } l) = k \rangle$

using *cdiv-mult-cmod-eq [of k l]* **by** (*simp add: ac-simps*)

lemma *minus-cdiv-mult-eq-cmod*:

$\langle k - k \text{ cdiv } l * l = k \text{ cmod } l \rangle$

by (*rule add-implies-diff [symmetric]*) (*fact cmod-cdiv-mult-eq*)

lemma *minus-mult-cdiv-eq-cmod*:

$\langle k - l * (k \text{ cdiv } l) = k \text{ cmod } l \rangle$
by (rule add-implies-diff [symmetric]) (fact cmod-mult-cdiv-eq)

lemma minus-cmod-eq-cdiv-mult:
 $\langle k - k \text{ cmod } l = k \text{ cdiv } l * l \rangle$
by (rule add-implies-diff [symmetric]) (fact cdiv-mult-cmod-eq)

lemma minus-cmod-eq-mult-cdiv:
 $\langle k - k \text{ cmod } l = l * (k \text{ cdiv } l) \rangle$
by (rule add-implies-diff [symmetric]) (fact mult-cdiv-cmod-eq)

lemma cdiv-0-eq [simp]:
 $\langle k \text{ cdiv } 0 = 0 \rangle$
by (simp add: centered-divide-def)

lemma cmod-0-eq [simp]:
 $\langle k \text{ cmod } 0 = k \rangle$
by (simp add: centered-modulo-def)

lemma cdiv-1-eq [simp]:
 $\langle k \text{ cdiv } 1 = k \rangle$
by (simp add: centered-divide-def)

lemma cmod-1-eq [simp]:
 $\langle k \text{ cmod } 1 = 0 \rangle$
by (simp add: centered-modulo-def)

lemma zero-cdiv-eq [simp]:
 $\langle 0 \text{ cdiv } k = 0 \rangle$
by (auto simp add: centered-divide-def not-less zdiv-eq-0-iff)

lemma zero-cmod-eq [simp]:
 $\langle 0 \text{ cmod } k = 0 \rangle$
by (auto simp add: centered-modulo-def not-less zmod-trivial-iff)

lemma cdiv-minus-eq:
 $\langle k \text{ cdiv } -l = -(k \text{ cdiv } l) \rangle$
by (simp add: centered-divide-def)

lemma cmod-minus-eq [simp]:
 $\langle k \text{ cmod } -l = k \text{ cmod } l \rangle$
by (simp add: centered-modulo-def)

lemma cdiv-abs-eq:
 $\langle k \text{ cdiv } |l| = \text{sgn } l * (k \text{ cdiv } l) \rangle$
by (simp add: centered-divide-def)

lemma cmod-abs-eq [simp]:
 $\langle k \text{ cmod } |l| = k \text{ cmod } l \rangle$

```

    by (simp add: centered-modulo-def)

lemma nonzero-mult-cdiv-cancel-right:
   $\langle k * l \text{ cdiv } l = k \rangle$  if  $\langle l \neq 0 \rangle$ 
proof -
  have  $\langle \text{sgn } l * k * |l| \text{ cdiv } l = k \rangle$ 
    using that by (simp add: centered-divide-def)
  with that show ?thesis
    by (simp add: ac-simps abs-sgn)
qed

lemma cdiv-self-eq [simp]:
   $\langle k \text{ cdiv } k = 1 \rangle$  if  $\langle k \neq 0 \rangle$ 
  using that nonzero-mult-cdiv-cancel-right [of k 1] by simp

lemma cmod-self-eq [simp]:
   $\langle k \text{ cmod } k = 0 \rangle$ 
proof -
  have  $\langle (\text{sgn } k * |k| + |k| \text{ div } 2) \text{ mod } |k| = |k| \text{ div } 2 \rangle$ 
    by (auto simp add: zmod-trivial-iff)
  also have  $\langle \text{sgn } k * |k| = k \rangle$ 
    by (simp add: abs-sgn)
  finally show ?thesis
    by (simp add: centered-modulo-def algebra-simps)
qed

lemma cmod-less-divisor:
   $\langle k \text{ cmod } l < |l| - |l| \text{ div } 2 \rangle$  if  $\langle l \neq 0 \rangle$ 
  using that pos-mod-bound [of  $\langle |l| \rangle$ ] by (simp add: centered-modulo-def)

lemma cmod-less-equal-divisor:
   $\langle k \text{ cmod } l \leq |l| \text{ div } 2 \rangle$  if  $\langle l \neq 0 \rangle$ 
proof -
  from that cmod-less-divisor [of l k]
  have  $\langle k \text{ cmod } l < |l| - |l| \text{ div } 2 \rangle$ 
    by simp
  also have  $\langle |l| - |l| \text{ div } 2 = |l| \text{ div } 2 + \text{of\_bool } (\text{odd } l) \rangle$ 
    by auto
  finally show ?thesis
    by (cases  $\langle \text{even } l \rangle$ ) simp-all
qed

lemma divisor-less-equal-cmod':
   $\langle |l| \text{ div } 2 - |l| \leq k \text{ cmod } l \rangle$  if  $\langle l \neq 0 \rangle$ 
proof -
  have  $\langle 0 \leq (k + |l| \text{ div } 2) \text{ mod } |l| \rangle$ 
    using that pos-mod-sign [of  $\langle |l| \rangle$ ] by simp
  then show ?thesis
    by (simp-all add: centered-modulo-def)

```


qed

lemma *divisor-less-equal-cmod*:
 $\langle \neg (|l| \text{ div } 2) \leq k \text{ cmod } l \rangle$ **if** $\langle l \neq 0 \rangle$
using *that divisor-less-equal-cmod'* [of l k]
by (*simp add: centered-modulo-def*)

lemma *abs-cmod-less-equal*:
 $\langle |k \text{ cmod } l| \leq |l| \text{ div } 2 \rangle$ **if** $\langle l \neq 0 \rangle$
using *that divisor-less-equal-cmod* [of l k]
by (*simp add: abs-le-iff cmod-less-equal-divisor*)

end

7 Order on characters

theory *Char-ord*
imports *Main*
begin

instantiation *char* :: *linorder*
begin

definition *less-eq-char* :: $\langle \text{char} \Rightarrow \text{char} \Rightarrow \text{bool} \rangle$
where $\langle c1 \leq c2 \longleftrightarrow \text{of-char } c1 \leq (\text{of-char } c2 :: \text{nat}) \rangle$

definition *less-char* :: $\langle \text{char} \Rightarrow \text{char} \Rightarrow \text{bool} \rangle$
where $\langle c1 < c2 \longleftrightarrow \text{of-char } c1 < (\text{of-char } c2 :: \text{nat}) \rangle$

instance
by *standard* (*auto simp add: less-eq-char-def less-char-def*)

end

lemma *less-eq-char-simp* [*simp, code*]:
 $\langle \text{Char } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ b7 \leq \text{Char } c0 \ c1 \ c2 \ c3 \ c4 \ c5 \ c6 \ c7$
 $\longleftrightarrow \text{lexordp-eq } [b7, b6, b5, b4, b3, b2, b1, b0] [c7, c6, c5, c4, c3, c2, c1, c0] \rangle$
by (*simp only: less-eq-char-def of-char-def char.sel horner-sum-less-eq-iff-lexordp-eq list.size*) *simp*

lemma *less-char-simp* [*simp, code*]:
 $\langle \text{Char } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ b7 < \text{Char } c0 \ c1 \ c2 \ c3 \ c4 \ c5 \ c6 \ c7$
 $\longleftrightarrow \text{ord-class.lexordp } [b7, b6, b5, b4, b3, b2, b1, b0] [c7, c6, c5, c4, c3, c2, c1, c0] \rangle$
by (*simp only: less-char-def of-char-def char.sel horner-sum-less-iff-lexordp list.size*) *simp*

instantiation *char* :: *distrib-lattice*

begin

definition $\langle (inf :: char \Rightarrow -) = min \rangle$

definition $\langle (sup :: char \Rightarrow -) = max \rangle$

instance

by *standard* (*auto simp add: inf-char-def sup-char-def max-min-distrib2*)

end

code-identifier

code-module *Char-ord* \rightarrow

(*SML*) *Str* **and** (*OCaml*) *Str* **and** (*Haskell*) *Str* **and** (*Scala*) *Str*

end

8 A generic phantom type

theory *Phantom-Type*

imports *Main*

begin

datatype (*'a*, *'b*) *phantom* = *phantom* (*of-phantom*: *'b*)

lemma *type-definition-phantom'*: *type-definition of-phantom phantom UNIV*

by(*unfold-locales*) *simp-all*

lemma *phantom-comp-of-phantom* [*simp*]: *phantom* \circ *of-phantom* = *id*

and *of-phantom-comp-phantom* [*simp*]: *of-phantom* \circ *phantom* = *id*

by(*simp-all add: o-def id-def*)

syntax *-Phantom* :: *type* \Rightarrow *logic* ($\langle (\langle indent=1 notation=\langle \text{mixfix } Phantom \rangle Phantom / (1'(-')) \rangle) \rangle$)

syntax-consts *-Phantom* == *phantom*

translations

Phantom(*'t*) \Rightarrow *CONST phantom* :: $- \Rightarrow$ (*'t*, $-$) *phantom*

typed-print-translation \langle

let

fun phantom-tr' ctxt (*Type* (***type-name*** \langle *fun* \rangle , $[-$, *Type* (***type-name*** \langle *phantom* \rangle , $[T, -]$])))) *ts* =

list-comb

(*Syntax.const* ***syntax-const*** \langle *-Phantom* \rangle \$ *Syntax-Phases.term-of-typ ctxt*

T, *ts*)

| *phantom-tr' - -* = *raise Match*;

in [(***const-syntax*** \langle *phantom* \rangle , *phantom-tr'*)] *end*

\rangle

lemma *of-phantom-inject* [*simp*]:

of-phantom *x* = *of-phantom* *y* \longleftrightarrow *x* = *y*

```

by(cases x y rule: phantom.exhaust[case-product phantom.exhaust]) simp
end

```

9 Cardinality of types

```

theory Cardinality
imports Phantom-Type
begin

```

9.1 Preliminary lemmas

```

lemma (in type-definition) univ:
  UNIV = Abs ' A
proof
  show Abs ' A  $\subseteq$  UNIV by (rule subset-UNIV)
  show UNIV  $\subseteq$  Abs ' A
  proof
    fix x :: 'b
    have x = Abs (Rep x) by (rule Rep-inverse [symmetric])
    moreover have Rep x  $\in$  A by (rule Rep)
    ultimately show x  $\in$  Abs ' A by (rule image-eqI)
  qed
qed

```

```

lemma (in type-definition) card: card (UNIV :: 'b set) = card A
  by (simp add: univ card-image inj-on-def Abs-inject)

```

9.2 Cardinalities of types

```

syntax -type-card :: type => nat ( $\langle \langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix CARD} \rangle \rangle \text{CARD} / (1'(-')) \rangle \rangle$ )

```

```

syntax-consts -type-card == card

```

```

translations CARD('t) => CONST card (CONST UNIV :: 't set)

```

```

print-translation <
  let
    fun card-univ-tr' ctxt [Const (const-syntax <UNIV>, Type (-, [T]))] =
      Syntax.const syntax-const <-type-card> $ Syntax-Phases.term-of-typ ctxt T
    in [(const-syntax <card>, card-univ-tr')] end
  >

```

```

lemma card-prod [simp]: CARD('a  $\times$  'b) = CARD('a) * CARD('b)
  unfolding UNIV-Times-UNIV [symmetric] by (simp only: card-cartesian-product)

```

```

lemma card-UNIV-sum: CARD('a + 'b) = (if CARD('a)  $\neq$  0  $\wedge$  CARD('b)  $\neq$  0
  then CARD('a) + CARD('b) else 0)
unfolding UNIV-Plus-UNIV [symmetric]

```

by(*auto simp add: card-eq-0-iff card-Plus simp del: UNIV-Plus-UNIV*)

lemma *card-sum* [*simp*]: $CARD('a + 'b) = CARD('a::finite) + CARD('b::finite)$
by(*simp add: card-UNIV-sum*)

lemma *card-UNIV-option*: $CARD('a\ option) = (if\ CARD('a) = 0\ then\ 0\ else\ CARD('a) + 1)$

proof –

have (*None* :: *'a option*) \notin *range Some* **by** *clarsimp*

thus *?thesis*

by (*simp add: UNIV-option-conv card-eq-0-iff finite-range-Some card-image*)

qed

lemma *card-option* [*simp*]: $CARD('a\ option) = Suc\ CARD('a::finite)$
by(*simp add: card-UNIV-option*)

lemma *card-UNIV-set*: $CARD('a\ set) = (if\ CARD('a) = 0\ then\ 0\ else\ 2 \wedge CARD('a))$
by(*simp add: card-eq-0-iff card-Pow flip: Pow-UNIV*)

lemma *card-set* [*simp*]: $CARD('a\ set) = 2 \wedge CARD('a::finite)$
by(*simp add: card-UNIV-set*)

lemma *card-nat* [*simp*]: $CARD(nat) = 0$
by (*simp add: card-eq-0-iff*)

lemma *card-fun*: $CARD('a \Rightarrow 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0 \vee CARD('b) = 1\ then\ CARD('b) \wedge CARD('a)\ else\ 0)$

proof –

have $CARD('a \Rightarrow 'b) = CARD('b) \wedge CARD('a)$ **if** $0 < CARD('a)$ **and** $0 < CARD('b)$

proof –

from *that* **have** *fin**a*: *finite* (*UNIV* :: *'a set*) **and** *fin**b*: *finite* (*UNIV* :: *'b set*)

by(*simp-all only: card-ge-0-finite*)

from *finite-distinct-list*[*OF fin**b*] **obtain** *bs*

where *bs*: *set* *bs* = (*UNIV* :: *'b set*) **and** *distb*: *distinct* *bs* **by** *blast*

from *finite-distinct-list*[*OF fin**a*] **obtain** *as*

where *as*: *set* *as* = (*UNIV* :: *'a set*) **and** *dista*: *distinct* *as* **by** *blast*

have *cb*: $CARD('b) = length\ bs$

unfolding *bs*[*symmetric*] *distinct-card*[*OF distb*] **..**

have *ca*: $CARD('a) = length\ as$

unfolding *as*[*symmetric*] *distinct-card*[*OF dista*] **..**

let *?xs* = *map* ($\lambda ys.\ the \circ map-of\ (zip\ as\ ys)$) (*List.n-lists* (*length as*) *bs*)

have *UNIV* = *set ?xs*

proof(*rule UNIV-eq-I*)

fix *f* :: *'a* \Rightarrow *'b*

from *as* **have** *f* = *the* $\circ map-of\ (zip\ as\ (map\ f\ as))$

by(*auto simp add: map-of-zip-map*)

thus *f* $\in set\ ?xs$ **using** *bs* **by**(*auto simp add: set-n-lists*)

qed

```

moreover have distinct ?xs unfolding distinct-map
proof(intro conjI distinct-n-lists distb inj-onI)
  fix xs ys :: 'b list
  assume xs: xs ∈ set (List.n-lists (length as) bs)
  and ys: ys ∈ set (List.n-lists (length as) bs)
  and eq: the ∘ map-of (zip as xs) = the ∘ map-of (zip as ys)
  from xs ys have [simp]: length xs = length as length ys = length as
  by(simp-all add: length-n-lists-elem)
  have map-of (zip as xs) = map-of (zip as ys)
  proof
    fix x
    from as bs have ∃ y. map-of (zip as xs) x = Some y ∃ y. map-of (zip as
ys) x = Some y
    by(simp-all add: map-of-zip-is-Some[symmetric])
    with eq show map-of (zip as xs) x = map-of (zip as ys) x
    by(auto dest: fun-cong[where x=x])
  qed
  with dista show xs = ys by(simp add: map-of-zip-inject)
qed
hence card (set ?xs) = length ?xs by(simp only: distinct-card)
moreover have length ?xs = length bs ^ length as by(simp add: length-n-lists)
ultimately show ?thesis using cb ca by simp
qed
moreover have CARD('a ⇒ 'b) = 1 if CARD('b) = 1
proof –
  from that obtain b where b: UNIV = {b :: 'b} by(auto simp add: card-Suc-eq)
  have eq: UNIV = {λx :: 'a. b :: 'b}
  proof(rule UNIV-eq-I)
    fix x :: 'a ⇒ 'b
    have x y = b for y
    proof –
      have x y ∈ UNIV ..
      thus ?thesis unfolding b by simp
    qed
    thus x ∈ {λx. b} by(auto)
  qed
  show ?thesis unfolding eq by simp
qed
ultimately show ?thesis
  by(auto simp del: One-nat-def)(auto simp add: card-eq-0-iff dest: finite-fun-UNIVD2
finite-fun-UNIVD1)
qed

corollary finite-UNIV-fun:
  finite (UNIV :: ('a ⇒ 'b) set) ⟷
  finite (UNIV :: 'a set) ∧ finite (UNIV :: 'b set) ∨ CARD('b) = 1
  (is ?lhs ⟷ ?rhs)
proof –
  have ?lhs ⟷ CARD('a ⇒ 'b) > 0 by(simp add: card-gt-0-iff)

```

```

also have ...  $\longleftrightarrow$   $CARD('a) > 0 \wedge CARD('b) > 0 \vee CARD('b) = 1$ 
  by(simp add: card-fun)
also have ... = ?rhs by(simp add: card-gt-0-iff)
finally show ?thesis .
qed

```

```

lemma card-literal:  $CARD(String.literal) = 0$ 
by(simp add: card-eq-0-iff infinite-literal)

```

9.3 Classes with at least 1 and 2

Class *finite* already captures "at least 1"

```

lemma zero-less-card-finite [simp]:  $0 < CARD('a::finite)$ 
  unfolding neq0-conv [symmetric] by simp

```

```

lemma one-le-card-finite [simp]:  $Suc\ 0 \leq CARD('a::finite)$ 
  by (simp add: less-Suc-eq-le [symmetric])

```

```

class CARD-1 =
  assumes CARD-1:  $CARD\ ('a) = 1$ 
begin

```

```

  subclass finite
  proof
    from CARD-1 show finite (UNIV :: 'a set)
    using finite-UNIV-fun by fastforce
  qed

```

end

Class for cardinality "at least 2"

```

class card2 = finite +
  assumes two-le-card:  $2 \leq CARD('a)$ 

```

```

lemma one-less-card:  $Suc\ 0 < CARD('a::card2)$ 
  using two-le-card [where 'a='a] by simp

```

```

lemma one-less-int-card:  $1 < int\ CARD('a::card2)$ 
  using one-less-card [where 'a='a] by simp

```

9.4 A type class for deciding finiteness of types

```

type-synonym 'a finite-UNIV = ('a, bool) phantom

```

```

class finite-UNIV =
  fixes finite-UNIV :: ('a, bool) phantom
  assumes finite-UNIV: finite-UNIV = Phantom('a) (finite (UNIV :: 'a set))

```

```

lemma finite-UNIV-code [code-unfold]:
  finite (UNIV :: 'a :: finite-UNIV set)
   $\longleftrightarrow$  of-phantom (finite-UNIV :: 'a finite-UNIV)
by(simp add: finite-UNIV)

```

9.5 A type class for computing the cardinality of types

```

definition is-list-UNIV :: 'a list  $\Rightarrow$  bool
where is-list-UNIV xs = (let c = CARD('a) in if c = 0 then False else size
  (remdups xs) = c)

```

```

lemma is-list-UNIV-iff: is-list-UNIV xs  $\longleftrightarrow$  set xs = UNIV
by(auto simp add: is-list-UNIV-def Let-def card-eq-0-iff List.card-set[symmetric]
  dest: subst[where P=finite, OF - finite-set] card-eq-UNIV-imp-eq-UNIV)

```

```

type-synonym 'a card-UNIV = ('a, nat) phantom

```

```

class card-UNIV = finite-UNIV +
  fixes card-UNIV :: 'a card-UNIV
  assumes card-UNIV: card-UNIV = Phantom('a) CARD('a)

```

9.6 Instantiations for *card-UNIV*

```

instantiation nat :: card-UNIV begin
definition finite-UNIV = Phantom(nat) False
definition card-UNIV = Phantom(nat) 0
instance by intro-classes (simp-all add: finite-UNIV-nat-def card-UNIV-nat-def)
end

```

```

instantiation int :: card-UNIV begin
definition finite-UNIV = Phantom(int) False
definition card-UNIV = Phantom(int) 0
instance by intro-classes (simp-all add: card-UNIV-int-def finite-UNIV-int-def)
end

```

```

instantiation natural :: card-UNIV begin
definition finite-UNIV = Phantom(natural) False
definition card-UNIV = Phantom(natural) 0
instance
  by standard
  (auto simp add: finite-UNIV-natural-def card-UNIV-natural-def card-eq-0-iff
    type-definition.univ [OF type-definition-natural] natural-eq-iff
    dest!: finite-imageD intro: inj-onI)
end

```

```

instantiation integer :: card-UNIV begin
definition finite-UNIV = Phantom(integer) False
definition card-UNIV = Phantom(integer) 0
instance
  by standard

```

```

    (auto simp add: finite-UNIV-integer-def card-UNIV-integer-def card-eq-0-iff
      type-definition.univ [OF type-definition-integer]
      dest!: finite-imageD intro: inj-onI)
  end

instantiation list :: (type) card-UNIV begin
definition finite-UNIV = Phantom('a list) False
definition card-UNIV = Phantom('a list) 0
instance by intro-classes (simp-all add: card-UNIV-list-def finite-UNIV-list-def
  infinite-UNIV-listI)
end

instantiation unit :: card-UNIV begin
definition finite-UNIV = Phantom(unit) True
definition card-UNIV = Phantom(unit) 1
instance by intro-classes (simp-all add: card-UNIV-unit-def finite-UNIV-unit-def)
end

instantiation bool :: card-UNIV begin
definition finite-UNIV = Phantom(bool) True
definition card-UNIV = Phantom(bool) 2
instance by (intro-classes)(simp-all add: card-UNIV-bool-def finite-UNIV-bool-def)
end

instantiation char :: card-UNIV begin
definition finite-UNIV = Phantom(char) True
definition card-UNIV = Phantom(char) 256
instance by intro-classes (simp-all add: card-UNIV-char-def card-UNIV-char fi-
  nite-UNIV-char-def)
end

instantiation prod :: (finite-UNIV, finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a  $\times$  'b)
  (of-phantom (finite-UNIV :: 'a finite-UNIV)  $\wedge$  of-phantom (finite-UNIV :: 'b
  finite-UNIV))
instance by intro-classes (simp add: finite-UNIV-prod-def finite-UNIV finite-prod)
end

instantiation prod :: (card-UNIV, card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a  $\times$  'b)
  (of-phantom (card-UNIV :: 'a card-UNIV) * of-phantom (card-UNIV :: 'b card-UNIV))
instance by intro-classes (simp add: card-UNIV-prod-def card-UNIV)
end

instantiation sum :: (finite-UNIV, finite-UNIV) finite-UNIV begin
definition finite-UNIV = Phantom('a + 'b)
  (of-phantom (finite-UNIV :: 'a finite-UNIV)  $\wedge$  of-phantom (finite-UNIV :: 'b
  finite-UNIV))
instance

```


by *intro-classes* (*simp add: finite-UNIV-sum-def finite-UNIV*)
end

instantiation *sum* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('a + 'b)

(*let* *ca* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*);

cb = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)

in if *ca* \neq 0 \wedge *cb* \neq 0 *then* *ca* + *cb* *else* 0)

instance **by** *intro-classes* (*auto simp add: card-UNIV-sum-def card-UNIV card-UNIV-sum*)

end

instantiation *fun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**

definition *finite-UNIV* = *Phantom*('a \Rightarrow 'b)

(*let* *cb* = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)

in *cb* = 1 \vee *of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*) \wedge *cb* \neq 0)

instance

by *intro-classes* (*auto simp add: finite-UNIV-fun-def Let-def card-UNIV finite-UNIV finite-UNIV-fun card-gt-0-iff*)

end

instantiation *fun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('a \Rightarrow 'b)

(*let* *ca* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*);

cb = *of-phantom* (*card-UNIV* :: 'b *card-UNIV*)

in if *ca* \neq 0 \wedge *cb* \neq 0 \vee *cb* = 1 *then* *cb* \wedge *ca* *else* 0)

instance **by** *intro-classes* (*simp add: card-UNIV-fun-def card-UNIV Let-def card-fun*)

end

instantiation *option* :: (*finite-UNIV*) *finite-UNIV* **begin**

definition *finite-UNIV* = *Phantom*('a *option*) (*of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*))

instance **by** *intro-classes* (*simp add: finite-UNIV-option-def finite-UNIV*)

end

instantiation *option* :: (*card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('a *option*)

(*let* *c* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*) *in if* *c* \neq 0 *then* *Suc* *c* *else* 0)

instance **by** *intro-classes* (*simp add: card-UNIV-option-def card-UNIV card-UNIV-option*)

end

instantiation *String.literal* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*String.literal*) *False*

definition *card-UNIV* = *Phantom*(*String.literal*) 0

instance

by *intro-classes* (*simp-all add: card-UNIV-literal-def finite-UNIV-literal-def finite-literal card-literal*)

end

instantiation *set* :: (*finite-UNIV*) *finite-UNIV* **begin**

definition *finite-UNIV* = *Phantom*('a set) (*of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*))

instance by *intro-classes* (*simp add: finite-UNIV-set-def finite-UNIV Finite-Set.finite-set*)
end

instantiation *set* :: (*card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('a set)

(*let* *c* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*) *in if* *c* = 0 *then* 0 *else* 2 [^] *c*)

instance by *intro-classes* (*simp add: card-UNIV-set-def card-UNIV-set card-UNIV*)
end

lemma *UNIV-finite-1*: *UNIV* = *set* [*finite-1.a*₁]

by(*auto intro: finite-1.exhaust*)

lemma *UNIV-finite-2*: *UNIV* = *set* [*finite-2.a*₁, *finite-2.a*₂]

by(*auto intro: finite-2.exhaust*)

lemma *UNIV-finite-3*: *UNIV* = *set* [*finite-3.a*₁, *finite-3.a*₂, *finite-3.a*₃]

by(*auto intro: finite-3.exhaust*)

lemma *UNIV-finite-4*: *UNIV* = *set* [*finite-4.a*₁, *finite-4.a*₂, *finite-4.a*₃, *finite-4.a*₄]

by(*auto intro: finite-4.exhaust*)

lemma *UNIV-finite-5*:

UNIV = *set* [*finite-5.a*₁, *finite-5.a*₂, *finite-5.a*₃, *finite-5.a*₄, *finite-5.a*₅]

by(*auto intro: finite-5.exhaust*)

instantiation *Enum.finite-1* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-1*) 1

instance

by *intro-classes* (*simp-all add: UNIV-finite-1 card-UNIV-finite-1-def finite-UNIV-finite-1-def*)

end

instantiation *Enum.finite-2* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-2*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-2*) 2

instance

by *intro-classes* (*simp-all add: UNIV-finite-2 card-UNIV-finite-2-def finite-UNIV-finite-2-def*)

end

instantiation *Enum.finite-3* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-3*) *True*

definition *card-UNIV* = *Phantom*(*Enum.finite-3*) 3

instance

by *intro-classes* (*simp-all add: UNIV-finite-3 card-UNIV-finite-3-def finite-UNIV-finite-3-def*)

end

instantiation *Enum.finite-4* :: *card-UNIV* **begin**

```

definition finite-UNIV = Phantom(Enum.finite-4) True
definition card-UNIV = Phantom(Enum.finite-4) 4
instance
  by intro-classes (simp-all add: UNIV-finite-4 card-UNIV-finite-4-def finite-UNIV-finite-4-def)
end

instantiation Enum.finite-5 :: card-UNIV begin
definition finite-UNIV = Phantom(Enum.finite-5) True
definition card-UNIV = Phantom(Enum.finite-5) 5
instance
  by intro-classes (simp-all add: UNIV-finite-5 card-UNIV-finite-5-def finite-UNIV-finite-5-def)
end

end

```

10 Code setup for sets with cardinality type information

theory *Code-Cardinality* **imports** *Cardinality* **begin**

Implement $CARD('a)$ via *card-UNIV-class.card-UNIV* and provide implementations for *finite*, *card*, (\subseteq) , and $(=)$ if the calling context already provides *finite-UNIV* and *card-UNIV* instances. If we implemented the latter always via *card-UNIV-class.card-UNIV*, we would require instances of essentially all element types, i.e., a lot of instantiation proofs and – at run time – possibly slow dictionary constructions.

context
begin

qualified definition *card-UNIV'* :: '*a* *card-UNIV*
where *card-UNIV'* = *Phantom*('a) *CARD*('a)

lemma *CARD-code* [*code-unfold*]:
 $CARD('a) = of_phantom\ (card-UNIV' :: 'a\ card-UNIV)$
by(*simp add: card-UNIV'-def*)

lemma *card-UNIV'-code* [*code*]:
 $card-UNIV' = card-UNIV$
by(*simp add: card-UNIV card-UNIV'-def*)

end

lemma *card-Compl*:
 $finite\ A \implies card\ (-\ A) = card\ (UNIV :: 'a\ set) - card\ (A :: 'a\ set)$
by (*metis Compl-eq-Diff-UNIV card-Diff-subset top-greatest*)

context **fixes** *xs* :: '*a* :: *finite-UNIV* *list*
begin

qualified definition $finite' :: 'a \text{ set} \Rightarrow bool$

where $[simp, code-abbrev]: finite' = finite$

lemma $finite'-code [code]:$

$finite' (set\ xs) \longleftrightarrow True$

$finite' (List.coset\ xs) \longleftrightarrow of-phantom\ (finite-UNIV :: 'a\ finite-UNIV)$

by($simp-all\ add: card-gt-0-iff\ finite-UNIV$)

end

context fixes $xs :: 'a :: card-UNIV\ list$

begin

qualified definition $card' :: 'a \text{ set} \Rightarrow nat$

where $[simp, code-abbrev]: card' = card$

lemma $card'-code [code]:$

$card' (set\ xs) = length\ (remdups\ xs)$

$card' (List.coset\ xs) = of-phantom\ (card-UNIV :: 'a\ card-UNIV) - length\ (remdups\ xs)$

by($simp-all\ add: List.card-set\ card-Compl\ card-UNIV$)

qualified definition $subset' :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow bool$

where $[simp, code-abbrev]: subset' = (\subseteq)$

lemma $subset'-code [code]:$

$subset'\ A\ (List.coset\ ys) \longleftrightarrow (\forall y \in set\ ys. y \notin A)$

$subset'\ (set\ ys)\ B \longleftrightarrow (\forall y \in set\ ys. y \in B)$

$subset'\ (List.coset\ xs)\ (set\ ys) \longleftrightarrow (let\ n = CARD('a)\ in\ n > 0 \wedge card(set\ (xs @ ys)) = n)$

by($auto\ simp\ add: Let-def\ card-gt-0-iff\ dest: card-eq-UNIV-imp-eq-UNIV\ intro: arg-cong[where\ f=card]$)

($metis\ finite-compl\ finite-set\ rev-finite-subset$)

qualified definition $eq-set :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow bool$

where $[simp, code-abbrev]: eq-set = (=)$

lemma $eq-set-code [code]:$

fixes ys

defines $rhs \equiv$

$let\ n = CARD('a)$

$in\ if\ n = 0\ then\ False\ else$

$let\ xs' = remdups\ xs; ys' = remdups\ ys$

$in\ length\ xs' + length\ ys' = n \wedge (\forall x \in set\ xs'. x \notin set\ ys') \wedge (\forall y \in set\ ys'.$

$y \notin set\ xs')$

shows $eq-set\ (List.coset\ xs)\ (set\ ys) \longleftrightarrow rhs$

and $eq-set\ (set\ ys)\ (List.coset\ xs) \longleftrightarrow rhs$

```

and eq-set (set xs) (set ys)  $\longleftrightarrow$  ( $\forall x \in \text{set } xs. x \in \text{set } ys$ )  $\wedge$  ( $\forall y \in \text{set } ys. y \in \text{set } xs$ )
and eq-set (List.coset xs) (List.coset ys)  $\longleftrightarrow$  ( $\forall x \in \text{set } xs. x \in \text{set } ys$ )  $\wedge$  ( $\forall y \in \text{set } ys. y \in \text{set } xs$ )
proof goal-cases
{
  case 1
  show ?case (is ?lhs  $\longleftrightarrow$  ?rhs)
  proof
    show ?rhs if ?lhs
    using that
    by (auto simp add: rhs-def Let-def List.card-set[symmetric]
      card-Un-Int[where A=set xs and B=— set xs] card-UNIV
      Compl-partition card-gt-0-iff dest: sym)(metis finite-compl finite-set)
    show ?lhs if ?rhs
    proof —
    have  $\llbracket \forall y \in \text{set } xs. y \notin \text{set } ys; \forall x \in \text{set } ys. x \notin \text{set } xs \rrbracket \implies \text{set } xs \cap \text{set } ys = \{\}$ 
by blast
    with that show ?thesis
    by (auto simp add: rhs-def Let-def List.card-set[symmetric]
      card-UNIV card-gt-0-iff card-Un-Int[where A=set xs and B=set ys]
      dest: card-eq-UNIV-imp-eq-UNIV split: if-split-asm)
    qed
  qed
}
moreover
case 2
ultimately show ?case unfolding eq-set-def by blast
next
case 3
show ?case unfolding eq-set-def List.coset-def by blast
next
case 4
show ?case unfolding eq-set-def List.coset-def by blast
qed

end

```

Provide more informative exceptions than Match for non-rewritten cases. If generated code raises one these exceptions, then a code equation calls the mentioned operator for an element type that is not an instance of *card-UNIV* and is therefore not implemented via *card-UNIV-class.card-UNIV*. Constrain the element type with sort *card-UNIV* to change this.

lemma *card-code* [code]:

card (set xs) = length (remdups xs)

card (List.coset xs) =

Code.abort (STR "card (List.coset -) requires type class instance card-UNIV")

($\lambda \cdot$. *card* (List.coset xs))

by (simp-all add: length-remdups-card-conv)

```

lemma coset-subseteq-set-code [code]:
  set  $xs \subseteq B = \text{list-all } (\lambda x. x \in B) \ xs$ 
   $A \subseteq \text{List.coset } ys = \text{list-all } (\lambda y. y \notin A) \ ys$ 
   $\text{List.coset } xs \subseteq \text{set } ys \longleftrightarrow$ 
  (if  $xs = [] \wedge ys = []$  then False
   else Code.abort
   (STR "subset-eq (List.coset -) (List.set -) requires type class instance card-UNIV")
   ( $\lambda -. \text{List.coset } xs \subseteq \text{set } ys$ ))
  by (auto simp add: list-all-iff)

notepad begin — test code setup
have  $\text{List.coset } [\text{True}] = \text{set } [\text{False}] \wedge$ 
   $\text{List.coset } [] \subseteq \text{List.set } [\text{True}, \text{False}] \wedge$ 
  finite ( $\text{List.coset } [\text{True}]$ )
  by eval

end

end

```

11 Eliminating pattern matches

```

theory Case-Converter
  imports Main
begin

definition missing-pattern-match :: String.literal  $\Rightarrow$  (unit  $\Rightarrow$  'a)  $\Rightarrow$  'a where
  [code del]: missing-pattern-match m f = f ()

lemma missing-pattern-match-cong [cong]:
   $m = m' \implies \text{missing-pattern-match } m \ f = \text{missing-pattern-match } m' \ f$ 
  by(rule arg-cong)

lemma missing-pattern-match-code [code-unfold]:
  missing-pattern-match = Code.abort
  unfolding missing-pattern-match-def Code.abort-def ..

ML-file <case-converter.ML>

end

```

12 Lazy types in generated code

```

theory Code-Lazy
imports Case-Converter
keywords
  code-lazy-type

```

```

activate-lazy-type
deactivate-lazy-type
activate-lazy-types
deactivate-lazy-types
print-lazy-types :: thy-decl
begin

```

This theory and the CodeLazy tool described in [3].

It hooks into Isabelle’s code generator such that the generated code evaluates a user-specified set of type constructors lazily, even in target languages with eager evaluation. The lazy type must be algebraic, i.e., values must be built from constructors and a corresponding case operator decomposes them. Every datatype and codatatype is algebraic and thus eligible for lazification.

12.1 The type *lazy*

```

typedef 'a lazy = UNIV :: 'a set ..
setup-lifting type-definition-lazy
lift-definition delay :: (unit  $\Rightarrow$  'a)  $\Rightarrow$  'a lazy is  $\lambda f. f ()$  .
lift-definition force :: 'a lazy  $\Rightarrow$  'a is  $\lambda x. x$  .

```

```

code-datatype delay
lemma force-delay [code]: force (delay f) = f () by transfer (rule refl)
lemma delay-force: delay ( $\lambda \cdot$ . force s) = s by transfer (rule refl)

```

```

definition termify-lazy2 :: 'a :: typerep lazy  $\Rightarrow$  term
  where termify-lazy2 x =
    Code-Evaluation.App (Code-Evaluation.Const (STR "Code-Lazy.delay") (TYPEREP((unit
 $\Rightarrow$  'a)  $\Rightarrow$  'a lazy)))
    (Code-Evaluation.Const (STR "Pure.dummy-pattern") (TYPEREP((unit  $\Rightarrow$ 
'a))))

```

```

definition termify-lazy ::
  (String.literal  $\Rightarrow$  'typerep  $\Rightarrow$  'term)  $\Rightarrow$ 
  ('term  $\Rightarrow$  'term  $\Rightarrow$  'term)  $\Rightarrow$ 
  (String.literal  $\Rightarrow$  'typerep  $\Rightarrow$  'term  $\Rightarrow$  'term)  $\Rightarrow$ 
  'typerep  $\Rightarrow$  ('typerep  $\Rightarrow$  'typerep  $\Rightarrow$  'typerep)  $\Rightarrow$  ('typerep  $\Rightarrow$  'typerep)  $\Rightarrow$ 
  ('a  $\Rightarrow$  'term)  $\Rightarrow$  'typerep  $\Rightarrow$  'a :: typerep lazy  $\Rightarrow$  'term  $\Rightarrow$  term
  where termify-lazy - - - - - x = termify-lazy2 x

```

```

declare [[code drop: Code-Evaluation.term-of :: - lazy  $\Rightarrow$  -]]

```

```

lemma term-of-lazy-code [code]:
  Code-Evaluation.term-of x  $\equiv$ 
    termify-lazy
      Code-Evaluation.Const Code-Evaluation.App Code-Evaluation.Abs
        TYPEREP(unit) ( $\lambda T U. typerep.Typerep (STR "fun") [T, U]$ ) ( $\lambda T. typerep.Typerep (STR "Code-Lazy.lazy") [T]$ )

```

```

    Code-Evaluation.term-of TYPEREP('a) x (Code-Evaluation.Const (STR ""))
  (TYPEREP(unit)))
  for x :: 'a :: {typerep, term-of} lazy
  by (rule term-of-anything)

```

The implementations of - lazy using language primitives cache forced values.

Term reconstruction for lazy looks into the lazy value and reconstructs it to the depth it has been evaluated. This is not done for Haskell as we do not know of any portable way to inspect whether a lazy value has been evaluated to or not.

```

code-printing code-module Lazy  $\rightarrow$  (SML) file ~/src/HOL/Library/Tools/lazy.ML
  for type-constructor lazy constant delay force termify-lazy
| type-constructor lazy  $\rightarrow$  (SML) - Lazy.lazy
| constant delay  $\rightarrow$  (SML) Lazy.lazy
| constant force  $\rightarrow$  (SML) Lazy.force
| constant termify-lazy  $\rightarrow$  (SML) Lazy.termify'-lazy

```

code-reserved (SML) Lazy

code-printing — For code generation within the Isabelle environment, we reuse the thread-safe implementation of lazy from ~/src/Pure/Concurrent/lazy.ML

```

  code-module Lazy  $\rightarrow$  (Eval)  $\hookrightarrow$  for constant undefined
| type-constructor lazy  $\rightarrow$  (Eval) - Lazy.lazy
| constant delay  $\rightarrow$  (Eval) Lazy.lazy
| constant force  $\rightarrow$  (Eval) Lazy.force
| code-module Termify-Lazy  $\rightarrow$  (Eval) file ~/src/HOL/Library/Tools/termify-lazy.ML
  for constant termify-lazy
| constant termify-lazy  $\rightarrow$  (Eval) Termify'-Lazy.termify'-lazy

```

code-reserved (Eval) Termify-Lazy

code-printing

```

  type-constructor lazy  $\rightarrow$  (OCaml) - Lazy.t
| constant delay  $\rightarrow$  (OCaml) Lazy.from'-fun
| constant force  $\rightarrow$  (OCaml) Lazy.force
| code-module Termify-Lazy  $\rightarrow$  (OCaml) file ~/src/HOL/Library/Tools/termify-lazy.ocaml
  for constant termify-lazy
| constant termify-lazy  $\rightarrow$  (OCaml) Termify'-Lazy.termify'-lazy

```

code-reserved (OCaml) Lazy Termify-Lazy

code-printing

```

  code-module Lazy  $\rightarrow$  (Haskell) file ~/src/HOL/Library/Tools/lazy.hs
  for type-constructor lazy constant delay force
| type-constructor lazy  $\rightarrow$  (Haskell) Lazy.Lazy -
| constant delay  $\rightarrow$  (Haskell) Lazy.delay
| constant force  $\rightarrow$  (Haskell) Lazy.force

```


code-reserved (*Haskell*) *Lazy*

code-printing

```
code-module Lazy  $\rightarrow$  (Scala) file  $\sim\sim$ /src/HOL/Library/Tools/lazy.scala
  for type-constructor lazy constant delay force termify-lazy
| type-constructor lazy  $\rightarrow$  (Scala) Lazy.Lazy[-]
| constant delay  $\rightarrow$  (Scala) Lazy.delay
| constant force  $\rightarrow$  (Scala) Lazy.force
| constant termify-lazy  $\rightarrow$  (Scala) Lazy.termify'-lazy
```

code-reserved (*Scala*) *Lazy*

Make evaluation with the simplifier respect *delays*.

```
lemma delay-lazy-cong: delay f = delay f by simp
setup  $\langle$  Code-Simp.map-ss (Simplifier.add-cong @{thm delay-lazy-cong})  $\rangle$ 
```

12.2 Implementation

ML-file \langle *code-lazy.ML* \rangle

```
setup  $\langle$ 
  Code-Preproc.add-functrans (lazy-datatype, Code-Lazy.transform-code-eqs)
 $\rangle$ 
```

end

13 Test infrastructure for the code generator

```
theory Code-Test
imports Main
keywords test-code :: diag
begin
```

13.1 YXML encoding for *term*

datatype (*plugins* *del*: *code* *size* *quickcheck*) *yxml-of-term* = *YXML*

```
lemma yot-anything: x = (y :: yxml-of-term)
by(cases x y rule: yxml-of-term.exhaust[case-product yxml-of-term.exhaust])(simp)
```

definition *yot-empty* :: *yxml-of-term* **where** [*code* *del*]: *yot-empty* = *YXML*

definition *yot-literal* :: *String.literal* \Rightarrow *yxml-of-term*

where [*code* *del*]: *yot-literal* - = *YXML*

definition *yot-append* :: *yxml-of-term* \Rightarrow *yxml-of-term* \Rightarrow *yxml-of-term*

where [*code* *del*]: *yot-append* - - = *YXML*

definition *yot-concat* :: *yxml-of-term* *list* \Rightarrow *yxml-of-term*

where [*code* *del*]: *yot-concat* - = *YXML*

Serialise *yxml-of-term* to native string of target language

code-printing type-constructor *yxml-of-term*

```

  ↪ (SML) string
  and (OCaml) string
  and (Haskell) String
  and (Scala) String
| constant yot-empty
  ↪ (SML)
  and (OCaml)
  and (Haskell)
  and (Scala)
| constant yot-literal
  ↪ (SML) -
  and (OCaml) -
  and (Haskell) -
  and (Scala) -
| constant yot-append
  ↪ (SML) String.concat [(-), (-)]
  and (OCaml) String.concat [(-); (-)]
  and (Haskell) infixr 5 ++
  and (Scala) infixl 5 +
| constant yot-concat
  ↪ (SML) String.concat
  and (OCaml) String.concat
  and (Haskell) Prelude.concat
  and (Scala) -.mkString()

```

Stripped-down implementations of Isabelle’s XML tree with YXML encoding as defined in `~/src/Pure/PIDE/xml.ML`, `~/src/Pure/PIDE/yxml.ML` sufficient to encode *term* as in `~/src/Pure/term_xml.ML`.

datatype (*plugins del: code size quickcheck*) *xml-tree* = *XML-Tree*

lemma *xml-tree-anything*: $x = (y :: \text{xml-tree})$

by(*cases* x *y* *rule: xml-tree.exhaust[case-product xml-tree.exhaust]*)(*simp*)

context begin

local-setup $\langle \text{Local-Theory.map-background-naming } (\text{Name-Space.mandatory-path } \text{xml}) \rangle$

type-synonym *attributes* = $(\text{String.literal} \times \text{String.literal}) \text{ list}$

type-synonym *body* = *xml-tree list*

definition *Elem* :: $\text{String.literal} \Rightarrow \text{attributes} \Rightarrow \text{xml-tree list} \Rightarrow \text{xml-tree}$

where [*code del*]: *Elem* - - - = *XML-Tree*

definition *Text* :: $\text{String.literal} \Rightarrow \text{xml-tree}$

where [*code del*]: *Text* - = *XML-Tree*

definition *node* :: $\text{xml-tree list} \Rightarrow \text{xml-tree}$

where *node* *ts* = *Elem* (*STR* *""*) [] *ts*

definition *tagged* :: *String.literal* \Rightarrow *String.literal option* \Rightarrow *xml-tree list* \Rightarrow *xml-tree*
where *tagged tag x ts* = *Elem tag* (case *x* of *None* \Rightarrow [] | *Some x'* \Rightarrow [(*STR "0"*, *x'*)] *ts*)

definition *list* **where** *list f xs* = *map* (*node* \circ *f*) *xs*

definition *X* :: *yaml-of-term* **where** *X* = *yot-literal* (*STR 0x05*)

definition *Y* :: *yaml-of-term* **where** *Y* = *yot-literal* (*STR 0x06*)

definition *XY* :: *yaml-of-term* **where** *XY* = *yot-append* *X Y*

definition *XYX* :: *yaml-of-term* **where** *XYX* = *yot-append* *XY X*

end

code-datatype *xml.Elem xml.Text*

definition *yaml-string-of-xml-tree* :: *xml-tree* \Rightarrow *yaml-of-term* \Rightarrow *yaml-of-term*
where [*code del*]: *yaml-string-of-xml-tree - -* = *YXML*

lemma *yaml-string-of-xml-tree-code* [*code*]:

yaml-string-of-xml-tree (*xml.Elem name atts ts*) *rest* =
yot-append *xml.XY* (
yot-append (*yot-literal name*) (
foldr ($\lambda(a, x)$ *rest*.
yot-append *xml.Y* (
yot-append (*yot-literal a*) (
yot-append (*yot-literal (STR "'')*) (
yot-append (*yot-literal x*) *rest*)))) *atts* (
foldr *yaml-string-of-xml-tree* *ts* (
yot-append *xml.XYX* *rest*))))
yaml-string-of-xml-tree (*xml.Text s*) *rest* = *yot-append* (*yot-literal s*) *rest*
by(*rule yot-anything*)+

definition *yaml-string-of-body* :: *xml.body* \Rightarrow *yaml-of-term*

where *yaml-string-of-body ts* = *foldr* *yaml-string-of-xml-tree* *ts* *yot-empty*

Encoding *term* into XML trees as defined in `~/src/Pure/term_xml.ML`.

definition *xml-of-typ* :: *Typerep.typerep* \Rightarrow *xml.body*

where [*code del*]: *xml-of-typ -* = [*XML-Tree*]

definition *xml-of-term* :: *Code-Evaluation.term* \Rightarrow *xml.body*

where [*code del*]: *xml-of-term -* = [*XML-Tree*]

lemma *xml-of-typ-code* [*code*]:

xml-of-typ (*typerep.Type t args*) = [*xml.tagged* (*STR "0"*) (*Some t*) (*xml.list* *xml-of-typ* *args*)]
by(*simp add: xml-of-typ-def xml-tree-anything*)

lemma *xml-of-term-code* [*code*]:

```

xml-of-term (Code-Evaluation.Const x ty) = [xml.tagged (STR "0") (Some x)
(xml-of-ty ty)]
xml-of-term (Code-Evaluation.App t1 t2) = [xml.tagged (STR "5") None [xml.node
(xml-of-term t1), xml.node (xml-of-term t2)]]
xml-of-term (Code-Evaluation.Abs x ty t) = [xml.tagged (STR "4") (Some x)
[xml.node (xml-of-ty ty), xml.node (xml-of-term t)]]
— FIXME: Code-Evaluation.Free is used only in HOL.Quickcheck-Narrowing to
represent uninstantiated parameters in constructors. Here, we always translate
them to Free variables.
xml-of-term (Code-Evaluation.Free x ty) = [xml.tagged (STR "1") (Some x)
(xml-of-ty ty)]
by (simp-all add: xml-of-term-def xml-tree-anything)

```

definition *yaml-string-of-term* :: *Code-Evaluation.term* \Rightarrow *yaml-of-term*
where *yaml-string-of-term* = *yaml-string-of-body* \circ *xml-of-term*

13.2 Test engine and drivers

ML-file *<code-test.ML>*

end

14 A combinator to build partial equivalence relations from a predicate and an equivalence relation

theory *Combine-PER*

imports *Main*

begin

unbundle *lattice-syntax*

definition *combine-per* :: (*'a* \Rightarrow *bool*) \Rightarrow (*'a* \Rightarrow *'a* \Rightarrow *bool*) \Rightarrow *'a* \Rightarrow *'a* \Rightarrow *bool*
where *combine-per* *P R* = ($\lambda x y. P x \wedge P y$) \sqcap *R*

lemma *combine-per-simp* [*simp*]:
 $combine-per\ P\ R\ x\ y \longleftrightarrow P\ x \wedge P\ y \wedge x \approx y$ **for** *R* (**infixl** $\langle \approx \rangle$ 50)
by (*simp add: combine-per-def*)

lemma *combine-per-top* [*simp*]: $combine-per\ \top\ R = R$
by (*simp add: fun-eq-iff*)

lemma *combine-per-eq* [*simp*]: $combine-per\ P\ HOL.eq = HOL.eq \sqcap (\lambda x y. P\ x)$
by (*auto simp add: fun-eq-iff*)

lemma *symp-combine-per*: $symp\ R \Longrightarrow symp\ (combine-per\ P\ R)$
by (*auto simp add: symp-def sym-def combine-per-def*)

```

lemma transp-combine-per: transp R  $\implies$  transp (combine-per P R)
  by (auto simp add: transp-def trans-def combine-per-def)

lemma combine-perI: P x  $\implies$  P y  $\implies$  x  $\approx$  y  $\implies$  combine-per P R x y for R
(infixl  $\langle \approx \rangle$  50)
  by (simp add: combine-per-def)

lemma symp-combine-per-symp: symp R  $\implies$  symp (combine-per P R)
  by (auto intro!: sympI elim: sympE)

lemma transp-combine-per-transp: transp R  $\implies$  transp (combine-per P R)
  by (auto intro!: transpI elim: transpE)

lemma equivp-combine-per-part-equivp [intro?]:
  fixes R (infixl  $\langle \approx \rangle$  50)
  assumes  $\exists x. P\ x$  and equivp R
  shows part-equivp (combine-per P R)
proof –
  from  $\langle \exists x. P\ x \rangle$  obtain x where P x ..
  moreover from  $\langle \text{equivp } R \rangle$  have x  $\approx$  x
    by (rule equivp-reflp)
  ultimately have  $\exists x. P\ x \wedge x \approx x$ 
    by blast
  with  $\langle \text{equivp } R \rangle$  show ?thesis
    by (auto intro!: part-equivpI symp-combine-per-symp transp-combine-per-transp
      elim: equivpE)
qed

end

```

15 Formalisation of chain-complete partial orders, continuity and admissibility

```

theory Complete-Partial-Order2
  imports Main
begin

unbundle lattice-syntax

lemma chain-transfer [transfer-rule]:
  includes lifting-syntax
  shows  $((A \implies A \implies (=)) \implies \text{rel-set } A \implies (=))$  Complete-Partial-Order.chain
Complete-Partial-Order.chain
  unfolding chain-def[abs-def] by transfer-prover

lemma linorder-chain [simp, intro!]:
  fixes Y :: - :: linorder set
  shows Complete-Partial-Order.chain  $(\leq)$  Y

```

by(*auto intro: chainI*)

lemma *fun-lub-apply*: $\bigwedge \text{Sup. fun-lub Sup } Y \ x = \text{Sup } ((\lambda f. f \ x) \ ' \ Y)$
by(*simp add: fun-lub-def image-def*)

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* $\{\}$ = $(\lambda -. \text{lub } \{\})$
by(*rule ext*)(*simp add: fun-lub-apply*)

lemma *chain-fun-ordD*:
assumes *Complete-Partial-Order.chain* (*fun-ord le*) *Y*
shows *Complete-Partial-Order.chain le* $((\lambda f. f \ x) \ ' \ Y)$
by(*rule chainI*)(*auto dest: chainD[OF assms] simp add: fun-ord-def*)

lemma *chain-Diff*:
Complete-Partial-Order.chain ord A
 \implies *Complete-Partial-Order.chain ord* (*A* – *B*)
by(*erule chain-subset*) *blast*

lemma *chain-rel-prodD1*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
 \implies *Complete-Partial-Order.chain orda* (*fst* ‘ *Y*)
by(*auto 4 3 simp add: chain-def*)

lemma *chain-rel-prodD2*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
 \implies *Complete-Partial-Order.chain ordb* (*snd* ‘ *Y*)
by(*auto 4 3 simp add: chain-def*)

context *ccpo begin*

lemma *ccpo-fun*: *class.ccpo* (*fun-lub Sup*) (*fun-ord* (\leq)) (*mk-less* (*fun-ord* (\leq)))
by *standard* (*auto 4 3 simp add: mk-less-def fun-ord-def fun-lub-apply*
intro: order.trans order.antisym chain-imageI ccpo-Sup-upper ccpo-Sup-least)

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain* (\leq) *Y* \implies *Sup Y* \leq *x*
 \longleftrightarrow $(\forall y \in Y. y \leq x)$
by (*meson local.ccpo-Sup-least local.ccpo-Sup-upper local.dual-order.trans*)

lemma *Sup-minus-bot*:
assumes *chain: Complete-Partial-Order.chain* (\leq) *A*
shows $\bigsqcup (A - \{\bigsqcup \{\}\}) = \bigsqcup A$
(is ?lhs = ?rhs)
proof (*rule order.antisym*)
show *?lhs* \leq *?rhs*
by (*blast intro: ccpo-Sup-least chain-Diff[OF chain] ccpo-Sup-upper[OF chain]*)
show *?rhs* \leq *?lhs*
proof (*rule ccpo-Sup-least [OF chain]*)
show $x \in A \implies x \leq$ *?lhs* **for** *x*

```

    by (cases x =  $\sqcup \{\}$ )
      (blast intro: ccpo-Sup-least chain-empty ccpo-Sup-upper[OF chain-Diff[OF
chain]])+
  qed
qed

```

lemma *mono-lub*:

```

  fixes le-b (infix  $\langle \sqsubseteq \rangle$  60)
  assumes chain: Complete-Partial-Order.chain (fun-ord  $(\leq)$ ) Y
  and mono:  $\bigwedge f. f \in Y \implies \text{monotone } le-b (\leq) f$ 
  shows monotone  $(\sqsubseteq) (\leq)$  (fun-lub Sup Y)
proof(rule monotoneI)
  fix x y
  assume x  $\sqsubseteq$  y

  have chain'':  $\bigwedge x. \text{Complete-Partial-Order.chain } (\leq) ((\lambda f. f x) ' Y)$ 
    using chain by(rule chain-imageI)(simp add: fun-ord-def)
  then show fun-lub Sup Y x  $\leq$  fun-lub Sup Y y unfolding fun-lub-apply
proof(rule ccpo-Sup-least)
  fix x'
  assume x'  $\in (\lambda f. f x) ' Y$ 
  then obtain f where f  $\in Y$  x' = f x by blast
  note  $\langle x' = f x \rangle$  also
  from  $\langle f \in Y \rangle \langle x \sqsubseteq y \rangle$  have f x  $\leq$  f y by(blast dest: mono monotoneD)
  also have  $\dots \leq \sqcup ((\lambda f. f y) ' Y)$  using chain''
    by(rule ccpo-Sup-upper)(simp add:  $\langle f \in Y \rangle$ )
  finally show x'  $\leq \sqcup ((\lambda f. f y) ' Y)$  .
qed
qed

```

context

```

  fixes le-b (infix  $\langle \sqsubseteq \rangle$  60) and Y f
  assumes chain: Complete-Partial-Order.chain le-b Y
  and mono1:  $\bigwedge y. y \in Y \implies \text{monotone } le-b (\leq) (\lambda x. f x y)$ 
  and mono2:  $\bigwedge a b. \llbracket x \in Y; a \sqsubseteq b; a \in Y; b \in Y \rrbracket \implies f x a \leq f x b$ 
begin

```

lemma *Sup-mono*:

```

  assumes le: x  $\sqsubseteq$  y and x: x  $\in Y$  and y: y  $\in Y$ 
  shows  $\sqcup (f x ' Y) \leq \sqcup (f y ' Y)$  (is -  $\leq$  ?rhs)
proof(rule ccpo-Sup-least)
  from chain show chain': Complete-Partial-Order.chain  $(\leq)$  (f x ' Y) when x  $\in Y$  for x
    by(rule chain-imageI) (insert that, auto dest: mono2)

  fix x'
  assume x'  $\in f x ' Y$ 
  then obtain y' where y'  $\in Y$  x' = f x y' by blast note this(2)
  also from monoI[OF  $\langle y' \in Y \rangle$ ] le have  $\dots \leq f y y'$  by(rule monotoneD)

```

```

also have ...  $\leq$  ?rhs using chain'[OF y]
by (auto intro!: ccpo-Sup-upper simp add:  $\langle y' \in Y \rangle$ )
finally show  $x' \leq$  ?rhs .
qed(rule x)

lemma diag-Sup:  $\bigsqcup ((\lambda x. \bigsqcup (f x \text{ ' } Y)) \text{ ' } Y) = \bigsqcup ((\lambda x. f x x) \text{ ' } Y)$  (is ?lhs = ?rhs)
proof(rule order.antisym)
  have chain1: Complete-Partial-Order.chain  $(\leq)$   $((\lambda x. \bigsqcup (f x \text{ ' } Y)) \text{ ' } Y)$ 
  using chain by(rule chain-imageI)(rule Sup-mono)
  have chain2:  $\bigwedge y'. y' \in Y \implies \text{Complete-Partial-Order.chain } (\leq) (f y' \text{ ' } Y)$  using
chain
  by(rule chain-imageI)(auto dest: mono2)
  have chain3: Complete-Partial-Order.chain  $(\leq)$   $((\lambda x. f x x) \text{ ' } Y)$ 
  using chain by(rule chain-imageI)(auto intro: monotoneD[OF mono1] mono2
order.trans)

show ?lhs  $\leq$  ?rhs using chain1
proof(rule ccpo-Sup-least)
  fix x'
  assume  $x' \in (\lambda x. \bigsqcup (f x \text{ ' } Y)) \text{ ' } Y$ 
  then obtain y' where  $y' \in Y$   $x' = \bigsqcup (f y' \text{ ' } Y)$  by blast note this(2)
  also have ...  $\leq$  ?rhs using chain2[OF  $\langle y' \in Y \rangle$ ]
  proof(rule ccpo-Sup-least)
    fix x
    assume  $x \in f y' \text{ ' } Y$ 
    then obtain y where  $y \in Y$  and  $x = f y' y$  by blast
    define y'' where  $y'' = (\text{if } y \sqsubseteq y' \text{ then } y' \text{ else } y)$ 
    from chain  $\langle y \in Y \rangle \langle y' \in Y \rangle$  have  $y \sqsubseteq y' \vee y' \sqsubseteq y$  by(rule chainD)
    hence  $f y' y \leq f y'' y''$  using  $\langle y \in Y \rangle \langle y' \in Y \rangle$ 
    by(auto simp add: y''-def intro: mono2 monotoneD[OF mono1])
    also from  $\langle y \in Y \rangle \langle y' \in Y \rangle$  have  $y'' \in Y$  by(simp add: y''-def)
    from chain3 have  $f y'' y'' \leq$  ?rhs by(rule ccpo-Sup-upper)(simp add:  $\langle y'' \in Y \rangle$ )
  finally show  $x \leq$  ?rhs by(simp add: x)
  qed
finally show  $x' \leq$  ?rhs .
qed

show ?rhs  $\leq$  ?lhs using chain3
proof(rule ccpo-Sup-least)
  fix y
  assume  $y \in (\lambda x. f x x) \text{ ' } Y$ 
  then obtain x where  $x \in Y$  and  $y = f x x$  by blast
  then show  $y \leq$  ?lhs
  by (metis (no-types, lifting) chain1 chain2 imageI ccpo-Sup-upper order.trans)
qed
qed
end

```


lemma *Sup-image-mono-le*:

fixes *le-b* (**infix** \sqsubseteq 60) **and** *Sup-b* (\sqcup)
assumes *ccpo*: *class.ccpo* *Sup-b* (\sqsubseteq) *lt-b*
assumes *chain*: *Complete-Partial-Order.chain* (\sqsubseteq) *Y*
and *mono*: $\bigwedge x y. \llbracket x \sqsubseteq y; x \in Y \rrbracket \implies f x \leq f y$
shows $\text{Sup } (f \text{ ` } Y) \leq f (\bigvee Y)$
proof(*rule ccpo-Sup-least*)
show *Complete-Partial-Order.chain* (\leq) (*f* ` *Y*)
using *chain* **by**(*rule chain-imageI*)(*rule mono*)

fix *x*
assume $x \in f \text{ ` } Y$
then obtain *y* **where** $y \in Y$ **and** $x = f y$ **by** *blast* **note** *this*(2)
also have $y \sqsubseteq \bigvee Y$ **using** *ccpo chain* $\langle y \in Y \rangle$ **by**(*rule ccpo.ccpo-Sup-upper*)
hence $f y \leq f (\bigvee Y)$ **using** $\langle y \in Y \rangle$ **by**(*rule mono*)
finally show $x \leq \dots$.
qed

lemma *swap-Sup*:

fixes *le-b* (**infix** \sqsubseteq 60)
assumes *Y*: *Complete-Partial-Order.chain* (\sqsubseteq) *Y*
and *Z*: *Complete-Partial-Order.chain* (*fun-ord* (\leq)) *Z*
and *mono*: $\bigwedge f. f \in Z \implies \text{monotone } (\sqsubseteq) (\leq) f$
shows $\bigsqcup ((\lambda x. \bigsqcup (x \text{ ` } Y)) \text{ ` } Z) = \bigsqcup ((\lambda x. \bigsqcup ((\lambda f. f x) \text{ ` } Z)) \text{ ` } Y)$
(is ?lhs = ?rhs)
proof(*cases Y = {}*)
case *True*
then show *?thesis*
by (*simp add: image-constant-conv cong del: SUP-cong-simp*)
next
case *False*
have *chain1*: $\bigwedge f. f \in Z \implies \text{Complete-Partial-Order.chain } (\leq) (f \text{ ` } Y)$
by(*rule chain-imageI[OF Y]*)(*rule monotoneD[OF mono]*)
have *chain2*: *Complete-Partial-Order.chain* (\leq) $((\lambda x. \bigsqcup (x \text{ ` } Y)) \text{ ` } Z)$ **using** *Z*
proof(*rule chain-imageI*)
fix *f g*
assume $f \in Z \ g \in Z$
and *fun-ord* (\leq) *f g*
from *chain1*[*OF* $\langle f \in Z \rangle$] **show** $\bigsqcup (f \text{ ` } Y) \leq \bigsqcup (g \text{ ` } Y)$
proof(*rule ccpo-Sup-least*)
fix *x*
assume $x \in f \text{ ` } Y$
then obtain *y* **where** $y \in Y$ $x = f y$ **by** *blast* **note** *this*(2)
also have $\dots \leq g y$ **using** $\langle \text{fun-ord } (\leq) f g \rangle$ **by**(*simp add: fun-ord-def*)
also have $\dots \leq \bigsqcup (g \text{ ` } Y)$ **using** *chain1*[*OF* $\langle g \in Z \rangle$]
by(*rule ccpo-Sup-upper*)(*simp add:* $\langle y \in Y \rangle$)
finally show $x \leq \bigsqcup (g \text{ ` } Y)$.

```

    qed
  qed
  have chain3:  $\bigwedge x. \text{Complete-Partial-Order.chain } (\leq) ((\lambda f. f x) ' Z)$ 
    using Z by (rule chain-imageI) (simp add: fun-ord-def)
  have chain4:  $\text{Complete-Partial-Order.chain } (\leq) ((\lambda x. \bigsqcup ((\lambda f. f x) ' Z)) ' Y)$ 
    using Y
  proof (rule chain-imageI)
    fix f x y
    assume  $x \sqsubseteq y$ 
    show  $\bigsqcup ((\lambda f. f x) ' Z) \leq \bigsqcup ((\lambda f. f y) ' Z)$  (is -  $\leq$  ?rhs) using chain3
  proof (rule ccpo-Sup-least)
    fix x'
    assume  $x' \in (\lambda f. f x) ' Z$ 
    then obtain f where  $f \in Z$   $x' = f x$  by blast
    then show  $x' \leq$  ?rhs
      by (metis (mono-tags, lifting)  $\langle x \sqsubseteq y \rangle$  chain3 imageI ccpo-Sup-upper
        order-trans mono monotoneD)
    qed
  qed

  from chain2 have ?lhs  $\leq$  ?rhs
  proof (rule ccpo-Sup-least)
    fix x
    assume  $x \in (\lambda x. \bigsqcup (x ' Y)) ' Z$ 
    then obtain f where  $f \in Z$   $x = \bigsqcup (f ' Y)$  by blast note this(2)
    also have  $\dots \leq$  ?rhs using chain1 [OF  $\langle f \in Z \rangle$ ]
  proof (rule ccpo-Sup-least)
    fix x'
    assume  $x' \in f ' Y$ 
    then obtain y where  $y \in Y$   $x' = f y$  by blast
    then show  $x' \leq$  ?rhs
      by (metis (mono-tags, lifting)  $\langle f \in Z \rangle$  chain3 chain4 imageI local.ccpo-Sup-upper
        order.trans)
    qed
  finally show  $x \leq$  ?rhs .
  qed

  moreover
  have ?rhs  $\leq$  ?lhs using chain4
  proof (rule ccpo-Sup-least)
    fix x
    assume  $x \in (\lambda x. \bigsqcup ((\lambda f. f x) ' Z)) ' Y$ 
    then obtain y where  $y \in Y$   $x = \bigsqcup ((\lambda f. f y) ' Z)$  by blast note this(2)
    also have  $\dots \leq$  ?lhs using chain3
  proof (rule ccpo-Sup-least)
    fix x'
    assume  $x' \in (\lambda f. f y) ' Z$ 
    then obtain f where  $f \in Z$   $x' = f y$  by blast
    then show  $x' \leq$  ?lhs
      by (metis (mono-tags, lifting)  $\langle y \in Y \rangle$  ccpo-Sup-below-iff chain1 chain2)
  qed
  qed

```

```

imageI
  ccpo-Sup-upper)
  qed
  finally show  $x \leq ?lhs$  .
  qed
  ultimately show  $?lhs = ?rhs$ 
    by (rule order.antisym)
  qed

lemma fixp-mono:
  assumes fg: fun-ord ( $\leq$ )  $f$   $g$ 
  and f: monotone ( $\leq$ ) ( $\leq$ )  $f$ 
  and g: monotone ( $\leq$ ) ( $\leq$ )  $g$ 
  shows ccpo-class.fixp  $f \leq$  ccpo-class.fixp  $g$ 
unfolding fixp-def
proof(rule ccpo-Sup-least)
  fix x
  assume  $x \in$  ccpo-class.iterates  $f$ 
  thus  $x \leq \bigsqcup$  ccpo-class.iterates  $g$ 
  proof induction
    case (step x)
    from f step.IH have  $f x \leq f (\bigsqcup$  ccpo-class.iterates  $g)$  by(rule monotoneD)
    also have  $\dots \leq g (\bigsqcup$  ccpo-class.iterates  $g)$  using fg by(simp add: fun-ord-def)
    also have  $\dots = \bigsqcup$  ccpo-class.iterates  $g$  by(fold fixp-def fixp-unfold[OF g]) simp
    finally show ?case .
  qed(blast intro: ccpo-Sup-least)
qed(rule chain-iterates[OF f])

context fixes ordb :: 'b  $\Rightarrow$  'b  $\Rightarrow$  bool (infix  $\langle \sqsubseteq \rangle$  60) begin

lemma iterates-mono:
  assumes f:  $f \in$  ccpo.iterates (fun-lub Sup) (fun-ord ( $\leq$ ))  $F$ 
  and mono:  $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \Longrightarrow \text{monotone } (\sqsubseteq) (\leq) (F f)$ 
  shows monotone ( $\sqsubseteq$ ) ( $\leq$ )  $f$ 
  using f
  by(induction rule: ccpo.iterates.induct[OF ccpo-fun, consumes 1])(blast intro:
mono mono-lub)+

lemma fixp-preserves-mono:
  assumes mono:  $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. F f x)$ 
  and mono2:  $\bigwedge f. \text{monotone } (\sqsubseteq) (\leq) f \Longrightarrow \text{monotone } (\sqsubseteq) (\leq) (F f)$ 
  shows monotone ( $\sqsubseteq$ ) ( $\leq$ ) (ccpo.fixp (fun-lub Sup) (fun-ord ( $\leq$ ))  $F$ )
  (is monotone - - ?fixp)
proof(rule monotoneI)
  have mono: monotone (fun-ord ( $\leq$ )) (fun-ord ( $\leq$ ))  $F$ 
  by(rule monotoneI)(auto simp add: fun-ord-def intro: monotoneD[OF mono])
  let ?iter = ccpo.iterates (fun-lub Sup) (fun-ord ( $\leq$ ))  $F$ 
  have chain:  $\bigwedge x. \text{Complete-Partial-Order.chain } (\leq) ((\lambda f. f x) \text{ ‘ ‘ } ?iter)$ 
  by(rule chain-imageI[OF ccpo.chain-iterates[OF ccpo-fun mono]])(simp add:

```

```

fun-ord-def)
  fix x y
  assume  $x \sqsubseteq y$ 
  have  $(\bigsqcup_{f \in ?iter}. f\ x) \leq (\bigsqcup_{f \in ?iter}. f\ y)$ 
    using chain
  proof(rule ccpo-Sup-least)
    fix x'
    assume  $x' \in (\lambda f. f\ x) \text{ ' } ?iter$ 
    then obtain f where  $f: f \in ?iter\ x' = f\ x$  by blast
    then have  $f\ x \leq f\ y$ 
      by (metis  $\langle x \sqsubseteq y \rangle$  iterates-mono mono2 monotoneD)
    also have  $f\ y \leq \bigsqcup ((\lambda f. f\ y) \text{ ' } ?iter)$ 
      using chain f local.ccpo-Sup-upper by auto
    finally show  $x' \leq \dots$ 
      using f(2) by blast
  qed
  then show  $?fix\ x \leq ?fix\ y$ 
    unfolding ccpo.fix-def[OF ccpo-fun] fun-lub-apply .
  qed

end

end

```

```

lemma monotone2monotone:
  assumes 2:  $\bigwedge x. \text{monotone ordb ordc } (\lambda y. f\ x\ y)$ 
  and t: monotone orda ordb  $(\lambda x. t\ x)$ 
  and 1:  $\bigwedge y. \text{monotone orda ordc } (\lambda x. f\ x\ y)$ 
  and trans: transp ordc
  shows monotone orda ordc  $(\lambda x. f\ x\ (t\ x))$ 
    using assms unfolding monotone-on-def by (metis UNIV-I transpE)

```

15.1 Continuity

definition *cont* :: $('a\ set \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$

where

$cont\ luba\ orda\ lubb\ ordb\ f \longleftrightarrow$
 $(\forall Y. \text{Complete-Partial-Order.chain orda } Y \longrightarrow Y \neq \{\} \longrightarrow f\ (luba\ Y) = lubb\ (f\text{ ' } Y))$

definition *mcont* :: $('a\ set \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$

where

$mcont\ luba\ orda\ lubb\ ordb\ f \longleftrightarrow$
 $\text{monotone orda ordb } f \wedge cont\ luba\ orda\ lubb\ ordb\ f$

15.1.1 Theorem collection *cont-intro*

named-theorems *cont-intro* *continuity* and *admissibility intro rules*

```

ML ⟨
  (* apply cont-intro rules as intro and try to solve
     the remaining of the emerging subgoals with simp *)
  fun cont-intro-tac ctxt =
    REPEAT-ALL-NEW (resolve-tac ctxt (rev (Named-Theorems.get ctxt named-the-
orems ⟨cont-intro⟩)))
    THEN-ALL-NEW (SOLVED' (simp-tac ctxt))

  fun cont-intro-simproc ctxt ct =
    let
      fun mk-stmt t = t
        |> HOLogic.mk-Trueprop
        |> Thm.ctrm-of ctxt
        |> Goal.init
      fun mk-thm t =
        if exists-subterm Term.is-Var t then
          NONE
        else
          case SINGLE (cont-intro-tac ctxt 1) (mk-stmt t) of
            SOME thm => SOME (Goal.finish ctxt thm RS @{thm Eq-TrueI})
          | NONE => NONE
    in
      case Thm.term-of ct of
        t as Const-⟨ccpo.admissible - for - - -> => mk-thm t
        | t as Const-⟨mcont - - for - - - -> => mk-thm t
        | t as Const-⟨monotone-on - - for - - - -> => mk-thm t
        | - => NONE
      end
      handle THM - => NONE
      | TYPE - => NONE
    end
  ⟩

```

```

simproc-setup cont-intro
  ( ccpo.admissible lub ord P
    | mcont lub ord lub' ord' f
    | monotone ord ord' f
  ) = ⟨K cont-intro-simproc⟩

```

```

lemmas [cont-intro] =
  call-mono
  let-mono
  if-mono
  option.const-mono
  tailrec.const-mono
  bind-mono

```

experiment begin

The following proof by simplification diverges if variables are not handled properly.

```

lemma ( $\wedge f. \text{monotone } R \ S \ f \implies \text{thesis}$ )  $\implies \text{monotone } R \ S \ g \implies \text{thesis}$ 
  by simp

end

declare if-mono[simp]

lemma monotone-id' [cont-intro]: monotone ord ord ( $\lambda x. x$ )
  by(simp add: monotone-def)

lemma monotone-applyI:
  monotone orda ordb F  $\implies$  monotone (fun-ord orda) ordb ( $\lambda f. F \ (f \ x)$ )
  by(rule monotoneI)(auto simp add: fun-ord-def dest: monotoneD)

lemma monotone-if-fun [partial-function-mono]:
   $\llbracket \text{monotone (fun-ord orda) (fun-ord ordb) } F; \text{monotone (fun-ord orda) (fun-ord ordb) } G \rrbracket$ 
   $\implies \text{monotone (fun-ord orda) (fun-ord ordb) } (\lambda f \ n. \text{if } c \ n \text{ then } F \ f \ n \text{ else } G \ f \ n)$ 
  by(simp add: monotone-def fun-ord-def)

lemma monotone-fun-apply-fun [partial-function-mono]:
  monotone (fun-ord (fun-ord ord)) (fun-ord ord) ( $\lambda f \ n. f \ t \ (g \ n)$ )
  by(rule monotoneI)(simp add: fun-ord-def)

lemma monotone-fun-ord-apply:
  monotone orda (fun-ord ordb) f  $\longleftrightarrow (\forall x. \text{monotone orda ordb } (\lambda y. f \ y \ x))$ 
  by(auto simp add: monotone-def fun-ord-def)

context preorder begin

declare transp-on-le[cont-intro]

lemma monotone-const [simp, cont-intro]: monotone ord ( $\leq$ ) ( $\lambda -. c$ )
  by(rule monotoneI) simp

end

lemma transp-le [cont-intro, simp]:
  class.preorder ord (mk-less ord)  $\implies \text{transp ord}$ 
  by(rule preorder.transp-on-le)

context partial-function-definitions begin

declare const-mono [cont-intro, simp]

lemma transp-le [cont-intro, simp]: transp leq
  by(rule transpI)(rule leq-trans)

lemma preorder [cont-intro, simp]: class.preorder leq (mk-less leq)

```

```

  by(unfold-locales)(auto simp add: mk-less-def intro: leq-refl leq-trans)

declare ccpo[cont-intro, simp]

end

lemma contI [intro?]:
  ( $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } \text{orda } Y; Y \neq \{\} \rrbracket \implies f \text{ (luba } Y) = \text{lubb (f ` Y)}$ )
   $\implies \text{cont luba orda lubb ordb f}$ 
  unfolding cont-def by blast

lemma contD:
   $\llbracket \text{cont luba orda lubb ordb f; Complete-Partial-Order.chain } \text{orda } Y; Y \neq \{\} \rrbracket$ 
   $\implies f \text{ (luba } Y) = \text{lubb (f ` Y)}$ 
  unfolding cont-def by blast

lemma cont-id [simp, cont-intro]:  $\bigwedge \text{Sup. cont Sup ord Sup ord id}$ 
  by(rule contI) simp

lemma cont-id' [simp, cont-intro]:  $\bigwedge \text{Sup. cont Sup ord Sup ord } (\lambda x. x)$ 
  by (simp add: Inf.INF-identity-eq contI)

lemma cont-applyI [cont-intro]:
  assumes cont: cont luba orda lubb ordb g
  shows cont (fun-lub luba) (fun-ord orda) lubb ordb ( $\lambda f. g \text{ (f x)}$ )
  using assms by (simp add: cont-def chain-fun-ordD fun-lub-apply image-image)

lemma call-cont: cont (fun-lub lub) (fun-ord ord) lub ord ( $\lambda f. f t$ )
  by(simp add: cont-def fun-lub-apply)

lemma cont-if [cont-intro]:
   $\llbracket \text{cont luba orda lubb ordb f; cont luba orda lubb ordb g} \rrbracket$ 
   $\implies \text{cont luba orda lubb ordb } (\lambda x. \text{if } c \text{ then } f x \text{ else } g x)$ 
  by(cases c) simp-all

lemma mcontI [intro?]:
   $\llbracket \text{monotone orda ordb f; cont luba orda lubb ordb f} \rrbracket \implies \text{mcont luba orda lubb ordb f}$ 
  by(simp add: mcont-def)

lemma mcont-mono: mcont luba orda lubb ordb f  $\implies \text{monotone orda ordb f}$ 
  by(simp add: mcont-def)

lemma mcont-cont [simp]: mcont luba orda lubb ordb f  $\implies \text{cont luba orda lubb ordb f}$ 
  by(simp add: mcont-def)

lemma mcont-monoD:

```

$\llbracket \text{mcont luba orda lubb ordb } f; \text{ orda } x \ y \rrbracket \implies \text{ordb } (f \ x) \ (f \ y)$
by(*auto simp add: mcont-def dest: monotoneD*)

lemma *mcont-contD*:

$\llbracket \text{mcont luba orda lubb ordb } f; \text{ Complete-Partial-Order.chain orda } Y; Y \neq \{\} \rrbracket$
 $\implies f \ (\text{luba } Y) = \text{lubb } (f \ ' \ Y)$
by(*auto simp add: mcont-def dest: contD*)

lemma *mcont-call* [*cont-intro, simp*]:

mcont (fun-lub lub) (fun-ord ord) lub ord ($\lambda f. f \ t$)
by(*simp add: mcont-def call-mono call-cont*)

lemma *mcont-id'* [*cont-intro, simp*]: *mcont lub ord lub ord* ($\lambda x. x$)

by(*simp add: mcont-def monotone-id'*)

lemma *mcont-applyI*:

mcont luba orda lubb ordb ($\lambda x. F \ x$) $\implies \text{mcont (fun-lub luba) (fun-ord orda) lubb ordb}$ ($\lambda f. F \ (f \ x)$)
by(*simp add: mcont-def monotone-applyI cont-applyI*)

lemma *mcont-if* [*cont-intro, simp*]:

$\llbracket \text{mcont luba orda lubb ordb } (\lambda x. f \ x); \text{mcont luba orda lubb ordb } (\lambda x. g \ x) \rrbracket$
 $\implies \text{mcont luba orda lubb ordb } (\lambda x. \text{if } c \text{ then } f \ x \text{ else } g \ x)$
by(*simp add: mcont-def cont-if*)

lemma *cont-fun-lub-apply*:

cont luba orda (fun-lub lubb) (fun-ord ordb) f $\longleftrightarrow (\forall x. \text{cont luba orda lubb ordb } (\lambda y. f \ y \ x))$
by(*simp add: cont-def fun-lub-def fun-eq-iff*)(*auto simp add: image-def*)

lemma *mcont-fun-lub-apply*:

mcont luba orda (fun-lub lubb) (fun-ord ordb) f $\longleftrightarrow (\forall x. \text{mcont luba orda lubb ordb } (\lambda y. f \ y \ x))$
by(*auto simp add: monotone-fun-ord-apply cont-fun-lub-apply mcont-def*)

context *ccpo* **begin**

lemma *cont-const* [*simp, cont-intro*]: *cont luba orda Sup* (\leq) ($\lambda x. c$)

by (*rule contI*) (*simp add: image-constant-conv cong del: SUP-cong-simp*)

lemma *mcont-const* [*cont-intro, simp*]:

mcont luba orda Sup (\leq) ($\lambda x. c$)
by(*simp add: mcont-def*)

lemma *cont-apply*:

assumes 2: $\bigwedge x. \text{cont lubb ordb Sup } (\leq) \ (\lambda y. f \ x \ y)$
and *t*: *cont luba orda lubb ordb* ($\lambda x. t \ x$)
and 1: $\bigwedge y. \text{cont luba orda Sup } (\leq) \ (\lambda x. f \ x \ y)$
and *mono*: *monotone orda ordb* ($\lambda x. t \ x$)

and *mono2*: $\bigwedge x. \text{monotone ord}b (\leq) (\lambda y. f\ x\ y)$
and *mono1*: $\bigwedge y. \text{monotone ord}a (\leq) (\lambda x. f\ x\ y)$
shows *cont luba orda Sup* $(\leq) (\lambda x. f\ x\ (t\ x))$
proof
fix *Y*
assume *chain*: *Complete-Partial-Order.chain orda Y* **and** $Y \neq \{\}$
moreover from *chain* **have** *chain'*: *Complete-Partial-Order.chain ordb* $(t \text{ ‘ } Y)$
by(*rule chain-imageI*)(*rule monotoneD*[*OF mono*])
ultimately show $f\ (\text{luba } Y)\ (t\ (\text{luba } Y)) = \bigsqcup ((\lambda x. f\ x\ (t\ x)) \text{ ‘ } Y)$
by(*simp add: contD*[*OF 1*] *contD*[*OF t*] *contD*[*OF 2*] *image-image*)
(rule diag-Sup[*OF chain*], *auto intro: monotone2monotone*[*OF mono2 mono*]
monotone-const transpI] *monotoneD*[*OF mono1*])
qed

lemma *mcont2mcont'*:
 $\llbracket \bigwedge x. \text{mcont lub' ord' Sup} (\leq) (\lambda y. f\ x\ y);$
 $\bigwedge y. \text{mcont lub ord Sup} (\leq) (\lambda x. f\ x\ y);$
 $\text{mcont lub ord lub' ord'} (\lambda y. t\ y) \rrbracket$
 $\implies \text{mcont lub ord Sup} (\leq) (\lambda x. f\ x\ (t\ x))$
unfolding *mcont-def* **by**(*blast intro: transp-on-le monotone2monotone cont-apply*)

lemma *mcont2mcont*:
 $\llbracket \text{mcont lub' ord' Sup} (\leq) (\lambda x. f\ x); \text{mcont lub ord lub' ord'} (\lambda x. t\ x) \rrbracket$
 $\implies \text{mcont lub ord Sup} (\leq) (\lambda x. f\ (t\ x))$
by(*rule mcont2mcont'*[*OF - mcont-const*])

context
fixes *ord* :: $'b \Rightarrow 'b \Rightarrow \text{bool}$ (**infix** $\langle \sqsubseteq \rangle$ 60)
and *lub* :: $'b \text{ set} \Rightarrow 'b$ ($\langle \bigvee \rangle$)
begin

lemma *cont-fun-lub-Sup*:
assumes *chainM*: *Complete-Partial-Order.chain* $(\text{fun-ord } (\leq))\ M$
and *mcont* [*rule-format*]: $\forall f \in M. \text{mcont lub } (\sqsubseteq)\ \text{Sup } (\leq)\ f$
shows *cont lub* $(\sqsubseteq)\ \text{Sup } (\leq)\ (\text{fun-lub Sup } M)$
proof(*rule contI*)
fix *Y*
assume *chain*: *Complete-Partial-Order.chain* $(\sqsubseteq)\ Y$
and *Y*: $Y \neq \{\}$
from *swap-Sup*[*OF chain chainM mcont*[*THEN mcont-mono*]]
show *fun-lub Sup M* $(\bigvee Y) = \bigsqcup (\text{fun-lub Sup } M \text{ ‘ } Y)$
by(*simp add: mcont-contD*[*OF mcont chain Y*] *fun-lub-apply cong: image-cong*)
qed

lemma *mcont-fun-lub-Sup*:
 $\llbracket \text{Complete-Partial-Order.chain } (\text{fun-ord } (\leq))\ M;$
 $\forall f \in M. \text{mcont lub ord Sup} (\leq)\ f \rrbracket$
 $\implies \text{mcont lub } (\sqsubseteq)\ \text{Sup } (\leq)\ (\text{fun-lub Sup } M)$
by(*simp add: mcont-def cont-fun-lub-Sup mono-lub*)

lemma *iterates-mcont*:

assumes $f: f \in \text{ccpo.iterates } (\text{fun-lub } \text{Sup}) \ (\text{fun-ord } (\leq)) \ F$
and $\text{mono}: \bigwedge f. \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ f \implies \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ (F \ f)$
shows $\text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ f$
using f
by(*induction rule: ccpo.iterates.induct[OF ccpo-fun, consumes 1, case-names step Sup]*)(*blast intro: mono mcont-fun-lub-Sup*)+

lemma *fixp-preserves-mcont*:

assumes $\text{mono}: \bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) \ (\leq) \ (\lambda f. F \ f \ x)$
and $\text{mcont}: \bigwedge f. \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ f \implies \text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ (F \ f)$
shows $\text{mcont lub } (\sqsubseteq) \ \text{Sup } (\leq) \ (\text{ccpo.fixp } (\text{fun-lub } \text{Sup}) \ (\text{fun-ord } (\leq)) \ F)$
(is $\text{mcont} \text{ - - - } ?\text{fixp}$ **)**
unfolding mcont-def
proof(*intro conjI monotoneI contI*)
have $\text{mono}: \text{monotone } (\text{fun-ord } (\leq)) \ (\text{fun-ord } (\leq)) \ F$
by(*rule monotoneI*)(*auto simp add: fun-ord-def intro: monotoneD[OF mono]*)
let $?iter = \text{ccpo.iterates } (\text{fun-lub } \text{Sup}) \ (\text{fun-ord } (\leq)) \ F$
have $\text{chain}: \bigwedge x. \text{Complete-Partial-Order.chain } (\leq) \ ((\lambda f. f \ x) \text{ ‘ } ?iter)$
by(*rule chain-imageI[OF ccpo.chain-iterates[OF ccpo-fun mono]]*)(*simp add: fun-ord-def*)

show $?fixp \ x \leq ?fixp \ y$ **if** $x \sqsubseteq y$ **for** $x \ y$

proof –

have $(\bigsqcup f \in ?iter. f \ x)$

$\leq (\bigsqcup f \in ?iter. f \ y)$

using chain

proof(*rule ccpo-Sup-least*)

fix x'

assume $x' \in (\lambda f. f \ x) \text{ ‘ } ?iter$

then obtain f **where** $f: f \in ?iter \ x' = f \ x$ **by** *blast*

then have $f \ x \leq f \ y$

by (*metis iterates-mcont mcont mcont-monoD that*)

also have $f \ y \leq \bigsqcup ((\lambda f. f \ y) \text{ ‘ } ?iter)$

using $\text{chain } f \ \text{local.ccpo-Sup-upper}$ **by** *auto*

finally show $x' \leq \dots$

using $f(2)$ **by** *blast*

qed

then show $?thesis$

by (*simp add: ccpo.fixp-def[OF ccpo-fun] fun-lub-apply*)

qed

show $?fixp \ (\bigvee Y) = \bigsqcup (?fixp \text{ ‘ } Y)$

if $\text{chain}: \text{Complete-Partial-Order.chain } (\sqsubseteq) \ Y$ **and** $Y: Y \neq \{\}$ **for** Y

proof –

have $f \ (\bigvee Y) = \bigsqcup (f \text{ ‘ } Y)$ **if** $f \in ?iter$ **for** f

using *that mcont chain Y*

by (*rule mcont-contD[OF iterates-mcont]*)

moreover have $\bigsqcup ((\lambda f. \bigsqcup (f \text{ ‘ } Y)) \text{ ‘ } ?iter) = \bigsqcup ((\lambda x. \bigsqcup ((\lambda f. f \ x) \text{ ‘ } ?iter)) \text{ ‘ } \dots$

```

Y)
  using chain ccpo.chain-iterates[OF ccpo-fun mono]
  by (rule swap-Sup)(rule mcont-mono[OF iterates-mcont[OF - mcont]])
ultimately show ?thesis
  unfolding ccpo.fixp-def[OF ccpo-fun]
  by (simp add: fun-lub-apply cong: image-cong)
qed
qed

end

context
  fixes F :: 'c  $\Rightarrow$  'c and U :: 'c  $\Rightarrow$  'b  $\Rightarrow$  'a and C :: ('b  $\Rightarrow$  'a)  $\Rightarrow$  'c and f
  assumes mono:  $\bigwedge x. \text{monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda f. U (F (C f)) x)$ 
  and eq:  $f \equiv C (\text{ccpo.fixp } (\text{fun-lub Sup}) (\text{fun-ord } (\leq)) (\lambda f. U (F (C f))))$ 
  and inverse:  $\bigwedge f. U (C f) = f$ 
begin

lemma fixp-preserves-mono-uc:
  assumes mono2:  $\bigwedge f. \text{monotone ord } (\leq) (U f) \implies \text{monotone ord } (\leq) (U (F f))$ 
  shows  $\text{monotone ord } (\leq) (U f)$ 
using fixp-preserves-mono[OF mono mono2] by (subst eq)(simp add: inverse)

lemma fixp-preserves-mcont-uc:
  assumes mcont:  $\bigwedge f. \text{mcont lubb ordb Sup } (\leq) (U f) \implies \text{mcont lubb ordb Sup } (\leq) (U (F f))$ 
  shows  $\text{mcont lubb ordb Sup } (\leq) (U f)$ 
using fixp-preserves-mcont[OF mono mcont] by (subst eq)(simp add: inverse)

end

lemmas fixp-preserves-mono1 = fixp-preserves-mono-uc[of  $\lambda x. x - \lambda x. x$ , OF - - refl]
lemmas fixp-preserves-mono2 =
  fixp-preserves-mono-uc[of case-prod - curry, unfolded case-prod-curry curry-case-prod, OF - - refl]
lemmas fixp-preserves-mono3 =
  fixp-preserves-mono-uc[of  $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$ , unfolded case-prod-curry curry-case-prod, OF - - refl]
lemmas fixp-preserves-mono4 =
  fixp-preserves-mono-uc[of  $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$ , unfolded case-prod-curry curry-case-prod, OF - - refl]

lemmas fixp-preserves-mcont1 = fixp-preserves-mcont-uc[of  $\lambda x. x - \lambda x. x$ , OF - - refl]
lemmas fixp-preserves-mcont2 =
  fixp-preserves-mcont-uc[of case-prod - curry, unfolded case-prod-curry curry-case-prod, OF - - refl]
lemmas fixp-preserves-mcont3 =

```

fixp-preserves-mcont-uc[of $\lambda f. \text{case-prod } (\text{case-prod } f) - \lambda f. \text{curry } (\text{curry } f)$, unfolded *case-prod-curry curry-case-prod*, *OF - - refl*]

lemmas *fixp-preserves-mcont₄* =

fixp-preserves-mcont-uc[of $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$, unfolded *case-prod-curry curry-case-prod*, *OF - - refl*]

end

lemma (in *preorder*) *monotone-if-bot*:

fixes *bot*

assumes *mono*: $\bigwedge x y. \llbracket x \leq y; \neg (x \leq \text{bound}) \rrbracket \implies \text{ord } (f x) (f y)$

and *bot*: $\bigwedge x. \neg x \leq \text{bound} \implies \text{ord bot } (f x) \text{ ord bot bot}$

shows *monotone* (\leq) *ord* ($\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x$)

by(*rule monotoneI*)(*auto intro: bot intro: mono order-trans*)

lemma (in *ccpo*) *mcont-if-bot*:

fixes *bot* **and** *lub* ($\langle \vee \rangle$) **and** *ord* (**infix** $\langle \sqsubseteq \rangle$ 60)

assumes *ccpo*: *class.ccpo* *lub* (\sqsubseteq) *lt*

and *mono*: $\bigwedge x y. \llbracket x \leq y; \neg x \leq \text{bound} \rrbracket \implies f x \sqsubseteq f y$

and *cont*: $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain } (\leq) Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg x \leq \text{bound} \rrbracket \implies f (\bigsqcup Y) = \bigvee (f \restriction Y)$

and *bot*: $\bigwedge x. \neg x \leq \text{bound} \implies \text{bot} \sqsubseteq f x$

shows *mcont* *Sup* (\leq) *lub* (\sqsubseteq) ($\lambda x. \text{if } x \leq \text{bound} \text{ then } \text{bot} \text{ else } f x$) (**is** *mcont - - ?g*)

proof(*intro mcontI contI*)

interpret *c*: *ccpo* *lub* (\sqsubseteq) *lt* **by**(*fact ccpo*)

show *monotone* (\leq) (\sqsubseteq) *?g* **by**(*rule monotone-if-bot*)(*simp-all add: mono bot*)

fix *Y*

assume *chain*: *Complete-Partial-Order.chain* (\leq) *Y* **and** *Y*: $Y \neq \{\}$

show *?g* ($\bigsqcup Y$) = $\bigvee (?g \restriction Y)$

proof(*cases* $Y \subseteq \{x. x \leq \text{bound}\}$)

case *True*

hence $\bigsqcup Y \leq \text{bound}$ **using** *chain* **by**(*auto intro: ccpo-Sup-least*)

moreover **have** $Y \cap \{x. \neg x \leq \text{bound}\} = \{\}$ **using** *True* **by** *auto*

ultimately **show** *?thesis* **using** *True Y*

by (*auto simp add: image-constant-conv cong del: c.SUP-cong-simp*)

next

case *False*

let *?Y* = $Y \cap \{x. \neg x \leq \text{bound}\}$

have *chain'*: *Complete-Partial-Order.chain* (\leq) *?Y*

using *chain* **by**(*rule chain-subset*) *simp*

from *False* **obtain** *y* **where** *ybound*: $\neg y \leq \text{bound}$ **and** *y*: $y \in Y$ **by** *blast*

hence $\neg \bigsqcup Y \leq \text{bound}$ **by** (*metis ccpo-Sup-upper chain order.trans*)

hence *?g* ($\bigsqcup Y$) = $f (\bigsqcup Y)$ **by** *simp*

also **have** $\bigsqcup Y \leq \bigsqcup ?Y$ **using** *chain*

proof(*rule ccpo-Sup-least*)

fix *x*

```

assume  $x: x \in Y$ 
show  $x \leq \sqcup ?Y$ 
proof(cases  $x \leq \text{bound}$ )
  case True
    with chainD[OF chain  $x\ y$ ] have  $x \leq y$  using ybound by(auto intro:
order-trans)
    thus ?thesis by(rule order-trans)(auto intro: ccpo-Sup-upper[OF chain']
simp add: y ybound)
    qed(auto intro: ccpo-Sup-upper[OF chain'] simp add:  $x$ )
  qed
  hence  $\sqcup Y = \sqcup ?Y$  by(rule order.antisym)(blast intro: ccpo-Sup-least[OF
chain'] ccpo-Sup-upper[OF chain'])
  hence  $f(\sqcup Y) = f(\sqcup ?Y)$  by simp
  also have  $f(\sqcup ?Y) = \bigvee(f' ?Y)$  using chain' by(rule cont)(insert y ybound,
auto)
  also have  $\bigvee(f' ?Y) = \bigvee(?g' Y)$ 
  proof(cases  $Y \cap \{x. x \leq \text{bound}\} = \{\}$ )
    case True
      hence  $f' ?Y = ?g' Y$  by auto
      thus ?thesis by(rule arg-cong)
    next
      case False
        have chain'': Complete-Partial-Order.chain ( $\sqsubseteq$ ) (insert bot ( $f' ?Y$ ))
          using chain by(auto intro!: chainI bot dest: chainD intro: mono)
        hence chain''': Complete-Partial-Order.chain ( $\sqsubseteq$ ) ( $f' ?Y$ ) by(rule chain-subset)
        blast
        have  $\text{bot} \sqsubseteq \bigvee(f' ?Y)$  using y ybound by(blast intro: c.order-trans[OF bot]
c.ccpo-Sup-upper[OF chain''''])
        hence  $\bigvee(\text{insert bot } (f' ?Y)) \sqsubseteq \bigvee(f' ?Y)$  using chain''
          by(auto intro: c.ccpo-Sup-least c.ccpo-Sup-upper[OF chain''''])
        with - have  $\dots = \bigvee(\text{insert bot } (f' ?Y))$ 
          by(rule c.order.antisym)(blast intro: c.ccpo-Sup-least[OF chain''''] c.ccpo-Sup-upper[OF
chain''''])
        also have  $\text{insert bot } (f' ?Y) = ?g' Y$  using False by auto
        finally show ?thesis .
      qed
    finally show ?thesis .
  qed
qed

```

context *partial-function-definitions* **begin**

lemma *mcont-const* [*cont-intro, simp*]:

mcont luba orda lub leq ($\lambda x. c$)

by(*rule ccpo.mcont-const*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemmas [*cont-intro, simp*] =

ccpo.cont-const[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *mono2mono*:

assumes *monotone ordb leq* $(\lambda y. f y)$ *monotone orda ordb* $(\lambda x. t x)$

shows *monotone orda leq* $(\lambda x. f (t x))$

using *assms* **by**(*rule monotone2monotone*) *simp-all*

lemmas *mcont2mcont'* = *ccpo.mcont2mcont'*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *mcont2mcont* = *ccpo.mcont2mcont*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono1* = *ccpo.fixp-preserves-mono1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono2* = *ccpo.fixp-preserves-mono2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono3* = *ccpo.fixp-preserves-mono3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono4* = *ccpo.fixp-preserves-mono4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont1* = *ccpo.fixp-preserves-mcont1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont2* = *ccpo.fixp-preserves-mcont2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont3* = *ccpo.fixp-preserves-mcont3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mcont4* = *ccpo.fixp-preserves-mcont4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *monotone-if-bot*:

fixes *bot*

assumes *g*: $\bigwedge x. g x = (\text{if } \text{leq } x \text{ bound then } \text{bot else } f x)$

and *mono*: $\bigwedge x y. \llbracket \text{leq } x y; \neg \text{leq } x \text{ bound} \rrbracket \implies \text{ord } (f x) (f y)$

and *bot*: $\bigwedge x. \neg \text{leq } x \text{ bound} \implies \text{ord bot } (f x) \text{ ord bot bot}$

shows *monotone leq ord g*

unfolding *g*[*abs-def*] **using** *preorder mono bot* **by**(*rule preorder.monotone-if-bot*)

lemma *mcont-if-bot*:

fixes *bot*

assumes *ccpo*: *class.ccpo lub' ord* (*mk-less ord*)

and *bot*: $\bigwedge x. \neg \text{leq } x \text{ bound} \implies \text{ord bot } (f x)$

and *g*: $\bigwedge x. g x = (\text{if } \text{leq } x \text{ bound then } \text{bot else } f x)$

and *mono*: $\bigwedge x y. \llbracket \text{leq } x y; \neg \text{leq } x \text{ bound} \rrbracket \implies \text{ord } (f x) (f y)$

and *cont*: $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain leq } Y; Y \neq \{\}; \bigwedge x. x \in Y \implies \neg \text{leq } x \text{ bound} \rrbracket \implies f (\text{lub } Y) = \text{lub}' (f \text{ ` } Y)$

shows *mcont lub leq lub' ord g*

unfolding *g*[*abs-def*] **using** *ccpo mono cont bot* **by**(*rule ccpo.mcont-if-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]])

end

15.2 Admissibility

lemma *admissible-subst*:

assumes *adm*: *ccpo.admissible luba orda* $(\lambda x. P\ x)$

and *mcont*: *mcont lubb ordb luba orda f*

shows *ccpo.admissible lubb ordb* $(\lambda x. P\ (f\ x))$

using *assms* **by** (*simp add: ccpo.admissible-def chain-imageI mcont-contD mcont-monoD*)

lemmas [*simp, cont-intro*] =

admissible-all

admissible-ball

admissible-const

admissible-conj

lemma *admissible-disj'* [*simp, cont-intro*]:

$\llbracket \text{class.ccpo lub ord (mk-less ord); ccpo.admissible lub ord } P; \text{ccpo.admissible lub ord } Q \rrbracket$

$\implies \text{ccpo.admissible lub ord } (\lambda x. P\ x \vee Q\ x)$

by(*rule ccpo.admissible-disj*)

lemma *admissible-imp'* [*cont-intro*]:

$\llbracket \text{class.ccpo lub ord (mk-less ord);$

ccpo.admissible lub ord $(\lambda x. \neg P\ x);$

ccpo.admissible lub ord $(\lambda x. Q\ x) \rrbracket$

$\implies \text{ccpo.admissible lub ord } (\lambda x. P\ x \longrightarrow Q\ x)$

unfolding *imp-conv-disj* **by**(*rule ccpo.admissible-disj*)

lemma *admissible-imp* [*cont-intro*]:

$(Q \implies \text{ccpo.admissible lub ord } (\lambda x. P\ x))$

$\implies \text{ccpo.admissible lub ord } (\lambda x. Q \longrightarrow P\ x)$

by(*rule ccpo.admissibleI*)(*auto dest: ccpo.admissibleD*)

lemma *admissible-not-mem'* [*THEN admissible-subst, cont-intro, simp*]:

shows *admissible-not-mem*: *ccpo.admissible Union* (\subseteq) $(\lambda A. x \notin A)$

by(*rule ccpo.admissibleI*) *auto*

lemma *admissible-eqI*:

assumes *f*: *cont luba orda lub ord* $(\lambda x. f\ x)$

and *g*: *cont luba orda lub ord* $(\lambda x. g\ x)$

shows *ccpo.admissible luba orda* $(\lambda x. f\ x = g\ x)$

by (*smt (verit, best) Sup.SUP-cong ccpo.admissible-def contD assms*)

corollary *admissible-eq-mcontI* [*cont-intro*]:

$\llbracket \text{mcont luba orda lub ord } (\lambda x. f\ x);$

mcont luba orda lub ord $(\lambda x. g\ x) \rrbracket$

$\implies \text{ccpo.admissible luba orda } (\lambda x. f\ x = g\ x)$

by(*rule admissible-eqI*)(*auto simp add: mcont-def*)

lemma *admissible-iff* [*cont-intro, simp*]:

$\llbracket \text{ccpo.admissible lub ord } (\lambda x. P\ x \longrightarrow Q\ x); \text{ccpo.admissible lub ord } (\lambda x. Q\ x \longrightarrow$

$P\ x) \]$
 $\implies \text{ccpo.admissible lub ord } (\lambda x. P\ x \longleftrightarrow Q\ x)$
 $\text{by}(\text{subst iff-conv-conj-imp})(\text{rule admissible-conj})$

context *ccpo* **begin**

lemma *admissible-leI*:

assumes *f*: *mcont luba orda Sup* (\leq) ($\lambda x. f\ x$)
and *g*: *mcont luba orda Sup* (\leq) ($\lambda x. g\ x$)
shows *ccpo.admissible luba orda* ($\lambda x. f\ x \leq g\ x$)
proof(*rule ccpo.admissibleI*)
fix *A*
assume *chain*: *Complete-Partial-Order.chain orda A*
and *le*: $\forall x \in A. f\ x \leq g\ x$
and *False*: $A \neq \{\}$
have $f\ (\text{luba } A) = \bigsqcup (f\ ' A)$ **by**(*simp add: mcont-contD[OF f] chain False*)
also have $\dots \leq \bigsqcup (g\ ' A)$
proof(*rule ccpo-Sup-least*)
from *chain* **show** *Complete-Partial-Order.chain* (\leq) ($f\ ' A$)
by(*rule chain-imageI*)(*rule mcont-monoD[OF f]*)
fix *x*
assume $x \in f\ ' A$
then obtain *y* **where** $y \in A\ x = f\ y$ **by** *blast* **note** *this*(2)
also have $f\ y \leq g\ y$ **using** *le* $\langle y \in A \rangle$ **by** *simp*
also have *Complete-Partial-Order.chain* (\leq) ($g\ ' A$)
using *chain* **by**(*rule chain-imageI*)(*rule mcont-monoD[OF g]*)
hence $g\ y \leq \bigsqcup (g\ ' A)$ **by**(*rule ccpo-Sup-upper*)(*simp add:* $\langle y \in A \rangle$)
finally show $x \leq \dots$.
qed
also have $\dots = g\ (\text{luba } A)$ **by**(*simp add: mcont-contD[OF g] chain False*)
finally show $f\ (\text{luba } A) \leq g\ (\text{luba } A)$.
qed
end

lemma *admissible-leI*:

fixes *ord* (**infix** $\langle \sqsubseteq \rangle$ 60) **and** *lub* ($\langle \bigvee \rangle$)
assumes *class.ccpo lub* (\sqsubseteq) (*mk-less* (\sqsubseteq))
and *mcont luba orda lub* (\sqsubseteq) ($\lambda x. f\ x$)
and *mcont luba orda lub* (\sqsubseteq) ($\lambda x. g\ x$)
shows *ccpo.admissible luba orda* ($\lambda x. f\ x \sqsubseteq g\ x$)
using *assms* **by**(*rule ccpo.admissible-leI*)

declare *ccpo-class.admissible-leI*[*cont-intro*]

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible Sup* (\leq) ($\lambda x. \neg (\leq) x\ y$)
by(*rule ccpo.admissibleI*)(*simp add: ccpo-Sup-below-iff*)

end

lemma (in preorder) preorder [cont-intro, simp]: class.preorder (\leq) (mk-less (\leq))
by(unfold-locales)(auto simp add: mk-less-def intro: order-trans)

context partial-function-definitions **begin**

lemmas [cont-intro, simp] =
 admissible-leI[OF Partial-Function.ccpo[OF partial-function-definitions-axioms]]
 ccpo.admissible-not-below[THEN admissible-subst, OF Partial-Function.ccpo[OF
 partial-function-definitions-axioms]]

end

setup ‹Sign.map-naming (Name-Space.mandatory-path ccpo)›

inductive compact :: ('a set \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool
for lub ord x
where compact:
 [ccpo.admissible lub ord ($\lambda y. \neg$ ord x y);
 ccpo.admissible lub ord ($\lambda y. x \neq y$)]
 \Rightarrow compact lub ord x

setup ‹Sign.map-naming Name-Space.parent-path›

context ccpo **begin**

lemma compactI:
 assumes ccpo.admissible Sup (\leq) ($\lambda y. \neg x \leq y$)
 shows ccpo.compact Sup (\leq) x
using assms
proof(rule ccpo.compact.intros)
 have neg: ($\lambda y. x \neq y$) = ($\lambda y. \neg x \leq y \vee \neg y \leq x$) **by**(auto)
 show ccpo.admissible Sup (\leq) ($\lambda y. x \neq y$)
by(subst neg)(rule admissible-disj admissible-not-below assms)+
qed

lemma compact-bot:
 assumes x = Sup {}
 shows ccpo.compact Sup (\leq) x
proof(rule compactI)
 show ccpo.admissible Sup (\leq) ($\lambda y. \neg x \leq y$) **using** assms
by(auto intro!: ccpo.admissibleI intro: ccpo-Sup-least chain-empty)
qed

end

lemma admissible-compact-neg' [THEN admissible-subst, cont-intro, simp]:

shows *admissible-compact-neq*: *ccpo.compact lub ord k* \implies *ccpo.admissible lub ord* ($\lambda x. k \neq x$)

by(*simp add: ccpo.compact.simps*)

lemma *admissible-neq-compact'* [*THEN admissible-subst, cont-intro, simp*]:

shows *admissible-neq-compact*: *ccpo.compact lub ord k* \implies *ccpo.admissible lub ord* ($\lambda x. x \neq k$)

by(*subst eq-commute*)(*rule admissible-compact-neq*)

context *partial-function-definitions* **begin**

lemmas [*cont-intro, simp*] = *ccpo.compact-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

end

context *ccpo* **begin**

lemma *fixp-strong-induct*:

assumes [*cont-intro*]: *ccpo.admissible Sup* (\leq) *P*

and *mono*: *monotone* (\leq) (\leq) *f*

and *bot*: *P* ($\sqcup \{\}$)

and *step*: $\bigwedge x. \llbracket x \leq \text{ccpo-class.fixp } f; P \ x \rrbracket \implies P \ (f \ x)$

shows *P* (*ccpo-class.fixp f*)

proof(*rule fixp-induct*[**where** *P*= $\lambda x. x \leq \text{ccpo-class.fixp } f \wedge P \ x$, *THEN conjunct2*])

note [*cont-intro*] = *admissible-leI*

show *ccpo.admissible Sup* (\leq) ($\lambda x. x \leq \text{ccpo-class.fixp } f \wedge P \ x$) **by** *simp*

next

show $\sqcup \{\} \leq \text{ccpo-class.fixp } f \wedge P \ (\sqcup \{\})$

by(*auto simp add: bot intro: ccpo-Sup-least chain-empty*)

next

fix *x*

assume $x \leq \text{ccpo-class.fixp } f \wedge P \ x$

thus $f \ x \leq \text{ccpo-class.fixp } f \wedge P \ (f \ x)$

by(*subst fixp-unfold*[*OF mono*])(*auto dest: monotoneD*[*OF mono*] *intro: step*)

qed(*rule mono*)

end

context *partial-function-definitions* **begin**

lemma *fixp-strong-induct-uc*:

fixes *F* :: $'c \Rightarrow 'c$

and *U* :: $'c \Rightarrow 'b \Rightarrow 'a$

and *C* :: $('b \Rightarrow 'a) \Rightarrow 'c$

and *P* :: $('b \Rightarrow 'a) \Rightarrow \text{bool}$

assumes *mono*: $\bigwedge x. \text{mono-body } (\lambda f. U \ (F \ (C \ f)) \ x)$

and *eq*: $f \equiv C \ (\text{fixp-fun } (\lambda f. U \ (F \ (C \ f))))$

```

and inverse:  $\bigwedge f. U (C f) = f$ 
and adm: ccpo.admissible lub-fun le-fun P
and bot:  $P (\lambda-. \text{lub } \{\})$ 
and step:  $\bigwedge f'. \llbracket P (U f'); \text{le-fun } (U f') (U f) \rrbracket \implies P (U (F f'))$ 
shows  $P (U f)$ 
unfolding eq inverse
apply (rule ccpo.fixp-strong-induct[OF ccpo adm])
  apply (insert mono, auto simp: monotone-def fun-ord-def bot fun-lub-def)[2]
apply (rule-tac f'5=C x in step)
  apply (simp-all add: inverse eq)
done

end

```

15.3 (=) as order

definition *lub-singleton* :: $('a \text{ set} \Rightarrow 'a) \Rightarrow \text{bool}$
where *lub-singleton* *lub* $\longleftrightarrow (\forall a. \text{lub } \{a\} = a)$

definition *the-Sup* :: $'a \text{ set} \Rightarrow 'a$
where *the-Sup* *A* = (*THE* *a. a* $\in A$)

lemma *lub-singleton-the-Sup* [*cont-intro, simp*]: *lub-singleton the-Sup*
by(*simp add: lub-singleton-def the-Sup-def*)

lemma (**in** *ccpo*) *lub-singleton: lub-singleton Sup*
by(*simp add: lub-singleton-def*)

lemma (**in** *partial-function-definitions*) *lub-singleton* [*cont-intro, simp*]: *lub-singleton lub*
by(*rule ccpo.lub-singleton*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemma *preorder-eq* [*cont-intro, simp*]:
class.preorder (=) (*mk-less* (=))
by(*unfold-locales*)(*simp-all add: mk-less-def*)

lemma *monotone-eqI* [*cont-intro*]:
assumes *class.preorder ord* (*mk-less ord*)
shows *monotone* (=) *ord f*

proof –
interpret *preorder ord mk-less ord* **by** *fact*
show ?thesis **by**(*simp add: monotone-def*)
qed

lemma *cont-eqI* [*cont-intro*]:
fixes *f* :: $'a \Rightarrow 'b$
assumes *lub-singleton lub*
shows *cont the-Sup* (=) *lub ord f*
proof(*rule contI*)

```

fix Y :: 'a set
assume Complete-Partial-Order.chain (=) Y Y ≠ {}
then obtain a where Y = {a} by(auto simp add: chain-def)
thus f (the-Sup Y) = lub (f ' Y) using assms
by(simp add: the-Sup-def lub-singleton-def)
qed

```

```

lemma mcont-eqI [cont-intro, simp]:
  [| class.preorder ord (mk-less ord); lub-singleton lub |]
  ⇒ mcont the-Sup (=) lub ord f
by(simp add: mcont-def cont-eqI monotone-eqI)

```

15.4 ccpo for products

```

definition prod-lub :: ('a set ⇒ 'a) ⇒ ('b set ⇒ 'b) ⇒ ('a × 'b) set ⇒ 'a × 'b
where prod-lub Sup-a Sup-b Y = (Sup-a (fst ' Y), Sup-b (snd ' Y))

```

```

lemma lub-singleton-prod-lub [cont-intro, simp]:
  [| lub-singleton luba; lub-singleton lubb |] ⇒ lub-singleton (prod-lub luba lubb)
by(simp add: lub-singleton-def prod-lub-def)

```

```

lemma prod-lub-empty [simp]: prod-lub luba lubb {} = (luba {}, lubb {})
by(simp add: prod-lub-def)

```

```

lemma preorder-rel-prodI [cont-intro, simp]:
  assumes class.preorder orda (mk-less orda)
  and class.preorder ordb (mk-less ordb)
  shows class.preorder (rel-prod orda ordb) (mk-less (rel-prod orda ordb))
proof –
  interpret a: preorder orda mk-less orda by fact
  interpret b: preorder ordb mk-less ordb by fact
  show ?thesis by(unfold-locales)(auto simp add: mk-less-def intro: a.order-trans
    b.order-trans)
qed

```

```

lemma order-rel-prodI:
  assumes a: class.order orda (mk-less orda)
  and b: class.order ordb (mk-less ordb)
  shows class.order (rel-prod orda ordb) (mk-less (rel-prod orda ordb))
  (is class.order ?ord ?ord')
proof(intro class.order.intro class.order-axioms.intro)
  interpret a: order orda mk-less orda by(fact a)
  interpret b: order ordb mk-less ordb by(fact b)
  show class.preorder ?ord ?ord' by(rule preorder-rel-prodI) unfold-locales

```

```

fix x y
assume ?ord x y ?ord y x
thus x = y by(cases x y rule: prod.exhaust[case-product prod.exhaust]) auto
qed

```

lemma *monotone-rel-prodI*:
 assumes *mono2*: $\bigwedge a. \text{monotone } \text{ordb } \text{ordc } (\lambda b. f \ (a, b))$
 and *mono1*: $\bigwedge b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f \ (a, b))$
 and *a*: *class.preorder* *orda* (*mk-less* *orda*)
 and *b*: *class.preorder* *ordb* (*mk-less* *ordb*)
 and *c*: *class.preorder* *ordc* (*mk-less* *ordc*)
 shows *monotone* (*rel-prod* *orda* *ordb*) *ordc* *f*
proof –
 interpret *a*: *preorder* *orda* *mk-less* *orda* **by**(*rule* *a*)
 interpret *b*: *preorder* *ordb* *mk-less* *ordb* **by**(*rule* *b*)
 interpret *c*: *preorder* *ordc* *mk-less* *ordc* **by**(*rule* *c*)
 show ?thesis **using** *mono2* *mono1*
 by(*auto* ? 2 *simp* *add*: *monotone-def* *intro*: *c.order-trans*)
qed

lemma *monotone-rel-prodD1*:
 assumes *mono*: *monotone* (*rel-prod* *orda* *ordb*) *ordc* *f*
 and *preorder*: *class.preorder* *ordb* (*mk-less* *ordb*)
 shows *monotone* *orda* *ordc* ($\lambda a. f \ (a, b)$)
proof –
 interpret *preorder* *ordb* *mk-less* *ordb* **by**(*rule* *preorder*)
 show ?thesis **using** *mono* **by**(*simp* *add*: *monotone-def*)
qed

lemma *monotone-rel-prodD2*:
 assumes *mono*: *monotone* (*rel-prod* *orda* *ordb*) *ordc* *f*
 and *preorder*: *class.preorder* *orda* (*mk-less* *orda*)
 shows *monotone* *ordb* *ordc* ($\lambda b. f \ (a, b)$)
proof –
 interpret *preorder* *orda* *mk-less* *orda* **by**(*rule* *preorder*)
 show ?thesis **using** *mono* **by**(*simp* *add*: *monotone-def*)
qed

lemma *monotone-case-prodI*:
 $\llbracket \bigwedge a. \text{monotone } \text{ordb } \text{ordc } (f \ a); \bigwedge b. \text{monotone } \text{orda } \text{ordc } (\lambda a. f \ a \ b);$
 class.preorder *orda* (*mk-less* *orda*); *class.preorder* *ordb* (*mk-less* *ordb*);
 class.preorder *ordc* (*mk-less* *ordc*) \rrbracket
 $\implies \text{monotone } (\text{rel-prod } \text{orda } \text{ordb}) \text{ordc } (\text{case-prod } f)$
by(*rule* *monotone-rel-prodI*) *simp-all*

lemma *monotone-case-prodD1*:
 assumes *mono*: *monotone* (*rel-prod* *orda* *ordb*) *ordc* (*case-prod* *f*)
 and *preorder*: *class.preorder* *ordb* (*mk-less* *ordb*)
 shows *monotone* *orda* *ordc* ($\lambda a. f \ a \ b$)
using *monotone-rel-prodD1* [*OF* *assms*] **by** *simp*

lemma *monotone-case-prodD2*:
 assumes *mono*: *monotone* (*rel-prod* *orda* *ordb*) *ordc* (*case-prod* *f*)

```

    and preorder: class.preorder orda (mk-less orda)
    shows monotone ordb ordc (f a)
    using monotone-rel-prodD2[OF assms] by simp

context
  fixes orda ordb ordc
  assumes a: class.preorder orda (mk-less orda)
    and b: class.preorder ordb (mk-less ordb)
    and c: class.preorder ordc (mk-less ordc)
begin

lemma monotone-rel-prod-iff:
  monotone (rel-prod orda ordb) ordc f  $\longleftrightarrow$ 
    ( $\forall a. \text{monotone ordb ordc } (\lambda b. f (a, b))$ )  $\wedge$ 
    ( $\forall b. \text{monotone orda ordc } (\lambda a. f (a, b))$ )
  using a b c by (blast intro: monotone-rel-prodI dest: monotone-rel-prodD1 mono-
    tone-rel-prodD2)

lemma monotone-case-prod-iff [simp]:
  monotone (rel-prod orda ordb) ordc (case-prod f)  $\longleftrightarrow$ 
    ( $\forall a. \text{monotone ordb ordc } (f a)$ )  $\wedge$  ( $\forall b. \text{monotone orda ordc } (\lambda a. f a b)$ )
  by (simp add: monotone-rel-prod-iff)

end

lemma monotone-case-prod-apply-iff:
  monotone orda ordb ( $\lambda x. (\text{case-prod } f x) y$ )  $\longleftrightarrow$  monotone orda ordb ( $\text{case-prod}$ 
    ( $\lambda a b. f a b y$ ))
  by (simp add: monotone-def)

lemma monotone-case-prod-applyD:
  monotone orda ordb ( $\lambda x. (\text{case-prod } f x) y$ )
 $\implies$  monotone orda ordb ( $\text{case-prod } (\lambda a b. f a b y)$ )
  by (simp add: monotone-case-prod-apply-iff)

lemma monotone-case-prod-applyI:
  monotone orda ordb ( $\text{case-prod } (\lambda a b. f a b y)$ )
 $\implies$  monotone orda ordb ( $\lambda x. (\text{case-prod } f x) y$ )
  by (simp add: monotone-case-prod-apply-iff)

lemma cont-case-prod-apply-iff:
  cont luba orda lubb ordb ( $\lambda x. (\text{case-prod } f x) y$ )  $\longleftrightarrow$  cont luba orda lubb ordb
    ( $\text{case-prod } (\lambda a b. f a b y)$ )
  by (simp add: cont-def split-def)

lemma cont-case-prod-applyI:
  cont luba orda lubb ordb ( $\text{case-prod } (\lambda a b. f a b y)$ )
 $\implies$  cont luba orda lubb ordb ( $\lambda x. (\text{case-prod } f x) y$ )

```

by(simp add: cont-case-prod-apply-iff)

lemma cont-case-prod-applyD:

cont luba orda lubb ordb $(\lambda x. (case-prod f x) y)$
 \implies cont luba orda lubb ordb $(case-prod (\lambda a b. f a b y))$
by(simp add: cont-case-prod-apply-iff)

lemma mcont-case-prod-apply-iff [simp]:

mcont luba orda lubb ordb $(\lambda x. (case-prod f x) y) \longleftrightarrow$
mcont luba orda lubb ordb $(case-prod (\lambda a b. f a b y))$
by(simp add: mcont-def monotone-case-prod-apply-iff cont-case-prod-apply-iff)

lemma cont-prodD1:

assumes cont: cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc f
and class.preorder orda (mk-less orda)
and luba: lub-singleton luba
shows cont lubb ordb lubc ordc $(\lambda y. f (x, y))$
proof(rule contI)
interpret preorder orda mk-less orda **by** fact

fix Y :: 'b set
let ?Y = {x} \times Y
assume Complete-Partial-Order.chain ordb Y Y $\neq \{\}$
hence Complete-Partial-Order.chain (rel-prod orda ordb) ?Y ?Y $\neq \{\}$
by(simp-all add: chain-def)
with cont **have** f (prod-lub luba lubb ?Y) = lubc (f ' ?Y) **by**(rule contD)
moreover **have** f ' ?Y = $(\lambda y. f (x, y))$ ' Y **by** auto
ultimately show f (x, lubb Y) = lubc $((\lambda y. f (x, y))$ ' Y) **using** luba
by(simp add: prod-lub-def 'Y $\neq \{\}$ lub-singleton-def)
qed

lemma cont-prodD2:

assumes cont: cont (prod-lub luba lubb) (rel-prod orda ordb) lubc ordc f
and class.preorder ordb (mk-less ordb)
and lubb: lub-singleton lubb
shows cont luba orda lubc ordc $(\lambda x. f (x, y))$
proof(rule contI)
interpret preorder ordb mk-less ordb **by** fact

fix Y
assume Y: Complete-Partial-Order.chain orda Y Y $\neq \{\}$
let ?Y = Y \times {y}
have f (luba Y, y) = f (prod-lub luba lubb ?Y)
using lubb **by**(simp add: prod-lub-def Y lub-singleton-def)
also from Y **have** Complete-Partial-Order.chain (rel-prod orda ordb) ?Y ?Y $\neq \{\}$
by(simp-all add: chain-def)
with cont **have** f (prod-lub luba lubb ?Y) = lubc (f ' ?Y) **by**(rule contD)
also have f ' ?Y = $(\lambda x. f (x, y))$ ' Y **by** auto

finally show $f \text{ (luba } Y, y) = \text{lubc } \dots$.
qed

lemma *cont-case-prodD1*:
 assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordc* (*case-prod f*)
 and *class.preorder orda* (*mk-less orda*)
 and *lub-singleton luba*
 shows *cont lubb ordb lubc ordc* (*f x*)
using *cont-prodD1*[*OF assms*] **by** *simp*

lemma *cont-case-prodD2*:
 assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordc* (*case-prod f*)
 and *class.preorder ordb* (*mk-less ordb*)
 and *lub-singleton lubb*
 shows *cont luba orda lubc ordc* ($\lambda x. f x y$)
using *cont-prodD2*[*OF assms*] **by** *simp*

context *ccpo* **begin**

lemma *cont-prodI*:
 assumes *mono*: *monotone* (*rel-prod orda ordb*) (\leq) *f*
 and *cont1*: $\bigwedge x. \text{cont lubb ordb } \text{Sup } (\leq) (\lambda y. f (x, y))$
 and *cont2*: $\bigwedge y. \text{cont luba orda } \text{Sup } (\leq) (\lambda x. f (x, y))$
 and *class.preorder orda* (*mk-less orda*)
 and *class.preorder ordb* (*mk-less ordb*)
 shows *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup* (\leq) *f*
proof(*rule contI*)
 interpret *a*: *preorder orda mk-less orda* **by** *fact*
 interpret *b*: *preorder ordb mk-less ordb* **by** *fact*

 fix *Y*
 assume *chain*: *Complete-Partial-Order.chain* (*rel-prod orda ordb*) *Y*
 and $Y \neq \{\}$
 have $f \text{ (prod-lub luba lubb } Y) = f \text{ (luba (fst ' } Y), \text{lubb (snd ' } Y))$
 by(*simp add: prod-lub-def*)
 also from *cont2* have $f \text{ (luba (fst ' } Y), \text{lubb (snd ' } Y)) = \bigsqcup ((\lambda x. f (x, \text{lubb (snd ' } Y))) \text{ ' } \text{fst ' } Y)$
 by(*rule contD*)(*simp-all add: chain-rel-prodD1*[*OF chain*] $\langle Y \neq \{\} \rangle$)
 also from *cont1* have $\bigwedge x. f (x, \text{lubb (snd ' } Y)) = \bigsqcup ((\lambda y. f (x, y)) \text{ ' } \text{snd ' } Y)$
 by(*rule contD*)(*simp-all add: chain-rel-prodD2*[*OF chain*] $\langle Y \neq \{\} \rangle$)
 hence $\bigsqcup ((\lambda x. f (x, \text{lubb (snd ' } Y))) \text{ ' } \text{fst ' } Y) = \bigsqcup ((\lambda x. \dots x) \text{ ' } \text{fst ' } Y)$ **by**
simp
 also have $\dots = \bigsqcup ((\lambda x. f (\text{fst } x, \text{snd } x)) \text{ ' } Y)$
 unfolding *image-image* **using** *chain*
proof (*rule diag-Sup*)
 show $\bigwedge y. y \in Y \implies \text{monotone} (\text{rel-prod orda ordb}) (\leq) (\lambda x. f (\text{fst } x, \text{snd } y))$
 by (*smt* (*verit*, *best*) *b.order-refl mono monotoneD monotoneI rel-prod-inject rel-prod-sel*)
qed (*use mono monotoneD in fastforce*)

finally show $f \text{ (prod-lub luba lubb } Y) = \bigsqcup (f \text{ ‘ } Y)$ **by** *simp*
qed

lemma *cont-case-prodI*:

assumes *monotone (rel-prod orda ordb) (\leq) (case-prod f)*
and $\bigwedge x. \text{ cont lubb ordb Sup } (\leq) (\lambda y. f \ x \ y)$
and $\bigwedge y. \text{ cont luba orda Sup } (\leq) (\lambda x. f \ x \ y)$
and *class.preorder orda (mk-less orda)*
and *class.preorder ordb (mk-less ordb)*
shows *cont (prod-lub luba lubb) (rel-prod orda ordb) Sup (\leq) (case-prod f)*
by(*rule cont-prodI*)(*simp-all add: asms*)

lemma *cont-case-prod-iff*:

$\llbracket \text{ monotone (rel-prod orda ordb) } (\leq) \text{ (case-prod f);}$
class.preorder orda (mk-less orda); lub-singleton luba;
*class.preorder ordb (mk-less ordb); lub-singleton lubb \rrbracket
 $\implies \text{ cont (prod-lub luba lubb) (rel-prod orda ordb) Sup } (\leq) \text{ (case-prod f) } \longleftrightarrow$
 $(\forall x. \text{ cont lubb ordb Sup } (\leq) (\lambda y. f \ x \ y)) \wedge (\forall y. \text{ cont luba orda Sup } (\leq) (\lambda x. f \ x \ y))$
by(*blast dest: cont-case-prodD1 cont-case-prodD2 intro: cont-case-prodI*)*

end

context *partial-function-definitions* **begin**

lemma *mono2mono2*:

assumes *f: monotone (rel-prod ordb ordc) leq ($\lambda(x, y). f \ x \ y$)*
and *t: monotone orda ordb ($\lambda x. t \ x$)*
and *t': monotone orda ordc ($\lambda x. t' \ x$)*
shows *monotone orda leq ($\lambda x. f \ (t \ x) \ (t' \ x)$)*
by (*metis (mono-tags, lifting) case-prod-conv monotoneD monotoneI rel-prod.intros asms*)

lemma *cont-case-prodI [cont-intro]*:

$\llbracket \text{ monotone (rel-prod orda ordb) leq (case-prod f);}$
 $\bigwedge x. \text{ cont lubb ordb lub leq } (\lambda y. f \ x \ y);$
 $\bigwedge y. \text{ cont luba orda lub leq } (\lambda x. f \ x \ y);$
class.preorder orda (mk-less orda);
*class.preorder ordb (mk-less ordb) \rrbracket
 $\implies \text{ cont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f)}$
by(*rule ccpo.cont-case-prodI*)(*rule Partial-Function.ccpo[OF partial-function-definitions-axioms]*)*

lemma *cont-case-prod-iff*:

$\llbracket \text{ monotone (rel-prod orda ordb) leq (case-prod f);}$
class.preorder orda (mk-less orda); lub-singleton luba;
*class.preorder ordb (mk-less ordb); lub-singleton lubb \rrbracket
 $\implies \text{ cont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f) } \longleftrightarrow$
 $(\forall x. \text{ cont lubb ordb lub leq } (\lambda y. f \ x \ y)) \wedge (\forall y. \text{ cont luba orda lub leq } (\lambda x. f \ x \ y))$
by(*blast dest: cont-case-prodD1 cont-case-prodD2 intro: cont-case-prodI*)*

lemma *mcont-case-prod-iff* [*simp*]:

$$\llbracket \text{class.preorder } \text{orda} \text{ (mk-less } \text{orda}); \text{ lub-singleton } \text{luba};$$

$$\text{class.preorder } \text{ordb} \text{ (mk-less } \text{ordb}); \text{ lub-singleton } \text{lubb} \rrbracket$$

$$\implies \text{mcont (prod-lub luba lubb) (rel-prod orda ordb) lub leq (case-prod f)} \longleftrightarrow$$

$$(\forall x. \text{mcont lubb ordb lub leq } (\lambda y. f x y)) \wedge (\forall y. \text{mcont luba orda lub leq } (\lambda x. f x y))$$
unfolding *mcont-def* **by**(*auto simp add: cont-case-prod-iff*)
end

lemma *mono2mono-case-prod* [*cont-intro*]:
assumes $\bigwedge x y. \text{monotone } \text{orda } \text{ordb} \text{ (}\lambda f. \text{pair } f x y\text{)}$
shows $\text{monotone } \text{orda } \text{ordb} \text{ (}\lambda f. \text{case-prod (pair } f\text{) } x\text{)}$
by(*rule monotoneI*)(*auto split: prod.split dest: monotoneD[OF assms]*)

15.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: *class.ccpo* *Sup* (\leq) ($<$)
by(*unfold-locales*)(*fast intro: Sup-upper Sup-least*)**+**

lemma *complete-lattice-ccpo'*: *class.ccpo* *Sup* (\leq) (*mk-less* (\leq))
by(*unfold-locales*)(*auto simp add: mk-less-def intro: Sup-upper Sup-least*)

lemma *complete-lattice-partial-function-definitions*:
partial-function-definitions (\leq) *Sup*
by(*unfold-locales*)(*auto intro: Sup-least Sup-upper*)

lemma *complete-lattice-partial-function-definitions-dual*:
partial-function-definitions (\geq) *Inf*
by(*unfold-locales*)(*auto intro: Inf-lower Inf-greatest*)

lemmas [*cont-intro, simp*] =
Partial-Function.ccpo[*OF complete-lattice-partial-function-definitions*]
Partial-Function.ccpo[*OF complete-lattice-partial-function-definitions-dual*]

lemma *mono2mono-inf*:
assumes $f: \text{monotone } \text{ord} \text{ (}\leq\text{) } (\lambda x. f x)$
and $g: \text{monotone } \text{ord} \text{ (}\leq\text{) } (\lambda x. g x)$
shows $\text{monotone } \text{ord} \text{ (}\leq\text{) } (\lambda x. f x \sqcap g x)$
by(*auto 4 3 dest: monotoneD[OF f] monotoneD[OF g] intro: le-infI1 le-infI2*)
intro!: monotoneI

lemma *mcont-const* [*simp*]: *mcont lub ord Sup* (\leq) ($\lambda \cdot. c$)
by(*rule ccpo.mcont-const*)[*OF complete-lattice-ccpo*]

lemma *mono2mono-sup*:

```

assumes  $f$ : monotone ord  $(\leq)$   $(\lambda x. f\ x)$ 
and  $g$ : monotone ord  $(\leq)$   $(\lambda x. g\ x)$ 
shows monotone ord  $(\leq)$   $(\lambda x. f\ x \sqcup g\ x)$ 
by(auto 4 3 intro!: monotoneI intro: sup.coboundedI1 sup.coboundedI2 dest: mono-
toneD[OF  $f$ ] monotoneD[OF  $g$ ])

```

lemma *Sup-image-sup*:

```

assumes  $Y \neq \{\}$ 
shows  $\bigsqcup ((\bigsqcup) x \in Y) = x \sqcup \bigsqcup Y$ 
proof(rule Sup-eqI)
  fix  $y$ 
  assume  $y \in (\bigsqcup) x \in Y$ 
  then obtain  $z$  where  $y = x \sqcup z$  and  $z \in Y$  by blast
  from  $\langle z \in Y \rangle$  have  $z \leq \bigsqcup Y$  by(rule Sup-upper)
  with - show  $y \leq x \sqcup \bigsqcup Y$  unfolding  $\langle y = x \sqcup z \rangle$  by(rule sup-mono) simp
next
  fix  $y$ 
  assume upper:  $\bigwedge z. z \in (\bigsqcup) x \in Y \implies z \leq y$ 
  show  $x \sqcup \bigsqcup Y \leq y$  unfolding Sup-insert[symmetric]
  proof(rule Sup-least)
    fix  $z$ 
    assume  $z \in \text{insert } x\ Y$ 
    from assms obtain  $z'$  where  $z' \in Y$  by blast
    let  $?z = \text{if } z \in Y \text{ then } x \sqcup z \text{ else } x \sqcup z'$ 
    have  $z \leq x \sqcup ?z$  using  $\langle z' \in Y \rangle$   $\langle z \in \text{insert } x\ Y \rangle$  by auto
    also have  $\dots \leq y$  by(rule upper)(auto split: if-split-asm intro:  $\langle z' \in Y \rangle$ )
    finally show  $z \leq y$  .
  qed
qed

```

lemma *mcont-sup1*: $mcont\ Sup (\leq)\ Sup (\leq) (\lambda y. x \sqcup y)$

```

by(auto 4 3 simp add: mcont-def sup.coboundedI1 sup.coboundedI2 intro!: mono-
toneI contI intro: Sup-image-sup[symmetric])

```

lemma *mcont-sup2*: $mcont\ Sup (\leq)\ Sup (\leq) (\lambda x. x \sqcup y)$

```

by(subst sup-commute)(rule mcont-sup1)

```

lemma *mcont2mcont-sup* [*cont-intro*, *simp*]:

```

 $\llbracket mcont\ lub\ ord\ Sup (\leq) (\lambda x. f\ x);$ 
 $mcont\ lub\ ord\ Sup (\leq) (\lambda x. g\ x) \rrbracket$ 
 $\implies mcont\ lub\ ord\ Sup (\leq) (\lambda x. f\ x \sqcup g\ x)$ 
by(best intro: ccpo.mcont2mcont'[OF complete-lattice-ccpo] mcont-sup1 mcont-sup2
ccpo.mcont-const[OF complete-lattice-ccpo])

```

end

lemmas [*cont-intro*] = *admissible-leI*[OF complete-lattice-ccpo]

context complete-distrib-lattice **begin**

lemma *mcont-inf1*: *mcont* *Sup* (\leq) *Sup* (\leq) ($\lambda y. x \sqcap y$)
by(*auto* *intro*: *monotoneI* *contI* *simp* *add*: *le-infI2* *inf-Sup* *mcont-def*)

lemma *mcont-inf2*: *mcont* *Sup* (\leq) *Sup* (\leq) ($\lambda x. x \sqcap y$)
by(*auto* *intro*: *monotoneI* *contI* *simp* *add*: *le-infI1* *Sup-inf* *mcont-def*)

lemma *mcont2mcont-inf* [*cont-intro*, *simp*]:
 \llbracket *mcont* *lub* *ord* *Sup* (\leq) ($\lambda x. f\ x$);
mcont *lub* *ord* *Sup* (\leq) ($\lambda x. g\ x$) \rrbracket
 \implies *mcont* *lub* *ord* *Sup* (\leq) ($\lambda x. f\ x \sqcap g\ x$)
by(*best* *intro*: *ccpo.mcont2mcont*'[*OF* *complete-lattice-ccpo*] *mcont-inf1* *mcont-inf2*
ccpo.mcont-const[*OF* *complete-lattice-ccpo*])

end

interpretation *lfp*: *partial-function-definitions* (\leq) :: - :: *complete-lattice* \Rightarrow - *Sup*
by(*rule* *complete-lattice-partial-function-definitions*)

declaration \langle *Partial-Function.init* *lfp* **term** \langle *lfp.fixp-fun* \rangle **term** \langle *lfp.mono-body* \rangle
 $\textcircled{\text{thm}}$ *lfp.fixp-rule-uc* $\textcircled{\text{thm}}$ *lfp.fixp-induct-uc* *NONE* \rangle

interpretation *gfp*: *partial-function-definitions* (\geq) :: - :: *complete-lattice* \Rightarrow - *Inf*
by(*rule* *complete-lattice-partial-function-definitions-dual*)

declaration \langle *Partial-Function.init* *gfp* **term** \langle *gfp.fixp-fun* \rangle **term** \langle *gfp.mono-body* \rangle
 $\textcircled{\text{thm}}$ *gfp.fixp-rule-uc* $\textcircled{\text{thm}}$ *gfp.fixp-induct-uc* *NONE* \rangle

lemma *insert-mono* [*partial-function-mono*]:
monotone (*fun-ord* (\subseteq)) (\subseteq) *A* \implies *monotone* (*fun-ord* (\subseteq)) (\subseteq) ($\lambda y. \text{insert } x\ (A\ y)$)
by(*rule* *monotoneI*)(*auto* *simp* *add*: *fun-ord-def* *dest*: *monotoneD*)

lemma *mono2mono-insert* [*THEN* *lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-insert*: *monotone* (\subseteq) (\subseteq) (*insert* *x*)
by(*rule* *monotoneI*) *blast*

lemma *mcont2mcont-insert*[*THEN* *lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-insert*: *mcont* *Union* (\subseteq) *Union* (\subseteq) (*insert* *x*)
by(*blast* *intro*: *mcontI* *contI* *monotone-insert*)

lemma *mono2mono-image* [*THEN* *lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-image*: *monotone* (\subseteq) (\subseteq) ($((\cdot) f)$)
by (*simp* *add*: *image-mono* *monoI*)

lemma *cont-image*: *cont* *Union* (\subseteq) *Union* (\subseteq) ($((\cdot) f)$)
by (*meson* *contI* *image-Union*)

lemma *mcont2mcont-image* [*THEN* *lfp.mcont2mcont*, *cont-intro*, *simp*]:

shows *mcont-image*: *mcont Union* (\subseteq) *Union* (\subseteq) ((\cdot) *f*)
by(*blast intro: mcontI monotone-image cont-image*)

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro, simp*]:
 $\text{monotone ord } (\subseteq) f \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup f x)$
by(*blast intro: monotoneI Sup-least Sup-upper dest: monotoneD*)

lemma *cont-Sup*:
assumes *cont lub ord Union* (\subseteq) *f*
shows *cont lub ord Sup* (\leq) ($\lambda x. \bigsqcup f x$)
proof –
have $\bigwedge Y. \llbracket \text{Complete-Partial-Order.chain ord } Y; Y \neq \{\} \rrbracket$
 $\implies \bigsqcup \bigcup (f \cdot Y) = (\bigsqcup_{x \in Y. \bigsqcup f x})$
by (*blast intro: Sup-least Sup-upper order-trans order.antisym*)
with *assms show ?thesis*
by (*force simp: cont-def*)
qed

lemma *mcont-Sup*: *mcont lub ord Union* (\subseteq) *f* \implies *mcont lub ord Sup* (\leq) ($\lambda x. \bigsqcup f x$)
unfolding *mcont-def* **by**(*blast intro: monotone-Sup cont-Sup*)

lemma *monotone-SUP*:
 $\llbracket \text{monotone ord } (\subseteq) f; \bigwedge y. \text{monotone ord } (\leq) (\lambda x. g x y) \rrbracket \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup_{y \in f x.} g x y)$
by(*rule monotoneI*)(*blast dest: monotoneD intro: Sup-upper order-trans intro!: Sup-least*)

lemma *monotone-SUP2*:
 $(\bigwedge y. y \in A \implies \text{monotone ord } (\leq) (\lambda x. g x y)) \implies \text{monotone ord } (\leq) (\lambda x. \bigsqcup_{y \in A.} g x y)$
by(*rule monotoneI*)(*blast intro: Sup-upper order-trans dest: monotoneD intro!: Sup-least*)

lemma *cont-SUP*:
assumes *f: mcont lub ord Union* (\subseteq) *f*
and *g: $\bigwedge y. mcont lub ord Sup$* (\leq) ($\lambda x. g x y$)
shows *cont lub ord Sup* (\leq) ($\lambda x. \bigsqcup_{y \in f x.} g x y$)
proof(*rule contI*)
fix *Y*
assume *chain: Complete-Partial-Order.chain ord Y*
and *Y: Y* $\neq \{\}$
show $\bigsqcup (g (\text{lub } Y) \cdot f (\text{lub } Y)) = \bigsqcup ((\lambda x. \bigsqcup (g x \cdot f x)) \cdot Y)$ (*is ?lhs = ?rhs*)
proof(*rule order.antisym*)
show *?lhs* \leq *?rhs*
proof(*rule Sup-least*)
fix *x*

```

assume  $x \in g \text{ (lub } Y) \text{ ' } f \text{ (lub } Y)$ 
with  $mcont\text{-}contD[OF \text{ } f \text{ chain } Y] \text{ } mcont\text{-}contD[OF \text{ } g \text{ chain } Y]$ 
obtain  $y \text{ } z$  where  $y \in Y \text{ } z \in f \text{ } y$ 
  and  $x: x = \bigsqcup ((\lambda x. g \text{ } x \text{ } z) \text{ ' } Y)$  by auto
show  $x \leq ?rhs$  unfolding  $x$ 
proof(rule Sup-least)
  fix  $u$ 
  assume  $u \in (\lambda x. g \text{ } x \text{ } z) \text{ ' } Y$ 
  then obtain  $y' \text{ } z'$  where  $u = g \text{ } y' \text{ } z' \text{ } y' \in Y$  by auto
  from  $chain \langle y \in Y \rangle \langle y' \in Y \rangle$  have  $ord \text{ } y \text{ } y' \vee ord \text{ } y' \text{ } y$  by(rule chainD)
  thus  $u \leq ?rhs$ 
  proof
    note  $\langle u = g \text{ } y' \text{ } z' \rangle$  also
    assume  $ord \text{ } y \text{ } y'$ 
    with  $f$  have  $f \text{ } y \subseteq f \text{ } y'$  by(rule mcont-monoD)
    with  $\langle z \in f \text{ } y \rangle$ 
    have  $g \text{ } y' \text{ } z' \leq \bigsqcup (g \text{ } y' \text{ ' } f \text{ } y')$  by(auto intro: Sup-upper)
    also have  $\dots \leq ?rhs$  using  $\langle y' \in Y \rangle$  by(auto intro: Sup-upper)
    finally show  $?thesis$  .
  next
    note  $\langle u = g \text{ } y' \text{ } z' \rangle$  also
    assume  $ord \text{ } y' \text{ } y$ 
    with  $g$  have  $g \text{ } y' \text{ } z' \leq g \text{ } y \text{ } z$  by(rule mcont-monoD)
    also have  $\dots \leq \bigsqcup (g \text{ } y \text{ ' } f \text{ } y)$  using  $\langle z \in f \text{ } y \rangle$ 
    by(auto intro: Sup-upper)
    also have  $\dots \leq ?rhs$  using  $\langle y \in Y \rangle$  by(auto intro: Sup-upper)
    finally show  $?thesis$  .
  qed
qed
qed
next
  show  $?rhs \leq ?lhs$ 
  proof(rule Sup-least)
    fix  $x$ 
    assume  $x \in (\lambda x. \bigsqcup (g \text{ } x \text{ ' } f \text{ } x)) \text{ ' } Y$ 
    then obtain  $y$  where  $x: x = \bigsqcup (g \text{ } y \text{ ' } f \text{ } y)$  and  $y \in Y$  by auto
    show  $x \leq ?lhs$  unfolding  $x$ 
    proof(rule Sup-least)
      fix  $u$ 
      assume  $u \in g \text{ } y \text{ ' } f \text{ } y$ 
      then obtain  $z$  where  $u = g \text{ } y \text{ } z \text{ } z \in f \text{ } y$  by auto
      note  $\langle u = g \text{ } y \text{ } z \rangle$ 
      also have  $g \text{ } y \text{ } z \leq \bigsqcup ((\lambda x. g \text{ } x \text{ } z) \text{ ' } Y)$ 
      using  $\langle y \in Y \rangle$  by(auto intro: Sup-upper)
      also have  $\dots = g \text{ (lub } Y) \text{ } z$  by(simp add: mcont-contD[OF g chain Y])
      also have  $\dots \leq ?lhs$  using  $\langle z \in f \text{ } y \rangle \langle y \in Y \rangle$ 
      by(auto intro: Sup-upper simp add: mcont-contD[OF f chain Y])
      finally show  $u \leq ?lhs$  .
    qed
  qed

```

qed
qed
qed

lemma *mcont-SUP* [*cont-intro*, *simp*]:
 $\llbracket \text{mcont lub ord Union } (\subseteq) f; \bigwedge y. \text{mcont lub ord Sup } (\leq) (\lambda x. g x y) \rrbracket$
 $\implies \text{mcont lub ord Sup } (\leq) (\lambda x. \bigsqcup_{y \in f x} g x y)$
by(*blast intro: mcontI cont-SUP monotone-SUP mcont-mono*)
end

lemma *admissible-Ball* [*cont-intro*, *simp*]:
 $\llbracket \bigwedge x. \text{ccpo.admissible lub ord } (\lambda A. P A x);$
 $\text{mcont lub ord Union } (\subseteq) f;$
 $\text{class.ccpo lub ord (mk-less ord)} \rrbracket$
 $\implies \text{ccpo.admissible lub ord } (\lambda A. \forall x \in f A. P A x)$
unfolding *Ball-def* **by** *simp*

lemma *admissible-Bex'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-Bex*: $\text{ccpo.admissible Union } (\subseteq) (\lambda A. \exists x \in A. P x)$
using *ccpo.admissible-def* **by** *fastforce*

15.6 Parallel fixpoint induction

context
fixes *luba* :: 'a set \Rightarrow 'a
and *orda* :: 'a \Rightarrow 'a \Rightarrow bool
and *lubb* :: 'b set \Rightarrow 'b
and *ordb* :: 'b \Rightarrow 'b \Rightarrow bool
assumes *a*: *class.ccpo luba orda (mk-less orda)*
and *b*: *class.ccpo lubb ordb (mk-less ordb)*
begin

interpretation *a*: *ccpo luba orda mk-less orda* **by**(*rule a*)
interpretation *b*: *ccpo lubb ordb mk-less ordb* **by**(*rule b*)

lemma *ccpo-rel-prodI*:
 $\text{class.ccpo (prod-lub luba lubb) (rel-prod orda ordb) (mk-less (rel-prod orda ordb))}$
 $(\text{is class.ccpo ?lub ?ord ?ord'})$
proof(*intro class.ccpo.intro class.ccpo-axioms.intro*)
show *class.order ?ord ?ord'*
by(*rule order-rel-prodI*) *intro-locales*
show $\bigwedge A x. \llbracket \text{Complete-Partial-Order.chain (rel-prod orda ordb) } A; x \in A \rrbracket$
 $\implies \text{rel-prod orda ordb } x (\text{prod-lub luba lubb } A)$
by (*simp add: a.ccpo-Sup-upper b.ccpo-Sup-upper chain-rel-prodD1 chain-rel-prodD2*
prod-lub-def rel-prod-sel)
show $\bigwedge A z. \llbracket \text{Complete-Partial-Order.chain (rel-prod orda ordb) } A;$
 $\bigwedge x. x \in A \implies \text{rel-prod orda ordb } x z \rrbracket$
 $\implies \text{rel-prod orda ordb (prod-lub luba lubb } A) z$

```

by (metis (full-types) a.ccpo-Sup-below-iff b.ccpo-Sup-least chain-rel-prodD1
    chain-rel-prodD2 imageE prod.sel prod-lub-def rel-prod-sel)
qed

interpretation ab: ccpo prod-lub luba lubb rel-prod orda ordb mk-less (rel-prod orda
    ordb)
by(rule ccpo-rel-prodI)

lemma monotone-map-prod [simp]:
    monotone (rel-prod orda ordb) (rel-prod ordc ordd) (map-prod f g)  $\longleftrightarrow$ 
    monotone orda ordc f  $\wedge$  monotone ordb ordd g
by(auto simp add: monotone-def)

lemma parallel-fixp-induct:
    assumes adm: ccpo.admissible (prod-lub luba lubb) (rel-prod orda ordb) ( $\lambda x. P$ 
    (fst x) (snd x))
    and f: monotone orda orda f
    and g: monotone ordb ordb g
    and bot:  $P$  (luba {}) (lubb {})
    and step:  $\bigwedge x y. P x y \implies P (f x) (g y)$ 
    shows  $P$  (ccpo.fixp luba orda f) (ccpo.fixp lubb ordb g)
proof –
    let ?lub = prod-lub luba lubb
    and ?ord = rel-prod orda ordb
    and ?P =  $\lambda(x, y). P x y$ 
    from adm have adm': ccpo.admissible ?lub ?ord ?P by(simp add: split-def)
    hence ?P (ccpo.fixp (prod-lub luba lubb) (rel-prod orda ordb) (map-prod f g))
    by(rule ab.fixp-induct)(auto simp add: f g step bot)
    also have ccpo.fixp (prod-lub luba lubb) (rel-prod orda ordb) (map-prod f g) =
    (ccpo.fixp luba orda f, ccpo.fixp lubb ordb g) is ?lhs = (?rhs1, ?rhs2))
    proof(rule ab.order.antisym)
    have ccpo.admissible ?lub ?ord ( $\lambda xy. ?ord xy$  (?rhs1, ?rhs2))
    by(rule admissible-leI[OF ccpo-rel-prodI])(auto simp add: prod-lub-def chain-empty)
    intro: a.ccpo-Sup-least b.ccpo-Sup-least
    thus ?ord ?lhs (?rhs1, ?rhs2)
    by(rule ab.fixp-induct)(auto 4 3 dest: monotoneD[OF f] monotoneD[OF g]
    simp add: b.fixp-unfold[OF g, symmetric] a.fixp-unfold[OF f, symmetric] f g intro:
    a.ccpo-Sup-least b.ccpo-Sup-least chain-empty)
    next
    have ccpo.admissible luba orda ( $\lambda x. orda x$  (fst ?lhs))
    by(rule admissible-leI[OF a])(auto intro: a.ccpo-Sup-least simp add: chain-empty)
    hence orda ?rhs1 (fst ?lhs) using f
    proof(rule a.fixp-induct)
    fix x
    assume orda x (fst ?lhs)
    thus orda (f x) (fst ?lhs)
    by(subst ab.fixp-unfold)(auto simp add: f g dest: monotoneD[OF f])
    qed(auto intro: a.ccpo-Sup-least chain-empty)
moreover

```



```

have ccpo.admissible lubb ordb ( $\lambda y. \text{ordb } y \text{ (snd ?lhs)}$ )
by(rule admissible-leI[OF b])(auto intro: b.ccpo-Sup-least simp add: chain-empty)
hence ordb ?rhs2 (snd ?lhs) using g
proof(rule b.fixp-induct)
  fix y
  assume ordb y (snd ?lhs)
  thus ordb (g y) (snd ?lhs)
    by (smt (verit, best) ab.fixp-unfold f g monotoneD monotone-map-prod
        snd-map-prod)
qed(auto intro: b.ccpo-Sup-least chain-empty)
ultimately show ?ord (?rhs1, ?rhs2) ?lhs
  by(simp add: rel-prod-conv split-beta)
qed
finally show ?thesis by simp
qed
end

```

lemma *parallel-fixp-induct-uc*:

```

assumes a: partial-function-definitions orda luba
and b: partial-function-definitions ordb lubb
and F:  $\bigwedge x. \text{monotone (fun-ord orda) orda } (\lambda f. U1 \text{ (F (C1 f)) } x)$ 
and G:  $\bigwedge y. \text{monotone (fun-ord ordb) ordb } (\lambda g. U2 \text{ (G (C2 g)) } y)$ 
and eq1:  $f \equiv C1 \text{ (ccpo.fixp (fun-lub luba) (fun-ord orda) } (\lambda f. U1 \text{ (F (C1 f)) })))$ 
and eq2:  $g \equiv C2 \text{ (ccpo.fixp (fun-lub lubb) (fun-ord ordb) } (\lambda g. U2 \text{ (G (C2 g)) })))$ 
and inverse:  $\bigwedge f. U1 \text{ (C1 f) } = f$ 
and inverse2:  $\bigwedge g. U2 \text{ (C2 g) } = g$ 
and adm: ccpo.admissible (prod-lub (fun-lub luba) (fun-lub lubb)) (rel-prod (fun-ord
orda) (fun-ord ordb)) ( $\lambda x. P \text{ (fst x) (snd x)}$ )
and bot:  $P \text{ (}\lambda-. \text{ luba } \{\}) \text{ (}\lambda-. \text{ lubb } \{\})$ 
and step:  $\bigwedge f g. P \text{ (U1 f) (U2 g) } \implies P \text{ (U1 (F f)) (U2 (G g))}$ 
shows  $P \text{ (U1 f) (U2 g)}$ 
  unfolding eq1 eq2 inverse inverse2
proof (rule parallel-fixp-induct[OF partial-function-definitions.ccpo[OF a] partial-function-definitions.ccpo[OF b] adm])
  show monotone (fun-ord orda) (fun-ord orda) ( $\lambda f. U1 \text{ (F (C1 f)) }$ )
    monotone (fun-ord ordb) (fun-ord ordb) ( $\lambda g. U2 \text{ (G (C2 g)) }$ )
    using F G by(simp-all add: monotone-def fun-ord-def)
  show  $P \text{ (fun-lub luba } \{\}) \text{ (fun-lub lubb } \{\})}$ 
    by (simp add: fun-lub-def bot)
  show  $\bigwedge x y. P \text{ x y } \implies P \text{ (U1 (F (C1 x))) (U2 (G (C2 y)))}$ 
    by (simp add: inverse inverse2 local.step)
qed

```

lemmas *parallel-fixp-induct-1-1* = *parallel-fixp-induct-uc*[
 of - - - $\lambda x. x - \lambda x. x - \lambda x. x - \lambda x. x$,
 OF - - - - - *refl refl*]

lemmas *parallel-fixp-induct-2-2* = *parallel-fixp-induct-uc*[

of - - - case-prod - curry case-prod - curry,
where $P = \lambda f g. P \text{ (curry } f) \text{ (curry } g),$
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - refl refl]
for P

lemma *monotone-fst: monotone (rel-prod orda ordb) orda fst*
by(*auto intro: monotoneI*)

lemma *mcont-fst: mcont (prod-lub luba lubb) (rel-prod orda ordb) luba orda fst*
by(*auto intro!: mcontI monotoneI contI simp add: prod-lub-def*)

lemma *mcont2mcont-fst [cont-intro, simp]:*
mcont lub ord (prod-lub luba lubb) (rel-prod orda ordb) t
 $\implies mcont lub ord luba orda (\lambda x. fst (t x))$
by (*simp add: mcont-def monotone-on-def prod-lub-def cont-def image-image rel-prod-sel*)

lemma *monotone-snd: monotone (rel-prod orda ordb) ordb snd*
by(*auto intro: monotoneI*)

lemma *mcont-snd: mcont (prod-lub luba lubb) (rel-prod orda ordb) lubb ordb snd*
by(*auto intro!: mcontI monotoneI contI simp add: prod-lub-def*)

lemma *mcont2mcont-snd [cont-intro, simp]:*
mcont lub ord (prod-lub luba lubb) (rel-prod orda ordb) t
 $\implies mcont lub ord lubb ordb (\lambda x. snd (t x))$
by(*auto intro!: mcontI monotoneI contI dest: mcont-monoD mcont-contD simp add: rel-prod-sel split-beta prod-lub-def image-image*)

lemma *monotone-Pair:*
 $\llbracket monotone ord orda f; monotone ord ordb g \rrbracket$
 $\implies monotone ord (rel-prod orda ordb) (\lambda x. (f x, g x))$
by(*simp add: monotone-def*)

lemma *cont-Pair:*
 $\llbracket cont lub ord luba orda f; cont lub ord lubb ordb g \rrbracket$
 $\implies cont lub ord (prod-lub luba lubb) (rel-prod orda ordb) (\lambda x. (f x, g x))$
by(*rule contI*)(*auto simp add: prod-lub-def image-image dest!: contD*)

lemma *mcont-Pair:*
 $\llbracket mcont lub ord luba orda f; mcont lub ord lubb ordb g \rrbracket$
 $\implies mcont lub ord (prod-lub luba lubb) (rel-prod orda ordb) (\lambda x. (f x, g x))$
by(*rule mcontI*)(*simp-all add: monotone-Pair mcont-mono cont-Pair*)

context *partial-function-definitions*
begin

Specialised versions of *mcont-call* for admissibility proofs for parallel fixpoint inductions

```

lemmas mcont-call-fst [cont-intro] = mcont-call[THEN mcont2mcont, OF mcont-fst]
lemmas mcont-call-snd [cont-intro] = mcont-call[THEN mcont2mcont, OF mcont-snd]
end

```

```

lemma map-option-mono [partial-function-mono]:
  mono-option B  $\implies$  mono-option ( $\lambda f. \text{map-option } g \ (B \ f)$ )
unfolding map-conv-bind-option by(rule bind-mono) simp-all

```

```

lemma compact-flat-lub [cont-intro]: ccpo.compact (flat-lub x) (flat-ord x) y
using flat-interpretation[THEN ccpo]
proof(rule ccpo.compactI[OF - ccpo.admissibleI])
  fix A
  assume chain: Complete-Partial-Order.chain (flat-ord x) A
  and A: A  $\neq \{\}$ 
  and *:  $\forall z \in A. \neg \text{flat-ord } x \ y \ z$ 
  from A obtain z where z  $\in A$  by blast
  with * have z:  $\neg \text{flat-ord } x \ y \ z$  ..
  hence y:  $x \neq y \ y \neq z$  by(auto simp add: flat-ord-def)
  have y  $\neq (\text{THE } z. z \in A - \{x\})$  if  $\neg A \subseteq \{x\}$ 
  proof -
    from that obtain z' where z'  $\in A$  z'  $\neq x$  by auto
    then have (THE z. z  $\in A - \{x\}$ ) = z'
      by(intro the-equality)(auto dest: chainD[OF chain] simp add: flat-ord-def)
    moreover have z'  $\neq y$  using  $\langle z' \in A \rangle$  * by(auto simp add: flat-ord-def)
    ultimately show ?thesis by simp
  qed
  with z show  $\neg \text{flat-ord } x \ y \ (\text{flat-lub } x \ A)$ 
  by(simp add: flat-ord-def flat-lub-def)
qed
end

```

```

theory Conditional-Parametricity
imports Main
keywords parametric-constant :: thy-decl
begin

```

```

context includes lifting-syntax begin

```

```

qualified definition Rel-match :: ( $'a \Rightarrow 'b \Rightarrow \text{bool}$ )  $\Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$  where
  Rel-match R x y = R x y

```

```

named-theorems parametricity-preprocess

```

```

lemma bi-unique-Rel-match [parametricity-preprocess]:
  bi-unique A = Rel-match (A  $\implies\implies$  A  $\implies\implies$  (=)) (=) (=)
  unfolding bi-unique-alt-def2 Rel-match-def ..

```

lemma *bi-total-Rel-match* [parametricity-preprocess]:
bi-total $A = \text{Rel-match } ((A == => (=)) == => (=)) \text{ All All}$
unfolding *bi-total-alt-def2 Rel-match-def* ..

lemma *is-equality-Rel*: *is-equality* $A \implies \text{Transfer.Rel } A \text{ } t \text{ } t$
by (*fact transfer-raw*)

lemma *Rel-Rel-match*: $\text{Transfer.Rel } R \text{ } x \text{ } y \implies \text{Rel-match } R \text{ } x \text{ } y$
unfolding *Rel-match-def Rel-def* .

lemma *Rel-match-Rel*: $\text{Rel-match } R \text{ } x \text{ } y \implies \text{Transfer.Rel } R \text{ } x \text{ } y$
unfolding *Rel-match-def Rel-def* .

lemma *Rel-Rel-match-eq*: $\text{Transfer.Rel } R \text{ } x \text{ } y = \text{Rel-match } R \text{ } x \text{ } y$
using *Rel-Rel-match Rel-match-Rel* **by** *fast*

lemma *Rel-match-app*:
assumes $\text{Rel-match } (A == => B) \text{ } f \text{ } g$ **and** $\text{Transfer.Rel } A \text{ } x \text{ } y$
shows $\text{Rel-match } B \text{ } (f \text{ } x) \text{ } (g \text{ } y)$
using *assms Rel-match-Rel Rel-app Rel-Rel-match* **by** *fast*

end

ML-file $\langle \text{conditional-parametricity.ML} \rangle$

end

theory *Confluence* **imports**
Main
begin

16 Confluence

definition *semiconfluentp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
semiconfluentp $r \longleftrightarrow r^{-1-1} \text{ } OO \text{ } r^{**} \leq r^{**} \text{ } OO \text{ } r^{-1-1**}$

definition *confluentp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
confluentp $r \longleftrightarrow r^{-1-1**} \text{ } OO \text{ } r^{**} \leq r^{**} \text{ } OO \text{ } r^{-1-1**}$

definition *strong-confluentp* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
strong-confluentp $r \longleftrightarrow r^{-1-1} \text{ } OO \text{ } r \leq r^{**} \text{ } OO \text{ } (r^{-1-1})^{**}$

lemma *semiconfluentpI* [intro?]:
semiconfluentp r **if** $\bigwedge x \text{ } y \text{ } z. \llbracket r \text{ } x \text{ } y; r^{**} \text{ } x \text{ } z \rrbracket \implies \exists u. r^{**} \text{ } y \text{ } u \wedge r^{**} \text{ } z \text{ } u$
using *that unfolding semiconfluentp-def rtranclp-conversep* **by** *blast*

lemma *semiconfluentpD*: $\exists u. r^{**} \text{ } y \text{ } u \wedge r^{**} \text{ } z \text{ } u$ **if** *semiconfluentp* $r \text{ } r \text{ } x \text{ } y \text{ } r^{**} \text{ } x \text{ } z$
using *that unfolding semiconfluentp-def rtranclp-conversep* **by** *blast*

lemma *confluentpI*:

confluentp *r* **if** $\bigwedge x y z. \llbracket r^{**} x y; r^{**} x z \rrbracket \implies \exists u. r^{**} y u \wedge r^{**} z u$
using that unfolding *confluentp-def rtranclp-conversep* **by** *blast*

lemma *confluentpD*: $\exists u. r^{**} y u \wedge r^{**} z u$ **if** *confluentp* *r* $r^{**} x y r^{**} x z$
using that unfolding *confluentp-def rtranclp-conversep* **by** *blast*

lemma *strong-confluentpI* [intro?]:
strong-confluentp *r* **if** $\bigwedge x y z. \llbracket r x y; r x z \rrbracket \implies \exists u. r^{**} y u \wedge r^{==} z u$
using that unfolding *strong-confluentp-def* **by** *blast*

lemma *strong-confluentpD*: $\exists u. r^{**} y u \wedge r^{==} z u$ **if** *strong-confluentp* *r* *r x y r x z*
using that unfolding *strong-confluentp-def* **by** *blast*

lemma *semiconfluentp-imp-confluentp*: *confluentp* *r* **if** *r*: *semiconfluentp* *r*
proof(*rule confluentpI*)
show $\exists u. r^{**} y u \wedge r^{**} z u$ **if** $r^{**} x y r^{**} x z$ **for** *x y z*
using *that(2,1)*
by(*induction arbitrary: y rule: converse-rtranclp-induct*)
 (*blast intro: rtranclp-trans dest: r[THEN semiconfluentpD]*)+
qed

lemma *confluentp-imp-semiconfluentp*: *semiconfluentp* *r* **if** *confluentp* *r*
using that **by**(*auto intro!: semiconfluentpI dest: confluentpD[OF that]*)

lemma *confluentp-eq-semiconfluentp*: *confluentp* *r* \longleftrightarrow *semiconfluentp* *r*
by(*blast intro: semiconfluentp-imp-confluentp confluentp-imp-semiconfluentp*)

lemma *confluentp-conv-strong-confluentp-rtranclp*:
confluentp *r* \longleftrightarrow *strong-confluentp* (r^{**})
by(*auto simp add: confluentp-def strong-confluentp-def rtranclp-conversep*)

lemma *strong-confluentp-into-semiconfluentp*:
semiconfluentp *r* **if** *r*: *strong-confluentp* *r*
proof
show $\exists u. r^{**} y u \wedge r^{**} z u$ **if** $r x y r^{**} x z$ **for** *x y z*
using *that(2,1)*
apply(*induction arbitrary: y rule: converse-rtranclp-induct*)
subgoal by *blast*
subgoal for *a b c*
by (*drule (1) strong-confluentpD[OF r, of a c]*)(*auto 10 0 intro: rtranclp-trans*)
done
qed

lemma *strong-confluentp-imp-confluentp*: *confluentp* *r* **if** *strong-confluentp* *r*
unfolding *confluentp-eq-semiconfluentp* **using that** **by**(*rule strong-confluentp-into-semiconfluentp*)

lemma *semiconfluentp-equivclp*: *equivclp* *r* $= r^{**} \circ r^{-1-1^{**}}$ **if** *r*: *semiconfluentp* *r*

```

proof(rule antisym[rotated] r-OO-conversep-into-equivclp predicate2I)+
  show ( $r^{**} \text{ OO } r^{-1-1**}$ )  $x \ y$  if equivclp  $r \ x \ y$  for  $x \ y$  using that unfolding
equivclp-def rtranclp-conversep
  by(induction rule: converse-rtranclp-induct)
  (blast elim!: symclpE intro: converse-rtranclp-into-rtranclp rtranclp-trans dest:
semiconfluentpD[OF  $r$ ])+
qed

```

```

end
theory Confluent-Quotient imports
  Confluence
begin

```

Functors with finite setters preserve wide intersection for any equivalence relation that respects the mapper.

```

lemma Inter-finite-subset:
  assumes  $\forall A \in \mathcal{A}. \text{finite } A$ 
  shows  $\exists \mathcal{B} \subseteq \mathcal{A}. \text{finite } \mathcal{B} \wedge (\bigcap \mathcal{B}) = (\bigcap \mathcal{A})$ 
proof(cases  $\mathcal{A} = \{\}$ )
  case False
  then obtain  $A$  where  $A: A \in \mathcal{A}$  by auto
  then have finA:  $\text{finite } A$  using assms by auto
  hence fin:  $\text{finite } (A - \bigcap \mathcal{A})$  by(rule finite-subset[rotated]) auto
  let  $?P = \lambda x. A \in \mathcal{A} \wedge x \notin A$ 
  define  $f$  where  $f \ x = \text{Eps } (?P \ x)$  for  $x$ 
  let  $?B = \text{insert } A \ (f \ ' (A - \bigcap \mathcal{A}))$ 
  have  $?P \ x \ (f \ x)$  if  $x \in A - \bigcap \mathcal{A}$  for  $x$  unfolding f-def by(rule someI-ex)(use
that  $A$  in auto)
  hence  $(\bigcap ?B) = (\bigcap \mathcal{A})$   $?B \subseteq \mathcal{A}$  using  $A$  by auto
  moreover have finite  $?B$  using fin by simp
  ultimately show ?thesis by blast
qed simp

```

```

locale wide-intersection-finite =
  fixes  $E :: 'Fa \Rightarrow 'Fa \Rightarrow \text{bool}$ 
  and mapFa ::  $('a \Rightarrow 'a) \Rightarrow 'Fa \Rightarrow 'Fa$ 
  and setFa ::  $'Fa \Rightarrow 'a \text{ set}$ 
  assumes equiv: equivp  $E$ 
  and map-E:  $E \ x \ y \Longrightarrow E \ (\text{mapFa } f \ x) \ (\text{mapFa } f \ y)$ 
  and map-id:  $\text{mapFa } \text{id} \ x = x$ 
  and map-cong:  $\forall a \in \text{setFa } x. f \ a = g \ a \Longrightarrow \text{mapFa } f \ x = \text{mapFa } g \ x$ 
  and set-map:  $\text{setFa } (\text{mapFa } f \ x) = f \ ' \ \text{setFa } x$ 
  and finite: finite (setFa  $x$ )
begin

```

```

lemma binary-intersection:
  assumes  $E \ y \ z$  and  $y: \text{setFa } y \subseteq Y$  and  $z: \text{setFa } z \subseteq Z$  and  $a: a \in Y \ a \in Z$ 
  shows  $\exists x. E \ x \ y \wedge \text{setFa } x \subseteq Y \wedge \text{setFa } x \subseteq Z$ 
proof –

```

```

let ?f =  $\lambda b.$  if  $b \in Z$  then  $b$  else  $a$ 
let ?u = mapFa ?f y
from  $\langle E\ y\ z \rangle$  have  $E\ ?u\ (\text{mapFa}\ ?f\ z)$  by(rule map-E)
also have mapFa ?f z = mapFa id z by(rule map-cong)(use z in auto)
also have ... = z by(rule map-id)
finally have  $E\ ?u\ y$  using  $\langle E\ y\ z \rangle$  equivp-symp[OF equiv] equivp-transp[OF equiv]
by blast
moreover have setFa ?u  $\subseteq Y$  using a y by(subst set-map) auto
moreover have setFa ?u  $\subseteq Z$  using a by(subst set-map) auto
ultimately show ?thesis by blast
qed

```

```

lemma finite-intersection:
  assumes E:  $\forall y \in A. E\ y\ z$ 
  and fin: finite A
  and sub:  $\forall y \in A. \text{setFa}\ y \subseteq Y\ y \wedge a \in Y\ y$ 
  shows  $\exists x. E\ x\ z \wedge (\forall y \in A. \text{setFa}\ x \subseteq Y\ y)$ 
  using fin E sub
proof(induction)
  case empty
  then show ?case using equivp-reflp[OF equiv, of z] by(auto)
next
  case (insert y A)
  then obtain x where x:  $E\ x\ z \wedge \forall y \in A. \text{setFa}\ x \subseteq Y\ y \wedge a \in Y\ y$  by auto
  hence set-x:  $\text{setFa}\ x \subseteq (\bigcap y \in A. Y\ y) \wedge a \in (\bigcap y \in A. Y\ y)$  by auto
  from insert.prem have E y z and set-y:  $\text{setFa}\ y \subseteq Y\ y \wedge a \in Y\ y$  by auto
  from  $\langle E\ y\ z \rangle \langle E\ x\ z \rangle$  have  $E\ x\ y$  using equivp-symp[OF equiv] equivp-transp[OF equiv] by blast
  from binary-intersection[OF this set-x(1) set-y(1) set-x(2) set-y(2)]
  obtain x' where  $E\ x'\ x \wedge \text{setFa}\ x' \subseteq \bigcap (Y\ 'A) \wedge \text{setFa}\ x' \subseteq Y\ y$  by blast
  then show ?case using  $\langle E\ x\ z \rangle$  equivp-transp[OF equiv] by blast
qed

```

```

lemma wide-intersection:
  assumes inter-nonempty:  $\bigcap Ss \neq \{\}$ 
  shows  $(\bigcap As \in Ss. \{(x, x'). E\ x\ x'\} \subseteq \{x. \text{setFa}\ x \subseteq As\}) \subseteq \{(x, x'). E\ x\ x'\}$  “
   $\{x. \text{setFa}\ x \subseteq \bigcap Ss\}$  (is ?lhs  $\subseteq$  ?rhs)
proof
  fix x
  assume lhs:  $x \in ?lhs$ 
  from inter-nonempty obtain a where a:  $\forall As \in Ss. a \in As$  by auto
  from lhs obtain y where y:  $\bigwedge As. As \in Ss \implies E\ (y\ As)\ x \wedge \text{setFa}\ (y\ As) \subseteq As$ 
  by atomize-elim(rule choice, auto)
  define Ts where  $Ts = (\lambda As. \text{insert}\ a\ (\text{setFa}\ (y\ As)))\ 'Ss$ 
  have Ts-subset:  $(\bigcap Ts) \subseteq (\bigcap Ss)$  using a unfolding Ts-def by(auto dest: y)
  have Ts-finite:  $\forall Bs \in Ts. \text{finite}\ Bs$  unfolding Ts-def by(auto dest: y intro: finite)
  from Inter-finite-subset[OF this] obtain Us
  where Us:  $Us \subseteq Ts$  and finite-Us: finite Us and Int-Us:  $(\bigcap Us) \subseteq (\bigcap Ts)$  by

```

force

```

let ?P =  $\lambda U$  As. As  $\in$  Ss  $\wedge$  U = insert a (setFa (y As))
define Y where Y U = Eps (?P U) for U
have Y: ?P U (Y U) if U  $\in$  Us for U unfolding Y-def
  by(rule someI-ex)(use that Us in  $\langle$ auto simp add: Ts-def $\rangle$ )
let ?f =  $\lambda U$ . y (Y U)
have *:  $\forall z \in (?f \text{ ` } Us). E z x$  by(auto dest!: Y y)
have **:  $\forall z \in (?f \text{ ` } Us). \text{setFa } z \subseteq \text{insert a (setFa } z) \wedge a \in \text{insert a (setFa } z)$  by
  auto
from finite-intersection[OF * - **] finite-Us obtain u
  where u: E u x and set-u:  $\forall z \in (?f \text{ ` } Us). \text{setFa } u \subseteq \text{insert a (setFa } z)$  by auto
from set-u have setFa u  $\subseteq (\bigcap Us)$  by(auto dest: Y)
with Int-Us Ts-subset have setFa u  $\subseteq (\bigcap Ss)$  by auto
with u show x  $\in$  ?rhs by auto
qed

```

end

Subdistributivity for quotients via confluence

```

lemma rtranclp-transp-reflp:  $R^{**} = R$  if transp R reflt R
  apply(rule ext iffI)+
  subgoal premises prems for x y using prems by(induction)(use that in  $\langle$ auto
    intro: refltD transpD $\rangle$ )
  subgoal by(rule r-into-rtranclp)
  done

```

```

lemma rtranclp-equivp:  $R^{**} = R$  if equivp R
  using that by(simp add: rtranclp-transp-reflp equivp-reflp-symp-transp)

```

locale confluent-quotient =

```

  fixes Rb :: 'Fb  $\Rightarrow$  'Fb  $\Rightarrow$  bool
    and Ea :: 'Fa  $\Rightarrow$  'Fa  $\Rightarrow$  bool
    and Eb :: 'Fb  $\Rightarrow$  'Fb  $\Rightarrow$  bool
    and Ec :: 'Fc  $\Rightarrow$  'Fc  $\Rightarrow$  bool
    and Eab :: 'Fab  $\Rightarrow$  'Fab  $\Rightarrow$  bool
    and Ebc :: 'Fbc  $\Rightarrow$  'Fbc  $\Rightarrow$  bool
    and  $\pi$ -Faba :: 'Fab  $\Rightarrow$  'Fa
    and  $\pi$ -Fabb :: 'Fab  $\Rightarrow$  'Fb
    and  $\pi$ -Fbcb :: 'Fbc  $\Rightarrow$  'Fb
    and  $\pi$ -Fbcc :: 'Fbc  $\Rightarrow$  'Fc
    and rel-Fab :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  'Fa  $\Rightarrow$  'Fb  $\Rightarrow$  bool
    and rel-Fbc :: ('b  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  'Fb  $\Rightarrow$  'Fc  $\Rightarrow$  bool
    and rel-Fac :: ('a  $\Rightarrow$  'c  $\Rightarrow$  bool)  $\Rightarrow$  'Fa  $\Rightarrow$  'Fc  $\Rightarrow$  bool
    and set-Fab :: 'Fab  $\Rightarrow$  ('a  $\times$  'b) set
    and set-Fbc :: 'Fbc  $\Rightarrow$  ('b  $\times$  'c) set
  assumes confluent: confluentp Rb
    and retract1-ab:  $\bigwedge x y. Rb (\pi\text{-Fabb } x) y \implies \exists z. Eab x z \wedge y = \pi\text{-Fabb } z \wedge$ 
    set-Fab z  $\subseteq$  set-Fab x
    and retract1-bc:  $\bigwedge x y. Rb (\pi\text{-Fbcb } x) y \implies \exists z. Ebc x z \wedge y = \pi\text{-Fbcb } z \wedge$ 

```


$set-Fbc\ z \subseteq set-Fbc\ x$
and *generated-b*: $Eb \leq equivclp\ Rb$
and *transp-a*: $transp\ Ea$
and *transp-c*: $transp\ Ec$
and *equivp-ab*: $equivp\ Eab$
and *equivp-bc*: $equivp\ Ebc$
and *in-rel-Fab*: $\bigwedge A\ x\ y. rel-Fab\ A\ x\ y \longleftrightarrow (\exists z. z \in \{x. set-Fab\ x \subseteq \{(x, y). A\ x\ y\}\} \wedge \pi-Faba\ z = x \wedge \pi-Fabb\ z = y)$
and *in-rel-Fbc*: $\bigwedge B\ x\ y. rel-Fbc\ B\ x\ y \longleftrightarrow (\exists z. z \in \{x. set-Fbc\ x \subseteq \{(x, y). B\ x\ y\}\} \wedge \pi-Fbcb\ z = x \wedge \pi-Fbcc\ z = y)$
and *rel-comp*: $\bigwedge A\ B. rel-Fac\ (A\ OO\ B) = rel-Fab\ A\ OO\ rel-Fbc\ B$
and $\pi-Faba-respect$: $rel-fun\ Eab\ Ea\ \pi-Faba\ \pi-Faba$
and $\pi-Fbcc-respect$: $rel-fun\ Ebc\ Ec\ \pi-Fbcc\ \pi-Fbcc$
begin

lemma *retract-ab*: $Rb^{**}\ (\pi-Fabb\ x)\ y \implies \exists z. Eab\ x\ z \wedge y = \pi-Fabb\ z \wedge set-Fab\ z \subseteq set-Fab\ x$
by(*induction rule*: *rtranclp-induct*)(*blast dest*: *retract1-ab intro*: *equivp-transp*[*OF equivp-ab*] *equivp-reflp*[*OF equivp-ab*])+

lemma *retract-bc*: $Rb^{**}\ (\pi-Fbcb\ x)\ y \implies \exists z. Ebc\ x\ z \wedge y = \pi-Fbcb\ z \wedge set-Fbc\ z \subseteq set-Fbc\ x$
by(*induction rule*: *rtranclp-induct*)(*blast dest*: *retract1-bc intro*: *equivp-transp*[*OF equivp-bc*] *equivp-reflp*[*OF equivp-bc*])+

lemma *subdistributivity*: $rel-Fab\ A\ OO\ Eb\ OO\ rel-Fbc\ B \leq Ea\ OO\ rel-Fac\ (A\ OO\ B)\ OO\ Ec$

proof(*rule predicate2I*; *elim relcomppE*)

fix $x\ y\ y'\ z$
assume $rel-Fab\ A\ x\ y$ **and** $Eb\ y\ y'$ **and** $rel-Fbc\ B\ y'\ z$
then obtain $xy\ y'z$
where xy : $set-Fab\ xy \subseteq \{(a, b). A\ a\ b\}\ x = \pi-Faba\ xy\ y = \pi-Fabb\ xy$
and $y'z$: $set-Fbc\ y'z \subseteq \{(a, b). B\ a\ b\}\ y' = \pi-Fbcb\ y'z\ z = \pi-Fbcc\ y'z$
by(*auto simp add*: *in-rel-Fab in-rel-Fbc*)
from $\langle Eb\ y\ y' \rangle$ **have** $equivclp\ Rb\ y\ y'$ **using** *generated-b* **by** *blast*
then obtain u **where** u : $Rb^{**}\ y\ u\ Rb^{**}\ y'\ u$
unfolding *semiconfluentp-equivclp*[*OF confluent*][*THEN confluentp-imp-semiconfluentp*]]
by(*auto simp add*: *rtranclp-conversep*)
with $xy\ y'z$ **obtain** $xy'\ y'z'$
where *retract1*: $Eab\ xy\ xy'\ \pi-Fabb\ xy' = u\ set-Fab\ xy' \subseteq set-Fab\ xy$
and *retract2*: $Ebc\ y'z\ y'z'\ \pi-Fbcb\ y'z' = u\ set-Fbc\ y'z' \subseteq set-Fbc\ y'z$
by(*auto dest!*: *retract-ab retract-bc*)
from *retract1*(1) xy **have** $Ea\ x\ (\pi-Faba\ xy')$ **by**(*auto dest*: $\pi-Faba-respect$][*THEN rel-funD*])
moreover have $rel-Fab\ A\ (\pi-Faba\ xy')\ u$ **using** xy *retract1* **by**(*auto simp add*: *in-rel-Fab*)
moreover have $rel-Fbc\ B\ u\ (\pi-Fbcc\ y'z')$ **using** $y'z$ *retract2* **by**(*auto simp add*: *in-rel-Fbc*)
moreover have $Ec\ (\pi-Fbcc\ y'z')\ z$ **using** $y'z$ *retract2* $y'z$ *equivp-symp*[*OF equivp-bc*]

```

    by(auto intro:  $\pi$ -Fbcc-respect[THEN rel-funD])
    ultimately show (Ea OO rel-Fac (A OO B) OO Ec) x z unfolding rel-compp
  by blast
qed

end

end

```

17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

```

theory Old-Datatype
imports Main
begin

```

17.1 The datatype universe

definition $Node = \{p. \exists f x k. p = (f :: nat \Rightarrow 'b + nat, x :: 'a + nat) \wedge f k = Inr\ 0\}$

```

typedef ('a, 'b) node = Node :: ((nat => 'b + nat) * ('a + nat)) set
morphisms Rep-Node Abs-Node
unfolding Node-def by auto

```

Datatypes will be represented by sets of type *node*

```

type-synonym 'a item      = ('a, unit) node set
type-synonym ('a, 'b) dtree = ('a, 'b) node set

```

definition $Push :: [('b + nat), nat \Rightarrow ('b + nat)] \Rightarrow (nat \Rightarrow ('b + nat))$

where $Push == (\%b\ h. case\ nat\ b\ h)$

definition $Push\text{-}Node :: [('b + nat), ('a, 'b)\ node] \Rightarrow ('a, 'b)\ node$
where $Push\text{-}Node == (\%n\ x. Abs\text{-}Node\ (apfst\ (Push\ n)\ (Rep\text{-}Node\ x)))$

definition $Atom :: ('a + nat) \Rightarrow ('a, 'b)\ dtree$

where $Atom == (\%x. \{Abs\text{-}Node((\%k. Inr\ 0, x))\})$

definition $Scons :: [('a, 'b)\ dtree, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$

where $Scons\ M\ N == (Push\text{-}Node\ (Inr\ 1)\ 'M)\ Un\ (Push\text{-}Node\ (Inr\ (Suc\ 1))\ 'N)$

definition $Leaf :: 'a \Rightarrow ('a, 'b)\ dtree$

where $Leaf == Atom \circ Inl$
definition $Numb :: nat \Rightarrow ('a, 'b) dtree$
where $Numb == Atom \circ Inr$

definition $In0 :: ('a, 'b) dtree \Rightarrow ('a, 'b) dtree$
where $In0(M) == Scons (Numb 0) M$
definition $In1 :: ('a, 'b) dtree \Rightarrow ('a, 'b) dtree$
where $In1(M) == Scons (Numb 1) M$

definition $Lim :: ('b \Rightarrow ('a, 'b) dtree) \Rightarrow ('a, 'b) dtree$
where $Lim f == \bigcup \{z. \exists x. z = Push\text{-}Node (Inl x) ' (f x)\}$

definition $ndepth :: ('a, 'b) node \Rightarrow nat$
where $ndepth(n) == (\% (f, x). LEAST k. f k = Inr 0) (Rep\text{-}Node n)$
definition $ntrunc :: [nat, ('a, 'b) dtree] \Rightarrow ('a, 'b) dtree$
where $ntrunc k N == \{n. n \in N \wedge ndepth(n) < k\}$

definition $uprod :: [(('a, 'b) dtree set, ('a, 'b) dtree set) \Rightarrow ('a, 'b) dtree set]$
where $uprod A B == UN x:A. UN y:B. \{ Scons x y \}$
definition $usum :: [(('a, 'b) dtree set, ('a, 'b) dtree set) \Rightarrow ('a, 'b) dtree set]$
where $usum A B == In0'A Un In1'B$

definition $Split :: [(('a, 'b) dtree, ('a, 'b) dtree) \Rightarrow 'c, ('a, 'b) dtree] \Rightarrow 'c$
where $Split c M == THE u. \exists x y. M = Scons x y \wedge u = c x y$

definition $Case :: [(('a, 'b) dtree) \Rightarrow 'c, [(('a, 'b) dtree) \Rightarrow 'c, ('a, 'b) dtree] \Rightarrow 'c]$
where $Case c d M == THE u. (\exists x. M = In0(x) \wedge u = c(x)) \vee (\exists y. M = In1(y) \wedge u = d(y))$

definition $dprod :: [(('a, 'b) dtree * ('a, 'b) dtree) set, ((('a, 'b) dtree * ('a, 'b) dtree) set) \Rightarrow ((('a, 'b) dtree * ('a, 'b) dtree) set)]$
where $dprod r s == UN (x, x'):r. UN (y, y'):s. \{(Scons x y, Scons x' y')\}$

definition $dsum :: [(('a, 'b) dtree * ('a, 'b) dtree) set, ((('a, 'b) dtree * ('a, 'b) dtree) set) \Rightarrow ((('a, 'b) dtree * ('a, 'b) dtree) set)]$
where $dsum r s == (UN (x, x'):r. \{(In0(x), In0(x'))\}) Un (UN (y, y'):s. \{(In1(y), In1(y'))\})$

lemma $apfst\text{-}convE$:

```

  [| q = apfst f p; !!x y. [| p = (x,y); q = (f(x),y) |] ==> R
  |] ==> R
by (force simp add: apfst-def)

```

```

lemma Push-inject1: Push i f = Push j g ==> i=j
apply (simp add: Push-def fun-eq-iff)
apply (drule-tac x=0 in spec, simp)
done

```

```

lemma Push-inject2: Push i f = Push j g ==> f=g
apply (auto simp add: Push-def fun-eq-iff)
apply (drule-tac x=Suc x in spec, simp)
done

```

```

lemma Push-inject:
  [| Push i f = Push j g; [| i=j; f=g |] ==> P |] ==> P
by (blast dest: Push-inject1 Push-inject2)

```

```

lemma Push-neq-K0: Push (Inr (Suc k)) f = (%z. Inr 0) ==> P
by (auto simp add: Push-def fun-eq-iff split: nat.split-asm)

```

```

lemmas Abs-Node-inj = Abs-Node-inject [THEN [2] rev-iffD1]

```

```

lemma Node-K0-I: (λk. Inr 0, a) ∈ Node
by (simp add: Node-def)

```

```

lemma Node-Push-I: p ∈ Node ==> apfst (Push i) p ∈ Node
apply (simp add: Node-def Push-def)
apply (fast intro!: apfst-conv nat.case(2)[THEN trans])
done

```

17.2 Freeness: Distinctness of Constructors

```

lemma Scons-not-Atom [iff]: Scons M N ≠ Atom(a)
unfolding Atom-def Scons-def Push-Node-def One-nat-def
by (blast intro: Node-K0-I Rep-Node [THEN Node-Push-I]
    dest!: Abs-Node-inj
    elim!: apfst-convE sym [THEN Push-neq-K0])

```

```

lemmas Atom-not-Scons [iff] = Scons-not-Atom [THEN not-sym]

```

```

lemma inj-Atom: inj(Atom)
apply (simp add: Atom-def)
apply (blast intro!: inj-onI Node-K0-I dest!: Abs-Node-inj)
done
lemmas Atom-inject = inj-Atom [THEN injD]

```

```

lemma Atom-Atom-eq [iff]: (Atom(a)=Atom(b)) = (a=b)
by (blast dest!: Atom-inject)

```

```

lemma inj-Leaf: inj(Leaf)
apply (simp add: Leaf-def o-def)
apply (rule inj-onI)
apply (erule Atom-inject [THEN Inl-inject])
done

```

```

lemmas Leaf-inject [dest!] = inj-Leaf [THEN injD]

```

```

lemma inj-Numb: inj(Numb)
apply (simp add: Numb-def o-def)
apply (rule inj-onI)
apply (erule Atom-inject [THEN Inr-inject])
done

```

```

lemmas Numb-inject [dest!] = inj-Numb [THEN injD]

```

```

lemma Push-Node-inject:
  [| Push-Node i m = Push-Node j n; [| i=j; m=n |] ==> P
  |] ==> P
apply (simp add: Push-Node-def)
apply (erule Abs-Node-inj [THEN apfst-convE])
apply (rule Rep-Node [THEN Node-Push-I])+
apply (erule sym [THEN apfst-convE])
apply (blast intro: Rep-Node-inject [THEN iffD1] trans sym elim!: Push-inject)
done

```

```

lemma Scons-inject-lemma1: Scons M N <= Scons M' N' ==> M<=M'
unfolding Scons-def One-nat-def
by (blast dest!: Push-Node-inject)

```

```

lemma Scons-inject-lemma2: Scons M N <= Scons M' N' ==> N<=N'
unfolding Scons-def One-nat-def

```

by (*blast dest!*: *Push-Node-inject*)

lemma *Scons-inject1*: $Scons\ M\ N = Scons\ M'\ N' \implies M=M'$
apply (*erule equalityE*)
apply (*iprover intro: equalityI Scons-inject-lemma1*)
done

lemma *Scons-inject2*: $Scons\ M\ N = Scons\ M'\ N' \implies N=N'$
apply (*erule equalityE*)
apply (*iprover intro: equalityI Scons-inject-lemma2*)
done

lemma *Scons-inject*:
 $[[\ Scons\ M\ N = Scons\ M'\ N';\ [\ M=M';\ N=N'\]\] \implies P\] \implies P$
by (*iprover dest: Scons-inject1 Scons-inject2*)

lemma *Scons-Scons-eq* [*iff*]: $(Scons\ M\ N = Scons\ M'\ N') = (M=M' \wedge N=N')$
by (*blast elim!*: *Scons-inject*)

lemma *Scons-not-Leaf* [*iff*]: $Scons\ M\ N \neq Leaf(a)$
unfolding *Leaf-def o-def* **by** (*rule Scons-not-Atom*)

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*]

lemma *Scons-not-Numb* [*iff*]: $Scons\ M\ N \neq Numb(k)$
unfolding *Numb-def o-def* **by** (*rule Scons-not-Atom*)

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*]

lemma *Leaf-not-Numb* [*iff*]: $Leaf(a) \neq Numb(k)$
by (*simp add: Leaf-def Numb-def*)

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*]

lemma *ndepth-K0*: $ndepth\ (Abs-Node(\%k.\ Inr\ 0,\ x)) = 0$
by (*simp add: ndepth-def Node-K0-I* [*THEN Abs-Node-inverse*] *Least-equality*)

lemma *ndepth-Push-Node-aux*:

case-nat (*Inr* (*Suc* *i*)) *f* *k* = *Inr* 0 \longrightarrow *Suc*(*LEAST* *x*. *f* *x* = *Inr* 0) \leq *k*
apply (*induct-tac* *k*, *auto*)
apply (*erule* *Least-le*)
done

lemma *ndepth-Push-Node*:

ndepth (*Push-Node* (*Inr* (*Suc* *i*)) *n*) = *Suc*(*ndepth*(*n*))
apply (*insert* *Rep-Node* [*of* *n*, *unfolded* *Node-def*])
apply (*auto simp add: ndepth-def Push-Node-def*
Rep-Node [*THEN* *Node-Push-I*, *THEN* *Abs-Node-inverse*])
apply (*rule* *Least-equality*)
apply (*auto simp add: Push-def ndepth-Push-Node-aux*)
apply (*erule* *LeastI*)
done

lemma *ntrunc-0* [*simp*]: *ntrunc* 0 *M* = {}
by (*simp add: ntrunc-def*)

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc* *k*) (*Atom* *a*) = *Atom*(*a*)
by (*auto simp add: Atom-def ntrunc-def ndepth-K0*)

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc* *k*) (*Leaf* *a*) = *Leaf*(*a*)
unfolding *Leaf-def o-def* **by** (*rule* *ntrunc-Atom*)

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc* *k*) (*Numb* *i*) = *Numb*(*i*)
unfolding *Numb-def o-def* **by** (*rule* *ntrunc-Atom*)

lemma *ntrunc-Scons* [*simp*]:

ntrunc (*Suc* *k*) (*Scons* *M* *N*) = *Scons* (*ntrunc* *k* *M*) (*ntrunc* *k* *N*)
unfolding *Scons-def ntrunc-def One-nat-def*
by (*auto simp add: ndepth-Push-Node*)

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc* 0) (*In0* *M*) = {}
apply (*simp add: In0-def*)
apply (*simp add: Scons-def*)
done

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc* *k*)) (*In0* *M*) = *In0* (*ntrunc* (*Suc* *k*)
M)
by (*simp add: In0-def*)

lemma *ntrunc-one-In1* [*simp*]: $\text{ntrunc } (\text{Suc } 0) (\text{In1 } M) = \{\}$
apply (*simp add: In1-def*)
apply (*simp add: Scons-def*)
done

lemma *ntrunc-In1* [*simp*]: $\text{ntrunc } (\text{Suc}(\text{Suc } k)) (\text{In1 } M) = \text{In1 } (\text{ntrunc } (\text{Suc } k) M)$
by (*simp add: In1-def*)

17.3 Set Constructions

lemma *uprodI* [*intro!*]: $\llbracket M \in A; N \in B \rrbracket \implies \text{Scons } M N \in \text{uprod } A B$
by (*simp add: uprod-def*)

lemma *uprodE* [*elim!*]:
 $\llbracket c \in \text{uprod } A B;$
 $\bigwedge x y. \llbracket x \in A; y \in B; c = \text{Scons } x y \rrbracket \implies P$
 $\rrbracket \implies P$
by (*auto simp add: uprod-def*)

lemma *uprodE2*: $\llbracket \text{Scons } M N \in \text{uprod } A B; \llbracket M \in A; N \in B \rrbracket \implies P \rrbracket \implies P$
by (*auto simp add: uprod-def*)

lemma *usum-In0I* [*intro*]: $M \in A \implies \text{In0}(M) \in \text{usum } A B$
by (*simp add: usum-def*)

lemma *usum-In1I* [*intro*]: $N \in B \implies \text{In1}(N) \in \text{usum } A B$
by (*simp add: usum-def*)

lemma *usumE* [*elim!*]:
 $\llbracket u \in \text{usum } A B;$
 $\bigwedge x. \llbracket x \in A; u = \text{In0}(x) \rrbracket \implies P;$
 $\bigwedge y. \llbracket y \in B; u = \text{In1}(y) \rrbracket \implies P$
 $\rrbracket \implies P$
by (*auto simp add: usum-def*)

lemma *In0-not-In1* [*iff*]: $\text{In0}(M) \neq \text{In1}(N)$
unfolding *In0-def In1-def One-nat-def* **by** *auto*

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*]

lemma *In0-inject*: $In0(M) = In0(N) ==> M=N$
by (*simp add: In0-def*)

lemma *In1-inject*: $In1(M) = In1(N) ==> M=N$
by (*simp add: In1-def*)

lemma *In0-eq [iff]*: $(In0\ M = In0\ N) = (M=N)$
by (*blast dest!: In0-inject*)

lemma *In1-eq [iff]*: $(In1\ M = In1\ N) = (M=N)$
by (*blast dest!: In1-inject*)

lemma *inj-In0*: *inj In0*
by (*blast intro!: inj-onI*)

lemma *inj-In1*: *inj In1*
by (*blast intro!: inj-onI*)

lemma *Lim-inject*: $Lim\ f = Lim\ g ==> f = g$
apply (*simp add: Lim-def*)
apply (*rule ext*)
apply (*blast elim!: Push-Node-inject*)
done

lemma *ntrunc-subsetI*: $ntrunc\ k\ M \leq M$
by (*auto simp add: ntrunc-def*)

lemma *ntrunc-subsetD*: $(!!k. ntrunc\ k\ M \leq N) ==> M \leq N$
by (*auto simp add: ntrunc-def*)

lemma *ntrunc-equality*: $(!!k. ntrunc\ k\ M = ntrunc\ k\ N) ==> M=N$
apply (*rule equalityI*)
apply (*rule-tac [!] ntrunc-subsetD*)
apply (*rule-tac [!] ntrunc-subsetI [THEN [2] subset-trans], auto*)
done

lemma *ntrunc-o-equality*:
 $[[!!k. (ntrunc(k) \circ h1) = (ntrunc(k) \circ h2)]] ==> h1=h2$
apply (*rule ntrunc-equality [THEN ext]*)
apply (*simp add: fun-eq-iff*)
done

lemma *uprod-mono*: $[[A \leq A'; B \leq B']] \implies \text{uprod } A \ B \leq \text{uprod } A' \ B'$
by (*simp add: uprod-def, blast*)

lemma *usum-mono*: $[[A \leq A'; B \leq B']] \implies \text{usum } A \ B \leq \text{usum } A' \ B'$
by (*simp add: usum-def, blast*)

lemma *Scons-mono*: $[[M \leq M'; N \leq N']] \implies \text{Scons } M \ N \leq \text{Scons } M' \ N'$
by (*simp add: Scons-def, blast*)

lemma *In0-mono*: $M \leq N \implies \text{In0}(M) \leq \text{In0}(N)$
by (*simp add: In0-def Scons-mono*)

lemma *In1-mono*: $M \leq N \implies \text{In1}(M) \leq \text{In1}(N)$
by (*simp add: In1-def Scons-mono*)

lemma *Split* [*simp*]: $\text{Split } c \ (\text{Scons } M \ N) = c \ M \ N$
by (*simp add: Split-def*)

lemma *Case-In0* [*simp*]: $\text{Case } c \ d \ (\text{In0 } M) = c(M)$
by (*simp add: Case-def*)

lemma *Case-In1* [*simp*]: $\text{Case } c \ d \ (\text{In1 } N) = d(N)$
by (*simp add: Case-def*)

lemma *ntrunc-UN1*: $\text{ntrunc } k \ (\text{UN } x. f(x)) = (\text{UN } x. \text{ntrunc } k \ (f \ x))$
by (*simp add: ntrunc-def, blast*)

lemma *Scons-UN1-x*: $\text{Scons } (\text{UN } x. f \ x) \ M = (\text{UN } x. \text{Scons } (f \ x) \ M)$
by (*simp add: Scons-def, blast*)

lemma *Scons-UN1-y*: $\text{Scons } M \ (\text{UN } x. f \ x) = (\text{UN } x. \text{Scons } M \ (f \ x))$
by (*simp add: Scons-def, blast*)

lemma *In0-UN1*: $\text{In0}(\text{UN } x. f(x)) = (\text{UN } x. \text{In0}(f(x)))$
by (*simp add: In0-def Scons-UN1-y*)

lemma *In1-UN1*: $\text{In1}(\text{UN } x. f(x)) = (\text{UN } x. \text{In1}(f(x)))$
by (*simp add: In1-def Scons-UN1-y*)

lemma *dprodI* [*intro!*]:
 $\llbracket (M, M') \in r; (N, N') \in s \rrbracket \implies (Scons\ M\ N, Scons\ M'\ N') \in dprod\ r\ s$
by (*auto simp add: dprod-def*)

lemma *dprodE* [*elim!*]:
 $\llbracket c \in dprod\ r\ s; \bigwedge x\ y\ x'\ y'. \llbracket (x, x') \in r; (y, y') \in s; c = (Scons\ x\ y, Scons\ x'\ y') \rrbracket \implies P \rrbracket \implies P$
by (*auto simp add: dprod-def*)

lemma *dsum-In0I* [*intro*]: $(M, M') \in r \implies (In0(M), In0(M')) \in dsum\ r\ s$
by (*auto simp add: dsum-def*)

lemma *dsum-In1I* [*intro*]: $(N, N') \in s \implies (In1(N), In1(N')) \in dsum\ r\ s$
by (*auto simp add: dsum-def*)

lemma *dsumE* [*elim!*]:
 $\llbracket w \in dsum\ r\ s; \bigwedge x\ x'. \llbracket (x, x') \in r; w = (In0(x), In0(x')) \rrbracket \implies P; \bigwedge y\ y'. \llbracket (y, y') \in s; w = (In1(y), In1(y')) \rrbracket \implies P \rrbracket \implies P$
by (*auto simp add: dsum-def*)

lemma *dprod-mono*: $\llbracket r \leq r'; s \leq s' \rrbracket \implies dprod\ r\ s \leq dprod\ r'\ s'$
by *blast*

lemma *dsum-mono*: $\llbracket r \leq r'; s \leq s' \rrbracket \implies dsum\ r\ s \leq dsum\ r'\ s'$
by *blast*

lemma *dprod-Sigma*: $(dprod\ (A \times B)\ (C \times D)) \leq (uprod\ A\ C) \times (uprod\ B\ D)$
by *blast*

lemmas *dprod-subset-Sigma* = *subset-trans* [*OF dprod-mono dprod-Sigma*]

lemma *dprod-subset-Sigma2*:
 $(dprod (Sigma A B) (Sigma C D)) \leq Sigma (uprod A C) (Split (\%x y. uprod (B x) (D y)))$
by *auto*

lemma *dsum-Sigma*: $(dsum (A \times B) (C \times D)) \leq (usum A C) \times (usum B D)$
by *blast*

lemmas *dsum-subset-Sigma* = *subset-trans* [*OF dsum-mono dsum-Sigma*]

lemma *Domain-dprod [simp]*: $Domain (dprod r s) = uprod (Domain r) (Domain s)$
by *auto*

lemma *Domain-dsum [simp]*: $Domain (dsum r s) = usum (Domain r) (Domain s)$
by *auto*

hides popular names

hide-type (**open**) *node item*

hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

ML-file $\langle \sim \sim /src/HOL/Tools/Old-Datatype/old-datatype.ML \rangle$

end

18 Bijections between natural numbers and other types

theory *Nat-Bijection*

imports *Main*

begin

18.1 Type $nat \times nat$

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition *triangle* :: $nat \Rightarrow nat$
where $triangle\ n = (n * Suc\ n) \div 2$

lemma *triangle-0 [simp]*: $triangle\ 0 = 0$
by (*simp add: triangle-def*)

lemma *triangle-Suc [simp]*: $triangle\ (Suc\ n) = triangle\ n + Suc\ n$
by (*simp add: triangle-def*)

definition *prod-encode* :: $\text{nat} \times \text{nat} \Rightarrow \text{nat}$
where *prod-encode* = $(\lambda(m, n). \text{triangle } (m + n) + m)$

In this auxiliary function, *triangle* $k + m$ is an invariant.

fun *prod-decode-aux* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \times \text{nat}$
where *prod-decode-aux* $k\ m =$
 (if $m \leq k$ then $(m, k - m)$ else *prod-decode-aux* (*Suc* k) ($m - \text{Suc } k$))

declare *prod-decode-aux.simps* [*simp del*]

definition *prod-decode* :: $\text{nat} \Rightarrow \text{nat} \times \text{nat}$
where *prod-decode* = *prod-decode-aux* 0

lemma *prod-encode-prod-decode-aux*: *prod-encode* (*prod-decode-aux* $k\ m$) = *triangle* $k + m$

proof (*induction* $k\ m$ *rule*: *prod-decode-aux.induct*)
case (*1* $k\ m$)
then show ?*case*
by (*simp add*: *prod-encode-def prod-decode-aux.simps*)
qed

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode* n) = n
by (*simp add*: *prod-decode-def prod-encode-prod-decode-aux*)

lemma *prod-decode-triangle-add*: *prod-decode* (*triangle* $k + m$) = *prod-decode-aux* $k\ m$

proof (*induct* k *arbitrary*: m)
case 0
then show ?*case*
by (*simp add*: *prod-decode-def*)
next
case (*Suc* k)
then show ?*case*
by (*metis ab-semigroup-add-class.add-ac*(1) *add-diff-cancel-left'* *le-add1 not-less-eq-eq prod-decode-aux.simps triangle-Suc*)
qed

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode* x) = x
unfolding *prod-encode-def*

proof (*induct* x)
case (*Pair* $a\ b$)
then show ?*case*
by (*simp add*: *prod-decode-triangle-add prod-decode-aux.simps*)
qed

lemma *inj-prod-encode*: *inj-on* *prod-encode* A
by (*rule inj-on-inverseI*) (*rule prod-encode-inverse*)

lemma *inj-prod-decode: inj-on prod-decode A*
by (rule *inj-on-inverseI*) (rule *prod-decode-inverse*)

lemma *surj-prod-encode: surj prod-encode*
by (rule *surjI*) (rule *prod-decode-inverse*)

lemma *surj-prod-decode: surj prod-decode*
by (rule *surjI*) (rule *prod-encode-inverse*)

lemma *bij-prod-encode: bij prod-encode*
by (rule *bijI* [OF *inj-prod-encode surj-prod-encode*])

lemma *bij-prod-decode: bij prod-decode*
by (rule *bijI* [OF *inj-prod-decode surj-prod-decode*])

lemma *prod-encode-eq [simp]: prod-encode x = prod-encode y \longleftrightarrow x = y*
by (rule *inj-prod-encode* [THEN *inj-eq*])

lemma *prod-decode-eq [simp]: prod-decode x = prod-decode y \longleftrightarrow x = y*
by (rule *inj-prod-decode* [THEN *inj-eq*])

Ordering properties

lemma *le-prod-encode-1: a \leq prod-encode (a, b)*
by (*simp add: prod-encode-def*)

lemma *le-prod-encode-2: b \leq prod-encode (a, b)*
by (*induct b*) (*simp-all add: prod-encode-def*)

18.2 Type $\text{nat} + \text{nat}$

definition *sum-encode :: nat + nat \Rightarrow nat*
where *sum-encode x = (case x of Inl a \Rightarrow 2 * a | Inr b \Rightarrow Suc (2 * b))*

definition *sum-decode :: nat \Rightarrow nat + nat*
where *sum-decode n = (if even n then Inl (n div 2) else Inr (n div 2))*

lemma *sum-encode-inverse [simp]: sum-decode (sum-encode x) = x*
by (*induct x*) (*simp-all add: sum-decode-def sum-encode-def*)

lemma *sum-decode-inverse [simp]: sum-encode (sum-decode n) = n*
by (*simp add: even-two-times-div-two sum-decode-def sum-encode-def*)

lemma *inj-sum-encode: inj-on sum-encode A*
by (rule *inj-on-inverseI*) (rule *sum-encode-inverse*)

lemma *inj-sum-decode: inj-on sum-decode A*
by (rule *inj-on-inverseI*) (rule *sum-decode-inverse*)

lemma *surj-sum-encode: surj sum-encode*

by (rule surjI) (rule sum-decode-inverse)

lemma surj-sum-decode: surj sum-decode
by (rule surjI) (rule sum-encode-inverse)

lemma bij-sum-encode: bij sum-encode
by (rule bijI [OF inj-sum-encode surj-sum-encode])

lemma bij-sum-decode: bij sum-decode
by (rule bijI [OF inj-sum-decode surj-sum-decode])

lemma sum-encode-eq: sum-encode $x = \text{sum-encode } y \longleftrightarrow x = y$
by (rule inj-sum-encode [THEN inj-eq])

lemma sum-decode-eq: sum-decode $x = \text{sum-decode } y \longleftrightarrow x = y$
by (rule inj-sum-decode [THEN inj-eq])

18.3 Type *int*

definition int-encode :: $\text{int} \Rightarrow \text{nat}$
where int-encode $i = \text{sum-encode } (\text{if } 0 \leq i \text{ then } \text{Inl } (\text{nat } i) \text{ else } \text{Inr } (\text{nat } (-i - 1)))$

definition int-decode :: $\text{nat} \Rightarrow \text{int}$
where int-decode $n = (\text{case sum-decode } n \text{ of } \text{Inl } a \Rightarrow \text{int } a \mid \text{Inr } b \Rightarrow - \text{int } b - 1)$

lemma int-encode-inverse [simp]: int-decode (int-encode x) = x
by (simp add: int-decode-def int-encode-def)

lemma int-decode-inverse [simp]: int-encode (int-decode n) = n
unfolding int-decode-def int-encode-def
using sum-decode-inverse [of n] **by** (cases sum-decode n) simp-all

lemma inj-int-encode: inj-on int-encode A
by (rule inj-on-inverseI) (rule int-encode-inverse)

lemma inj-int-decode: inj-on int-decode A
by (rule inj-on-inverseI) (rule int-decode-inverse)

lemma surj-int-encode: surj int-encode
by (rule surjI) (rule int-decode-inverse)

lemma surj-int-decode: surj int-decode
by (rule surjI) (rule int-encode-inverse)

lemma bij-int-encode: bij int-encode
by (rule bijI [OF inj-int-encode surj-int-encode])

lemma *bij-int-decode: bij int-decode*
by (rule *bijI* [OF *inj-int-decode surj-int-decode*])

lemma *int-encode-eq: int-encode x = int-encode y \longleftrightarrow x = y*
by (rule *inj-int-encode* [THEN *inj-eq*])

lemma *int-decode-eq: int-decode x = int-decode y \longleftrightarrow x = y*
by (rule *inj-int-decode* [THEN *inj-eq*])

18.4 Type *nat list*

fun *list-encode* :: *nat list* \Rightarrow *nat*
where
list-encode [] = 0
| *list-encode* (x # xs) = *Suc* (*prod-encode* (x, *list-encode* xs))

function *list-decode* :: *nat* \Rightarrow *nat list*
where
list-decode 0 = []
| *list-decode* (*Suc* n) = (case *prod-decode* n of (x, y) \Rightarrow x # *list-decode* y)
by *pat-completeness auto*

termination *list-decode*

proof –
have $\bigwedge n x y. (x, y) = \text{prod-decode } n \implies y < \text{Suc } n$
by (*metis le-imp-less-Suc le-prod-encode-2 prod-decode-inverse*)
then show ?thesis
using *termination* **by** *blast*
qed

lemma *list-encode-inverse* [*simp*]: *list-decode* (*list-encode* x) = x
by (*induct* x rule: *list-encode.induct*) *simp-all*

lemma *list-decode-inverse* [*simp*]: *list-encode* (*list-decode* n) = n

proof (*induct* n rule: *list-decode.induct*)
case (2 n)
then show ?case
by (*metis list-encode.simps(2) list-encode-inverse prod-decode-inverse surj-pair*)
qed *auto*

lemma *inj-list-encode: inj-on list-encode A*
by (rule *inj-on-inverseI*) (rule *list-encode-inverse*)

lemma *inj-list-decode: inj-on list-decode A*
by (rule *inj-on-inverseI*) (rule *list-decode-inverse*)

lemma *surj-list-encode: surj list-encode*
by (rule *surjI*) (rule *list-decode-inverse*)

lemma *surj-list-decode*: *surj list-decode*
by (rule *surjI*) (rule *list-encode-inverse*)

lemma *bij-list-encode*: *bij list-encode*
by (rule *bijI* [OF *inj-list-encode surj-list-encode*])

lemma *bij-list-decode*: *bij list-decode*
by (rule *bijI* [OF *inj-list-decode surj-list-decode*])

lemma *list-encode-eq*: *list-encode x = list-encode y \longleftrightarrow x = y*
by (rule *inj-list-encode* [THEN *inj-eq*])

lemma *list-decode-eq*: *list-decode x = list-decode y \longleftrightarrow x = y*
by (rule *inj-list-decode* [THEN *inj-eq*])

18.5 Finite sets of naturals

18.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite (Suc -‘ F) \longleftrightarrow finite F*
proof

have $F \subseteq \text{insert } 0 \text{ (Suc -‘ Suc -‘ F)}$
using *nat.nchotomy* **by** *force*
moreover
assume *finite (Suc -‘ F)*
then have *finite (insert 0 (Suc -‘ Suc -‘ F))*
by *blast*
ultimately show *finite F*
using *finite-subset* **by** *blast*
qed (*force intro: finite-vimageI inj-Suc*)

lemma *vimage-Suc-insert-0*: *Suc -‘ insert 0 A = Suc -‘ A*
by *auto*

lemma *vimage-Suc-insert-Suc*: *Suc -‘ insert (Suc n) A = insert n (Suc -‘ A)*
by *auto*

lemma *div2-even-ext-nat*:
fixes $x\ y :: \text{nat}$
assumes $x \text{ div } 2 = y \text{ div } 2$
and $\text{even } x \longleftrightarrow \text{even } y$
shows $x = y$
proof –
from $\langle \text{even } x \longleftrightarrow \text{even } y \rangle$ **have** $x \text{ mod } 2 = y \text{ mod } 2$
by (*simp only: even-iff-mod-2-eq-zero*) *auto*
with *assms* **have** $x \text{ div } 2 * 2 + x \text{ mod } 2 = y \text{ div } 2 * 2 + y \text{ mod } 2$
by *simp*
then show *?thesis*
by *simp*
qed

18.5.2 From sets to naturals

definition *set-encode* :: *nat set* \Rightarrow *nat*
where *set-encode* = *sum* ((\cap) 2)

lemma *set-encode-empty* [*simp*]: *set-encode* {} = 0
by (*simp add: set-encode-def*)

lemma *set-encode-inf*: \neg *finite* *A* \implies *set-encode* *A* = 0
by (*simp add: set-encode-def*)

lemma *set-encode-insert* [*simp*]: *finite* *A* \implies $n \notin A \implies$ *set-encode* (*insert* *n* *A*)
 = $2^{\wedge}n +$ *set-encode* *A*
by (*simp add: set-encode-def*)

lemma *even-set-encode-iff*: *finite* *A* \implies *even* (*set-encode* *A*) \longleftrightarrow $0 \notin A$
by (*induct set: finite*) (*auto simp: set-encode-def*)

lemma *set-encode-vimage-Suc*: *set-encode* (*Suc* -‘ *A*) = *set-encode* *A* *div* 2

proof (*induction A rule: infinite-finite-induct*)

case (*infinite* *A*)

then show ?*case*

by (*simp add: finite-vimage-Suc-iff set-encode-inf*)

next

case (*insert* *x* *A*)

show ?*case*

proof (*cases* *x*)

case 0

with insert show ?*thesis*

by (*simp add: even-set-encode-iff vimage-Suc-insert-0*)

next

case (*Suc* *y*)

with insert show ?*thesis*

by (*simp add: finite-vimageI add.commute vimage-Suc-insert-Suc*)

qed

qed *auto*

lemmas *set-encode-div-2* = *set-encode-vimage-Suc* [*symmetric*]

18.5.3 From naturals to sets

definition *set-decode* :: *nat* \Rightarrow *nat set*
where *set-decode* *x* = {*n*. *odd* (*x* *div* 2 \wedge *n*)}

lemma *set-decode-0* [*simp*]: $0 \in$ *set-decode* *x* \longleftrightarrow *odd* *x*
by (*simp add: set-decode-def*)

lemma *set-decode-Suc* [*simp*]: *Suc* *n* \in *set-decode* *x* \longleftrightarrow *n* \in *set-decode* (*x* *div* 2)
by (*simp add: set-decode-def div-mult2-eq*)

lemma *set-decode-zero* [simp]: *set-decode* 0 = {}
by (*simp add: set-decode-def*)

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* -‘ *set-decode x*
by *auto*

lemma *set-decode-plus-power-2*:
 $n \notin \text{set-decode } z \implies \text{set-decode } (2 \wedge n + z) = \text{insert } n (\text{set-decode } z)$
proof (*induct n arbitrary: z*)
case 0
show ?*case*
proof (*rule set-eqI*)
show $q \in \text{set-decode } (2 \wedge 0 + z) \longleftrightarrow q \in \text{insert } 0 (\text{set-decode } z)$ **for** *q*
by (*induct q*) (*use 0 in simp-all*)
qed
next
case (*Suc n*)
show ?*case*
proof (*rule set-eqI*)
show $q \in \text{set-decode } (2 \wedge \text{Suc } n + z) \longleftrightarrow q \in \text{insert } (\text{Suc } n) (\text{set-decode } z)$ **for** *q*
by (*induct q*) (*use Suc in simp-all*)
qed
qed

lemma *finite-set-decode* [simp]: *finite* (*set-decode n*)
proof (*induction n rule: less-induct*)
case (*less n*)
show ?*case*
proof (*cases n = 0*)
case *False*
then show ?*thesis*
using *less.IH* [*of n div 2*] *finite-vimage-Suc-iff set-decode-div-2* **by** *auto*
qed *auto*
qed

18.5.4 Proof of isomorphism

lemma *set-decode-inverse* [simp]: *set-encode* (*set-decode n*) = *n*
proof (*induction n rule: less-induct*)
case (*less n*)
show ?*case*
proof (*cases n = 0*)
case *False*
then have *set-encode* (*set-decode* (*n div 2*)) = *n div 2*
using *less.IH* **by** *auto*
then show ?*thesis*
by (*metis div2-even-ext-nat even-set-encode-iff finite-set-decode set-decode-0 set-decode-div-2 set-encode-div-2*)

qed *auto*
qed

lemma *set-encode-inverse* [*simp*]: *finite A* \implies *set-decode* (*set-encode A*) = *A*
proof (*induction rule: finite-induct*)
 case (*insert x A*)
 then show *?case*
 by (*simp add: set-decode-plus-power-2*)
qed *auto*

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
 by (*rule inj-on-inverseI [where g = set-decode]*) *simp*

lemma *set-encode-eq*: *finite A* \implies *finite B* \implies *set-encode A* = *set-encode B* \longleftrightarrow *A = B*
 by (*rule iffI*) (*simp-all add: inj-onD [OF inj-on-set-encode]*)

lemma *subset-decode-imp-le*:
 assumes *set-decode m* \subseteq *set-decode n*
 shows *m* \leq *n*
proof –
 have *n* = *m* + *set-encode* (*set-decode n* – *set-decode m*)
 proof –
 obtain *A B* **where**
 m = *set-encode A* *finite A*
 n = *set-encode B* *finite B*
 by (*metis finite-set-decode set-decode-inverse*)
 with *assms* **show** *?thesis*
 by *auto* (*simp add: set-encode-def add.commute sum.subset-diff*)
 qed
 then show *?thesis*
 by (*metis le-add1*)
qed

end

19 Encoding (almost) everything into natural numbers

theory *Countable*
imports *Old-Datatype HOL.Rat Nat-Bijection*
begin

19.1 The class of countable types

class *countable* =
 assumes *ex-inj*: \exists *to-nat* :: '*a* \Rightarrow *nat*. *inj to-nat*

```

lemma countable-classI:
  fixes f :: 'a  $\Rightarrow$  nat
  assumes  $\bigwedge x y. f\ x = f\ y \implies x = y$ 
  shows OFCLASS('a, countable-class)
proof (intro-classes, rule exI)
  show inj f
    by (rule injI [OF assms]) assumption
qed

```

19.2 Conversion functions

```

definition to-nat :: 'a::countable  $\Rightarrow$  nat where
  to-nat = (SOME f. inj f)

```

```

definition from-nat :: nat  $\Rightarrow$  'a::countable where
  from-nat = inv (to-nat :: 'a  $\Rightarrow$  nat)

```

```

lemma inj-to-nat [simp]: inj to-nat
  by (rule exE-some [OF ex-inj]) (simp add: to-nat-def)

```

```

lemma inj-on-to-nat[simp, intro]: inj-on to-nat S
  using inj-to-nat by (auto simp: inj-on-def)

```

```

lemma surj-from-nat [simp]: surj from-nat
  unfolding from-nat-def by (simp add: inj-imp-surj-inv)

```

```

lemma to-nat-split [simp]: to-nat x = to-nat y  $\longleftrightarrow$  x = y
  using injD [OF inj-to-nat] by auto

```

```

lemma from-nat-to-nat [simp]:
  from-nat (to-nat x) = x
  by (simp add: from-nat-def)

```

19.3 Finite types are countable

```

subclass (in finite) countable
proof
  have finite (UNIV::'a set) by (rule finite-UNIV)
  with finite-conv-nat-seg-image [of UNIV::'a set]
  obtain n and f :: nat  $\Rightarrow$  'a
    where UNIV = f ` {i. i < n} by auto
  then have surj f unfolding surj-def by auto
  then have inj (inv f) by (rule surj-imp-inj-inv)
  then show  $\exists$  to-nat :: 'a  $\Rightarrow$  nat. inj to-nat by (rule exI[of inj])
qed

```

19.4 Automatically proving countability of old-style datatypes

```

context
begin

```

qualified inductive *finite-item* :: 'a *Old-Datatype.item* \Rightarrow bool **where**
 undefined: *finite-item* *undefined*
 | *In0*: *finite-item* *x* \Longrightarrow *finite-item* (*Old-Datatype.In0* *x*)
 | *In1*: *finite-item* *x* \Longrightarrow *finite-item* (*Old-Datatype.In1* *x*)
 | *Leaf*: *finite-item* (*Old-Datatype.Leaf* *a*)
 | *Scons*: \llbracket *finite-item* *x*; *finite-item* *y* $\rrbracket \Longrightarrow$ *finite-item* (*Old-Datatype.Scons* *x* *y*)

qualified function *nth-item* :: nat \Rightarrow ('a::countable) *Old-Datatype.item*

where

nth-item 0 = *undefined*
 | *nth-item* (*Suc* *n*) =
 (case *sum-decode* *n* of
 Inl *i* \Rightarrow
 (case *sum-decode* *i* of
 Inl *j* \Rightarrow *Old-Datatype.In0* (*nth-item* *j*)
 | *Inr* *j* \Rightarrow *Old-Datatype.In1* (*nth-item* *j*))
 | *Inr* *i* \Rightarrow
 (case *sum-decode* *i* of
 Inl *j* \Rightarrow *Old-Datatype.Leaf* (*from-nat* *j*)
 | *Inr* *j* \Rightarrow
 (case *prod-decode* *j* of
 (*a*, *b*) \Rightarrow *Old-Datatype.Scons* (*nth-item* *a*) (*nth-item* *b*))))
by *pat-completeness auto*

lemma *le-sum-encode-Inl*: $x \leq y \Longrightarrow x \leq$ *sum-encode* (*Inl* *y*)

unfolding *sum-encode-def* **by** *simp*

lemma *le-sum-encode-Inr*: $x \leq y \Longrightarrow x \leq$ *sum-encode* (*Inr* *y*)

unfolding *sum-encode-def* **by** *simp*

qualified termination

by (*relation measure id*)

 (*auto simp flip: sum-encode-eq prod-encode-eq*

simp: le-imp-less-Suc le-sum-encode-Inl le-sum-encode-Inr

le-prod-encode-1 le-prod-encode-2)

lemma *nth-item-covers*: *finite-item* *x* $\Longrightarrow \exists n.$ *nth-item* *n* = *x*

proof (*induct set: finite-item*)

 case *undefined*

have *nth-item* 0 = *undefined* **by** *simp*

thus ?*case* ..

next

 case (*In0* *x*)

then obtain *n* **where** *nth-item* *n* = *x* **by** *fast*

hence *nth-item* (*Suc* (*sum-encode* (*Inl* (*sum-encode* (*Inl* *n*)))))) = *Old-Datatype.In0*

x **by** *simp*

thus ?*case* ..

next

```

    case (In1 x)
    then obtain n where nth-item n = x by fast
    hence nth-item (Suc (sum-encode (Inl (sum-encode (Inr n))))) = Old-Datatype.In1
x by simp
    thus ?case ..
next
    case (Leaf a)
    have nth-item (Suc (sum-encode (Inr (sum-encode (Inl (to-nat a)))))) = Old-Datatype.Leaf
a
    by simp
    thus ?case ..
next
    case (Scons x y)
    then obtain i j where nth-item i = x and nth-item j = y by fast
    hence nth-item
    (Suc (sum-encode (Inr (sum-encode (Inr (prod-encode (i, j))))))) = Old-Datatype.Scons
x y
    by simp
    thus ?case ..
qed

```

theorem countable-datatype:

```

    fixes Rep :: 'b  $\Rightarrow$  ('a::countable) Old-Datatype.item
    fixes Abs :: ('a::countable) Old-Datatype.item  $\Rightarrow$  'b
    fixes rep-set :: ('a::countable) Old-Datatype.item  $\Rightarrow$  bool
    assumes type: type-definition Rep Abs (Collect rep-set)
    assumes finite-item:  $\bigwedge x. \text{rep-set } x \implies \text{finite-item } x$ 
    shows OFCLASS('b, countable-class)
proof
    define f where f y = (LEAST n. nth-item n = Rep y) for y
    {
      fix y :: 'b
      have rep-set (Rep y)
      using type-definition.Rep [OF type] by simp
      hence finite-item (Rep y)
      by (rule finite-item)
      hence  $\exists n. \text{nth-item } n = \text{Rep } y$ 
      by (rule nth-item-covers)
      hence nth-item (f y) = Rep y
      unfolding f-def by (rule LeastI-ex)
      hence Abs (nth-item (f y)) = y
      using type-definition.Rep-inverse [OF type] by simp
    }
    hence inj f
    by (rule inj-on-inverseI)
    thus  $\exists f::'b \Rightarrow \text{nat}. \text{inj } f$ 
    by - (rule exI)
qed

```

```

ML <
  fun old-countable-datatype-tac ctxt =
    SUBGOAL (fn (goal, -) =>
      let
        val ty-name =
          (case goal of
            (- $ Const (const-name <Pure.type>, Type (type-name <itself>, [Type
              (n, -)]))) => n
          | - => raise Match)
        val typedef-info = hd (Typedef.get-info ctxt ty-name)
        val typedef-thm = #type-definition (snd typedef-info)
        val pred-name =
          (case HOLogic.dest-Trueprop (Thm.concl-of typedef-thm) of
            (- $ - $ - $ (- $ Const (n, -))) => n
          | - => raise Match)
        val induct-info = Inductive.the-inductive-global ctxt pred-name
        val pred-names = #names (fst induct-info)
        val induct-thms = #inducts (snd induct-info)
        val alist = pred-names ~~ induct-thms
        val induct-thm = the (AList.lookup (op =) alist pred-name)
        val vars = rev (Term.add-vars (Thm.prop-of induct-thm) [])
        val insts = vars |> map (fn (-, T) => try (Thm.cterm-of ctxt)
          (Const (const-name <Countable.finite-item>, T)))
        val induct-thm' = Thm.instantiate' [] insts induct-thm
        val rules = @{thms finite-item.intros}
      in
        SOLVED' (fn i => EVERY
          [resolve-tac ctxt @{thms countable-datatype} i,
            resolve-tac ctxt [typedef-thm] i,
            eresolve-tac ctxt [induct-thm'] i,
            REPEAT (resolve-tac ctxt rules i ORELSE assume-tac ctxt i)]) 1
      end)
    )
end

```

19.5 Automatically proving countability of datatypes

ML-file <../Tools/BNF/bnf-lfp-countable.ML>

```

ML <
  fun countable-datatype-tac ctxt st =
    (case try <HEADGOAL (old-countable-datatype-tac ctxt) st> of
      SOME res => res
    | NONE => BNF-LFP-Countable.countable-datatype-tac ctxt st);

  (* compatibility *)
  fun countable-tac ctxt =
    SELECT-GOAL (countable-datatype-tac ctxt);

```


›

```
method-setup countable-datatype = ⟨
  Scan.succeed (SIMPLE-METHOD o countable-datatype-tac)
› prove countable class instances for datatypes
```

19.6 More Countable types

Naturals

```
instance nat :: countable
  by (rule countable-classI [of id]) simp
```

Pairs

```
instance prod :: (countable, countable) countable
  by (rule countable-classI [of λ(x, y). prod-encode (to-nat x, to-nat y)])
    (auto simp add: prod-encode-eq)
```

Sums

```
instance sum :: (countable, countable) countable
  by (rule countable-classI [of (λx. case x of Inl a ⇒ to-nat (False, to-nat a)
    | Inr b ⇒ to-nat (True, to-nat b))])
    (simp split: sum.split-asm)
```

Integers

```
instance int :: countable
  by (rule countable-classI [of int-encode]) (simp add: int-encode-eq)
```

Options

```
instance option :: (countable) countable
  by countable-datatype
```

Lists

```
instance list :: (countable) countable
  by countable-datatype
```

String literals

```
instance String.literal :: countable
  by (rule countable-classI [of to-nat ∘ String.explode]) (simp add: String.explode-inject)
```

Functions

```
instance fun :: (finite, countable) countable
proof
  obtain xs :: 'a list where xs: set xs = UNIV
    using finite-list [OF finite-UNIV] ..
  show ∃ to-nat::('a ⇒ 'b) ⇒ nat. inj to-nat
proof
  show inj (λf. to-nat (map f xs))
    by (rule injI, simp add: xs fun-eq-iff)
```

```

qed
qed

Typereps
instance typerep :: countable
  by countable-datatype

```

19.7 The rationals are countably infinite

definition *nat-to-rat-surj* :: *nat* \Rightarrow *rat* **where**
nat-to-rat-surj *n* = (let (*a*, *b*) = prod-decode *n* in Fract (int-decode *a*) (int-decode *b*))

lemma *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*
unfolding *surj-def*
proof
 fix *r*::*rat*
 show $\exists n. r = \text{nat-to-rat-surj } n$
proof (*cases* *r*)
 fix *i j* **assume** [*simp*]: *r* = Fract *i j* **and** *j* > 0
 have *r* = (let *m* = int-encode *i*; *n* = int-encode *j* in nat-to-rat-surj (prod-encode (*m*, *n*)))
 by (*simp* add: Let-def nat-to-rat-surj-def)
 thus $\exists n. r = \text{nat-to-rat-surj } n$ **by** (auto *simp*: Let-def)
 qed
qed

lemma *Rats-eq-range-nat-to-rat-surj*: $\mathbb{Q} = \text{range nat-to-rat-surj}$
 by (*simp* add: Rats-def surj-nat-to-rat-surj)

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
 $\mathbb{Q} = \text{range (of-rat } \circ \text{ nat-to-rat-surj)}$
 using *surj-nat-to-rat-surj*
 by (auto *simp*: Rats-def image-def surj-def) (blast intro: arg-cong[**where** *f* = of-rat])

lemma *surj-of-rat-nat-to-rat-surj*:
 $r \in \mathbb{Q} \implies \exists n. r = \text{of-rat (nat-to-rat-surj } n)$
 by (*simp* add: Rats-eq-range-of-rat-o-nat-to-rat-surj image-def)

end

instance *rat* :: countable
proof
 show $\exists \text{to-nat}::\text{rat} \Rightarrow \text{nat. inj to-nat}$
proof
 have *surj nat-to-rat-surj*

```

      by (rule surj-nat-to-rat-surj)
    then show inj (inv nat-to-rat-surj)
      by (rule surj-imp-inj-inv)
  qed
qed

theorem rat-denum:  $\exists f :: \text{nat} \Rightarrow \text{rat}. \text{surj } f$ 
  using surj-nat-to-rat-surj by metis

end

```

20 Infinite Sets and Related Concepts

```

theory Infinite-Set
  imports Main
begin

```

20.1 The set of natural numbers is infinite

```

lemma infinite-nat-iff-unbounded-le:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n \geq m. n \in S)$ 
  for  $S :: \text{nat set}$ 
  using frequently-cofinite[of  $\lambda x. x \in S$ ]
  by (simp add: cofinite-eq-sequentially frequently-def eventually-sequentially)

lemma infinite-nat-iff-unbounded:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n > m. n \in S)$ 
  for  $S :: \text{nat set}$ 
  using frequently-cofinite[of  $\lambda x. x \in S$ ]
  by (simp add: cofinite-eq-sequentially frequently-def eventually-at-top-dense)

lemma finite-nat-iff-bounded:  $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{..<k\})$ 
  for  $S :: \text{nat set}$ 
  using infinite-nat-iff-unbounded-le[of  $S$ ] by (simp add: subset-eq) (metis not-le)

lemma finite-nat-iff-bounded-le:  $\text{finite } S \longleftrightarrow (\exists k. S \subseteq \{.. k\})$ 
  for  $S :: \text{nat set}$ 
  using infinite-nat-iff-unbounded[of  $S$ ] by (simp add: subset-eq) (metis not-le)

lemma finite-nat-bounded:  $\text{finite } S \implies \exists k. S \subseteq \{..<k\}$ 
  for  $S :: \text{nat set}$ 
  by (simp add: finite-nat-iff-bounded)

```

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k , there is some larger number that is an element of the set.

```

lemma unbounded-k-infinite:  $\forall m > k. \exists n > m. n \in S \implies \text{infinite } (S :: \text{nat set})$ 
  by (metis finite-nat-set-iff-bounded gt-ex order-less-not-sym order-less-trans)

lemma nat-not-finite:  $\text{finite } (\text{UNIV} :: \text{nat set}) \implies \text{False}$ 
  by simp

```

```

lemma range-inj-infinite:
  fixes  $f :: \text{nat} \Rightarrow 'a$ 
  assumes inj  $f$ 
  shows infinite (range  $f$ )
proof
  assume finite (range  $f$ )
  from this assms have finite (UNIV::nat set)
    by (rule finite-imageD)
  then show False by simp
qed

```

20.2 The set of integers is also infinite

```

lemma infinite-int-iff-infinite-nat-abs:  $\text{infinite } S \longleftrightarrow \text{infinite } ((\text{nat} \circ \text{abs}) ' S)$ 
  for  $S :: \text{int set}$ 
proof (unfold Not-eq-iff, rule iffI)
  assume finite  $((\text{nat} \circ \text{abs}) ' S)$ 
  then have finite (nat ' (abs '  $S$ ))
    by (simp add: image-image cong: image-cong)
  moreover have inj-on nat (abs '  $S$ )
    by (rule inj-onI) auto
  ultimately have finite (abs '  $S$ )
    by (rule finite-imageD)
  then show finite  $S$ 
    by (rule finite-image-absD)
qed simp

```

```

proposition infinite-int-iff-unbounded-le:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| \geq m \wedge n \in S)$ 
  for  $S :: \text{int set}$ 
  by (simp add: infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded-le o-def image-def)
    (metis abs-ge-zero nat-le-eq-zle le-nat-iff)

```

```

proposition infinite-int-iff-unbounded:  $\text{infinite } S \longleftrightarrow (\forall m. \exists n. |n| > m \wedge n \in S)$ 
  for  $S :: \text{int set}$ 
  by (simp add: infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded o-def image-def)
    (metis (full-types) nat-le-iff nat-mono not-le)

```

```

proposition finite-int-iff-bounded:  $\text{finite } S \longleftrightarrow (\exists k. \text{abs } ' S \subseteq \{..<k\})$ 
  for  $S :: \text{int set}$ 
  using infinite-int-iff-unbounded-le[of  $S$ ] by (simp add: subset-eq) (metis not-le)

```

```

proposition finite-int-iff-bounded-le:  $\text{finite } S \longleftrightarrow (\exists k. \text{abs } ' S \subseteq \{.. k\})$ 
  for  $S :: \text{int set}$ 
  using infinite-int-iff-unbounded[of  $S$ ] by (simp add: subset-eq) (metis not-le)

```

lemma *infinite-split*: — courtesy of Michael Schmidt
fixes $S :: 'a \text{ set}$
assumes *infinite* S
obtains $A \ B$
where $A \subseteq S \ B \subseteq S$ *infinite* A *infinite* B $A \cap B = \{\}$
proof—
obtain $f :: \text{nat} \Rightarrow 'a$
where $f\text{-inj}$: *inj* f **and** $f\text{-img}$: *range* $f \subseteq S$
using *assms infinite-countable-subset* **by** *blast*
let $?A = \text{range } (\lambda n. f (2*n))$
let $?B = \text{range } (\lambda n. f (2*n+1))$
have $a\text{-inf}$: *infinite* $?A$
using *finite-imageD*[*of* $\lambda n. f (2*n)$ *UNIV*] $f\text{-inj}$ *infinite-UNIV-nat* **unfolding**
inj-def
by *fastforce*
have $b\text{-inf}$: *infinite* $?B$
using *finite-imageD*[*of* $\lambda n. f (2*n+1)$ *UNIV*] $f\text{-inj}$ *infinite-UNIV-nat* **unfolding**
inj-def
by *fastforce*
from $f\text{-inj}$ **have** $?A \cap ?B = \{\}$ **unfolding** *inj-def disjoint-iff* **using** *double-not-eq-Suc-double*
by *auto*
from *that*[*OF* - - $a\text{-inf}$ $b\text{-inf}$ *this*] $f\text{-img}$ **show** $?thesis$
by *blast*
qed

20.3 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM* [*simp*]: $\neg (\text{INFM } x. P \ x) \longleftrightarrow (\text{MOST } x. \neg P \ x)$
by (*rule not-frequently*)

lemma *not-MOST* [*simp*]: $\neg (\text{MOST } x. P \ x) \longleftrightarrow (\text{INFM } x. \neg P \ x)$
by (*rule not-eventually*)

lemma *INFM-const* [*simp*]: $(\text{INFM } x::'a. P) \longleftrightarrow P \wedge \text{infinite } (\text{UNIV}::'a \text{ set})$
by (*simp add: frequently-const-iff*)

lemma *MOST-const* [*simp*]: $(\text{MOST } x::'a. P) \longleftrightarrow P \vee \text{finite } (\text{UNIV}::'a \text{ set})$
by (*simp add: eventually-const-iff*)

lemma *INFM-imp-distrib*: $(\text{INFM } x. P \ x \longrightarrow Q \ x) \longleftrightarrow ((\text{MOST } x. P \ x) \longrightarrow (\text{INFM } x. Q \ x))$
by (*rule frequently-imp-iff*)

lemma *MOST-imp-iff*: $\text{MOST } x. P \ x \Longrightarrow (\text{MOST } x. P \ x \longrightarrow Q \ x) \longleftrightarrow (\text{MOST } x. Q \ x)$

by (*auto intro: eventually-rev-mp eventually-mono*)

lemma *INFM-conjI*: $INFM\ x.\ P\ x \implies MOST\ x.\ Q\ x \implies INFM\ x.\ P\ x \wedge Q\ x$
by (*rule frequently-rev-mp[of P]*) (*auto elim: eventually-mono*)

Properties of quantifiers with injective functions.

lemma *INFM-inj*: $INFM\ x.\ P\ (f\ x) \implies inj\ f \implies INFM\ x.\ P\ x$
using *finite-vimageI*[of $\{x.\ P\ x\}$ *f*] **by** (*auto simp: frequently-cofinite*)

lemma *MOST-inj*: $MOST\ x.\ P\ x \implies inj\ f \implies MOST\ x.\ P\ (f\ x)$
using *finite-vimageI*[of $\{x.\ \neg P\ x\}$ *f*] **by** (*auto simp: eventually-cofinite*)

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
 $\neg (INFM\ x.\ x = a)$
 $\neg (INFM\ x.\ a = x)$
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-neq* [*simp*]:
 $MOST\ x.\ x \neq a$
 $MOST\ x.\ a \neq x$
unfolding *eventually-cofinite* **by** *simp-all*

lemma *INFM-neq* [*simp*]:
 $(INFM\ x::'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV::'a\ set)$
 $(INFM\ x::'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV::'a\ set)$
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-eq* [*simp*]:
 $(MOST\ x::'a.\ x = a) \longleftrightarrow finite\ (UNIV::'a\ set)$
 $(MOST\ x::'a.\ a = x) \longleftrightarrow finite\ (UNIV::'a\ set)$
unfolding *eventually-cofinite* **by** *simp-all*

lemma *MOST-eq-imp*:
 $MOST\ x.\ x = a \longrightarrow P\ x$
 $MOST\ x.\ a = x \longrightarrow P\ x$
unfolding *eventually-cofinite* **by** *simp-all*

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_{\infty} n.\ P\ n) \longleftrightarrow (\exists m.\ \forall n > m.\ P\ n)$
for $P :: nat \Rightarrow bool$
by (*auto simp add: eventually-cofinite finite-nat-iff-bounded-le subset-eq simp flip: not-le*)

lemma *MOST-nat-le*: $(\forall_{\infty} n.\ P\ n) \longleftrightarrow (\exists m.\ \forall n \geq m.\ P\ n)$
for $P :: nat \Rightarrow bool$
by (*auto simp add: eventually-cofinite finite-nat-iff-bounded subset-eq simp flip: not-le*)

lemma *INFM-nat*: $(\exists_{\infty} n. P n) \longleftrightarrow (\forall m. \exists n > m. P n)$
for $P :: \text{nat} \Rightarrow \text{bool}$
by (*simp add: frequently-cofinite infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: $(\exists_{\infty} n. P n) \longleftrightarrow (\forall m. \exists n \geq m. P n)$
for $P :: \text{nat} \Rightarrow \text{bool}$
by (*simp add: frequently-cofinite infinite-nat-iff-unbounded-le*)

lemma *MOST-INFM*: $\text{infinite } (UNIV::'a \text{ set}) \implies \text{MOST } x::'a. P x \implies \text{INFM } x::'a. P x$
by (*simp add: eventually-frequently*)

lemma *MOST-Suc-iff*: $(\text{MOST } n. P (\text{Suc } n)) \longleftrightarrow (\text{MOST } n. P n)$
by (*simp add: cofinite-eq-sequentially*)

lemma *MOST-SucI*: $\text{MOST } n. P n \implies \text{MOST } n. P (\text{Suc } n)$
and *MOST-SucD*: $\text{MOST } n. P (\text{Suc } n) \implies \text{MOST } n. P n$
by (*simp-all add: MOST-Suc-iff*)

lemma *MOST-ge-nat*: $\text{MOST } n::\text{nat}. m \leq n$
by (*simp add: cofinite-eq-sequentially*)

— legacy names

lemma *Inf-many-def*: $\text{Inf-many } P \longleftrightarrow \text{infinite } \{x. P x\}$ **by** (*fact frequently-cofinite*)

lemma *Alm-all-def*: $\text{Alm-all } P \longleftrightarrow \neg (\text{INFM } x. \neg P x)$ **by** *simp*

lemma *INFM-iff-infinite*: $(\text{INFM } x. P x) \longleftrightarrow \text{infinite } \{x. P x\}$ **by** (*fact frequently-cofinite*)

lemma *MOST-iff-cofinite*: $(\text{MOST } x. P x) \longleftrightarrow \text{finite } \{x. \neg P x\}$ **by** (*fact eventually-cofinite*)

lemma *INFM-EX*: $(\exists_{\infty} x. P x) \implies (\exists x. P x)$ **by** (*fact frequently-ex*)

lemma *ALL-MOST*: $\forall x. P x \implies \forall_{\infty} x. P x$ **by** (*fact always-eventually*)

lemma *INFM-mono*: $\exists_{\infty} x. P x \implies (\bigwedge x. P x \implies Q x) \implies \exists_{\infty} x. Q x$ **by** (*fact frequently-elim1*)

lemma *MOST-mono*: $\forall_{\infty} x. P x \implies (\bigwedge x. P x \implies Q x) \implies \forall_{\infty} x. Q x$ **by** (*fact eventually-mono*)

lemma *INFM-disj-distrib*: $(\exists_{\infty} x. P x \vee Q x) \longleftrightarrow (\exists_{\infty} x. P x) \vee (\exists_{\infty} x. Q x)$ **by** (*fact frequently-disj-iff*)

lemma *MOST-rev-mp*: $\forall_{\infty} x. P x \implies \forall_{\infty} x. P x \longrightarrow Q x \implies \forall_{\infty} x. Q x$ **by** (*fact eventually-rev-mp*)

lemma *MOST-conj-distrib*: $(\forall_{\infty} x. P x \wedge Q x) \longleftrightarrow (\forall_{\infty} x. P x) \wedge (\forall_{\infty} x. Q x)$ **by** (*fact eventually-conj-iff*)

lemma *MOST-conjI*: $\text{MOST } x. P x \implies \text{MOST } x. Q x \implies \text{MOST } x. P x \wedge Q x$
by (*fact eventually-conj*)

lemma *INFM-finite-Bex-distrib*: $\text{finite } A \implies (\text{INFM } y. \exists x \in A. P x y) \longleftrightarrow (\exists x \in A. \text{INFM } y. P x y)$ **by** (*fact frequently-bex-finite-distrib*)

lemma *MOST-finite-Ball-distrib*: $\text{finite } A \implies (\text{MOST } y. \forall x \in A. P x y) \longleftrightarrow (\forall x \in A. \text{MOST } y. P x y)$ **by** (*fact eventually-ball-finite-distrib*)

lemma *INFM-E*: $\text{INFM } x. P x \implies (\bigwedge x. P x \implies \text{thesis}) \implies \text{thesis}$ **by** (*fact frequentlyE*)

lemma *MOST-I*: $(\bigwedge x. P x) \implies \text{MOST } x. P x$ **by** (*rule eventuallyI*)
lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

20.4 Enumeration of an Infinite Set

The set’s element type must be wellordered (e.g. the natural numbers).

Could be generalized to $\text{enumerate}' S n = (\text{SOME } t. t \in s \wedge \text{finite } \{s \in S. s < t\} \wedge \text{card } \{s \in S. s < t\} = n)$.

primrec (*in wellorder*) *enumerate* :: ‘a set \Rightarrow nat \Rightarrow ‘a
where
enumerate-0: $\text{enumerate } S \ 0 = (\text{LEAST } n. n \in S)$
| *enumerate-Suc*: $\text{enumerate } S \ (\text{Suc } n) = \text{enumerate } (S - \{\text{LEAST } n. n \in S\}) \ n$

lemma *enumerate-Suc'*: $\text{enumerate } S \ (\text{Suc } n) = \text{enumerate } (S - \{\text{enumerate } S \ 0\}) \ n$
by *simp*

lemma *enumerate-in-set*: $\text{infinite } S \implies \text{enumerate } S \ n \in S$

proof (*induct n arbitrary: S*)
case 0
then show ?*case*
by (*fastforce intro: LeastI dest!: infinite-imp-nonempty*)
next
case (*Suc n*)
then show ?*case*
by *simp (metis DiffE infinite-remove)*
qed

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: $\text{infinite } S \implies \text{enumerate } S \ n < \text{enumerate } S \ (\text{Suc } n)$

proof (*induction n arbitrary: S*)
case 0
then have $\text{enumerate } S \ 0 \leq \text{enumerate } S \ (\text{Suc } 0)$
by (*simp add: enumerate-0 Least-le enumerate-in-set*)
moreover have $\text{enumerate } (S - \{\text{enumerate } S \ 0\}) \ 0 \in S - \{\text{enumerate } S \ 0\}$
by (*meson 0.premis enumerate-in-set infinite-remove*)
then have $\text{enumerate } S \ 0 \neq \text{enumerate } (S - \{\text{enumerate } S \ 0\}) \ 0$
by *auto*
ultimately show ?*case*
by (*simp add: enumerate-Suc'*)
next
case (*Suc n*)
then show ?*case*
by (*simp add: enumerate-Suc'*)
qed

lemma *enumerate-mono*: $m < n \implies \text{infinite } S \implies \text{enumerate } S \ m < \text{enumerate } S \ n$

$S\ n$
by (*induct* $m\ n$ *rule: less-Suc-induct*) (*auto intro: enumerate-step*)

lemma *enumerate-mono-iff* [*simp*]:
 $\text{infinite } S \implies \text{enumerate } S\ m < \text{enumerate } S\ n \longleftrightarrow m < n$
by (*metis enumerate-mono less-asym less-linear*)

lemma *enumerate-mono-le-iff* [*simp*]:
 $\text{infinite } S \implies \text{enumerate } S\ m \leq \text{enumerate } S\ n \longleftrightarrow m \leq n$
by (*meson enumerate-mono-iff not-le*)

lemma *le-enumerate*:
assumes S : *infinite* S
shows $n \leq \text{enumerate } S\ n$
using S
proof (*induct* n)
case 0
then show ?*case* **by** *simp*
next
case (*Suc* n)
then have $n \leq \text{enumerate } S\ n$ **by** *simp*
also note *enumerate-mono*[*of* n *Suc* n , *OF* - $\langle \text{infinite } S \rangle$]
finally show ?*case* **by** *simp*
qed

lemma *infinite-enumerate*:
assumes fS : *infinite* S
shows $\exists r::\text{nat} \Rightarrow \text{nat. strict-mono } r \wedge (\forall n. r\ n \in S)$
unfolding *strict-mono-def*
using *enumerate-in-set*[*OF* fS] *enumerate-mono*[*of* - - S] fS **by** *blast*

lemma *enumerate-Suc''*:
fixes $S :: 'a::\text{wellorder}$ *set*
assumes *infinite* S
shows $\text{enumerate } S\ (\text{Suc } n) = (\text{LEAST } s. s \in S \wedge \text{enumerate } S\ n < s)$
using *assms*
proof (*induct* n *arbitrary: S*)
case 0
then have $\forall s \in S. \text{enumerate } S\ 0 \leq s$
by (*auto simp: enumerate.simps intro: Least-le*)
then show ?*case*
unfolding *enumerate-Suc'* *enumerate-0*[*of* $S - \{\text{enumerate } S\ 0\}$]
by (*intro arg-cong*[**where** $f = \text{Least}$] *ext*) *auto*
next
case (*Suc* n S)
show ?*case*
proof (*rule order.antisym*)
have S : *infinite* ($S - \{\text{wellorder-class.enumerate } S\ 0\}$)
using *Suc* **by** *auto*

```

show wellorder-class.enumerate  $S$  (Suc (Suc  $n$ ))  $\leq$  (LEAST  $s$ .  $s \in S \wedge$ 
wellorder-class.enumerate  $S$  (Suc  $n$ )  $< s$ )
using enumerate-mono[OF zero-less-Suc]  $\langle$ infinite  $S\rangle$   $S$ 
by (smt (verit, best) LeastI-ex Suc.hyps enumerate-0 enumerate-Suc enumer-
ate-in-set
      enumerate-step insertE insert-Diff linorder-not-less not-less-Least)
qed (simp add: Least-le Suc.premis enumerate-in-set)
qed

```

lemma *enumerate-Ex*:

```

fixes  $S :: \text{nat set}$ 
assumes  $S$ : infinite  $S$ 
and  $s$ :  $s \in S$ 
shows  $\exists n$ . enumerate  $S$   $n = s$ 
using  $s$ 
proof (induct  $s$  rule: less-induct)
case (less  $s$ )
show ?case
proof (cases  $\exists y \in S$ .  $y < s$ )
case True
let ? $y = \text{Max } \{s' \in S. s' < s\}$ 
from True have  $y$ :  $\bigwedge x. ?y < x \longleftrightarrow (\forall s' \in S. s' < s \longrightarrow s' < x)$ 
by (subst Max-less-iff) auto
then have  $y$ -in:  $?y \in \{s' \in S. s' < s\}$ 
by (intro Max-in) auto
with less.hyps[of ? $y$ ] obtain  $n$  where enumerate  $S$   $n = ?y$ 
by auto
with  $S$  have enumerate  $S$  (Suc  $n$ )  $= s$ 
by (auto simp:  $y$  less enumerate-Suc'' intro!: Least-equality)
then show ?thesis by auto
next
case False
then have  $\forall t \in S. s \leq t$  by auto
with  $\langle s \in S \rangle$  show ?thesis
by (auto intro!: exI[of - 0] Least-equality simp: enumerate-0)
qed
qed

```

lemma *inj-enumerate*:

```

fixes  $S :: 'a::\text{wellorder set}$ 
assumes  $S$ : infinite  $S$ 
shows inj (enumerate  $S$ )
unfolding inj-on-def
proof clarsimp
show  $\bigwedge x y$ . enumerate  $S$   $x = \text{enumerate } S y \implies x = y$ 
by (metis neq-iff enumerate-mono[OF -  $\langle$ infinite  $S\rangle$ ])
qed

```

To generalise this, we’d need a condition that all initial segments were finite

```

lemma bij-enumerate:
  fixes  $S :: \text{nat set}$ 
  assumes  $S: \text{infinite } S$ 
  shows bij-betw (enumerate  $S$ ) UNIV  $S$ 
proof –
  have  $\forall s \in S. \exists i. \text{enumerate } S \ i = s$ 
    using enumerate-Ex[OF  $S$ ] by auto
  moreover note  $\langle \text{infinite } S \rangle$  inj-enumerate
  ultimately show ?thesis
    unfolding bij-betw-def by (auto intro: enumerate-in-set)
qed

```

```

lemma
  fixes  $S :: \text{nat set}$ 
  assumes  $S: \text{infinite } S$ 
  shows range-enumerate: range (enumerate  $S$ ) =  $S$ 
    and strict-mono-enumerate: strict-mono (enumerate  $S$ )
  by (auto simp add: bij-betw-imp-surj-on bij-enumerate assms strict-mono-def)

```

A pair of weird and wonderful lemmas from HOL Light.

```

lemma finite-transitivity-chain:
  assumes finite  $A$ 
    and  $R: \bigwedge x. \neg R \ x \ x \ \bigwedge x \ y \ z. \llbracket R \ x \ y; R \ y \ z \rrbracket \implies R \ x \ z$ 
    and  $A: \bigwedge x. x \in A \implies \exists y. y \in A \wedge R \ x \ y$ 
  shows  $A = \{\}$ 
    using  $\langle \text{finite } A \rangle$   $A$ 
proof (induct  $A$ )
  case empty
    then show ?case by simp
next
  case (insert  $a$   $A$ )
    have False
      using  $R(1)[\text{of } a] \ R(2)[\text{of } - \ a] \ \text{insert}(3,4)$  by blast
    thus ?case ..
qed

```

```

corollary Union-maximal-sets:
  assumes finite  $\mathcal{F}$ 
  shows  $\bigcup \{T \in \mathcal{F}. \forall U \in \mathcal{F}. \neg T \subset U\} = \bigcup \mathcal{F}$ 
    (is ?lhs = ?rhs)
proof
  show ?lhs  $\subseteq$  ?rhs by force
  show ?rhs  $\subseteq$  ?lhs
proof (rule Union-subsetI)
  fix  $S$ 
  assume  $S \in \mathcal{F}$ 
  have  $\{T \in \mathcal{F}. S \subseteq T\} = \{\}$ 
    if  $\neg (\exists y. y \in \{T \in \mathcal{F}. \forall U \in \mathcal{F}. \neg T \subset U\} \wedge S \subseteq y)$ 
proof –

```

```

have §:  $\bigwedge x. x \in \mathcal{F} \wedge S \subseteq x \implies \exists y. y \in \mathcal{F} \wedge S \subseteq y \wedge x \subset y$ 
  using that by (blast intro: dual-order.trans psubset-imp-subset)
show ?thesis
proof (rule finite-transitivity-chain [of -  $\lambda T U. S \subseteq T \wedge T \subset U$ ])
qed (use assms in <auto intro: §>)
qed
with < $S \in \mathcal{F}$ > show  $\exists y. y \in \{T \in \mathcal{F}. \forall U \in \mathcal{F}. \neg T \subset U\} \wedge S \subseteq y$ 
  by blast
qed
qed

```

20.5 Properties of *wellorder-class.enumerate* on finite sets

lemma *finite-enumerate-in-set*: $\llbracket \text{finite } S; n < \text{card } S \rrbracket \implies \text{enumerate } S \ n \in S$

proof (induction *n* arbitrary: *S*)

```

case 0
then show ?case
  by (metis all-not-in-conv card.empty enumerate.simps(1) not-less0 wellorder-Least-lemma(1))
next
case (Suc n)
then have wellorder-class.enumerate (S - {LEAST n. n ∈ S}) n ∈ S
  by (metis Diff-empty Diff-insert0 Suc-lessD Suc-less-eq card.insert-remove
    finite-Diff insert-Diff insert-Diff-single insert-iff)
then
show ?case
  using Suc.premis Suc.IH [of S - {LEAST n. n ∈ S}]
  by (simp add: enumerate.simps)
qed

```

lemma *finite-enumerate-step*: $\llbracket \text{finite } S; \text{Suc } n < \text{card } S \rrbracket \implies \text{enumerate } S \ n < \text{enumerate } S \ (\text{Suc } n)$

proof (induction *n* arbitrary: *S*)

```

case 0
then have enumerate S 0 ≤ enumerate S (Suc 0)
  by (simp add: Least-le enumerate.simps(1) finite-enumerate-in-set)
moreover have enumerate (S - {enumerate S 0}) 0 ∈ S - {enumerate S 0}
  by (metis 0 Suc-lessD Suc-less-eq card-Suc-Diff1 enumerate-in-set finite-enumerate-in-set)
then have enumerate S 0 ≠ enumerate (S - {enumerate S 0}) 0
  by auto
ultimately show ?case
  by (simp add: enumerate-Suc')
next
case (Suc n)
then show ?case
  by (simp add: enumerate-Suc' finite-enumerate-in-set)
qed

```

lemma *finite-enumerate-mono*: $\llbracket m < n; \text{finite } S; n < \text{card } S \rrbracket \implies \text{enumerate } S \ m < \text{enumerate } S \ n$

```

    by (induct m n rule: less-Suc-induct) (auto intro: finite-enumerate-step)

lemma finite-enumerate-mono-iff [simp]:
   $\llbracket \text{finite } S; m < \text{card } S; n < \text{card } S \rrbracket \implies \text{enumerate } S \ m < \text{enumerate } S \ n \longleftrightarrow m < n$ 
  by (metis finite-enumerate-mono less-asm less-linear)

lemma finite-le-enumerate:
  assumes finite S n < card S
  shows  $n \leq \text{enumerate } S \ n$ 
  using assms
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have  $n \leq \text{enumerate } S \ n$  by simp
  also note finite-enumerate-mono[of n Suc n, OF - ⟨finite S⟩]
  finally show ?case
    using Suc.premis(2) Suc-leI by blast
qed

lemma finite-enumerate:
  assumes fS: finite S
  shows  $\exists r :: \text{nat} \Rightarrow \text{nat}. \text{strict-mono-on } \{.. < \text{card } S\} \ r \wedge (\forall n < \text{card } S. r \ n \in S)$ 
  unfolding strict-mono-def
  using finite-enumerate-in-set[OF fS] finite-enumerate-mono[of - - S] fS
  by (metis lessThan-iff strict-mono-on-def)

lemma finite-enumerate-Suc'':
  fixes S :: 'a::wellorder set
  assumes finite S Suc n < card S
  shows  $\text{enumerate } S \ (\text{Suc } n) = (\text{LEAST } s. s \in S \wedge \text{enumerate } S \ n < s)$ 
  using assms
proof (induction n arbitrary: S)
  case 0
  then have  $\forall s \in S. \text{enumerate } S \ 0 \leq s$ 
    by (auto simp: enumerate.simps intro: Least-le)
  then show ?case
    unfolding enumerate-Suc' enumerate-0[of S - {enumerate S 0}]
    by (metis Diff-iff dual-order.strict-iff-order singletonD singletonI)
next
  case (Suc n S)
  then have  $\text{Suc } n < \text{card } (S - \{\text{enumerate } S \ 0\})$ 
    using Suc.premis(2) finite-enumerate-in-set by force
  then show ?case
    apply (subst (1 2) enumerate-Suc')
    apply (simp add: Suc)
    apply (intro arg-cong[where f = Least] HOL.ext)

```

```

    using finite-enumerate-mono[OF zero-less-Suc ‹finite S›, of n] Suc.prem
    by (auto simp flip: enumerate-Suc)
qed

lemma finite-enumerate-initial-segment:
  fixes S :: 'a::wellorder set
  assumes finite S and n: n < card (S ∩ {..

```

and $s: s \in S$
shows $\exists n < \text{card } S. \text{ enumerate } S \ n = s$
using $s \ S$
proof (*induction s arbitrary: S rule: less-induct*)
case (*less s*)
show $?case$
proof (*cases $\exists y \in S. y < s$*)
case *True*
let $?T = S \cap \{..<s\}$
have *finite ?T*
using *less.premis(2) by blast*
have $TS: \text{card } ?T < \text{card } S$
using *less.premis by (blast intro: psubset-card-mono [OF <finite S>])*
from *True* **have** $y: \bigwedge x. \text{Max } ?T < x \longleftrightarrow (\forall s' \in S. s' < s \longrightarrow s' < x)$
by (*subst Max-less-iff*) (*auto simp: <finite ?T>*)
then have $y\text{-in}: \text{Max } ?T \in \{s' \in S. s' < s\}$
using *Max-in <finite ?T> by fastforce*
with *less.IH[of Max ?T ?T]* **obtain** n **where** $n: \text{enumerate } ?T \ n = \text{Max } ?T \ n$
 $< \text{card } ?T$
using *<finite ?T> by blast*
then have $\text{Suc } n < \text{card } S$
using *TS less-trans-Suc by blast*
with $S \ n$ **have** $\text{enumerate } S \ (\text{Suc } n) = s$
by (*subst finite-enumerate-Suc''*) (*auto simp: y finite-enumerate-initial-segment*
less finite-enumerate-Suc'' intro!: Least-equality)
then show $?thesis$
using *<Suc n < card S> by blast*
next
case *False*
then have $\forall t \in S. s \leq t$ **by** *auto*
moreover have $0 < \text{card } S$
using *card-0-eq less.premis by blast*
ultimately show $?thesis$
using *<s ∈ S>*
by (*auto intro!: exI[of - 0] Least-equality simp: enumerate-0*)
qed
qed

lemma *finite-enum-subset:*

assumes $\bigwedge i. i < \text{card } X \implies \text{enumerate } X \ i = \text{enumerate } Y \ i$ **and** *finite X finite*
 $Y \ \text{card } X \leq \text{card } Y$
shows $X \subseteq Y$
by (*metis assms finite-enumerate-Ex finite-enumerate-in-set less-le-trans subsetI*)

lemma *finite-enum-ext:*

assumes $\bigwedge i. i < \text{card } X \implies \text{enumerate } X \ i = \text{enumerate } Y \ i$ **and** *finite X finite*
 $Y \ \text{card } X = \text{card } Y$
shows $X = Y$
by (*intro antisym finite-enum-subset*) (*auto simp: assms*)

```

lemma finite-bij-enumerate:
  fixes  $S :: 'a::wellorder\ set$ 
  assumes  $S: finite\ S$ 
  shows  $bij\_betw\ (enumerate\ S)\ \{.. $card\ S\}\ S$ 
proof -
  have  $\bigwedge n\ m. \llbracket n \neq m; n < card\ S; m < card\ S \rrbracket \implies enumerate\ S\ n \neq enumerate\ S\ m$ 
  using finite-enumerate-mono[ $OF - \langle finite\ S \rangle$ ] by (auto simp: neq-iff)
  then have  $inj\_on\ (enumerate\ S)\ \{.. $card\ S\}$ 
  by (auto simp: inj-on-def)
  moreover have  $\forall s \in S. \exists i < card\ S. enumerate\ S\ i = s$ 
  using finite-enumerate-Ex[ $OF\ S$ ] by auto
  moreover note  $\langle finite\ S \rangle$ 
  ultimately show ?thesis
  unfolding bij-betw-def by (auto intro: finite-enumerate-in-set)
qed

lemma ex-bij-betw-strict-mono-card:
  fixes  $M :: 'a::wellorder\ set$ 
  assumes  $finite\ M$ 
  obtains  $h$  where  $bij\_betw\ h\ \{.. $card\ M\}\ M$  and  $strict\_mono\_on\ \{.. $card\ M\}\ h$ 
proof
  show  $bij\_betw\ (enumerate\ M)\ \{.. $card\ M\}\ M$ 
  by (simp add: asms finite-bij-enumerate)
  show  $strict\_mono\_on\ \{.. $card\ M\}\ (enumerate\ M)$ 
  by (simp add: asms finite-enumerate-mono strict-mono-on-def)
qed

end$$$$$$ 
```

21 Countable sets

```

theory Countable-Set
imports Countable Infinite-Set
begin

```

21.1 Predicate for countable sets

```

definition countable ::  $'a\ set \Rightarrow bool$  where
   $countable\ S \longleftrightarrow (\exists f::'a \Rightarrow nat. inj\_on\ f\ S)$ 

```

```

lemma countable-as-injective-image-subset:  $countable\ S \longleftrightarrow (\exists f. \exists K::nat\ set. S = f\ `K \wedge inj\_on\ f\ K)$ 
by (metis countable-def inj-on-the-inv-into the-inv-into-onto)

```

```

lemma countableE:
  assumes  $S: countable\ S$  obtains  $f :: 'a \Rightarrow nat$  where  $inj\_on\ f\ S$ 
  using  $S$  by (auto simp: countable-def)

```


lemma *countableI*: *inj-on* (*f*::'*a* \Rightarrow *nat*) *S* \Longrightarrow *countable S*
by (*auto simp: countable-def*)

lemma *countableI'*: *inj-on* (*f*::'*a* \Rightarrow '*b*::*countable*) *S* \Longrightarrow *countable S*
using *comp-inj-on*[*of f S to-nat*] **by** (*auto intro: countableI*)

lemma *countableE-bij*:
assumes *S*: *countable S* **obtains** *f* :: *nat* \Rightarrow '*a* **and** *C* :: *nat set* **where** *bij-betw*
f C S
using *S* **by** (*blast elim: countableE dest: inj-on-imp-bij-betw bij-betw-inv*)

lemma *countableI-bij*: *bij-betw f* (*C*::*nat set*) *S* \Longrightarrow *countable S*
by (*blast intro: countableI bij-betw-inv-into bij-betw-imp-inj-on*)

lemma *countable-finite*: *finite S* \Longrightarrow *countable S*
by (*blast dest: finite-imp-inj-to-nat-seg countableI*)

lemma *countableI-bij1*: *bij-betw f A B* \Longrightarrow *countable A* \Longrightarrow *countable B*
by (*blast elim: countableE-bij intro: bij-betw-trans countableI-bij*)

lemma *countableI-bij2*: *bij-betw f B A* \Longrightarrow *countable A* \Longrightarrow *countable B*
by (*blast elim: countableE-bij intro: bij-betw-trans bij-betw-inv-into countableI-bij*)

lemma *countable-iff-bij*[*simp*]: *bij-betw f A B* \Longrightarrow *countable A* \longleftrightarrow *countable B*
by (*blast intro: countableI-bij1 countableI-bij2*)

lemma *countable-subset*: *A* \subseteq *B* \Longrightarrow *countable B* \Longrightarrow *countable A*
by (*auto simp: countable-def intro: inj-on-subset*)

lemma *countableI-type*[*intro, simp*]: *countable* (*A*:: '*a* :: *countable set*)
using *countableI*[*of to-nat A*] **by** *auto*

21.2 Enumerate a countable set

lemma *countableE-infinite*:
assumes *countable S* *infinite S*
obtains *e* :: '*a* \Rightarrow *nat* **where** *bij-betw e S UNIV*
proof –
obtain *f* :: '*a* \Rightarrow *nat* **where** *inj-on f S*
using \langle *countable S* \rangle **by** (*rule countableE*)
then have *bij-betw f S* (*f*'*S*)
unfolding *bij-betw-def* **by** *simp*
moreover
from \langle *inj-on f S* \rangle \langle *infinite S* \rangle **have** *inf-fS*: *infinite* (*f*'*S*)
by (*auto dest: finite-imageD*)
then have *bij-betw* (*the-inv-into UNIV* (*enumerate* (*f*'*S*))) (*f*'*S*) *UNIV*
by (*intro bij-betw-the-inv-into bij-enumerate*)
ultimately have *bij-betw* (*the-inv-into UNIV* (*enumerate* (*f*'*S*)) \circ *f*) *S UNIV*

by (rule bij-betw-trans)
 then show thesis ..
 qed

lemma countable-infiniteE':
 assumes countable A infinite A
 obtains g where bij-betw g (UNIV :: nat set) A
 by (meson assms bij-betw-inv countableE-infinite)

lemma countable-enum-cases:
 assumes countable S
 obtains (finite) f :: 'a \Rightarrow nat where finite S bij-betw f S {.. $\text{card } S$ }
 | (infinite) f :: 'a \Rightarrow nat where infinite S bij-betw f S UNIV
 using ex-bij-betw-finite-nat[of S] countableE-infinite <countable S>
 by (cases finite S) (auto simp add: atLeast0LessThan)

definition to-nat-on :: 'a set \Rightarrow 'a \Rightarrow nat where
 to-nat-on S = (SOME f. if finite S then bij-betw f S {.. $\text{card } S$ } else bij-betw f S UNIV)

definition from-nat-into :: 'a set \Rightarrow nat \Rightarrow 'a where
 from-nat-into S n = (if n \in to-nat-on S ' S then inv-into S (to-nat-on S) n else SOME s. s \in S)

lemma to-nat-on-finite: finite S \implies bij-betw (to-nat-on S) S {.. $\text{card } S$ }
 using ex-bij-betw-finite-nat unfolding to-nat-on-def
 by (intro someI2-ex[where Q= λf . bij-betw f S {.. $\text{card } S$]]) (auto simp add: atLeast0LessThan)

lemma to-nat-on-infinite: countable S \implies infinite S \implies bij-betw (to-nat-on S) S UNIV
 using countableE-infinite unfolding to-nat-on-def
 by (intro someI2-ex[where Q= λf . bij-betw f S UNIV]) auto

lemma bij-betw-from-nat-into-finite: finite S \implies bij-betw (from-nat-into S) {.. $\text{card } S$ } S
 unfolding from-nat-into-def[abs-def]
 using to-nat-on-finite[of S]
 apply (subst bij-betw-cong)
 apply (split if-split)
 apply (simp add: bij-betw-def)
 apply (auto cong: bij-betw-cong
 intro: bij-betw-inv-into to-nat-on-finite)
 done

lemma bij-betw-from-nat-into: countable S \implies infinite S \implies bij-betw (from-nat-into S) UNIV S
 unfolding from-nat-into-def[abs-def]
 using to-nat-on-infinite[of S, unfolded bij-betw-def]

by (*auto cong: bij-betw-cong intro: bij-betw-inv-into to-nat-on-infinite*)

The sum/product over the enumeration of a finite set equals simply the sum/product over the set

context *comm-monoid-set*
begin

lemma *card-from-nat-into*:

$F (\lambda i. h (from-nat-into A i)) \{..<card A\} = F h A$

proof (*cases finite A*)

case *True*

have $F (\lambda i. h (from-nat-into A i)) \{..<card A\} = F h (from-nat-into A \text{ ‘ } \{..<card A\})$

by (*metis True bij-betw-def bij-betw-from-nat-into-finite reindex-cong*)

also have $... = F h A$

by (*metis True bij-betw-def bij-betw-from-nat-into-finite*)

finally show *?thesis* .

qed *auto*

end

lemma *countable-as-injective-image*:

assumes *countable A infinite A*

obtains $f :: nat \Rightarrow 'a$ **where** $A = range f \text{ inj } f$

by (*metis bij-betw-def bij-betw-from-nat-into [OF assms]*)

lemma *inj-on-to-nat-on[intro]*: *countable A $\implies inj-on (to-nat-on A) A$*

using *to-nat-on-infinite[of A] to-nat-on-finite[of A]*

by (*cases finite A*) (*auto simp: bij-betw-def*)

lemma *to-nat-on-inj[simp]*:

countable A $\implies a \in A \implies b \in A \implies to-nat-on A a = to-nat-on A b \longleftrightarrow a = b$

using *inj-on-to-nat-on[of A]* **by** (*auto dest: inj-onD*)

lemma *from-nat-into-to-nat-on[simp]*: *countable A $\implies a \in A \implies from-nat-into A (to-nat-on A a) = a$*

by (*auto simp: from-nat-into-def intro!: inv-into-f-f*)

lemma *subset-range-from-nat-into*: *countable A $\implies A \subseteq range (from-nat-into A)$*

by (*auto intro: from-nat-into-to-nat-on[symmetric]*)

lemma *from-nat-into*: *$A \neq \{\}$ $\implies from-nat-into A n \in A$*

unfolding *from-nat-into-def* **by** (*metis equalsOI inv-into-into someI-ex*)

lemma *range-from-nat-into-subset*: *$A \neq \{\} \implies range (from-nat-into A) \subseteq A$*

using *from-nat-into[of A]* **by** *auto*

lemma *range-from-nat-into[simp]*: *$A \neq \{\} \implies countable A \implies range (from-nat-into A) = A$*

by (*metis equalityI range-from-nat-into-subset subset-range-from-nat-into*)

lemma *image-to-nat-on*: $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A \text{ ‘ } A = \text{UNIV}$
using *to-nat-on-infinite*[of *A*] **by** (*simp add: bij-betw-def*)

lemma *to-nat-on-surj*: $\text{countable } A \implies \text{infinite } A \implies \exists a \in A. \text{to-nat-on } A \ a = n$
by (*metis (no-types) image-iff iso-tuple-UNIV-I image-to-nat-on*)

lemma *to-nat-on-from-nat-into*[*simp*]: $n \in \text{to-nat-on } A \text{ ‘ } A \implies \text{to-nat-on } A \ (\text{from-nat-into } A \ n) = n$
by (*simp add: f-inv-into-f from-nat-into-def*)

lemma *to-nat-on-from-nat-into-infinite*[*simp*]:
 $\text{countable } A \implies \text{infinite } A \implies \text{to-nat-on } A \ (\text{from-nat-into } A \ n) = n$
by (*metis image-iff to-nat-on-surj to-nat-on-from-nat-into*)

lemma *from-nat-into-inj*:
 $\text{countable } A \implies m \in \text{to-nat-on } A \text{ ‘ } A \implies n \in \text{to-nat-on } A \text{ ‘ } A \implies$
 $\text{from-nat-into } A \ m = \text{from-nat-into } A \ n \longleftrightarrow m = n$
by (*subst to-nat-on-inj[symmetric, of *A*]) auto*

lemma *from-nat-into-inj-infinite*[*simp*]:
 $\text{countable } A \implies \text{infinite } A \implies \text{from-nat-into } A \ m = \text{from-nat-into } A \ n \longleftrightarrow m = n$
using *image-to-nat-on*[of *A*] *from-nat-into-inj*[of *A m n*] **by** *simp*

lemma *eq-from-nat-into-iff*:
 $\text{countable } A \implies x \in A \implies i \in \text{to-nat-on } A \text{ ‘ } A \implies x = \text{from-nat-into } A \ i \longleftrightarrow$
 $i = \text{to-nat-on } A \ x$
by *auto*

lemma *from-nat-into-surj*: $\text{countable } A \implies a \in A \implies \exists n. \text{from-nat-into } A \ n = a$
by (*rule exI*[of - *to-nat-on A a*]) *simp*

lemma *from-nat-into-inject*[*simp*]:
 $A \neq \{\} \implies \text{countable } A \implies B \neq \{\} \implies \text{countable } B \implies \text{from-nat-into } A =$
 $\text{from-nat-into } B \longleftrightarrow A = B$
by (*metis range-from-nat-into*)

lemma *inj-on-from-nat-into*: $\text{inj-on from-nat-into } (\{A. A \neq \{\} \wedge \text{countable } A\})$
unfolding *inj-on-def* **by** *auto*

21.3 Closure properties of countability

lemma *countable-SIGMA*[*intro, simp*]:
 $\text{countable } I \implies (\bigwedge i. i \in I \implies \text{countable } (A \ i)) \implies \text{countable } (\text{SIGMA } i : I. A \ i)$
by (*intro countableI*’[of $\lambda(i, a). (\text{to-nat-on } I \ i, \text{to-nat-on } (A \ i) \ a)$]) (*auto simp*:

inj-on-def)

lemma *countable-image*[*intro, simp*]:

assumes *countable A*

shows *countable (f‘A)*

proof –

obtain *g :: 'a ⇒ nat* **where** *inj-on g A*

using *assms* **by** (*rule countableE*)

moreover have *inj-on (inv-into A f) (f‘A) inv-into A f ‘ f ‘ A ⊆ A*

by (*auto intro: inj-on-inv-into inv-into-into*)

ultimately show *?thesis*

by (*blast dest: comp-inj-on inj-on-subset intro: countableI*)

qed

lemma *countable-image-inj-on: countable (f ‘ A) ⇒ inj-on f A ⇒ countable A*

by (*metis countable-image the-inv-into-onto*)

lemma *countable-image-inj-Int-vimage:*

$\llbracket \text{inj-on } f \ S; \text{countable } A \rrbracket \Longrightarrow \text{countable } (S \cap f^{-1} A)$

by (*meson countable-image-inj-on countable-subset image-subset-iff-subset-vimage inf-le2 inj-on-Int*)

lemma *countable-image-inj-gen:*

$\llbracket \text{inj-on } f \ S; \text{countable } A \rrbracket \Longrightarrow \text{countable } \{x \in S. f \ x \in A\}$

using *countable-image-inj-Int-vimage*

by (*auto simp: vimage-def Collect-conj-eq*)

lemma *countable-image-inj-eq:*

$\text{inj-on } f \ S \Longrightarrow \text{countable}(f^{-1} S) \longleftrightarrow \text{countable } S$

using *countable-image-inj-on* **by** *blast*

lemma *countable-image-inj:*

$\llbracket \text{countable } A; \text{inj } f \rrbracket \Longrightarrow \text{countable } \{x. f \ x \in A\}$

by (*metis (mono-tags, lifting) countable-image-inj-eq countable-subset image-Collect-subsetI inj-on-inverseI the-inv-f-f*)

lemma *countable-UN*[*intro, simp*]:

fixes *I :: 'i set* **and** *A :: 'i => 'a set*

assumes *I: countable I*

assumes *A: $\bigwedge i. i \in I \Longrightarrow \text{countable } (A \ i)$*

shows *countable ($\bigcup i \in I. A \ i$)*

proof –

have $(\bigcup i \in I. A \ i) = \text{snd } (SIGMA \ i : I. A \ i)$ **by** (*auto simp: image-iff*)

then show *?thesis* **by** (*simp add: assms*)

qed

lemma *countable-Un*[*intro*]: *countable A ⇒ countable B ⇒ countable (A ∪ B)*

by (*rule countable-UN[of {True, False} λ True ⇒ A | False ⇒ B, simplified]*)

(*simp split: bool.split*)

lemma *countable-Un-iff*[simp]: $\text{countable } (A \cup B) \longleftrightarrow \text{countable } A \wedge \text{countable } B$
by (*metis countable-Un countable-subset inf-sup-ord*(3,4))

lemma *countable-Plus*[intro, simp]:
 $\text{countable } A \implies \text{countable } B \implies \text{countable } (A <+> B)$
by (*simp add: Plus-def*)

lemma *countable-empty*[intro, simp]: $\text{countable } \{\}$
by (*blast intro: countable-finite*)

lemma *countable-insert*[intro, simp]: $\text{countable } A \implies \text{countable } (\text{insert } a \ A)$
using *countable-Un*[of $\{a\}$ A] **by** (*auto simp: countable-finite*)

lemma *countable-Int1*[intro, simp]: $\text{countable } A \implies \text{countable } (A \cap B)$
by (*force intro: countable-subset*)

lemma *countable-Int2*[intro, simp]: $\text{countable } B \implies \text{countable } (A \cap B)$
by (*blast intro: countable-subset*)

lemma *countable-INT*[intro, simp]: $i \in I \implies \text{countable } (A \ i) \implies \text{countable } (\bigcap_{i \in I} A \ i)$
by (*blast intro: countable-subset*)

lemma *countable-Diff*[intro, simp]: $\text{countable } A \implies \text{countable } (A - B)$
by (*blast intro: countable-subset*)

lemma *countable-insert-eq* [simp]: $\text{countable } (\text{insert } x \ A) = \text{countable } A$
by *auto* (*metis Diff-insert-absorb countable-Diff insert-absorb*)

lemma *countable-vimage*: $B \subseteq \text{range } f \implies \text{countable } (f^{-1} B) \implies \text{countable } B$
by (*metis Int-absorb2 countable-image image-vimage-eq*)

lemma *surj-countable-vimage*: $\text{surj } f \implies \text{countable } (f^{-1} B) \implies \text{countable } B$
by (*metis countable-vimage top-greatest*)

lemma *countable-Collect*[simp]: $\text{countable } A \implies \text{countable } \{a \in A. \varphi \ a\}$
by (*metis Collect-conj-eq Int-absorb Int-commute Int-def countable-Int1*)

lemma *countable-Image*:
assumes $\bigwedge y. y \in Y \implies \text{countable } (X \ \{y\})$
assumes *countable Y*
shows *countable* $(X \ \{Y\})$
proof –
have *countable* $(X \ \{(\bigcup_{y \in Y} \{y\})\})$
unfolding *Image-UN* **by** (*intro countable-UN assms*)
then show *?thesis* **by** *simp*
qed

```

lemma countable-relpow:
  fixes  $X :: 'a \text{ rel}$ 
  assumes  $\text{Image-}X: \bigwedge Y. \text{countable } Y \implies \text{countable } (X \text{ `` } Y)$ 
  assumes  $Y: \text{countable } Y$ 
  shows  $\text{countable } ((X \text{ `` } i) \text{ `` } Y)$ 
  using  $Y$  by (induct  $i$  arbitrary:  $Y$ ) (auto simp: relcomp-Image Image- $X$ )

lemma countable-funpow:
  fixes  $f :: 'a \text{ set} \Rightarrow 'a \text{ set}$ 
  assumes  $\bigwedge A. \text{countable } A \implies \text{countable } (f A)$ 
  and  $\text{countable } A$ 
  shows  $\text{countable } ((f \text{ `` } n) A)$ 
by(induction  $n$ )(simp-all add: assms)

lemma countable-rtrancI:
  ( $\bigwedge Y. \text{countable } Y \implies \text{countable } (X \text{ `` } Y)$ )  $\implies \text{countable } Y \implies \text{countable } (X^* \text{ `` } Y)$ 
  unfolding rtrancI-is-UN-relpow UN-Image by (intro countable-UN countableI-type countable-relpow)

lemma countable-lists[intro, simp]:
  assumes  $A: \text{countable } A$  shows  $\text{countable } (\text{lists } A)$ 
proof –
  have  $\text{countable } (\text{lists } (\text{range } (\text{from-nat-into } A)))$ 
    by (auto simp: lists-image)
  with  $A$  show ?thesis
    by (auto dest: subset-range-from-nat-into countable-subset lists-mono)
qed

lemma Collect-finite-eq-lists:  $\text{Collect finite} = \text{set `` lists UNIV}$ 
  using finite-list by auto

lemma countable-Collect-finite:  $\text{countable } (\text{Collect } (\text{finite}::'a::\text{countable set} \Rightarrow \text{bool}))$ 
  by (simp add: Collect-finite-eq-lists)

lemma countable-int:  $\text{countable } \mathbb{Z}$ 
  unfolding Ints-def by auto

lemma countable-rat:  $\text{countable } \mathbb{Q}$ 
  unfolding Rats-def by auto

lemma Collect-finite-subset-eq-lists:  $\{A. \text{finite } A \wedge A \subseteq T\} = \text{set `` lists } T$ 
  using finite-list by (auto simp: lists-eq-set)

lemma countable-Collect-finite-subset:
   $\text{countable } T \implies \text{countable } \{A. \text{finite } A \wedge A \subseteq T\}$ 
  unfolding Collect-finite-subset-eq-lists by auto

lemma countable-Fpow:  $\text{countable } S \implies \text{countable } (F\text{pow } S)$ 

```

using *countable-Collect-finite-subset*
by (*force simp add: Fpow-def conj-commute*)

lemma *countable-set-option* [*simp*]: *countable (set-option x)*
by (*cases x*) *auto*

21.4 Misc lemmas

lemma *countable-subset-image*:
 $\text{countable } B \wedge B \subseteq (f \text{ ‘ } A) \longleftrightarrow (\exists A'. \text{countable } A' \wedge A' \subseteq A \wedge (B = f \text{ ‘ } A'))$
(is ?lhs = ?rhs)
proof
assume *?lhs*
show *?rhs*
by (*rule exI [where x=inv-into A f ‘ B]*)
(use <?lhs> in <auto simp: f-inv-into-f subset-iff image-inv-into-cancel inv-into-into>)
next
assume *?rhs*
then show *?lhs* **by** *force*
qed

lemma *ex-subset-image-inj*:
 $(\exists T. T \subseteq f \text{ ‘ } S \wedge P T) \longleftrightarrow (\exists T. T \subseteq S \wedge \text{inj-on } f T \wedge P (f \text{ ‘ } T))$
by (*auto simp: subset-image-inj*)

lemma *all-subset-image-inj*:
 $(\forall T. T \subseteq f \text{ ‘ } S \longrightarrow P T) \longleftrightarrow (\forall T. T \subseteq S \wedge \text{inj-on } f T \longrightarrow P (f \text{ ‘ } T))$
by (*metis subset-image-inj*)

lemma *ex-countable-subset-image-inj*:
 $(\exists T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \wedge P T) \longleftrightarrow$
 $(\exists T. \text{countable } T \wedge T \subseteq S \wedge \text{inj-on } f T \wedge P (f \text{ ‘ } T))$
by (*metis countable-image-inj-eq subset-image-inj*)

lemma *all-countable-subset-image-inj*:
 $(\forall T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \longrightarrow P T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \wedge$
 $\text{inj-on } f T \longrightarrow P (f \text{ ‘ } T))$
by (*metis countable-image-inj-eq subset-image-inj*)

lemma *ex-countable-subset-image*:
 $(\exists T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \wedge P T) \longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge P (f$
 $\text{ ‘ } T))$
by (*metis countable-subset-image*)

lemma *all-countable-subset-image*:
 $(\forall T. \text{countable } T \wedge T \subseteq f \text{ ‘ } S \longrightarrow P T) \longleftrightarrow (\forall T. \text{countable } T \wedge T \subseteq S \longrightarrow$
 $P (f \text{ ‘ } T))$
by (*metis countable-subset-image*)

lemma *countable-image-eq*:

countable($f \text{ ‘ } S$) $\longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ‘ } S = f \text{ ‘ } T)$

by (*metis countable-image countable-image-inj-eq order-refl subset-image-inj*)

lemma *countable-image-eq-inj*:

countable($f \text{ ‘ } S$) $\longleftrightarrow (\exists T. \text{countable } T \wedge T \subseteq S \wedge f \text{ ‘ } S = f \text{ ‘ } T \wedge \text{inj-on } f \text{ } T)$

by (*metis countable-image-inj-eq order-refl subset-image-inj*)

lemma *infinite-countable-subset'*:

assumes X : *infinite* X **shows** $\exists C \subseteq X. \text{countable } C \wedge \text{infinite } C$

proof –

obtain $f :: \text{nat} \Rightarrow 'a$ **where** $\text{inj } f \text{ range } f \subseteq X$

using *infinite-countable-subset [OF X]* **by** *blast*

then show *?thesis*

by (*intro exI[of - range f]*) (*auto simp: range-inj-infinite*)

qed

lemma *countable-all*:

assumes S : *countable* S

shows $(\forall s \in S. P \text{ } s) \longleftrightarrow (\forall n :: \text{nat}. \text{from-nat-into } S \text{ } n \in S \longrightarrow P (\text{from-nat-into } S \text{ } n))$

using $S[\text{THEN subset-range-from-nat-into}]$ **by** *auto*

lemma *finite-sequence-to-countable-set*:

assumes *countable* X

obtains F **where** $\bigwedge i. F \text{ } i \subseteq X \wedge i. F \text{ } i \subseteq F (\text{Suc } i) \wedge i. \text{finite } (F \text{ } i) (\bigcup i. F \text{ } i) = X$

proof –

show *thesis*

apply (*rule that[of $\lambda i. \text{if } X = \{\} \text{ then } \{\} \text{ else from-nat-into } X \text{ ‘ } \{..i\}$]*)

apply (*auto simp add: image-iff intro: from-nat-into split: if-splits*)

using *assms from-nat-into-surj* **by** (*fastforce cong: image-cong*)

qed

lemma *transfer-countable[transfer-rule]*:

bi-unique $R \implies \text{rel-fun } (\text{rel-set } R) (=) \text{countable countable}$

by (*rule rel-funI, erule (1) bi-unique-rel-set-lemma*)

(*auto dest: countable-image-inj-on*)

21.5 Uncountable

abbreviation *uncountable where*

uncountable $A \equiv \neg \text{countable } A$

lemma *uncountable-def*: *uncountable* $A \longleftrightarrow A \neq \{\} \wedge \neg (\exists f :: (\text{nat} \Rightarrow 'a). \text{range } f = A)$

by (*auto intro: inj-on-inv-into simp: countable-def*)

(*metis all-not-in-conv inj-on-iff-surj subset-UNIV*)

lemma *uncountable-bij-betw*: $\text{bij-betw } f \ A \ B \implies \text{uncountable } B \implies \text{uncountable } A$
unfolding *bij-betw-def* **by** (*metis countable-image*)

lemma *uncountable-infinite*: $\text{uncountable } A \implies \text{infinite } A$
by (*metis countable-finite*)

lemma *uncountable-minus-countable*:
 $\text{uncountable } A \implies \text{countable } B \implies \text{uncountable } (A - B)$
using *countable-Un[of B A - B]* **by** *auto*

lemma *countable-Diff-eq [simp]*: $\text{countable } (A - \{x\}) = \text{countable } A$
by (*meson countable-Diff countable-empty countable-insert uncountable-minus-countable*)

Every infinite set can be covered by a pairwise disjoint family of infinite sets. This version doesn’t achieve equality, as it only covers a countable subset

lemma *infinite-infinite-partition*:
assumes *infinite A*
obtains $C :: \text{nat} \Rightarrow 'a \text{ set}$
where $\text{pairwise } (\lambda i \ j. \text{disjnt } (C \ i) \ (C \ j)) \ UNIV \ (\bigcup i. C \ i) \subseteq A \ \wedge \ i. \text{infinite } (C \ i)$
proof –
obtain $f :: \text{nat} \Rightarrow 'a$ **where** $\text{range } f \subseteq A$ **inj** f
using *assms infinite-countable-subset* **by** *blast*
let $?C = \lambda i. \text{range } (\lambda j. f \ (\text{prod-encode } (i, j)))$
show *thesis*
proof
show $\text{pairwise } (\lambda i \ j. \text{disjnt } (?C \ i) \ (?C \ j)) \ UNIV$
by (*auto simp: pairwise-def disjnt-def inj-on-eq-iff [OF <inj f>] inj-on-eq-iff [OF inj-prod-encode, of - UNIV]*)
show $(\bigcup i. ?C \ i) \subseteq A$
using $\langle \text{range } f \subseteq A \rangle$ **by** *blast*
have $\text{infinite } (\text{range } (\lambda j. f \ (\text{prod-encode } (i, j))))$ **for** i
by (*rule range-inj-infinite*) (*meson Pair-inject <inj f> inj-def prod-encode-eq*)
then show $\bigwedge i. \text{infinite } (?C \ i)$
using *that* **by** *auto*
qed
qed
end

22 Countable Complete Lattices

theory *Countable-Complete-Lattices*
imports *Main Countable-Set*
begin

lemma *UNIV-nat-eq*: $UNIV = \text{insert } 0 \ (\text{range } \text{Suc})$
by (*metis UNIV-eq-I nat.nchotomy insertCI rangeI*)

```

class countable-complete-lattice = lattice + Inf + Sup + bot + top +
  assumes ccInf-lower: countable A  $\implies x \in A \implies \text{Inf } A \leq x$ 
  assumes ccInf-greatest: countable A  $\implies (\bigwedge x. x \in A \implies z \leq x) \implies z \leq \text{Inf } A$ 
  assumes ccSup-upper: countable A  $\implies x \in A \implies x \leq \text{Sup } A$ 
  assumes ccSup-least: countable A  $\implies (\bigwedge x. x \in A \implies x \leq z) \implies \text{Sup } A \leq z$ 
  assumes ccInf-empty [simp]: Inf {} = top
  assumes ccSup-empty [simp]: Sup {} = bot
begin

subclass bounded-lattice
proof
  fix a
  show bot  $\leq a$  by (auto intro: ccSup-least simp only: ccSup-empty [symmetric])
  show a  $\leq$  top by (auto intro: ccInf-greatest simp only: ccInf-empty [symmetric])
qed

lemma ccINF-lower: countable A  $\implies i \in A \implies (\text{INF } i \in A. f i) \leq f i$ 
  using ccInf-lower [of f ‘ A] by simp

lemma ccINF-greatest: countable A  $\implies (\bigwedge i. i \in A \implies u \leq f i) \implies u \leq (\text{INF } i \in A. f i)$ 
  using ccInf-greatest [of f ‘ A] by auto

lemma ccSUP-upper: countable A  $\implies i \in A \implies f i \leq (\text{SUP } i \in A. f i)$ 
  using ccSup-upper [of f ‘ A] by simp

lemma ccSUP-least: countable A  $\implies (\bigwedge i. i \in A \implies f i \leq u) \implies (\text{SUP } i \in A. f i) \leq u$ 
  using ccSup-least [of f ‘ A] by auto

lemma ccInf-lower2: countable A  $\implies u \in A \implies u \leq v \implies \text{Inf } A \leq v$ 
  using ccInf-lower [of A u] by auto

lemma ccINF-lower2: countable A  $\implies i \in A \implies f i \leq u \implies (\text{INF } i \in A. f i) \leq u$ 
  using ccINF-lower [of A i f] by auto

lemma ccSup-upper2: countable A  $\implies u \in A \implies v \leq u \implies v \leq \text{Sup } A$ 
  using ccSup-upper [of A u] by auto

lemma ccSUP-upper2: countable A  $\implies i \in A \implies u \leq f i \implies u \leq (\text{SUP } i \in A. f i)$ 
  using ccSUP-upper [of A i f] by auto

lemma le-ccInf-iff: countable A  $\implies b \leq \text{Inf } A \iff (\forall a \in A. b \leq a)$ 
  by (auto intro: ccInf-greatest dest: ccInf-lower)

lemma le-ccINF-iff: countable A  $\implies u \leq (\text{INF } i \in A. f i) \iff (\forall i \in A. u \leq f i)$ 

```

using *le-ccInf-iff* [of $f \text{ ‘ } A$] **by** *simp*

lemma *ccSup-le-iff*: *countable* $A \implies \text{Sup } A \leq b \longleftrightarrow (\forall a \in A. a \leq b)$
by (*auto intro: ccSup-least dest: ccSup-upper*)

lemma *ccSUP-le-iff*: *countable* $A \implies (\text{SUP } i \in A. f i) \leq u \longleftrightarrow (\forall i \in A. f i \leq u)$
using *ccSup-le-iff* [of $f \text{ ‘ } A$] **by** *simp*

lemma *ccInf-insert* [*simp*]: *countable* $A \implies \text{Inf } (\text{insert } a A) = \text{inf } a (\text{Inf } A)$
by (*force intro: le-infI le-infI1 le-infI2 order.antisym ccInf-greatest ccInf-lower*)

lemma *ccINF-insert* [*simp*]: *countable* $A \implies (\text{INF } x \in \text{insert } a A. f x) = \text{inf } (f a) (\text{Inf } (f \text{ ‘ } A))$
unfolding *image-insert* **by** *simp*

lemma *ccSup-insert* [*simp*]: *countable* $A \implies \text{Sup } (\text{insert } a A) = \text{sup } a (\text{Sup } A)$
by (*force intro: le-supI le-supI1 le-supI2 order.antisym ccSup-least ccSup-upper*)

lemma *ccSUP-insert* [*simp*]: *countable* $A \implies (\text{SUP } x \in \text{insert } a A. f x) = \text{sup } (f a) (\text{Sup } (f \text{ ‘ } A))$
unfolding *image-insert* **by** *simp*

lemma *ccINF-empty* [*simp*]: $(\text{INF } x \in \{\}. f x) = \text{top}$
unfolding *image-empty* **by** *simp*

lemma *ccSUP-empty* [*simp*]: $(\text{SUP } x \in \{\}. f x) = \text{bot}$
unfolding *image-empty* **by** *simp*

lemma *ccInf-superset-mono*: *countable* $A \implies B \subseteq A \implies \text{Inf } A \leq \text{Inf } B$
by (*auto intro: ccInf-greatest ccInf-lower countable-subset*)

lemma *ccSup-subset-mono*: *countable* $B \implies A \subseteq B \implies \text{Sup } A \leq \text{Sup } B$
by (*auto intro: ccSup-least ccSup-upper countable-subset*)

lemma *ccInf-mono*:
assumes [*intro*]: *countable* B *countable* A
assumes $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$
shows $\text{Inf } A \leq \text{Inf } B$
proof (*rule ccInf-greatest*)
fix b **assume** $b \in B$
with *assms* **obtain** a **where** $a \in A$ **and** $a \leq b$ **by** *blast*
from $\langle a \in A \rangle$ **have** $\text{Inf } A \leq a$ **by** (*rule ccInf-lower[rotated]*) *auto*
with $\langle a \leq b \rangle$ **show** $\text{Inf } A \leq b$ **by** *auto*
qed *auto*

lemma *ccINF-mono*:
countable $A \implies \text{countable } B \implies (\bigwedge m. m \in B \implies \exists n \in A. f n \leq g m) \implies (\text{INF } n \in A. f n) \leq (\text{INF } n \in B. g n)$
using *ccInf-mono* [of $g \text{ ‘ } B f \text{ ‘ } A$] **by** *auto*

lemma *ccSup-mono*:

assumes [intro]: *countable B countable A*
assumes $\bigwedge a. a \in A \implies \exists b \in B. a \leq b$
shows $\text{Sup } A \leq \text{Sup } B$
proof (rule *ccSup-least*)
fix *a* **assume** $a \in A$
with *assms* **obtain** *b* **where** $b \in B$ **and** $a \leq b$ **by** *blast*
from $\langle b \in B \rangle$ **have** $b \leq \text{Sup } B$ **by** (rule *ccSup-upper[rotated]*) *auto*
with $\langle a \leq b \rangle$ **show** $a \leq \text{Sup } B$ **by** *auto*
qed *auto*

lemma *ccSUP-mono*:

countable A \implies countable B \implies $(\bigwedge n. n \in A \implies \exists m \in B. f n \leq g m) \implies (\text{SUP } n \in A. f n) \leq (\text{SUP } n \in B. g n)$
using *ccSup-mono* [of *g ‘ B f ‘ A*] **by** *auto*

lemma *ccINF-superset-mono*:

countable A $\implies B \subseteq A \implies$ $(\bigwedge x. x \in B \implies f x \leq g x) \implies (\text{INF } x \in A. f x) \leq (\text{INF } x \in B. g x)$
by (*blast intro: ccINF-mono countable-subset dest: subsetD*)

lemma *ccSUP-subset-mono*:

countable B $\implies A \subseteq B \implies$ $(\bigwedge x. x \in A \implies f x \leq g x) \implies (\text{SUP } x \in A. f x) \leq (\text{SUP } x \in B. g x)$
by (*blast intro: ccSUP-mono countable-subset dest: subsetD*)

lemma *less-eq-ccInf-inter*: *countable A \implies countable B $\implies \text{sup } (\text{Inf } A) (\text{Inf } B) \leq \text{Inf } (A \cap B)$*

by (*auto intro: ccInf-greatest ccInf-lower*)

lemma *ccSup-inter-less-eq*: *countable A \implies countable B $\implies \text{Sup } (A \cap B) \leq \text{inf } (\text{Sup } A) (\text{Sup } B)$*

by (*auto intro: ccSup-least ccSup-upper*)

lemma *ccInf-union-distrib*: *countable A \implies countable B $\implies \text{Inf } (A \cup B) = \text{inf } (\text{Inf } A) (\text{Inf } B)$*

by (rule *order.antisym*) (*auto intro: ccInf-greatest ccInf-lower le-infI1 le-infI2*)

lemma *ccINF-union*:

countable A \implies countable B $\implies (\text{INF } i \in A \cup B. M i) = \text{inf } (\text{INF } i \in A. M i) (\text{INF } i \in B. M i)$

by (*auto intro!: order.antisym ccINF-mono intro: le-infI1 le-infI2 ccINF-greatest ccINF-lower*)

lemma *ccSup-union-distrib*: *countable A \implies countable B $\implies \text{Sup } (A \cup B) = \text{sup } (\text{Sup } A) (\text{Sup } B)$*

by (rule *order.antisym*) (*auto intro: ccSup-least ccSup-upper le-supI1 le-supI2*)

lemma *ccSUP-union*:

countable A \implies countable B \implies (SUP $i \in A \cup B$. M i) = sup (SUP $i \in A$. M i) (SUP $i \in B$. M i)
by (auto intro!: order.antisym ccSUP-mono intro: le-supI1 le-supI2 ccSUP-least ccSUP-upper)

lemma *ccINF-inf-distrib*: *countable A \implies inf (INF $a \in A$. f a) (INF $a \in A$. g a) = (INF $a \in A$. inf (f a) (g a))*

by (rule order.antisym) (rule ccINF-greatest, auto intro: le-infI1 le-infI2 ccINF-lower ccINF-mono)

lemma *ccSUP-sup-distrib*: *countable A \implies sup (SUP $a \in A$. f a) (SUP $a \in A$. g a) = (SUP $a \in A$. sup (f a) (g a))*

by (rule order.antisym[rotated]) (rule ccSUP-least, auto intro: le-supI1 le-supI2 ccSUP-upper ccSUP-mono)

lemma *ccINF-const [simp]*: *A $\neq \{\}$ \implies (INF $i \in A$. f) = f*
unfolding *image-constant-conv* **by** *auto*

lemma *ccSUP-const [simp]*: *A $\neq \{\}$ \implies (SUP $i \in A$. f) = f*
unfolding *image-constant-conv* **by** *auto*

lemma *ccINF-top [simp]*: *(INF $x \in A$. top) = top*
by (cases A = $\{\}$) *simp-all*

lemma *ccSUP-bot [simp]*: *(SUP $x \in A$. bot) = bot*
by (cases A = $\{\}$) *simp-all*

lemma *ccINF-commute*: *countable A \implies countable B \implies (INF $i \in A$. INF $j \in B$. f i j) = (INF $j \in B$. INF $i \in A$. f i j)*
by (iprover intro: ccINF-lower ccINF-greatest order-trans order.antisym)

lemma *ccSUP-commute*: *countable A \implies countable B \implies (SUP $i \in A$. SUP $j \in B$. f i j) = (SUP $j \in B$. SUP $i \in A$. f i j)*
by (iprover intro: ccSUP-upper ccSUP-least order-trans order.antisym)

end

context

fixes *a :: 'a::countable-complete-lattice, linorder*
begin

lemma *less-ccSup-iff*: *countable S \implies a < Sup S \longleftrightarrow ($\exists x \in S$. a < x)*
unfolding *not-le [symmetric]* **by** (subst ccSup-le-iff) *auto*

lemma *less-ccSUP-iff*: *countable A \implies a < (SUP $i \in A$. f i) \longleftrightarrow ($\exists x \in A$. a < f x)*
using *less-ccSup-iff [of f ' A]* **by** *simp*

lemma *ccInf-less-iff*: $\text{countable } S \implies \text{Inf } S < a \longleftrightarrow (\exists x \in S. x < a)$
unfolding *not-le [symmetric]* **by** (*subst le-ccInf-iff*) *auto*

lemma *ccINF-less-iff*: $\text{countable } A \implies (\text{INF } i \in A. f i) < a \longleftrightarrow (\exists x \in A. f x < a)$
using *ccInf-less-iff [of f ‘ A]* **by** *simp*

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
assumes *sup-ccInf*: $\text{countable } B \implies \text{sup } a (\text{Inf } B) = (\text{INF } b \in B. \text{sup } a b)$
assumes *inf-ccSup*: $\text{countable } B \implies \text{inf } a (\text{Sup } B) = (\text{SUP } b \in B. \text{inf } a b)$
begin

lemma *sup-ccINF*:
 $\text{countable } B \implies \text{sup } a (\text{INF } b \in B. f b) = (\text{INF } b \in B. \text{sup } a (f b))$
by (*simp only: sup-ccInf image-image countable-image*)

lemma *inf-ccSUP*:
 $\text{countable } B \implies \text{inf } a (\text{SUP } b \in B. f b) = (\text{SUP } b \in B. \text{inf } a (f b))$
by (*simp only: inf-ccSup image-image countable-image*)

subclass *distrib-lattice*

proof

fix *a b c*

from *sup-ccInf*[*of {b, c} a*] **have** $\text{sup } a (\text{Inf } \{b, c\}) = (\text{INF } d \in \{b, c\}. \text{sup } a d)$
by *simp*

then show $\text{sup } a (\text{inf } b c) = \text{inf } (\text{sup } a b) (\text{sup } a c)$

by *simp*

qed

lemma *ccInf-sup*:
 $\text{countable } B \implies \text{sup } (\text{Inf } B) a = (\text{INF } b \in B. \text{sup } b a)$
by (*simp add: sup-ccInf sup-commute*)

lemma *ccSup-inf*:
 $\text{countable } B \implies \text{inf } (\text{Sup } B) a = (\text{SUP } b \in B. \text{inf } b a)$
by (*simp add: inf-ccSup inf-commute*)

lemma *ccINF-sup*:
 $\text{countable } B \implies \text{sup } (\text{INF } b \in B. f b) a = (\text{INF } b \in B. \text{sup } (f b) a)$
by (*simp add: sup-ccINF sup-commute*)

lemma *ccSUP-inf*:
 $\text{countable } B \implies \text{inf } (\text{SUP } b \in B. f b) a = (\text{SUP } b \in B. \text{inf } (f b) a)$
by (*simp add: inf-ccSUP inf-commute*)

lemma *ccINF-sup-distrib2*:
 $\text{countable } A \implies \text{countable } B \implies \text{sup } (\text{INF } a \in A. f a) (\text{INF } b \in B. g b) = (\text{INF } a \in A. \text{sup } (\text{INF } b \in B. f a) (g b))$

by (*subst ccINF-commute*) (*simp-all add: sup-ccINF ccINF-sup*)

lemma *ccSUP-inf-distrib2*:

countable A \implies countable B \implies inf (SUP a \in A. f a) (SUP b \in B. g b) = (SUP a \in A. SUP b \in B. inf (f a) (g b))

by (*subst ccSUP-commute*) (*simp-all add: inf-ccSUP ccSUP-inf*)

context

fixes *f* :: 'a \Rightarrow 'b::countable-complete-lattice

assumes *mono f*

begin

lemma *mono-ccInf*:

countable A \implies f (Inf A) \leq (INF x \in A. f x)

using \langle *mono f* \rangle

by (*auto intro!: countable-complete-lattice-class.ccINF-greatest intro: ccInf-lower dest: monoD*)

lemma *mono-ccSup*:

countable A \implies (SUP x \in A. f x) \leq f (Sup A)

using \langle *mono f* \rangle **by** (*auto intro: countable-complete-lattice-class.ccSUP-least cc-Sup-upper dest: monoD*)

lemma *mono-ccINF*:

countable I \implies f (INF i \in I. A i) \leq (INF x \in I. f (A x))

by (*intro countable-complete-lattice-class.ccINF-greatest monoD[OF \langle mono f \rangle] ccINF-lower*)

lemma *mono-ccSUP*:

countable I \implies (SUP x \in I. f (A x)) \leq f (SUP i \in I. A i)

by (*intro countable-complete-lattice-class.ccSUP-least monoD[OF \langle mono f \rangle] cc-SUP-upper*)

end

end

22.0.1 Instances of countable complete lattices

instance *fun* :: (type, countable-complete-lattice) countable-complete-lattice

by *standard*

(*auto simp: le-fun-def intro!: ccSUP-upper ccSUP-least ccINF-lower ccINF-greatest*)

subclass (**in** *complete-lattice*) countable-complete-lattice

by *standard* (*auto intro: Sup-upper Sup-least Inf-lower Inf-greatest*)

subclass (**in** *complete-distrib-lattice*) countable-complete-distrib-lattice

by *standard* (*auto intro: sup-Inf inf-Sup*)

end

23 Type of (at Most) Countable Sets

```
theory Countable-Set-Type
imports Countable-Set
begin
```

23.1 Cardinal stuff

```
context
  includes cardinal-syntax
begin
```

```
lemma countable-card-of-nat: countable A  $\longleftrightarrow$   $|A| \leq_o |UNIV::nat\ set|$ 
  unfolding countable-def card-of-ordLeq[symmetric] by auto
```

```
lemma countable-card-le-natLeq: countable A  $\longleftrightarrow$   $|A| \leq_o natLeq$ 
  unfolding countable-card-of-nat using card-of-nat ordLeq-ordIso-trans ordIso-symmetric
  by blast
```

```
lemma countable-or-card-of:
  assumes countable A
  shows  $(finite\ A \wedge |A| <_o |UNIV::nat\ set|) \vee$ 
     $(infinite\ A \wedge |A| =_o |UNIV::nat\ set|)$ 
  by (metis assms countable-card-of-nat infinite-iff-card-of-nat ordIso-iff-ordLeq
    ordLeq-iff-ordLess-or-ordIso)
```

```
lemma countable-cases-card-of[elim]:
  assumes countable A
  obtains  $(Fin)\ finite\ A\ |A| <_o |UNIV::nat\ set|$ 
     $| (Inf)\ infinite\ A\ |A| =_o |UNIV::nat\ set|$ 
  using assms countable-or-card-of by blast
```

```
lemma countable-or:
  countable A  $\implies (\exists f::'a \Rightarrow nat.\ finite\ A \wedge inj-on\ f\ A) \vee (\exists f::'a \Rightarrow nat.\ infinite\ A$ 
 $\wedge\ bij-betw\ f\ A\ UNIV)$ 
  by (elim countable-enum-cases) fastforce+
```

```
lemma countable-cases[elim]:
  assumes countable A
  obtains  $(Fin)\ f::'a \Rightarrow nat\ \textbf{where}\ finite\ A\ inj-on\ f\ A$ 
     $| (Inf)\ f::'a \Rightarrow nat\ \textbf{where}\ infinite\ A\ bij-betw\ f\ A\ UNIV$ 
  using assms countable-or by metis
```

```
lemma countable-ordLeq:
  assumes  $|A| \leq_o |B|$  and countable B
  shows countable A
  using assms unfolding countable-card-of-nat by(rule ordLeq-transitive)
```

```

lemma countable-ordLess:
assumes AB:  $|A| < o |B|$  and B: countable B
shows countable A
using countable-ordLeq[OF ordLess-imp-ordLeq[OF AB] B] .

end

```

23.2 The type of countable sets

```

typedef 'a cset = {A :: 'a set. countable A} morphisms rset acset
by (rule exI[of - {}]) simp

```

```

setup-lifting type-definition-cset

```

```

declare
  rset-inverse[simp]
  acset-inverse[Transfer.transferred, unfolded mem-Collect-eq, simp]
  acset-inject[Transfer.transferred, unfolded mem-Collect-eq, simp]
  rset[Transfer.transferred, unfolded mem-Collect-eq, simp]

```

```

instantiation cset :: (type) {bounded-lattice-bot, distrib-lattice, minus}
begin

```

```

lift-definition bot-cset :: 'a cset is {} parametric empty-transfer by simp

```

```

lift-definition less-eq-cset :: 'a cset  $\Rightarrow$  'a cset  $\Rightarrow$  bool
is subset-eq parametric subset-transfer .

```

```

definition less-cset :: 'a cset  $\Rightarrow$  'a cset  $\Rightarrow$  bool
where  $xs < ys \equiv xs \leq ys \wedge xs \neq (ys::'a\ cset)$ 

```

```

lemma less-cset-transfer[transfer-rule]:
includes lifting-syntax
assumes [transfer-rule]: bi-unique A
shows ((pcr-cset A) ==> (pcr-cset A) ==> (=)) ( $\subset$ ) ( $<$ )
unfolding less-cset-def[abs-def] psubset-eq[abs-def] by transfer-prover

```

```

lift-definition sup-cset :: 'a cset  $\Rightarrow$  'a cset  $\Rightarrow$  'a cset
is union parametric union-transfer by simp

```

```

lift-definition inf-cset :: 'a cset  $\Rightarrow$  'a cset  $\Rightarrow$  'a cset
is inter parametric inter-transfer by simp

```

```

lift-definition minus-cset :: 'a cset  $\Rightarrow$  'a cset  $\Rightarrow$  'a cset
is minus parametric Diff-transfer by simp

```

```

instance by standard (transfer; auto)+

```

end

abbreviation *cempty* :: 'a cset **where** *cempty* \equiv *bot*

abbreviation *csubset-eq* :: 'a cset \Rightarrow 'a cset \Rightarrow bool **where** *csubset-eq* *xs ys* \equiv *xs* \leq *ys*

abbreviation *csubset* :: 'a cset \Rightarrow 'a cset \Rightarrow bool **where** *csubset* *xs ys* \equiv *xs* < *ys*

abbreviation *cUn* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset **where** *cUn* *xs ys* \equiv *sup xs ys*

abbreviation *cInt* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset **where** *cInt* *xs ys* \equiv *inf xs ys*

abbreviation *cDiff* :: 'a cset \Rightarrow 'a cset \Rightarrow 'a cset **where** *cDiff* *xs ys* \equiv *minus xs ys*

lift-definition *cin* :: 'a \Rightarrow 'a cset \Rightarrow bool **is** (\in) **parametric** *member-transfer*

lift-definition *cinsert* :: 'a \Rightarrow 'a cset \Rightarrow 'a cset **is** *insert* **parametric** *Lifting-Set.insert-transfer*
by (*rule countable-insert*)

abbreviation *csingle* :: 'a \Rightarrow 'a cset **where** *csingle* *x* \equiv *cinsert x cempty*

lift-definition *cimage* :: ('a \Rightarrow 'b) \Rightarrow 'a cset \Rightarrow 'b cset **is** (') **parametric** *image-transfer*
by (*rule countable-image*)

lift-definition *cBall* :: 'a cset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **is** *Ball* **parametric** *Ball-transfer*

lift-definition *cBex* :: 'a cset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **is** *Bex* **parametric** *Bex-transfer*

lift-definition *cUnion* :: 'a cset cset \Rightarrow 'a cset **is** *Union* **parametric** *Union-transfer*
using *countable-UN* [*of - id*] **by** *auto*

abbreviation (*input*) *cUNION* :: 'a cset \Rightarrow ('a \Rightarrow 'b cset) \Rightarrow 'b cset
where *cUNION* *A f* \equiv *cUnion (cimage f A)*

lemma *Union-conv-UNION*: $\bigcup A = \bigcup (id \text{ ` } A)$
by *simp*

lemmas *cset-eqI* = *set-eqI*[*Transfer.transferred*]

lemmas *cset-eq-iff*[*no-atp*] = *set-eq-iff*[*Transfer.transferred*]

lemmas *cBallI*[*intro!*] = *ballI*[*Transfer.transferred*]

lemmas *cbspec*[*dest?*] = *bspec*[*Transfer.transferred*]

lemmas *cBallE*[*elim*] = *ballE*[*Transfer.transferred*]

lemmas *cBexI*[*intro*] = *bexI*[*Transfer.transferred*]

lemmas *rev-cBexI*[*intro?*] = *rev-bexI*[*Transfer.transferred*]

lemmas *cBexCI* = *bexCI*[*Transfer.transferred*]

lemmas *cBexE*[*elim!*] = *bexE*[*Transfer.transferred*]

lemmas *cBall-triv*[*simp*] = *ball-triv*[*Transfer.transferred*]

lemmas *cBex-triv*[*simp*] = *bex-triv*[*Transfer.transferred*]

lemmas *cBex-triv-one-point1*[*simp*] = *bex-triv-one-point1*[*Transfer.transferred*]

lemmas *cBex-triv-one-point2*[*simp*] = *bex-triv-one-point2*[*Transfer.transferred*]

lemmas *cBex-one-point1*[*simp*] = *bex-one-point1*[*Transfer.transferred*]

lemmas *cBex-one-point2*[*simp*] = *bex-one-point2*[*Transfer.transferred*]

lemmas *cBall-one-point1*[*simp*] = *ball-one-point1*[*Transfer.transferred*]

lemmas *cBall-one-point2*[*simp*] = *ball-one-point2*[*Transfer.transferred*]

lemmas *cBall-conj-distrib* = *ball-conj-distrib*[*Transfer.transferred*]

```

lemmas cBex-disj-distrib = bex-disj-distrib[Transfer.transferred]
lemmas cBall-cong = ball-cong[Transfer.transferred]
lemmas cBex-cong = bex-cong[Transfer.transferred]
lemmas csubsetI[intro!] = subsetI[Transfer.transferred]
lemmas csubsetD[elim, intro?] = subsetD[Transfer.transferred]
lemmas rev-csubsetD[no-atp, intro?] = rev-subsetD[Transfer.transferred]
lemmas csubsetCE[no-atp, elim] = subsetCE[Transfer.transferred]
lemmas csubset-eq[no-atp] = subset-eq[Transfer.transferred]
lemmas contra-csubsetD[no-atp] = contra-subsetD[Transfer.transferred]
lemmas csubset-refl = subset-refl[Transfer.transferred]
lemmas csubset-trans = subset-trans[Transfer.transferred]
lemmas cset-rev-mp = rev-subsetD[Transfer.transferred]
lemmas cset-mp = subsetD[Transfer.transferred]
lemmas csubset-not-fsubset-eq[code] = subset-not-subset-eq[Transfer.transferred]
lemmas eq-cmem-trans = eq-mem-trans[Transfer.transferred]
lemmas csubset-antisym[intro!] = subset-antisym[Transfer.transferred]
lemmas cequalityD1 = equalityD1[Transfer.transferred]
lemmas cequalityD2 = equalityD2[Transfer.transferred]
lemmas cequalityE = equalityE[Transfer.transferred]
lemmas cequalityCE[elim] = equalityCE[Transfer.transferred]
lemmas eqcset-imp-iff = eqset-imp-iff[Transfer.transferred]
lemmas eqcelem-imp-iff = egelem-imp-iff[Transfer.transferred]
lemmas cempty-iff[simp] = empty-iff[Transfer.transferred]
lemmas cempty-fsubsetI[iff] = empty-subsetI[Transfer.transferred]
lemmas equals-cemptyI = equalsOI[Transfer.transferred]
lemmas equals-cemptyD = equalsOD[Transfer.transferred]
lemmas cBall-cempty[simp] = ball-empty[Transfer.transferred]
lemmas cBex-cempty[simp] = bex-empty[Transfer.transferred]
lemmas cInt-iff[simp] = Int-iff[Transfer.transferred]
lemmas cIntI[intro!] = IntI[Transfer.transferred]
lemmas cIntD1 = IntD1[Transfer.transferred]
lemmas cIntD2 = IntD2[Transfer.transferred]
lemmas cIntE[elim!] = IntE[Transfer.transferred]
lemmas cUn-iff[simp] = Un-iff[Transfer.transferred]
lemmas cUnI1[elim?] = UnI1[Transfer.transferred]
lemmas cUnI2[elim?] = UnI2[Transfer.transferred]
lemmas cUnCI[intro!] = UnCI[Transfer.transferred]
lemmas cuUnE[elim!] = UnE[Transfer.transferred]
lemmas cDiff-iff[simp] = Diff-iff[Transfer.transferred]
lemmas cDiffI[intro!] = DiffI[Transfer.transferred]
lemmas cDiffD1 = DiffD1[Transfer.transferred]
lemmas cDiffD2 = DiffD2[Transfer.transferred]
lemmas cDiffE[elim!] = DiffE[Transfer.transferred]
lemmas cinsert-iff[simp] = insert-iff[Transfer.transferred]
lemmas cinsertI1 = insertI1[Transfer.transferred]
lemmas cinsertI2 = insertI2[Transfer.transferred]
lemmas cinsertE[elim!] = insertE[Transfer.transferred]
lemmas cinsertCI[intro!] = insertCI[Transfer.transferred]
lemmas csubset-cinsert-iff = subset-insert-iff[Transfer.transferred]

```

```

lemmas cinsert-ident = insert-ident[Transfer.transferred]
lemmas csingletonI[intro!,no-atp] = singletonI[Transfer.transferred]
lemmas csingletonD[dest!,no-atp] = singletonD[Transfer.transferred]
lemmas fsingletonE = csingletonD [elim-format]
lemmas csingleton-iff = singleton-iff[Transfer.transferred]
lemmas csingleton-inject[dest!] = singleton-inject[Transfer.transferred]
lemmas csingleton-finsert-inj-eq[iff,no-atp] = singleton-insert-inj-eq[Transfer.transferred]
lemmas csingleton-finsert-inj-eq'[iff,no-atp] = singleton-insert-inj-eq'[Transfer.transferred]
lemmas csubset-csingletonD = subset-singletonD[Transfer.transferred]
lemmas cDiff-single-cinsert = Diff-single-insert[Transfer.transferred]
lemmas cdoubleton-eq-iff = doubleton-eq-iff[Transfer.transferred]
lemmas cUn-csingleton-iff = Un-singleton-iff[Transfer.transferred]
lemmas csingleton-cUn-iff = singleton-Un-iff[Transfer.transferred]
lemmas cimage-eqI[simp, intro] = image-eqI[Transfer.transferred]
lemmas cimageI = imageI[Transfer.transferred]
lemmas rev-cimage-eqI = rev-image-eqI[Transfer.transferred]
lemmas cimageE[elim!] = imageE[Transfer.transferred]
lemmas Compr-cimage-eq = Compr-image-eq[Transfer.transferred]
lemmas cimage-cUn = image-Un[Transfer.transferred]
lemmas cimage-iff = image-iff[Transfer.transferred]
lemmas cimage-csubset-iff[no-atp] = image-subset-iff[Transfer.transferred]
lemmas cimage-csubsetI = image-subsetI[Transfer.transferred]
lemmas cimage-ident[simp] = image-ident[Transfer.transferred]
lemmas if-split-cin1 = if-split-mem1[Transfer.transferred]
lemmas if-split-cin2 = if-split-mem2[Transfer.transferred]
lemmas cpsubsetI[intro!,no-atp] = psubsetI[Transfer.transferred]
lemmas cpsubsetE[elim!,no-atp] = psubsetE[Transfer.transferred]
lemmas cpsubset-finsert-iff = psubset-insert-iff[Transfer.transferred]
lemmas cpsubset-eq = psubset-eq[Transfer.transferred]
lemmas cpsubset-imp-fsubset = psubset-imp-subset[Transfer.transferred]
lemmas cpsubset-trans = psubset-trans[Transfer.transferred]
lemmas cpsubsetD = psubsetD[Transfer.transferred]
lemmas cpsubset-csubset-trans = psubset-subset-trans[Transfer.transferred]
lemmas csubset-cpsubset-trans = subset-psubset-trans[Transfer.transferred]
lemmas cpsubset-imp-ex-fmem = psubset-imp-ex-mem[Transfer.transferred]
lemmas csubset-cinsertI = subset-insertI[Transfer.transferred]
lemmas csubset-cinsertI2 = subset-insertI2[Transfer.transferred]
lemmas csubset-cinsert = subset-insert[Transfer.transferred]
lemmas cUn-upper1 = Un-upper1[Transfer.transferred]
lemmas cUn-upper2 = Un-upper2[Transfer.transferred]
lemmas cUn-least = Un-least[Transfer.transferred]
lemmas cInt-lower1 = Int-lower1[Transfer.transferred]
lemmas cInt-lower2 = Int-lower2[Transfer.transferred]
lemmas cInt-greatest = Int-greatest[Transfer.transferred]
lemmas cDiff-csubset = Diff-subset[Transfer.transferred]
lemmas cDiff-csubset-conv = Diff-subset-conv[Transfer.transferred]
lemmas csubset-cempty[simp] = subset-empty[Transfer.transferred]
lemmas not-cpsubset-cempty[iff] = not-psubset-empty[Transfer.transferred]
lemmas cinsert-is-cUn = insert-is-Un[Transfer.transferred]

```

```

lemmas cinsert-not-empty[simp] = insert-not-empty[Transfer.transferred]
lemmas empty-not-cinsert = empty-not-insert[Transfer.transferred]
lemmas cinsert-absorb = insert-absorb[Transfer.transferred]
lemmas cinsert-absorb2[simp] = insert-absorb2[Transfer.transferred]
lemmas cinsert-commute = insert-commute[Transfer.transferred]
lemmas cinsert-csubset[simp] = insert-subset[Transfer.transferred]
lemmas cinsert-cinter-cinsert[simp] = insert-inter-insert[Transfer.transferred]
lemmas cinsert-disjoint[simp,no-atp] = insert-disjoint[Transfer.transferred]
lemmas disjoint-cinsert[simp,no-atp] = disjoint-insert[Transfer.transferred]
lemmas cimage-empty[simp] = image-empty[Transfer.transferred]
lemmas cimage-cinsert[simp] = image-insert[Transfer.transferred]
lemmas cimage-constant = image-constant[Transfer.transferred]
lemmas cimage-constant-conv = image-constant-conv[Transfer.transferred]
lemmas cimage-cimage = image-image[Transfer.transferred]
lemmas cinsert-cimage[simp] = insert-image[Transfer.transferred]
lemmas cimage-is-empty[iff] = image-is-empty[Transfer.transferred]
lemmas empty-is-cimage[iff] = empty-is-image[Transfer.transferred]
lemmas cimage-cong = image-cong[Transfer.transferred]
lemmas cimage-cInt-csubset = image-Int-subset[Transfer.transferred]
lemmas cimage-cDiff-csubset = image-diff-subset[Transfer.transferred]
lemmas cInt-absorb = Int-absorb[Transfer.transferred]
lemmas cInt-left-absorb = Int-left-absorb[Transfer.transferred]
lemmas cInt-commute = Int-commute[Transfer.transferred]
lemmas cInt-left-commute = Int-left-commute[Transfer.transferred]
lemmas cInt-assoc = Int-assoc[Transfer.transferred]
lemmas cInt-ac = Int-ac[Transfer.transferred]
lemmas cInt-absorb1 = Int-absorb1[Transfer.transferred]
lemmas cInt-absorb2 = Int-absorb2[Transfer.transferred]
lemmas cInt-empty-left = Int-empty-left[Transfer.transferred]
lemmas cInt-empty-right = Int-empty-right[Transfer.transferred]
lemmas disjoint-iff-cnot-equal = disjoint-iff-not-equal[Transfer.transferred]
lemmas cInt-cUn-distrib = Int-Un-distrib[Transfer.transferred]
lemmas cInt-cUn-distrib2 = Int-Un-distrib2[Transfer.transferred]
lemmas cInt-csubset-iff[no-atp, simp] = Int-subset-iff[Transfer.transferred]
lemmas cUn-absorb = Un-absorb[Transfer.transferred]
lemmas cUn-left-absorb = Un-left-absorb[Transfer.transferred]
lemmas cUn-commute = Un-commute[Transfer.transferred]
lemmas cUn-left-commute = Un-left-commute[Transfer.transferred]
lemmas cUn-assoc = Un-assoc[Transfer.transferred]
lemmas cUn-ac = Un-ac[Transfer.transferred]
lemmas cUn-absorb1 = Un-absorb1[Transfer.transferred]
lemmas cUn-absorb2 = Un-absorb2[Transfer.transferred]
lemmas cUn-empty-left = Un-empty-left[Transfer.transferred]
lemmas cUn-empty-right = Un-empty-right[Transfer.transferred]
lemmas cUn-cinsert-left[simp] = Un-insert-left[Transfer.transferred]
lemmas cUn-cinsert-right[simp] = Un-insert-right[Transfer.transferred]
lemmas cInt-cinsert-left = Int-insert-left[Transfer.transferred]
lemmas cInt-cinsert-left-if0[simp] = Int-insert-left-if0[Transfer.transferred]
lemmas cInt-cinsert-left-if1[simp] = Int-insert-left-if1[Transfer.transferred]

```

```

lemmas cInt-cinsert-right = Int-insert-right[Transfer.transferred]
lemmas cInt-cinsert-right-if0[simp] = Int-insert-right-if0[Transfer.transferred]
lemmas cInt-cinsert-right-if1[simp] = Int-insert-right-if1[Transfer.transferred]
lemmas cUn-cInt-distrib = Un-Int-distrib[Transfer.transferred]
lemmas cUn-cInt-distrib2 = Un-Int-distrib2[Transfer.transferred]
lemmas cUn-cInt-crazy = Un-Int-crazy[Transfer.transferred]
lemmas csubset-cUn-eq = subset-Un-eq[Transfer.transferred]
lemmas cUn-cempty[iff] = Un-empty[Transfer.transferred]
lemmas cUn-csubset-iff[no-atp, simp] = Un-subset-iff[Transfer.transferred]
lemmas cUn-cDiff-cInt = Un-Diff-Int[Transfer.transferred]
lemmas cDiff-cInt2 = Diff-Int2[Transfer.transferred]
lemmas cUn-cInt-assoc-eq = Un-Int-assoc-eq[Transfer.transferred]
lemmas cBall-cUn = ball-Un[Transfer.transferred]
lemmas cBex-cUn = bex-Un[Transfer.transferred]
lemmas cDiff-eq-cempty-iff[simp, no-atp] = Diff-eq-empty-iff[Transfer.transferred]
lemmas cDiff-cancel[simp] = Diff-cancel[Transfer.transferred]
lemmas cDiff-idemp[simp] = Diff-idemp[Transfer.transferred]
lemmas cDiff-triv = Diff-triv[Transfer.transferred]
lemmas cempty-cDiff[simp] = empty-Diff[Transfer.transferred]
lemmas cDiff-cempty[simp] = Diff-empty[Transfer.transferred]
lemmas cDiff-cinsert0[simp, no-atp] = Diff-insert0[Transfer.transferred]
lemmas cDiff-cinsert = Diff-insert[Transfer.transferred]
lemmas cDiff-cinsert2 = Diff-insert2[Transfer.transferred]
lemmas cinsert-cDiff-if = insert-Diff-if[Transfer.transferred]
lemmas cinsert-cDiff1[simp] = insert-Diff1[Transfer.transferred]
lemmas cinsert-cDiff-single[simp] = insert-Diff-single[Transfer.transferred]
lemmas cinsert-cDiff = insert-Diff[Transfer.transferred]
lemmas cDiff-cinsert-absorb = Diff-insert-absorb[Transfer.transferred]
lemmas cDiff-disjoint[simp] = Diff-disjoint[Transfer.transferred]
lemmas cDiff-partition = Diff-partition[Transfer.transferred]
lemmas double-cDiff = double-diff[Transfer.transferred]
lemmas cUn-cDiff-cancel[simp] = Un-Diff-cancel[Transfer.transferred]
lemmas cUn-cDiff-cancel2[simp] = Un-Diff-cancel2[Transfer.transferred]
lemmas cDiff-cUn = Diff-Un[Transfer.transferred]
lemmas cDiff-cInt = Diff-Int[Transfer.transferred]
lemmas cUn-cDiff = Un-Diff[Transfer.transferred]
lemmas cInt-cDiff = Int-Diff[Transfer.transferred]
lemmas cDiff-cInt-distrib = Diff-Int-distrib[Transfer.transferred]
lemmas cDiff-cInt-distrib2 = Diff-Int-distrib2[Transfer.transferred]
lemmas cset-eq-csubset = set-eq-subset[Transfer.transferred]
lemmas csubset-iff[no-atp] = subset-iff[Transfer.transferred]
lemmas csubset-iff-psubset-eq = subset-iff-psubset-eq[Transfer.transferred]
lemmas all-not-cin-conv[simp] = all-not-in-conv[Transfer.transferred]
lemmas ex-cin-conv = ex-in-conv[Transfer.transferred]
lemmas cimage-mono = image-mono[Transfer.transferred]
lemmas cinsert-mono = insert-mono[Transfer.transferred]
lemmas cunion-mono = Un-mono[Transfer.transferred]
lemmas cinter-mono = Int-mono[Transfer.transferred]
lemmas cminus-mono = Diff-mono[Transfer.transferred]

```

```

lemmas cin-mono = in-mono[Transfer.transferred]
lemmas cLeast-mono = Least-mono[Transfer.transferred]
lemmas cequalityI = equalityI[Transfer.transferred]
lemmas cUN-iff [simp] = UN-iff[Transfer.transferred]
lemmas cUN-I [intro] = UN-I[Transfer.transferred]
lemmas cUN-E [elim!] = UN-E[Transfer.transferred]
lemmas cUN-upper = UN-upper[Transfer.transferred]
lemmas cUN-least = UN-least[Transfer.transferred]
lemmas cUN-cinsert-distrib = UN-insert-distrib[Transfer.transferred]
lemmas cUN-empty [simp] = UN-empty[Transfer.transferred]
lemmas cUN-empty2 [simp] = UN-empty2[Transfer.transferred]
lemmas cUN-absorb = UN-absorb[Transfer.transferred]
lemmas cUN-cinsert [simp] = UN-insert[Transfer.transferred]
lemmas cUN-cUn [simp] = UN-Un[Transfer.transferred]
lemmas cUN-cUN-flatten = UN-UN-flatten[Transfer.transferred]
lemmas cUN-csubset-iff = UN-subset-iff[Transfer.transferred]
lemmas cUN-constant [simp] = UN-constant[Transfer.transferred]
lemmas cimage-cUnion = image-Union[Transfer.transferred]
lemmas cUNION-cempty-conv [simp] = UNION-empty-conv[Transfer.transferred]
lemmas cBall-cUN = ball-UN[Transfer.transferred]
lemmas cBex-cUN = bex-UN[Transfer.transferred]
lemmas cUn-eq-cUN = Un-eq-UN[Transfer.transferred]
lemmas cUN-mono = UN-mono[Transfer.transferred]
lemmas cimage-cUN = image-UN[Transfer.transferred]
lemmas cUN-csingleton [simp] = UN-singleton[Transfer.transferred]

```

23.3 Additional lemmas

23.3.1 empty

lemma emptyE [elim!]: $\text{cin } a \text{ empty} \implies P$ **by** simp

23.3.2 cinsert

lemma countable-insert-iff: $\text{countable } (\text{insert } x \ A) \longleftrightarrow \text{countable } A$
by (metis Diff-eq-empty-iff countable-empty countable-insert subset-insertI uncountable-minus-countable)

lemma set-cinsert:

assumes cin $x \ A$

obtains B **where** $A = \text{cinsert } x \ B$ **and** $\neg \text{cin } x \ B$

using assms **by** transfer(erule Set.set-insert, simp add: countable-insert-iff)

lemma mk-disjoint-cinsert: $\text{cin } a \ A \implies \exists B. A = \text{cinsert } a \ B \wedge \neg \text{cin } a \ B$
by (rule exI[where $x = \text{cDiff } A \ (\text{csingle } a)$]) blast

23.3.3 cimage

lemma subset-cimage-iff: $\text{csubset-eq } B \ (\text{cimage } f \ A) \longleftrightarrow (\exists AA. \text{csubset-eq } AA \ A \wedge B = \text{cimage } f \ AA)$

by *transfer (metis countable-subset image-mono mem-Collect-eq subset-imageE)*

23.3.4 bounded quantification

lemma *cBex-simps [simp, no-atp]:*

$$\begin{aligned} \bigwedge A P Q. cBex A (\lambda x. P x \wedge Q) &= (cBex A P \wedge Q) \\ \bigwedge A P Q. cBex A (\lambda x. P \wedge Q x) &= (P \wedge cBex A Q) \\ \bigwedge P. cBex empty P &= False \\ \bigwedge a B P. cBex (cinsert a B) P &= (P a \vee cBex B P) \\ \bigwedge A P f. cBex (cimage f A) P &= cBex A (\lambda x. P (f x)) \\ \bigwedge A P. (\neg cBex A P) &= cBall A (\lambda x. \neg P x) \end{aligned}$$

by *auto*

lemma *cBall-simps [simp, no-atp]:*

$$\begin{aligned} \bigwedge A P Q. cBall A (\lambda x. P x \vee Q) &= (cBall A P \vee Q) \\ \bigwedge A P Q. cBall A (\lambda x. P \vee Q x) &= (P \vee cBall A Q) \\ \bigwedge A P Q. cBall A (\lambda x. P \longrightarrow Q x) &= (P \longrightarrow cBall A Q) \\ \bigwedge A P Q. cBall A (\lambda x. P x \longrightarrow Q) &= (cBex A P \longrightarrow Q) \\ \bigwedge P. cBall empty P &= True \\ \bigwedge a B P. cBall (cinsert a B) P &= (P a \wedge cBall B P) \\ \bigwedge A P f. cBall (cimage f A) P &= cBall A (\lambda x. P (f x)) \\ \bigwedge A P. (\neg cBall A P) &= cBex A (\lambda x. \neg P x) \end{aligned}$$

by *auto*

lemma *atomize-cBall:*

$$(\bigwedge x. cin x A \implies P x) == Trueprop (cBall A (\lambda x. P x))$$

unfolding *atomize-all atomize-imp*

by *(rule equal-intr-rule; blast)*

23.3.5 cUnion

lemma *cUNION-cimage: cUNION (cimage f A) g = cUNION A (g o f)*

by *transfer simp*

23.4 Setup for Lifting/Transfer

23.4.1 Relator and predicator properties

lift-definition *rel-cset :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a cset \Rightarrow 'b cset \Rightarrow bool*

is *rel-set parametric rel-set-transfer .*

lemma *rel-cset-alt-def:*

$$\begin{aligned} rel-cset R a b &\iff \\ (\forall t \in rcset a. \exists u \in rcset b. R t u) \wedge \\ (\forall t \in rcset b. \exists u \in rcset a. R u t) \end{aligned}$$

by(*simp add: rel-cset-def rel-set-def*)

lemma *rel-cset-iff:*

$$\begin{aligned} rel-cset R a b &\iff \\ (\forall t. cin t a \longrightarrow (\exists u. cin u b \wedge R t u)) \wedge \end{aligned}$$

$(\forall t. \text{cin } t \ b \longrightarrow (\exists u. \text{cin } u \ a \wedge R \ u \ t))$
by *transfer(auto simp add: rel-set-def)*

lemma *rel-cset-cUNION*:

$\llbracket \text{rel-cset } Q \ A \ B; \text{rel-fun } Q \ (\text{rel-cset } R) \ f \ g \rrbracket$
 $\implies \text{rel-cset } R \ (\text{cUnion } (\text{cimage } f \ A)) \ (\text{cUnion } (\text{cimage } g \ B))$
unfolding *rel-fun-def* **by** *transfer(erule rel-set-UNION, simp add: rel-fun-def)*

lemma *rel-cset-csingle-iff* [*simp*]: $\text{rel-cset } R \ (\text{csingle } x) \ (\text{csingle } y) \longleftrightarrow R \ x \ y$
by *transfer(auto simp add: rel-set-def)*

23.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemmas *empty-parametric* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *cinsert-parametric* [*transfer-rule*]:
 $(A \implies \text{rel-cset } A \implies \text{rel-cset } A) \text{ cinsert cinsert}$
unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cUn-parametric* [*transfer-rule*]:
 $(\text{rel-cset } A \implies \text{rel-cset } A \implies \text{rel-cset } A) \text{ cUn cUn}$
unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cUnion-parametric* [*transfer-rule*]:
 $(\text{rel-cset } (\text{rel-cset } A) \implies \text{rel-cset } A) \text{ cUnion cUnion}$
unfolding *rel-fun-def*
by *transfer (auto simp: rel-set-def,metis+)*

lemma *cimage-parametric* [*transfer-rule*]:
 $((A \implies B) \implies \text{rel-cset } A \implies \text{rel-cset } B) \text{ cimage cimage}$
unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cBall-parametric* [*transfer-rule*]:
 $(\text{rel-cset } A \implies (A \implies (=)) \implies (=)) \text{ cBall cBall}$
unfolding *rel-cset-iff rel-fun-def* **by** *blast*

lemma *cBex-parametric* [*transfer-rule*]:
 $(\text{rel-cset } A \implies (A \implies (=)) \implies (=)) \text{ cBex cBex}$
unfolding *rel-cset-iff rel-fun-def* **by** *blast*

lemma *rel-cset-parametric* [*transfer-rule*]:
 $((A \implies B \implies (=)) \implies \text{rel-cset } A \implies \text{rel-cset } B \implies (=))$
rel-cset rel-cset
unfolding *rel-fun-def*
using *rel-set-transfer[unfolded rel-fun-def, rule-format, Transfer.transferred, where*

$A = A$ and $B = B]$
 by *simp*

Rules requiring bi-unique, bi-total or right-total relations

lemma *cin-parametric* [*transfer-rule*]:
 $bi\text{-}unique\ A \implies (A \implies rel\text{-}cset\ A \implies (=))\ cin\ cin$
unfolding *rel-fun-def* *rel-cset-iff* *bi-unique-def* **by** *metis*

lemma *cInt-parametric* [*transfer-rule*]:
 $bi\text{-}unique\ A \implies (rel\text{-}cset\ A \implies rel\text{-}cset\ A \implies rel\text{-}cset\ A)\ cInt\ cInt$
unfolding *rel-fun-def*
using *inter-transfer* [*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*]
by *blast*

lemma *cDiff-parametric* [*transfer-rule*]:
 $bi\text{-}unique\ A \implies (rel\text{-}cset\ A \implies rel\text{-}cset\ A \implies rel\text{-}cset\ A)\ cDiff\ cDiff$
unfolding *rel-fun-def*
using *Diff-transfer* [*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *csubset-parametric* [*transfer-rule*]:
 $bi\text{-}unique\ A \implies (rel\text{-}cset\ A \implies rel\text{-}cset\ A \implies (=))\ csubset\text{-}eq\ csubset\text{-}eq$
unfolding *rel-fun-def*
using *subset-transfer* [*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

end

lifting-update *cset.lifting*
lifting-forget *cset.lifting*

23.5 Registration as BNF

context
includes *cardinal-syntax*
begin

lemma *card-of-countable-sets-range*:
fixes $A :: 'a\ set$
shows $|\{X. X \subseteq A \wedge countable\ X \wedge X \neq \{\}\}| \leq_o |\{f::nat \Rightarrow 'a. range\ f \subseteq A\}|$
proof (*intro card-of-ordLeqI* [*of from-nat-into*])
qed (*use inj-on-from-nat-into in <auto simp: inj-on-def>*)

lemma *card-of-countable-sets-Func*:
 $|\{X. X \subseteq A \wedge countable\ X \wedge X \neq \{\}\}| \leq_o |A| \wedge^c natLeq$
using *card-of-countable-sets-range* *card-of-Func-UNIV* [*THEN ordIso-symmetric*]
unfolding *cexp-def* *Field-natLeq* *Field-card-of*
by (*rule ordLeq-ordIso-trans*)

lemma *ordLeq-countable-subsets*:
 $|A| \leq_o |\{X. X \subseteq A \wedge countable\ X\}|$

```

proof –
  have  $\bigwedge a. a \in A \implies \{a\} \in \{X. X \subseteq A \wedge \text{countable } X\}$ 
    by auto
  with card-of-ordLeqI[of  $\lambda a. \{a\}$ ] show ?thesis
    using inj-singleton by blast
qed

end

lemma finite-countable-subset:
  finite  $\{X. X \subseteq A \wedge \text{countable } X\} \longleftrightarrow \text{finite } A$ 
  using card-of-ordLeq-infinite ordLeq-countable-subsets by force

lemma rcset-to-rcset: countable  $A \implies \text{rcset } (\text{the-inv } \text{rcset } A) = A$ 
  including cset.lifting
  by (meson CollectI f-the-inv-into-f inj-on-inverseI rangeI rcset-induct
    rcset-inverse)

lemma Collect-Int-Times:  $\{(x, y). R \ x \ y\} \cap A \times B = \{(x, y). R \ x \ y \wedge x \in A \wedge y \in B\}$ 
  by auto

lemma rel-cset-aux:
   $(\forall t \in \text{rcset } a. \exists u \in \text{rcset } b. R \ t \ u) \wedge (\forall t \in \text{rcset } b. \exists u \in \text{rcset } a. R \ u \ t) \longleftrightarrow$ 
   $((\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} (\text{cimage fst}))^{-1-1} \text{ OO}$ 
   $\text{BNF-Def.Grp } \{x. \text{rcset } x \subseteq \{(a, b). R \ a \ b\}\} (\text{cimage snd})) \ a \ b \ (\text{is } ?L = ?R)$ 
proof
  assume ?L
  define R' where  $R' = \text{the-inv } \text{rcset } (\text{Collect } (\text{case-prod } R) \cap (\text{rcset } a \times \text{rcset } b))$ 
    (is - = the-inv rcset ?L')
  have L: countable ?L' by auto
  hence *:  $\text{rcset } R' = ?L'$  unfolding R'-def by (intro rcset-to-rcset)
  thus ?R unfolding Grp-def relcompp.simps conversep.simps including cset.lifting
  proof (intro CollectI case-prodI exI[of - a] exI[of - b] exI[of - R'] conjI refl)
    from *  $\langle ?L \rangle$  show  $a = \text{cimage fst } R'$  by transfer (auto simp: image-def Collect-Int-Times)
    from *  $\langle ?L \rangle$  show  $b = \text{cimage snd } R'$  by transfer (auto simp: image-def Collect-Int-Times)
  qed simp-all
next
  assume ?R thus ?L unfolding Grp-def relcompp.simps conversep.simps
    by (simp add: subset-eq Ball-def)(transfer, auto simp add: cimage.rep-eq,metis
snd-conv, metis fst-conv)
qed

context
  includes cardinal-syntax
begin

```

```

bnf 'a cset
  map: cimage
  sets: rcset
  bd: card-suc natLeq
  wits: empty
  rel: rel-cset
proof –
  show cimage id = id by auto
next
  fix f g show cimage (g ∘ f) = cimage g ∘ cimage f by fastforce
next
  fix C f g assume eq:  $\bigwedge a. a \in \text{rcset } C \implies f a = g a$ 
  thus cimage f C = cimage g C including cset.lifting by transfer force
next
  fix f show rcset ∘ cimage f = (') f ∘ rcset including cset.lifting by transfer'
fastforce
next
  show card-order (card-suc natLeq) by (rule card-order-card-suc[OF natLeq-card-order])
next
  show cfinite (card-suc natLeq) using Cfinite-card-suc[OF natLeq-Cfinite
natLeq-card-order]
  by simp
next
  show regularCard (card-suc natLeq) using natLeq-card-order natLeq-Cfinite
  by (rule regularCard-card-suc)
next
  fix C
  have |rcset C| ≤o natLeq including cset.lifting by transfer (unfold count-
able-card-le-natLeq)
  then show |rcset C| <o card-suc natLeq
  using card-suc-greater natLeq-card-order ordLeq-ordLess-trans by blast
next
  fix R S
  show rel-cset R OO rel-cset S ≤ rel-cset (R OO S)
  unfolding rel-cset-alt-def[abs-def] by fast
next
  fix R
  show rel-cset R = (λx y. ∃ z. rcset z ⊆ {(x, y). R x y} ∧
cimage fst z = x ∧ cimage snd z = y)
  unfolding rel-cset-alt-def[abs-def] rel-cset-aux[unfolded OO-Grp-alt] by simp
qed(simp add: bot-cset.rep-eq)

end

end

```

24 Debugging facilities for code generated towards Isabelle/ML

```

theory Debug
imports Main
begin

context
begin

qualified definition trace :: String.literal  $\Rightarrow$  unit where
  [simp]: trace s = ()

qualified definition tracing :: String.literal  $\Rightarrow$  'a  $\Rightarrow$  'a where
  [simp]: tracing s = id

lemma [code]:
  tracing s = (let u = trace s in id)
  by simp

qualified definition flush :: 'a  $\Rightarrow$  unit where
  [simp]: flush x = ()

qualified definition flushing :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b where
  [simp]: flushing x = id

lemma [code, code-unfold]:
  flushing x = (let u = flush x in id)
  by simp

qualified definition timing :: String.literal  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'b where
  [simp]: timing s f x = f x

end

code-printing
  constant Debug.trace  $\mapsto$  (Eval) Output.tracing
| constant Debug.flush  $\mapsto$  (Eval) Output.tracing/ (@{make'-string} -) — note
indirection via antiquotation
| constant Debug.timing  $\mapsto$  (Eval) Timing.timeap'-msg

code-reserved (Eval) Output Timing

end

```

25 Sequence of Properties on Subsequences

```

theory Diagonal-Subsequence

```

imports *Complex-Main*
begin

locale *subseqs* =
fixes $P :: \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{bool}$
assumes *ex-subseq*: $\bigwedge n \ s. \text{strict-mono } (s :: \text{nat} \Rightarrow \text{nat}) \implies \exists r'. \text{strict-mono } r' \wedge P \ n \ (s \circ r')$
begin

definition *reduce* **where** $\text{reduce } s \ n = (\text{SOME } r' :: \text{nat} \Rightarrow \text{nat}. \text{strict-mono } r' \wedge P \ n \ (s \circ r'))$

lemma *subseq-reduce*[*intro, simp*]:
 $\text{strict-mono } s \implies \text{strict-mono } (\text{reduce } s \ n)$
unfolding *reduce-def* **by** (*rule someI2-ex[OF ex-subseq]*) *auto*

lemma *reduce-holds*:
 $\text{strict-mono } s \implies P \ n \ (s \circ \text{reduce } s \ n)$
unfolding *reduce-def* **by** (*rule someI2-ex[OF ex-subseq]*) (*auto simp: o-def*)

primrec *seqseq* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{seqseq } 0 = \text{id}$
 $|\ \text{seqseq } (\text{Suc } n) = \text{seqseq } n \circ \text{reduce } (\text{seqseq } n) \ n$

lemma *subseq-seqseq*[*intro, simp*]: $\text{strict-mono } (\text{seqseq } n)$
proof (*induct n*)
case 0 **thus** ?case **by** (*simp add: strict-mono-def*)
next
case (*Suc n*) **thus** ?case **by** (*subst seqseq.simps*) (*auto intro!: strict-mono-o*)
qed

lemma *seqseq-holds*:
 $P \ n \ (\text{seqseq } (\text{Suc } n))$
proof –
have $P \ n \ (\text{seqseq } n \circ \text{reduce } (\text{seqseq } n) \ n)$
by (*intro reduce-holds subseq-seqseq*)
thus ?thesis **by** *simp*
qed

definition *diagseq* :: $\text{nat} \Rightarrow \text{nat}$ **where** $\text{diagseq } i = \text{seqseq } i \ i$

lemma *diagseq-mono*: $\text{diagseq } n < \text{diagseq } (\text{Suc } n)$
proof –
have $\text{diagseq } n < \text{seqseq } n \ (\text{Suc } n)$
using *subseq-seqseq[of n]* **by** (*simp add: diagseq-def strict-mono-def*)
also have $\dots \leq \text{seqseq } n \ (\text{reduce } (\text{seqseq } n) \ n \ (\text{Suc } n))$
using *strict-mono-less-eq seq-suble* **by** *blast*
also have $\dots = \text{diagseq } (\text{Suc } n)$ **by** (*simp add: diagseq-def*)
finally show ?thesis .

qed

lemma *subseq-diagseq: strict-mono diagseq*
using *diagseq-mono* **by** (*simp add: strict-mono-Suc-iff diagseq-def*)

primrec *fold-reduce* **where**
fold-reduce $n\ 0 = \text{id}$
| *fold-reduce* $n\ (\text{Suc } k) = \text{fold-reduce } n\ k \circ \text{reduce } (\text{seqseq } (n + k))\ (n + k)$

lemma *subseq-fold-reduce[intro, simp]: strict-mono (fold-reduce $n\ k$)*
proof (*induct k*)
case ($\text{Suc } k$) **from** *strict-mono-o[OF this subseq-reduce]* **show** ?case **by** (*simp add: o-def*)
qed (*simp add: strict-mono-def*)

lemma *ex-subseq-reduce-index: seqseq $(n + k) = \text{seqseq } n \circ \text{fold-reduce } n\ k$*
by (*induct k*) *simp-all*

lemma *seqseq-fold-reduce: seqseq $n = \text{fold-reduce } 0\ n$*
by (*induct n*) (*simp-all*)

lemma *diagseq-fold-reduce: diagseq $n = \text{fold-reduce } 0\ n\ n$*
using *seqseq-fold-reduce* **by** (*simp add: diagseq-def*)

lemma *fold-reduce-add: fold-reduce $0\ (m + n) = \text{fold-reduce } 0\ m \circ \text{fold-reduce } m\ n$*
by (*induct n*) *simp-all*

lemma *diagseq-add: diagseq $(k + n) = (\text{seqseq } k \circ (\text{fold-reduce } k\ n))\ (k + n)$*
proof –
have *diagseq $(k + n) = \text{fold-reduce } 0\ (k + n)\ (k + n)$*
by (*simp add: diagseq-fold-reduce*)
also have $\dots = (\text{seqseq } k \circ \text{fold-reduce } k\ n)\ (k + n)$
unfolding *fold-reduce-add seqseq-fold-reduce ..*
finally show ?thesis .
qed

lemma *diagseq-sub:*
assumes $m \leq n$ **shows** *diagseq $n = (\text{seqseq } m \circ (\text{fold-reduce } m\ (n - m)))\ n$*
using *diagseq-add[of $m\ n - m$]* *assms* **by** *simp*

lemma *subseq-diagonal-rest: strict-mono $(\lambda x. \text{fold-reduce } k\ x\ (k + x))$*
unfolding *strict-mono-Suc-iff fold-reduce.simps o-def*
proof
fix n
have *fold-reduce $k\ n\ (k + n) < \text{fold-reduce } k\ n\ (k + \text{Suc } n)$* (**is** ?lhs < -)
by (*auto intro: strict-monoD*)
also have $\dots \leq \text{fold-reduce } k\ n\ (\text{reduce } (\text{seqseq } (k + n))\ (k + n)\ (k + \text{Suc } n))$
by (*auto intro: less-mono-imp-le-mono seq-suble strict-monoD*)

finally show ?lhs <
qed

lemma diagseq-seqseq: $\text{diagseq} \circ ((+) k) = (\text{seqseq } k \circ (\lambda x. \text{fold-reduce } k \ x \ (k + x)))$
by (auto simp: o-def diagseq-add)

lemma diagseq-holds:
assumes subseq-stable: $\bigwedge r \ s \ n. \text{strict-mono } r \implies P \ n \ s \implies P \ n \ (s \circ r)$
shows $P \ k \ (\text{diagseq} \circ ((+) (\text{Suc } k)))$
unfolding diagseq-seqseq by (intro subseq-stable subseq-diagonal-rest seqseq-holds)

end

end

26 Common discrete functions

theory Discrete-Functions
imports Complex-Main
begin

26.1 Discrete logarithm

fun floor-log :: $\text{nat} \Rightarrow \text{nat}$
where [simp del]: $\text{floor-log } n = (\text{if } n < 2 \text{ then } 0 \text{ else } \text{Suc } (\text{floor-log } (n \text{ div } 2)))$

lemma floor-log-induct [consumes 1, case-names one double]:

fixes $n :: \text{nat}$
assumes $n > 0$
assumes one: $P \ 1$
assumes double: $\bigwedge n. n \geq 2 \implies P \ (n \text{ div } 2) \implies P \ n$
shows $P \ n$
using $\langle n > 0 \rangle$ proof (induct n rule: floor-log.induct)
fix n
assume $\neg n < 2 \implies$
 $0 < n \text{ div } 2 \implies P \ (n \text{ div } 2)$
then have *: $n \geq 2 \implies P \ (n \text{ div } 2)$ by simp
assume $n > 0$
show $P \ n$
proof (cases $n = 1$)
case True
with one show ?thesis by simp
next
case False
with $\langle n > 0 \rangle$ have $n \geq 2$ by auto
with * have $P \ (n \text{ div } 2)$.
with $\langle n \geq 2 \rangle$ show ?thesis by (rule double)
qed

qed

lemma *floor-log-zero* [*simp*]: *floor-log* 0 = 0
 by (*simp add: floor-log.simps*)

lemma *floor-log-one* [*simp*]: *floor-log* 1 = 0
 by (*simp add: floor-log.simps*)

lemma *floor-log-Suc-zero* [*simp*]: *floor-log* (*Suc* 0) = 0
 using *floor-log-one* by *simp*

lemma *floor-log-rec*: $n \geq 2 \implies \text{floor-log } n = \text{Suc } (\text{floor-log } (n \text{ div } 2))$
 by (*simp add: floor-log.simps*)

lemma *floor-log-twice* [*simp*]: $n \neq 0 \implies \text{floor-log } (2 * n) = \text{Suc } (\text{floor-log } n)$
 by (*simp add: floor-log-rec*)

lemma *floor-log-half* [*simp*]: *floor-log* (*n div* 2) = *floor-log* *n* - 1
proof (*cases n < 2*)

case *True*

then have $n = 0 \vee n = 1$ by *arith*

then show ?thesis by (*auto simp del: One-nat-def*)

next

case *False*

then show ?thesis by (*simp add: floor-log-rec*)

qed

lemma *floor-log-power* [*simp*]: *floor-log* ($2^{\wedge} n$) = *n*
 by (*induct n*) *simp-all*

lemma *floor-log-mono*: *mono floor-log*

proof

fix *m n* :: *nat*

assume $m \leq n$

then show *floor-log* *m* \leq *floor-log* *n*

proof (*induct m arbitrary: n rule: floor-log.induct*)

case (1 *m*)

then have *mn2*: $m \text{ div } 2 \leq n \text{ div } 2$ by *arith*

show *floor-log* *m* \leq *floor-log* *n*

proof (*cases m \geq 2*)

case *False*

then have $m = 0 \vee m = 1$ by *arith*

then show ?thesis by (*auto simp del: One-nat-def*)

next

case *True* then have $\neg m < 2$ by *simp*

with *mn2* have $n \geq 2$ by *arith*

from *True* have *m2-0*: $m \text{ div } 2 \neq 0$ by *arith*

with *mn2* have *n2-0*: $n \text{ div } 2 \neq 0$ by *arith*

from $\neg m < 2$ 1.hyps *mn2* have *floor-log* ($m \text{ div } 2$) \leq *floor-log* ($n \text{ div } 2$)

```

by blast
  with m2-0 n2-0 have floor-log (2 * (m div 2)) ≤ floor-log (2 * (n div 2))
by simp
  with m2-0 n2-0 ⟨m ≥ 2⟩ ⟨n ≥ 2⟩ show ?thesis by (simp only: floor-log-rec
[of m] floor-log-rec [of n]) simp
qed
qed
qed

```

```

lemma floor-log-exp2-le:
  assumes n > 0
  shows 2 ^ floor-log n ≤ n
  using assms
proof (induct n rule: floor-log-induct)
  case one
  then show ?case by simp
next
  case (double n)
  with floor-log-mono have floor-log n ≥ Suc 0
  by (simp add: floor-log.simps)
  assume 2 ^ floor-log (n div 2) ≤ n div 2
  with ⟨n ≥ 2⟩ have 2 ^ (floor-log n - Suc 0) ≤ n div 2 by simp
  then have 2 ^ (floor-log n - Suc 0) * 2 ^ 1 ≤ n div 2 * 2 by simp
  with ⟨floor-log n ≥ Suc 0⟩ have 2 ^ floor-log n ≤ n div 2 * 2
  unfolding power-add [symmetric] by simp
  also have n div 2 * 2 ≤ n by (cases even n) simp-all
  finally show ?case .
qed

```

```

lemma floor-log-exp2-gt: 2 * 2 ^ floor-log n > n
proof (cases n > 0)
  case True
  thus ?thesis
proof (induct n rule: floor-log-induct)
  case (double n)
  thus ?case
  by (cases even n) (auto elim!: evenE oddE simp: field-simps floor-log.simps)
qed simp-all
qed simp-all

```

```

lemma floor-log-exp2-ge: 2 * 2 ^ floor-log n ≥ n
  using floor-log-exp2-gt[of n] by simp

```

```

lemma floor-log-le-iff: m ≤ n ⟹ floor-log m ≤ floor-log n
  by (rule monoD [OF floor-log-mono])

```

```

lemma floor-log-eqI:
  assumes n > 0 2^k ≤ n n < 2 * 2^k
  shows floor-log n = k

```

proof (*rule antisym*)
from $\langle n > 0 \rangle$ **have** $2^{\text{floor-log } n} \leq n$ **by** (*rule floor-log-exp2-le*)
also have $\dots < 2^{\text{Suc } k}$ **using** *assms* **by** *simp*
finally have $\text{floor-log } n < \text{Suc } k$ **by** (*subst (asm) power-strict-increasing-iff*)
simp-all
thus $\text{floor-log } n \leq k$ **by** *simp*
next
have $2^k \leq n$ **by** *fact*
also have $\dots < 2^{\text{Suc } (\text{floor-log } n)}$ **by** (*simp add: floor-log-exp2-gt*)
finally have $k < \text{Suc } (\text{floor-log } n)$ **by** (*subst (asm) power-strict-increasing-iff*)
simp-all
thus $k \leq \text{floor-log } n$ **by** *simp*
qed

lemma *floor-log-altdef*: $\text{floor-log } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \text{nat } \lfloor \log 2 (\text{real-of-nat } n) \rfloor)$

proof (*cases n = 0*)
case *False*
have $\lfloor \log 2 (\text{real-of-nat } n) \rfloor = \text{int } (\text{floor-log } n)$
proof (*rule floor-unique*)
from *False* **have** $2^{\text{floor-log } n} \leq \text{real } n$
by (*simp add: powr-realpow floor-log-exp2-le*)
hence $\log 2 (2^{\text{floor-log } n}) \leq \log 2 (\text{real } n)$
using *False* **by** (*subst log-le-cancel-iff*) *simp-all*
also have $\log 2 (2^{\text{floor-log } n}) = \text{real } (\text{floor-log } n)$ **by** *simp*
finally show $\text{real-of-int } (\text{int } (\text{floor-log } n)) \leq \log 2 (\text{real } n)$ **by** *simp*
next
have $\text{real } n < \text{real } (2 * 2^{\text{floor-log } n})$
by (*subst of-nat-less-iff*) (*rule floor-log-exp2-gt*)
also have $\dots = 2^{\text{floor-log } n + 1}$
by (*simp add: powr-add powr-realpow*)
finally have $\log 2 (\text{real } n) < \log 2 \dots$
using *False* **by** (*subst log-less-cancel-iff*) *simp-all*
also have $\dots = \text{real } (\text{floor-log } n) + 1$ **by** *simp*
finally show $\log 2 (\text{real } n) < \text{real-of-int } (\text{int } (\text{floor-log } n)) + 1$ **by** *simp*
qed
thus *?thesis* **by** *simp*
qed *simp-all*

26.2 Discrete square root

definition *floor-sqrt* :: $\text{nat} \Rightarrow \text{nat}$
where $\text{floor-sqrt } n = \text{Max } \{m. m^2 \leq n\}$

lemma *floor-sqrt-aux*:

fixes $n :: \text{nat}$
shows *finite* $\{m. m^2 \leq n\}$ **and** $\{m. m^2 \leq n\} \neq \{\}$
proof –
have **: $m \leq n$ **if** $m^2 \leq n$ **for** m

```

    using that by (cases m) (simp-all add: power2-eq-square)
  then have  $\{m. m^2 \leq n\} \subseteq \{m. m \leq n\}$  by auto
  then show finite  $\{m. m^2 \leq n\}$  by (rule finite-subset) rule
  have  $0^2 \leq n$  by simp
  then show *:  $\{m. m^2 \leq n\} \neq \{\}$  by blast
qed

```

```

lemma floor-sqrt-unique:
  assumes  $m^2 \leq n < (Suc\ m)^2$ 
  shows floor-sqrt  $n = m$ 
proof -
  have  $m' \leq m$  if  $m'^2 \leq n$  for  $m'$ 
  proof -
    note that
    also note assms(2)
    finally have  $m' < Suc\ m$  by (rule power-less-imp-less-base) simp-all
    thus  $m' \leq m$  by simp
  qed
  with  $\langle m^2 \leq n \rangle$  floor-sqrt-aux[of  $n$ ] show ?thesis unfolding floor-sqrt-def
  by (intro antisym Max.boundedI Max.coboundedI) simp-all
qed

```

```

lemma floor-sqrt-inverse-power2 [simp]: floor-sqrt  $(n^2) = n$ 
proof -
  have  $\{m. m \leq n\} \neq \{\}$  by auto
  then have  $Max\ \{m. m \leq n\} \leq n$  by auto
  then show ?thesis
    by (auto simp add: floor-sqrt-def power2-nat-le-eq-le intro: antisym)
qed

```

```

lemma floor-sqrt-zero [simp]: floor-sqrt 0 = 0
  using floor-sqrt-inverse-power2 [of 0] by simp

```

```

lemma floor-sqrt-one [simp]: floor-sqrt 1 = 1
  using floor-sqrt-inverse-power2 [of 1] by simp

```

```

lemma floor-sqrt-Suc-0 [simp]:
   $\langle floor-sqrt\ (Suc\ 0) = 1 \rangle$ 
  using floor-sqrt-inverse-power2 [of 1] by simp

```

```

lemma mono-floor-sqrt: mono floor-sqrt
proof
  fix  $m\ n :: nat$ 
  have *:  $0 * 0 \leq m$  by simp
  assume  $m \leq n$ 
  then show floor-sqrt  $m \leq floor-sqrt\ n$ 
    by (auto intro!: Max-mono  $\langle 0 * 0 \leq m \rangle$  finite-less-ub simp add: power2-eq-square
    floor-sqrt-def)
qed

```

lemma *mono-floor-sqrt'*: $m \leq n \implies \text{floor-sqrt } m \leq \text{floor-sqrt } n$
using *mono-floor-sqrt unfolding mono-def by auto*

lemma *floor-sqrt-greater-zero-iff* [*simp*]: $\text{floor-sqrt } n > 0 \iff n > 0$

proof –

have *: $0 < \text{Max } \{m. m^2 \leq n\} \iff (\exists a \in \{m. m^2 \leq n\}. 0 < a)$
by (*rule Max-gr-iff*) (*fact floor-sqrt-aux*) +

show ?thesis

proof

assume $0 < \text{floor-sqrt } n$

then have $0 < \text{Max } \{m. m^2 \leq n\}$ **by** (*simp add: floor-sqrt-def*)

with * **show** $0 < n$ **by** (*auto dest: power2-nat-le-imp-le*)

next

assume $0 < n$

then have $1^2 \leq n \wedge 0 < (1::\text{nat})$ **by** *simp*

then have $\exists q. q^2 \leq n \wedge 0 < q$ **..**

with * **have** $0 < \text{Max } \{m. m^2 \leq n\}$ **by** *blast*

then show $0 < \text{floor-sqrt } n$ **by** (*simp add: floor-sqrt-def*)

qed

qed

lemma *floor-sqrt-power2-le* [*simp*]: $(\text{floor-sqrt } n)^2 \leq n$

proof (*cases n > 0*)

case *False* **then show** ?thesis **by** *simp*

next

case *True* **then have** $\text{floor-sqrt } n > 0$ **by** *simp*

then have *mono* (*times* (*Max* $\{m. m^2 \leq n\}$)) **by** (*auto intro: mono-times-nat simp add: floor-sqrt-def*)

then have *: $\text{Max } \{m. m^2 \leq n\} * \text{Max } \{m. m^2 \leq n\} = \text{Max } (\text{times } (\text{Max } \{m. m^2 \leq n\}) ' \{m. m^2 \leq n\})$

using *floor-sqrt-aux [of n]* **by** (*rule mono-Max-commute*)

have $\bigwedge a. a * a \leq n \implies \text{Max } \{m. m * m \leq n\} * a \leq n$

proof –

fix *q*

assume $q * q \leq n$

show $\text{Max } \{m. m * m \leq n\} * q \leq n$

proof (*cases q > 0*)

case *False* **then show** ?thesis **by** *simp*

next

case *True* **then have** *mono* (*times q*) **by** (*rule mono-times-nat*)

then have $q * \text{Max } \{m. m * m \leq n\} = \text{Max } (\text{times } q ' \{m. m * m \leq n\})$

using *floor-sqrt-aux [of n]* **by** (*auto simp add: power2-eq-square intro: mono-Max-commute*)

then have $\text{Max } \{m. m * m \leq n\} * q = \text{Max } (\text{times } q ' \{m. m * m \leq n\})$

by (*simp add: ac-simps*)

moreover have *finite* ($(*) q ' \{m. m * m \leq n\}$)

by (*metis* (*mono-tags*) *finite-imageI* *finite-less-ub le-square*)

moreover have $\exists x. x * x \leq n$

```

    by (metis ‹ $q * q \leq n$ ›)
  ultimately show ?thesis
  by simp (metis ‹ $q * q \leq n$ › le-cases mult-le-mono1 mult-le-mono2 order-trans)
qed
qed
then have Max ((*) (Max { $m. m * m \leq n$ }) ‹{ $m. m * m \leq n$ }›)  $\leq n$ 
  apply (subst Max-le-iff)
  apply (metis (mono-tags) finite-imageI finite-less-ub le-square)
  apply auto
  apply (metis le0 mult-0-right)
  done
with * show ?thesis by (simp add: floor-sqrt-def power2-eq-square)
qed

```

```

lemma floor-sqrt-le: floor-sqrt  $n \leq n$ 
  using floor-sqrt-aux [of  $n$ ] by (auto simp add: floor-sqrt-def intro: power2-nat-le-imp-le)

```

Additional facts about the discrete square root, thanks to Julian Bien-darra, Manuel Eberl

```

lemma Suc-floor-sqrt-power2-gt:  $n < (Suc (floor-sqrt n))^2$ 
  using Max-ge[OF floor-sqrt-aux(1), of floor-sqrt  $n + 1$   $n$ ]
  by (cases  $n < (Suc (floor-sqrt n))^2$ ) (simp-all add: floor-sqrt-def)

```

```

lemma le-floor-sqrt-iff:  $x \leq floor-sqrt y \iff x^2 \leq y$ 
proof -
  have  $x \leq floor-sqrt y \iff (\exists z. z^2 \leq y \wedge x \leq z)$ 
    using Max-ge-iff[OF floor-sqrt-aux, of  $x$   $y$ ] by (simp add: floor-sqrt-def)
  also have  $\dots \iff x^2 \leq y$ 
  proof safe
    fix  $z$  assume  $x \leq z$ 
    thus  $x^2 \leq y$  by (intro le-trans[of  $x^2 z^2 y$ ]) (simp-all add: power2-nat-le-eq-le)
  qed auto
  finally show ?thesis .
qed

```

```

lemma le-floor-sqrtI:  $x^2 \leq y \implies x \leq floor-sqrt y$ 
  by (simp add: le-floor-sqrt-iff)

```

```

lemma floor-sqrt-le-iff:
  ‹ $floor-sqrt y \leq x \iff (\forall z. z^2 \leq y \implies z \leq x)$ ›
  using Max.bounded-iff [OF floor-sqrt-aux]
  by (simp add: floor-sqrt-def)

```

```

lemma floor-sqrt-leI:
   $(\bigwedge z. z^2 \leq y \implies z \leq x) \implies floor-sqrt y \leq x$ 
  by (simp add: floor-sqrt-le-iff)

```

```

lemma floor-sqrt-less-eq-half:
  ‹ $floor-sqrt n \leq Suc n \div 2$ ›

```

```

proof (rule floor-sqrt-leI)
  fix m
  assume  $\langle m^2 \leq n \rangle$ 
  have  $\langle m < \text{Suc } (\text{Suc } n \text{ div } 2) \rangle$ 
  proof (rule ccontr, unfold not-less)
    assume  $\langle \text{Suc } (\text{Suc } n \text{ div } 2) \leq m \rangle$ 
    then have  $\langle (\text{Suc } (\text{Suc } n \text{ div } 2))^2 \leq m^2 \rangle$ 
    by simp
    then have  $\langle (\text{Suc } (\text{Suc } n \text{ div } 2))^2 \leq n \rangle$ 
    using  $\langle m^2 \leq n \rangle$  by (rule order-trans)
    then show False
    by (simp only: Suc-eq-plus1 power2-sum algebra-simps) auto
  qed
  then show  $\langle m \leq \text{Suc } n \text{ div } 2 \rangle$ 
  by simp
qed

```

lemma floor-sqrt-Suc:

$\text{floor-sqrt } (\text{Suc } n) = (\text{if } \exists m. \text{Suc } n = m^2 \text{ then } \text{Suc } (\text{floor-sqrt } n) \text{ else } \text{floor-sqrt } n)$

proof cases

```

  assume  $\exists m. \text{Suc } n = m^2$ 
  then obtain m where m-def:  $\text{Suc } n = m^2$  by blast
  then have lhs:  $\text{floor-sqrt } (\text{Suc } n) = m$  by simp
  from m-def floor-sqrt-power2-le[of n]
    have  $(\text{floor-sqrt } n)^2 < m^2$  by linarith
  with power2-less-imp-less have lt-m:  $\text{floor-sqrt } n < m$  by blast
  from m-def Suc-floor-sqrt-power2-gt[of n]
    have  $m^2 \leq (\text{Suc } (\text{floor-sqrt } n))^2$ 
    by linarith
  with power2-nat-le-eq-le have  $m \leq \text{Suc } (\text{floor-sqrt } n)$  by blast
  with lt-m have  $m = \text{Suc } (\text{floor-sqrt } n)$  by simp
  with lhs m-def show ?thesis by fastforce

```

next

```

  assume asm:  $\neg (\exists m. \text{Suc } n = m^2)$ 
  hence  $\text{Suc } n \neq (\text{floor-sqrt } (\text{Suc } n))^2$  by simp
  with floor-sqrt-power2-le[of Suc n]
    have  $\text{floor-sqrt } (\text{Suc } n) \leq \text{floor-sqrt } n$  by (intro le-floor-sqrtI) linarith
  moreover have  $\text{floor-sqrt } (\text{Suc } n) \geq \text{floor-sqrt } n$ 
    by (intro monoD[OF mono-floor-sqrt]) simp-all
  ultimately show ?thesis using asm by simp

```

qed

Computation by divide and conquer

definition floor-sqrt-between :: $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \rangle$

where floor-sqrt-between-eq:

$\langle \text{floor-sqrt-between } m \ q \ n =$
 $(\text{if } \text{floor-sqrt } n \in \{m..<m + q\} \text{ then } \text{floor-sqrt } n \text{ else } 0) \rangle$

— The 0 is not for relevant regular computation and can be chosen arbitrarily.

lemma *floor-sqrt-between-out-of-bounds:*

$\langle \text{floor-sqrt-between } m \ 0 \ n = 0 \rangle$

by (*simp add: floor-sqrt-between-eq*)

lemma *floor-sqrt-between-singleton:*

$\langle \text{floor-sqrt-between } m \ (\text{Suc } 0) \ n =$

$(\text{if } m^2 \leq n \wedge n < (\text{Suc } m)^2 \text{ then } m \text{ else } 0) \rangle$

by (*auto simp add: floor-sqrt-between-eq Suc-floor-sqrt-power2-gt floor-sqrt-unique*)

lemma *floor-sqrt-between-rec:*

$\langle \text{floor-sqrt-between } m \ q \ n = ($

let

$r = q \text{ div } 2;$

$p = m + r;$

$s = p^2$

in

if $s = n$

then p

else if $s < n$

then $\text{floor-sqrt-between } (m + r) \ (q - r) \ n$

else $\text{floor-sqrt-between } m \ r \ n$

\rangle **if** $\langle q > 0 \rangle$

using *that le-floor-sqrt-iff [of $\langle m + q \text{ div } 2 \rangle \ n]$*

by (*auto simp add: floor-sqrt-between-eq Let-def not-less*)

lemma *floor-sqrt-between-code [code]:*

$\langle \text{floor-sqrt-between } m \ q \ n = ($

if $q = 0$ *then* 0

else if $q = 1$

then if $m^2 \leq n \wedge n < (\text{Suc } m)^2$

then m

else 0

else

let

$r = q \text{ div } 2;$

$p = m + r;$

$s = p^2$

in

if $s = n$

then p

else if $s < n$

then $\text{floor-sqrt-between } (m + r) \ (q - r) \ n$

else $\text{floor-sqrt-between } m \ r \ n$

\rangle

proof –

consider $\langle q = 0 \rangle \mid \langle q = 1 \rangle \mid \langle q \geq 2 \rangle$

by (*cases $\langle q \geq 2 \rangle$; cases q simp-all*)

then show *?thesis*

```

    by cases
      (simp-all add: floor-sqrt-between-out-of-bounds floor-sqrt-between-singleton
        floor-sqrt-between-rec)
  qed

lemma [code]:
  ⌊floor-sqrt n = floor-sqrt-between 0 (Suc (Suc n div 2)) n⌋
  using floor-sqrt-less-eq-half [of n] by (simp add: floor-sqrt-between-eq)

end

```

27 Pi and Function Sets

```

theory FuncSet
  imports Main
  abbrevs PiE = PiE
  and PIE = ΠE
begin

definition Pi :: 'a set ⇒ ('a ⇒ 'b set) ⇒ ('a ⇒ 'b) set
  where Pi A B = {f. ∀ x. x ∈ A ⟶ f x ∈ B x}

definition extensional :: 'a set ⇒ ('a ⇒ 'b) set
  where extensional A = {f. ∀ x. x ∉ A ⟶ f x = undefined}

definition restrict :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'a ⇒ 'b
  where restrict f A = (λx. if x ∈ A then f x else undefined)

abbreviation funcset :: 'a set ⇒ 'b set ⇒ ('a ⇒ 'b) set
  where funcset A B ≡ Pi A (λ-. B)

open-bundle funcset-syntax
begin
  notation funcset (infixr ⌊⟶⌋ 60)
end

```

```

syntax
  -Pi :: pttrn ⇒ 'a set ⇒ 'b set ⇒ ('a ⇒ 'b) set
    (⌊⟨⟨indent=3 notation=⟨binder Π∈⟩Π -∈-./ -⟩ 10⟩
  -lam :: pttrn ⇒ 'a set ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
    (⌊⟨⟨indent=3 notation=⟨binder λ∈⟩λ-∈-./ -⟩ [0, 0, 3] 3⟩
syntax-consts
  -Pi ⇒ Pi and
  -lam ⇒ restrict
translations
  Π x∈A. B ⇒ CONST Pi A (λx. B)
  λx∈A. f ⇒ CONST restrict (λx. f) A

```

```

definition compose :: 'a set ⇒ ('b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'c)

```

where $\text{compose } A \ g \ f = (\lambda x \in A. \ g \ (f \ x))$

27.1 Basic Properties of Pi

lemma $Pi-I[\text{intro!}]$: $(\bigwedge x. x \in A \implies f \ x \in B \ x) \implies f \in Pi \ A \ B$
by $(\text{simp add: } Pi\text{-def})$

lemma $Pi-I'[\text{simp}]$: $(\bigwedge x. x \in A \longrightarrow f \ x \in B \ x) \implies f \in Pi \ A \ B$
by $(\text{simp add: } Pi\text{-def})$

lemma funcsetI : $(\bigwedge x. x \in A \implies f \ x \in B) \implies f \in A \rightarrow B$
by $(\text{simp add: } Pi\text{-def})$

lemma $Pi\text{-mem}$: $f \in Pi \ A \ B \implies x \in A \implies f \ x \in B \ x$
by $(\text{simp add: } Pi\text{-def})$

lemma $Pi\text{-iff}$: $f \in Pi \ I \ X \longleftrightarrow (\forall i \in I. f \ i \in X \ i)$
unfolding $Pi\text{-def}$ **by** auto

lemma $PiE[\text{elim}]$: $f \in Pi \ A \ B \implies (f \ x \in B \ x \implies Q) \implies (x \notin A \implies Q) \implies Q$
by $(\text{auto simp: } Pi\text{-def})$

lemma $Pi\text{-cong}$: $(\bigwedge w. w \in A \implies f \ w = g \ w) \implies f \in Pi \ A \ B \longleftrightarrow g \in Pi \ A \ B$
by $(\text{auto simp: } Pi\text{-def})$

lemma $\text{funcset-id}[\text{simp}]$: $(\lambda x. x) \in A \rightarrow A$
by auto

lemma funcset-mem : $f \in A \rightarrow B \implies x \in A \implies f \ x \in B$
by $(\text{simp add: } Pi\text{-def})$

lemma funcset-image : $f \in A \rightarrow B \implies f \ ' \ A \subseteq B$
by auto

lemma $\text{image-subset-iff-funcset}$: $F \ ' \ A \subseteq B \longleftrightarrow F \in A \rightarrow B$
by auto

lemma $\text{funcset-to-empty-iff}$: $A \rightarrow \{\} = (\text{if } A = \{\} \text{ then } UNIV \text{ else } \{\})$
by auto

lemma $Pi\text{-eq-empty}[\text{simp}]$: $(\Pi x \in A. B \ x) = \{\} \longleftrightarrow (\exists x \in A. B \ x = \{\})$

proof –

have $\exists x \in A. B \ x = \{\}$ **if** $\bigwedge f. \exists y. y \in A \wedge f \ y \notin B \ y$
using that $[\text{of } \lambda u. \text{SOME } y. y \in B \ u] \text{ some-in-eq}$ **by** blast
then show $?thesis$
by force

qed

lemma $Pi\text{-empty}[\text{simp}]$: $Pi \ \{\} \ B = UNIV$

by (simp add: Pi-def)

lemma *Pi-Int*: $Pi\ I\ E \cap Pi\ I\ F = (\Pi\ i \in I. E\ i \cap F\ i)$
by *auto*

lemma *Pi-UN*:

fixes $A :: nat \Rightarrow 'i \Rightarrow 'a\ set$

assumes *finite I*

and *mono*: $\bigwedge i\ n\ m. i \in I \implies n \leq m \implies A\ n\ i \subseteq A\ m\ i$

shows $(\bigcup n. Pi\ I\ (A\ n)) = (\Pi\ i \in I. \bigcup n. A\ n\ i)$

proof (intro *set-eqI iffI*)

fix *f*

assume $f \in (\Pi\ i \in I. \bigcup n. A\ n\ i)$

then have $\forall i \in I. \exists n. f\ i \in A\ n\ i$

by *auto*

from *bchoice*[*OF this*] obtain *n* where $n: f\ i \in A\ (n\ i)\ i\ \text{if } i \in I\ \text{for } i$

by *auto*

obtain *k* where $k: n\ i \leq k\ \text{if } i \in I\ \text{for } i$

using $\langle finite\ I \rangle$ *finite-nat-set-iff-bounded-le*[*of n'I*] by *auto*

have $f \in Pi\ I\ (A\ k)$

proof (intro *Pi-I*)

fix *i*

assume $i \in I$

from *mono*[*OF this*, *of n i k*] *k*[*OF this*] *n*[*OF this*]

show $f\ i \in A\ k\ i$ by *auto*

qed

then show $f \in (\bigcup n. Pi\ I\ (A\ n))$

by *auto*

qed *auto*

lemma *Pi-UNIV* [*simp*]: $A \rightarrow UNIV = UNIV$

by (simp add: Pi-def)

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x. x \in A \implies B\ x \subseteq C\ x) \implies Pi\ A\ B \subseteq Pi\ A\ C$

by *auto*

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \implies Pi\ A\ B \subseteq Pi\ A'\ B$

by *auto*

lemma *prod-final*:

assumes $1: fst \circ f \in Pi\ A\ B$

and $2: snd \circ f \in Pi\ A\ C$

shows $f \in (\Pi\ z \in A. B\ z \times C\ z)$

proof (rule *Pi-I*)

fix *z*

assume $z: z \in A$

have $f\ z = (fst\ (f\ z),\ snd\ (f\ z))$

by *simp*
 also have $\dots \in B\ z \times C\ z$
 by (*metis SigmaI PiE o-apply 1 2 z*)
 finally show $f\ z \in B\ z \times C\ z$.
 qed

lemma *Pi-split-domain*[*simp*]: $x \in \text{Pi } (I \cup J)\ X \longleftrightarrow x \in \text{Pi } I\ X \wedge x \in \text{Pi } J\ X$
 by (*auto simp: Pi-def*)

lemma *Pi-split-insert-domain*[*simp*]: $x \in \text{Pi } (\text{insert } i\ I)\ X \longleftrightarrow x \in \text{Pi } I\ X \wedge x\ i \in X\ i$
 by (*auto simp: Pi-def*)

lemma *Pi-cancel-fupd-range*[*simp*]: $i \notin I \implies x \in \text{Pi } I\ (B(i := b)) \longleftrightarrow x \in \text{Pi } I\ B$
 by (*auto simp: Pi-def*)

lemma *Pi-cancel-fupd*[*simp*]: $i \notin I \implies x(i := a) \in \text{Pi } I\ B \longleftrightarrow x \in \text{Pi } I\ B$
 by (*auto simp: Pi-def*)

lemma *Pi-fupd-iff*: $i \in I \implies f \in \text{Pi } I\ (B(i := A)) \longleftrightarrow f \in \text{Pi } (I - \{i\})\ B \wedge f\ i \in A$
 using *mk-disjoint-insert* by *fastforce*

lemma *fst-Pi*: $\text{fst} \in A \times B \rightarrow A$ and *snd-Pi*: $\text{snd} \in A \times B \rightarrow B$
 by *auto*

27.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \rightarrow B \implies g \in B \rightarrow C \implies \text{compose } A\ g\ f \in A \rightarrow C$
 by (*simp add: Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
 assumes $f \in A \rightarrow B$
 shows $\text{compose } A\ h\ (\text{compose } A\ g\ f) = \text{compose } A\ (\text{compose } B\ h\ g)\ f$
 using *assms* by (*simp add: fun-eq-iff Pi-def compose-def restrict-def*)

lemma *compose-eq*: $x \in A \implies \text{compose } A\ g\ f\ x = g\ (f\ x)$
 by (*simp add: compose-def restrict-def*)

lemma *surj-compose*: $f\ ` A = B \implies g\ ` B = C \implies \text{compose } A\ g\ f\ ` A = C$
 by (*auto simp add: image-def compose-eq*)

27.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \implies (\bigwedge i. i \in J \implies f\ i = g\ i) \implies \text{restrict } f\ I = \text{restrict } g\ J$
 by (*auto simp: restrict-def fun-eq-iff simp-implies-def*)

lemma *restrictI[intro!]*: $(\bigwedge x. x \in A \implies f x \in B x) \implies (\lambda x \in A. f x) \in \text{Pi } A \ B$
by (*simp add: Pi-def restrict-def*)

lemma *restrict-apply[simp]*: $(\lambda y \in A. f y) x = (\text{if } x \in A \text{ then } f x \text{ else undefined})$
by (*simp add: restrict-def*)

lemma *restrict-apply'*: $x \in A \implies (\lambda y \in A. f y) x = f x$
by *simp*

lemma *restrict-ext*: $(\bigwedge x. x \in A \implies f x = g x) \implies (\lambda x \in A. f x) = (\lambda x \in A. g x)$
by (*simp add: fun-eq-iff Pi-def restrict-def*)

lemma *restrict-UNIV*: $\text{restrict } f \text{ UNIV} = f$
by (*simp add: restrict-def*)

lemma *inj-on-restrict-eq [simp]*: $\text{inj-on } (\text{restrict } f A) A \longleftrightarrow \text{inj-on } f A$
by (*simp add: inj-on-def restrict-def*)

lemma *inj-on-restrict-iff*: $A \subseteq B \implies \text{inj-on } (\text{restrict } f B) A \longleftrightarrow \text{inj-on } f A$
by (*metis inj-on-cong restrict-def subset-iff*)

lemma *Id-compose*: $f \in A \rightarrow B \implies f \in \text{extensional } A \implies \text{compose } A (\lambda y \in B. y)$
 $f = f$
by (*auto simp add: fun-eq-iff compose-def extensional-def Pi-def*)

lemma *compose-Id*: $g \in A \rightarrow B \implies g \in \text{extensional } A \implies \text{compose } A g (\lambda x \in A. x) = g$
by (*auto simp add: fun-eq-iff compose-def extensional-def Pi-def*)

lemma *image-restrict-eq [simp]*: $(\text{restrict } f A) \text{ ` } A = f \text{ ` } A$
by (*auto simp add: restrict-def*)

lemma *restrict-restrict[simp]*: $\text{restrict } (\text{restrict } f A) B = \text{restrict } f (A \cap B)$
unfolding *restrict-def* **by** (*simp add: fun-eq-iff*)

lemma *restrict-fupd[simp]*: $i \notin I \implies \text{restrict } (f (i := x)) I = \text{restrict } f I$
by (*auto simp: restrict-def*)

lemma *restrict-upd[simp]*: $i \notin I \implies (\text{restrict } f I)(i := y) = \text{restrict } (f(i := y))$
 $(\text{insert } i I)$
by (*auto simp: fun-eq-iff*)

lemma *restrict-Pi-cancel*: $\text{restrict } x I \in \text{Pi } I A \longleftrightarrow x \in \text{Pi } I A$
by (*auto simp: restrict-def Pi-def*)

lemma *sum-restrict' [simp]*: $\text{sum}' (\lambda i \in I. g i) I = \text{sum}' (\lambda i. g i) I$
by (*simp add: sum.G-def conj-commute cong: conj-cong*)

lemma *prod-restrict' [simp]*: $\text{prod}' (\lambda i \in I. g i) I = \text{prod}' (\lambda i. g i) I$

by (simp add: prod.G-def conj-commute cong: conj-cong)

27.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

lemma *bij-betwI*:

assumes $f \in A \rightarrow B$
 and $g \in B \rightarrow A$
 and $g \cdot f: \bigwedge x. x \in A \implies g (f x) = x$
 and $f \cdot g: \bigwedge y. y \in B \implies f (g y) = y$
 shows *bij-betw* $f A B$
 unfolding *bij-betw-def*

proof

show *inj-on* $f A$
 by (metis *g-f inj-on-def*)
 have $f \cdot A \subseteq B$
 using $\langle f \in A \rightarrow B \rangle$ by auto
 moreover
 have $B \subseteq f \cdot A$
 by auto (metis *Pi-mem* $\langle g \in B \rightarrow A \rangle$ *f-g image-iff*)
 ultimately show $f \cdot A = B$
 by blast

qed

lemma *bij-betw-imp-funcset*: *bij-betw* $f A B \implies f \in A \rightarrow B$

by (auto simp add: *bij-betw-def*)

lemma *inj-on-compose*: *bij-betw* $f A B \implies \text{inj-on } g B \implies \text{inj-on } (\text{compose } A g f)$
 A

by (auto simp add: *bij-betw-def inj-on-def compose-eq*)

lemma *bij-betw-compose*: *bij-betw* $f A B \implies \text{bij-betw } g B C \implies \text{bij-betw } (\text{compose } A g f) A C$

by (simp add: *bij-betw-def inj-on-compose surj-compose*)

lemma *bij-betw-restrict-eq* [simp]: *bij-betw* $(\text{restrict } f A) A B = \text{bij-betw } f A B$

by (simp add: *bij-betw-def*)

27.5 Extensionality

lemma *extensional-empty*[simp]: *extensional* $\{\}$ = $\{\lambda x. \text{undefined}\}$

unfolding *extensional-def* by auto

lemma *extensional-arb*: $f \in \text{extensional } A \implies x \notin A \implies f x = \text{undefined}$

by (simp add: *extensional-def*)

lemma *restrict-extensional* [simp]: *restrict* $f A \in \text{extensional } A$

by (simp add: *restrict-def extensional-def*)

lemma *compose-extensional* [*simp*]: *compose A f g* \in *extensional A*
by (*simp add: compose-def*)

lemma *extensionalityI*:
assumes *f* \in *extensional A*
and *g* \in *extensional A*
and $\bigwedge x. x \in A \implies f\ x = g\ x$
shows *f* = *g*
using *assms* **by** (*force simp add: fun-eq-iff extensional-def*)

lemma *extensional-restrict*: *f* \in *extensional A* \implies *restrict f A* = *f*
by (*rule extensionalityI[OF restrict-extensional]*) *auto*

lemma *extensional-subset*: *f* \in *extensional A* $\implies A \subseteq B \implies f \in$ *extensional B*
unfolding *extensional-def* **by** *auto*

lemma *inv-into-funcset*: *f* ‘ *A* = *B* $\implies (\lambda x \in B. \text{inv-into } A\ f\ x) \in B \rightarrow A$
by (*unfold inv-into-def*) (*fast intro: someI2*)

lemma *compose-inv-into-id*: *bij-betw f A B* \implies *compose A* ($\lambda y \in B. \text{inv-into } A\ f\ y$)
f = ($\lambda x \in A. x$)
by (*smt (verit, best) bij-betwE bij-betw-inv-into-left compose-def restrict-apply' restrict-ext*)

lemma *compose-id-inv-into*: *f* ‘ *A* = *B* \implies *compose B f* ($\lambda y \in B. \text{inv-into } A\ f\ y$)
= ($\lambda x \in B. x$)
by (*smt (verit, best) compose-def f-inv-into-f restrict-apply' restrict-ext*)

lemma *extensional-insert*[*intro, simp*]:
assumes *a* \in *extensional (insert i I)*
shows *a(i := b)* \in *extensional (insert i I)*
using *assms* **unfolding** *extensional-def* **by** *auto*

lemma *extensional-Int*[*simp*]: *extensional I* \cap *extensional I'* = *extensional (I* \cap *I')*
unfolding *extensional-def* **by** *auto*

lemma *extensional-UNIV*[*simp*]: *extensional UNIV* = *UNIV*
by (*auto simp: extensional-def*)

lemma *restrict-extensional-sub*[*intro*]: *A* $\subseteq B \implies$ *restrict f A* \in *extensional B*
unfolding *restrict-def extensional-def* **by** *auto*

lemma *extensional-insert-undefined*[*intro, simp*]:
a \in *extensional (insert i I)* $\implies a(i := \text{undefined}) \in$ *extensional I*
unfolding *extensional-def* **by** *auto*

lemma *extensional-insert-cancel*[*intro, simp*]:

$a \in \text{extensional } I \implies a \in \text{extensional } (\text{insert } i \ I)$
unfolding *extensional-def* **by** *auto*

27.6 Cardinality

lemma *card-inj*: $f \in A \rightarrow B \implies \text{inj-on } f \ A \implies \text{finite } B \implies \text{card } A \leq \text{card } B$
by (*rule card-inj-on-le*) *auto*

lemma *card-bij*:
assumes $f \in A \rightarrow B$ *inj-on* $f \ A$
and $g \in B \rightarrow A$ *inj-on* $g \ B$
and *finite* A *finite* B
shows $\text{card } A = \text{card } B$
using *assms* **by** (*blast intro: card-inj order-antisym*)

27.7 Extensional Function Spaces

definition $PiE :: 'a \text{ set} \Rightarrow ('a \Rightarrow 'b \text{ set}) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
where $PiE \ S \ T = Pi \ S \ T \cap \text{extensional } S$

abbreviation $Pi_E \ A \ B \equiv PiE \ A \ B$

syntax
 $-PiE :: ptrn \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a \Rightarrow 'b) \text{ set}$
 ($\langle \langle \text{indent}=3 \ \text{notation}=\langle \text{binder } \Pi_E \in \rangle \Pi_E \ -\in \cdot / \ - \rangle \ 10 \rangle$)

syntax-consts
 $-PiE \equiv Pi_E$

translations
 $\Pi_E \ x \in A. \ B \equiv \text{CONST } Pi_E \ A \ (\lambda x. \ B)$

abbreviation *extensional-funcset* :: $'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a \Rightarrow 'b) \text{ set}$ (**infixr** \rightarrow_E)
 60)
where $A \rightarrow_E B \equiv (\Pi_E \ i \in A. \ B)$

lemma *extensional-funcset-def*: $\text{extensional-funcset } S \ T = (S \rightarrow T) \cap \text{extensional } S$
by (*simp add: PiE-def*)

lemma *PiE-empty-domain*[*simp*]: $Pi_E \ \{\} \ T = \{\lambda x. \ \text{undefined}\}$
unfolding *PiE-def* **by** *simp*

lemma *PiE-UNIV-domain*: $Pi_E \ \text{UNIV} \ T = Pi \ \text{UNIV} \ T$
unfolding *PiE-def* **by** *simp*

lemma *PiE-empty-range*[*simp*]: $i \in I \implies F \ i = \{\} \implies (\Pi_E \ i \in I. \ F \ i) = \{\}$
unfolding *PiE-def* **by** *auto*

lemma *PiE-eq-empty-iff*: $Pi_E \ I \ F = \{\} \longleftrightarrow (\exists i \in I. \ F \ i = \{\})$
proof
assume $Pi_E \ I \ F = \{\}$

```

show  $\exists i \in I. F\ i = \{\}$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then have  $\forall i. \exists y. (i \in I \longrightarrow y \in F\ i) \wedge (i \notin I \longrightarrow y = \text{undefined})$ 
    by auto
  from choice[OF this]
  obtain f where  $\forall x. (x \in I \longrightarrow f\ x \in F\ x) \wedge (x \notin I \longrightarrow f\ x = \text{undefined}) ..$ 
  then have  $f \in Pi_E\ I\ F$ 
    by (auto simp: extensional-def PiE-def)
  with  $\langle Pi_E\ I\ F = \{\} \rangle$  show False
    by auto
qed
qed (auto simp: PiE-def)

```

```

lemma PiE-arb:  $f \in Pi_E\ S\ T \Longrightarrow x \notin S \Longrightarrow f\ x = \text{undefined}$ 
  unfolding PiE-def by auto (auto dest!: extensional-arb)

```

```

lemma PiE-mem:  $f \in Pi_E\ S\ T \Longrightarrow x \in S \Longrightarrow f\ x \in T\ x$ 
  unfolding PiE-def by auto

```

```

lemma PiE-fun-upd:  $y \in T\ x \Longrightarrow f \in Pi_E\ S\ T \Longrightarrow f(x := y) \in Pi_E\ (\text{insert } x\ S)$ 
  unfolding PiE-def extensional-def by auto

```

```

lemma fun-upd-in-PiE:  $x \notin S \Longrightarrow f \in Pi_E\ (\text{insert } x\ S)\ T \Longrightarrow f(x := \text{undefined})$ 
   $\in Pi_E\ S\ T$ 
  unfolding PiE-def extensional-def by auto

```

```

lemma PiE-insert-eq:  $Pi_E\ (\text{insert } x\ S)\ T = (\lambda(y, g). g(x := y))\ ` (T\ x \times Pi_E\ S\ T)$ 

```

```

proof –
  have  $f \in (\lambda(y, g). g(x := y))\ ` (T\ x \times Pi_E\ S\ T)$  if  $f \in Pi_E\ (\text{insert } x\ S)\ T$   $x \notin S$ 
    for f
    using that
    by (auto intro!: image-eqI[where x=(f\ x, f(x := undefined))]) intro: fun-upd-in-PiE PiE-mem
  moreover
    have  $f \in (\lambda(y, g). g(x := y))\ ` (T\ x \times Pi_E\ S\ T)$  if  $f \in Pi_E\ (\text{insert } x\ S)\ T$   $x \in S$ 
      for f
      using that
      by (auto intro!: image-eqI[where x=(f\ x, f)]) intro: fun-upd-in-PiE PiE-mem simp: insert-absorb
    ultimately show ?thesis
      by (auto intro: PiE-fun-upd)
qed

```

```

lemma PiE-Int:  $Pi_E\ I\ A \cap Pi_E\ I\ B = Pi_E\ I\ (\lambda x. A\ x \cap B\ x)$ 
  by (auto simp: PiE-def)

```

lemma *PiE-cong*: $(\bigwedge i. i \in I \implies A\ i = B\ i) \implies \text{Pi}_E\ I\ A = \text{Pi}_E\ I\ B$
unfolding *PiE-def* **by** (*auto simp: Pi-cong*)

lemma *PiE-E* [*elim*]:
assumes $f \in \text{Pi}_E\ A\ B$
obtains $x \in A$ **and** $f\ x \in B\ x$
 $\mid x \notin A$ **and** $f\ x = \text{undefined}$
using *assms* **by** (*auto simp: Pi-def PiE-def extensional-def*)

lemma *PiE-I*[*intro!*]:
 $(\bigwedge x. x \in A \implies f\ x \in B\ x) \implies (\bigwedge x. x \notin A \implies f\ x = \text{undefined}) \implies f \in \text{Pi}_E\ A\ B$
by (*simp add: PiE-def extensional-def*)

lemma *PiE-mono*: $(\bigwedge x. x \in A \implies B\ x \subseteq C\ x) \implies \text{Pi}_E\ A\ B \subseteq \text{Pi}_E\ A\ C$
by *auto*

lemma *PiE-iff*: $f \in \text{Pi}_E\ I\ X \longleftrightarrow (\forall i \in I. f\ i \in X\ i) \wedge f \in \text{extensional}\ I$
by (*simp add: PiE-def Pi-iff*)

lemma *restrict-PiE-iff*: $\text{restrict}\ f\ I \in \text{Pi}_E\ I\ X \longleftrightarrow (\forall i \in I. f\ i \in X\ i)$
by (*simp add: PiE-iff*)

lemma *ext-funcset-to-sing-iff* [*simp*]: $A \rightarrow_E \{a\} = \{\lambda x \in A. a\}$
by (*auto simp: PiE-def Pi-iff extensionalityI*)

lemma *PiE-restrict*[*simp*]: $f \in \text{Pi}_E\ A\ B \implies \text{restrict}\ f\ A = f$
by (*simp add: extensional-restrict PiE-def*)

lemma *restrict-PiE*[*simp*]: $\text{restrict}\ f\ I \in \text{Pi}_E\ I\ S \longleftrightarrow f \in \text{Pi}\ I\ S$
by (*auto simp: PiE-iff*)

lemma *PiE-eq-subset*:
assumes *ne*: $\bigwedge i. i \in I \implies F\ i \neq \{\}$ $\bigwedge i. i \in I \implies F'\ i \neq \{\}$
and *eq*: $\text{Pi}_E\ I\ F = \text{Pi}_E\ I\ F'$
and $i \in I$
shows $F\ i \subseteq F'\ i$
proof
fix x
assume $x \in F\ i$
with *ne* **have** $\forall j. \exists y. (j \in I \longrightarrow y \in F\ j \wedge (i = j \longrightarrow x = y)) \wedge (j \notin I \longrightarrow y = \text{undefined})$
by *auto*
from *choice*[*OF this*] **obtain** f
where f : $\forall j. (j \in I \longrightarrow f\ j \in F\ j \wedge (i = j \longrightarrow x = f\ j)) \wedge (j \notin I \longrightarrow f\ j = \text{undefined})$..
then have $f \in \text{Pi}_E\ I\ F$
by (*auto simp: extensional-def PiE-def*)
then have $f \in \text{Pi}_E\ I\ F'$

```

    using assms by simp
  then show  $x \in F' i$ 
    using  $f \langle i \in I \rangle$  by (auto simp: PiE-def)
qed

```

lemma *PiE-eq-iff-not-empty*:

```

  assumes  $ne: \bigwedge i. i \in I \implies F i \neq \{\}$   $\bigwedge i. i \in I \implies F' i \neq \{\}$ 
  shows  $Pi_E I F = Pi_E I F' \longleftrightarrow (\forall i \in I. F i = F' i)$ 
proof (intro iffI ballI)
  fix i
  assume  $eq: Pi_E I F = Pi_E I F'$ 
  assume  $i: i \in I$ 
  show  $F i = F' i$ 
    using PiE-eq-subset[of I F F', OF ne eq i]
    using PiE-eq-subset[of I F' F, OF ne(2,1) eq[symmetric] i]
    by auto
qed (auto simp: PiE-def)

```

lemma *PiE-eq-iff*: $Pi_E I F = Pi_E I F' \longleftrightarrow (\forall i \in I. F i = F' i) \vee ((\exists i \in I. F i = \{\}) \wedge (\exists i \in I. F' i = \{\}))$

proof (intro iffI disjCI)

```

  assume  $eq[simp]: Pi_E I F = Pi_E I F'$ 
  assume  $\neg ((\exists i \in I. F i = \{\}) \wedge (\exists i \in I. F' i = \{\}))$ 
  then have  $(\forall i \in I. F i \neq \{\}) \wedge (\forall i \in I. F' i \neq \{\})$ 
    using PiE-eq-empty-iff[of I F] PiE-eq-empty-iff[of I F'] by auto
  with PiE-eq-iff-not-empty[of I F F'] show  $\forall i \in I. F i = F' i$ 
    by auto

```

next

```

  assume  $(\forall i \in I. F i = F' i) \vee ((\exists i \in I. F i = \{\}) \wedge (\exists i \in I. F' i = \{\}))$ 
  then show  $Pi_E I F = Pi_E I F'$ 
    using PiE-eq-empty-iff[of I F] PiE-eq-empty-iff[of I F'] by (auto simp: PiE-def)
qed

```

lemma *extensional-funcset-fun-upd-restricts-rangeI*:

```

 $\forall y \in S. f x \neq f y \implies f \in (insert x S) \rightarrow_E T \implies f(x := undefined) \in S \rightarrow_E$ 
 $(T - \{f x\})$ 
  unfolding extensional-funcset-def extensional-def
  by (auto split: if-split-asm)

```

lemma *extensional-funcset-fun-upd-extends-rangeI*:

```

  assumes  $a \in T$   $f \in S \rightarrow_E (T - \{a\})$ 
  shows  $f(x := a) \in insert x S \rightarrow_E T$ 
  using assms unfolding extensional-funcset-def extensional-def by auto

```

lemma *subset-PiE*:

```

 $Pi_E I S \subseteq Pi_E I T \longleftrightarrow Pi_E I S = \{\} \vee (\forall i \in I. S i \subseteq T i)$  (is ?lhs  $\longleftrightarrow$  -  $\vee$  ?rhs)
proof (cases  $Pi_E I S = \{\}$ )
  case False

```

```

moreover have ?lhs = ?rhs
proof
  assume L: ?lhs
  have  $\bigwedge i. i \in I \implies S\ i \neq \{\}$ 
    using False PiE-eq-empty-iff by blast
  with L show ?rhs
    by (simp add: PiE-Int PiE-eq-iff inf.absorb-iff2)
qed auto
ultimately show ?thesis
  by simp
qed simp

lemma PiE-eq:  $PiE\ I\ S = PiE\ I\ T \longleftrightarrow PiE\ I\ S = \{\} \wedge PiE\ I\ T = \{\} \vee (\forall i \in I. S\ i = T\ i)$ 
  by (auto simp: PiE-eq-iff PiE-eq-empty-iff)

lemma PiE-UNIV [simp]:  $PiE\ UNIV\ (\lambda i. UNIV) = UNIV$ 
  by blast

lemma image-projection-PiE:
   $(\lambda f. f\ i) \text{ ' } (PiE\ I\ S) = (if\ PiE\ I\ S = \{\} \text{ then } \{\} \text{ else if } i \in I \text{ then } S\ i \text{ else } \{undefined\})$ 
proof –
  have  $(\lambda f. f\ i) \text{ ' } PiE\ I\ S = S\ i$  if  $i \in I$  if  $f \in PiE\ I\ S$  for  $f$ 
proof –
  have  $x \in S\ i \implies \exists f \in PiE\ I\ S. x = f\ i$  for  $x$ 
    using that
    by (force intro: bexI [where  $x = \lambda k. if\ k = i \text{ then } x \text{ else } f\ k$ ])
  then show ?thesis
    using that by force
qed
then show ?thesis
  by (smt (verit) PiE-arb equals0I image-cong image-constant image-empty)
qed

lemma PiE-singleton:
  assumes  $f \in extensional\ A$ 
  shows  $PiE\ A\ (\lambda x. \{f\ x\}) = \{f\}$ 
proof –
  have  $g = f$  if  $g \in PiE\ A\ (\lambda x. \{f\ x\})$  for  $g$ 
proof –
  from that have  $g\ x = f\ x$  for  $x$ 
    using assms by (cases  $x \in A$ ) (auto simp: extensional-def)
  then show ?thesis by (simp add: fun-eq-iff)
qed
with assms show ?thesis
  by (auto simp: extensional-def)
qed

```

lemma *PiE-eq-singleton*: $(\Pi_E i \in I. S\ i) = \{\lambda i \in I. f\ i\} \longleftrightarrow (\forall i \in I. S\ i = \{f\ i\})$
by (*metis* (*mono-tags*, *lifting*) *PiE-eq PiE-singleton insert-not-empty restrict-apply'*
restrict-extensional)

lemma *PiE-over-singleton-iff*: $(\Pi_E x \in \{a\}. B\ x) = (\bigcup b \in B\ a. \{\lambda x \in \{a\}. b\})$

proof –

have $\exists x a \in B\ a. x = (\lambda x \in \{a\}. x a)$ **if** $x a \in B\ a$ **and** $x \in \text{extensional } \{a\}$ **for** x

using *that PiE-singleton* **by** *fastforce*

then show *?thesis*

by (*auto simp: PiE-iff split: if-split-asm*)

qed

lemma *all-PiE-elements*:

$(\forall z \in \text{PiE } I\ S. \forall i \in I. P\ i\ (z\ i)) \longleftrightarrow \text{PiE } I\ S = \{\} \vee (\forall i \in I. \forall x \in S\ i. P\ i\ x)$

(**is** *?lhs = ?rhs*)

proof (*cases PiE I S = {}*)

case *False*

then obtain f **where** $f: \bigwedge i. i \in I \implies f\ i \in S\ i$

by *fastforce*

show *?thesis*

proof

assume $L: ?lhs$

have $P\ i\ x$

if $i \in I$ $x \in S\ i$ **for** $i\ x$

proof –

have $(\lambda j \in I. \text{if } j=i \text{ then } x \text{ else } f\ j) \in \text{PiE } I\ S$

by (*simp add: f that(2)*)

then have $P\ i\ ((\lambda j \in I. \text{if } j=i \text{ then } x \text{ else } f\ j)\ i)$

using L **that** **by** *blast*

with that **show** *?thesis*

by *simp*

qed

then show *?rhs*

by (*simp add: False*)

qed *fastforce*

qed *simp*

lemma *PiE-ext*: $\llbracket x \in \text{PiE } k\ s; y \in \text{PiE } k\ s; \bigwedge i. i \in k \implies x\ i = y\ i \rrbracket \implies x = y$

by (*metis ext PiE-E*)

27.7.1 Injective Extensional Function Spaces

lemma *extensional-funcset-fun-upd-inj-onI*:

assumes $f \in S \rightarrow_E (T - \{a\})$

and *inj-on f S*

shows *inj-on (f(x := a)) S*

using *assms*

unfolding *extensional-funcset-def* **by** (*auto intro!: inj-on-fun-updI*)

lemma *extensional-funcset-extend-domain-inj-on-eq*:

assumes $x \notin S$
shows $\{f. f \in (\text{insert } x \ S) \rightarrow_E T \wedge \text{inj-on } f \ (\text{insert } x \ S)\} =$
 $(\lambda(y, g). g(x:=y)) \ \cdot \ \{(y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge \text{inj-on } g \ S\}$
proof –
have *False* **if** $f \in S \rightarrow_E T - \{a\}$ **and** $a = (\text{if } y = x \text{ then } a \text{ else } f \ y)$ **and** $y \in S$
for $a \ f \ y$
using *assms that* **by** (*auto dest! PiE-mem split if-split-asm*)
moreover
have $\exists b. b \in S \rightarrow_E T - \{f \ x\} \wedge \text{inj-on } b \ S \wedge f = b(x := f \ x)$
if $f \in \text{insert } x \ S \rightarrow_E T$ **and** $\text{inj-on } f \ S$ **and** $\forall x b \in S. f \ x \neq f \ x b$ **for** f
using *that*
unfolding *inj-on-def*
by (*smt (verit, ccfv-threshold) PiE-restrict fun-upd-apply fun-upd-triv insert-Diff insert-iff*
restrict-PiE-iff restrict-upd)
ultimately show *?thesis*
using *assms*
apply (*auto simp: image-iff intro: extensional-funcset-fun-upd-inj-onI*
extensional-funcset-fun-upd-extends-rangeI del: PiE-I PiE-E)
apply (*smt (verit, best) PiE-cong PiE-mem inj-on-def insertCI*)
apply *blast*
done
qed

lemma *extensional-funcset-extend-domain-inj-onI*:

assumes $x \notin S$
shows $\text{inj-on } (\lambda(y, g). g(x := y)) \ \{(y, g). y \in T \wedge g \in S \rightarrow_E (T - \{y\}) \wedge$
 $\text{inj-on } g \ S\}$
using *assms*
by (*simp add: inj-on-def*) (*metis PiE-restrict fun-upd-apply restrict-fupd*)

27.7.2 Misc properties of functions, composition and restriction from HOL Light

lemma *function-factors-left-gen*:

$(\forall x \ y. P \ x \wedge P \ y \wedge g \ x = g \ y \longrightarrow f \ x = f \ y) \longleftrightarrow (\exists h. \forall x. P \ x \longrightarrow f \ x = h(g \ x))$
(is ?lhs = ?rhs)

proof

assume $L: ?lhs$
then show *?rhs*
apply (*rule-tac x=f o inv-into (Collect P) g in exI*)
unfolding *o-def*
by (*metis (mono-tags, opaque-lifting) f-inv-into-f imageI inv-into-into mem-Collect-eq*)
qed *auto*

lemma *function-factors-left*: $(\forall x \ y. (g \ x = g \ y) \longrightarrow (f \ x = f \ y)) \longleftrightarrow (\exists h. f = h \circ g)$

using *function-factors-left-gen* [*of* $\lambda x. \text{True } g \ f$] **unfolding** *o-def* **by** *blast*

lemma *function-factors-right-gen*: $(\forall x. P\ x \longrightarrow (\exists y. g\ y = f\ x)) \longleftrightarrow (\exists h. \forall x. P\ x \longrightarrow f\ x = g(h\ x))$
by *metis*

lemma *function-factors-right*: $(\forall x. \exists y. g\ y = f\ x) \longleftrightarrow (\exists h. f = g \circ h)$
unfolding *o-def* **by** *metis*

lemma *restrict-compose-right*: $\text{restrict}\ (g \circ \text{restrict}\ f\ S)\ S = \text{restrict}\ (g \circ f)\ S$
by *auto*

lemma *restrict-compose-left*: $f\ ‘\ S \subseteq T \implies \text{restrict}\ (\text{restrict}\ g\ T \circ f)\ S = \text{restrict}\ (g \circ f)\ S$
by *fastforce*

27.7.3 Cardinality

lemma *finite-PiE*: $\text{finite}\ S \implies (\bigwedge i. i \in S \implies \text{finite}\ (T\ i)) \implies \text{finite}\ (\Pi_E\ i \in S. T\ i)$
by (*induct* *S* *arbitrary*: *T* *rule*: *finite-induct*) (*simp-all* *add*: *PiE-insert-eq*)

lemma *inj-combinator*: $x \notin S \implies \text{inj-on}\ (\lambda(y, g). g(x := y))\ (T\ x \times \Pi_E\ S\ T)$

proof (*safe intro!*: *inj-onI ext*)
fix *f y g z*
assume $x \notin S$
assume $fg: f \in \Pi_E\ S\ T\ g \in \Pi_E\ S\ T$
assume $f(x := y) = g(x := z)$
then have $*$: $\bigwedge i. (f(x := y))\ i = (g(x := z))\ i$
unfolding *fun-eq-iff* **by** *auto*
from *this*[*of x*] **show** $y = z$ **by** *simp*
fix *i* **from** $*[of\ i]$ $\langle x \notin S \rangle fg$ **show** $f\ i = g\ i$
by (*auto split*: *if-split-asm simp*: *PiE-def extensional-def*)
qed

lemma *card-PiE*: $\text{finite}\ S \implies \text{card}\ (\Pi_E\ i \in S. T\ i) = (\prod_{i \in S.} \text{card}\ (T\ i))$

proof (*induct* *rule*: *finite-induct*)
case *empty*
then show *?case*
by *auto*
next
case (*insert* *x S*)
then show *?case*
by (*simp* *add*: *PiE-insert-eq inj-combinator card-image card-cartesian-product*)
qed

lemma *card-funcsetE*: $\text{finite}\ A \implies \text{card}\ (A \rightarrow_E B) = \text{card}\ B \wedge \text{card}\ A$
by (*subst* *card-PiE*) *auto*

lemma *card-inj-on-subset-funcset*:


```

assumes finB: finite B
and finC: finite C
and AB:  $A \subseteq B$ 
shows  $\text{card } \{f \in B \rightarrow_E C. \text{inj-on } f \ A\} =$ 
 $\text{card } C \wedge (\text{card } B - \text{card } A) * \text{prod } ((-) (\text{card } C)) \{0 \ ..< \text{card } A\}$ 
proof –
define D where  $D = B - A$ 
from AB have  $B: B = A \cup D$  and disj:  $A \cap D = \{\}$ 
unfolding D-def by auto
have sub:  $\text{card } B - \text{card } A = \text{card } D$ 
unfolding D-def using finB AB
by (metis card-Diff-subset finite-subset)
from finB B have finite A finite D by auto
then show ?thesis
unfolding sub unfolding B using disj
proof (induct A rule: finite-induct)
case empty
from card-funcsetE[OF this(1), of C] show ?case
by auto
next
case (insert a A)
have  $\{f. f \in \text{insert } a \ A \cup D \rightarrow_E C \wedge \text{inj-on } f \ (\text{insert } a \ A)\} =$ 
 $\{f(a := c) \mid f \ c. f \in A \cup D \rightarrow_E C \wedge \text{inj-on } f \ A \wedge c \in C - f \ ' A\}$ 
(is ?l = ?r)
proof
show ?r  $\subseteq$  ?l
by (auto intro: inj-on-fun-updI split: if-splits)
have  $f \in ?r$  if  $f: f \in ?l$  for  $f$ 
proof –
let ?g  $= f(a := \text{undefined})$ 
let ?h  $= ?g(a := f \ a)$ 
have mem:  $f \ a \in C - ?g \ ' A$  using insert(1,2,4,5) f by auto
from  $f$  have  $f: f \in \text{insert } a \ A \cup D \rightarrow_E C \wedge \text{inj-on } f \ (\text{insert } a \ A)$  by auto
hence  $?g \in A \cup D \rightarrow_E C \wedge \text{inj-on } ?g \ A$  using  $\langle a \notin A \rangle \langle \text{insert } a \ A \cap D = \{\} \rangle$ 
by (auto split: if-splits simp: inj-on-def)
with mem have  $?h \in ?r$  by blast
also have  $?h = f$  by auto
finally show ?thesis .
qed
then show ?l  $\subseteq$  ?r by auto
qed
also have  $\dots = (\lambda (f, c). f \ (a := c)) \ ' ($ 
 $\text{Sigma } \{f . f \in A \cup D \rightarrow_E C \wedge \text{inj-on } f \ A\} \ (\lambda f. C - f \ ' A))$ 
by auto
also have  $\text{card } (\dots) = \text{card } (\text{Sigma } \{f . f \in A \cup D \rightarrow_E C \wedge \text{inj-on } f \ A\} \ (\lambda f.$ 
 $C - f \ ' A))$ 
proof (rule card-image, intro inj-onI, clarsimp, goal-cases)
case (1 f c g d)
let  $?f = f(a := c, a := \text{undefined})$ 

```

```

let ?g = g(a := d, a := undefined)
from 1 have id: f(a := c) = g(a := d)
  by auto
from fun-upd-eqD[OF id]
have cd: c = d
  by auto
from id have ?f = ?g
  by auto
also have ?f = f using ⟨f ∈ A ∪ D →E C⟩ insert(1,2,4,5)
  by (intro ext, auto)
also have ?g = g using ⟨g ∈ A ∪ D →E C⟩ insert(1,2,4,5)
  by (intro ext, auto)
finally show f = g ∧ c = d
  using cd by auto
qed
also have ... = (∑ f ∈ {f ∈ A ∪ D →E C. inj-on f A}. card (C - f ‘ A))
  by (rule card-SigmaI, rule finite-subset[of - A ∪ D →E C],
      insert ⟨finite C⟩ ⟨finite D⟩ ⟨finite A⟩, auto intro!: finite-PiE)
also have ... = (∑ f ∈ {f ∈ A ∪ D →E C. inj-on f A}. card C - card A)
  by (rule sum.cong[OF refl], subst card-Diff-subset, insert ⟨finite A⟩, auto simp:
card-image)
also have ... = (card C - card A) * card {f ∈ A ∪ D →E C. inj-on f A}
  by simp
also have ... = card C ^ card D * ((card C - card A) * prod ((-) (card C))
{0..

```

27.8 The pigeonhole principle

An alternative formulation of this is that for a function mapping a finite set A of cardinality m to a finite set B of cardinality n , there exists an element $y \in B$ that is hit at least $\lceil \frac{m}{n} \rceil$ times. However, since we do not have real numbers or rounding yet, we state it in the following equivalent form:

lemma *pigeonhole-card*:

assumes $f \in A \rightarrow B$ *finite A finite B B ≠ {}*
shows $\exists y \in B. \text{card } (f - \{y\} \cap A) * \text{card } B \geq \text{card } A$

proof –

from *assms* **have** $\text{card } B > 0$

by *auto*

define M **where** $M = \text{Max } ((\lambda y. \text{card } (f - \{y\} \cap A)) \text{ ‘ } B)$

have $A = (\bigcup y \in B. f - \{y\} \cap A)$

using *assms* **by** *auto*

also have $\text{card } \dots = (\sum i \in B. \text{card } (f - \{i\} \cap A))$

```

    using assms by (subst card-UN-disjoint) auto
  also have ... ≤ (∑ i∈B. M)
    unfolding M-def using assms by (intro sum-mono Max.coboundedI) auto
  also have ... = card B * M
    by simp
  finally have *: M * card B ≥ card A
    by (simp add: mult-ac)
  from assms have M ∈ (λy. card (f - ‘{y} ∩ A)) ‘ B
    unfolding M-def by (intro Max-in) auto
  with * show ?thesis
    by blast
qed

```

27.9 Products of sums

lemma *prod-sum-PiE*:

```

  fixes f :: 'a ⇒ 'b ⇒ 'c :: comm-semiring-1
  assumes finite: finite A and finite: ∧x. x ∈ A ⇒ finite (B x)
  shows (∏ x∈A. ∑ y∈B x. f x y) = (∑ g∈PiE A B. ∏ x∈A. f x (g x))
    using assms
  proof (induction A rule: finite-induct)
    case empty
      thus ?case by auto
    next
      case (insert x A)
        have (∑ g∈PiE (insert x A) B. ∏ x∈insert x A. f x (g x)) =
          (∑ g∈PiE (insert x A) B. f x (g x) * (∏ x'∈A. f x' (g x')))
          using insert by simp
        also have (λg. ∏ x'∈A. f x' (g x')) = (λg. ∏ x'∈A. f x' (if x' = x then undefined
          else g x'))
          using insert by (intro ext prod.cong) auto
        also have (∑ g∈PiE (insert x A) B. f x (g x) * ... g) =
          (∑ (y,g)∈B x × PiE A B. f x y * (∏ x'∈A. f x' (g x')))
          using insert.premis insert.hyps
        by (intro sum.reindex-bij-witness[of - λ(y,g). g(x := y) λg. (g x, g(x := unde-
          fined))])
          (auto simp: PiE-def extensional-def)
        also have ... = (∑ y∈B x. ∑ g∈PiE A B. f x y * (∏ x'∈A. f x' (g x')))
          by (subst sum.cartesian-product) auto
        also have ... = (∑ y∈B x. f x y) * (∑ g∈PiE A B. ∏ x'∈A. f x' (g x'))
          using insert by (subst sum.swap) (simp add: sum-distrib-left sum-distrib-right)
        also have (∑ g∈PiE A B. ∏ x'∈A. f x' (g x')) = (∏ x∈A. ∑ y∈B x. f x y)
          using insert.premis by (intro insert.IH [symmetric]) auto
        also have (∑ y∈B x. f x y) * ... = (∏ x∈insert x A. ∑ y∈B x. f x y)
          using insert.hyps by simp
        finally show ?case ..
      qed
    end
  end

```

28 Partitions and Disjoint Sets

```
theory Disjoint-Sets
  imports FuncSet
begin
```

```
lemma mono-imp-UN-eq-last: mono A  $\implies (\bigcup_{i \leq n}. A\ i) = A\ n$ 
  unfolding mono-def by auto
```

28.1 Set of Disjoint Sets

```
abbreviation disjoint :: 'a set set  $\Rightarrow$  bool where disjoint  $\equiv$  pairwise disjnt
```

```
lemma disjoint-def: disjoint A  $\longleftrightarrow (\forall a \in A. \forall b \in A. a \neq b \longrightarrow a \cap b = \{\})$ 
  unfolding pairwise-def disjnt-def by auto
```

```
lemma disjointI:
  ( $\bigwedge a\ b. a \in A \implies b \in A \implies a \neq b \implies a \cap b = \{\}$ )  $\implies$  disjoint A
  unfolding disjoint-def by auto
```

```
lemma disjointD:
  disjoint A  $\implies a \in A \implies b \in A \implies a \neq b \implies a \cap b = \{\}$ 
  unfolding disjoint-def by auto
```

```
lemma disjoint-image: inj-on f ( $\bigcup A$ )  $\implies$  disjoint A  $\implies$  disjoint (( $\cdot$ ) f ` A)
  unfolding inj-on-def disjoint-def by blast
```

```
lemma assumes disjoint (A  $\cup$  B)
  shows disjoint-unionD1: disjoint A and disjoint-unionD2: disjoint B
  using assms by (simp-all add: disjoint-def)
```

```
lemma disjoint-INT:
  assumes *:  $\bigwedge i. i \in I \implies$  disjoint (F i)
  shows disjoint  $\{\bigcap_{i \in I}. X\ i \mid X. \forall i \in I. X\ i \in F\ i\}$ 
proof (safe intro!: disjointI del: equalityI)
  fix A B :: 'a  $\Rightarrow$  'b set assume  $(\bigcap_{i \in I}. A\ i) \neq (\bigcap_{i \in I}. B\ i)$ 
  then obtain i where A i  $\neq$  B i i  $\in$  I
    by auto
  moreover assume  $\forall i \in I. A\ i \in F\ i \ \forall i \in I. B\ i \in F\ i$ 
  ultimately show  $(\bigcap_{i \in I}. A\ i) \cap (\bigcap_{i \in I}. B\ i) = \{\}$ 
    using *[OF  $\langle i \in I \rangle$ , THEN disjointD, of A i B i]
    by (auto simp flip: INT-Int-distrib)
qed
```

```
lemma diff-Union-pairwise-disjoint:
  assumes pairwise disjnt  $\mathcal{A}\ \mathcal{B} \subseteq \mathcal{A}$ 
  shows  $\bigcup \mathcal{A} - \bigcup \mathcal{B} = \bigcup (\mathcal{A} - \mathcal{B})$ 
proof -
  have False
    if x:  $x \in A\ x \in B$  and AB:  $A \in \mathcal{A}\ A \notin \mathcal{B}\ B \in \mathcal{B}$  for  $x\ A\ B$ 
```

```

proof –
  have  $A \cap B = \{\}$ 
    using assms disjointD AB by blast
with  $x$  show ?thesis
  by blast
qed
then show ?thesis by auto
qed

```

lemma *Int-Union-pairwise-disjoint:*

```

assumes pairwise disjnt ( $\mathcal{A} \cup \mathcal{B}$ )
shows  $\bigcup \mathcal{A} \cap \bigcup \mathcal{B} = \bigcup (\mathcal{A} \cap \mathcal{B})$ 
proof –
  have False
    if  $x$ :  $x \in A \ x \in B$  and  $AB$ :  $A \in \mathcal{A} \ A \notin \mathcal{B} \ B \in \mathcal{B}$  for  $x \ A \ B$ 
  proof –
    have  $A \cap B = \{\}$ 
      using assms disjointD AB by blast
    with  $x$  show ?thesis
    by blast
  qed
then show ?thesis by auto
qed

```

lemma *psubset-Union-pairwise-disjoint:*

```

assumes  $\mathcal{B}$ : pairwise disjnt  $\mathcal{B}$  and  $\mathcal{A} \subset \mathcal{B} - \{\{\}\}$ 
shows  $\bigcup \mathcal{A} \subset \bigcup \mathcal{B}$ 
unfolding psubset-eq
proof
show  $\bigcup \mathcal{A} \subseteq \bigcup \mathcal{B}$ 
  using assms by blast
have  $\mathcal{A} \subseteq \mathcal{B} \cup (\mathcal{B} - \mathcal{A} \cap (\mathcal{B} - \{\{\}\})) \neq \{\}$ 
  using assms by blast+
then show  $\bigcup \mathcal{A} \neq \bigcup \mathcal{B}$ 
  using diff-Union-pairwise-disjoint [OF  $\mathcal{B}$ ] by blast
qed

```

28.1.1 Family of Disjoint Sets

definition *disjoint-family-on* :: $('i \Rightarrow 'a \text{ set}) \Rightarrow 'i \text{ set} \Rightarrow \text{bool}$ **where**
disjoint-family-on $A \ S \longleftrightarrow (\forall m \in S. \forall n \in S. m \neq n \longrightarrow A \ m \cap A \ n = \{\})$

abbreviation *disjoint-family* $A \equiv \text{disjoint-family-on } A \ \text{UNIV}$

lemma *disjoint-family-elem-disjnt:*

```

assumes infinite  $A$  finite  $C$ 
and df: disjoint-family-on  $B \ A$ 
obtains  $x$  where  $x \in A$  disjnt  $C \ (B \ x)$ 
proof –

```

```

have False if *:  $\forall x \in A. \exists y. y \in C \wedge y \in B \ x$ 
proof -
  obtain g where g:  $\forall x \in A. g \ x \in C \wedge g \ x \in B \ x$ 
  using * by metis
  with df have inj-on g A
  by (fastforce simp add: inj-on-def disjoint-family-on-def)
  then have infinite (g ‘ A)
  using ⟨infinite A⟩ finite-image-iff by blast
  then show False
  by (meson ⟨finite C⟩ finite-subset g image-subset-iff)
qed
then show ?thesis
by (force simp: disjnt-iff intro: that)
qed

```

lemma *disjoint-family-onD*:
 $disjoint-family-on \ A \ I \implies i \in I \implies j \in I \implies i \neq j \implies A \ i \cap A \ j = \{\}$
by (*auto simp: disjoint-family-on-def*)

lemma *disjoint-family-subset*: $disjoint-family \ A \implies (\bigwedge x. B \ x \subseteq A \ x) \implies disjoint-family \ B$
by (*force simp add: disjoint-family-on-def*)

lemma *disjoint-family-on-insert*:
 $i \notin I \implies disjoint-family-on \ A \ (insert \ i \ I) \longleftrightarrow A \ i \cap (\bigcup_{i \in I}. A \ i) = \{\} \wedge$
 $disjoint-family-on \ A \ I$
by (*fastforce simp: disjoint-family-on-def*)

lemma *disjoint-family-on-bisimulation*:
assumes *disjoint-family-on f S*
and $\bigwedge n \ m. n \in S \implies m \in S \implies n \neq m \implies f \ n \cap f \ m = \{\} \implies g \ n \cap g \ m = \{\}$
shows *disjoint-family-on g S*
using *assms unfolding disjoint-family-on-def* **by** *auto*

lemma *disjoint-family-on-mono*:
 $A \subseteq B \implies disjoint-family-on \ f \ B \implies disjoint-family-on \ f \ A$
unfolding *disjoint-family-on-def* **by** *auto*

lemma *disjoint-family-Suc*:
 $(\bigwedge n. A \ n \subseteq A \ (Suc \ n)) \implies disjoint-family \ (\lambda i. A \ (Suc \ i) - A \ i)$
using *lift-Suc-mono-le[of A]*
by (*auto simp add: disjoint-family-on-def*)
(metis insert-absorb insert-subset le-SucE le-antisym not-le-imp-less less-imp-le)

lemma *disjoint-family-on-disjoint-image*:
 $disjoint-family-on \ A \ I \implies disjoint \ (A \ ‘ I)$
unfolding *disjoint-family-on-def disjoint-def* **by** *force*

lemma *disjoint-family-on-vimageI*: $\text{disjoint-family-on } F \ I \implies \text{disjoint-family-on } (\lambda i. f - ' F \ i) \ I$

by (*auto simp: disjoint-family-on-def*)

lemma *disjoint-image-disjoint-family-on*:

assumes d : *disjoint* $(A - ' I)$ **and** i : *inj-on* $A \ I$

shows *disjoint-family-on* $A \ I$

unfolding *disjoint-family-on-def*

proof (*intro ballI impI*)

fix $n \ m$ **assume** nm : $m \in I \ n \in I$ **and** $n \neq m$

with i [*THEN inj-onD, of n m*] **show** $A \ n \cap A \ m = \{\}$

by (*intro disjointD[OF d]*) *auto*

qed

lemma *disjoint-family-on-iff-disjoint-image*:

assumes $\bigwedge i. i \in I \implies A \ i \neq \{\}$

shows *disjoint-family-on* $A \ I \longleftrightarrow \text{disjoint} (A - ' I) \wedge \text{inj-on } A \ I$

proof

assume *disjoint-family-on* $A \ I$

then show $\text{disjoint} (A - ' I) \wedge \text{inj-on } A \ I$

by (*metis (mono-tags, lifting) assms disjoint-family-onD disjoint-family-on-disjoint-image inf.idem inj-onI*)

qed (*use disjoint-image-disjoint-family-on in metis*)

lemma *card-UN-disjoint'*:

assumes *disjoint-family-on* $A \ I \wedge i. i \in I \implies \text{finite} (A \ i) \ \text{finite } I$

shows $\text{card} (\bigcup_{i \in I}. A \ i) = (\sum_{i \in I}. \text{card} (A \ i))$

using *assms* **by** (*simp add: card-UN-disjoint disjoint-family-on-def*)

lemma *disjoint-UN*:

assumes F : $\bigwedge i. i \in I \implies \text{disjoint} (F \ i)$ **and** $*$: *disjoint-family-on* $(\lambda i. \bigcup (F \ i)) \ I$

shows *disjoint* $(\bigcup_{i \in I}. F \ i)$

proof (*safe intro!: disjointI del: equalityI*)

fix $A \ B \ i \ j$ **assume** $A \neq B \ A \in F \ i \ i \in I \ B \in F \ j \ j \in I$

show $A \cap B = \{\}$

proof *cases*

assume $i = j$ **with** F [*of i*] $\langle i \in I \rangle \langle A \in F \ i \rangle \langle B \in F \ j \rangle \langle A \neq B \rangle$ **show** $A \cap B = \{\}$

by (*auto dest: disjointD*)

next

assume $i \neq j$

with $*$ $\langle i \in I \rangle \langle j \in I \rangle$ **have** $(\bigcup (F \ i)) \cap (\bigcup (F \ j)) = \{\}$

by (*rule disjoint-family-onD*)

with $\langle A \in F \ i \rangle \langle i \in I \rangle \langle B \in F \ j \rangle \langle j \in I \rangle$

show $A \cap B = \{\}$

by *auto*

qed

qed

lemma *distinct-list-bind*:

assumes *distinct xs* $\bigwedge x. x \in \text{set } xs \implies \text{distinct } (f\ x)$
 disjoint-family-on (*set* \circ *f*) (*set xs*)
shows *distinct* (*List.bind xs f*)
using *assms*
by (*induction xs*)
 (*auto simp: disjoint-family-on-def distinct-map inj-on-def set-list-bind*)

lemma *bij-betw-UNION-disjoint*:

assumes *disj*: *disjoint-family-on* *A' I*
assumes *bij*: $\bigwedge i. i \in I \implies \text{bij-betw } f\ (A\ i)\ (A'\ i)$
shows *bij-betw* *f* $(\bigcup_{i \in I}. A\ i)\ (\bigcup_{i \in I}. A'\ i)$
unfolding *bij-betw-def*
proof
 from *bij* **show** *eq*: $f\ ' \bigcup (A\ ' I) = \bigcup (A'\ ' I)$
 by (*auto simp: bij-betw-def image-UN*)
show *inj-on* *f* $(\bigcup (A\ ' I))$
proof (*rule inj-onI, clarify*)
 fix *i j x y* **assume** *A*: $i \in I\ j \in I\ x \in A\ i\ y \in A\ j$ **and** *B*: $f\ x = f\ y$
 from *A* *bij*[*of i*] *bij*[*of j*] **have** $f\ x \in A'\ i\ f\ y \in A'\ j$
 by (*auto simp: bij-betw-def*)
 with *B* **have** $A'\ i \cap A'\ j \neq \{\}$ **by** *auto*
 with *disj A* **have** $i = j$ **unfolding** *disjoint-family-on-def* **by** *blast*
 with *A B* *bij*[*of i*] **show** $x = y$ **by** (*auto simp: bij-betw-def dest: inj-onD*)
qed
qed

lemma *disjoint-union*: *disjoint C* \implies *disjoint B* $\implies \bigcup C \cap \bigcup B = \{\}$ \implies *disjoint* $(C \cup B)$

using *disjoint-UN*[*of* $\{C, B\} \lambda x. x$] **by** (*auto simp add: disjoint-family-on-def*)

Sum/product of the union of a finite disjoint family

context *comm-monoid-set*

begin

lemma *UNION-disjoint-family*:

assumes *finite I* **and** $\forall i \in I. \text{finite } (A\ i)$
 and *disjoint-family-on* *A I*
shows $F\ g\ (\bigcup (A\ ' I)) = F\ (\lambda x. F\ g\ (A\ x))\ I$
using *assms* **unfolding** *disjoint-family-on-def* **by** (*rule UNION-disjoint*)

lemma *Union-disjoint-sets*:

assumes $\forall A \in C. \text{finite } A$ **and** *disjoint C*
shows $F\ g\ (\bigcup C) = (F \circ F)\ g\ C$
using *assms* **unfolding** *disjoint-def* **by** (*rule Union-disjoint*)

end

The union of an infinite disjoint family of non-empty sets is infinite.

lemma *infinite-disjoint-family-imp-infinite-UNION*:
assumes $\neg \text{finite } A \wedge x. x \in A \implies f x \neq \{\}$ *disjoint-family-on f A*
shows $\neg \text{finite } (\bigcup (f \text{ ` } A))$
proof –
define g **where** $g x = (\text{SOME } y. y \in f x)$ **for** x
have $g: g x \in f x$ **if** $x \in A$ **for** x
unfolding $g\text{-def}$ **by** (*rule someI-ex, insert assms(2) that*) *blast*
have *inj-on-g: inj-on g A*
proof (*rule inj-onI, rule ccontr*)
fix $x y$ **assume** $A: x \in A \ y \in A \ g x = g y \ x \neq y$
with $g[\text{of } x] \ g[\text{of } y]$ **have** $g x \in f x \ g x \in f y$ **by** *auto*
with $A \langle x \neq y \rangle$ *assms* **show** *False*
by (*auto simp: disjoint-family-on-def inj-on-def*)
qed
from g **have** $g \text{ ` } A \subseteq \bigcup (f \text{ ` } A)$ **by** *blast*
moreover from *inj-on-g* $\langle \neg \text{finite } A \rangle$ **have** $\neg \text{finite } (g \text{ ` } A)$
using *finite-imageD* **by** *blast*
ultimately show *?thesis* **using** *finite-subset* **by** *blast*
qed

28.2 Construct Disjoint Sequences

definition *disjointed* :: $(\text{nat} \Rightarrow 'a \text{ set}) \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$ **where**
 $\text{disjointed } A \ n = A \ n - (\bigcup_{i \in \{0..<n\}} A \ i)$

lemma *finite-UN-disjointed-eq*: $(\bigcup_{i \in \{0..<n\}} \text{disjointed } A \ i) = (\bigcup_{i \in \{0..<n\}} A \ i)$
proof (*induct n*)
case 0 **show** *?case* **by** *simp*
next
case (*Suc n*)
thus *?case* **by** (*simp add: atLeastLessThanSuc disjointed-def*)
qed

lemma *UN-disjointed-eq*: $(\bigcup i. \text{disjointed } A \ i) = (\bigcup i. A \ i)$
by (*rule UN-finite2-eq [where k=0]*)
(simp add: finite-UN-disjointed-eq)

lemma *less-disjoint-disjointed*: $m < n \implies \text{disjointed } A \ m \cap \text{disjointed } A \ n = \{\}$
by (*auto simp add: disjointed-def*)

lemma *disjoint-family-disjointed*: *disjoint-family (disjointed A)*
by (*simp add: disjoint-family-on-def*)
(metis neq-iff Int-commute less-disjoint-disjointed)

lemma *disjointed-subset*: $\text{disjointed } A \ n \subseteq A \ n$
by (*auto simp add: disjointed-def*)

lemma *disjointed-0[simp]*: $\text{disjointed } A \ 0 = A \ 0$

by (simp add: disjointed-def)

lemma *disjointed-mono*: $\text{mono } A \implies \text{disjointed } A \text{ (Suc } n) = A \text{ (Suc } n) - A \text{ } n$
using *mono-imp-UN-eq-last*[of *A*] **by** (simp add: disjointed-def atLeastLessThanSuc-atLeastAtMost atLeast0AtMost)

28.3 Partitions

Partitions P of a set A . We explicitly disallow empty sets.

definition *partition-on* :: $'a \text{ set} \Rightarrow 'a \text{ set set} \Rightarrow \text{bool}$

where

$\text{partition-on } A \ P \longleftrightarrow \bigcup P = A \wedge \text{disjoint } P \wedge \{\} \notin P$

lemma *partition-onI*:

$\bigcup P = A \implies (\bigwedge p \ q. p \in P \implies q \in P \implies p \neq q \implies \text{disjnt } p \ q) \implies \{\} \notin P$
 $\implies \text{partition-on } A \ P$

by (auto simp: partition-on-def pairwise-def)

lemma *partition-onD1*: $\text{partition-on } A \ P \implies A = \bigcup P$

by (auto simp: partition-on-def)

lemma *partition-onD2*: $\text{partition-on } A \ P \implies \text{disjoint } P$

by (auto simp: partition-on-def)

lemma *partition-onD3*: $\text{partition-on } A \ P \implies \{\} \notin P$

by (auto simp: partition-on-def)

28.4 Constructions of partitions

lemma *partition-on-empty*: $\text{partition-on } \{\} \ P \longleftrightarrow P = \{\}$

unfolding *partition-on-def* **by** *fastforce*

lemma *partition-on-space*: $A \neq \{\} \implies \text{partition-on } A \ \{A\}$

by (auto simp: partition-on-def disjoint-def)

lemma *partition-on-singletons*: $\text{partition-on } A \ ((\lambda x. \{x\}) \text{ ` } A)$

by (auto simp: partition-on-def disjoint-def)

lemma *partition-on-transform*:

assumes $P: \text{partition-on } A \ P$

assumes $F\text{-UN}$: $\bigcup (F \text{ ` } P) = F (\bigcup P)$ **and** $F\text{-disjnt}$: $\bigwedge p \ q. p \in P \implies q \in P$
 $\implies \text{disjnt } p \ q \implies \text{disjnt } (F \ p) \ (F \ q)$

shows $\text{partition-on } (F \ A) \ (F \text{ ` } P - \{\{\}\})$

proof –

have $\bigcup (F \text{ ` } P - \{\{\}\}) = F \ A$

unfolding $P[THEN \text{partition-onD1}] \ F\text{-UN}[symmetric]$ **by** *auto*

with P **show** *?thesis*

by (auto simp add: partition-on-def pairwise-def intro!: $F\text{-disjnt}$)

qed

lemma *partition-on-restrict*: $\text{partition-on } A \ P \implies \text{partition-on } (B \cap A) \ ((\cap) \ B \text{ ‘ } P - \{\{\}\})$
by (*intro partition-on-transform*) (*auto simp: disjnt-def*)

lemma *partition-on-vimage*: $\text{partition-on } A \ P \implies \text{partition-on } (f - \text{‘ } A) \ ((-\text{‘}) \ f \text{ ‘ } P - \{\{\}\})$
by (*intro partition-on-transform*) (*auto simp: disjnt-def*)

lemma *partition-on-inj-image*:
assumes P : $\text{partition-on } A \ P$ **and** f : $\text{inj-on } f \ A$
shows $\text{partition-on } (f \text{ ‘ } A) \ ((\text{‘}) \ f \text{ ‘ } P - \{\{\}\})$
proof (*rule partition-on-transform[OF P]*)
show $p \in P \implies q \in P \implies \text{disjnt } p \ q \implies \text{disjnt } (f \text{ ‘ } p) \ (f \text{ ‘ } q)$ **for** $p \ q$
using f [*THEN inj-onD*] P [*THEN partition-onD1*] **by** (*auto simp: disjnt-def*)
qed *auto*

lemma *partition-on-insert*:
assumes $\text{disjnt } p \ (\bigcup P)$
shows $\text{partition-on } A \ (\text{insert } p \ P) \longleftrightarrow \text{partition-on } (A - p) \ P \wedge p \subseteq A \wedge p \neq \{\}$
using *assms*
by (*auto simp: partition-on-def disjnt-iff pairwise-insert*)

28.5 Finiteness of partitions

lemma *finitely-many-partition-on*:
assumes $\text{finite } A$
shows $\text{finite } \{P. \text{partition-on } A \ P\}$
proof (*rule finite-subset*)
show $\{P. \text{partition-on } A \ P\} \subseteq \text{Pow } (\text{Pow } A)$
unfolding *partition-on-def* **by** *auto*
show $\text{finite } (\text{Pow } (\text{Pow } A))$
using *assms* **by** *simp*
qed

lemma *finite-elements*: $\text{finite } A \implies \text{partition-on } A \ P \implies \text{finite } P$
using *partition-onD1*[*of A P*] **by** (*simp add: finite-UnionD*)

lemma *product-partition*:
assumes $\text{partition-on } A \ P$ **and** $\bigwedge p. p \in P \implies \text{finite } p$
shows $\text{card } A = (\sum p \in P. \text{card } p)$
using *assms* **unfolding** *partition-on-def* **by** (*meson card-Union-disjoint*)

28.6 Equivalence of partitions and equivalence classes

lemma *partition-on-quotient*:
assumes r : $\text{equiv } A \ r$
shows $\text{partition-on } A \ (A // r)$
proof (*rule partition-onI*)
from r **have** $r \subseteq A \times A$ **and** $\text{refl-on } A \ r$

```

    by (auto elim: equivE)
  then show  $\bigcup (A // r) = A \setminus \{\}$   $\notin A // r$ 
    by (auto simp: refl-on-def quotient-def)

  fix p q assume  $p \in A // r$   $q \in A // r$   $p \neq q$ 
  then obtain x y where  $x \in A$   $y \in A$   $p = r \text{ `` } \{x\}$   $q = r \text{ `` } \{y\}$ 
    by (auto simp: quotient-def)
  with r equiv-class-eq-iff[OF r, of x y]  $\langle p \neq q \rangle$  show disjoint p q
    by (auto simp: disjoint-equiv-class)
qed

lemma equiv-partition-on:
  assumes P: partition-on A P
  shows equiv A  $\{(x, y). \exists p \in P. x \in p \wedge y \in p\}$ 
proof (rule equivI)
  have  $A = \bigcup P$ 
    using P by (auto simp: partition-on-def)

  show  $\{(x, y). \exists p \in P. x \in p \wedge y \in p\} \subseteq A \times A$ 
    unfolding  $\langle A = \bigcup P \rangle$  by blast

  show refl-on A  $\{(x, y). \exists p \in P. x \in p \wedge y \in p\}$ 
    unfolding refl-on-def  $\langle A = \bigcup P \rangle$  by auto
next
  show trans  $\{(x, y). \exists p \in P. x \in p \wedge y \in p\}$ 
    using P by (auto simp only: trans-def disjoint-def partition-on-def)
next
  show sym  $\{(x, y). \exists p \in P. x \in p \wedge y \in p\}$ 
    by (auto simp only: sym-def)
qed

lemma partition-on-eq-quotient:
  assumes P: partition-on A P
  shows  $A // \{(x, y). \exists p \in P. x \in p \wedge y \in p\} = P$ 
  unfolding quotient-def
proof safe
  fix x assume  $x \in A$ 
  then obtain p where  $p \in P$   $x \in p \wedge q. q \in P \implies x \in q \implies p = q$ 
    using P by (auto simp: partition-on-def disjoint-def)
  then have  $\{y. \exists p \in P. x \in p \wedge y \in p\} = p$ 
    by (safe intro!: bexI[of - p]) simp
  then show  $\{(x, y). \exists p \in P. x \in p \wedge y \in p\} \text{ `` } \{\{x\} \in P\}$ 
    by (simp add:  $\langle p \in P \rangle$ )
next
  fix p assume  $p \in P$ 
  then have  $p \neq \{\}$ 
    using P by (auto simp: partition-on-def)
  then obtain x where  $x \in p$ 
    by auto

```

then have $x \in A \wedge q. q \in P \implies x \in q \implies p = q$
using $P \langle p \in P \rangle$ **by** (*auto simp: partition-on-def disjoint-def*)
with $\langle p \in P \rangle \langle x \in p \rangle$ **have** $\{y. \exists p \in P. x \in p \wedge y \in p\} = p$
by (*safe intro!: bexI[of - p] simp*)
then show $p \in (\bigcup x \in A. \{(x, y). \exists p \in P. x \in p \wedge y \in p\})$ “ $\{x\}$ ”
by (*auto intro: $\langle x \in A \rangle$*)
qed

lemma *partition-on-alt: partition-on $A P \longleftrightarrow (\exists r. \text{equiv } A r \wedge P = A // r)$*
by (*auto simp: partition-on-eq-quotient intro!: partition-on-quotient intro: equiv-partition-on*)

lemma (*in comm-monoid-set*) *partition:*
assumes *finite X partition-on $X A$*
shows $F g X = F (\lambda B. F g B) A$
proof –
have *finite A*
using *assms finite-UnionD partition-onD1* **by** *auto*
have [*intro*]: *finite B if $B \in A$ for B*
by (*rule finite-subset[OF - assms(1)]*)
(use assms that in \langle auto simp: partition-on-def \rangle)
have $F g X = F g (\bigcup A)$
using *assms* **by** (*simp add: partition-on-def*)
also have $\dots = (F \circ F) g A$
by (*rule Union-disjoint*) *(use assms \langle finite $A \rangle$ in \langle auto simp: partition-on-def disjoint-def \rangle)*
finally show *?thesis*
by *simp*
qed

If h is an involution on X with no fixed points in X and $f(h(x)) = -f(x)$
 then $\sum_{x \in X} f(x) = 0$.

This is easy to show in a ring with characteristic not equal to 2, since
 then we can do

$$\sum_{x \in X} f(x) = \sum_{x \in X} f(h(x)) = - \sum_{x \in X} f(x)$$

and therefore $2 \sum_{x \in X} f(x) = 0$.

However, the following proof also works in rings of characteristic 2. The
 idea is to simply partition X into a disjoint union of doubleton sets of the
 form $\{x, h(x)\}$.

lemma *sum-involution-eq-0:*
assumes *$f \cdot h$: $\bigwedge x. x \in X \implies f(h x) + f x = 0$*
assumes *h : $\bigwedge x. x \in X \implies h x \in X \wedge x \in X \implies h(h x) = x \wedge x \in X \implies h x \neq x$*
shows $(\sum x \in X. f x) = 0$
proof (*cases finite X*)
assume *fin: finite X*
define R **where** $R = \{(x, y). x \in X \wedge y \in X \wedge (x = y \vee h x = y)\}$

```

have R: equiv X R
  unfolding equiv-def R-def using h(2,3)
  by (auto simp: refl-on-def sym-def trans-def)
define A where A = X // R
have partition: partition-on X A
  unfolding A-def using R by (rule partition-on-quotient)

have ( $\sum x \in X. f x$ ) = ( $\sum B \in A. \sum x \in B. f x$ )
  by (subst sum.partition[OF fin partition]) auto
also have ... = ( $\sum B \in A. 0$ )
proof (rule sum.cong)
  fix B assume B: B ∈ A
  then obtain x where x: x ∈ X and [simp]: B = R “ {x}
    using R by (metis A-def quotientE)
  have R “ {x} = {x, h x} h x ≠ x
    using h x(1) by (auto simp: R-def)
  thus ( $\sum x \in B. f x$ ) = 0
    using x f-h[of x] by (simp add: add.commute)
qed auto
finally show ?thesis
  by simp
qed auto

```

28.7 Refinement of partitions

definition *refines* :: 'a set \Rightarrow 'a set set \Rightarrow 'a set set \Rightarrow bool
 where *refines* A P Q \equiv
 $\text{partition-on } A \ P \wedge \text{partition-on } A \ Q \wedge (\forall X \in P. \exists Y \in Q. X \subseteq Y)$

lemma *refines-refl*: $\text{partition-on } A \ P \implies \text{refines } A \ P \ P$
 using *refines-def* by blast

lemma *refines-asym1*:
 assumes *refines* A P Q *refines* A Q P
 shows $P \subseteq Q$
proof
 fix X
 assume X ∈ P
 then obtain Y X' where Y ∈ Q X ⊆ Y X' ∈ P Y ⊆ X'
 by (meson assms *refines-def*)
 then have X' = X
 using *assms*(2) **unfolding** *partition-on-def* *refines-def*
 by (metis ⟨X ∈ P⟩ ⟨X ⊆ Y⟩ *disjnt-self-iff-empty disjnt-subset1 pairwiseD*)
 then show X ∈ Q
 using ⟨X ⊆ Y⟩ ⟨Y ∈ Q⟩ ⟨Y ⊆ X'⟩ by force
qed

lemma *refines-asym*: $\llbracket \text{refines } A \ P \ Q; \text{refines } A \ Q \ P \rrbracket \implies P=Q$
 by (meson *antisym-conv* *refines-asym1*)

lemma *refines-trans*: $\llbracket \text{refines } A \ P \ Q; \text{refines } A \ Q \ R \rrbracket \implies \text{refines } A \ P \ R$
by (*meson order.trans refines-def*)

lemma *refines-obtains-subset*:
assumes *refines* $A \ P \ Q$ $q \in Q$
shows *partition-on* $q \ \{p \in P. \ p \subseteq q\}$
proof –
have $p \subseteq q \vee \text{disjnt } p \ q$ **if** $p \in P$ **for** p
using *that assms unfolding refines-def partition-on-def disjoint-def*
by (*metis disjoint-def disjoint-subset1*)
with *assms* **have** $q \subseteq \text{Union } \{p \in P. \ p \subseteq q\}$
using *assms*
by (*clarsimp simp: refines-def disjoint-iff partition-on-def*) (*metis Union-iff*)
with *assms* **have** $q = \text{Union } \{p \in P. \ p \subseteq q\}$
by *auto*
then show *?thesis*
using *assms* **by** (*auto simp: refines-def disjoint-def partition-on-def*)
qed

28.8 The coarsest common refinement of a set of partitions

definition *common-refinement* :: 'a set set \Rightarrow 'a set set
where *common-refinement* $\mathcal{P} \equiv (\bigcup f \in (\Pi_E \ P \in \mathcal{P}. \ P). \ \{\bigcap (f \ ' \ \mathcal{P})\}) - \{\{\}\}$

With non-extensional function space

lemma *common-refinement*: *common-refinement* $\mathcal{P} = (\bigcup f \in (\Pi \ P \in \mathcal{P}. \ P). \ \{\bigcap (f \ ' \ \mathcal{P})\}) - \{\{\}\}$
(is *?lhs = ?rhs*)

proof
show *?rhs* \subseteq *?lhs*
apply (*clarsimp simp add: common-refinement-def PiE-def Ball-def*)
by (*metis restrict-Pi-cancel image-restrict-eq restrict-extensional*)
qed (*auto simp add: common-refinement-def PiE-def*)

lemma *common-refinement-exists*: $\llbracket X \in \text{common-refinement } \mathcal{P}; \ P \in \mathcal{P} \rrbracket \implies \exists R \in \mathcal{P}. \ X \subseteq R$
by (*auto simp add: common-refinement*)

lemma *Union-common-refinement*: $\bigcup (\text{common-refinement } \mathcal{P}) = (\bigcap \ P \in \mathcal{P}. \ \bigcup P)$

proof
show $(\bigcap \ P \in \mathcal{P}. \ \bigcup P) \subseteq \bigcup (\text{common-refinement } \mathcal{P})$
proof (*clarsimp simp: common-refinement*)
fix x
assume $\forall P \in \mathcal{P}. \ \exists X \in P. \ x \in X$
then obtain F **where** $F: \bigwedge P. \ P \in \mathcal{P} \implies F \ P \in P \wedge x \in F \ P$
by *metis*
then have $x \in \bigcap (F \ ' \ \mathcal{P})$
by *force*
with F **show** $\exists X \in (\bigcup x \in \Pi \ P \in \mathcal{P}. \ P. \ \{\bigcap (x \ ' \ \mathcal{P})\}) - \{\{\}\}. \ x \in X$

```

    by (auto simp add: Pi-iff Bex-def)
  qed
qed (auto simp: common-refinement-def)

lemma partition-on-common-refinement:
  assumes A:  $\bigwedge P. P \in \mathcal{P} \implies \text{partition-on } A \ P \text{ and } \mathcal{P} \neq \{\}$ 
  shows partition-on A (common-refinement  $\mathcal{P}$ )
proof (rule partition-onI)
  show  $\bigcup (\text{common-refinement } \mathcal{P}) = A$ 
    using assms by (simp add: partition-on-def Union-common-refinement)
  fix P Q
  assume P  $\in$  common-refinement  $\mathcal{P}$  and Q  $\in$  common-refinement  $\mathcal{P}$  and  $P \neq Q$ 
  then obtain f g where f:  $f \in (\Pi_E P \in \mathcal{P}. P)$  and P:  $P = \bigcap (f \text{ ' } \mathcal{P})$  and  $P \neq \{\}$ 
    and g:  $g \in (\Pi_E P \in \mathcal{P}. P)$  and Q:  $Q = \bigcap (g \text{ ' } \mathcal{P})$  and  $Q \neq \{\}$ 
    by (auto simp add: common-refinement-def)
  have f=g if  $x \in P$   $x \in Q$  for x
  proof (rule extensionalityI [of -  $\mathcal{P}$ ])
    fix R
    assume R  $\in \mathcal{P}$ 
    with that P Q f g A [unfolded partition-on-def, OF  $\langle R \in \mathcal{P} \rangle$ ]
    show f R = g R
      by (metis INT-E Int-iff PiE-iff disjointD emptyE)
  qed (use PiE-iff f g in auto)
  then show disjoint P Q
    by (metis P Q  $\langle P \neq Q \rangle$  disjoint-iff)
qed (simp add: common-refinement-def)

lemma refines-common-refinement:
  assumes  $\bigwedge P. P \in \mathcal{P} \implies \text{partition-on } A \ P \ P \in \mathcal{P}$ 
  shows refines A (common-refinement  $\mathcal{P}$ ) P
  unfolding refines-def
proof (intro conjI strip)
  fix X
  assume X  $\in$  common-refinement  $\mathcal{P}$ 
  with assms show  $\exists Y \in \mathcal{P}. X \subseteq Y$ 
    by (auto simp: common-refinement-def)
qed (use assms partition-on-common-refinement in auto)

The common refinement is itself refined by any other

lemma common-refinement-coarsest:
  assumes  $\bigwedge P. P \in \mathcal{P} \implies \text{partition-on } A \ P \text{ partition-on } A \ R \bigwedge P. P \in \mathcal{P} \implies \text{refines } A \ R \ P \ \mathcal{P} \neq \{\}$ 
  shows refines A R (common-refinement  $\mathcal{P}$ )
  unfolding refines-def
proof (intro conjI ballI partition-on-common-refinement)
  fix X
  assume X  $\in R$ 

```



```

have  $\exists p \in P. X \subseteq p$  if  $P \in \mathcal{P}$  for  $P$ 
  by (meson  $\langle X \in R \rangle$  assms( $\beta$ ) refines-def that)
then obtain  $F$  where  $f: \bigwedge P. P \in \mathcal{P} \implies F P \in P \wedge X \subseteq F P$ 
  by metis
with  $\langle \text{partition-on } A \ R \rangle \langle X \in R \rangle \langle \mathcal{P} \neq \{\} \rangle$ 
have  $\bigcap (F \text{ ` } \mathcal{P}) \in \text{common-refinement } \mathcal{P}$ 
  apply (simp add: partition-on-def common-refinement Pi-iff Bex-def)
  by (metis (no-types, lifting) cINF-greatest subset-empty)
with  $f$  show  $\exists Y \in \text{common-refinement } \mathcal{P}. X \subseteq Y$ 
  by (metis  $\langle \mathcal{P} \neq \{\} \rangle$  cINF-greatest)
qed (use assms in auto)

```

```

lemma finite-common-refinement:
  assumes finite  $\mathcal{P} \bigwedge P. P \in \mathcal{P} \implies \text{finite } P$ 
  shows finite (common-refinement  $\mathcal{P}$ )
proof –
  have finite ( $\prod_{E} P \in \mathcal{P}. P$ )
    by (simp add: assms finite-PiE)
  then show ?thesis
    by (auto simp: common-refinement-def)
qed

```

```

lemma card-common-refinement:
  assumes finite  $\mathcal{P} \bigwedge P. P \in \mathcal{P} \implies \text{finite } P$ 
  shows card (common-refinement  $\mathcal{P}$ )  $\leq (\prod P \in \mathcal{P}. \text{card } P)$ 
proof –
  have card (common-refinement  $\mathcal{P}$ )  $\leq \text{card } (\bigcup f \in (\prod_{E} P \in \mathcal{P}. P). \{\bigcap (f \text{ ` } \mathcal{P})\})$ 
    unfolding common-refinement-def by (meson card-Diff1-le)
  also have  $\dots \leq (\sum f \in (\prod_{E} P \in \mathcal{P}. P). \text{card}\{\bigcap (f \text{ ` } \mathcal{P})\})$ 
    by (metis assms finite-PiE card-UN-le)
  also have  $\dots = \text{card}(\prod_{E} P \in \mathcal{P}. P)$ 
    by simp
  also have  $\dots = (\prod P \in \mathcal{P}. \text{card } P)$ 
    by (simp add: assms(1) card-PiE dual-order.eq-iff)
  finally show ?thesis .
qed

```

end

29 Type of finite sets defined as a subtype of sets

```

theory FSet
imports Main Countable
begin

```

29.1 Definition of the type

```

typedef 'a fset = {A :: 'a set. finite A} morphisms fset Abs-fset
by auto

```

setup-lifting *type-definition-fset*

29.2 Basic operations and type class instantiations

instantiation *fset* :: (*finite*) *finite*

begin

instance by (*standard*; *transfer*; *simp*)

end

instantiation *fset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}

begin

lift-definition *bot-fset* :: 'a *fset* is {} **parametric** *empty-transfer* by *simp*

lift-definition *less-eq-fset* :: 'a *fset* \Rightarrow 'a *fset* \Rightarrow *bool* is *subset-eq* **parametric** *subset-transfer*

.

definition *less-fset* :: 'a *fset* \Rightarrow 'a *fset* \Rightarrow *bool* **where** $xs < ys \equiv xs \leq ys \wedge xs \neq (ys::'a \text{ fset})$

lemma *less-fset-transfer*[*transfer-rule*]:

includes *lifting-syntax*

assumes [*transfer-rule*]: *bi-unique A*

shows ((*pcr-fset A*) \implies (*pcr-fset A*) \implies (=)) (\subset) ($<$)

unfolding *less-fset-def*[*abs-def*] *psubset-eq*[*abs-def*] **by** *transfer-prover*

lift-definition *sup-fset* :: 'a *fset* \Rightarrow 'a *fset* \Rightarrow 'a *fset* is *union* **parametric** *union-transfer* by *simp*

lift-definition *inf-fset* :: 'a *fset* \Rightarrow 'a *fset* \Rightarrow 'a *fset* is *inter* **parametric** *inter-transfer*

by *simp*

lift-definition *minus-fset* :: 'a *fset* \Rightarrow 'a *fset* \Rightarrow 'a *fset* is *minus* **parametric** *Diff-transfer*

by *simp*

instance

by (*standard*; *transfer*; *auto*)+

end

abbreviation *fempty* :: 'a *fset* ($\langle \{\} \rangle$) **where** $\{\} \equiv \text{bot}$

abbreviation *fsubset-eq* :: 'a *fset* \Rightarrow 'a *fset* \Rightarrow *bool* (**infix** $\langle |\subseteq| \rangle$ 50) **where** $xs |\subseteq| ys \equiv xs \leq ys$

abbreviation *fsubset* :: 'a *fset* \Rightarrow 'a *fset* \Rightarrow *bool* (**infix** $\langle |\subset| \rangle$ 50) **where** $xs |\subset| ys$

$\equiv xs < ys$

abbreviation *funion* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset (**infixl** $\langle |\cup| \rangle$ 65) **where** $xs |\cup|$

$ys \equiv \text{sup } xs \text{ } ys$

abbreviation *finter* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset (**infixl** $\langle |\cap| \rangle$ 65) **where** $xs |\cap|$

$ys \equiv \text{inf } xs \text{ } ys$

abbreviation *fminus* :: 'a fset \Rightarrow 'a fset \Rightarrow 'a fset (**infixl** $\langle |-| \rangle$ 65) **where** $xs |-|$

$ys \equiv \text{minus } xs \text{ } ys$

instantiation *fset* :: (equal) equal

begin

definition *HOL.equal* $A \ B \longleftrightarrow A \sqsubseteq B \wedge B \sqsubseteq A$

instance by *intro-classes* (auto simp add: equal-fset-def)

end

instantiation *fset* :: (type) conditionally-complete-lattice

begin

context includes *lifting-syntax*

begin

lemma *right-total-Inf-fset-transfer*:

assumes [*transfer-rule*]: *bi-unique* A **and** [*transfer-rule*]: *right-total* A

shows (*rel-set* (*rel-set* A) $====>$ *rel-set* A)

($\lambda S.$ if finite ($\bigcap S \cap \text{Collect } (\text{Domainp } A)$) then $\bigcap S \cap \text{Collect } (\text{Domainp } A)$
else {})

($\lambda S.$ if finite (*Inf* S) then *Inf* S else {})

by *transfer-prover*

lemma *Inf-fset-transfer*:

assumes [*transfer-rule*]: *bi-unique* A **and** [*transfer-rule*]: *bi-total* A

shows (*rel-set* (*rel-set* A) $====>$ *rel-set* A) ($\lambda A.$ if finite (*Inf* A) then *Inf* A else {})

($\lambda A.$ if finite (*Inf* A) then *Inf* A else {})

by *transfer-prover*

lift-definition *Inf-fset* :: 'a fset set \Rightarrow 'a fset **is** $\lambda A.$ if finite (*Inf* A) then *Inf* A
else {}

parametric *right-total-Inf-fset-transfer* *Inf-fset-transfer* **by** *simp*

lemma *Sup-fset-transfer*:

assumes [*transfer-rule*]: *bi-unique* A

shows (*rel-set* (*rel-set* A) $====>$ *rel-set* A) ($\lambda A.$ if finite (*Sup* A) then *Sup* A
else {})

($\lambda A.$ if finite (*Sup* A) then *Sup* A else {}) **by** *transfer-prover*

lift-definition *Sup-fset* :: 'a fset set \Rightarrow 'a fset **is** $\lambda A.$ if finite (*Sup* A) then *Sup* A
else {}

parametric *Sup-fset-transfer* **by** *simp*

lemma *finite-Sup*: $\exists z. \text{finite } z \wedge (\forall a. a \in X \longrightarrow a \leq z) \implies \text{finite } (\text{Sup } X)$
by (*auto intro: finite-subset*)

lemma *transfer-bdd-below*[*transfer-rule*]: $(\text{rel-set } (\text{pcr-fset } (=)) \implies (=)) \text{ bdd-below}$
bdd-below
by *auto*

end

instance

proof

fix $x\ z :: 'a \text{ fset}$
fix $X :: 'a \text{ fset set}$
{
assume $x \in X \text{ bdd-below } X$
then show $\text{Inf } X \sqsubseteq x$ **by** *transfer auto*
next
assume $X \neq \{\}$ $(\bigwedge x. x \in X \implies z \sqsubseteq x)$
then show $z \sqsubseteq \text{Inf } X$ **by** *transfer (clarsimp, blast)*
next
assume $x \in X \text{ bdd-above } X$
then obtain z **where** $x \in X (\bigwedge x. x \in X \implies x \sqsubseteq z)$
by (*auto simp: bdd-above-def*)
then show $x \sqsubseteq \text{Sup } X$
by *transfer (auto intro!: finite-Sup)*
next
assume $X \neq \{\}$ $(\bigwedge x. x \in X \implies x \sqsubseteq z)$
then show $\text{Sup } X \sqsubseteq z$ **by** *transfer (clarsimp, blast)*
}
qed
end

instantiation *fset* :: (*finite*) *complete-lattice*
begin

lift-definition *top-fset* :: '*a* *fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer*
UNIV-transfer
by *simp*

instance

by (*standard; transfer; auto*)

end

instantiation *fset* :: (*finite*) *complete-boolean-algebra*
begin

lift-definition *uminus-fset* :: '*a* *fset* \Rightarrow '*a* *fset* **is** *uminus*
parametric *right-total-Compl-transfer Compl-transfer* **by** *simp*

instance

by (*standard*; *transfer*) (*simp-all add: Inf-Sup Diff-eq*)
end

abbreviation *fUNIV* :: 'a::finite fset **where** *fUNIV* \equiv *top*

abbreviation *fminus* :: 'a::finite fset \Rightarrow 'a fset ($\langle | - | \rightarrow [81] 80 \rangle$) **where** $| - | \ x \equiv$
uminus x

declare *top-fset.rep-eq*[*simp*]

29.3 Other operations

lift-definition *finset* :: 'a \Rightarrow 'a fset \Rightarrow 'a fset **is insert** **parametric** *Lifting-Set.insert-transfer*
by *simp*

syntax

-fset :: args \Rightarrow 'a fset ($\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix finite set enumeration} \rangle \{ | - | \} \rangle \rangle$)

syntax-consts

-fset \Rightarrow *finset*

translations

$\{ | x, xs | \} == \text{CONST } finset \ x \ \{ | xs | \}$

$\{ | x | \} == \text{CONST } finset \ x \ \{ | | \}$

abbreviation *fmember* :: 'a \Rightarrow 'a fset \Rightarrow bool (**infix** $\langle | \in | \rangle 50$) **where**
 $x \ | \in \ X \equiv x \in fset \ X$

abbreviation *not-fmember* :: 'a \Rightarrow 'a fset \Rightarrow bool (**infix** $\langle | \notin | \rangle 50$) **where**
 $x \ | \notin \ X \equiv x \notin fset \ X$

context

begin

qualified abbreviation *Ball* :: 'a fset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **where**
Ball *X* $\equiv \text{Set.Ball } (fset \ X)$

alias *fBall* = *FSet.Ball*

qualified abbreviation *Bex* :: 'a fset \Rightarrow ('a \Rightarrow bool) \Rightarrow bool **where**
Bex *X* $\equiv \text{Set.Bex } (fset \ X)$

alias *fBex* = *FSet.Bex*

end

syntax (*input*)

-fBall :: pttm \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite !} \rangle \rangle$
 $(- / | : | -) . / - \rangle [0, 0, 10] 10$)

-fBex :: pttm \Rightarrow 'a fset \Rightarrow bool \Rightarrow bool ($\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder finite$

$\text{?} \gg \text{?} \langle -/|:- \rangle ./ - \rangle [0, 0, 10] 10)$
 $\text{-fBex1} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \text{ ?} \rangle \rangle \text{?} \langle -/:- \rangle ./ - \rangle [0, 0, 10] 10)$

syntax

$\text{-fBall} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \forall \rangle \rangle \forall \langle -/|\in|- \rangle ./ - \rangle [0, 0, 10] 10)$
 $\text{-fBex} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \exists \rangle \rangle \exists \langle -/|\in|- \rangle ./ - \rangle [0, 0, 10] 10)$
 $\text{-fBnex} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \# \rangle \rangle \# \langle -/|\in|- \rangle ./ - \rangle [0, 0, 10] 10)$
 $\text{-fBex1} :: \text{pttrn} \Rightarrow 'a \text{ fset} \Rightarrow \text{bool} \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \exists ! \rangle \rangle \exists ! \langle -/|\in|- \rangle ./ - \rangle [0, 0, 10] 10)$

syntax-consts

$\text{-fBall} \text{-fBnex} \Rightarrow \text{fBall} \text{ and}$
 $\text{-fBex} \Rightarrow \text{fBex} \text{ and}$
 $\text{-fBex1} \Rightarrow \text{Ex1}$

translations

$\forall x | \in | A. P \Rightarrow \text{CONST FSet.Ball } A (\lambda x. P)$
 $\exists x | \in | A. P \Rightarrow \text{CONST FSet.Bex } A (\lambda x. P)$
 $\# x | \in | A. P \Rightarrow \text{CONST fBall } A (\lambda x. \neg P)$
 $\exists ! x | \in | A. P \Rightarrow \exists ! x. x | \in | A \wedge P$

typed-print-translation \langle

$[(\text{const-syntax} \langle \text{fBall} \rangle, \text{Syntax-Trans.preserve-binder-abs2-tr}' \text{syntax-const} \langle \text{-fBall} \rangle),$
 $(\text{const-syntax} \langle \text{fBex} \rangle, \text{Syntax-Trans.preserve-binder-abs2-tr}' \text{syntax-const} \langle \text{-fBex} \rangle)]$
 \rangle — to avoid eta-contraction of body

syntax

$\text{-setlessfAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \forall \rangle \rangle \forall \langle -|\subset|- \rangle ./ - \rangle [0, 0, 10] 10)$
 $\text{-setlessfEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \exists \rangle \rangle \exists \langle -|\subset|- \rangle ./ - \rangle [0, 0, 10] 10)$
 $\text{-setlefAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \forall \rangle \rangle \forall \langle -|\subseteq|- \rangle ./ - \rangle [0, 0, 10] 10)$
 $\text{-setlefEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder finite} \exists \rangle \rangle \exists \langle -|\subseteq|- \rangle ./ - \rangle [0, 0, 10] 10)$

syntax-consts

$\text{-setlessfAll} \text{-setlefAll} \Rightarrow \text{All} \text{ and}$
 $\text{-setlessfEx} \text{-setlefEx} \Rightarrow \text{Ex}$

translations

$\forall A | \subset | B. P \Rightarrow \forall A. A | \subset | B \longrightarrow P$
 $\exists A | \subset | B. P \Rightarrow \exists A. A | \subset | B \wedge P$
 $\forall A | \subseteq | B. P \Rightarrow \forall A. A | \subseteq | B \longrightarrow P$
 $\exists A | \subseteq | B. P \Rightarrow \exists A. A | \subseteq | B \wedge P$

context includes *lifting-syntax*
begin

lemma *fmember-transfer0*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ($A \implies \text{pcr-fset } A \implies (=)$) (\in) ($|\in|$)
by *transfer-prover*

lemma *fBall-transfer0*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ($\text{pcr-fset } A \implies (A \implies (=)) \implies (=)$) (*Ball*) (*fBall*)
by *transfer-prover*

lemma *fBex-transfer0*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ($\text{pcr-fset } A \implies (A \implies (=)) \implies (=)$) (*Bex*) (*fBex*)
by *transfer-prover*

lift-definition *ffilter* :: ($'a \Rightarrow \text{bool}$) \Rightarrow $'a \text{ fset} \Rightarrow 'a \text{ fset}$ **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* **by** *simp*

lift-definition *fPow* :: $'a \text{ fset} \Rightarrow 'a \text{ fset fset}$ **is** *Pow* **parametric** *Pow-transfer*
by (*simp add: finite-subset*)

lift-definition *fcard* :: $'a \text{ fset} \Rightarrow \text{nat}$ **is** *card* **parametric** *card-transfer* .

lift-definition *fimage* :: ($'a \Rightarrow 'b$) \Rightarrow $'a \text{ fset} \Rightarrow 'b \text{ fset}$ (**infixr** $\langle | \rangle$ 90) **is** *image*
parametric *image-transfer* **by** *simp*

lift-definition *fthe-elem* :: $'a \text{ fset} \Rightarrow 'a$ **is** *the-elem* .

lift-definition *fbind* :: $'a \text{ fset} \Rightarrow ('a \Rightarrow 'b \text{ fset}) \Rightarrow 'b \text{ fset}$ **is** *Set.bind* **parametric**
bind-transfer
by (*simp add: Set.bind-def*)

lift-definition *ffUnion* :: $'a \text{ fset fset} \Rightarrow 'a \text{ fset}$ **is** *Union* **parametric** *Union-transfer*
by *simp*

lift-definition *ffold* :: ($'a \Rightarrow 'b \Rightarrow 'b$) \Rightarrow $'b \Rightarrow 'a \text{ fset} \Rightarrow 'b$ **is** *Finite-Set.fold* .

lift-definition *fset-of-list* :: $'a \text{ list} \Rightarrow 'a \text{ fset}$ **is** *set* **by** (*rule finite-set*)

lift-definition *sorted-list-of-fset* :: $'a::\text{linorder} \text{ fset} \Rightarrow 'a \text{ list}$ **is** *sorted-list-of-set* .

29.4 Transferred lemmas from Set.thy

lemma *fset-eqI*: $(\bigwedge x. (x \in A) = (x \in B)) \implies A = B$

by (*rule set-eqI*[*Transfer.transferred*])

lemma *fset-eq-iff*[*no-atp*]: $(A = B) = (\forall x. (x \in A) = (x \in B))$
by (*rule set-eq-iff*[*Transfer.transferred*])

lemma *fBall*[*no-atp*]: $(\bigwedge x. x \in A \implies P x) \implies fBall A P$
by (*rule ballI*[*Transfer.transferred*])

lemma *fbspec*[*no-atp*]: $fBall A P \implies x \in A \implies P x$
by (*rule bspec*[*Transfer.transferred*])

lemma *fBallE*[*no-atp*]: $fBall A P \implies (P x \implies Q) \implies (x \notin A \implies Q) \implies Q$
by (*rule ballE*[*Transfer.transferred*])

lemma *fBexI*[*no-atp*]: $P x \implies x \in A \implies fBex A P$
by (*rule bexI*[*Transfer.transferred*])

lemma *rev-fBexI*[*no-atp*]: $x \in A \implies P x \implies fBex A P$
by (*rule rev-bexI*[*Transfer.transferred*])

lemma *fBexCI*[*no-atp*]: $(fBall A (\lambda x. \neg P x) \implies P a) \implies a \in A \implies fBex A P$
by (*rule bexCI*[*Transfer.transferred*])

lemma *fBexE*[*no-atp*]: $fBex A P \implies (\bigwedge x. x \in A \implies P x \implies Q) \implies Q$
by (*rule bexE*[*Transfer.transferred*])

lemma *fBall-triv*[*no-atp*]: $fBall A (\lambda x. P) = ((\exists x. x \in A) \longrightarrow P)$
by (*rule ball-triv*[*Transfer.transferred*])

lemma *fBex-triv*[*no-atp*]: $fBex A (\lambda x. P) = ((\exists x. x \in A) \wedge P)$
by (*rule bex-triv*[*Transfer.transferred*])

lemma *fBex-triv-one-point1*[*no-atp*]: $fBex A (\lambda x. x = a) = (a \in A)$
by (*rule bex-triv-one-point1*[*Transfer.transferred*])

lemma *fBex-triv-one-point2*[*no-atp*]: $fBex A ((=) a) = (a \in A)$
by (*rule bex-triv-one-point2*[*Transfer.transferred*])

lemma *fBex-one-point1*[*no-atp*]: $fBex A (\lambda x. x = a \wedge P x) = (a \in A \wedge P a)$
by (*rule bex-one-point1*[*Transfer.transferred*])

lemma *fBex-one-point2*[*no-atp*]: $fBex A (\lambda x. a = x \wedge P x) = (a \in A \wedge P a)$
by (*rule bex-one-point2*[*Transfer.transferred*])

lemma *fBall-one-point1*[*no-atp*]: $fBall A (\lambda x. x = a \longrightarrow P x) = (a \in A \longrightarrow P a)$
by (*rule ball-one-point1*[*Transfer.transferred*])

lemma *fBall-one-point2*[*no-atp*]: $fBall A (\lambda x. a = x \longrightarrow P x) = (a \in A \longrightarrow P a)$

a)

by (*rule ball-one-point2*[*Transfer.transferred*])**lemma** *fBall-conj-distrib*: $fBall\ A\ (\lambda x. P\ x \wedge Q\ x) = (fBall\ A\ P \wedge fBall\ A\ Q)$ **by** (*rule ball-conj-distrib*[*Transfer.transferred*])**lemma** *fBex-disj-distrib*: $fBex\ A\ (\lambda x. P\ x \vee Q\ x) = (fBex\ A\ P \vee fBex\ A\ Q)$ **by** (*rule bex-disj-distrib*[*Transfer.transferred*])**lemma** *fBall-cong*[*fundef-cong*]: $A = B \implies (\bigwedge x. x \models B \implies P\ x = Q\ x) \implies fBall\ A\ P = fBall\ B\ Q$ **by** (*rule ball-cong*[*Transfer.transferred*])**lemma** *fBex-cong*[*fundef-cong*]: $A = B \implies (\bigwedge x. x \models B \implies P\ x = Q\ x) \implies fBex\ A\ P = fBex\ B\ Q$ **by** (*rule bex-cong*[*Transfer.transferred*])**lemma** *fsubsetI*[*intro!*]: $(\bigwedge x. x \models A \implies x \models B) \implies A \sqsubseteq B$ **by** (*rule subsetI*[*Transfer.transferred*])**lemma** *fsubsetD*[*elim, intro?*]: $A \sqsubseteq B \implies c \models A \implies c \models B$ **by** (*rule subsetD*[*Transfer.transferred*])**lemma** *rev-fsubsetD*[*no-atp, intro?*]: $c \models A \implies A \sqsubseteq B \implies c \models B$ **by** (*rule rev-subsetD*[*Transfer.transferred*])**lemma** *fsubsetCE*[*no-atp, elim*]: $A \sqsubseteq B \implies (c \not\models A \implies P) \implies (c \models B \implies P) \implies P$ **by** (*rule subsetCE*[*Transfer.transferred*])**lemma** *fsubset-eq*[*no-atp*]: $(A \sqsubseteq B) = fBall\ A\ (\lambda x. x \models B)$ **by** (*rule subset-eq*[*Transfer.transferred*])**lemma** *contra-fsubsetD*[*no-atp*]: $A \sqsubseteq B \implies c \not\models B \implies c \not\models A$ **by** (*rule contra-subsetD*[*Transfer.transferred*])**lemma** *fsubset-refl*: $A \sqsubseteq A$ **by** (*rule subset-refl*[*Transfer.transferred*])**lemma** *fsubset-trans*: $A \sqsubseteq B \implies B \sqsubseteq C \implies A \sqsubseteq C$ **by** (*rule subset-trans*[*Transfer.transferred*])**lemma** *fset-rev-mp*: $c \models A \implies A \sqsubseteq B \implies c \models B$ **by** (*rule rev-subsetD*[*Transfer.transferred*])**lemma** *fset-mp*: $A \sqsubseteq B \implies c \models A \implies c \models B$ **by** (*rule subsetD*[*Transfer.transferred*])**lemma** *fsubset-not-fsubset-eq*[*code*]: $(A \sqsubset B) = (A \sqsubseteq B \wedge \neg B \sqsubseteq A)$

by (rule subset-not-subset-eq[Transfer.transferred])

lemma eq-fmem-trans: $a = b \implies b \in A \implies a \in A$
 by (rule eq-mem-trans[Transfer.transferred])

lemma fsubset-antisym[intro!]: $A \subseteq B \implies B \subseteq A \implies A = B$
 by (rule subset-antisym[Transfer.transferred])

lemma fequalityD1: $A = B \implies A \subseteq B$
 by (rule equalityD1[Transfer.transferred])

lemma fequalityD2: $A = B \implies B \subseteq A$
 by (rule equalityD2[Transfer.transferred])

lemma fequalityE: $A = B \implies (A \subseteq B \implies B \subseteq A \implies P) \implies P$
 by (rule equalityE[Transfer.transferred])

lemma fequalityCE[elim]:
 $A = B \implies (c \in A \implies c \in B \implies P) \implies (c \notin A \implies c \notin B \implies P) \implies P$
 by (rule equalityCE[Transfer.transferred])

lemma eqfset-imp-iff: $A = B \implies (x \in A) = (x \in B)$
 by (rule eqfset-imp-iff[Transfer.transferred])

lemma eqfelem-imp-iff: $x = y \implies (x \in A) = (y \in A)$
 by (rule eqfelem-imp-iff[Transfer.transferred])

lemma fempty-iff[simp]: $(c \in \{\}) = \text{False}$
 by (rule empty-iff[Transfer.transferred])

lemma fempty-fsubsetI[iff]: $\{\} \subseteq x$
 by (rule empty-subsetI[Transfer.transferred])

lemma equalsffemptyI: $(\bigwedge y. y \in A \implies \text{False}) \implies A = \{\}$
 by (rule equals0I[Transfer.transferred])

lemma equalsffemptyD: $A = \{\} \implies a \notin A$
 by (rule equals0D[Transfer.transferred])

lemma fBall-fempty[simp]: $fBall \{\} P = \text{True}$
 by (rule ball-empty[Transfer.transferred])

lemma fBex-fempty[simp]: $fBex \{\} P = \text{False}$
 by (rule bex-empty[Transfer.transferred])

lemma fPow-iff[iff]: $(A \in fPow B) = (A \subseteq B)$
 by (rule Pow-iff[Transfer.transferred])

lemma fPowI: $A \subseteq B \implies A \in fPow B$

by (rule *PowI*[*Transfer.transferred*])

lemma *fPowD*: $A \in fPow\ B \implies A \subseteq B$
by (rule *PowD*[*Transfer.transferred*])

lemma *fPow-bottom*: $\{\mid\} \in fPow\ B$
by (rule *Pow-bottom*[*Transfer.transferred*])

lemma *fPow-top*: $A \in fPow\ A$
by (rule *Pow-top*[*Transfer.transferred*])

lemma *fPow-not-empty*: $fPow\ A \neq \{\mid\}$
by (rule *Pow-not-empty*[*Transfer.transferred*])

lemma *finter-iff[simp]*: $(c \in A \mid \cap B) = (c \in A \wedge c \in B)$
by (rule *Int-iff*[*Transfer.transferred*])

lemma *finterI[intro!]*: $c \in A \implies c \in B \implies c \in A \mid \cap B$
by (rule *IntI*[*Transfer.transferred*])

lemma *finterD1*: $c \in A \mid \cap B \implies c \in A$
by (rule *IntD1*[*Transfer.transferred*])

lemma *finterD2*: $c \in A \mid \cap B \implies c \in B$
by (rule *IntD2*[*Transfer.transferred*])

lemma *finterE[elim!]*: $c \in A \mid \cap B \implies (c \in A \implies c \in B \implies P) \implies P$
by (rule *IntE*[*Transfer.transferred*])

lemma *funion-iff[simp]*: $(c \in A \mid \cup B) = (c \in A \vee c \in B)$
by (rule *Un-iff*[*Transfer.transferred*])

lemma *funionI1[elim?]*: $c \in A \implies c \in A \mid \cup B$
by (rule *UnI1*[*Transfer.transferred*])

lemma *funionI2[elim?]*: $c \in B \implies c \in A \mid \cup B$
by (rule *UnI2*[*Transfer.transferred*])

lemma *funionCI[intro!]*: $(c \notin B \implies c \in A) \implies c \in A \mid \cup B$
by (rule *UnCI*[*Transfer.transferred*])

lemma *funionE[elim!]*: $c \in A \mid \cup B \implies (c \in A \implies P) \implies (c \in B \implies P) \implies P$
by (rule *UnE*[*Transfer.transferred*])

lemma *fminus-iff[simp]*: $(c \in A \mid - B) = (c \in A \wedge c \notin B)$
by (rule *Diff-iff*[*Transfer.transferred*])

lemma *fminusI[intro!]*: $c \in A \implies c \notin B \implies c \in A \mid - B$

by (rule *DiffI*[*Transfer.transferred*])

lemma *fminusD1*: $c \mid\in\mid A \mid-\mid B \implies c \mid\in\mid A$
by (rule *DiffD1*[*Transfer.transferred*])

lemma *fminusD2*: $c \mid\in\mid A \mid-\mid B \implies c \mid\in\mid B \implies P$
by (rule *DiffD2*[*Transfer.transferred*])

lemma *fminusE*[*elim!*]: $c \mid\in\mid A \mid-\mid B \implies (c \mid\in\mid A \implies c \not\mid\in\mid B \implies P) \implies P$
by (rule *DiffE*[*Transfer.transferred*])

lemma *finert-iff*[*simp*]: $(a \mid\in\mid \text{finert } b \ A) = (a = b \vee a \mid\in\mid A)$
by (rule *insert-iff*[*Transfer.transferred*])

lemma *finertI1*: $a \mid\in\mid \text{finert } a \ B$
by (rule *insertI1*[*Transfer.transferred*])

lemma *finertI2*: $a \mid\in\mid B \implies a \mid\in\mid \text{finert } b \ B$
by (rule *insertI2*[*Transfer.transferred*])

lemma *finertE*[*elim!*]: $a \mid\in\mid \text{finert } b \ A \implies (a = b \implies P) \implies (a \mid\in\mid A \implies P) \implies P$
by (rule *insertE*[*Transfer.transferred*])

lemma *finertCI*[*intro!*]: $(a \not\mid\in\mid B \implies a = b) \implies a \mid\in\mid \text{finert } b \ B$
by (rule *insertCI*[*Transfer.transferred*])

lemma *fsubset-finert-iff*:
 $(A \mid\subseteq\mid \text{finert } x \ B) = (\text{if } x \mid\in\mid A \text{ then } A \mid-\mid \{x\} \mid\subseteq\mid B \text{ else } A \mid\subseteq\mid B)$
by (rule *subset-insert-iff*[*Transfer.transferred*])

lemma *finert-ident*: $x \not\mid\in\mid A \implies x \not\mid\in\mid B \implies (\text{finert } x \ A = \text{finert } x \ B) = (A = B)$
by (rule *insert-ident*[*Transfer.transferred*])

lemma *fsingletonI*[*intro!,no-atp*]: $a \mid\in\mid \{|a|\}$
by (rule *singletonI*[*Transfer.transferred*])

lemma *fsingletonD*[*dest!,no-atp*]: $b \mid\in\mid \{|a|\} \implies b = a$
by (rule *singletonD*[*Transfer.transferred*])

lemma *fsingleton-iff*: $(b \mid\in\mid \{|a|\}) = (b = a)$
by (rule *singleton-iff*[*Transfer.transferred*])

lemma *fsingleton-inject*[*dest!*]: $\{|a|\} = \{|b|\} \implies a = b$
by (rule *singleton-inject*[*Transfer.transferred*])

lemma *fsingleton-finert-inj-eq*[*iff,no-atp*]: $(\{|b|\} = \text{finert } a \ A) = (a = b \wedge A \mid\subseteq\mid \{|b|\})$

by (rule *singleton-insert-inj-eq*[*Transfer.transferred*])

lemma *fsingleton-finsert-inj-eq'*[*iff,no-atp*]: $(finsert\ a\ A = \{|b|\}) = (a = b \wedge A \subseteq \{|b|\})$

by (rule *singleton-insert-inj-eq'*[*Transfer.transferred*])

lemma *fsubset-fsingletonD*: $A \subseteq \{|x|\} \implies A = \{|\}\vee A = \{|x|\}$

by (rule *subset-singletonD*[*Transfer.transferred*])

lemma *fminus-single-finsert*: $A \dashv \{|x|\} \subseteq B \implies A \subseteq finsert\ x\ B$

by (rule *Diff-single-insert*[*Transfer.transferred*])

lemma *fdoubleton-eq-iff*: $(\{|a, b|\} = \{|c, d|\}) = (a = c \wedge b = d \vee a = d \wedge b = c)$

by (rule *doubleton-eq-iff*[*Transfer.transferred*])

lemma *funion-fsingleton-iff*:

$(A \cup B = \{|x|\}) = (A = \{|\} \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{|\} \vee A = \{|x|\} \wedge B = \{|x|\})$

by (rule *Un-singleton-iff*[*Transfer.transferred*])

lemma *fsingleton-funion-iff*:

$(\{|x|\} = A \cup B) = (A = \{|\} \wedge B = \{|x|\} \vee A = \{|x|\} \wedge B = \{|\} \vee A = \{|x|\} \wedge B = \{|x|\})$

by (rule *singleton-Un-iff*[*Transfer.transferred*])

lemma *fimage-eqI*[*simp, intro*]: $b = f\ x \implies x \in A \implies b \in f \mid A$

by (rule *image-eqI*[*Transfer.transferred*])

lemma *fimageI*: $x \in A \implies f\ x \in f \mid A$

by (rule *imageI*[*Transfer.transferred*])

lemma *rev-fimage-eqI*: $x \in A \implies b = f\ x \implies b \in f \mid A$

by (rule *rev-image-eqI*[*Transfer.transferred*])

lemma *fimageE*[*elim!*]: $b \in f \mid A \implies (\bigwedge x. b = f\ x \implies x \in A \implies thesis) \implies thesis$

by (rule *imageE*[*Transfer.transferred*])

lemma *Compr-fimage-eq*: $\{x. x \in f \mid A \wedge P\ x\} = f \mid \{x. x \in A \wedge P\ (f\ x)\}$

by (rule *Compr-image-eq*[*Transfer.transferred*])

lemma *fimage-funion*: $f \mid (A \cup B) = f \mid A \cup f \mid B$

by (rule *image-Un*[*Transfer.transferred*])

lemma *fimage-iff*: $(z \in f \mid A) = fBex\ A\ (\lambda x. z = f\ x)$

by (rule *image-iff*[*Transfer.transferred*])

lemma *fimage-fsubset-iff*[*no-atp*]: $(f \mid A \subseteq B) = fBall\ A\ (\lambda x. f\ x \in B)$

by (rule image-subset-iff[Transfer.transferred])

lemma *fimage-fsubsetI*: $(\bigwedge x. x \in A \implies f x \in B) \implies f \mid A \subseteq B$
 by (rule image-subsetI[Transfer.transferred])

lemma *fimage-ident[simp]*: $(\lambda x. x) \mid Y = Y$
 by (rule image-ident[Transfer.transferred])

lemma *if-split-fmem1*: $((\text{if } Q \text{ then } x \text{ else } y) \in b) = ((Q \longrightarrow x \in b) \wedge (\neg Q \longrightarrow y \in b))$
 by (rule if-split-mem1[Transfer.transferred])

lemma *if-split-fmem2*: $(a \in (\text{if } Q \text{ then } x \text{ else } y)) = ((Q \longrightarrow a \in x) \wedge (\neg Q \longrightarrow a \in y))$
 by (rule if-split-mem2[Transfer.transferred])

lemma *pfssubsetI[intro!,no-atp]*: $A \subseteq B \implies A \neq B \implies A \subset B$
 by (rule psubsetI[Transfer.transferred])

lemma *pfssubsetE[elim!,no-atp]*: $A \subset B \implies (A \subseteq B \implies \neg B \subseteq A \implies R) \implies R$
 by (rule psubsetE[Transfer.transferred])

lemma *pfssubset-finsert-iff*:
 $(A \subset \text{finsert } x B) =$
 $(\text{if } x \in B \text{ then } A \subset B \text{ else if } x \in A \text{ then } A \mid \{x\} \subset B \text{ else } A \subseteq B)$
 by (rule psubset-insert-iff[Transfer.transferred])

lemma *pfssubset-eq*: $(A \subset B) = (A \subseteq B \wedge A \neq B)$
 by (rule psubset-eq[Transfer.transferred])

lemma *pfssubset-imp-fsubset*: $A \subset B \implies A \subseteq B$
 by (rule psubset-imp-subset[Transfer.transferred])

lemma *pfssubset-trans*: $A \subset B \implies B \subset C \implies A \subset C$
 by (rule psubset-trans[Transfer.transferred])

lemma *pfssubsetD*: $A \subset B \implies c \in A \implies c \in B$
 by (rule psubsetD[Transfer.transferred])

lemma *pfssubset-fsubset-trans*: $A \subset B \implies B \subseteq C \implies A \subset C$
 by (rule psubset-subset-trans[Transfer.transferred])

lemma *fsubset-pfssubset-trans*: $A \subseteq B \implies B \subset C \implies A \subset C$
 by (rule subset-psubset-trans[Transfer.transferred])

lemma *pfssubset-imp-ex-fmem*: $A \subset B \implies \exists b. b \in B \mid A$
 by (rule psubset-imp-ex-mem[Transfer.transferred])

lemma *fimage-fPow-mono*: $f \mid^{\dagger} A \mid \subseteq B \implies (\mid^{\dagger}) f \mid^{\dagger} fPow A \mid \subseteq fPow B$
by (rule *image-Pow-mono*[*Transfer.transferred*])

lemma *fimage-fPow-surj*: $f \mid^{\dagger} A = B \implies (\mid^{\dagger}) f \mid^{\dagger} fPow A = fPow B$
by (rule *image-Pow-surj*[*Transfer.transferred*])

lemma *fsubset-finertI*: $B \mid \subseteq finert a B$
by (rule *subset-inertI*[*Transfer.transferred*])

lemma *fsubset-finertI2*: $A \mid \subseteq B \implies A \mid \subseteq finert b B$
by (rule *subset-inertI2*[*Transfer.transferred*])

lemma *fsubset-finert*: $x \notin A \implies (A \mid \subseteq finert x B) = (A \mid \subseteq B)$
by (rule *subset-inert*[*Transfer.transferred*])

lemma *funion-upper1*: $A \mid \subseteq A \mid \cup B$
by (rule *Un-upper1*[*Transfer.transferred*])

lemma *funion-upper2*: $B \mid \subseteq A \mid \cup B$
by (rule *Un-upper2*[*Transfer.transferred*])

lemma *funion-least*: $A \mid \subseteq C \implies B \mid \subseteq C \implies A \mid \cup B \mid \subseteq C$
by (rule *Un-least*[*Transfer.transferred*])

lemma *finter-lower1*: $A \mid \cap B \mid \subseteq A$
by (rule *Int-lower1*[*Transfer.transferred*])

lemma *finter-lower2*: $A \mid \cap B \mid \subseteq B$
by (rule *Int-lower2*[*Transfer.transferred*])

lemma *finter-greatest*: $C \mid \subseteq A \implies C \mid \subseteq B \implies C \mid \subseteq A \mid \cap B$
by (rule *Int-greatest*[*Transfer.transferred*])

lemma *fminus-fsubset*: $A \mid - B \mid \subseteq A$
by (rule *Diff-subset*[*Transfer.transferred*])

lemma *fminus-fsubset-conv*: $(A \mid - B \mid \subseteq C) = (A \mid \subseteq B \mid \cup C)$
by (rule *Diff-subset-conv*[*Transfer.transferred*])

lemma *fsubset-fempty[simp]*: $(A \mid \subseteq \{\mid\}) = (A = \{\mid\})$
by (rule *subset-empty*[*Transfer.transferred*])

lemma *not-pfsubset-fempty[iff]*: $\neg A \mid \subset \{\mid\}$
by (rule *not-psubset-empty*[*Transfer.transferred*])

lemma *finert-is-funion*: $finert a A = \{\mid a\} \mid \cup A$
by (rule *insert-is-Un*[*Transfer.transferred*])

lemma *finert-not-fempty[simp]*: $finert a A \neq \{\mid\}$

by (rule insert-not-empty[Transfer.transferred])

lemma fempty-not-finsert: $\{\|\} \neq \text{finsert } a \ A$
 by (rule empty-not-insert[Transfer.transferred])

lemma finsert-absorb: $a \in A \implies \text{finsert } a \ A = A$
 by (rule insert-absorb[Transfer.transferred])

lemma finsert-absorb2[simp]: $\text{finsert } x \ (\text{finsert } x \ A) = \text{finsert } x \ A$
 by (rule insert-absorb2[Transfer.transferred])

lemma finsert-commute: $\text{finsert } x \ (\text{finsert } y \ A) = \text{finsert } y \ (\text{finsert } x \ A)$
 by (rule insert-commute[Transfer.transferred])

lemma finsert-fsubset[simp]: $(\text{finsert } x \ A \subseteq B) = (x \in B \wedge A \subseteq B)$
 by (rule insert-subset[Transfer.transferred])

lemma finsert-inter-finsert[simp]: $\text{finsert } a \ A \mid \cap \mid \text{finsert } a \ B = \text{finsert } a \ (A \mid \cap \mid B)$
 by (rule insert-inter-insert[Transfer.transferred])

lemma finsert-disjoint[simp,no-atp]:
 $(\text{finsert } a \ A \mid \cap \mid B = \{\|\}) = (a \notin B \wedge A \mid \cap \mid B = \{\|\})$
 $(\{\|\} = \text{finsert } a \ A \mid \cap \mid B) = (a \notin B \wedge \{\|\} = A \mid \cap \mid B)$
 by (rule insert-disjoint[Transfer.transferred])+

lemma disjoint-finsert[simp,no-atp]:
 $(B \mid \cap \mid \text{finsert } a \ A = \{\|\}) = (a \notin B \wedge B \mid \cap \mid A = \{\|\})$
 $(\{\|\} = A \mid \cap \mid \text{finsert } b \ B) = (b \notin A \wedge \{\|\} = A \mid \cap \mid B)$
 by (rule disjoint-insert[Transfer.transferred])+

lemma fimage-fempty[simp]: $f \mid \upharpoonright \mid \{\|\} = \{\|\}$
 by (rule image-empty[Transfer.transferred])

lemma fimage-finsert[simp]: $f \mid \upharpoonright \mid \text{finsert } a \ B = \text{finsert } (f \ a) \ (f \mid \upharpoonright \mid B)$
 by (rule image-insert[Transfer.transferred])

lemma fimage-constant: $x \in A \implies (\lambda x. \ c) \mid \upharpoonright \mid A = \{|c\}$
 by (rule image-constant[Transfer.transferred])

lemma fimage-constant-conv: $(\lambda x. \ c) \mid \upharpoonright \mid A = (\text{if } A = \{\|\} \text{ then } \{\|\} \text{ else } \{|c\})$
 by (rule image-constant-conv[Transfer.transferred])

lemma fimage-fimage: $f \mid \upharpoonright \mid g \mid \upharpoonright \mid A = (\lambda x. \ f \ (g \ x)) \mid \upharpoonright \mid A$
 by (rule image-image[Transfer.transferred])

lemma finsert-fimage[simp]: $x \in A \implies \text{finsert } (f \ x) \ (f \mid \upharpoonright \mid A) = f \mid \upharpoonright \mid A$
 by (rule insert-image[Transfer.transferred])

lemma fimage-is-fempty[iff]: $(f \mid \upharpoonright \mid A = \{\|\}) = (A = \{\|\})$

by (rule *image-is-empty*[*Transfer.transferred*])

lemma *fempty-is-fimage[iff]*: $(\{\|\} = f \mid^{\dagger} A) = (A = \{\|\})$
by (rule *empty-is-image*[*Transfer.transferred*])

lemma *fimage-cong*: $M = N \implies (\bigwedge x. x \in N \implies f x = g x) \implies f \mid^{\dagger} M = g \mid^{\dagger} N$
by (rule *image-cong*[*Transfer.transferred*])

lemma *fimage-finter-fsubset*: $f \mid^{\dagger} (A \mid \cap B) \subseteq f \mid^{\dagger} A \mid \cap f \mid^{\dagger} B$
by (rule *image-Int-subset*[*Transfer.transferred*])

lemma *fimage-fminus-fsubset*: $f \mid^{\dagger} A \mid - f \mid^{\dagger} B \subseteq f \mid^{\dagger} (A \mid - B)$
by (rule *image-diff-subset*[*Transfer.transferred*])

lemma *finter-absorb*: $A \mid \cap A = A$
by (rule *Int-absorb*[*Transfer.transferred*])

lemma *finter-left-absorb*: $A \mid \cap (A \mid \cap B) = A \mid \cap B$
by (rule *Int-left-absorb*[*Transfer.transferred*])

lemma *finter-commute*: $A \mid \cap B = B \mid \cap A$
by (rule *Int-commute*[*Transfer.transferred*])

lemma *finter-left-commute*: $A \mid \cap (B \mid \cap C) = B \mid \cap (A \mid \cap C)$
by (rule *Int-left-commute*[*Transfer.transferred*])

lemma *finter-assoc*: $A \mid \cap B \mid \cap C = A \mid \cap (B \mid \cap C)$
by (rule *Int-assoc*[*Transfer.transferred*])

lemma *finter-ac*:
 $A \mid \cap B \mid \cap C = A \mid \cap (B \mid \cap C)$
 $A \mid \cap (A \mid \cap B) = A \mid \cap B$
 $A \mid \cap B = B \mid \cap A$
 $A \mid \cap (B \mid \cap C) = B \mid \cap (A \mid \cap C)$
by (rule *Int-ac*[*Transfer.transferred*])+

lemma *finter-absorb1*: $B \subseteq A \implies A \mid \cap B = B$
by (rule *Int-absorb1*[*Transfer.transferred*])

lemma *finter-absorb2*: $A \subseteq B \implies A \mid \cap B = A$
by (rule *Int-absorb2*[*Transfer.transferred*])

lemma *finter-fempty-left*: $\{\|\} \mid \cap B = \{\|\}$
by (rule *Int-empty-left*[*Transfer.transferred*])

lemma *finter-fempty-right*: $A \mid \cap \{\|\} = \{\|\}$
by (rule *Int-empty-right*[*Transfer.transferred*])

lemma *disjoint-iff-fnot-equal*: $(A \mid \cap \mid B = \{\mid\}) = fBall\ A\ (\lambda x. fBall\ B\ ((\neq)\ x))$
by (rule *disjoint-iff-not-equal*[*Transfer.transferred*])

lemma *finter-union-distrib*: $A \mid \cap \mid (B \mid \cup \mid C) = A \mid \cap \mid B \mid \cup \mid (A \mid \cap \mid C)$
by (rule *Int-Un-distrib*[*Transfer.transferred*])

lemma *finter-union-distrib2*: $B \mid \cup \mid C \mid \cap \mid A = B \mid \cap \mid A \mid \cup \mid (C \mid \cap \mid A)$
by (rule *Int-Un-distrib2*[*Transfer.transferred*])

lemma *finter-fsubset-iff*[*no-atp, simp*]: $(C \mid \subseteq \mid A \mid \cap \mid B) = (C \mid \subseteq \mid A \wedge C \mid \subseteq \mid B)$
by (rule *Int-subset-iff*[*Transfer.transferred*])

lemma *funion-absorb*: $A \mid \cup \mid A = A$
by (rule *Un-absorb*[*Transfer.transferred*])

lemma *funion-left-absorb*: $A \mid \cup \mid (A \mid \cup \mid B) = A \mid \cup \mid B$
by (rule *Un-left-absorb*[*Transfer.transferred*])

lemma *funion-commute*: $A \mid \cup \mid B = B \mid \cup \mid A$
by (rule *Un-commute*[*Transfer.transferred*])

lemma *funion-left-commute*: $A \mid \cup \mid (B \mid \cup \mid C) = B \mid \cup \mid (A \mid \cup \mid C)$
by (rule *Un-left-commute*[*Transfer.transferred*])

lemma *funion-assoc*: $A \mid \cup \mid B \mid \cup \mid C = A \mid \cup \mid (B \mid \cup \mid C)$
by (rule *Un-assoc*[*Transfer.transferred*])

lemma *funion-ac*:
 $A \mid \cup \mid B \mid \cup \mid C = A \mid \cup \mid (B \mid \cup \mid C)$
 $A \mid \cup \mid (A \mid \cup \mid B) = A \mid \cup \mid B$
 $A \mid \cup \mid B = B \mid \cup \mid A$
 $A \mid \cup \mid (B \mid \cup \mid C) = B \mid \cup \mid (A \mid \cup \mid C)$
by (rule *Un-ac*[*Transfer.transferred*])+

lemma *funion-absorb1*: $A \mid \subseteq \mid B \implies A \mid \cup \mid B = B$
by (rule *Un-absorb1*[*Transfer.transferred*])

lemma *funion-absorb2*: $B \mid \subseteq \mid A \implies A \mid \cup \mid B = A$
by (rule *Un-absorb2*[*Transfer.transferred*])

lemma *funion-fempty-left*: $\{\mid\} \mid \cup \mid B = B$
by (rule *Un-empty-left*[*Transfer.transferred*])

lemma *funion-fempty-right*: $A \mid \cup \mid \{\mid\} = A$
by (rule *Un-empty-right*[*Transfer.transferred*])

lemma *funion-finsert-left*[*simp*]: $finsert\ a\ B \mid \cup \mid C = finsert\ a\ (B \mid \cup \mid C)$
by (rule *Un-insert-left*[*Transfer.transferred*])

lemma *funion-finsert-right[simp]*: $A \mid\cup\mid \text{finsert } a \ B = \text{finsert } a \ (A \mid\cup\mid B)$
by (rule *Un-insert-right*[*Transfer.transferred*])

lemma *finter-finsert-left*: $\text{finsert } a \ B \mid\cap\mid C = (\text{if } a \mid\in\mid C \text{ then } \text{finsert } a \ (B \mid\cap\mid C) \text{ else } B \mid\cap\mid C)$
by (rule *Int-insert-left*[*Transfer.transferred*])

lemma *finter-finsert-left-iffempty[simp]*: $a \mid\notin\mid C \implies \text{finsert } a \ B \mid\cap\mid C = B \mid\cap\mid C$
by (rule *Int-insert-left-if0*[*Transfer.transferred*])

lemma *finter-finsert-left-if1[simp]*: $a \mid\in\mid C \implies \text{finsert } a \ B \mid\cap\mid C = \text{finsert } a \ (B \mid\cap\mid C)$
by (rule *Int-insert-left-if1*[*Transfer.transferred*])

lemma *finter-finsert-right*:
 $A \mid\cap\mid \text{finsert } a \ B = (\text{if } a \mid\in\mid A \text{ then } \text{finsert } a \ (A \mid\cap\mid B) \text{ else } A \mid\cap\mid B)$
by (rule *Int-insert-right*[*Transfer.transferred*])

lemma *finter-finsert-right-iffempty[simp]*: $a \mid\notin\mid A \implies A \mid\cap\mid \text{finsert } a \ B = A \mid\cap\mid B$
by (rule *Int-insert-right-if0*[*Transfer.transferred*])

lemma *finter-finsert-right-if1[simp]*: $a \mid\in\mid A \implies A \mid\cap\mid \text{finsert } a \ B = \text{finsert } a \ (A \mid\cap\mid B)$
by (rule *Int-insert-right-if1*[*Transfer.transferred*])

lemma *funion-finter-distrib*: $A \mid\cup\mid (B \mid\cap\mid C) = A \mid\cup\mid B \mid\cap\mid (A \mid\cup\mid C)$
by (rule *Un-Int-distrib*[*Transfer.transferred*])

lemma *funion-finter-distrib2*: $B \mid\cap\mid C \mid\cup\mid A = B \mid\cup\mid A \mid\cap\mid (C \mid\cup\mid A)$
by (rule *Un-Int-distrib2*[*Transfer.transferred*])

lemma *funion-finter-crazy*:
 $A \mid\cap\mid B \mid\cup\mid (B \mid\cap\mid C) \mid\cup\mid (C \mid\cap\mid A) = A \mid\cup\mid B \mid\cap\mid (B \mid\cup\mid C) \mid\cap\mid (C \mid\cup\mid A)$
by (rule *Un-Int-crazy*[*Transfer.transferred*])

lemma *fsubset-funion-eq*: $(A \mid\subseteq\mid B) = (A \mid\cup\mid B = B)$
by (rule *subset-Un-eq*[*Transfer.transferred*])

lemma *funion-fempty[iff]*: $(A \mid\cup\mid B = \{\mid\}) = (A = \{\mid\} \wedge B = \{\mid\})$
by (rule *Un-empty*[*Transfer.transferred*])

lemma *funion-fsubset-iff[no-atp, simp]*: $(A \mid\cup\mid B \mid\subseteq\mid C) = (A \mid\subseteq\mid C \wedge B \mid\subseteq\mid C)$
by (rule *Un-subset-iff*[*Transfer.transferred*])

lemma *funion-fminus-finter*: $A \mid-\mid B \mid\cup\mid (A \mid\cap\mid B) = A$
by (rule *Un-Diff-Int*[*Transfer.transferred*])

lemma *ffunion-empty[simp]*: $\text{ffUnion } \{\mid\} = \{\mid\}$

by (rule Union-empty[Transfer.transferred])

lemma *ffunion-mono*: $A \subseteq B \implies \text{ffUnion } A \subseteq \text{ffUnion } B$
 by (rule Union-mono[Transfer.transferred])

lemma *ffunion-insert[simp]*: $\text{ffUnion } (\text{finsert } a \ B) = a \cup \text{ffUnion } B$
 by (rule Union-insert[Transfer.transferred])

lemma *fminus-finter2*: $A \cap C \mid B \cap C = A \cap C \mid B$
 by (rule Diff-Int2[Transfer.transferred])

lemma *funion-finter-assoc-eq*: $(A \cap B \cup C = A \cap (B \cup C)) = (C \subseteq A)$
 by (rule Un-Int-assoc-eq[Transfer.transferred])

lemma *fBall-funion*: $\text{fBall } (A \cup B) \ P = (\text{fBall } A \ P \wedge \text{fBall } B \ P)$
 by (rule ball-Un[Transfer.transferred])

lemma *fBex-funion*: $\text{fBex } (A \cup B) \ P = (\text{fBex } A \ P \vee \text{fBex } B \ P)$
 by (rule bex-Un[Transfer.transferred])

lemma *fminus-eq-fempty-iff[simp,no-atp]*: $(A \mid B = \{\}) = (A \subseteq B)$
 by (rule Diff-eq-empty-iff[Transfer.transferred])

lemma *fminus-cancel[simp]*: $A \mid A = \{\}$
 by (rule Diff-cancel[Transfer.transferred])

lemma *fminus-idemp[simp]*: $A \mid B \mid B = A \mid B$
 by (rule Diff-idemp[Transfer.transferred])

lemma *fminus-triv*: $A \cap B = \{\} \implies A \mid B = A$
 by (rule Diff-triv[Transfer.transferred])

lemma *fempty-fminus[simp]*: $\{\} \mid A = \{\}$
 by (rule empty-Diff[Transfer.transferred])

lemma *fminus-fempty[simp]*: $A \mid \{\} = A$
 by (rule Diff-empty[Transfer.transferred])

lemma *fminus-finsertffempty[simp,no-atp]*: $x \notin A \implies A \mid \text{finsert } x \ B = A \mid B$
 by (rule Diff-insert0[Transfer.transferred])

lemma *fminus-finsert*: $A \mid \text{finsert } a \ B = A \mid B \mid \{a\}$
 by (rule Diff-insert[Transfer.transferred])

lemma *fminus-finsert2*: $A \mid \text{finsert } a \ B = A \mid \{a\} \mid B$
 by (rule Diff-insert2[Transfer.transferred])

lemma *finsert-fminus-if*: $\text{finsert } x \ A \mid B = (\text{if } x \in B \text{ then } A \mid B \text{ else } \text{finsert } x \ A)$

$x (A \mid - \mid B)$

by (rule *insert-Diff-if*[*Transfer.transferred*])

lemma *finert-fminus1[simp]*: $x \in B \implies \text{finert } x A \mid - \mid B = A \mid - \mid B$

by (rule *insert-Diff1*[*Transfer.transferred*])

lemma *finert-fminus-single[simp]*: $\text{finert } a (A \mid - \mid \{|a|\}) = \text{finert } a A$

by (rule *insert-Diff-single*[*Transfer.transferred*])

lemma *finert-fminus*: $a \in A \implies \text{finert } a (A \mid - \mid \{|a|\}) = A$

by (rule *insert-Diff*[*Transfer.transferred*])

lemma *fminus-finert-absorb*: $x \notin A \implies \text{finert } x A \mid - \mid \{|x|\} = A$

by (rule *Diff-insert-absorb*[*Transfer.transferred*])

lemma *fminus-disjoint[simp]*: $A \mid \cap \mid (B \mid - \mid A) = \{|\}\}$

by (rule *Diff-disjoint*[*Transfer.transferred*])

lemma *fminus-partition*: $A \subseteq B \implies A \mid \cup \mid (B \mid - \mid A) = B$

by (rule *Diff-partition*[*Transfer.transferred*])

lemma *double-fminus*: $A \subseteq B \implies B \subseteq C \implies B \mid - \mid (C \mid - \mid A) = A$

by (rule *double-diff*[*Transfer.transferred*])

lemma *funion-fminus-cancel[simp]*: $A \mid \cup \mid (B \mid - \mid A) = A \mid \cup \mid B$

by (rule *Un-Diff-cancel*[*Transfer.transferred*])

lemma *funion-fminus-cancel2[simp]*: $B \mid - \mid A \mid \cup \mid A = B \mid \cup \mid A$

by (rule *Un-Diff-cancel2*[*Transfer.transferred*])

lemma *fminus-funion*: $A \mid - \mid (B \mid \cup \mid C) = A \mid - \mid B \mid \cap \mid (A \mid - \mid C)$

by (rule *Diff-Un*[*Transfer.transferred*])

lemma *fminus-finter*: $A \mid - \mid (B \mid \cap \mid C) = A \mid - \mid B \mid \cup \mid (A \mid - \mid C)$

by (rule *Diff-Int*[*Transfer.transferred*])

lemma *funion-fminus*: $A \mid \cup \mid B \mid - \mid C = A \mid - \mid C \mid \cup \mid (B \mid - \mid C)$

by (rule *Un-Diff*[*Transfer.transferred*])

lemma *finter-fminus*: $A \mid \cap \mid B \mid - \mid C = A \mid \cap \mid (B \mid - \mid C)$

by (rule *Int-Diff*[*Transfer.transferred*])

lemma *fminus-finter-distrib*: $C \mid \cap \mid (A \mid - \mid B) = C \mid \cap \mid A \mid - \mid (C \mid \cap \mid B)$

by (rule *Diff-Int-distrib*[*Transfer.transferred*])

lemma *fminus-finter-distrib2*: $A \mid - \mid B \mid \cap \mid C = A \mid \cap \mid C \mid - \mid (B \mid \cap \mid C)$

by (rule *Diff-Int-distrib2*[*Transfer.transferred*])

lemma *fUNIV-bool[no-atp]*: $fUNIV = \{False, True\}$

by (rule UNIV-bool[Transfer.transferred])

lemma *fPow-fempty[simp]*: $fPow \{\mid\} = \{\{\mid\}\}$
 by (rule Pow-empty[Transfer.transferred])

lemma *fPow-finset*: $fPow (finset a A) = fPow A \mid\cup\mid finset a \mid\mid fPow A$
 by (rule Pow-insert[Transfer.transferred])

lemma *funion-fPow-fsubset*: $fPow A \mid\cup\mid fPow B \mid\subseteq\mid fPow (A \mid\cup\mid B)$
 by (rule Un-Pow-subset[Transfer.transferred])

lemma *fPow-finter-eq[simp]*: $fPow (A \mid\cap\mid B) = fPow A \mid\cap\mid fPow B$
 by (rule Pow-Int-eq[Transfer.transferred])

lemma *fset-eq-fsubset*: $(A = B) = (A \mid\subseteq\mid B \wedge B \mid\subseteq\mid A)$
 by (rule set-eq-subset[Transfer.transferred])

lemma *fsubset-iff[no-atp]*: $(A \mid\subseteq\mid B) = (\forall t. t \mid\in\mid A \longrightarrow t \mid\in\mid B)$
 by (rule subset-iff[Transfer.transferred])

lemma *fsubset-iff-pfsubset-eq*: $(A \mid\subseteq\mid B) = (A \mid\subset\mid B \vee A = B)$
 by (rule subset-iff-psubset-eq[Transfer.transferred])

lemma *all-not-fin-conv[simp]*: $(\forall x. x \mid\notin\mid A) = (A = \{\mid\})$
 by (rule all-not-in-conv[Transfer.transferred])

lemma *ex-fin-conv*: $(\exists x. x \mid\in\mid A) = (A \neq \{\mid\})$
 by (rule ex-in-conv[Transfer.transferred])

lemma *fimage-mono*: $A \mid\subseteq\mid B \Longrightarrow f \mid\mid A \mid\subseteq\mid f \mid\mid B$
 by (rule image-mono[Transfer.transferred])

lemma *fPow-mono*: $A \mid\subseteq\mid B \Longrightarrow fPow A \mid\subseteq\mid fPow B$
 by (rule Pow-mono[Transfer.transferred])

lemma *finset-mono*: $C \mid\subseteq\mid D \Longrightarrow finset a C \mid\subseteq\mid finset a D$
 by (rule insert-mono[Transfer.transferred])

lemma *funion-mono*: $A \mid\subseteq\mid C \Longrightarrow B \mid\subseteq\mid D \Longrightarrow A \mid\cup\mid B \mid\subseteq\mid C \mid\cup\mid D$
 by (rule Un-mono[Transfer.transferred])

lemma *finter-mono*: $A \mid\subseteq\mid C \Longrightarrow B \mid\subseteq\mid D \Longrightarrow A \mid\cap\mid B \mid\subseteq\mid C \mid\cap\mid D$
 by (rule Int-mono[Transfer.transferred])

lemma *fminus-mono*: $A \mid\subseteq\mid C \Longrightarrow D \mid\subseteq\mid B \Longrightarrow A \mid-\mid B \mid\subseteq\mid C \mid-\mid D$
 by (rule Diff-mono[Transfer.transferred])

lemma *fin-mono*: $A \mid\subseteq\mid B \Longrightarrow x \mid\in\mid A \longrightarrow x \mid\in\mid B$
 by (rule in-mono[Transfer.transferred])

lemma *fthe-felem-eq[simp]*: $fthe\text{-}elem\ \{|x|\} = x$
by (*rule the-elem-eq[Transfer.transferred]*)

lemma *fLeast-mono*:
 $mono\ f \implies fBex\ S\ (\lambda x. fBall\ S\ ((\leq)\ x)) \implies (LEAST\ y. y\ |\in|\ f\ |\cdot|\ S) = f$
 $(LEAST\ x. x\ |\in|\ S)$
by (*rule Least-mono[Transfer.transferred]*)

lemma *fbind-fbind*: $fbind\ (fbind\ A\ B)\ C = fbind\ A\ (\lambda x. fbind\ (B\ x)\ C)$
by (*rule Set.bind-bind[Transfer.transferred]*)

lemma *fempty-fbind[simp]*: $fbind\ \{|\}\ f = \{|\}$
by (*rule empty-bind[Transfer.transferred]*)

lemma *nonempty-fbind-const*: $A \neq \{|\} \implies fbind\ A\ (\lambda\cdot. B) = B$
by (*rule nonempty-bind-const[Transfer.transferred]*)

lemma *fbind-const*: $fbind\ A\ (\lambda\cdot. B) = (if\ A = \{|\}\ then\ \{|\}\ else\ B)$
by (*rule bind-const[Transfer.transferred]*)

lemma *ffmember-filter[simp]*: $(x\ |\in|\ ffilter\ P\ A) = (x\ |\in|\ A \wedge P\ x)$
by *transfer simp*

lemma *fequalityI*: $A\ |\subseteq|\ B \implies B\ |\subseteq|\ A \implies A = B$
by (*rule equalityI[Transfer.transferred]*)

lemma *fset-of-list-simps[simp]*:
 $fset\text{-}of\text{-}list\ [] = \{|\}$
 $fset\text{-}of\text{-}list\ (x21\ \# x22) = finset\ x21\ (fset\text{-}of\text{-}list\ x22)$
by (*rule set-simps[Transfer.transferred]*)**+**

lemma *fset-of-list-append[simp]*: $fset\text{-}of\text{-}list\ (xs\ @\ ys) = fset\text{-}of\text{-}list\ xs\ |\cup|\ fset\text{-}of\text{-}list\ ys$
by (*rule set-append[Transfer.transferred]*)

lemma *fset-of-list-rev[simp]*: $fset\text{-}of\text{-}list\ (rev\ xs) = fset\text{-}of\text{-}list\ xs$
by (*rule set-rev[Transfer.transferred]*)

lemma *fset-of-list-map[simp]*: $fset\text{-}of\text{-}list\ (map\ f\ xs) = f\ |\cdot|\ fset\text{-}of\text{-}list\ xs$
by (*rule set-map[Transfer.transferred]*)

29.5 Additional lemmas

29.5.1 *ffUnion*

lemma *ffUnion-union-distrib[simp]*: $ffUnion\ (A\ |\cup|\ B) = ffUnion\ A\ |\cup|\ ffUnion\ B$
by (*rule Union-Un-distrib[Transfer.transferred]*)

29.5.2 *fbind*

lemma *fbind-cong*[*fundef-cong*]: $A = B \implies (\bigwedge x. x \in B \implies f x = g x) \implies fbind A f = fbind B g$
by *transfer force*

29.5.3 *fsingleton*

lemma *fsingletonE*: $b \in \{a\} \implies (b = a \implies thesis) \implies thesis$
by (*rule fsingletonD* [*elim-format*])

29.5.4 *fempty*

lemma *fempty-ffilter*[*simp*]: $ffilter (\lambda-. False) A = \{\}$
by *transfer auto*

lemma *femptyE* [*elim!*]: $a \in \{\} \implies P$
by *simp*

29.5.5 *fset*

lemma *fset-simps*[*simp*]:
 $fset \{\} = \{\}$
 $fset (finsert x X) = insert x (fset X)$
by (*rule bot-fset.rep-eq finsert.rep-eq*) $+$

lemma *finite-fset* [*simp*]:
shows *finite* (*fset* *S*)
by *transfer simp*

lemmas *fset-cong* = *fset-inject*

lemma *filter-fset* [*simp*]:
shows $fset (ffilter P xs) = Collect P \cap fset xs$
by *transfer auto*

lemma *inter-fset*[*simp*]: $fset (A \cap B) = fset A \cap fset B$
by (*rule inf-fset.rep-eq*)

lemma *union-fset*[*simp*]: $fset (A \cup B) = fset A \cup fset B$
by (*rule sup-fset.rep-eq*)

lemma *minus-fset*[*simp*]: $fset (A - B) = fset A - fset B$
by (*rule minus-fset.rep-eq*)

29.5.6 *ffilter*

lemma *subset-ffilter*:
 $ffilter P A \subseteq ffilter Q A = (\forall x. x \in A \longrightarrow P x \longrightarrow Q x)$

by *transfer auto*

lemma *eq-ffilter*:

$(\text{ffilter } P \ A = \text{ffilter } Q \ A) = (\forall x. x \in A \longrightarrow P \ x = Q \ x)$

by *transfer auto*

lemma *pfssubset-ffilter*:

$(\bigwedge x. x \in A \implies P \ x \implies Q \ x) \implies (x \in A \wedge \neg P \ x \wedge Q \ x) \implies$
 $\text{ffilter } P \ A \mid\subseteq \text{ffilter } Q \ A$

unfolding *less-fset-def* **by** (*auto simp add: subset-ffilter eq-ffilter*)

29.5.7 *fset-of-list*

lemma *fset-of-list-filter[simp]*:

$\text{fset-of-list } (\text{filter } P \ xs) = \text{ffilter } P \ (\text{fset-of-list } xs)$

by *transfer auto*

lemma *fset-of-list-subset[intro]*:

$\text{set } xs \subseteq \text{set } ys \implies \text{fset-of-list } xs \mid\subseteq \text{fset-of-list } ys$

by *transfer simp*

lemma *fset-of-list-elem*: $(x \in \text{fset-of-list } xs) \longleftrightarrow (x \in \text{set } xs)$

by *transfer simp*

29.5.8 *fininsert*

lemma *set-fininsert*:

assumes $x \in A$

obtains B **where** $A = \text{fininsert } x \ B$ **and** $x \notin B$

using *assms* **by** *transfer (metis Set.set-insert finite-insert)*

lemma *mk-disjoint-fininsert*: $a \in A \implies \exists B. A = \text{fininsert } a \ B \wedge a \notin B$

by (*rule exI [where $x = A \mid - \mid \{a\}$]*) *blast*

lemma *fininsert-eq-iff*:

assumes $a \notin A$ **and** $b \notin B$

shows $(\text{fininsert } a \ A = \text{fininsert } b \ B) =$

$(\text{if } a = b \text{ then } A = B \text{ else } \exists C. A = \text{fininsert } b \ C \wedge b \notin C \wedge B = \text{fininsert } a \ C \wedge$

$a \notin C)$

using *assms* **by** *transfer (force simp: insert-eq-iff)*

29.5.9 *fimage*

lemma *subset-fimage-iff*: $(B \mid\subseteq f \mid^{\circ} A) = (\exists AA. AA \mid\subseteq A \wedge B = f \mid^{\circ} AA)$

by *transfer (metis mem-Collect-eq rev-finite-subset subset-image-iff)*

lemma *fimage-strict-mono*:

assumes *inj-on* f (*fset* B) **and** $A \mid\subseteq B$

shows $f \mid^{\circ} A \mid\subseteq f \mid^{\circ} B$

— TODO: Configure transfer framework to lift $\llbracket \text{inj-on } ?f \text{ } ?B; ?A \subset ?B \rrbracket \implies ?f \text{ } ?A \subset ?f \text{ } ?B$.

proof (*rule pfssubsetI*)

from $\langle A \mid \subset \mid B \rangle$ **have** $A \mid \subseteq \mid B$
 by (*rule pfssubset-imp-fsubset*)
 thus $f \mid \upharpoonright A \mid \subseteq \mid f \mid \upharpoonright B$
 by (*rule fimage-mono*)

next

from $\langle A \mid \subset \mid B \rangle$ **have** $A \mid \subseteq \mid B$ **and** $A \neq B$
 by (*simp-all add: pfssubset-eq*)

have $fset A \neq fset B$
 using $\langle A \neq B \rangle$
 by (*simp add: fset-cong*)
 hence $f \text{ } fset A \neq f \text{ } fset B$
 using $\langle A \mid \subseteq \mid B \rangle$
 by (*simp add: inj-on-image-eq-iff[OF $\langle \text{inj-on } f \text{ } (fset B) \rangle$] less-eq-fset.rep-eq*)
 hence $fset (f \mid \upharpoonright A) \neq fset (f \mid \upharpoonright B)$
 by (*simp add: fimage.rep-eq*)
 thus $f \mid \upharpoonright A \neq f \mid \upharpoonright B$
 by (*simp add: fset-cong*)

qed

29.5.10 bounded quantification

lemma *bex-simps* [*simp, no-atp*]:

$\bigwedge A P Q. fBex A (\lambda x. P x \wedge Q) = (fBex A P \wedge Q)$
 $\bigwedge A P Q. fBex A (\lambda x. P \wedge Q x) = (P \wedge fBex A Q)$
 $\bigwedge P. fBex \{\mid\} P = False$
 $\bigwedge a B P. fBex (fininsert a B) P = (P a \vee fBex B P)$
 $\bigwedge A P f. fBex (f \mid \upharpoonright A) P = fBex A (\lambda x. P (f x))$
 $\bigwedge A P. (\neg fBex A P) = fBall A (\lambda x. \neg P x)$

by *auto*

lemma *ball-simps* [*simp, no-atp*]:

$\bigwedge A P Q. fBall A (\lambda x. P x \vee Q) = (fBall A P \vee Q)$
 $\bigwedge A P Q. fBall A (\lambda x. P \vee Q x) = (P \vee fBall A Q)$
 $\bigwedge A P Q. fBall A (\lambda x. P \longrightarrow Q x) = (P \longrightarrow fBall A Q)$
 $\bigwedge A P Q. fBall A (\lambda x. P x \longrightarrow Q) = (fBex A P \longrightarrow Q)$
 $\bigwedge P. fBall \{\mid\} P = True$
 $\bigwedge a B P. fBall (fininsert a B) P = (P a \wedge fBall B P)$
 $\bigwedge A P f. fBall (f \mid \upharpoonright A) P = fBall A (\lambda x. P (f x))$
 $\bigwedge A P. (\neg fBall A P) = fBex A (\lambda x. \neg P x)$

by *auto*

lemma *atomize-fBall*:

$(\bigwedge x. x \mid \in \mid A \implies P x) == Trueprop (fBall A (\lambda x. P x))$
 by (*simp add: Set.atomize-ball*)

lemma *fBall-mono*[*mono*]: $P \leq Q \implies fBall\ S\ P \leq fBall\ S\ Q$
by *auto*

lemma *fBex-mono*[*mono*]: $P \leq Q \implies fBex\ S\ P \leq fBex\ S\ Q$
by *auto*

end

29.5.11 *fcard*

lemma *fcard-empty*:
 $fcard\ \{\|\} = 0$
by *transfer (rule card.empty)*

lemma *fcard-finsert-disjoint*:
 $x \notin A \implies fcard\ (finsert\ x\ A) = Suc\ (fcard\ A)$
by *transfer (rule card-insert-disjoint)*

lemma *fcard-finsert-if*:
 $fcard\ (finsert\ x\ A) = (if\ x \in A\ then\ fcard\ A\ else\ Suc\ (fcard\ A))$
by *transfer (rule card-insert-if)*

lemma *fcard-0-eq* [*simp*, *no-atp*]:
 $fcard\ A = 0 \longleftrightarrow A = \{\|\}$
by *transfer (rule card-0-eq)*

lemma *fcard-Suc-fminus1*:
 $x \in A \implies Suc\ (fcard\ (A\ |-| \ \{|x|\})) = fcard\ A$
by *transfer (rule card-Suc-Diff1)*

lemma *fcard-fminus-fsingleton*:
 $x \in A \implies fcard\ (A\ |-| \ \{|x|\}) = fcard\ A - 1$
by *transfer (rule card-Diff-singleton)*

lemma *fcard-fminus-fsingleton-if*:
 $fcard\ (A\ |-| \ \{|x|\}) = (if\ x \in A\ then\ fcard\ A - 1\ else\ fcard\ A)$
by *transfer (rule card-Diff-singleton-if)*

lemma *fcard-fminus-finsert*[*simp*]:
assumes $a \in A$ **and** $a \notin B$
shows $fcard\ (A\ |-| \ finsert\ a\ B) = fcard\ (A\ |-| \ B) - 1$
using *assms* **by** *transfer (rule card-Diff-insert)*

lemma *fcard-finsert*: $fcard\ (finsert\ x\ A) = Suc\ (fcard\ (A\ |-| \ \{|x|\}))$
by *transfer (rule card.insert-remove)*

lemma *fcard-finsert-le*: $fcard\ A \leq fcard\ (finsert\ x\ A)$
by *transfer (rule card-insert-le)*

lemma *fcard-mono*:

$$A \mid\subseteq\mid B \implies \text{fcard } A \leq \text{fcard } B$$

by *transfer* (rule *card-mono*)

lemma *fcard-seteq*: $A \mid\subseteq\mid B \implies \text{fcard } B \leq \text{fcard } A \implies A = B$

by *transfer* (rule *card-seteq*)

lemma *pfssubset-fcard-mono*: $A \mid\subset\mid B \implies \text{fcard } A < \text{fcard } B$

by *transfer* (rule *pfssubset-card-mono*)

lemma *fcard-union-finter*:

$$\text{fcard } A + \text{fcard } B = \text{fcard } (A \mid\cup\mid B) + \text{fcard } (A \mid\cap\mid B)$$

by *transfer* (rule *card-Un-Int*)

lemma *fcard-union-disjoint*:

$$A \mid\cap\mid B = \{\mid\} \implies \text{fcard } (A \mid\cup\mid B) = \text{fcard } A + \text{fcard } B$$

by *transfer* (rule *card-Un-disjoint*)

lemma *fcard-union-fsubset*:

$$B \mid\subseteq\mid A \implies \text{fcard } (A \mid-\mid B) = \text{fcard } A - \text{fcard } B$$

by *transfer* (rule *card-Diff-subset*)

lemma *diff-fcard-le-fcard-fminus*:

$$\text{fcard } A - \text{fcard } B \leq \text{fcard } (A \mid-\mid B)$$

by *transfer* (rule *diff-card-le-card-Diff*)

lemma *fcard-fminus1-less*: $x \mid\in\mid A \implies \text{fcard } (A \mid-\mid \{|x|\}) < \text{fcard } A$

by *transfer* (rule *card-Diff1-less*)

lemma *fcard-fminus2-less*:

$$x \mid\in\mid A \implies y \mid\in\mid A \implies \text{fcard } (A \mid-\mid \{|x|\} \mid-\mid \{|y|\}) < \text{fcard } A$$

by *transfer* (rule *card-Diff2-less*)

lemma *fcard-fminus1-le*: $\text{fcard } (A \mid-\mid \{|x|\}) \leq \text{fcard } A$

by *transfer* (rule *card-Diff1-le*)

lemma *fcard-pfssubset*: $A \mid\subseteq\mid B \implies \text{fcard } A < \text{fcard } B \implies A < B$

by *transfer* (rule *card-pfssubset*)

29.5.12 sorted-list-of-fset

lemma *sorted-list-of-fset-simps[simp]*:

$$\text{set } (\text{sorted-list-of-fset } S) = \text{fset } S$$

$$\text{fset-of-list } (\text{sorted-list-of-fset } S) = S$$

by (*transfer*, *simp*)⁺

29.5.13 ffold

context *comp-fun-commute*

begin

```

lemma ffold-empty[simp]:  $\text{ffold } f \ z \ \{\mid\} = z$ 
  by (rule fold-empty[Transfer.transferred])

lemma ffold-finsert [simp]:
  assumes  $x \notin A$ 
  shows  $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ A)$ 
  using assms by (transfer fixing: f) (rule fold-insert)

lemma ffold-fun-left-comm:
   $f \ x \ (\text{ffold } f \ z \ A) = \text{ffold } f \ (f \ x \ z) \ A$ 
  by (transfer fixing: f) (rule fold-fun-left-comm)

lemma ffold-finsert2:
   $x \notin A \implies \text{ffold } f \ z \ (\text{finsert } x \ A) = \text{ffold } f \ (f \ x \ z) \ A$ 
  by (transfer fixing: f) (rule fold-insert2)

lemma ffold-rec:
  assumes  $x \in A$ 
  shows  $\text{ffold } f \ z \ A = f \ x \ (\text{ffold } f \ z \ (A \mid - \ \{\mid x\}))$ 
  using assms by (transfer fixing: f) (rule fold-rec)

lemma ffold-finsert-remove:
   $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ (A \mid - \ \{\mid x\}))$ 
  by (transfer fixing: f) (rule fold-insert-remove)
end

lemma ffold-fimage:
  assumes inj-on  $g \ (fset \ A)$ 
  shows  $\text{ffold } f \ z \ (g \ \mid \ A) = \text{ffold } (f \circ g) \ z \ A$ 
using assms by transfer' (rule fold-image)

lemma ffold-cong:
  assumes comp-fun-commute  $f \ \text{comp-fun-commute } g$ 
   $\bigwedge x. x \in A \implies f \ x = g \ x$ 
  and  $s = t$  and  $A = B$ 
  shows  $\text{ffold } f \ s \ A = \text{ffold } g \ t \ B$ 
  using assms[unfolded comp-fun-commute-def]
  by transfer (meson Finite-Set.fold-cong subset-UNIV)

context comp-fun-idem
begin

lemma ffold-finsert-idem:
   $\text{ffold } f \ z \ (\text{finsert } x \ A) = f \ x \ (\text{ffold } f \ z \ A)$ 
  by (transfer fixing: f) (rule fold-insert-idem)

declare ffold-finsert [simp del] ffold-finsert-idem [simp]

lemma ffold-finsert-idem2:

```

$\text{ffold } f \ z \ (\text{fininsert } x \ A) = \text{ffold } f \ (f \ x \ z) \ A$
by (*transfer fixing: f*) (*rule fold-insert-idem2*)

end

29.5.14 $(|\subset|)$

lemma *wfP-pfssubset*: $\text{wfP } (|\subset|)$
proof (*rule wfp-if-convertible-to-nat*)
show $\bigwedge x \ y. \ x \ |\subset| \ y \implies \text{fcard } x < \text{fcard } y$
by (*rule pfssubset-fcard-mono*)
qed

29.5.15 Group operations

locale *comm-monoid-fset* = *comm-monoid*
begin

sublocale *set*: *comm-monoid-set* ..

lift-definition $F :: ('b \Rightarrow 'a) \Rightarrow 'b \text{ fset} \Rightarrow 'a \text{ is set.F}$.

lemma *cong[fundef-cong]*: $A = B \implies (\bigwedge x. \ x \ |\in| \ B \implies g \ x = h \ x) \implies F \ g \ A = F \ h \ B$
by (*rule set.cong[Transfer.transferred]*)

lemma *cong-simp[cong]*:
 $\llbracket A = B; \bigwedge x. \ x \ |\in| \ B =_{\text{simp}} \implies g \ x = h \ x \rrbracket \implies F \ g \ A = F \ h \ B$
unfolding *simp-implies-def* **by** (*auto cong: cong*)

end

context *comm-monoid-add* **begin**

sublocale *fsum*: *comm-monoid-fset plus 0*
rewrites *comm-monoid-set.F plus 0 = sum*
defines *fsum = fsum.F*

proof –
show *comm-monoid-fset (+) 0* **by** *standard*

show *comm-monoid-set.F (+) 0 = sum* **unfolding** *sum-def* ..
qed

end

29.5.16 Semilattice operations

locale *semilattice-fset* = *semilattice*
begin

sublocale *set*: *semilattice-set* ..

lift-definition $F :: 'a \text{ fset} \Rightarrow 'a \text{ is set}.F$.

lemma *eq-fold*: $F (\text{finsert } x \ A) = \text{ffold } f \ x \ A$
by *transfer* (*rule set.eq-fold*)

lemma *singleton* [*simp*]: $F \{|x|\} = x$
by *transfer* (*rule set.singleton*)

lemma *insert-not-elem*: $x \notin A \Longrightarrow A \neq \{|\}\Longrightarrow F (\text{finsert } x \ A) = x * F \ A$
by *transfer* (*rule set.insert-not-elem*)

lemma *in-idem*: $x \in A \Longrightarrow x * F \ A = F \ A$
by *transfer* (*rule set.in-idem*)

lemma *insert* [*simp*]: $A \neq \{|\}\Longrightarrow F (\text{finsert } x \ A) = x * F \ A$
by *transfer* (*rule set.insert*)

end

locale *semilattice-order-fset* = *binary?*: *semilattice-order* + *semilattice-fset*
begin

end

context *linorder* **begin**

sublocale *fMin*: *semilattice-order-fset* *min* *less-eq* *less*
rewrites *semilattice-set.F* *min* = *Min*
defines *fMin* = *fMin.F*

proof –

show *semilattice-order-fset* *min* (\leq) ($<$) **by** *standard*

show *semilattice-set.F* *min* = *Min* **unfolding** *Min-def* ..

qed

sublocale *fMax*: *semilattice-order-fset* *max* *greater-eq* *greater*
rewrites *semilattice-set.F* *max* = *Max*
defines *fMax* = *fMax.F*

proof –

show *semilattice-order-fset* *max* (\geq) ($>$)
by *standard*

show *semilattice-set.F* *max* = *Max*

unfolding *Max-def* ..

qed

end

lemma *mono-fMax-commute*: $\text{mono } f \implies A \neq \{\mid\} \implies f (fMax A) = fMax (f \mid A)$
by *transfer (rule mono-Max-commute)*

lemma *mono-fMin-commute*: $\text{mono } f \implies A \neq \{\mid\} \implies f (fMin A) = fMin (f \mid A)$
by *transfer (rule mono-Min-commute)*

lemma *fMax-in[simp]*: $A \neq \{\mid\} \implies fMax A \mid A$
by *transfer (rule Max-in)*

lemma *fMin-in[simp]*: $A \neq \{\mid\} \implies fMin A \mid A$
by *transfer (rule Min-in)*

lemma *fMax-ge[simp]*: $x \mid A \implies x \leq fMax A$
by *transfer (rule Max-ge)*

lemma *fMin-le[simp]*: $x \mid A \implies fMin A \leq x$
by *transfer (rule Min-le)*

lemma *fMax-eqI*: $(\bigwedge y. y \mid A \implies y \leq x) \implies x \mid A \implies fMax A = x$
by *transfer (rule Max-eqI)*

lemma *fMin-eqI*: $(\bigwedge y. y \mid A \implies x \leq y) \implies x \mid A \implies fMin A = x$
by *transfer (rule Min-eqI)*

lemma *fMax-finsert[simp]*: $fMax (finsert x A) = (\text{if } A = \{\mid\} \text{ then } x \text{ else } max x (fMax A))$
by *transfer simp*

lemma *fMin-finsert[simp]*: $fMin (finsert x A) = (\text{if } A = \{\mid\} \text{ then } x \text{ else } min x (fMin A))$
by *transfer simp*

context *linorder* **begin**

lemma *fset-linorder-max-induct[case-names fempty finsert]*:
assumes $P \{\mid\}$
and $\bigwedge x S. [\forall y. y \mid S \longrightarrow y < x; P S] \implies P (finsert x S)$
shows $P S$
proof –

note *Domainp-forall-transfer[transfer-rule]*
show *?thesis*
using *assms* **by** (*transfer fixing: less*) (*auto intro: finite-linorder-max-induct*)
qed


```

lemma fset-linorder-min-induct [case-names fempty finsert]:
  assumes  $P \{\mid\}$ 
  and  $\bigwedge x S. \llbracket \forall y. y \in S \longrightarrow y > x; P S \rrbracket \Longrightarrow P (finsert\ x\ S)$ 
  shows  $P\ S$ 
proof –

  note Domainp-forall-transfer [transfer-rule]
  show ?thesis
  using assms by (transfer fixing: less) (auto intro: finite-linorder-min-induct)
qed

end

```

29.6 Choice in fsets

```

lemma fset-choice:
  assumes  $\forall x. x \in A \longrightarrow (\exists y. P\ x\ y)$ 
  shows  $\exists f. \forall x. x \in A \longrightarrow P\ x\ (f\ x)$ 
  using assms by transfermetis

```

29.7 Induction and Cases rules for fsets

```

lemma fset-exhaust [case-names empty insert, cases type: fset]:
  assumes fempty-case:  $S = \{\mid\} \Longrightarrow P$ 
  and finsert-case:  $\bigwedge x S'. S = finsert\ x\ S' \Longrightarrow P$ 
  shows  $P$ 
  using assms by transfer blast

```

```

lemma fset-induct [case-names empty insert]:
  assumes fempty-case:  $P \{\mid\}$ 
  and finsert-case:  $\bigwedge x S. P\ S \Longrightarrow P (finsert\ x\ S)$ 
  shows  $P\ S$ 
proof –

```

```

  note Domainp-forall-transfer [transfer-rule]
  show ?thesis
  using assms by transfer (auto intro: finite-induct)
qed

```

```

lemma fset-induct-stronger [case-names empty insert, induct type: fset]:
  assumes empty-fset-case:  $P \{\mid\}$ 
  and insert-fset-case:  $\bigwedge x S. \llbracket x \notin S; P\ S \rrbracket \Longrightarrow P (finsert\ x\ S)$ 
  shows  $P\ S$ 
proof –

```

```

  note Domainp-forall-transfer [transfer-rule]
  show ?thesis
  using assms by transfer (auto intro: finite-induct)
qed

```

lemma *fset-card-induct*:

assumes *empty-fset-case*: $P \{\mid\}$
and *card-fset-Suc-case*: $\bigwedge S \ T. \text{Suc} (\text{fcard } S) = (\text{fcard } T) \implies P \ S \implies P \ T$
shows $P \ S$
proof (*induct* S)
case *empty*
show $P \ \{\mid\}$ **by** (*rule empty-fset-case*)
next
case (*insert* $x \ S$)
have h : $P \ S$ **by** *fact*
have $x \notin S$ **by** *fact*
then have $\text{Suc} (\text{fcard } S) = \text{fcard} (\text{finsert } x \ S)$
by *transfer auto*
then show $P (\text{finsert } x \ S)$
using h *card-fset-Suc-case* **by** *simp*
qed

lemma *fset-strong-cases*:

obtains $xs = \{\mid\}$
 $\mid ys \ x$ **where** $x \notin ys$ **and** $xs = \text{finsert } x \ ys$
by *auto*

lemma *fset-induct2*:

$P \ \{\mid\} \ \{\mid\} \implies$
 $(\bigwedge x \ xs. \ x \notin xs \implies P (\text{finsert } x \ xs) \ \{\mid\}) \implies$
 $(\bigwedge y \ ys. \ y \notin ys \implies P \ \{\mid\} (\text{finsert } y \ ys)) \implies$
 $(\bigwedge x \ xs \ y \ ys. \ \llbracket P \ xs \ ys; \ x \notin xs; \ y \notin ys \rrbracket \implies P (\text{finsert } x \ xs) (\text{finsert } y \ ys)) \implies$
 $P \ xsa \ ysa$
by (*induct* xsa *arbitrary*: ysa ; *metis fset-induct-stronger*)

29.8 Lemmas depending on induction

lemma *ffUnion-fsubset-iff*: $\text{ffUnion } A \mid\subseteq B \longleftrightarrow \text{fBall } A \ (\lambda x. \ x \mid\subseteq B)$
by (*induction* A) *simp-all*

29.9 Setup for Lifting/Transfer

29.9.1 Relator and predicate properties

lift-definition *rel-fset* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ fset} \Rightarrow 'b \text{ fset} \Rightarrow \text{bool}$ **is** *rel-set*
parametric *rel-set-transfer* .

lemma *rel-fset-alt-def*: $\text{rel-fset } R = (\lambda A \ B. (\forall x. \exists y. x \in A \longrightarrow y \in B \wedge R \ x \ y) \wedge (\forall y. \exists x. y \in B \longrightarrow x \in A \wedge R \ x \ y))$
by *transfer'* (*metis (no-types, opaque-lifting) rel-set-def*)

lemma *finite-rel-set*:

assumes *fin*: *finite* X *finite* Z
assumes R - S : *rel-set* $(R \circ O \ S) \ X \ Z$
shows $\exists Y. \text{finite } Y \wedge \text{rel-set } R \ X \ Y \wedge \text{rel-set } S \ Y \ Z$

proof –
obtain $f\ g$ **where** $f: \forall x \in X. R\ x\ (f\ x) \wedge (\exists z \in Z. S\ (f\ x)\ z)$
and $g: \forall z \in Z. S\ (g\ z)\ z \wedge (\exists x \in X. R\ x\ (g\ z))$
using $R\text{-}S[\text{unfolded rel-set-def OO-def}]$ **by** *metis*

let $?Y = f\ ' X \cup g\ ' Z$
have *finite* $?Y$ **by** (*simp add: fin*)
moreover have *rel-set* $R\ X\ ?Y$
unfolding *rel-set-def*
using $f\ g$ **by** *clarsimp blast*
moreover have *rel-set* $S\ ?Y\ Z$
unfolding *rel-set-def*
using $f\ g$ **by** *clarsimp blast*
ultimately show *?thesis* **by** *metis*
qed

29.9.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemma *fempty-transfer* [*transfer-rule*]:
rel-fset $A\ \{\|\}\ \{\|\}$
by (*rule empty-transfer[Transfer.transferred]*)

lemma *finsert-transfer* [*transfer-rule*]:
 $(A\ ==>\ rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ A)\ finsert\ finsert$
unfolding *rel-fun-def rel-fset-alt-def* **by** *blast*

lemma *funion-transfer* [*transfer-rule*]:
 $(rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ A)\ funion\ funion$
unfolding *rel-fun-def rel-fset-alt-def* **by** *blast*

lemma *ffUnion-transfer* [*transfer-rule*]:
 $(rel\text{-}fset\ (rel\text{-}fset\ A)\ ==>\ rel\text{-}fset\ A)\ ffUnion\ ffUnion$
unfolding *rel-fun-def rel-fset-alt-def* **by** *transfer (simp, fast)*

lemma *fimage-transfer* [*transfer-rule*]:
 $((A\ ==>\ B)\ ==>\ rel\text{-}fset\ A\ ==>\ rel\text{-}fset\ B)\ fimage\ fimage$
unfolding *rel-fun-def rel-fset-alt-def* **by** *simp blast*

lemma *fBall-transfer* [*transfer-rule*]:
 $(rel\text{-}fset\ A\ ==>\ (A\ ==>\ (=))\ ==>\ (=))\ fBall\ fBall$
unfolding *rel-fset-alt-def rel-fun-def* **by** *blast*

lemma *fBex-transfer* [*transfer-rule*]:
 $(rel\text{-}fset\ A\ ==>\ (A\ ==>\ (=))\ ==>\ (=))\ fBex\ fBex$
unfolding *rel-fset-alt-def rel-fun-def* **by** *blast*

lemma *fPow-transfer* [*transfer-rule*]:
 (*rel-fset* *A* \implies *rel-fset* (*rel-fset* *A*)) *fPow* *fPow*
unfolding *rel-fun-def*
using *Pow-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*]
by *blast*

lemma *rel-fset-transfer* [*transfer-rule*]:
 ((*A* \implies *B* \implies (=)) \implies *rel-fset* *A* \implies *rel-fset* *B* \implies (=))
rel-fset *rel-fset*
unfolding *rel-fun-def*
using *rel-set-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*, **where**
A = *A* **and** *B* = *B*]
by *simp*

lemma *bind-transfer* [*transfer-rule*]:
 (*rel-fset* *A* \implies (*A* \implies *rel-fset* *B*) \implies *rel-fset* *B*) *fbind* *fbind*
unfolding *rel-fun-def*
using *bind-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by**
blast

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows (*A* \implies *rel-fset* *A* \implies (=)) ($|\in|$) ($|\in|$)
using *assms* **unfolding** *rel-fun-def* *rel-fset-alt-def* *bi-unique-def* **by** *metis*

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows (*rel-fset* *A* \implies *rel-fset* *A* \implies *rel-fset* *A*) *finter* *finter*
using *assms* **unfolding** *rel-fun-def*
using *inter-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by**
blast

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows (*rel-fset* *A* \implies *rel-fset* *A* \implies *rel-fset* *A*) ($|-|$) ($|-|$)
using *assms* **unfolding** *rel-fun-def*
using *Diff-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by**
blast

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique* *A*
shows (*rel-fset* *A* \implies *rel-fset* *A* \implies (=)) ($|\subseteq|$) ($|\subseteq|$)
using *assms* **unfolding** *rel-fun-def*
using *subset-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by**
blast

```

lemma fSup-transfer [transfer-rule]:
  bi-unique A  $\implies$  (rel-set (rel-fset A)  $\implies$  rel-fset A) Sup Sup
  unfolding rel-fun-def
  apply clarify
  apply transfer'
  using Sup-fset-transfer[unfolded rel-fun-def] by blast

lemma fInf-transfer [transfer-rule]:
  assumes bi-unique A and bi-total A
  shows (rel-set (rel-fset A)  $\implies$  rel-fset A) Inf Inf
  using assms unfolding rel-fun-def
  apply clarify
  apply transfer'
  using Inf-fset-transfer[unfolded rel-fun-def] by blast

lemma ffilter-transfer [transfer-rule]:
  assumes bi-unique A
  shows ((A  $\implies$  (=))  $\implies$  rel-fset A  $\implies$  rel-fset A) ffilter ffilter
  using assms Lifting-Set.filter-transfer
  unfolding rel-fun-def by (metis ffilter.rep-eq rel-fset.rep-eq)

lemma card-transfer [transfer-rule]:
  bi-unique A  $\implies$  (rel-fset A  $\implies$  (=)) fcard fcard
  using card-transfer unfolding rel-fun-def
  by (metis fcard.rep-eq rel-fset.rep-eq)

end

lifting-update fset.lifting
lifting-forget fset.lifting

```

29.10 BNF setup

```

context
includes fset.lifting
begin

```

```

lemma rel-fset-alt:
  rel-fset R a b  $\longleftrightarrow$  ( $\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u$ )  $\wedge$  ( $\forall t \in \text{fset } b. \exists u \in \text{fset } a. R \ u \ t$ )
  by transfer (simp add: rel-set-def)

```

```

lemma fset-to-fset: finite A  $\implies$  fset (the-inv fset A) = A
  by (metis CollectI f-the-inv-into-f fset-cases fset-cong inj-onI rangeI)

```

```

lemma rel-fset-aux:
  ( $\forall t \in \text{fset } a. \exists u \in \text{fset } b. R \ t \ u$ )  $\wedge$  ( $\forall u \in \text{fset } b. \exists t \in \text{fset } a. R \ t \ u$ )  $\longleftrightarrow$ 

```

```

((BNF-Def.Grp {a. fset a ⊆ {(a, b). R a b}} (fimage fst))-1-1 OO
BNF-Def.Grp {a. fset a ⊆ {(a, b). R a b}} (fimage snd)) a b (is ?L = ?R)
proof
  assume ?L
  define R' where R' =
    the-inv fset (Collect (case-prod R) ∩ (fset a × fset b)) (is - = the-inv fset ?L')
  have finite ?L' by (intro finite-Int[OF disjI2] finite-cartesian-product) (transfer,
simp)+
  hence *: fset R' = ?L' unfolding R'-def by (intro fset-to-fset)
  show ?R unfolding Grp-def relcompp.simps conversep.simps
  proof (intro CollectI case-prodI exI[of - a] exI[of - b] exI[of - R'] conjI refl)
    from * show a = fimage fst R' using conjunct1[OF ‹?L›]
    by (transfer, auto simp add: image-def Int-def split: prod.splits)
    from * show b = fimage snd R' using conjunct2[OF ‹?L›]
    by (transfer, auto simp add: image-def Int-def split: prod.splits)
  qed (auto simp add: *)
next
  assume ?R thus ?L unfolding Grp-def relcompp.simps conversep.simps
  using Product-Type.Collect-case-prodD by blast
qed

bnf 'a fset
  map: fimage
  sets: fset
  bd: natLeq
  wits: {||}
  rel: rel-fset
apply -
  apply transfer' apply simp
  apply transfer' apply force
  apply transfer apply force
  apply transfer' apply force
  apply (rule natLeq-card-order)
  apply (rule natLeq-cinfinite)
  apply (rule regularCard-natLeq)
  apply transfer apply (metis finite-iff-ordLess-natLeq)
  apply (fastforce simp: rel-fset-alt)
  apply (simp add: Grp-def relcompp.simps conversep.simps fun-eq-iff rel-fset-alt
    rel-fset-aux[unfolded OO-Grp-alt])
  apply transfer apply simp
done

lemma rel-fset-fset: rel-set χ (fset A1) (fset A2) = rel-fset χ A1 A2
  by (simp add: rel-fset.rep-eq)

end

declare
  fset.map-comp[simp]

```

fset.map-id[simp]
fset.set-map[simp]

29.11 Size setup

context includes *fset.lifting*

begin

lift-definition *size-fset* :: (*'a* \Rightarrow *nat*) \Rightarrow *'a fset* \Rightarrow *nat* **is** $\lambda f. \text{sum } (Suc \circ f) .$

end

instantiation *fset* :: (*type*) *size*

begin

definition *size-fset* **where**

size-fset-overloaded-def: *size-fset* = *FSet.size-fset* ($\lambda-. 0$)

instance ..

end

lemma *size-fset-simps[simp]*: *size-fset* *f* *X* = $(\sum x \in \text{fset } X. \text{Suc } (f \ x))$

by (*rule* *size-fset-def*[*THEN meta-eq-to-obj-eq*, *THEN fun-cong*, *THEN fun-cong*,
unfolded map-fun-def comp-def id-apply])

lemma *size-fset-overloaded-simps[simp]*: *size* *X* = $(\sum x \in \text{fset } X. \text{Suc } 0)$

by (*rule* *size-fset-simps*[*of* $\lambda-. 0$, *unfolded add-0-left add-0-right*,
folded size-fset-overloaded-def])

lemma *fset-size-o-map*: *inj* *f* $\implies \text{size-fset } g \circ \text{fimage } f = \text{size-fset } (g \circ f)$

unfolding *fun-eq-iff*

by (*simp* *add*: *inj-def inj-onI sum.reindex*)

setup <

BNF-LFP-Size.register-size-global **type-name** *fset* **const-name** *size-fset*

@{*thm size-fset-overloaded-def*} @{*thms size-fset-simps size-fset-overloaded-simps*}

@{*thms fset-size-o-map*}

>

lifting-update *fset.lifting*

lifting-forget *fset.lifting*

29.12 Advanced relator customization

Set vs. sum relators:

lemma *rel-set-rel-sum[simp]*:

rel-set (*rel-sum* χ φ) *A1* *A2* \longleftrightarrow

rel-set χ (*Inl* $-' A1$) (*Inl* $-' A2$) \wedge *rel-set* φ (*Inr* $-' A1$) (*Inr* $-' A2$)

(**is** $?L \longleftrightarrow ?Rl \wedge ?Rr$)

proof *safe*

assume *L*: $?L$

show $?Rl$ **unfolding** *rel-set-def Bex-def vimage-eq* **proof** *safe*

fix *l1* **assume** *Inl* *l1* $\in A1$

```

then obtain a2 where a2: a2 ∈ A2 and rel-sum χ φ (Inl l1) a2
using L unfolding rel-set-def by auto
then obtain l2 where a2 = Inl l2 ∧ χ l1 l2 by (cases a2, auto)
thus ∃ l2. Inl l2 ∈ A2 ∧ χ l1 l2 using a2 by auto
next
fix l2 assume Inl l2 ∈ A2
then obtain a1 where a1: a1 ∈ A1 and rel-sum χ φ a1 (Inl l2)
using L unfolding rel-set-def by auto
then obtain l1 where a1 = Inl l1 ∧ χ l1 l2 by (cases a1, auto)
thus ∃ l1. Inl l1 ∈ A1 ∧ χ l1 l2 using a1 by auto
qed
show ?Rr unfolding rel-set-def Bex-def vimage-eq proof safe
fix r1 assume Inr r1 ∈ A1
then obtain a2 where a2: a2 ∈ A2 and rel-sum χ φ (Inr r1) a2
using L unfolding rel-set-def by auto
then obtain r2 where a2 = Inr r2 ∧ φ r1 r2 by (cases a2, auto)
thus ∃ r2. Inr r2 ∈ A2 ∧ φ r1 r2 using a2 by auto
next
fix r2 assume Inr r2 ∈ A2
then obtain a1 where a1: a1 ∈ A1 and rel-sum χ φ a1 (Inr r2)
using L unfolding rel-set-def by auto
then obtain r1 where a1 = Inr r1 ∧ φ r1 r2 by (cases a1, auto)
thus ∃ r1. Inr r1 ∈ A1 ∧ φ r1 r2 using a1 by auto
qed
next
assume Rl: ?Rl and Rr: ?Rr
show ?L unfolding rel-set-def Bex-def vimage-eq proof safe
fix a1 assume a1: a1 ∈ A1
show ∃ a2. a2 ∈ A2 ∧ rel-sum χ φ a1 a2
proof(cases a1)
case (Inl l1) then obtain l2 where Inl l2 ∈ A2 ∧ χ l1 l2
using Rl a1 unfolding rel-set-def by blast
thus ?thesis unfolding Inl by auto
next
case (Inr r1) then obtain r2 where Inr r2 ∈ A2 ∧ φ r1 r2
using Rr a1 unfolding rel-set-def by blast
thus ?thesis unfolding Inr by auto
qed
next
fix a2 assume a2: a2 ∈ A2
show ∃ a1. a1 ∈ A1 ∧ rel-sum χ φ a1 a2
proof(cases a2)
case (Inl l2) then obtain l1 where Inl l1 ∈ A1 ∧ χ l1 l2
using Rl a2 unfolding rel-set-def by blast
thus ?thesis unfolding Inl by auto
next
case (Inr r2) then obtain r1 where Inr r1 ∈ A1 ∧ φ r1 r2
using Rr a2 unfolding rel-set-def by blast
thus ?thesis unfolding Inr by auto

```


qed
 qed
 qed

29.12.1 Countability

lemma *exists-fset-of-list*: $\exists xs. \text{fset-of-list } xs = S$
including *fset.lifting*
by *transfer (rule finite-list)*

lemma *fset-of-list-surj*[*simp, intro*]: *surj fset-of-list*
by (*metis exists-fset-of-list surj-def*)

instance *fset* :: (*countable*) *countable*

proof

obtain *to-nat* :: 'a list \Rightarrow nat **where** *inj to-nat*
by (*metis ex-inj*)
moreover have *inj (inv fset-of-list)*
using *fset-of-list-surj* **by** (*rule surj-imp-inj-inv*)
ultimately have *inj (to-nat \circ inv fset-of-list)*
by (*rule inj-compose*)
thus $\exists \text{to-nat}::'a \text{fset} \Rightarrow \text{nat. } \text{inj to-nat}$
by *auto*

qed

29.13 Quickcheck setup

Setup adapted from sets.

notation *Quickcheck-Exhaustive.orelse* (**infixr** <orelse> 55)

context

includes *term-syntax*

begin

definition [*code-unfold*]:

valterm-femptyset = *Code-Evaluation.valtermify* ($\{\|\}$:: ('a :: *typerep*) *fset*)

definition [*code-unfold*]:

valtermify-finsert *x s* = *Code-Evaluation.valtermify* *finsert* $\{\cdot\}$ (*x* :: ('a :: *typerep* *
 -)) $\{\cdot\}$ *s*

end

instantiation *fset* :: (*exhaustive*) *exhaustive*

begin

fun *exhaustive-fset* **where**

exhaustive-fset *f i* = (*if i = 0 then None else* (*f* $\{\|\}$ *orelse* *exhaustive-fset* ($\lambda A. f$
A *orelse* *Quickcheck-Exhaustive.exhaustive* ($\lambda x. \text{if } x \in A \text{ then None else } f$ (*finsert*

$x A)) (i - 1)) (i - 1)))$

instance ..

end

instantiation *fset* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-fset* **where**

full-exhaustive-fset *f* *i* = (*if* *i* = 0 *then* None *else* (*f* *valterm-femptyset* *orelse* *full-exhaustive-fset* ($\lambda A. f A$ *orelse* Quickcheck-Exhaustive.full-exhaustive ($\lambda x. \text{if } fst\ x \in |fst\ A| \text{ then } None \text{ else } f (valtermify-finsert\ x\ A)) (i - 1)) (i - 1)))$

instance ..

end

no-notation Quickcheck-Exhaustive.orelse (**infixr** <orelse> 55)

instantiation *fset* :: (*random*) *random*
begin

context

includes *state-combinator-syntax*

begin

fun *random-aux-fset* :: *natural* \Rightarrow *natural* \Rightarrow *natural* \times *natural* \Rightarrow (*a fset* \times (*unit* \Rightarrow *term*)) \times *natural* \times *natural* **where**

random-aux-fset 0 *j* = Quickcheck-Random.collapse (*Random.select-weight* [(1, *Pair valterm-femptyset*)]) |

random-aux-fset (*Code-Numeral.Suc* *i*) *j* =

Quickcheck-Random.collapse (*Random.select-weight*

[(1, *Pair valterm-femptyset*),

(*Code-Numeral.Suc* *i*,

Quickcheck-Random.random *j* $\circ \rightarrow (\lambda x. \text{random-aux-fset } i\ j \circ \rightarrow (\lambda s. \text{Pair}$

(*valtermify-finsert* *x* *s*))))])

lemma [*code*]:

random-aux-fset *i* *j* =

Quickcheck-Random.collapse (*Random.select-weight* [(1, *Pair valterm-femptyset*),

(*i*, Quickcheck-Random.random *j* $\circ \rightarrow (\lambda x. \text{random-aux-fset } (i - 1)\ j \circ \rightarrow (\lambda s. \text{Pair}$

Pair (*valtermify-finsert* *x* *s*))))])

proof (*induct* *i* *rule*: *natural.induct*)

case zero

show ?*case* **by** (*subst select-weight-drop-zero[symmetric]*) (*simp add: less-natural-def*)

next

case (*Suc* *i*)

show ?*case* **by** (*simp only: random-aux-fset.simps Suc-natural-minus-one*)

qed

definition *random-fset* *i* = *random-aux-fset* *i* *i*

instance ..

end

end

29.14 Code Generation Setup

The following *code-unfold* lemmas are so the pre-processor of the code generator will perform conversions like, e.g., $(x \in f \mid f \mid f \text{ set-of-list } xs) = (x \in f \text{ ' set } xs)$.

declare

ffilter.rep-eq[*code-unfold*]
fimage.rep-eq[*code-unfold*]
finsert.rep-eq[*code-unfold*]
fset-of-list.rep-eq[*code-unfold*]
inf-fset.rep-eq[*code-unfold*]
minus-fset.rep-eq[*code-unfold*]
sup-fset.rep-eq[*code-unfold*]
uminus-fset.rep-eq[*code-unfold*]

end

30 Type of finite maps defined as a subtype of maps

theory *Finite-Map*

imports *FSet* *AList* *Conditional-Parametricity*

abbrevs (= = \subseteq_f

begin

30.1 Auxiliary constants and lemmas over *map*

parametric-constant *map-add-transfer*[*transfer-rule*]: *map-add-def*

parametric-constant *map-of-transfer*[*transfer-rule*]: *map-of-def*

context includes *lifting-syntax* **begin**

abbreviation *rel-map* :: (*'b* \Rightarrow *'c* \Rightarrow *bool*) \Rightarrow (*'a* \rightarrow *'b*) \Rightarrow (*'a* \rightarrow *'c*) \Rightarrow *bool* **where**
rel-map *f* \equiv (=) \implies *rel-option* *f*

lemma *ran-transfer*[*transfer-rule*]: (*rel-map* *A* \implies *rel-set* *A*) *ran* *ran*

proof

fix *m* *n*

```

assume rel-map A m n
show rel-set A (ran m) (ran n)
proof (rule rel-setI)
  fix x
  assume  $x \in \text{ran } m$ 
  then obtain a where  $m \ a = \text{Some } x$ 
    unfolding ran-def by auto

  have rel-option A (m a) (n a)
    using  $\langle \text{rel-map } A \ m \ n \rangle$ 
    by (auto dest: rel-funD)
  then obtain y where  $n \ a = \text{Some } y \ A \ x \ y$ 
    unfolding  $\langle m \ a = - \rangle$ 
    by cases auto
  then show  $\exists y \in \text{ran } n. \ A \ x \ y$ 
    unfolding ran-def by blast
next
  fix y
  assume  $y \in \text{ran } n$ 
  then obtain a where  $n \ a = \text{Some } y$ 
    unfolding ran-def by auto

  have rel-option A (m a) (n a)
    using  $\langle \text{rel-map } A \ m \ n \rangle$ 
    by (auto dest: rel-funD)
  then obtain x where  $m \ a = \text{Some } x \ A \ x \ y$ 
    unfolding  $\langle n \ a = - \rangle$ 
    by cases auto
  then show  $\exists x \in \text{ran } m. \ A \ x \ y$ 
    unfolding ran-def by blast
qed
qed

lemma ran-alt-def:  $\text{ran } m = (\text{the } \circ m) \ ` \ \text{dom } m$ 
unfolding ran-def dom-def by force

parametric-constant dom-transfer[transfer-rule]: dom-def

definition map-upd ::  $'a \Rightarrow 'b \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$  where
map-upd k v m = m(k  $\mapsto$  v)

parametric-constant map-upd-transfer[transfer-rule]: map-upd-def

definition map-filter ::  $('a \Rightarrow \text{bool}) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$  where
map-filter P m = ( $\lambda x. \text{if } P \ x \text{ then } m \ x \text{ else } \text{None}$ )

parametric-constant map-filter-transfer[transfer-rule]: map-filter-def

lemma map-filter-map-of[simp]:  $\text{map-filter } P \ (\text{map-of } m) = \text{map-of } [(k, -) \leftarrow m].$ 

```

```

P k]
proof
  fix x
  show map-filter P (map-of m) x = map-of [(k, -) ← m. P k] x
    by (induct m) (auto simp: map-filter-def)
qed

```

```

lemma map-filter-finite[intro]:
  assumes finite (dom m)
  shows finite (dom (map-filter P m))
proof -
  from assms have ⟨finite (dom (λx. if P x then m x else None))⟩
    by (rule rev-finite-subset) (auto split: if-splits)
  then show ?thesis
    by (simp add: map-filter-def)
qed

```

definition $\text{map-drop} :: 'a \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **where**
 $\text{map-drop } a = \text{map-filter } (\lambda a'. a' \neq a)$

parametric-constant $\text{map-drop-transfer[transfer-rule]}: \text{map-drop-def}$

definition $\text{map-drop-set} :: 'a \text{ set} \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **where**
 $\text{map-drop-set } A = \text{map-filter } (\lambda a. a \notin A)$

parametric-constant $\text{map-drop-set-transfer[transfer-rule]}: \text{map-drop-set-def}$

definition $\text{map-restrict-set} :: 'a \text{ set} \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b)$ **where**
 $\text{map-restrict-set } A = \text{map-filter } (\lambda a. a \in A)$

parametric-constant $\text{map-restrict-set-transfer[transfer-rule]}: \text{map-restrict-set-def}$

definition $\text{map-pred} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \rightarrow 'b) \Rightarrow \text{bool}$ **where**
 $\text{map-pred } P \ m \longleftrightarrow (\forall x. \text{case } m \ x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } y \Rightarrow P \ x \ y)$

parametric-constant $\text{map-pred-transfer[transfer-rule]}: \text{map-pred-def}$

definition $\text{rel-map-on-set} :: 'a \text{ set} \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'c)$
 $\Rightarrow \text{bool}$ **where**
 $\text{rel-map-on-set } S \ P = \text{eq-onp } (\lambda x. x \in S) ==> \text{rel-option } P$

definition $\text{set-of-map} :: ('a \rightarrow 'b) \Rightarrow ('a \times 'b) \text{ set}$ **where**
 $\text{set-of-map } m = \{(k, v) \mid k \ v. \ m \ k = \text{Some } v\}$

lemma $\text{set-of-map-alt-def}: \text{set-of-map } m = (\lambda k. (k, \text{the } (m \ k))) \text{ ` dom } m$
unfolding $\text{set-of-map-def dom-def}$
by *auto*

lemma $\text{set-of-map-finite}: \text{finite } (\text{dom } m) \Longrightarrow \text{finite } (\text{set-of-map } m)$

unfolding *set-of-map-alt-def*
by *auto*

lemma *set-of-map-inj: inj set-of-map*
proof
 fix *x y*
 assume *set-of-map x = set-of-map y*
 hence $(x\ a = \text{Some } b) = (y\ a = \text{Some } b)$ **for** *a b*
 unfolding *set-of-map-def* **by** *auto*
 hence $x\ k = y\ k$ **for** *k*
 by (*metis not-None-eq*)
 thus $x = y$ **..**
qed

lemma *dom-comp: dom (m \circ_m n) \subseteq dom n*
unfolding *map-comp-def dom-def*
by (*auto split: option.splits*)

lemma *dom-comp-finite: finite (dom n) \implies finite (dom (map-comp m n))*
by (*metis finite-subset dom-comp*)

parametric-constant *map-comp-transfer[transfer-rule]: map-comp-def*

end

30.2 Abstract characterisation

typedef (*'a, 'b*) *fmap* = $\{m. \text{finite } (\text{dom } m)\}$ **::** (*'a \rightarrow 'b*) *set*
 morphisms *fmlookup Abs-fmap*
proof
 show *Map.empty* $\in \{m. \text{finite } (\text{dom } m)\}$
 by *auto*
qed

setup-lifting *type-definition-fmap*

lemma *dom-fmlookup-finite[intro, simp]: finite (dom (fmlookup m))*
using *fmap.fmlookup* **by** *auto*

lemma *fmap-ext:*
 assumes $\bigwedge x. \text{fmlookup } m\ x = \text{fmlookup } n\ x$
 shows $m = n$
using *assms*
by *transfer' auto*

30.3 Operations

context
 includes *fset.lifting*
begin

lift-definition $fmran :: ('a, 'b) fmap \Rightarrow 'b fset$
is ran
parametric $ran-transfer$
by $(rule\ finite-ran)$

lemma $fmlookup-ran-iff: y \in | fmran\ m \longleftrightarrow (\exists x. fmlookup\ m\ x = Some\ y)$
by $transfer' (auto simp: ran-def)$

lemma $fmranI: fmlookup\ m\ x = Some\ y \Longrightarrow y \in | fmran\ m$ **by** $(auto simp: fmlookup-ran-iff)$

lemma $fmranE[elim]:$
assumes $y \in | fmran\ m$
obtains x **where** $fmlookup\ m\ x = Some\ y$
using $assms$ **by** $(auto simp: fmlookup-ran-iff)$

lift-definition $fmdom :: ('a, 'b) fmap \Rightarrow 'a fset$
is dom
parametric $dom-transfer$
.

lemma $fmlookup-dom-iff: x \in | fmdom\ m \longleftrightarrow (\exists a. fmlookup\ m\ x = Some\ a)$
by $transfer' auto$

lemma $fmdom-notI: fmlookup\ m\ x = None \Longrightarrow x \notin | fmdom\ m$ **by** $(simp add: fmlookup-dom-iff)$

lemma $fmdomI: fmlookup\ m\ x = Some\ y \Longrightarrow x \in | fmdom\ m$ **by** $(simp add: fmlookup-dom-iff)$

lemma $fmdom-notD[dest]: x \notin | fmdom\ m \Longrightarrow fmlookup\ m\ x = None$ **by** $(simp add: fmlookup-dom-iff)$

lemma $fmdomE[elim]:$
assumes $x \in | fmdom\ m$
obtains y **where** $fmlookup\ m\ x = Some\ y$
using $assms$ **by** $(auto simp: fmlookup-dom-iff)$

lift-definition $fmdom' :: ('a, 'b) fmap \Rightarrow 'a set$
is dom
parametric $dom-transfer$
.

lemma $fmlookup-dom'-iff: x \in fmdom'\ m \longleftrightarrow (\exists a. fmlookup\ m\ x = Some\ a)$
by $transfer' auto$

lemma $fmdom'-notI: fmlookup\ m\ x = None \Longrightarrow x \notin fmdom'\ m$ **by** $(simp add: fmlookup-dom'-iff)$

lemma $fmdom'I: fmlookup\ m\ x = Some\ y \Longrightarrow x \in fmdom'\ m$ **by** $(simp add: fmlookup-dom'-iff)$

lemma *fmdom'-notD*[*dest*]: $x \notin \text{fmdom}' m \implies \text{fmlookup } m x = \text{None}$ **by** (*simp add: fmlookup-dom'-iff*)

lemma *fmdom'E*[*elim*]:
assumes $x \in \text{fmdom}' m$
obtains $x y$ **where** $\text{fmlookup } m x = \text{Some } y$
using *assms* **by** (*auto simp: fmlookup-dom'-iff*)

lemma *fmdom'-alt-def*: $\text{fmdom}' m = \text{fset } (\text{fmdom } m)$
by *transfer' force*

lemma *finite-fmdom'*[*simp*]: *finite* ($\text{fmdom}' m$)
unfolding *fmdom'-alt-def* **by** *simp*

lemma *dom-fmlookup*[*simp*]: $\text{dom } (\text{fmlookup } m) = \text{fmdom}' m$
by *transfer' simp*

lift-definition *fmempty* :: (*'a*, *'b*) *fmap*
is *Map.empty*
by *simp*

lemma *fmempty-lookup*[*simp*]: $\text{fmlookup } \text{fmempty } x = \text{None}$
by *transfer' simp*

lemma *fmdom-empty*[*simp*]: $\text{fmdom } \text{fmempty} = \{\}\}$ **by** *transfer' simp*
lemma *fmdom'-empty*[*simp*]: $\text{fmdom}' \text{fmempty} = \{\}$ **by** *transfer' simp*
lemma *fmran-empty*[*simp*]: $\text{fmran } \text{fmempty} = \text{fempty}$ **by** *transfer' (auto simp: ran-def map-filter-def)*

lift-definition *fmupd* :: *'a* \Rightarrow *'b* \Rightarrow (*'a*, *'b*) *fmap* \Rightarrow (*'a*, *'b*) *fmap*
is *map-upd*
parametric *map-upd-transfer*
unfolding *map-upd-def*[*abs-def*]
by *simp*

lemma *fmupd-lookup*[*simp*]: $\text{fmlookup } (\text{fmupd } a b m) a' = (\text{if } a = a' \text{ then } \text{Some } b \text{ else } \text{fmlookup } m a')$
by *transfer' (auto simp: map-upd-def)*

lemma *fmdom-fmupd*[*simp*]: $\text{fmdom } (\text{fmupd } a b m) = \text{finsert } a (\text{fmdom } m)$ **by** *transfer (simp add: map-upd-def)*
lemma *fmdom'-fmupd*[*simp*]: $\text{fmdom}' (\text{fmupd } a b m) = \text{insert } a (\text{fmdom}' m)$ **by** *transfer (simp add: map-upd-def)*

lemma *fmupd-reorder-neq*:
assumes $a \neq b$
shows $\text{fmupd } a x (\text{fmupd } b y m) = \text{fmupd } b y (\text{fmupd } a x m)$
using *assms*
by *transfer' (auto simp: map-upd-def)*

lemma *fmupd-idem[simp]*: $\text{fmupd } a \ x \ (\text{fmupd } a \ y \ m) = \text{fmupd } a \ x \ m$
by *transfer'* (*auto simp: map-upd-def*)

lift-definition *fmfilter* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \text{fmap} \Rightarrow ('a, 'b) \text{fmap}$
is *map-filter*
parametric *map-filter-transfer*
by *auto*

lemma *fmdom-filter[simp]*: $\text{fmdom } (\text{fmfilter } P \ m) = \text{ffilter } P \ (\text{fmdom } m)$
by *transfer'* (*auto simp: map-filter-def split: if-splits*)

lemma *fmdom'-filter[simp]*: $\text{fmdom}' (\text{fmfilter } P \ m) = \text{Set.filter } P \ (\text{fmdom}' \ m)$
by *transfer'* (*auto simp: map-filter-def split: if-splits*)

lemma *fmlookup-filter[simp]*: $\text{fmlookup } (\text{fmfilter } P \ m) \ x = (\text{if } P \ x \ \text{then } \text{fmlookup } m \ x \ \text{else } \text{None})$
by *transfer'* (*auto simp: map-filter-def*)

lemma *fmfilter-empty[simp]*: $\text{fmfilter } P \ \text{fmempty} = \text{fmempty}$
by *transfer'* (*auto simp: map-filter-def*)

lemma *fmfilter-true[simp]*:
assumes $\bigwedge x \ y. \text{fmlookup } m \ x = \text{Some } y \Longrightarrow P \ x$
shows $\text{fmfilter } P \ m = m$
proof (*rule fmap-ext*)
fix x
have $\text{fmlookup } m \ x = \text{None}$ **if** $\neg P \ x$
using *that assms* **by** *fastforce*
then show $\text{fmlookup } (\text{fmfilter } P \ m) \ x = \text{fmlookup } m \ x$
by *simp*
qed

lemma *fmfilter-false[simp]*:
assumes $\bigwedge x \ y. \text{fmlookup } m \ x = \text{Some } y \Longrightarrow \neg P \ x$
shows $\text{fmfilter } P \ m = \text{fmempty}$
using *assms* **by** *transfer'* (*fastforce simp: map-filter-def*)

lemma *fmfilter-comp[simp]*: $\text{fmfilter } P \ (\text{fmfilter } Q \ m) = \text{fmfilter } (\lambda x. P \ x \wedge Q \ x) \ m$
by *transfer'* (*auto simp: map-filter-def*)

lemma *fmfilter-comm*: $\text{fmfilter } P \ (\text{fmfilter } Q \ m) = \text{fmfilter } Q \ (\text{fmfilter } P \ m)$
unfolding *fmfilter-comp* **by** *meson*

lemma *fmfilter-cong[cong]*:
assumes $\bigwedge x \ y. \text{fmlookup } m \ x = \text{Some } y \Longrightarrow P \ x = Q \ x$
shows $\text{fmfilter } P \ m = \text{fmfilter } Q \ m$
proof (*rule fmap-ext*)

```

fix x
have fmlookup m x = None if P x ≠ Q x
  using that assms by fastforce
then show fmlookup (fmfilter P m) x = fmlookup (fmfilter Q m) x
  by auto
qed

```

```

lemma fmfilter-cong'[fundef-cong]:
  assumes m = n ∧ x. x ∈ fmdom' m ⇒ P x = Q x
  shows fmfilter P m = fmfilter Q n
using assms(2) unfolding assms(1)
by (rule fmfilter-cong) (metis fmdom'I)

```

```

lemma fmfilter-upd[simp]:
  fmfilter P (fmupd x y m) = (if P x then fmupd x y (fmfilter P m) else fmfilter P
m)
by transfer' (auto simp: map-upd-def map-filter-def)

```

```

lift-definition fmdrop :: 'a ⇒ ('a, 'b) fmap ⇒ ('a, 'b) fmap
  is map-drop
  parametric map-drop-transfer
unfolding map-drop-def by auto

```

```

lemma fmdrop-lookup[simp]: fmlookup (fmdrop a m) a = None
by transfer' (auto simp: map-drop-def map-filter-def)

```

```

lift-definition fmdrop-set :: 'a set ⇒ ('a, 'b) fmap ⇒ ('a, 'b) fmap
  is map-drop-set
  parametric map-drop-set-transfer
unfolding map-drop-set-def by auto

```

```

lift-definition fmdrop-fset :: 'a fset ⇒ ('a, 'b) fmap ⇒ ('a, 'b) fmap
  is map-drop-set
  parametric map-drop-set-transfer
unfolding map-drop-set-def by auto

```

```

lift-definition fmrestrict-set :: 'a set ⇒ ('a, 'b) fmap ⇒ ('a, 'b) fmap
  is map-restrict-set
  parametric map-restrict-set-transfer
unfolding map-restrict-set-def by auto

```

```

lift-definition fmrestrict-fset :: 'a fset ⇒ ('a, 'b) fmap ⇒ ('a, 'b) fmap
  is map-restrict-set
  parametric map-restrict-set-transfer
unfolding map-restrict-set-def by auto

```

```

lemma fmfilter-alt-defs:
  fmdrop a = fmfilter (λa'. a' ≠ a)
  fmdrop-set A = fmfilter (λa. a ∉ A)

```

$fmdrop\text{-}fset\ B = fmfilt\ (\lambda a. a \notin B)$
 $fmrestrict\text{-}set\ A = fmfilt\ (\lambda a. a \in A)$
 $fmrestrict\text{-}fset\ B = fmfilt\ (\lambda a. a \in B)$
by (transfer'; simp add: map-drop-def map-drop-set-def map-restrict-set-def)+

lemma $fmdom\text{-}drop[simp]$: $fmdom\ (fmdrop\ a\ m) = fmdom\ m - \{a\}$ **unfolding**
 $fmfilter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdom'\text{-}drop[simp]$: $fmdom'\ (fmdrop\ a\ m) = fmdom'\ m - \{a\}$ **unfolding**
 $fmfilter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdom'\text{-}drop\text{-}set[simp]$: $fmdom'\ (fmdrop\text{-}set\ A\ m) = fmdom'\ m - A$ **un-**
folding $fmfilter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdom\text{-}drop\text{-}fset[simp]$: $fmdom\ (fmdrop\text{-}fset\ A\ m) = fmdom\ m - A$ **un-**
folding $fmfilter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdom'\text{-}restrict\text{-}set$: $fmdom'\ (fmrestrict\text{-}set\ A\ m) \subseteq A$ **unfolding** $fmfil-$
 $ter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdom\text{-}restrict\text{-}fset$: $fmdom\ (fmrestrict\text{-}fset\ A\ m) \subseteq A$ **unfolding** $fmfil-$
 $ter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdrop\text{-}fmupd$: $fmdrop\ x\ (fmupd\ y\ z\ m) = (if\ x = y\ then\ fmdrop\ x\ m\ else\ fmupd\ y\ z\ (fmdrop\ x\ m))$

by transfer' (auto simp: map-drop-def map-filter-def map-upd-def)

lemma $fmdrop\text{-}idle$: $x \notin fmdom\ B \implies fmdrop\ x\ B = B$

by transfer' (auto simp: map-drop-def map-filter-def)

lemma $fmdrop\text{-}idle'$: $x \notin fmdom'\ B \implies fmdrop\ x\ B = B$

by transfer' (auto simp: map-drop-def map-filter-def)

lemma $fmdrop\text{-}fmupd\text{-}same$: $fmdrop\ x\ (fmupd\ x\ y\ m) = fmdrop\ x\ m$

by transfer' (auto simp: map-drop-def map-filter-def map-upd-def)

lemma $fmdom'\text{-}restrict\text{-}set\text{-}precise$: $fmdom'\ (fmrestrict\text{-}set\ A\ m) = fmdom'\ m \cap A$

unfolding $fmfilter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdom'\text{-}restrict\text{-}fset\text{-}precise$: $fmdom\ (fmrestrict\text{-}fset\ A\ m) = fmdom\ m \cap A$

unfolding $fmfilter\text{-}alt\text{-}defs$ **by** auto

lemma $fmdom'\text{-}drop\text{-}fset[simp]$: $fmdom'\ (fmdrop\text{-}fset\ A\ m) = fmdom'\ m - fset\ A$

unfolding $fmfilter\text{-}alt\text{-}defs$ **by** transfer' (auto simp: map-filter-def split: if-splits)

lemma $fmdom'\text{-}restrict\text{-}fset$: $fmdom'\ (fmrestrict\text{-}fset\ A\ m) \subseteq fset\ A$

unfolding $fmfilter\text{-}alt\text{-}defs$ **by** transfer' (auto simp: map-filter-def)

lemma $fmlookup\text{-}drop[simp]$:

$fmlookup\ (fmdrop\ a\ m)\ x = (if\ x \neq a\ then\ fmlookup\ m\ x\ else\ None)$

unfolding $fmfilter\text{-}alt\text{-}defs$ **by** simp

lemma *fmlookup-drop-set[simp]*:
 $\text{fmlookup } (\text{fmdrop-set } A \ m) \ x = (\text{if } x \notin A \text{ then fmlookup } m \ x \text{ else None})$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmlookup-drop-fset[simp]*:
 $\text{fmlookup } (\text{fmdrop-fset } A \ m) \ x = (\text{if } x \notin A \text{ then fmlookup } m \ x \text{ else None})$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmlookup-restrict-set[simp]*:
 $\text{fmlookup } (\text{fmrestrict-set } A \ m) \ x = (\text{if } x \in A \text{ then fmlookup } m \ x \text{ else None})$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmlookup-restrict-fset[simp]*:
 $\text{fmlookup } (\text{fmrestrict-fset } A \ m) \ x = (\text{if } x \in A \text{ then fmlookup } m \ x \text{ else None})$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-dom[simp]*: $\text{fmrestrict-set } (\text{fmdom}' \ m) \ m = m$
by (*rule fmap-ext*) *auto*

lemma *fmrestrict-fset-dom[simp]*: $\text{fmrestrict-fset } (\text{fmdom } m) \ m = m$
by (*rule fmap-ext*) *auto*

lemma *fmdrop-empty[simp]*: $\text{fmdrop } a \ \text{fmempty} = \text{fmempty}$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-empty[simp]*: $\text{fmdrop-set } A \ \text{fmempty} = \text{fmempty}$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-empty[simp]*: $\text{fmdrop-fset } A \ \text{fmempty} = \text{fmempty}$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-fmdom[simp]*: $\text{fmdrop-fset } (\text{fmdom } A) \ A = \text{fmempty}$
by *transfer'* (*auto simp: map-drop-set-def map-filter-def*)

lemma *fmdrop-set-fmdom[simp]*: $\text{fmdrop-set } (\text{fmdom}' \ A) \ A = \text{fmempty}$
by *transfer'* (*auto simp: map-drop-set-def map-filter-def*)

lemma *fmrestrict-set-empty[simp]*: $\text{fmrestrict-set } A \ \text{fmempty} = \text{fmempty}$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-empty[simp]*: $\text{fmrestrict-fset } A \ \text{fmempty} = \text{fmempty}$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-null[simp]*: $\text{fmdrop-set } \{\} \ m = m$
by (*rule fmap-ext*) *auto*

lemma *fmdrop-fset-null[simp]*: $\text{fmdrop-fset } \{\|\} \ m = m$
by (*rule fmap-ext*) *auto*

lemma *fmdrop-set-single[simp]*: $\text{fmdrop-set } \{a\} \ m = \text{fmdrop } a \ m$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-single[simp]*: $\text{fmdrop-fset } \{|a|\} \ m = \text{fmdrop } a \ m$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-null[simp]*: $\text{fmrestrict-set } \{\} \ m = \text{fmempty}$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-null[simp]*: $\text{fmrestrict-fset } \{|\} \ m = \text{fmempty}$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-comm*: $\text{fmdrop } a \ (\text{fmdrop } b \ m) = \text{fmdrop } b \ (\text{fmdrop } a \ m)$
unfolding *fmfilter-alt-defs* **by** (*rule fmfilter-comm*)

lemma *fmdrop-set-insert[simp]*: $\text{fmdrop-set } (\text{insert } x \ S) \ m = \text{fmdrop } x \ (\text{fmdrop-set } S \ m)$
by (*rule fmap-ext*) *auto*

lemma *fmdrop-fset-insert[simp]*: $\text{fmdrop-fset } (\text{finsert } x \ S) \ m = \text{fmdrop } x \ (\text{fmdrop-fset } S \ m)$
by (*rule fmap-ext*) *auto*

lemma *fmrestrict-set-twice[simp]*: $\text{fmrestrict-set } S \ (\text{fmrestrict-set } T \ m) = \text{fmrestrict-set } (S \cap T) \ m$
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmrestrict-fset-twice[simp]*: $\text{fmrestrict-fset } S \ (\text{fmrestrict-fset } T \ m) = \text{fmrestrict-fset } (S \ |\cap| \ T) \ m$
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmrestrict-set-drop[simp]*: $\text{fmrestrict-set } S \ (\text{fmdrop } b \ m) = \text{fmrestrict-set } (S - \{b\}) \ m$
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmrestrict-fset-drop[simp]*: $\text{fmrestrict-fset } S \ (\text{fmdrop } b \ m) = \text{fmrestrict-fset } (S - \{| \ b \ | \}) \ m$
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-fmrestrict-set[simp]*: $\text{fmdrop } b \ (\text{fmrestrict-set } S \ m) = \text{fmrestrict-set } (S - \{b\}) \ m$
by (*rule fmap-ext*) *auto*

lemma *fmdrop-fmrestrict-fset[simp]*: $\text{fmdrop } b \ (\text{fmrestrict-fset } S \ m) = \text{fmrestrict-fset } (S - \{| \ b \ | \}) \ m$
by (*rule fmap-ext*) *auto*

lemma *fmdrop-idem[simp]*: $\text{fmdrop } a \ (\text{fmdrop } a \ m) = \text{fmdrop } a \ m$
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-set-twice*[simp]: *fmdrop-set* *S* (*fmdrop-set* *T* *m*) = *fmdrop-set* (*S* \cup *T*) *m*

unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-fset-twice*[simp]: *fmdrop-fset* *S* (*fmdrop-fset* *T* *m*) = *fmdrop-fset* (*S* \cup *T*) *m*

unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-set-fmdrop*[simp]: *fmdrop-set* *S* (*fmdrop* *b* *m*) = *fmdrop-set* (*insert* *b* *S*) *m*

by (*rule* *fmap-ext*) *auto*

lemma *fmdrop-fset-fmdrop*[simp]: *fmdrop-fset* *S* (*fmdrop* *b* *m*) = *fmdrop-fset* (*insert* *b* *S*) *m*

by (*rule* *fmap-ext*) *auto*

lift-definition *fmadd* :: (*'a*, *'b*) *fmap* \Rightarrow (*'a*, *'b*) *fmap* \Rightarrow (*'a*, *'b*) *fmap* (**infixl** $\langle ++_f \rangle$ 100)

is *map-add*

parametric *map-add-transfer*

by *simp*

lemma *fmlookup-add*[simp]:

fmlookup (*m* $++_f$ *n*) *x* = (if *x* \in *fmdom* *n* then *fmlookup* *n* *x* else *fmlookup* *m* *x*)

by *transfer'* (*auto* *simp*: *map-add-def* *split*: *option.splits*)

lemma *fmdom-add*[simp]: *fmdom* (*m* $++_f$ *n*) = *fmdom* *m* \cup *fmdom* *n* **by** *transfer'* *auto*

lemma *fmdom'-add*[simp]: *fmdom'* (*m* $++_f$ *n*) = *fmdom'* *m* \cup *fmdom'* *n* **by** *transfer'* *auto*

lemma *fmadd-drop-left-dom*: *fmdrop-fset* (*fmdom* *n*) *m* $++_f$ *n* = *m* $++_f$ *n*

by (*rule* *fmap-ext*) *auto*

lemma *fmadd-restrict-right-dom*: *fmrestrict-fset* (*fmdom* *n*) (*m* $++_f$ *n*) = *n*

by (*rule* *fmap-ext*) *auto*

lemma *fmfilter-add-distrib*[simp]: *fmfilter* *P* (*m* $++_f$ *n*) = *fmfilter* *P* *m* $++_f$ *fmfilter* *P* *n*

by *transfer'* (*auto* *simp*: *map-filter-def* *map-add-def*)

lemma *fmdrop-add-distrib*[simp]: *fmdrop* *a* (*m* $++_f$ *n*) = *fmdrop* *a* *m* $++_f$ *fmdrop* *a* *n*

unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-add-distrib*[simp]: *fmdrop-set* *A* (*m* $++_f$ *n*) = *fmdrop-set* *A* *m* $++_f$ *fmdrop-set* *A* *n*

unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-add-distrib*[*simp*]: *fmdrop-fset A (m ++_f n) = fmdrop-fset A m ++_f fmdrop-fset A n*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-add-distrib*[*simp*]:
fmrestrict-set A (m ++_f n) = fmrestrict-set A m ++_f fmrestrict-set A n
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-add-distrib*[*simp*]:
fmrestrict-fset A (m ++_f n) = fmrestrict-fset A m ++_f fmrestrict-fset A n
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmadd-empty*[*simp*]: *fmempty ++_f m = m m ++_f fmempty = m*
by (*transfer'*; *auto*)+

lemma *fmadd-idempotent*[*simp*]: *m ++_f m = m*
by *transfer'* (*auto simp: map-add-def split: option.splits*)

lemma *fmadd-assoc*[*simp*]: *m ++_f (n ++_f p) = m ++_f n ++_f p*
by *transfer'* *simp*

lemma *fmadd-fmupd*[*simp*]: *m ++_f fmupd a b n = fmupd a b (m ++_f n)*
by (*rule fmap-ext*) *simp*

lift-definition *fmprered* :: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*'a*, *'b*) *fmap* \Rightarrow *bool*
is *map-pred*
parametric *map-pred-transfer*
.

lemma *fmpreredI*[*intro*]:
assumes $\bigwedge x y. \text{fmlookup } m \ x = \text{Some } y \Longrightarrow P \ x \ y$
shows *fmprered P m*
using *assms*
by *transfer'* (*auto simp: map-pred-def split: option.splits*)

lemma *fmpreredD*[*dest*]: *fmprered P m \Longrightarrow fmlookup m x = Some y \Longrightarrow P x y*
by *transfer'* (*auto simp: map-pred-def split: option.split-asm*)

lemma *fmprered-iff*: *fmprered P m \longleftrightarrow ($\forall x y. \text{fmlookup } m \ x = \text{Some } y \longrightarrow P \ x \ y$)*
by *auto*

lemma *fmprered-alt-def*: *fmprered P m \longleftrightarrow fBall (fmdom m) ($\lambda x. P \ x \ (\text{the } (\text{fmlookup } m \ x))$)*
unfolding *fmprered-iff*
using *fmdomI* **by** *fastforce*

lemma *fmprered-mono-strong*:

assumes $\bigwedge x y. \text{fmlookup } m \ x = \text{Some } y \implies P \ x \ y \implies Q \ x \ y$
shows $\text{fmpred } P \ m \implies \text{fmpred } Q \ m$
using *assms* **unfolding** *fmpred-iff* **by** *auto*

lemma *fmpred-mono*[*mono*]: $P \leq Q \implies \text{fmpred } P \leq \text{fmpred } Q$
by *auto*

lemma *fmpred-empty*[*intro!*, *simp*]: $\text{fmpred } P \ \text{fmempty}$
by *auto*

lemma *fmpred-upd*[*intro*]: $\text{fmpred } P \ m \implies P \ x \ y \implies \text{fmpred } P \ (\text{fmupd } x \ y \ m)$
by *transfer'* (*auto simp: map-pred-def map-upd-def*)

lemma *fmpred-updD*[*dest*]: $\text{fmpred } P \ (\text{fmupd } x \ y \ m) \implies P \ x \ y$
by *auto*

lemma *fmpred-add*[*intro*]: $\text{fmpred } P \ m \implies \text{fmpred } P \ n \implies \text{fmpred } P \ (m \ ++_f \ n)$
by *transfer'* (*auto simp: map-pred-def map-add-def split: option.splits*)

lemma *fmpred-filter*[*intro*]: $\text{fmpred } P \ m \implies \text{fmpred } P \ (\text{fmfilter } Q \ m)$
by *transfer'* (*auto simp: map-pred-def map-filter-def*)

lemma *fmpred-drop*[*intro*]: $\text{fmpred } P \ m \implies \text{fmpred } P \ (\text{fmdrop } a \ m)$
by (*auto simp: fmfilter-alt-defs*)

lemma *fmpred-drop-set*[*intro*]: $\text{fmpred } P \ m \implies \text{fmpred } P \ (\text{fmdrop-set } A \ m)$
by (*auto simp: fmfilter-alt-defs*)

lemma *fmpred-drop-fset*[*intro*]: $\text{fmpred } P \ m \implies \text{fmpred } P \ (\text{fmdrop-fset } A \ m)$
by (*auto simp: fmfilter-alt-defs*)

lemma *fmpred-restrict-set*[*intro*]: $\text{fmpred } P \ m \implies \text{fmpred } P \ (\text{fmrestrict-set } A \ m)$
by (*auto simp: fmfilter-alt-defs*)

lemma *fmpred-restrict-fset*[*intro*]: $\text{fmpred } P \ m \implies \text{fmpred } P \ (\text{fmrestrict-fset } A \ m)$
by (*auto simp: fmfilter-alt-defs*)

lemma *fmpred-cases*[*consumes 1*]:
assumes $\text{fmpred } P \ m$
obtains $(\text{none}) \ \text{fmlookup } m \ x = \text{None} \mid (\text{some}) \ y \ \textbf{where} \ \text{fmlookup } m \ x = \text{Some } y \ P \ x \ y$
using *assms* **by** *auto*

lift-definition *fmsubset* :: $('a, 'b) \text{fmap} \Rightarrow ('a, 'b) \text{fmap} \Rightarrow \text{bool}$ (**infix** \subseteq_f 50)
is *map-le*

.

lemma *fmsubset-alt-def*: $m \subseteq_f n \longleftrightarrow \text{fmpred } (\lambda k v. \text{fmlookup } n \ k = \text{Some } v) \ m$
by *transfer'* (*auto simp: map-pred-def map-le-def dom-def split: option.splits*)

lemma *fmsubset-pred*: $\text{fmpred } P \ m \implies n \subseteq_f m \implies \text{fmpred } P \ n$
unfolding *fmsubset-alt-def fmpred-iff*
by *auto*

lemma *fmsubset-filter-mono*: $m \subseteq_f n \implies \text{fmfilter } P \ m \subseteq_f \text{fmfilter } P \ n$
unfolding *fmsubset-alt-def fmpred-iff*
by *auto*

lemma *fmsubset-drop-mono*: $m \subseteq_f n \implies \text{fmdrop } a \ m \subseteq_f \text{fmdrop } a \ n$
unfolding *fmfilter-alt-defs* **by** (rule *fmsubset-filter-mono*)

lemma *fmsubset-drop-set-mono*: $m \subseteq_f n \implies \text{fmdrop-set } A \ m \subseteq_f \text{fmdrop-set } A \ n$
unfolding *fmfilter-alt-defs* **by** (rule *fmsubset-filter-mono*)

lemma *fmsubset-drop-fset-mono*: $m \subseteq_f n \implies \text{fmdrop-fset } A \ m \subseteq_f \text{fmdrop-fset } A \ n$
unfolding *fmfilter-alt-defs* **by** (rule *fmsubset-filter-mono*)

lemma *fmsubset-restrict-set-mono*: $m \subseteq_f n \implies \text{fmrestrict-set } A \ m \subseteq_f \text{fmrestrict-set } A \ n$
unfolding *fmfilter-alt-defs* **by** (rule *fmsubset-filter-mono*)

lemma *fmsubset-restrict-fset-mono*: $m \subseteq_f n \implies \text{fmrestrict-fset } A \ m \subseteq_f \text{fmrestrict-fset } A \ n$
unfolding *fmfilter-alt-defs* **by** (rule *fmsubset-filter-mono*)

lemma *fmfilter-subset[simp]*: $\text{fmfilter } P \ m \subseteq_f m$
unfolding *fmsubset-alt-def fmpred-iff* **by** *auto*

lemma *fmsubset-drop[simp]*: $\text{fmdrop } a \ m \subseteq_f m$
unfolding *fmfilter-alt-defs* **by** (rule *fmfilter-subset*)

lemma *fmsubset-drop-set[simp]*: $\text{fmdrop-set } S \ m \subseteq_f m$
unfolding *fmfilter-alt-defs* **by** (rule *fmfilter-subset*)

lemma *fmsubset-drop-fset[simp]*: $\text{fmdrop-fset } S \ m \subseteq_f m$
unfolding *fmfilter-alt-defs* **by** (rule *fmfilter-subset*)

lemma *fmsubset-restrict-set[simp]*: $\text{fmrestrict-set } S \ m \subseteq_f m$
unfolding *fmfilter-alt-defs* **by** (rule *fmfilter-subset*)

lemma *fmsubset-restrict-fset[simp]*: $\text{fmrestrict-fset } S \ m \subseteq_f m$
unfolding *fmfilter-alt-defs* **by** (rule *fmfilter-subset*)

lift-definition *fset-of-fmap* :: $('a, 'b) \text{ fmap} \Rightarrow ('a \times 'b) \text{ fset}$ **is** *set-of-map*
by (rule *set-of-map-finite*)

lemma *fset-of-fmap-inj[intro, simp]*: $\text{inj } \text{fset-of-fmap}$

apply *rule*
apply *transfer'*
using *set-of-map-inj* **unfolding** *inj-def* **by** *auto*

lemma *fset-of-fmap-iff[simp]*: $(a, b) \in | \text{fset-of-fmap } m \longleftrightarrow \text{fmlookup } m \ a = \text{Some } b$
by *transfer'* (*auto simp: set-of-map-def*)

lemma *fset-of-fmap-iff'*: $(a, b) \in \text{fset } (\text{fset-of-fmap } m) \longleftrightarrow \text{fmlookup } m \ a = \text{Some } b$
by *simp*

lift-definition *fmap-of-list* :: $('a \times 'b) \text{ list} \Rightarrow ('a, 'b) \text{ fmap}$
is *map-of*
parametric *map-of-transfer*
by (*rule finite-dom-map-of*)

lemma *fmap-of-list-simps[simp]*:
 $\text{fmap-of-list } [] = \text{fmempty}$
 $\text{fmap-of-list } ((k, v) \# \text{ kvs}) = \text{fmupd } k \ v \ (\text{fmap-of-list } \text{ kvs})$
by (*transfer, simp add: map-upd-def*) $+$

lemma *fmap-of-list-app[simp]*: $\text{fmap-of-list } (xs @ ys) = \text{fmap-of-list } ys ++_f \text{fmap-of-list } xs$
by *transfer' simp*

lemma *fmupd-alt-def*: $\text{fmupd } k \ v \ m = m ++_f \text{fmap-of-list } [(k, v)]$
by *simp*

lemma *fmprpred-of-list[intro]*:
assumes $\bigwedge k \ v. (k, v) \in \text{set } xs \implies P \ k \ v$
shows $\text{fmprpred } P \ (\text{fmap-of-list } xs)$
using *assms*
by (*induction xs*) (*transfer'*; *auto simp: map-pred-def*) $+$

lemma *fmap-of-list-SomeD*: $\text{fmlookup } (\text{fmap-of-list } xs) \ k = \text{Some } v \implies (k, v) \in \text{set } xs$
by *transfer'* (*auto dest: map-of-SomeD*)

lemma *fmdom-fmap-of-list[simp]*: $\text{fmdom } (\text{fmap-of-list } xs) = \text{fset-of-list } (\text{map fst } xs)$
by *transfer'* (*simp add: dom-map-of-conv-image-fst*)

lift-definition *fmrel-on-fset* :: $'a \text{ fset} \Rightarrow ('b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('a, 'b) \text{ fmap} \Rightarrow ('a, 'c) \text{ fmap} \Rightarrow \text{bool}$
is *rel-map-on-set*
.

lemma *fmrel-on-fset-alt-def*: $\text{fmrel-on-fset } S \ P \ m \ n \longleftrightarrow \text{fBall } S \ (\lambda x. \text{rel-option } P$

(fmlookup m x) (fmlookup n x))
by transfer' (auto simp: rel-map-on-set-def eq-onp-def rel-fun-def)

lemma fmrel-on-fsetI[*intro*]:
 assumes $\bigwedge x. x \in S \implies \text{rel-option } P \text{ (fmlookup m x) (fmlookup n x)}$
 shows fmrel-on-fset S P m n
by (simp add: assms fmrel-on-fset-alt-def)

lemma fmrel-on-fset-mono[*mono*]: $R \leq Q \implies \text{fmrel-on-fset } S \ R \leq \text{fmrel-on-fset } S \ Q$
unfolding fmrel-on-fset-alt-def[*abs-def*]
using option.rel-mono **by** blast

lemma fmrel-on-fsetD: $x \in S \implies \text{fmrel-on-fset } S \ P \ m \ n \implies \text{rel-option } P \text{ (fmlookup m x) (fmlookup n x)}$
unfolding fmrel-on-fset-alt-def
by auto

lemma fmrel-on-fsubset: $\text{fmrel-on-fset } S \ R \ m \ n \implies T \subseteq S \implies \text{fmrel-on-fset } T \ R \ m \ n$
unfolding fmrel-on-fset-alt-def
by auto

lemma fmrel-on-fset-unionI:
 $\text{fmrel-on-fset } A \ R \ m \ n \implies \text{fmrel-on-fset } B \ R \ m \ n \implies \text{fmrel-on-fset } (A \cup B) \ R \ m \ n$
unfolding fmrel-on-fset-alt-def
by auto

lemma fmrel-on-fset-updateI:
 assumes fmrel-on-fset S P m n P v₁ v₂
 shows fmrel-on-fset (finset k S) P (fmupd k v₁ m) (fmupd k v₂ n)
using assms
unfolding fmrel-on-fset-alt-def
by auto

lift-definition fmimage :: ('a, 'b) fmap \Rightarrow 'a fset \Rightarrow 'b fset **is** $\lambda m \ S. \{b \mid a \ b. \ m \ a = \text{Some } b \wedge a \in S\}$
by (smt (verit, del-insts) Collect-mono-iff finite-surj ran-alt-def ran-def)

lemma fmimage-alt-def: $\text{fmimage } m \ S = \text{fmran } (\text{fmrestrict-fset } S \ m)$
by transfer' (auto simp: ran-def map-restrict-set-def map-filter-def)

lemma fmimage-empty[*simp*]: $\text{fmimage } m \ \text{fempty} = \text{fempty}$
by transfer' auto

lemma fmimage-subset-ran[*simp*]: $\text{fmimage } m \ S \subseteq \text{fmran } m$
by transfer' (auto simp: ran-def)

lemma *fmimage-dom[simp]*: $\text{fmimage } m (\text{fmdom } m) = \text{fmran } m$
by *transfer'* (*auto simp: ran-def*)

lemma *fmimage-inter*: $\text{fmimage } m (A \mid\cap\mid B) \mid\subseteq\mid \text{fmimage } m A \mid\cap\mid \text{fmimage } m B$
by *transfer'* *auto*

lemma *fmimage-inter-dom[simp]*:
 $\text{fmimage } m (\text{fmdom } m \mid\cap\mid A) = \text{fmimage } m A$
 $\text{fmimage } m (A \mid\cap\mid \text{fmdom } m) = \text{fmimage } m A$
by (*transfer'*; *auto*)⁺

lemma *fmimage-union[simp]*: $\text{fmimage } m (A \mid\cup\mid B) = \text{fmimage } m A \mid\cup\mid \text{fmimage } m B$
by *transfer'* *auto*

lemma *fmimage-Union[simp]*: $\text{fmimage } m (\text{ffUnion } A) = \text{ffUnion } (\text{fmimage } m \mid\mid A)$
by *transfer'* *auto*

lemma *fmimage-filter[simp]*: $\text{fmimage } (\text{fmfilter } P \ m) \ A = \text{fmimage } m (\text{ffilter } P \ A)$
by *transfer'* (*auto simp: map-filter-def*)

lemma *fmimage-drop[simp]*: $\text{fmimage } (\text{fmdrop } a \ m) \ A = \text{fmimage } m (A - \{|a|\})$
by (*simp add: fmimage-alt-def*)

lemma *fmimage-drop-fset[simp]*: $\text{fmimage } (\text{fmdrop-fset } B \ m) \ A = \text{fmimage } m (A - B)$
by *transfer'* (*auto simp: map-filter-def map-drop-set-def*)

lemma *fmimage-restrict-fset[simp]*: $\text{fmimage } (\text{fmrestrict-fset } B \ m) \ A = \text{fmimage } m (A \mid\cap\mid B)$
by *transfer'* (*auto simp: map-filter-def map-restrict-set-def*)

lemma *fmfilter-ran[simp]*: $\text{fmran } (\text{fmfilter } P \ m) = \text{fmimage } m (\text{ffilter } P \ (\text{fmdom } m))$
by *transfer'* (*auto simp: ran-def map-filter-def*)

lemma *fmran-drop[simp]*: $\text{fmran } (\text{fmdrop } a \ m) = \text{fmimage } m (\text{fmdom } m - \{|a|\})$
by *transfer'* (*auto simp: ran-def map-drop-def map-filter-def*)

lemma *fmran-drop-fset[simp]*: $\text{fmran } (\text{fmdrop-fset } A \ m) = \text{fmimage } m (\text{fmdom } m - A)$
by *transfer'* (*auto simp: ran-def map-drop-set-def map-filter-def*)

lemma *fmran-restrict-fset*: $\text{fmran } (\text{fmrestrict-fset } A \ m) = \text{fmimage } m (\text{fmdom } m \mid\cap\mid A)$
by *transfer'* (*auto simp: ran-def map-restrict-set-def map-filter-def*)

lemma *fmlookup-image-iff*: $y \mid\in\mid \text{fmimage } m \ A \longleftrightarrow (\exists x. \text{fmlookup } m \ x = \text{Some } y)$

$y \wedge x \mid \in \mid A$)
by *transfer'* (*auto simp: ran-def*)

lemma *fmimageI*: $\text{fmlookup } m \ x = \text{Some } y \implies x \mid \in \mid A \implies y \mid \in \mid \text{fmimage } m \ A$
by (*auto simp: fmlookup-image-iff*)

lemma *fmimageE*[*elim*]:
assumes $y \mid \in \mid \text{fmimage } m \ A$
obtains x **where** $\text{fmlookup } m \ x = \text{Some } y \ x \mid \in \mid A$
using *assms* **by** (*auto simp: fmlookup-image-iff*)

lift-definition *fmcomp* :: $('b, 'c) \text{fmap} \Rightarrow ('a, 'b) \text{fmap} \Rightarrow ('a, 'c) \text{fmap}$ (**infixl** \circ_f 55)
is *map-comp*
parametric *map-comp-transfer*
by (*rule dom-comp-finite*)

lemma *fmlookup-comp*[*simp*]: $\text{fmlookup } (m \circ_f n) \ x = \text{Option.bind } (\text{fmlookup } n \ x)$
(fmlookup m)
by *transfer'* (*auto simp: map-comp-def split: option.splits*)

end

30.4 BNF setup

lift-bnf $('a, \text{fmran}': 'b) \text{fmap}$ [*wits: Map.empty*]
for *map: fmmap*
rel: fmrel
by *auto*

declare *fmap.pred-mono*[*mono*]

lemma *fmran'-alt-def*: $\text{fmran}' \ m = \text{fset } (\text{fmran } m)$
including *fset.lifting*
by *transfer'* (*auto simp: ran-def fun-eq-iff*)

lemma *fmlookup-ran'-iff*: $y \in \text{fmran}' \ m \longleftrightarrow (\exists x. \text{fmlookup } m \ x = \text{Some } y)$
by *transfer'* (*auto simp: ran-def*)

lemma *fmran'I*: $\text{fmlookup } m \ x = \text{Some } y \implies y \in \text{fmran}' \ m$
by (*auto simp: fmlookup-ran'-iff*)

lemma *fmran'E*[*elim*]:
assumes $y \in \text{fmran}' \ m$
obtains x **where** $\text{fmlookup } m \ x = \text{Some } y$
using *assms* **by** (*auto simp: fmlookup-ran'-iff*)

lemma *fmrel-iff*: $\text{fmrel } R \ m \ n \longleftrightarrow (\forall x. \text{rel-option } R \ (\text{fmlookup } m \ x) \ (\text{fmlookup } n \ x))$

$x))$
by *transfer'* (*auto simp: rel-fun-def*)

lemma *fmrelI[intro]*:
assumes $\bigwedge x. \text{rel-option } R \text{ (fmlookup } m \text{ } x) \text{ (fmlookup } n \text{ } x)$
shows *fmrel* *R m n*
using *assms*
by *transfer' auto*

lemma *fmrel-upd[intro]*: *fmrel P m n* \implies *P x y* \implies *fmrel P (fmupd k x m) (fmupd k y n)*
by *transfer' (auto simp: map-upd-def rel-fun-def)*

lemma *fmrelD[dest]*: *fmrel P m n* \implies *rel-option P (fmlookup m x) (fmlookup n x)*
by *transfer' (auto simp: rel-fun-def)*

lemma *fmrel-addI[intro]*:
assumes *fmrel P m n fmrel P a b*
shows *fmrel P (m ++_f a) (n ++_f b)*
by (*smt (verit, del-insts) assms domIff fmdom.rep-eq fmlookup-add fmrel-iff option.rel-sel*)

lemma *fmrel-cases[consumes 1]*:
assumes *fmrel P m n*
obtains (*none*) *fmlookup m x = None fmlookup n x = None*
 | (*some*) *a b* **where** *fmlookup m x = Some a fmlookup n x = Some b P a b*
proof –
from *assms* **have** *rel-option P (fmlookup m x) (fmlookup n x)*
by *auto*
then show *thesis*
using *none some*
by (*cases rule: option.rel-cases*) *auto*
qed

lemma *fmrel-filter[intro]*: *fmrel P m n* \implies *fmrel P (fmfilter Q m) (fmfilter Q n)*
unfolding *fmrel-iff* **by** *auto*

lemma *fmrel-drop[intro]*: *fmrel P m n* \implies *fmrel P (fmdrop a m) (fmdrop a n)*
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-drop-set[intro]*: *fmrel P m n* \implies *fmrel P (fmdrop-set A m) (fmdrop-set A n)*
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-drop-fset[intro]*: *fmrel P m n* \implies *fmrel P (fmdrop-fset A m) (fmdrop-fset A n)*
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-restrict-set[intro]*: *fmrel P m n* \implies *fmrel P (fmrestrict-set A m)*

(*fmrestrict-set* *A n*)
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-restrict-fset[intro]*: *fmrel P m n \implies fmrel P (fmrestrict-fset A m)*
(*fmrestrict-fset A n*)
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-on-fset-fmrel-restrict*:
fmrel-on-fset S P m n \longleftrightarrow fmrel P (fmrestrict-fset S m) (fmrestrict-fset S n)
unfolding *fmrel-on-fset-alt-def fmrel-iff*
by *auto*

lemma *fmrel-on-fset-refl-strong*:
assumes $\bigwedge x y. x \in S \implies \text{fmlookup } m \ x = \text{Some } y \implies P \ y \ y$
shows *fmrel-on-fset S P m m*
unfolding *fmrel-on-fset-fmrel-restrict fmrel-iff*
using *assms*
by (*simp add: option.rel-sel*)

lemma *fmrel-on-fset-addI*:
assumes *fmrel-on-fset S P m n fmrel-on-fset S P a b*
shows *fmrel-on-fset S P (m ++_f a) (n ++_f b)*
using *assms*
unfolding *fmrel-on-fset-fmrel-restrict*
by *auto*

lemma *fmrel-fmdom-eq*:
assumes *fmrel P x y*
shows *fmdom x = fmdom y*
proof –
have *a \in fmdom x \longleftrightarrow a \in fmdom y* **for** *a*
proof –
have *rel-option P (fmlookup x a) (fmlookup y a)*
using *assms* **by** (*simp add: fmrel-iff*)
thus *?thesis*
by *cases (auto intro: fmdomI)*
qed
thus *?thesis*
by *auto*
qed

lemma *fmrel-fmdom'-eq*: *fmrel P x y \implies fmdom' x = fmdom' y*
unfolding *fmdom'-alt-def*
by (*metis fmrel-fmdom-eq*)

lemma *fmrel-rel-fmran*:
assumes *fmrel P x y*
shows *rel-fset P (fmran x) (fmran y)*
proof –

```

{
  fix b
  assume b |∈| fmran x
  then obtain a where fmlookup x a = Some b
    by auto
  moreover have rel-option P (fmlookup x a) (fmlookup y a)
    using assms by auto
  ultimately have ∃ b'. b' |∈| fmran y ∧ P b b'
    by (metis option-rel-Some1 fmranI)
}
moreover
{
  fix b
  assume b |∈| fmran y
  then obtain a where fmlookup y a = Some b
    by auto
  moreover have rel-option P (fmlookup x a) (fmlookup y a)
    using assms by auto
  ultimately have ∃ b'. b' |∈| fmran x ∧ P b' b
    by (metis option-rel-Some2 fmranI)
}
ultimately show ?thesis
  unfolding rel-fset-alt-def
  by auto
qed

lemma fmrel-rel-fmran': fmrel P x y ⟹ rel-set P (fmran' x) (fmran' y)
  unfolding fmran'-alt-def
  by (metis fmrel-rel-fmran rel-fset-fset)

lemma pred-fmap-fmpred[simp]: pred-fmap P = fmpred (λ-. P)
  unfolding fmap.pred-set fmran'-alt-def
  using fmranI by fastforce

lemma pred-fmap-id[simp]: pred-fmap id (fmap f m) ⟷ pred-fmap f m
  unfolding fmap.pred-set fmap.set-map
  by simp

lemma pred-fmapD: pred-fmap P m ⟹ x |∈| fmran m ⟹ P x
  by auto

lemma fmlookup-map[simp]: fmlookup (fmap f m) x = map-option f (fmlookup m x)
  by transfer' auto

lemma fmpred-map[simp]: fmpred P (fmap f m) ⟷ fmpred (λk v. P k (f v)) m
  unfolding fmpred-iff pred-fmap-def fmap.set-map
  by auto

```


lemma *fmprid-id[simp]*: $\text{fmprid } (\lambda\cdot. \text{id}) (\text{fmmap } f \ m) \longleftrightarrow \text{fmprid } (\lambda\cdot. f) \ m$
by *simp*

lemma *fmmap-add[simp]*: $\text{fmmap } f \ (m \ ++_f \ n) = \text{fmmap } f \ m \ ++_f \ \text{fmmap } f \ n$
by *transfer' (auto simp: map-add-def fun-eq-iff split: option.splits)*

lemma *fmmap-empty[simp]*: $\text{fmmap } f \ \text{fmempty} = \text{fmempty}$
by *transfer auto*

lemma *fmdom-map[simp]*: $\text{fmdom } (\text{fmmap } f \ m) = \text{fmdom } m$
including *fset.lifting*
by *transfer' simp*

lemma *fmdom'-map[simp]*: $\text{fmdom}' (\text{fmmap } f \ m) = \text{fmdom}' m$
by *transfer' simp*

lemma *fmran-fmmap[simp]*: $\text{fmran } (\text{fmmap } f \ m) = f \ \mid \! \! \mid \ \text{fmran } m$
including *fset.lifting*
by *transfer' (auto simp: ran-def)*

lemma *fmran'-fmmap[simp]*: $\text{fmran}' (\text{fmmap } f \ m) = f \ \text{' } \text{fmran}' m$
by *transfer' (auto simp: ran-def)*

lemma *fmfilter-fmmap[simp]*: $\text{fmfilter } P \ (\text{fmmap } f \ m) = \text{fmmap } f \ (\text{fmfilter } P \ m)$
by *transfer' (auto simp: map-filter-def)*

lemma *fmdrop-fmmap[simp]*: $\text{fmdrop } a \ (\text{fmmap } f \ m) = \text{fmmap } f \ (\text{fmdrop } a \ m)$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-fmmap[simp]*: $\text{fmdrop-set } A \ (\text{fmmap } f \ m) = \text{fmmap } f \ (\text{fmdrop-set } A \ m)$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-fmmap[simp]*: $\text{fmdrop-fset } A \ (\text{fmmap } f \ m) = \text{fmmap } f \ (\text{fmdrop-fset } A \ m)$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-fmmap[simp]*: $\text{fmrestrict-set } A \ (\text{fmmap } f \ m) = \text{fmmap } f \ (\text{fmrestrict-set } A \ m)$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-fmmap[simp]*: $\text{fmrestrict-fset } A \ (\text{fmmap } f \ m) = \text{fmmap } f \ (\text{fmrestrict-fset } A \ m)$
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmmap-subset[intro]*: $m \subseteq_f n \implies \text{fmmap } f \ m \subseteq_f \text{fmmap } f \ n$
by *transfer' (auto simp: map-le-def)*

lemma *fmmap-fset-of-fmap*: $\text{fset-of-fmap } (\text{fmmap } f \ m) = (\lambda(k, v). (k, f \ v)) \ \mid \! \! \mid$

```

fset-of-fmap m
  including fset.lifting
  by transfer' (auto simp: set-of-map-def)

```

```

lemma fmmmap-fmupd: fmmmap f (fmupd x y m) = fmupd x (f y) (fmmmap f m)
  by transfer' (auto simp: fun-eq-iff map-upd-def)

```

30.5 size setup

```

definition size-fmap :: ('a  $\Rightarrow$  nat)  $\Rightarrow$  ('b  $\Rightarrow$  nat)  $\Rightarrow$  ('a, 'b) fmap  $\Rightarrow$  nat where
[simp]: size-fmap f g m = size-fset ( $\lambda(a, b). f\ a + g\ b$ ) (fset-of-fmap m)

```

```

instantiation fmap :: (type, type) size begin

```

```

definition size-fmap where
size-fmap-overloaded-def: size-fmap = Finite-Map.size-fmap ( $\lambda-. 0$ ) ( $\lambda-. 0$ )

```

```

instance ..

```

```

end

```

```

lemma size-fmap-overloaded-simps[simp]: size x = size (fset-of-fmap x)
  unfolding size-fmap-overloaded-def
  by simp

```

```

lemma fmap-size-o-map: size-fmap f g  $\circ$  fmmmap h = size-fmap f (g  $\circ$  h)

```

```

proof -
  have inj: inj-on ( $\lambda(k, v). (k, h\ v)$ ) (fset (fset-of-fmap m)) for m
    using inj-on-def by force
  show ?thesis
    unfolding size-fmap-def
    apply (clarsimp simp: fun-eq-iff fmmmap-fset-of-fmap sum.reindex [OF inj])
    by (rule sum.cong) (auto split: prod.splits)
qed

```

```

setup <
BNF-LFP-Size.register-size-global type-name <fmap> const-name <size-fmap>
  @{thm size-fmap-overloaded-def} @{{thms size-fmap-def size-fmap-overloaded-simps}
  @{{thms fmap-size-o-map}}
>

```

30.6 Additional operations

```

lift-definition fmmmap-keys :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  ('a, 'b) fmap  $\Rightarrow$  ('a, 'c) fmap is
   $\lambda f\ m\ a. \text{map-option } (f\ a) (m\ a)$ 
  unfolding dom-def
  by simp

```

```

lemma fmpred-fmmmap-keys[simp]: fmpred P (fmmmap-keys f m) = fmpred ( $\lambda a\ b. P\ a\ (f\ a\ b)$ ) m

```

by *transfer'* (*auto simp: map-pred-def split: option.splits*)

lemma *fmdom-fmmap-keys[simp]*: *fmdom (fmmap-keys f m) = fmdom m*
including *fset.lifting*
by *transfer' auto*

lemma *fmlookup-fmmap-keys[simp]*: *fmlookup (fmmap-keys f m) x = map-option (f x) (fmlookup m x)*
by *transfer' simp*

lemma *fmfilter-fmmap-keys[simp]*: *fmfilter P (fmmap-keys f m) = fmmap-keys f (fmfilter P m)*
by *transfer' (auto simp: map-filter-def)*

lemma *fmdrop-fmmap-keys[simp]*: *fmdrop a (fmmap-keys f m) = fmmap-keys f (fmdrop a m)*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-fmmap-keys[simp]*: *fmdrop-set A (fmmap-keys f m) = fmmap-keys f (fmdrop-set A m)*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-fmmap-keys[simp]*: *fmdrop-fset A (fmmap-keys f m) = fmmap-keys f (fmdrop-fset A m)*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-fmmap-keys[simp]*: *fmrestrict-set A (fmmap-keys f m) = fmmap-keys f (fmrestrict-set A m)*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-fmmap-keys[simp]*: *fmrestrict-fset A (fmmap-keys f m) = fmmap-keys f (fmrestrict-fset A m)*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmmap-keys-subset[intro]*: $m \subseteq_f n \implies \text{fmmap-keys } f \ m \subseteq_f \text{fmmap-keys } f \ n$
by *transfer' (auto simp: map-le-def dom-def)*

definition *sorted-list-of-fmap* :: $('a::\text{linorder}, 'b) \text{fmap} \Rightarrow ('a \times 'b) \text{list}$ **where**
sorted-list-of-fmap m = map ($\lambda k. (k, \text{the } (\text{fmlookup } m \ k)$)) (sorted-list-of-fset (fmdom m))

lemma *list-all-sorted-list[simp]*: *list-all P (sorted-list-of-fmap m) = fmpred (curry P) m*

unfolding *sorted-list-of-fmap-def curry-def list.pred-map*
by *(smt (verit, best) Ball-set comp-def fmpred-alt-def sorted-list-of-fset-simps(1))*

lemma *map-of-sorted-list[simp]*: *map-of (sorted-list-of-fmap m) = fmlookup m*
unfolding *sorted-list-of-fmap-def*

```

including fset.lifting
by transfer (simp add: map-of-map-keys)

```

30.7 Additional properties

```

lemma fmchoice':
  assumes finite S  $\forall x \in S. \exists y. Q\ x\ y$ 
  shows  $\exists m. \text{fmdom}'\ m = S \wedge \text{fmpred}\ Q\ m$ 
proof –
  obtain f where f:  $Q\ x\ (f\ x)$  if  $x \in S$  for x
  using assms by metis
  define f' where  $f'\ x = (\text{if } x \in S \text{ then } \text{Some } (f\ x) \text{ else } \text{None})$  for x

  have eq-onp  $(\lambda m. \text{finite } (\text{dom } m))\ f'\ f'$ 
    unfolding eq-onp-def f'-def dom-def using assms by auto

  show ?thesis
    apply (rule exI[where  $x = \text{Abs-fmap } f'$ ])
    apply (subst fmpred.abs-eq, fact)
    apply (subst fmdom'.abs-eq, fact)
    unfolding f'-def dom-def map-pred-def using f
    by auto
qed

```

30.8 Lifting/transfer setup

```

context includes lifting-syntax begin

```

```

lemma fmempty-transfer[simp, intro, transfer-rule]: fmrel P fmempty fmempty
  by transfer auto

```

```

lemma fmadd-transfer[transfer-rule]:
   $(\text{fmrel } P \implies \text{fmrel } P \implies \text{fmrel } P)\ \text{fmadd fmadd}$ 
  by (intro fmrel-addI rel-funI)

```

```

lemma fmupd-transfer[transfer-rule]:
   $((=) \implies P \implies \text{fmrel } P \implies \text{fmrel } P)\ \text{fmupd fmupd}$ 
  by auto

```

```

end

```

```

lemma Quotient-fmap-bnf[quot-map]:
  assumes Quotient R Abs Rep T
  shows Quotient (fmrel R) (fmmap Abs) (fmmap Rep) (fmrel T)
unfolding Quotient-alt-def4 proof safe
  fix m n
  assume fmrel T m n
  then have fmlookup (fmmap Abs m) x = fmlookup n x for x
    using assms unfolding Quotient-alt-def
    by (cases rule: fmrel-cases[where  $x = x$ ]) auto

```

```

    then show fmmmap Abs m = n
      by (rule fmap-ext)
next
  fix m
  show fmrel T (fmmmap Rep m) m
    unfolding fmap.rel-map
    by (metis (mono-tags) Quotient-alt-def assms fmap.rel-refl)
next
  from assms have R = T OO T-1-1
    unfolding Quotient-alt-def4 by simp
  then show fmrel R = fmrel T OO (fmrel T)-1-1
    by (simp add: fmap.rel-compp fmap.rel-conversep)
qed

```

30.9 View as datatype

```

lemma fmap-distinct[simp]:
  fmempty ≠ fmupd k v m
  fmupd k v m ≠ fmempty
  by (transfer'; auto simp: map-upd-def fun-eq-iff)+

```

lifting-update *fmap.lifting*

```

lemma fmap-exhaust[cases type: fmap]:
  obtains (fmempty) m = fmempty
    | (fmupd) x y m' where m = fmupd x y m' x |∉| fmdom m'
using that including fmap.lifting and fset.lifting
proof transfer
  fix m P
  assume finite (dom m)
  assume empty: P if m = Map.empty
  assume map-upd: P if finite (dom m') m = map-upd x y m' x ∉ dom m' for x
y m'

```

```

show P
proof (cases m = Map.empty)
case True thus ?thesis using empty by simp
next
case False
hence dom m ≠ {} by simp
then obtain x where x ∈ dom m by blast

```

```

let ?m' = map-drop x m

```

```

show ?thesis
proof (rule map-upd)
  show finite (dom ?m')
    using ⟨finite (dom m)⟩
    unfolding map-drop-def

```

```

      by auto
    next
      show  $m = \text{map-upd } x \ (\text{the } (m \ x)) \ ?m'$ 
      using  $\langle x \in \text{dom } m \rangle$  unfolding map-drop-def map-filter-def map-upd-def
      by auto
    next
      show  $x \notin \text{dom } ?m'$ 
      unfolding map-drop-def map-filter-def
      by auto
    qed
  qed
qed

```

```

lemma fmap-induct[case-names fmempty fmupd, induct type: fmap]:
  assumes  $P \text{ fmempty}$ 
  assumes  $(\bigwedge x \ y \ m. P \ m \implies \text{fmlookup } m \ x = \text{None} \implies P \ (\text{fmupd } x \ y \ m))$ 
  shows  $P \ m$ 
proof (induction fmdom m arbitrary: m rule: fset-induct-stronger)
  case empty
  hence  $m = \text{fmempty}$ 
  by (metis fmrestrict-fset-dom fmrestrict-fset-null)
  with assms show  $?case$ 
  by simp
next
  case (insert x S)
  hence  $S = \text{fmdom } (\text{fmdrop } x \ m)$ 
  by auto
  with insert have  $P \ (\text{fmdrop } x \ m)$ 
  by auto
  moreover
  obtain  $y$  where  $\text{fmlookup } m \ x = \text{Some } y$ 
  using insert.hyps by force
  hence  $m = \text{fmupd } x \ y \ (\text{fmdrop } x \ m)$ 
  by (auto intro: fmap-ext)
  ultimately show  $?case$ 
  by (metis assms(2) fmdrop-lookup)
qed

```

30.10 Code setup

instantiation *fmap* :: (*type*, *equal*) *equal* **begin**

definition *equal-fmap* $\equiv \text{fmrel } \text{HOL.equal}$

instance **proof**

```

  fix  $m \ n :: ('a, 'b) \text{fmap}$ 
  have  $\text{fmrel } (=) \ m \ n \longleftrightarrow (m = n)$ 
    by transfer' (simp add: option.rel-eq rel-fun-eq)
  then show equal-class.equal  $m \ n \longleftrightarrow (m = n)$ 

```

unfolding *equal-fmap-def*
by (*simp add: equal-eq[abs-def]*)

qed

end

lemma *fBall-alt-def*: $fBall\ S\ P \longleftrightarrow (\forall x. x \in S \longrightarrow P\ x)$
by *force*

lemma *fmrel-code*:

$fmrel\ R\ m\ n \longleftrightarrow$
 $fBall\ (fmdom\ m)\ (\lambda x. rel-option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)) \wedge$
 $fBall\ (fmdom\ n)\ (\lambda x. rel-option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x))$

unfolding *fmrel-iff fmlookup-dom-iff fBall-alt-def*
by (*metis option.collapse option.rel-sel*)

lemmas [*code*] =

fmrel-code
fmran'-alt-def
fmdom'-alt-def
fmfilter-alt-defs
pred-fmap-fmpred
fmsubset-alt-def
fmupd-alt-def
fmrel-on-fset-alt-def
fmpred-alt-def

code-datatype *fmap-of-list*

quickcheck-generator *fmap constructors: fmap-of-list*

context *includes fset.lifting* **begin**

lemma *fmlookup-of-list[code]*: $fmlookup\ (fmap-of-list\ m) = map-of\ m$
by *transfer simp*

lemma *fmempty-of-list[code]*: $fmempty = fmap-of-list\ []$
by *transfer simp*

lemma *fmran-of-list[code]*: $fmran\ (fmap-of-list\ m) = snd\ |\cdot|'\ fset-of-list\ (AList.clearjunk\ m)$
by *transfer (auto simp: ran-map-of)*

lemma *fmdom-of-list[code]*: $fmdom\ (fmap-of-list\ m) = fst\ |\cdot|'\ fset-of-list\ m$
by *transfer (auto simp: dom-map-of-conv-image-fst)*

lemma *fmfilter-of-list[code]*: $fmfilter\ P\ (fmap-of-list\ m) = fmap-of-list\ (filter\ (\lambda(k, -). P\ k)\ m)$
by *transfer' auto*

lemma *fmadd-of-list*[code]: *fmap-of-list* *m* ++_f *fmap-of-list* *n* = *fmap-of-list* (*AList.merge* *m* *n*)
by *transfer* (*simp add: merge-conv'*)

lemma *fmmap-of-list*[code]: *fmmap* *f* (*fmap-of-list* *m*) = *fmap-of-list* (*map* (*apsnd* *f*) *m*)
apply *transfer*
by (*metis* (*no-types*, *lifting*) *apsnd-conv map-eq-conv map-of-map old.prod.case old.prod.exhaust*)

lemma *fmmap-keys-of-list*[code]:
fmmap-keys *f* (*fmap-of-list* *m*) = *fmap-of-list* (*map* ($\lambda(a, b). (a, f\ a\ b)$) *m*)
apply *transfer*
subgoal for *f m* **by** (*induction* *m*) (*auto simp: apsnd-def map-prod-def fun-eq-iff*)
done

lemma *fmimage-of-list*[code]:
fmimage (*fmap-of-list* *m*) *A* = *fset-of-list* (*map snd* (*filter* ($\lambda(k, -). k \in A$) (*AList.clearjunk* *m*)))
apply (*subst fmimage-alt-def*)
apply (*subst fmfilter-alt-defs*)
apply (*subst fmfilter-of-list*)
apply (*subst fmran-of-list*)
apply *transfer'*
by (*metis* *AList.restrict-eq clearjunk-restrict list.set-map*)

lemma *fmcomp-list*[code]:
fmap-of-list *m* \circ_f *fmap-of-list* *n* = *fmap-of-list* (*AList.compose* *n* *m*)
by (*rule fmap-ext*) (*simp add: fmlookup-of-list compose-conv map-comp-def split: option.splits*)

end

30.11 Instances

lemma *exists-map-of*:
assumes *finite* (*dom* *m*) **shows** $\exists xs. \text{map-of } xs = m$
using *assms*
proof (*induction dom m arbitrary: m*)
case empty
hence *m* = *Map.empty*
by *auto*
moreover have *map-of* [] = *Map.empty*
by *simp*
ultimately show ?*case*
by *blast*
next
case (*insert* *x* *F*)


```

hence  $F = \text{dom } (\text{map-drop } x \ m)$ 
  unfolding map-drop-def map-filter-def dom-def by auto
with insert have  $\exists xs'. \text{map-of } xs' = \text{map-drop } x \ m$ 
  by auto
then obtain  $xs'$  where  $\text{map-of } xs' = \text{map-drop } x \ m$ 
..
moreover obtain  $y$  where  $m \ x = \text{Some } y$ 
  using insert unfolding dom-def by blast
ultimately have  $\text{map-of } ((x, y) \# xs') = m$ 
  using  $\langle \text{insert } x \ F = \text{dom } m \rangle$ 
  unfolding map-drop-def map-filter-def
  by auto
thus ?case
..
qed

```

```

lemma exists-fmap-of-list:  $\exists xs. \text{fmap-of-list } xs = m$ 
by transfer (rule exists-map-of)

```

```

lemma fmap-of-list-surj[simp, intro]: surj fmap-of-list
proof -
  have  $x \in \text{range } \text{fmap-of-list}$  for  $x :: ('a, 'b) \text{fmap}$ 
  unfolding image-iff
  using exists-fmap-of-list by (metis UNIV-I)
  thus ?thesis by auto
qed

```

```

instance fmap :: (countable, countable) countable
proof
  obtain to-nat ::  $('a \times 'b) \text{list} \Rightarrow \text{nat}$  where inj to-nat
  by (metis ex-inj)
  moreover have inj (inv fmap-of-list)
  using fmap-of-list-surj by (rule surj-imp-inj-inv)
  ultimately have inj (to-nat  $\circ$  inv fmap-of-list)
  by (rule inj-compose)
  thus  $\exists \text{to-nat}::('a, 'b) \text{fmap} \Rightarrow \text{nat}. \text{inj to-nat}$ 
  by auto
qed

```

```

instance fmap :: (finite, finite) finite
proof
  show finite (UNIV ::  $('a, 'b) \text{fmap set}$ )
  by (rule finite-imageD) auto
qed

```

```

lifting-update fmap.lifting
lifting-forget fmap.lifting

```

30.12 Tests

export-code

*Ball fset fmrel fmran fmran' fmdom fmdom' fmpred pred-fmap fmsubset fmupd
 fmrel-on-fset
 fmdrop fmdrop-set fmdrop-fset fmrestrict-set fmrestrict-fset fmimage fmlookup
 fmempty
 fmfilter fmadd fmmmap fmmmap-keys fmcomp
 checking SML Scala Haskell? OCaml?*

— *lifting* through *fmap*

experiment begin

context includes *fset.lifting* begin

lift-definition *test1* :: ('a, 'b fset) fmap **is** fmempty :: ('a, 'b set) fmap
 by *auto*

lift-definition *test2* :: 'a \Rightarrow 'b \Rightarrow ('a, 'b fset) fmap **is** $\lambda a\ b.$ fmupd a {b} fmempty
 by *auto*

end

end

end

31 Disjoint FSets

theory *Disjoint-FSets*

imports

HOL-Library.Finite-Map

Disjoint-Sets

begin

context

includes *fset.lifting*

begin

lift-definition *fdisjnt* :: 'a fset \Rightarrow 'a fset \Rightarrow bool **is** *disjnt* .

lemma *fdisjnt-alt-def*: *fdisjnt* *M N* \longleftrightarrow (*M* $\mid\cap\mid$ *N* = {||})
 by *transfer (simp add: disjnt-def)*

lemma *fdisjnt-insert*: *x* \notin *N* \Longrightarrow *fdisjnt* *M N* \Longrightarrow *fdisjnt* (*insert* *x M*) *N*
 by *transfer' (rule disjnt-insert)*

lemma *fdisjnt-subset-right*: *N'* $\mid\subseteq\mid$ *N* \Longrightarrow *fdisjnt* *M N* \Longrightarrow *fdisjnt* *M N'*

unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-subset-left*: $N' \sqsubseteq N \implies fdisjnt\ N\ M \implies fdisjnt\ N'\ M$
unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-union-right*: $fdisjnt\ M\ A \implies fdisjnt\ M\ B \implies fdisjnt\ M\ (A \sqcup B)$
unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-union-left*: $fdisjnt\ A\ M \implies fdisjnt\ B\ M \implies fdisjnt\ (A \sqcup B)\ M$
unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-swap*: $fdisjnt\ M\ N \implies fdisjnt\ N\ M$
including *fset.lifting* **by** *transfer'* (*auto simp: disjnt-def*)

lemma *distinct-append-fset*:
assumes *distinct xs distinct ys fdisjnt (fset-of-list xs) (fset-of-list ys)*
shows *distinct (xs @ ys)*
using *assms*
by *transfer' (simp add: disjnt-def)*

lemma *fdisjnt-contrI*:
assumes $\bigwedge x. x \sqsubseteq M \implies x \sqsubseteq N \implies False$
shows *fdisjnt M N*
using *assms*
by *transfer' (auto simp: disjnt-def)*

lemma *fdisjnt-Union-left*: $fdisjnt\ (ffUnion\ S)\ T \longleftrightarrow fBall\ S\ (\lambda S. fdisjnt\ S\ T)$
by *transfer' (auto simp: disjnt-def)*

lemma *fdisjnt-Union-right*: $fdisjnt\ T\ (ffUnion\ S) \longleftrightarrow fBall\ S\ (\lambda S. fdisjnt\ T\ S)$
by *transfer' (auto simp: disjnt-def)*

lemma *fdisjnt-ge-max*: $fBall\ X\ (\lambda x. x > fMax\ Y) \implies fdisjnt\ X\ Y$
by *transfer (auto intro: disjnt-ge-max)*

end

lemma *fmadd-disjnt*: $fdisjnt\ (fmdom\ m)\ (fmdom\ n) \implies m ++_f n = n ++_f m$
unfolding *fdisjnt-alt-def*
including *fset.lifting* **and** *fmap.lifting*
apply *transfer*
apply (*rule ext*)
apply (*auto simp: map-add-def split: option.splits*)
done

end

32 Lists with elements distinct as canonical example for datatype invariants

```

theory Dlist
imports Confluent-Quotient
begin

32.1 The type of distinct lists

typedef 'a dlist = {xs::'a list. distinct xs}
  morphisms list-of-dlist Abs-dlist
proof
  show [] ∈ {xs. distinct xs} by simp
qed

context begin

qualified definition dlist-eq where dlist-eq = BNF-Def.vimage2p remdups remdups
(=)

qualified lemma equivp-dlist-eq: equivp dlist-eq
  unfolding dlist-eq-def by(rule equivp-vimage2p)(rule identity-equivp)

qualified definition abs-dlist :: 'a list ⇒ 'a dlist where abs-dlist = Abs-dlist o
remdups

definition qcr-dlist :: 'a list ⇒ 'a dlist ⇒ bool where qcr-dlist x y ⟷ y =
abs-dlist x

qualified lemma Quotient-dlist-remdups: Quotient dlist-eq abs-dlist list-of-dlist
qcr-dlist
  unfolding Quotient-def dlist-eq-def qcr-dlist-def vimage2p-def abs-dlist-def
by (auto simp add: fun-eq-iff Abs-dlist-inject
list-of-dlist[simplified] list-of-dlist-inverse distinct-remdups-id)

end

locale Quotient-dlist begin
setup-lifting Dlist.Quotient-dlist-remdups Dlist.equivp-dlist-eq[THEN equivp-reflp2]
end

setup-lifting type-definition-dlist

lemma dlist-eq-iff:
  dxs = dys ⟷ list-of-dlist dxs = list-of-dlist dys
  by (simp add: list-of-dlist-inject)

lemma dlist-eqI:
  list-of-dlist dxs = list-of-dlist dys ⟹ dxs = dys

```

by (*simp add: dlist-eq-iff*)

Formal, totaled constructor for *'a dlist*:

definition *Dlist* :: *'a list* \Rightarrow *'a dlist* **where**
Dlist xs = *Abs-dlist (remdups xs)*

lemma *distinct-list-of-dlist* [*simp, intro*]:
distinct (list-of-dlist dxs)
using *list-of-dlist [of dxs]* **by** *simp*

lemma *list-of-dlist-Dlist* [*simp*]:
list-of-dlist (Dlist xs) = *remdups xs*
by (*simp add: Dlist-def Abs-dlist-inverse*)

lemma *remdups-list-of-dlist* [*simp*]:
remdups (list-of-dlist dxs) = *list-of-dlist dxs*
by *simp*

lemma *Dlist-list-of-dlist* [*simp, code abstype*]:
Dlist (list-of-dlist dxs) = *dxs*
by (*simp add: Dlist-def list-of-dlist-inverse distinct-remdups-id*)

Fundamental operations:

context
begin

qualified definition *empty* :: *'a dlist* **where**
empty = *Dlist []*

qualified definition *insert* :: *'a* \Rightarrow *'a dlist* \Rightarrow *'a dlist* **where**
insert x dxs = *Dlist (List.insert x (list-of-dlist dxs))*

qualified definition *remove* :: *'a* \Rightarrow *'a dlist* \Rightarrow *'a dlist* **where**
remove x dxs = *Dlist (remove1 x (list-of-dlist dxs))*

qualified definition *map* :: (*'a* \Rightarrow *'b*) \Rightarrow *'a dlist* \Rightarrow *'b dlist* **where**
map f dxs = *Dlist (remdups (List.map f (list-of-dlist dxs)))*

qualified definition *filter* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a dlist* \Rightarrow *'a dlist* **where**
filter P dxs = *Dlist (List.filter P (list-of-dlist dxs))*

qualified definition *rotate* :: *nat* \Rightarrow *'a dlist* \Rightarrow *'a dlist* **where**
rotate n dxs = *Dlist (List.rotate n (list-of-dlist dxs))*

end

Derived operations:

context
begin

qualified definition $null :: 'a\ dlist \Rightarrow bool$ **where**
 $null\ dxs = List.null\ (list-of-dlist\ dxs)$

qualified definition $member :: 'a\ dlist \Rightarrow 'a \Rightarrow bool$ **where**
 $member\ dxs = List.member\ (list-of-dlist\ dxs)$

qualified definition $length :: 'a\ dlist \Rightarrow nat$ **where**
 $length\ dxs = List.length\ (list-of-dlist\ dxs)$

qualified definition $fold :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ dlist \Rightarrow 'b \Rightarrow 'b$ **where**
 $fold\ f\ dxs = List.fold\ f\ (list-of-dlist\ dxs)$

qualified definition $foldr :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ dlist \Rightarrow 'b \Rightarrow 'b$ **where**
 $foldr\ f\ dxs = List.foldr\ f\ (list-of-dlist\ dxs)$

end

32.2 Executable version obeying invariant

lemma $list-of-dlist-empty$ [*simp*, *code abstract*]:
 $list-of-dlist\ Dlist.empty = []$
by (*simp add: Dlist.empty-def*)

lemma $list-of-dlist-insert$ [*simp*, *code abstract*]:
 $list-of-dlist\ (Dlist.insert\ x\ dxs) = List.insert\ x\ (list-of-dlist\ dxs)$
by (*simp add: Dlist.insert-def*)

lemma $list-of-dlist-remove$ [*simp*, *code abstract*]:
 $list-of-dlist\ (Dlist.remove\ x\ dxs) = remove1\ x\ (list-of-dlist\ dxs)$
by (*simp add: Dlist.remove-def*)

lemma $list-of-dlist-map$ [*simp*, *code abstract*]:
 $list-of-dlist\ (Dlist.map\ f\ dxs) = remdups\ (List.map\ f\ (list-of-dlist\ dxs))$
by (*simp add: Dlist.map-def*)

lemma $list-of-dlist-filter$ [*simp*, *code abstract*]:
 $list-of-dlist\ (Dlist.filter\ P\ dxs) = List.filter\ P\ (list-of-dlist\ dxs)$
by (*simp add: Dlist.filter-def*)

lemma $list-of-dlist-rotate$ [*simp*, *code abstract*]:
 $list-of-dlist\ (Dlist.rotate\ n\ dxs) = List.rotate\ n\ (list-of-dlist\ dxs)$
by (*simp add: Dlist.rotate-def*)

Explicit executable conversion

definition $dlist-of-list$ [*simp*]:
 $dlist-of-list = Dlist$

lemma [*code abstract*]:
 $list-of-dlist\ (dlist-of-list\ xs) = remdups\ xs$

```

by simp
  Equality
instantiation dlist :: (equal) equal
begin

definition HOL.equal dxs dys  $\longleftrightarrow$  HOL.equal (list-of-dlist dxs) (list-of-dlist dys)

instance
  by standard (simp add: equal-dlist-def equal list-of-dlist-inject)

end

declare equal-dlist-def [code]

lemma [code nbe]: HOL.equal (dxs :: 'a::equal dlist) dxs  $\longleftrightarrow$  True
  by (fact equal-refl)

```

32.3 Induction principle and case distinction

```

lemma dlist-induct [case-names empty insert, induct type: dlist]:
  assumes empty: P Dlist.empty
  assumes insrt:  $\bigwedge x \text{ dxs. } \neg \text{Dlist.member dxs } x \implies P \text{ dxs} \implies P (\text{Dlist.insert } x \text{ dxs})$ 
  shows P dxs
proof (cases dxs)
  case (Abs-dlist xs)
  then have distinct xs and dxs: dxs = Dlist xs
    by (simp-all add: Dlist-def distinct-remdups-id)
  from  $\langle \text{distinct } xs \rangle$  have P (Dlist xs)
  proof (induct xs)
    case Nil from empty show ?case by (simp add: Dlist.empty-def)
  next
    case (Cons x xs)
    then have  $\neg \text{Dlist.member (Dlist xs) } x$  and P (Dlist xs)
      by (simp-all add: Dlist.member-def)
    with insrt have P (Dlist.insert x (Dlist xs)) .
    with Cons show ?case by (simp add: Dlist.insert-def distinct-remdups-id)
  qed
  with dxs show P dxs by simp
qed

lemma dlist-case [cases type: dlist]:
  obtains (empty) dxs = Dlist.empty
    | (insert) x dys where  $\neg \text{Dlist.member dys } x$  and dxs = Dlist.insert x dys
proof (cases dxs)
  case (Abs-dlist xs)
  then have dxs: dxs = Dlist xs and distinct: distinct xs
    by (simp-all add: Dlist-def distinct-remdups-id)
  show thesis

```

```

proof (cases xs)
  case Nil with dxs
    have dxs = Dlist.empty by (simp add: Dlist.empty-def)
    with empty show ?thesis .
  next
    case (Cons x xs)
    with dxs distinct have  $\neg$  Dlist.member (Dlist xs) x
      and dxs = Dlist.insert x (Dlist xs)
      by (simp-all add: Dlist.member-def Dlist.insert-def distinct-remdups-id)
    with insert show ?thesis .
qed
qed

```

32.4 Functorial structure

```

functor map: map
  by (simp-all add: remdups-map-remdups fun-eq-iff dlist-eq-iff)

```

32.5 Quickcheck generators

quickcheck-generator *dlist predicate: distinct constructors: Dlist.empty, Dlist.insert*

32.6 BNF instance

context begin

```

qualified inductive double :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool where
  double (xs @ ys) (xs @ x # ys) if x  $\in$  set ys

```

```

qualified lemma strong-confluentp-double: strong-confluentp double
proof

```

```

  fix xs ys zs :: 'a list
  assume ys: double xs ys and zs: double xs zs
  consider (left) as y bs z cs where xs = as @ bs @ cs ys = as @ y # bs @ cs zs
= as @ bs @ z # cs y  $\in$  set (bs @ cs) z  $\in$  set cs
  | (right) as y bs z cs where xs = as @ bs @ cs ys = as @ bs @ y # cs zs = as
@ z # bs @ cs y  $\in$  set cs z  $\in$  set (bs @ cs)
  proof –
    show thesis using ys zs
    by(clarsimp simp add: double.simps append-eq-append-conv2)(auto intro: that)
  qed
  then show  $\exists$  us. double** ys us  $\wedge$  double== zs us
  proof cases
    case left
    let ?us = as @ y # bs @ z # cs
    have double ys ?us double zs ?us using left
    by(auto 4 4 simp add: double.simps)(metis append-Cons append-assoc)+
    then show ?thesis by blast
  next
    case right

```



```

    let ?us = as @ z # bs @ y # cs
    have double ys ?us double zs ?us using right
      by(auto 4 4 simp add: double.simps)(metis append-Cons append-assoc)+
    then show ?thesis by blast
  qed
qed

qualified lemma double-Cons1 [simp]: double xs (x # xs) if x ∈ set xs
  using double.intros[of x xs []] that by simp

qualified lemma double-Cons-same [simp]: double xs ys  $\implies$  double (x # xs) (x # ys)
  by(auto simp add: double.simps Cons-eq-append-conv)

qualified lemma doubles-Cons-same: double** xs ys  $\implies$  double** (x # xs) (x # ys)
  by(induction rule: rtrancpl-induct)(auto intro: rtrancpl.rtrancpl-into-rtrancpl)

qualified lemma remdups-into-doubles: double** (remdups xs) xs
  by(induction xs)(auto intro: doubles-Cons-same rtrancpl.rtrancpl-into-rtrancpl)

qualified lemma dlist-eq-into-doubles: Dlist.dlist-eq  $\leq$  equivclp double
  by(auto 4 4 simp add: Dlist.dlist-eq-def vimage2p-def
    intro: equivclp-trans converse-rtrancpl-into-equivclp rtrancpl-into-equivclp remdups-into-doubles)

qualified lemma factor-double-map: double (map f xs) ys  $\implies$   $\exists$  zs. Dlist.dlist-eq
  xs zs  $\wedge$  ys = map f zs  $\wedge$  set zs  $\subseteq$  set xs
  by(auto simp add: double.simps Dlist.dlist-eq-def vimage2p-def map-eq-append-conv)
    (metis (no-types, opaque-lifting) list.simps(9) map-append remdups.simps(2)
    remdups-append2 set-append set-eq-subset set-remdups)

qualified lemma dlist-eq-set-eq: Dlist.dlist-eq xs ys  $\implies$  set xs = set ys
  by(simp add: Dlist.dlist-eq-def vimage2p-def)(metis set-remdups)

qualified lemma dlist-eq-map-respect: Dlist.dlist-eq xs ys  $\implies$  Dlist.dlist-eq (map
  f xs) (map f ys)
  by(clarsimp simp add: Dlist.dlist-eq-def vimage2p-def)(metis remdups-map-remdups)

qualified lemma confluent-quotient-dlist:
  confluent-quotient double Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq
  Dlist.dlist-eq
    (map fst) (map snd) (map fst) (map snd) list-all2 list-all2 list-all2 set set
  by(unfold-locales)(auto intro: strong-confluentp-imp-confluentp strong-confluentp-double
    dest: factor-double-map dlist-eq-into-doubles[THEN predicate2D] dlist-eq-set-eq
    simp add: list.in-rel list.rel-compp dlist-eq-map-respect Dlist.equivp-dlist-eq equivp-imp-transp)

lifting-update dlist.lifting
lifting-forget dlist.lifting

```

end

context begin

interpretation *Quotient-dlist*: *Quotient-dlist* .

lift-bnf (*plugins del: code*) '*a dlist*

subgoal for *A B* **by**(*rule confluent-quotient.subdistributivity*[*OF Dlist.confluent-quotient-dlist*])

subgoal by(*force dest: Dlist.dlist-eq-set-eq intro: equivp-reflp*[*OF Dlist.equivp-dlist-eq*])

done

qualified lemma *list-of-dlist-transfer*[*transfer-rule*]:

bi-unique R \implies (*rel-fun* (*Quotient-dlist.pcr-dlist R*) (*list-all2 R*)) *remdups list-of-dlist*

unfolding *rel-fun-def Quotient-dlist.pcr-dlist-def qcr-dlist-def Dlist.abs-dlist-def*

by (*auto simp: Abs-dlist-inverse intro!: remdups-transfer*[*THEN rel-funD*])

lemma *list-of-dlist-map-dlist*[*simp*]:

list-of-dlist (*map-dlist f xs*) = *remdups* (*map f* (*list-of-dlist xs*))

by *transfer* (*auto simp: remdups-map-remdups*)

end

end

33 Type of dual ordered lattices

theory *Dual-Ordered-Lattice*

imports *Main*

begin

The *dual* of an ordered structure is an isomorphic copy of the underlying type, with the \leq relation defined as the inverse of the original one.

The class of lattices is closed under formation of dual structures. This means that for any theorem of lattice theory, the dualized statement holds as well; this important fact simplifies many proofs of lattice theory.

typedef '*a dual* = *UNIV* :: '*a set*

morphisms *undual dual* ..

setup-lifting *type-definition-dual*

code-datatype *dual*

lemma *dual-eqI*:

x = y **if** *undual x = undual y*

using *that* **by** *transfer assumption*

lemma *dual-eq-iff*:

x = y \longleftrightarrow *undual x = undual y*

```

by transfer simp

lemma eq-dual-iff [iff]:
   $dual\ x = dual\ y \longleftrightarrow x = y$ 
by transfer simp

lemma undual-dual [simp, code]:
   $undual\ (dual\ x) = x$ 
by transfer rule

lemma dual-undual [simp]:
   $dual\ (undual\ x) = x$ 
by transfer rule

lemma undual-comp-dual [simp]:
   $undual \circ dual = id$ 
by (simp add: fun-eq-iff)

lemma dual-comp-undual [simp]:
   $dual \circ undual = id$ 
by (simp add: fun-eq-iff)

lemma inj-dual:
  inj dual
by (rule injI) simp

lemma inj-undual:
  inj undual
by (rule injI) (rule dual-eqI)

lemma surj-dual:
  surj dual
by (rule surjI [of - undual]) simp

lemma surj-undual:
  surj undual
by (rule surjI [of - dual]) simp

lemma bij-dual:
  bij dual
using inj-dual surj-dual by (rule bijI)

lemma bij-undual:
  bij undual
using inj-undual surj-undual by (rule bijI)

instance dual :: (finite) finite
proof
  from finite have finite (range dual :: 'a dual set)

```

```

    by (rule finite-imageI)
  then show finite (UNIV :: 'a dual set)
    by (simp add: surj-dual)
qed

instantiation dual :: (equal) equal
begin

lift-definition equal-dual :: 'a dual  $\Rightarrow$  'a dual  $\Rightarrow$  bool
  is HOL.equal .

instance
  by (standard; transfer) (simp add: equal)

end

```

33.1 Pointwise ordering

```

instantiation dual :: (ord) ord
begin

lift-definition less-eq-dual :: 'a dual  $\Rightarrow$  'a dual  $\Rightarrow$  bool
  is ( $\geq$ ) .

lift-definition less-dual :: 'a dual  $\Rightarrow$  'a dual  $\Rightarrow$  bool
  is ( $>$ ) .

instance ..

end

lemma dual-less-eqI:
   $x \leq y$  if undual  $y \leq$  undual  $x$ 
  using that by transfer assumption

lemma dual-less-eq-iff:
   $x \leq y \iff$  undual  $y \leq$  undual  $x$ 
  by transfer simp

lemma less-eq-dual-iff [iff]:
  dual  $x \leq$  dual  $y \iff y \leq x$ 
  by transfer simp

lemma dual-lessI:
   $x < y$  if undual  $y <$  undual  $x$ 
  using that by transfer assumption

lemma dual-less-iff:
   $x < y \iff$  undual  $y <$  undual  $x$ 

```

by *transfer simp*

lemma *less-dual-iff* [*iff*]:

$dual\ x < dual\ y \longleftrightarrow y < x$

by *transfer simp*

instance *dual* :: (*preorder*) *preorder*

by (*standard*; *transfer*) (*auto simp add: less-le-not-le intro: order-trans*)

instance *dual* :: (*order*) *order*

by (*standard*; *transfer*) *simp*

33.2 Binary infimum and supremum

instantiation *dual* :: (*sup*) *inf*

begin

lift-definition *inf-dual* :: '*a dual* \Rightarrow '*a dual* \Rightarrow '*a dual*

is *sup* .

instance ..

end

lemma *undual-inf-eq* [*simp*]:

$undual\ (inf\ x\ y) = sup\ (undual\ x)\ (undual\ y)$

by (*fact inf-dual.rep-eq*)

lemma *dual-sup-eq* [*simp*]:

$dual\ (sup\ x\ y) = inf\ (dual\ x)\ (dual\ y)$

by *transfer rule*

instantiation *dual* :: (*inf*) *sup*

begin

lift-definition *sup-dual* :: '*a dual* \Rightarrow '*a dual* \Rightarrow '*a dual*

is *inf* .

instance ..

end

lemma *undual-sup-eq* [*simp*]:

$undual\ (sup\ x\ y) = inf\ (undual\ x)\ (undual\ y)$

by (*fact sup-dual.rep-eq*)

lemma *dual-inf-eq* [*simp*]:

$dual\ (inf\ x\ y) = sup\ (dual\ x)\ (dual\ y)$

by *transfer simp*

```
instance dual :: (semilattice-sup) semilattice-inf
  by (standard; transfer) simp-all
```

```
instance dual :: (semilattice-inf) semilattice-sup
  by (standard; transfer) simp-all
```

```
instance dual :: (lattice) lattice ..
```

```
instance dual :: (distrib-lattice) distrib-lattice
  by (standard; transfer) (fact inf-sup-distrib1)
```

33.3 Top and bottom elements

```
instantiation dual :: (top) bot
begin
```

```
lift-definition bot-dual :: 'a dual
  is top .
```

```
instance ..
```

```
end
```

```
lemma undual-bot-eq [simp]:
  undual bot = top
  by (fact bot-dual.rep-eq)
```

```
lemma dual-top-eq [simp]:
  dual top = bot
  by transfer rule
```

```
instantiation dual :: (bot) top
begin
```

```
lift-definition top-dual :: 'a dual
  is bot .
```

```
instance ..
```

```
end
```

```
lemma undual-top-eq [simp]:
  undual top = bot
  by (fact top-dual.rep-eq)
```

```
lemma dual-bot-eq [simp]:
  dual bot = top
  by transfer rule
```

```

instance dual :: (order-top) order-bot
  by (standard; transfer) simp

instance dual :: (order-bot) order-top
  by (standard; transfer) simp

instance dual :: (bounded-lattice-top) bounded-lattice-bot ..

instance dual :: (bounded-lattice-bot) bounded-lattice-top ..

instance dual :: (bounded-lattice) bounded-lattice ..

```

33.4 Complement

```

instantiation dual :: (uminus) uminus
begin

lift-definition uminus-dual :: 'a dual  $\Rightarrow$  'a dual
  is uminus .

instance ..

end

lemma undual-uminus-eq [simp]:
  undual (− x) = − undual x
  by (fact uminus-dual.rep-eq)

lemma dual-uminus-eq [simp]:
  dual (− x) = − dual x
  by transfer rule

instantiation dual :: (boolean-algebra) boolean-algebra
begin

lift-definition minus-dual :: 'a dual  $\Rightarrow$  'a dual  $\Rightarrow$  'a dual
  is  $\lambda x y. - (y - x)$  .

instance
  by (standard; transfer) (simp-all add: diff-eq ac-simps)

end

lemma undual-minus-eq [simp]:
  undual (x − y) = − (undual y − undual x)
  by (fact minus-dual.rep-eq)

lemma dual-minus-eq [simp]:

```

$dual (x - y) = - (dual y - dual x)$
by *transfer simp*

33.5 Complete lattice operations

The class of complete lattices is closed under formation of dual structures.

instantiation $dual :: (Sup) Inf$
begin

lift-definition $Inf-dual :: 'a dual set \Rightarrow 'a dual$
is Sup .

instance ..

end

lemma $undual-Inf-eq [simp]$:
 $undual (Inf A) = Sup (undual ' A)$
by (*fact Inf-dual.rep-eq*)

lemma $dual-Sup-eq [simp]$:
 $dual (Sup A) = Inf (dual ' A)$
by *transfer simp*

instantiation $dual :: (Inf) Sup$
begin

lift-definition $Sup-dual :: 'a dual set \Rightarrow 'a dual$
is Inf .

instance ..

end

lemma $undual-Sup-eq [simp]$:
 $undual (Sup A) = Inf (undual ' A)$
by (*fact Sup-dual.rep-eq*)

lemma $dual-Inf-eq [simp]$:
 $dual (Inf A) = Sup (dual ' A)$
by *transfer simp*

instance $dual :: (complete-lattice) complete-lattice$
by (*standard; transfer*) (*auto intro: Inf-lower Sup-upper Inf-greatest Sup-least*)

context

fixes $f :: 'a::complete-lattice \Rightarrow 'a$
and $g :: 'a dual \Rightarrow 'a dual$
assumes *mono f*

defines $g \equiv \text{dual} \circ f \circ \text{undual}$
begin

private lemma *mono-dual*:

mono g

proof

fix $x\ y :: 'a\ \text{dual}$

assume $x \leq y$

then have $\text{undual}\ y \leq \text{undual}\ x$

by (*simp add: dual-less-eq-iff*)

with $\langle \text{mono } f \rangle$ **have** $f\ (\text{undual}\ y) \leq f\ (\text{undual}\ x)$

by (*rule monoD*)

then have $(\text{dual} \circ f \circ \text{undual})\ x \leq (\text{dual} \circ f \circ \text{undual})\ y$

by *simp*

then show $g\ x \leq g\ y$

by (*simp add: g-def*)

qed

lemma *lfp-dual-gfp*:

$\text{lfp}\ f = \text{undual}\ (\text{gfp}\ g)$ (**is** $?lhs = ?rhs$)

proof (*rule antisym*)

have $\text{dual}\ (\text{undual}\ (g\ (\text{gfp}\ g))) \leq \text{dual}\ (f\ (\text{undual}\ (\text{gfp}\ g)))$

by (*simp add: g-def*)

with *mono-dual* **have** $f\ (\text{undual}\ (\text{gfp}\ g)) \leq \text{undual}\ (\text{gfp}\ g)$

by (*simp add: gfp-unfold [where f = g, symmetric] dual-less-eq-iff*)

then show $?lhs \leq ?rhs$

by (*rule lfp-lowerbound*)

from $\langle \text{mono } f \rangle$ **have** $\text{dual}\ (\text{lfp}\ f) \leq \text{dual}\ (\text{undual}\ (\text{gfp}\ g))$

by (*simp add: lfp-fixpoint gfp-upperbound g-def*)

then show $?rhs \leq ?lhs$

by (*simp only: less-eq-dual-iff*)

qed

lemma *gfp-dual-lfp*:

$\text{gfp}\ f = \text{undual}\ (\text{lfp}\ g)$

proof –

have *mono* $(\lambda x. \text{undual}\ (\text{undual}\ x))$

by (*rule monoI*) (*simp add: dual-less-eq-iff*)

moreover have *mono* $(\lambda a. \text{dual}\ (\text{dual}\ (f\ a)))$

using $\langle \text{mono } f \rangle$ **by** (*auto intro: monoI dest: monoD*)

moreover have $\text{gfp}\ f = \text{gfp}\ (\lambda x. \text{undual}\ (\text{undual}\ (\text{dual}\ (\text{dual}\ (f\ x)))))$

by *simp*

ultimately have $\text{undual}\ (\text{undual}\ (\text{gfp}\ (\lambda x. \text{dual}$

$(\text{dual}\ (f\ (\text{undual}\ (\text{undual}\ x)))))) =$

$\text{gfp}\ (\lambda x. \text{undual}\ (\text{undual}\ (\text{dual}\ (\text{dual}\ (f\ x)))))$

by (*subst gfp-rolling [where g = $\lambda x. \text{undual}\ (\text{undual}\ x)$] simp-all*)

then have $\text{gfp}\ f =$

undual

$(\text{undual}$

```

      (gfp (λx. dual (dual (f (undual (undual x)))))))
    by simp
  also have ... = undual (undual (gfp (dual ∘ g ∘ undual)))
    by (simp add: comp-def g-def)
  also have ... = undual (lfp g)
    using mono-dual by (simp only: Dual-Ordered-Lattice.lfp-dual-gfp)
  finally show ?thesis .
qed

end

  Finally
lifting-update dual.lifting
lifting-forget dual.lifting

end

```

34 Equipollence and Other Relations Connected with Cardinality

```

theory Equipollence
  imports FuncSet Countable-Set
begin

```

34.1 Eqpoll

```

definition eqpoll :: 'a set ⇒ 'b set ⇒ bool (infixl <≈> 50)
  where eqpoll A B ≡ ∃ f. bij-betw f A B

definition lepoll :: 'a set ⇒ 'b set ⇒ bool (infixl <≲> 50)
  where lepoll A B ≡ ∃ f. inj-on f A ∧ f ' A ⊆ B

definition lesspoll :: 'a set ⇒ 'b set ⇒ bool (infixl <≺> 50)
  where A ≺ B == A ≲ B ∧ ¬(A ≈ B)

lemma lepoll-def': lepoll A B ≡ ∃ f. inj-on f A ∧ f ∈ A → B
  by (simp add: Pi-iff image-subset-iff lepoll-def)

lemma eqpoll-empty-iff-empty [simp]: A ≈ {} ⟷ A = {}
  by (simp add: bij-betw-iff-bijections eqpoll-def)

lemma lepoll-empty-iff-empty [simp]: A ≲ {} ⟷ A = {}
  by (auto simp: lepoll-def)

lemma not-lesspoll-empty: ¬ A ≺ {}
  by (simp add: lesspoll-def)

```

lemma *lepoll-relational-full*:

assumes $\bigwedge y. y \in B \implies \exists x. x \in A \wedge R\ x\ y$

and $\bigwedge x\ y\ y'. \llbracket x \in A; y \in B; y' \in B; R\ x\ y; R\ x\ y' \rrbracket \implies y = y'$

shows $B \lesssim A$

proof –

obtain f **where** $f: \bigwedge y. y \in B \implies f\ y \in A \wedge R\ (f\ y)\ y$

using *assms* **by** *metis*

with *assms* **have** *inj-on* $f\ B$

by (*metis inj-onI*)

with f **show** *?thesis*

unfolding *lepoll-def* **by** *blast*

qed

lemma *eqpoll-iff-card-of-ordIso*: $A \approx B \longleftrightarrow \text{ordIso2}\ (\text{card-of}\ A)\ (\text{card-of}\ B)$

by (*simp add: card-of-ordIso eqpoll-def*)

lemma *eqpoll-refl [iff]*: $A \approx A$

by (*simp add: card-of-refl eqpoll-iff-card-of-ordIso*)

lemma *eqpoll-finite-iff*: $A \approx B \implies \text{finite}\ A \longleftrightarrow \text{finite}\ B$

by (*meson bij-betw-finite eqpoll-def*)

lemma *eqpoll-iff-card*:

assumes *finite* A *finite* B

shows $A \approx B \longleftrightarrow \text{card}\ A = \text{card}\ B$

using *assms* **by** (*auto simp: bij-betw-iff-card eqpoll-def*)

lemma *eqpoll-singleton-iff*: $A \approx \{x\} \longleftrightarrow (\exists u. A = \{u\})$

by (*metis card.infinite card-1-singleton-iff eqpoll-finite-iff eqpoll-iff-card not-less-eq-eq*)

lemma *eqpoll-doubleton-iff*: $A \approx \{x, y\} \longleftrightarrow (\exists u\ v. A = \{u, v\} \wedge (u=v \longleftrightarrow x=y))$

proof (*cases* $x=y$)

case *True*

then show *?thesis*

by (*simp add: eqpoll-singleton-iff*)

next

case *False*

then show *?thesis*

by (*smt (verit, ccfv-threshold) card-1-singleton-iff card-Suc-eq-finite eqpoll-finite-iff eqpoll-iff-card finite.insertI singleton-iff*)

qed

lemma *lepoll-antisym*:

assumes $A \lesssim B\ B \lesssim A$ **shows** $A \approx B$

using *assms* **unfolding** *eqpoll-def lepoll-def* **by** (*metis Schroeder-Bernstein*)

lemma *lepoll-trans [trans]*:

assumes $A \lesssim B\ B \lesssim C$ **shows** $A \lesssim C$

proof –

```

obtain  $f\ g$  where  $fg$ :  $\text{inj-on } f\ A\ \text{inj-on } g\ B$  and  $f : A \rightarrow B\ g \in B \rightarrow C$ 
  by (metis assms lepoll-def')
then have  $g \circ f \in A \rightarrow C$ 
  by auto
with  $fg$  show ?thesis
  unfolding lepoll-def
  by (metis  $\langle f \in A \rightarrow B \rangle\ \text{comp-inj-on}\ \text{image-subset-iff-funcset}\ \text{inj-on-subset}$ )
qed

```

```

lemma lepoll-trans1 [trans]:  $\llbracket A \approx B; B \lesssim C \rrbracket \implies A \lesssim C$ 
  by (meson card-of-ordLeq eqpoll-iff-card-of-ordIso lepoll-def lepoll-trans ordIso-iff-ordLeq)

```

```

lemma lepoll-trans2 [trans]:  $\llbracket A \lesssim B; B \approx C \rrbracket \implies A \lesssim C$ 
  by (metis bij-betw-def eqpoll-def lepoll-def lepoll-trans order-refl)

```

```

lemma eqpoll-sym:  $A \approx B \implies B \approx A$ 
  unfolding eqpoll-def
  using bij-betw-the-inv-into by auto

```

```

lemma eqpoll-trans [trans]:  $\llbracket A \approx B; B \approx C \rrbracket \implies A \approx C$ 
  unfolding eqpoll-def using bij-betw-trans by blast

```

```

lemma eqpoll-imp-lepoll:  $A \approx B \implies A \lesssim B$ 
  unfolding eqpoll-def lepoll-def by (metis bij-betw-def order-refl)

```

```

lemma subset-imp-lepoll:  $A \subseteq B \implies A \lesssim B$ 
  by (force simp: lepoll-def)

```

```

lemma lepoll-refl [iff]:  $A \lesssim A$ 
  by (simp add: subset-imp-lepoll)

```

```

lemma lepoll-iff:  $A \lesssim B \longleftrightarrow (\exists g. A \subseteq g \text{ ‘ } B)$ 
  unfolding lepoll-def
proof safe
  fix  $g$  assume  $A \subseteq g \text{ ‘ } B$ 
  then show  $\exists f. \text{inj-on } f\ A \wedge f \text{ ‘ } A \subseteq B$ 
    by (rule-tac  $x=\text{inv-into } B\ g$  in exI) (auto simp: inv-into-into inj-on-inv-into)
qed (metis image-mono the-inv-into-onto)

```

```

lemma empty-lepoll [iff]:  $\{\} \lesssim A$ 
  by (simp add: lepoll-iff)

```

```

lemma subset-image-lepoll:  $B \subseteq f \text{ ‘ } A \implies B \lesssim A$ 
  by (auto simp: lepoll-iff)

```

```

lemma image-lepoll:  $f \text{ ‘ } A \lesssim A$ 
  by (auto simp: lepoll-iff)

```

```

lemma infinite-le-lepoll:  $\text{infinite } A \longleftrightarrow (\text{UNIV}::\text{nat set}) \lesssim A$ 

```

by (simp add: infinite-iff-countable-subset lepoll-def)

lemma lepoll-Pow-self: $A \lesssim \text{Pow } A$
unfolding lepoll-def inj-def
proof (intro exI conjI)
 show inj-on $(\lambda x. \{x\}) A$
 by (auto simp: inj-on-def)
qed auto

lemma eqpoll-iff-bijections:
 $A \approx B \longleftrightarrow (\exists f g. (\forall x \in A. f x \in B \wedge g(f x) = x) \wedge (\forall y \in B. g y \in A \wedge f(g y) = y))$
 by (auto simp: eqpoll-def bij-betw-iff-bijections)

lemma lepoll-restricted-funspace:
 $\{f. f ' A \subseteq B \wedge \{x. f x \neq k x\} \subseteq A \wedge \text{finite } \{x. f x \neq k x\}\} \lesssim \text{Fpow } (A \times B)$
proof –
 have *: $\exists U \in \text{Fpow } (A \times B). f = (\lambda x. \text{if } \exists y. (x, y) \in U \text{ then SOME } y. (x, y) \in U \text{ else } k x)$
 if $f ' A \subseteq B \wedge \{x. f x \neq k x\} \subseteq A \wedge \text{finite } \{x. f x \neq k x\}$ **for** f
 apply (rule-tac $x = (\lambda x. (x, f x))$) ‘ $\{x. f x \neq k x\}$ ’ **in** *beXI*
 using that **by** (auto simp: image-def Fpow-def)
 show ?thesis
 apply (rule subset-image-lepoll [where $f = \lambda U x. \text{if } \exists y. (x, y) \in U \text{ then } @y. (x, y) \in U \text{ else } k x$])
 using * **by** (auto simp: image-def)
qed

lemma singleton-lepoll: $\{x\} \lesssim \text{insert } y A$
 by (force simp: lepoll-def)

lemma singleton-epoll: $\{x\} \approx \{y\}$
 by (blast intro: lepoll-antisym singleton-lepoll)

lemma subset-singleton-iff-lepoll: $(\exists x. S \subseteq \{x\}) \longleftrightarrow S \lesssim \{()\}$
 using lepoll-iff **by** fastforce

lemma infinite-insert-lepoll:
 assumes infinite A **shows** $\text{insert } a A \lesssim A$
proof –
 obtain $f :: \text{nat} \Rightarrow 'a$ **where** inj f **and** $f: \text{range } f \subseteq A$
 using assms infinite-countable-subset **by** blast
 let ? $g = (\lambda z. \text{if } z = a \text{ then } f 0 \text{ else if } z \in \text{range } f \text{ then } f (\text{Suc } (\text{inv } f z)) \text{ else } z)$
 show ?thesis
 unfolding lepoll-def
proof (intro exI conjI)
 show inj-on ? g (insert $a A$)
 using inj-on-eq-iff [OF ‘inj f ’]
 by (auto simp: inj-on-def)

```

  show ?g ‘ insert a A  $\subseteq$  A
    using f by auto
qed
qed

```

lemma *infinite-insert-epoll*: $\text{infinite } A \implies \text{insert } a \ A \approx A$
 by (*simp add: lepoll-antisym infinite-insert-lepoll subset-imp-lepoll subset-insertI*)

lemma *finite-lepoll-infinite*:
 assumes *infinite A finite B* shows $B \lesssim A$
proof –
 have $B \lesssim (\text{UNIV}::\text{nat set})$
 unfolding *lepoll-def*
 using *finite-imp-inj-to-nat-seg* [*OF* $\langle \text{finite } B \rangle$] by blast
 then show ?thesis
 using $\langle \text{infinite } A \rangle$ *infinite-le-lepoll lepoll-trans* by auto
qed

lemma *countable-lepoll*: $\llbracket \text{countable } A; B \lesssim A \rrbracket \implies \text{countable } B$
 by (*meson countable-image countable-subset lepoll-iff*)

lemma *countable-epoll*: $\llbracket \text{countable } A; B \approx A \rrbracket \implies \text{countable } B$
 using *countable-lepoll eqpoll-imp-lepoll* by blast

34.2 The strict relation

lemma *lesspoll-not-refl* [*iff*]: $\sim (i \prec i)$
 by (*simp add: lepoll-antisym lesspoll-def*)

lemma *lesspoll-imp-lepoll*: $A \prec B \implies A \lesssim B$
 by (*unfold lesspoll-def, blast*)

lemma *lepoll-iff-leqpoll*: $A \lesssim B \longleftrightarrow A \prec B \mid A \approx B$
 using *eqpoll-imp-lepoll lesspoll-def* by blast

lemma *lesspoll-trans* [*trans*]: $\llbracket X \prec Y; Y \prec Z \rrbracket \implies X \prec Z$
 by (*meson eqpoll-sym lepoll-antisym lepoll-trans lepoll-trans1 lesspoll-def*)

lemma *lesspoll-trans1* [*trans*]: $\llbracket X \lesssim Y; Y \prec Z \rrbracket \implies X \prec Z$
 by (*meson eqpoll-sym lepoll-antisym lepoll-trans lepoll-trans1 lesspoll-def*)

lemma *lesspoll-trans2* [*trans*]: $\llbracket X \prec Y; Y \lesssim Z \rrbracket \implies X \prec Z$
 by (*meson eqpoll-imp-lepoll eqpoll-sym lepoll-antisym lepoll-trans lesspoll-def*)

lemma *eq-lesspoll-trans* [*trans*]: $\llbracket X \approx Y; Y \prec Z \rrbracket \implies X \prec Z$
 using *eqpoll-imp-lepoll lesspoll-trans1* by blast

lemma *lesspoll-eq-trans* [*trans*]: $\llbracket X \prec Y; Y \approx Z \rrbracket \implies X \prec Z$
 using *eqpoll-imp-lepoll lesspoll-trans2* by blast

lemma *lesspoll-Pow-self*: $A \prec \text{Pow } A$
unfolding *lesspoll-def* *bij-betw-def* *eqpoll-def*
by (*meson* *lepoll-Pow-self* *Cantors-theorem*)

lemma *finite-lesspoll-infinite*:
assumes *infinite* A *finite* B **shows** $B \prec A$
by (*meson* *assms* *eqpoll-finite-iff* *finite-lepoll-infinite* *lesspoll-def*)

lemma *countable-lesspoll*: $\llbracket \text{countable } A; B \prec A \rrbracket \implies \text{countable } B$
using *countable-lepoll* *lesspoll-def* **by** *blast*

lemma *lepoll-iff-card-le*: $\llbracket \text{finite } A; \text{finite } B \rrbracket \implies A \lesssim B \longleftrightarrow \text{card } A \leq \text{card } B$
by (*simp* *add*: *inj-on-iff-card-le* *lepoll-def*)

lemma *lepoll-iff-finite-card*: $A \lesssim \{..<n::\text{nat}\} \longleftrightarrow \text{finite } A \wedge \text{card } A \leq n$
by (*metis* *card-lessThan* *finite-lessThan* *finite-surj* *lepoll-iff* *lepoll-iff-card-le*)

lemma *eqpoll-iff-finite-card*: $A \approx \{..<n::\text{nat}\} \longleftrightarrow \text{finite } A \wedge \text{card } A = n$
by (*metis* *card-lessThan* *eqpoll-finite-iff* *eqpoll-iff-card* *finite-lessThan*)

lemma *lesspoll-iff-finite-card*: $A \prec \{..<n::\text{nat}\} \longleftrightarrow \text{finite } A \wedge \text{card } A < n$
by (*metis* *eqpoll-iff-finite-card* *lepoll-iff-finite-card* *lesspoll-def* *order-less-le*)

34.3 Mapping by an injection

lemma *inj-on-image-eqpoll-self*: $\text{inj-on } f \ A \implies f' \ A \approx A$
by (*meson* *bij-betw-def* *eqpoll-def* *eqpoll-sym*)

lemma *inj-on-image-lepoll-1* [*simp*]:
assumes $\text{inj-on } f \ A$ **shows** $f' \ A \lesssim B \longleftrightarrow A \lesssim B$
by (*meson* *assms* *image-lepoll* *lepoll-def* *lepoll-trans* *order-refl*)

lemma *inj-on-image-lepoll-2* [*simp*]:
assumes $\text{inj-on } f \ B$ **shows** $A \lesssim f' \ B \longleftrightarrow A \lesssim B$
by (*meson* *assms* *eq-iff* *image-lepoll* *lepoll-def* *lepoll-trans*)

lemma *inj-on-image-lesspoll-1* [*simp*]:
assumes $\text{inj-on } f \ A$ **shows** $f' \ A \prec B \longleftrightarrow A \prec B$
by (*meson* *assms* *image-lepoll* *le-less* *lepoll-def* *lesspoll-trans1*)

lemma *inj-on-image-lesspoll-2* [*simp*]:
assumes $\text{inj-on } f \ B$ **shows** $A \prec f' \ B \longleftrightarrow A \prec B$
by (*meson* *assms* *eqpoll-sym* *inj-on-image-eqpoll-self* *lesspoll-eq-trans*)

lemma *inj-on-image-eqpoll-1* [*simp*]:
assumes $\text{inj-on } f \ A$ **shows** $f' \ A \approx B \longleftrightarrow A \approx B$
by (*metis* *assms* *eqpoll-trans* *inj-on-image-eqpoll-self* *eqpoll-sym*)

lemma *inj-on-image-epoll-2* [simp]:
assumes *inj-on f B* **shows** $A \approx f \cdot B \longleftrightarrow A \approx B$
by (*metis assms inj-on-image-epoll-1 epoll-sym*)

34.4 Inserting elements into sets

lemma *insert-lepoll-insertD*:
assumes *insert u A \lesssim insert v B* *u \notin A* *v \notin B* **shows** $A \lesssim B$
proof –
obtain *f* **where** *inj: inj-on f (insert u A)* **and** *fim: f \cdot (insert u A) \subseteq insert v B
by (*meson assms lepoll-def*)
show ?thesis
unfolding *lepoll-def*
proof (*intro exI conjI*)
let ?g = $\lambda x \in A. \text{if } f x = v \text{ then } f u \text{ else } f x$
show *inj-on ?g A*
using *inj $\langle u \notin A \rangle$* **by** (*auto simp: inj-on-def*)
show ?g \cdot $A \subseteq B$
using *fim $\langle u \notin A \rangle$* *image-subset-iff inj inj-on-image-mem-iff* **by** *fastforce*
qed
qed*

lemma *insert-epoll-insertD*: $\llbracket \text{insert } u A \approx \text{insert } v B; u \notin A; v \notin B \rrbracket \implies A \approx B$
by (*meson insert-lepoll-insertD epoll-imp-lepoll epoll-sym lepoll-antisym*)

lemma *insert-lepoll-cong*:
assumes $A \lesssim B$ *b \notin B* **shows** *insert a A \lesssim insert b B*
proof –
obtain *f* **where** *f: inj-on f A* *f \cdot A \subseteq B
by (*meson assms lepoll-def*)
let ?f = $\lambda u \in \text{insert } a A. \text{if } u=a \text{ then } b \text{ else } f u$
show ?thesis
unfolding *lepoll-def*
proof (*intro exI conjI*)
show *inj-on ?f (insert a A)*
using *f $\langle b \notin B \rangle$* **by** (*auto simp: inj-on-def*)
show ?f \cdot $\text{insert } a A \subseteq \text{insert } b B$
using *f $\langle b \notin B \rangle$* **by** *auto*
qed
qed*

lemma *insert-epoll-cong*:
 $\llbracket A \approx B; a \notin A; b \notin B \rrbracket \implies \text{insert } a A \approx \text{insert } b B$
by (*meson epoll-imp-lepoll epoll-sym insert-lepoll-cong lepoll-antisym*)

lemma *insert-epoll-insert-iff*:
 $\llbracket a \notin A; b \notin B \rrbracket \implies \text{insert } a A \approx \text{insert } b B \longleftrightarrow A \approx B$
by (*meson insert-epoll-insertD insert-epoll-cong*)

lemma *insert-lepoll-insert-iff*:

$\llbracket a \notin A; b \notin B \rrbracket \implies (\text{insert } a \ A \lesssim \text{insert } b \ B) \longleftrightarrow (A \lesssim B)$

by (*meson insert-lepoll-insertD insert-lepoll-cong*)

lemma *less-imp-insert-lepoll*:

assumes $A \prec B$ **shows** $\text{insert } a \ A \lesssim B$

proof –

obtain f **where** $\text{inj-on } f \ A \ f \ ' \ A \subseteq B$

using *assms* **by** (*metis bij-betw-def eqpoll-def lepoll-def lesspoll-def psubset-eq*)

then obtain b **where** $b \in B \ b \notin f \ ' \ A$

by *auto*

show *?thesis*

unfolding *lepoll-def*

proof (*intro exI conjI*)

show $\text{inj-on } (f(a:=b)) \ (\text{insert } a \ A)$

using $b \in \text{inj-on } f \ A$ **by** (*auto simp: inj-on-def*)

show $(f(a:=b)) \ ' \ \text{insert } a \ A \subseteq B$

using $\langle f \ ' \ A \subseteq B \rangle$ **by** (*auto simp: b*)

qed

qed

lemma *finite-insert-lepoll*: $\text{finite } A \implies (\text{insert } a \ A \lesssim A) \longleftrightarrow (a \in A)$

proof (*induction A rule: finite-induct*)

case ($\text{insert } x \ A$)

then show *?case*

by (*metis insertI2 insert-lepoll-insert-iff insert-subsetI lepoll-trans subsetI subset-imp-lepoll*)

qed *auto*

34.5 Binary sums and unions

lemma *Un-lepoll-mono*:

assumes $A \lesssim C \ B \lesssim D \ \text{disjnt } C \ D$ **shows** $A \cup B \lesssim C \cup D$

proof –

obtain $f \ g$ **where** $\text{inj: inj-on } f \ A \ \text{inj-on } g \ B$ **and** $fg: f \ ' \ A \subseteq C \ g \ ' \ B \subseteq D$

by (*meson assms lepoll-def*)

have $\text{inj-on } (\lambda x. \text{if } x \in A \text{ then } f \ x \text{ else } g \ x) \ (A \cup B)$

using $\text{inj } \langle \text{disjnt } C \ D \rangle \ fg$ **unfolding** *disjnt-iff*

by (*fastforce intro: inj-onI dest: inj-on-contrad split: if-split-asm*)

with fg **show** *?thesis*

unfolding *lepoll-def*

by (*rule-tac x= $\lambda x. \text{if } x \in A \text{ then } f \ x \text{ else } g \ x$ in exI*) *auto*

qed

lemma *Un-eqpoll-cong*: $\llbracket A \approx C; B \approx D; \text{disjnt } A \ B; \text{disjnt } C \ D \rrbracket \implies A \cup B \approx C \cup D$

by (*meson Un-lepoll-mono eqpoll-imp-lepoll eqpoll-sym lepoll-antisym*)

lemma *sum-lepoll-mono*:

assumes $A \lesssim C \lesssim D$ **shows** $A <+> B \lesssim C <+> D$
proof –
obtain $f\ g$ **where** $\text{inj-on } f\ A\ f\ ' A \subseteq C\ \text{inj-on } g\ B\ g\ ' B \subseteq D$
by (*meson assms lepoll-def*)
then show ?thesis
unfolding lepoll-def
by (*rule-tac x=case-sum (Inl ∘ f) (Inr ∘ g) in exI (force simp: inj-on-def)*)
qed

lemma *sum-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A <+> B \approx C <+> D$
by (*meson eqpoll-imp-lepoll eqpoll-sym lepoll-antisym sum-lepoll-mono*)

34.6 Binary Cartesian products

lemma *times-square-lepoll*: $A \lesssim A \times A$
unfolding lepoll-def inj-def
proof (*intro exI conjI*)
show $\text{inj-on } (\lambda x. (x, x))\ A$
by (*auto simp: inj-on-def*)
qed *auto*

lemma *times-commute-epoll*: $A \times B \approx B \times A$
unfolding eqpoll-def
by (*force intro: bij-betw-byWitness [where $f = \lambda(x, y). (y, x)$ and $f' = \lambda(x, y). (y, x)$]*)

lemma *times-assoc-epoll*: $(A \times B) \times C \approx A \times (B \times C)$
unfolding eqpoll-def
by (*force intro: bij-betw-byWitness [where $f = \lambda((x, y), z). (x, (y, z))$ and $f' = \lambda(x, (y, z)). ((x, y), z)$]*)

lemma *times-singleton-epoll*: $\{a\} \times A \approx A$
proof –
have $\{a\} \times A = (\lambda x. (a, x))\ ' A$
by *auto*
also have $\dots \approx A$
proof (*rule inj-on-image-epoll-self*)
show $\text{inj-on } (\text{Pair } a)\ A$
by (*auto simp: inj-on-def*)
qed
finally show ?thesis .
qed

lemma *times-lepoll-mono*:
assumes $A \lesssim C \lesssim D$ **shows** $A \times B \lesssim C \times D$
proof –
obtain $f\ g$ **where** $\text{inj-on } f\ A\ f\ ' A \subseteq C\ \text{inj-on } g\ B\ g\ ' B \subseteq D$
by (*meson assms lepoll-def*)
then show ?thesis

unfolding *lepoll-def*
by (*rule-tac* $x=\lambda(x,y). (f\ x, g\ y)$ **in** *exI*) (*auto simp: inj-on-def*)
qed

lemma *times-epoll-cong*: $\llbracket A \approx C; B \approx D \rrbracket \implies A \times B \approx C \times D$
by (*metis eqpoll-imp-lepoll eqpoll-sym lepoll-antisym times-lepoll-mono*)

lemma
assumes $B \neq \{\}$ **shows** *lepoll-times1*: $A \lesssim A \times B$ **and** *lepoll-times2*: $A \lesssim B \times A$
using *assms lepoll-iff* **by** *fastforce+*

lemma *times-0-epoll*: $\{\} \times A \approx \{\}$
by (*simp add: eqpoll-iff-bijections*)

lemma *Sigma-inj-lepoll-mono*:
assumes h : *inj-on* $h\ A\ h\ 'A \subseteq C$ **and** $\bigwedge x. x \in A \implies B\ x \lesssim D\ (h\ x)$
shows *Sigma* $A\ B \lesssim$ *Sigma* $C\ D$
proof –
have $\bigwedge x. x \in A \implies \exists f. \text{inj-on } f\ (B\ x) \wedge f\ ' (B\ x) \subseteq D\ (h\ x)$
by (*meson assms lepoll-def*)
then obtain f **where** $\bigwedge x. x \in A \implies \text{inj-on } (f\ x)\ (B\ x) \wedge f\ x\ ' B\ x \subseteq D\ (h\ x)$
by *metis*
with h **show** *?thesis*
unfolding *lepoll-def inj-on-def*
by (*rule-tac* $x=\lambda(x,y). (h\ x, f\ x\ y)$ **in** *exI*) *force*
qed

lemma *Sigma-lepoll-mono*:
assumes $A \subseteq C$ $\bigwedge x. x \in A \implies B\ x \lesssim D\ x$ **shows** *Sigma* $A\ B \lesssim$ *Sigma* $C\ D$
using *Sigma-inj-lepoll-mono* [*of id*] *assms* **by** *auto*

lemma *sum-times-distrib-epoll*: $(A\ <+>\ B) \times C \approx (A \times C)\ <+>\ (B \times C)$
unfolding *eqpoll-def*
proof
show *bij-betw* $(\lambda(x,z). \text{case-sum}(\lambda y. \text{Inl}(y,z))\ (\lambda y. \text{Inr}(y,z))\ x)\ ((A\ <+>\ B) \times C)\ (A \times C\ <+>\ B \times C)$
by (*rule bij-betw-byWitness* [**where** $f' = \text{case-sum } (\lambda(x,z). (\text{Inl } x, z))\ (\lambda(y,z). (\text{Inr } y, z))$] *auto*)
qed

lemma *Sigma-epoll-cong*:
assumes h : *bij-betw* $h\ A\ C$ **and** BD : $\bigwedge x. x \in A \implies B\ x \approx D\ (h\ x)$
shows *Sigma* $A\ B \approx$ *Sigma* $C\ D$
proof (*intro lepoll-antisym*)
show *Sigma* $A\ B \lesssim$ *Sigma* $C\ D$
by (*metis Sigma-inj-lepoll-mono bij-betw-def eqpoll-imp-lepoll subset-refl assms*)
have *inj-on* $(\text{inv-into } A\ h)\ C \wedge \text{inv-into } A\ h\ ' C \subseteq A$
by (*metis bij-betw-def bij-betw-inv-into h set-eq-subset*)

then show $\text{Sigma } C \ D \lesssim \text{Sigma } A \ B$
by (*smt* (*verit*, *best*) *BD Sigma-inj-lepoll-mono bij-betw-inv-into-right eqpoll-sym*
h image-subset-iff lepoll-refl lepoll-trans2)
qed

lemma *prod-insert-epoll*:

assumes $a \notin A$ **shows** $\text{insert } a \ A \times B \approx B <+> A \times B$
unfolding *eqpoll-def*
proof
show *bij-betw* ($\lambda(x,y). \text{ if } x=a \text{ then } \text{Inl } y \text{ else } \text{Inr } (x,y)$) (*insert* $a \ A \times B$) ($B <+> A \times B$)
by (*rule bij-betw-byWitness* [**where** $f' = \text{case-sum } (\lambda y. (a,y)) \text{ id}$] (*auto simp: assms*))
qed

34.7 General Unions

lemma *Union-epoll-Times*:

assumes $B: \bigwedge x. x \in A \implies F \ x \approx B$ **and** *disj: pairwise* ($\lambda x \ y. \text{disjnt } (F \ x) \ (F \ y)$) A
shows $\bigcup_{x \in A} F \ x \approx A \times B$
proof (*rule lepoll-antisym*)
obtain b **where** $b: \bigwedge x. x \in A \implies \text{bij-betw } (b \ x) \ (F \ x) \ B$
using B **unfolding** *eqpoll-def* **by** *metis*
show $\bigcup (F \ ' A) \lesssim A \times B$
unfolding *lepoll-def*
proof (*intro exI conjI*)
define χ **where** $\chi \equiv \lambda z. \text{THE } x. x \in A \wedge z \in F \ x$
have $\chi: \chi \ z = x \text{ if } x \in A \ z \in F \ x \text{ for } x \ z$
unfolding $\chi\text{-def}$
by (*smt* (*verit*, *best*) *disj disjnt-iff pairwiseD that(1,2) theI-unique*)
let $?f = \lambda z. (\chi \ z, b \ (\chi \ z) \ z)$
show *inj-on* $?f \ (\bigcup (F \ ' A))$
unfolding *inj-on-def*
by *clarify* (*metis* $\chi \ b \text{bij-betw-inv-into-left}$)
show $?f \ ' \bigcup (F \ ' A) \subseteq A \times B$
using $\chi \ b \text{bij-betwE}$ **by** *blast*
qed
show $A \times B \lesssim \bigcup (F \ ' A)$
unfolding *lepoll-def*
proof (*intro exI conjI*)
let $?f = \lambda(x,y). \text{inv-into } (F \ x) \ (b \ x) \ y$
have $*$: *inv-into* $(F \ x) \ (b \ x) \ y \in F \ x$ **if** $x \in A \ y \in B$ **for** $x \ y$
by (*metis* $b \text{bij-betw-imp-surj-on inv-into-into that}$)
then show *inj-on* $?f \ (A \times B)$
unfolding *inj-on-def*
by *clarsimp* (*metis* (*mono-tags, lifting*) $b \text{bij-betw-inv-into-right disj disjnt-iff pairwiseD}$)
show $?f \ ' (A \times B) \subseteq \bigcup (F \ ' A)$

by *clarsimp* (*metis* *b* *bij-betw-imp-surj-on* *inv-into-into*)
 qed
 qed

lemma *UN-lepoll-UN*:

assumes *A*: $\bigwedge x. x \in A \implies B\ x \lesssim C\ x$
 and *disj*: *pairwise* ($\lambda x\ y. \text{disjnt}\ (C\ x)\ (C\ y)$) *A*
 shows $\bigcup (B' A) \lesssim \bigcup (C' A)$
proof –
 obtain *f* **where** *f*: $\bigwedge x. x \in A \implies \text{inj-on}\ (f\ x)\ (B\ x) \wedge f\ x\ ' (B\ x) \subseteq (C\ x)$
 using *A* **unfolding** *lepoll-def* **by** *metis*
 show *?thesis*
unfolding *lepoll-def*
proof (*intro exI conjI*)
 define χ **where** $\chi \equiv \lambda z. @x. x \in A \wedge z \in B\ x$
 have $\chi\ z \in A \wedge z \in B\ (\chi\ z)$ **if** $x \in A\ z \in B\ x$ **for** $x\ z$
unfolding $\chi\text{-def}$ **by** (*metis* (*mono-tags*, *lifting*) *someI-ex that*)
 let $?f = \lambda z. (f\ (\chi\ z)\ z)$
 show *inj-on* $?f\ (\bigcup (B' A))$
 using *disj f* **unfolding** *inj-on-def* *disjnt-iff* *pairwise-def* *image-subset-iff*
by (*metis UN-iff* χ)
 show $?f\ ' \bigcup (B' A) \subseteq \bigcup (C' A)$
 using $\chi\ f$ **unfolding** *image-subset-iff* **by** *blast*
 qed
 qed

lemma *UN-epoll-UN*:

assumes *A*: $\bigwedge x. x \in A \implies B\ x \approx C\ x$
 and *B*: *pairwise* ($\lambda x\ y. \text{disjnt}\ (B\ x)\ (B\ y)$) *A*
 and *C*: *pairwise* ($\lambda x\ y. \text{disjnt}\ (C\ x)\ (C\ y)$) *A*
 shows $(\bigcup_{x \in A} B\ x) \approx (\bigcup_{x \in A} C\ x)$
proof (*rule lepoll-antisym*)
 show $\bigcup (B' A) \lesssim \bigcup (C' A)$
by (*meson A C UN-lepoll-UN eqpoll-imp-lepoll*)
 show $\bigcup (C' A) \lesssim \bigcup (B' A)$
by (*simp add: A B UN-lepoll-UN eqpoll-imp-lepoll eqpoll-sym*)
 qed

34.8 General Cartesian products (Pi)

lemma *PiE-sing-epoll-self*: $(\{a\} \rightarrow_E B) \approx B$

proof –
 have *1*: $x = y$
if $x \in \{a\} \rightarrow_E B\ y \in \{a\} \rightarrow_E B\ x\ a = y\ a$ **for** $x\ y$
by (*metis IntD2 PiE-def extensionalityI singletonD that*)
 have *2*: $x \in (\lambda h. h\ a)\ ' (\{a\} \rightarrow_E B)$ **if** $x \in B$ **for** x
 using *that* **by** (*rule-tac* $x = \lambda z \in \{a\}. x$ **in** *image-eqI*) *auto*
 show *?thesis*
unfolding *eqpoll-def* *bij-betw-def* *inj-on-def*

by (*force intro: 1 2*)
qed

lemma *lepoll-funcset-right*:

assumes $B \lesssim B'$ shows $A \rightarrow_E B \lesssim A \rightarrow_E B'$

proof –

obtain f where f : *inj-on* f B f ‘ $B \subseteq B'$

by (*meson assms lepoll-def*)

let $?G = \lambda g. \lambda z \in A. f(g\ z)$

have *inj-on* $?G$ $(A \rightarrow_E B)$

using f by (*smt (verit, best) PiE-ext PiE-mem inj-on-def restrict-apply'*)

moreover have $?G$ ‘ $(A \rightarrow_E B) \subseteq (A \rightarrow_E B')$

using f by *fastforce*

ultimately show *?thesis*

by (*meson lepoll-def*)

qed

lemma *lepoll-funcset-left*:

assumes $B \neq \{\}$ $A \lesssim A'$

shows $A \rightarrow_E B \lesssim A' \rightarrow_E B$

proof –

obtain b where $b \in B$

using *assms* by *blast*

obtain f where *inj-on* f A and *fin*: f ‘ $A \subseteq A'$

using *assms* by (*auto simp: lepoll-def*)

then obtain h where h : $\bigwedge x. x \in A \implies h\ (f\ x) = x$

using *the-inv-into-f-f* by *fastforce*

let $?F = \lambda g. \lambda u \in A'. \text{if } h\ u \in A \text{ then } g(h\ u) \text{ else } b$

show *?thesis*

unfolding *lepoll-def inj-on-def*

proof (*intro exI conjI ballI impI ext*)

fix $k\ l\ x$

assume k : $k \in A \rightarrow_E B$ and l : $l \in A \rightarrow_E B$ and $?F\ k = ?F\ l$

then have $?F\ k\ (f\ x) = ?F\ l\ (f\ x)$

by *simp*

then show $k\ x = l\ x$

by (*smt (verit, best) PiE-arb fin h image-subset-iff k l restrict-apply'*)

next

show $?F$ ‘ $(A \rightarrow_E B) \subseteq A' \rightarrow_E B$

using $\langle b \in B \rangle$ by *force*

qed

qed

lemma *lepoll-funcset*:

$\llbracket B \neq \{\}; A \lesssim A'; B \lesssim B' \rrbracket \implies A \rightarrow_E B \lesssim A' \rightarrow_E B'$

by (*rule lepoll-trans [OF lepoll-funcset-right lepoll-funcset-left]*) *auto*

lemma *lepoll-PiE*:

assumes $\bigwedge i. i \in A \implies B\ i \lesssim C\ i$

shows $PiE\ A\ B \lesssim PiE\ A\ C$
proof –
 obtain f where $f: \bigwedge i. i \in A \implies inj\text{-}on\ (f\ i)\ (B\ i) \wedge (f\ i) \text{ ‘ } B\ i \subseteq C\ i$
 using *assms unfolding lepoll-def by metis*
 let $?G = \lambda g. \lambda i \in A. f\ i\ (g\ i)$
 have $inj\text{-}on\ ?G\ (PiE\ A\ B)$
 by (*smt (verit, ccfv-SIG) PiE-ext PiE-iff f inj-on-def restrict-apply*)
 moreover have $?G \text{ ‘ } (PiE\ A\ B) \subseteq (PiE\ A\ C)$
 using f by *fastforce*
 ultimately show *?thesis*
 by (*meson lepoll-def*)
qed

lemma *card-le-PiE-subindex*:
 assumes $A \subseteq A'\ Pi_E\ A'\ B \neq \{\}$
 shows $PiE\ A\ B \lesssim PiE\ A'\ B$
proof –
 have $\bigwedge x. x \in A' \implies \exists y. y \in B\ x$
 using *assms by blast*
 then obtain g where $g: \bigwedge x. x \in A' \implies g\ x \in B\ x$
 by *metis*
 let $?F = \lambda f\ x. \text{if } x \in A \text{ then } f\ x \text{ else if } x \in A' \text{ then } g\ x \text{ else undefined}$
 have $Pi_E\ A\ B \subseteq (\lambda f. \text{restrict } f\ A) \text{ ‘ } Pi_E\ A'\ B$
proof
 show $f \in Pi_E\ A\ B \implies f \in (\lambda f. \text{restrict } f\ A) \text{ ‘ } Pi_E\ A'\ B$ **for** f
 using $\langle A \subseteq A' \rangle$
 by (*rule-tac x=?F f in image-eqI (auto simp: g fun-eq-iff)*)
qed
 then have $Pi_E\ A\ B \lesssim (\lambda f. \lambda i \in A. f\ i) \text{ ‘ } Pi_E\ A'\ B$
 by (*simp add: subset-imp-lepoll*)
 also have $\dots \lesssim Pi_E\ A'\ B$
 by (*rule image-lepoll*)
 finally show *?thesis* .
qed

lemma *finite-restricted-funspace*:
 assumes *finite A finite B*
 shows *finite {f. f ‘ A ⊆ B ∧ {x. f x ≠ k x} ⊆ A} (is finite ?F)*
proof (*rule finite-subset*)
 show *finite ((λU x. if ∃ y. (x,y) ∈ U then @y. (x,y) ∈ U else k x) ‘ Pow(A × B)) (is finite ?G)*
 using *assms by auto*
 show $?F \subseteq ?G$
proof
 fix f
 assume $f \in ?F$
 then show $f \in ?G$

```

    by (rule-tac x=(λx. (x,f x)) ‘{x. f x ≠ k x} in image-eqI) (auto simp: fun-eq-iff
image-def)
  qed
qed

```

proposition *finite-PiE-iff*:

$finite(PiE\ I\ S) \longleftrightarrow PiE\ I\ S = \{\} \vee finite\ \{i \in I. \sim(\exists a. S\ i \subseteq \{a\})\} \wedge (\forall i \in I. finite(S\ i))$

(is ?lhs = ?rhs)

proof (cases $PiE\ I\ S = \{\}$)

case *False*

define J where $J \equiv \{i \in I. \nexists a. S\ i \subseteq \{a\}\}$

show ?thesis

proof

assume L : ?lhs

have *infinite* ($PiE\ I\ S$) if *infinite* J

proof –

have $(UNIV::nat\ set) \lesssim (UNIV::(nat \Rightarrow bool)\ set)$

proof –

have $\forall N::nat\ set. inj-on\ (=)\ N$

by (simp add: inj-on-def)

then show ?thesis

by (meson infinite-iff-countable-subset infinite-le-lepoll top.extremum)

qed

also have $\dots = (UNIV::nat\ set) \rightarrow_E (UNIV::bool\ set)$

by *auto*

also have $\dots \lesssim J \rightarrow_E (UNIV::bool\ set)$

by (metis empty-not-UNIV infinite-le-lepoll lepoll-funcset-left that)

also have $\dots \lesssim PiE\ J\ S$

proof –

have *: $(UNIV::bool\ set) \lesssim S\ i$ if $i \in I$ and §: $\forall a. \neg S\ i \subseteq \{a\}$ for i

proof –

obtain $a\ b$ where $\{a,b\} \subseteq S\ i$ and $a \neq b$

by (metis § empty-subsetI insert-subset subsetI)

then show ?thesis

by (metis Set.set-insert UNIV-bool insert-iff insert-lepoll-cong insert-subset singleton-lepoll)

qed

show ?thesis

by (auto simp: * J-def intro: lepoll-PiE)

qed

also have $\dots \lesssim PiE\ I\ S$

using *False* by (auto simp: J-def intro: card-le-PiE-subindex)

finally have $(UNIV::nat\ set) \lesssim PiE\ I\ S$.

then show ?thesis

by (simp add: infinite-le-lepoll)

qed

moreover have *finite* ($S\ i$) if $i \in I$ for i


```

proof (rule finite-subset)
  obtain  $f$  where  $f: f \in PiE\ I\ S$ 
    using False by blast
  show  $S\ i \subseteq (\lambda f. f\ i) \text{ ‘ } PiE\ I\ S$ 
  proof
    show  $s \in (\lambda f. f\ i) \text{ ‘ } PiE\ I\ S$  if  $s \in S\ i$  for  $s$ 
      using that  $f\ \langle i \in I \rangle$ 
      by (rule-tac  $x=\lambda j. \text{ if } j = i \text{ then } s \text{ else } f\ j$  in image-eqI) auto
    qed
  next
    show finite  $((\lambda x. x\ i) \text{ ‘ } PiE\ I\ S)$ 
      using L by blast
    qed
  ultimately show ?rhs
    using L
    by (auto simp: J-def False)
  next
    assume  $R: ?rhs$ 
    have  $\forall i \in I - J. \exists a. S\ i = \{a\}$ 
      using False J-def by blast
    then obtain  $a$  where  $a: \forall i \in I - J. S\ i = \{a\ i\}$ 
      by metis
    let  $?F = \{f. f \text{ ‘ } J \subseteq (\bigcup i \in J. S\ i) \wedge \{i. f\ i \neq (\text{if } i \in I \text{ then } a\ i \text{ else undefined})\}$ 
     $\subseteq J\}$ 
    have  $*$ : finite  $(PiE\ I\ S)$ 
      if finite  $J$  and  $\forall i \in I. \text{finite } (S\ i)$ 
    proof (rule finite-subset)
      have  $\bigwedge f j. \llbracket f \in PiE\ I\ S; j \in J \rrbracket \implies f\ j \in \bigcup (S \text{ ‘ } J)$ 
        using J-def by blast
      moreover
        have  $\bigwedge f j. \llbracket f \in PiE\ I\ S; f\ j \neq (\text{if } j \in I \text{ then } a\ j \text{ else undefined}) \rrbracket \implies j \in J$ 
          by (metis DiffI PiE-E a singletonD)
        ultimately show  $PiE\ I\ S \subseteq ?F$  by force
      show finite  $?F$ 
      proof (rule finite-restricted-funspace [OF  $\langle \text{finite } J \rangle$ ])
        show finite  $(\bigcup (S \text{ ‘ } J))$ 
        using that J-def by blast
      qed
    qed
  show ?lhs
    using  $R$  by (auto simp:  $*$  J-def)
  qed
qed auto

```

corollary *finite-funcset-iff*:

$\text{finite}(I \rightarrow_E S) \longleftrightarrow (\exists a. S \subseteq \{a\}) \vee I = \{\} \vee \text{finite } I \wedge \text{finite } S$
by (*fastforce simp: finite-PiE-iff PiE-eq-empty-iff dest: subset-singletonD*)

34.9 Misc other resultd

lemma *lists-lepoll-mono*:

assumes $A \lesssim B$ shows *lists* $A \lesssim$ *lists* B

proof –

obtain f where f : *inj-on* f $A \rightarrow B$

by (*meson* *assms* *lepoll-def*)

moreover have *inj-on* (*map* f) (*lists* A)

using f **unfolding** *inj-on-def*

by *clarsimp* (*metis* *list.inj-map-strong*)

ultimately show *?thesis*

unfolding *lepoll-def* **by** *force*

qed

lemma *lepoll-lists*: $A \lesssim$ *lists* A

unfolding *lepoll-def* *inj-on-def* **by** (*rule-tac* $x = \lambda x. [x]$ **in** *exI*) *auto*

Dedekind’s definition of infinite set

lemma *infinite-iff-psubset*: *infinite* $A \longleftrightarrow (\exists B. B \subset A \wedge A \approx B)$

proof

assume *infinite* A

then obtain $f :: \text{nat} \Rightarrow 'a$ where *inj* f and f : *range* $f \subseteq A$

by (*meson* *infinite-countable-subset*)

define C where $C \equiv A - \text{range } f$

have C : $A = \text{range } f \cup C$ *range* $f \cap C = \{\}$

using f **by** (*auto* *simp*: C -def)

have $*$: *range* $(f \circ \text{Suc}) \subset \text{range } f$

using *inj-eq* [*OF* $\langle \text{inj } f \rangle$] **by** (*fastforce* *simp*: *set-eq-iff*)

have *range* $f \cup C \approx \text{range } (f \circ \text{Suc}) \cup C$

proof (*intro* *Un-eqpoll-cong*)

show *range* $f \approx \text{range } (f \circ \text{Suc})$

by (*meson* $\langle \text{inj } f \rangle$ *eqpoll-refl* *inj-Suc* *inj-compose* *inj-on-image-eqpoll-2*)

show *disjnt* (*range* f) C

by (*simp* *add*: C *disjnt-def*)

then show *disjnt* (*range* $(f \circ \text{Suc})$) C

using $*$ *disjnt-subset1* **by** *blast*

qed *auto*

moreover have *range* $(f \circ \text{Suc}) \cup C \subset A$

using $*$ f C -def **by** *blast*

ultimately show $\exists B \subset A. A \approx B$

by (*metis* $C(1)$)

next

assume $\exists B \subset A. A \approx B$ then show *infinite* A

by (*metis* *card-subset-eq* *eqpoll-finite-iff* *eqpoll-iff-card* *psubsetE*)

qed

lemma *infinite-iff-psubset-le*: *infinite* $A \longleftrightarrow (\exists B. B \subset A \wedge A \lesssim B)$

by (*meson* *eqpoll-imp-lepoll* *infinite-iff-psubset* *lepoll-antisym* *psubsetE* *subset-imp-lepoll*)

end

```

theory Simps-Case-Conv
imports Case-Converter
  keywords simps-of-case case-of-simps :: thy-decl
  abbrevs simps-of-case case-of-simps =
begin

ML-file  $\langle \textit{simps-case-conv.ML} \rangle$ 

end

theory Extended
  imports Simps-Case-Conv
begin

datatype  $'a \textit{ extended} = \textit{Fin } 'a \mid \textit{Pinf } (\langle \infty \rangle) \mid \textit{Minf } (\langle -\infty \rangle)$ 

instantiation extended ::  $(\textit{order})\textit{order}$ 
begin

fun less-eq-extended ::  $'a \textit{ extended} \Rightarrow 'a \textit{ extended} \Rightarrow \textit{bool}$  where
 $\textit{Fin } x \leq \textit{Fin } y = (x \leq y) \mid$ 
 $- \leq \textit{Pinf} = \textit{True} \mid$ 
 $\textit{Minf} \leq - = \textit{True} \mid$ 
 $(-\textit{:}'a \textit{ extended}) \leq - = \textit{False}$ 

case-of-simps less-eq-extended-case: less-eq-extended.simps

definition less-extended ::  $'a \textit{ extended} \Rightarrow 'a \textit{ extended} \Rightarrow \textit{bool}$  where
 $((x\textit{:}'a \textit{ extended}) < y) = (x \leq y \wedge \neg y \leq x)$ 

instance
  by intro-classes (auto simp: less-extended-def less-eq-extended-case split: extended.splits)

end

instance extended ::  $(\textit{linorder})\textit{linorder}$ 
  by intro-classes (auto simp: less-eq-extended-case split: extended.splits)

lemma Minf-le[simp]:  $\textit{Minf} \leq y$ 
by(cases y) auto
lemma le-Pinf[simp]:  $x \leq \textit{Pinf}$ 
by(cases x) auto
lemma le-Minf[simp]:  $x \leq \textit{Minf} \longleftrightarrow x = \textit{Minf}$ 
by(cases x) auto
lemma Pinf-le[simp]:  $\textit{Pinf} \leq x \longleftrightarrow x = \textit{Pinf}$ 

```

by(*cases x*) *auto*

lemma *less-extended-simps[simp]*:
 $Fin\ x < Fin\ y = (x < y)$
 $Fin\ x < Pinf = True$
 $Fin\ x < Minf = False$
 $Pinf < h = False$
 $Minf < Fin\ x = True$
 $Minf < Pinf = True$
 $l < Minf = False$
by (*auto simp add: less-extended-def*)

lemma *min-extended-simps[simp]*:
 $min\ (Fin\ x)\ (Fin\ y) = Fin(min\ x\ y)$
 $min\ xx\ Pinf = xx$
 $min\ xx\ Minf = Minf$
 $min\ Pinf\ yy = yy$
 $min\ Minf\ yy = Minf$
by (*auto simp add: min-def*)

lemma *max-extended-simps[simp]*:
 $max\ (Fin\ x)\ (Fin\ y) = Fin(max\ x\ y)$
 $max\ xx\ Pinf = Pinf$
 $max\ xx\ Minf = xx$
 $max\ Pinf\ yy = Pinf$
 $max\ Minf\ yy = yy$
by (*auto simp add: max-def*)

instantiation *extended* :: (*zero*)*zero*
begin
definition $0 = Fin(0::'a)$
instance ..
end

declare *zero-extended-def[symmetric, code-post]*

instantiation *extended* :: (*one*)*one*
begin
definition $1 = Fin(1::'a)$
instance ..
end

declare *one-extended-def[symmetric, code-post]*

instantiation *extended* :: (*plus*)*plus*
begin

The following definition of addition is totalized to make it asociative and commutative. Normally the sum of plus and minus infinity is undefined.

```

fun plus-extended where
  Fin  $x + \textit{Fin } y = \textit{Fin}(x+y)$  |
  Fin  $x + \textit{Pinf} = \textit{Pinf}$  |
  Pinf  $+ \textit{Fin } x = \textit{Pinf}$  |
  Pinf  $+ \textit{Pinf} = \textit{Pinf}$  |
  Minf  $+ \textit{Fin } y = \textit{Minf}$  |
  Fin  $x + \textit{Minf} = \textit{Minf}$  |
  Minf  $+ \textit{Minf} = \textit{Minf}$  |
  Minf  $+ \textit{Pinf} = \textit{Pinf}$  |
  Pinf  $+ \textit{Minf} = \textit{Pinf}$ 

case-of-simps plus-case: plus-extended.simps

instance ..

end

instance extended :: (ab-semigroup-add)ab-semigroup-add
  by intro-classes (simp-all add: ac-simps plus-case split: extended.splits)

instance extended :: (ordered-ab-semigroup-add)ordered-ab-semigroup-add
  by intro-classes (auto simp: add-left-mono plus-case split: extended.splits)

instance extended :: (comm-monoid-add)comm-monoid-add
proof
  fix  $x :: 'a$  extended show  $0 + x = x$  unfolding zero-extended-def by(cases
x)auto
qed

instantiation extended :: (uminus)uminus
begin

fun uminus-extended where
   $-(\textit{Fin } x) = \textit{Fin } (-x)$  |
   $-\textit{Pinf} = \textit{Minf}$  |
   $-\textit{Minf} = \textit{Pinf}$ 

instance ..

end

instantiation extended :: (ab-group-add)minus
begin
definition  $x - y = x + -(y::'a \textit{ extended})$ 
instance ..
end

```

lemma *minus-extended-simps*[*simp*]:

$Fin\ x - Fin\ y = Fin(x - y)$
 $Fin\ x - Pinf = Minf$
 $Fin\ x - Minf = Pinf$
 $Pinf - Fin\ y = Pinf$
 $Pinf - Minf = Pinf$
 $Minf - Fin\ y = Minf$
 $Minf - Pinf = Minf$
 $Minf - Minf = Pinf$
 $Pinf - Pinf = Pinf$

by (*simp-all add: minus-extended-def*)

Numerals:

instance *extended* :: ($\{ab\text{-semigroup-add,one}\}$)*numeral* ..

lemma *Fin-numeral*[*code-post*]: $Fin(\text{numeral } w) = \text{numeral } w$

apply (*induct w rule: num-induct*)

apply (*simp only: numeral-One one-extended-def*)

apply (*simp only: numeral-inc one-extended-def plus-extended.simps(1)[symmetric]*)

done

lemma *Fin-neg-numeral*[*code-post*]: $Fin(-\text{numeral } w) = -\text{numeral } w$

by (*simp only: Fin-numeral uminus-extended.simps[symmetric]*)

instantiation *extended* :: (*lattice*)*bounded-lattice*

begin

definition *bot* = *Minf*

definition *top* = *Pinf*

fun *inf-extended* :: 'a *extended* \Rightarrow 'a *extended* \Rightarrow 'a *extended* **where**

inf-extended (*Fin i*) (*Fin j*) = *Fin (inf i j)* |

inf-extended *a Minf* = *Minf* |

inf-extended *Minf a* = *Minf* |

inf-extended *Pinf a* = *a* |

inf-extended *a Pinf* = *a*

fun *sup-extended* :: 'a *extended* \Rightarrow 'a *extended* \Rightarrow 'a *extended* **where**

sup-extended (*Fin i*) (*Fin j*) = *Fin (sup i j)* |

sup-extended *a Pinf* = *Pinf* |

sup-extended *Pinf a* = *Pinf* |

sup-extended *Minf a* = *a* |

sup-extended *a Minf* = *a*

case-of-simps *inf-extended-case*: *inf-extended.simps*

case-of-simps *sup-extended-case*: *sup-extended.simps*

```

instance
  by (intro-classes) (auto simp: inf-extended-case sup-extended-case less-eq-extended-case
    bot-extended-def top-extended-def split: extended.splits)
end

end

```

35 Continuity and iterations

```

theory Order-Continuity
imports Complex-Main Countable-Complete-Lattices
begin

```

```

lemma SUP-nat-binary:
  (sup A (SUP x ∈ Collect ((<) (0::nat)). B)) = (sup A B::'a::countable-complete-lattice)
  apply (subst image-constant)
  apply auto
done

```

```

lemma INF-nat-binary:
  (inf A (INF x ∈ Collect ((<) (0::nat)). B)) = (inf A B::'a::countable-complete-lattice)
  apply (subst image-constant)
  apply auto
done

```

The name *continuous* is already taken in *Complex-Main*, so we use *sup-continuous* and *inf-continuous*. These names appear sometimes in literature and have the advantage that these names are duals.

```

named-theorems order-continuous-intros

```

35.1 Continuity for complete lattices

```

definition

```

```

  sup-continuous :: ('a::countable-complete-lattice ⇒ 'b::countable-complete-lattice)
  ⇒ bool

```

```

where

```

```

  sup-continuous F ⟷ (∀ M::nat ⇒ 'a. mono M ⟶ F (SUP i. M i) = (SUP i.
  F (M i)))

```

```

lemma sup-continuousD: sup-continuous F ⟹ mono M ⟹ F (SUP i::nat. M
i) = (SUP i. F (M i))

```

```

  by (auto simp: sup-continuous-def)

```

```

lemma sup-continuous-mono:

```

```

  mono F if sup-continuous F

```

```

proof

```

```

  fix A B :: 'a

```

```

assume  $A \leq B$ 
let  $?f = \lambda n :: \text{nat. if } n = 0 \text{ then } A \text{ else } B$ 
from  $\langle A \leq B \rangle$  have  $\text{incseq } ?f$ 
  by (auto intro: monoI)
with  $\langle \text{sup-continuous } F \rangle$  have  $*$ :  $F (\text{SUP } i. ?f i) = (\text{SUP } i. F (?f i))$ 
  by (auto dest: sup-continuousD)
from  $\langle A \leq B \rangle$  have  $B = \text{sup } A B$ 
  by (simp add: le-iff-sup)
then have  $F B = F (\text{sup } A B)$ 
  by simp
also have  $\dots = \text{sup } (F A) (F B)$ 
  using  $*$  by (simp add: if-distrib SUP-nat-binary cong del: SUP-cong)
finally show  $F A \leq F B$ 
  by (simp add: le-iff-sup)
qed

```

```

lemma [order-continuous-intros]:
  shows sup-continuous-const:  $\text{sup-continuous } (\lambda x. c)$ 
  and sup-continuous-id:  $\text{sup-continuous } (\lambda x. x)$ 
  and sup-continuous-apply:  $\text{sup-continuous } (\lambda f. f x)$ 
  and sup-continuous-fun:  $(\bigwedge s. \text{sup-continuous } (\lambda x. P x s)) \implies \text{sup-continuous } P$ 
  and sup-continuous-If:  $\text{sup-continuous } F \implies \text{sup-continuous } G \implies \text{sup-continuous } (\lambda f. \text{if } C \text{ then } F f \text{ else } G f)$ 
  by (auto simp: sup-continuous-def image-comp)

```

```

lemma sup-continuous-compose:
  assumes  $f$ : sup-continuous  $f$  and  $g$ : sup-continuous  $g$ 
  shows sup-continuous  $(\lambda x. f (g x))$ 
  unfolding sup-continuous-def
proof safe
  fix  $M :: \text{nat} \Rightarrow 'c$ 
  assume  $M$ : mono  $M$ 
  then have mono  $(\lambda i. g (M i))$ 
    using sup-continuous-mono[OF  $g$ ] by (auto simp: mono-def)
  with  $M$  show  $f (g (\text{Sup } (M \text{ ' UNIV}))) = (\text{SUP } i. f (g (M i)))$ 
    by (auto simp: sup-continuous-def g[THEN sup-continuousD] f[THEN sup-continuousD])
qed

```

```

lemma sup-continuous-sup[order-continuous-intros]:
  sup-continuous  $f \implies \text{sup-continuous } g \implies \text{sup-continuous } (\lambda x. \text{sup } (f x) (g x))$ 
  by (simp add: sup-continuous-def ccSUP-sup-distrib)

```

```

lemma sup-continuous-inf[order-continuous-intros]:
  fixes  $P Q :: 'a :: \text{countable-complete-lattice} \Rightarrow 'b :: \text{countable-complete-distrib-lattice}$ 
  assumes  $P$ : sup-continuous  $P$  and  $Q$ : sup-continuous  $Q$ 
  shows sup-continuous  $(\lambda x. \text{inf } (P x) (Q x))$ 
  unfolding sup-continuous-def
proof (safe intro!: antisym)

```



```

fix  $M :: \text{nat} \Rightarrow 'a$  assume  $M: \text{incseq } M$ 
have  $\inf (P (SUP\ i.\ M\ i)) (Q (SUP\ i.\ M\ i)) \leq (SUP\ j.\ \inf (P (M\ i)) (Q (M\ j)))$ 
by (simp add: sup-continuousD[OF P M] sup-continuousD[OF Q M] inf-ccSUP ccSUP-inf)
also have  $\dots \leq (SUP\ i.\ \inf (P (M\ i)) (Q (M\ i)))$ 
proof (intro ccSUP-least)
fix  $i\ j$  from  $M$  assms[THEN sup-continuous-mono] show  $\inf (P (M\ i)) (Q (M\ j)) \leq (SUP\ i.\ \inf (P (M\ i)) (Q (M\ i)))$ 
by (intro ccSUP-upper2[of - sup i j] inf-mono) (auto simp: mono-def)
qed auto
finally show  $\inf (P (SUP\ i.\ M\ i)) (Q (SUP\ i.\ M\ i)) \leq (SUP\ i.\ \inf (P (M\ i)) (Q (M\ i)))$  .

```

```

show  $(SUP\ i.\ \inf (P (M\ i)) (Q (M\ i))) \leq \inf (P (SUP\ i.\ M\ i)) (Q (SUP\ i.\ M\ i))$ 
unfolding sup-continuousD[OF P M] sup-continuousD[OF Q M] by (intro ccSUP-least inf-mono ccSUP-upper) auto
qed

```

lemma *sup-continuous-and[order-continuous-intros]:*
 $\text{sup-continuous } P \implies \text{sup-continuous } Q \implies \text{sup-continuous } (\lambda x. P\ x \wedge Q\ x)$
using *sup-continuous-inf[of P Q]* **by** *simp*

lemma *sup-continuous-or[order-continuous-intros]:*
 $\text{sup-continuous } P \implies \text{sup-continuous } Q \implies \text{sup-continuous } (\lambda x. P\ x \vee Q\ x)$
by (*auto simp: sup-continuous-def*)

lemma *sup-continuous-lfp:*
assumes *sup-continuous F* **shows** $\text{lfp } F = (SUP\ i.\ (F \rightsquigarrow i)\ \text{bot})$ (*is* $\text{lfp } F = ?U$)
proof (*rule antisym*)
note $\text{mono} = \text{sup-continuous-mono}[OF \langle \text{sup-continuous } F \rangle]$
show $?U \leq \text{lfp } F$
proof (*rule SUP-least*)
fix i **show** $(F \rightsquigarrow i)\ \text{bot} \leq \text{lfp } F$
proof (*induct i*)
case (*Suc i*)
have $(F \rightsquigarrow \text{Suc } i)\ \text{bot} = F ((F \rightsquigarrow i)\ \text{bot})$ **by** *simp*
also have $\dots \leq F (\text{lfp } F)$ **by** (*rule monoD[OF mono Suc]*)
also have $\dots = \text{lfp } F$ **by** (*simp add: lfp-fixpoint[OF mono]*)
finally show *?case* .
qed *simp*
qed
show $\text{lfp } F \leq ?U$
proof (*rule lfp-lowerbound*)
have $\text{mono } (\lambda i::\text{nat}.\ (F \rightsquigarrow i)\ \text{bot})$
proof –
have $(F \rightsquigarrow i)\ \text{bot} \leq (F \rightsquigarrow (\text{Suc } i))\ \text{bot}$ **for** $i::\text{nat}$
proof (*induct i*)

```

      case 0 show ?case by simp
    next
      case Suc thus ?case using monoD[OF mono Suc] by auto
    qed
    thus ?thesis by (auto simp add: mono-iff-le-Suc)
  qed
  hence  $F \text{ ?}U = (SUP\ i. (F \rightsquigarrow Suc\ i)\ bot)$ 
    using ‹sup-continuous F› by (simp add: sup-continuous-def)
  also have  $\dots \leq \text{?}U$ 
    by (fast intro: SUP-least SUP-upper)
  finally show  $F \text{ ?}U \leq \text{?}U$  .
qed
qed

lemma lfp-transfer-bounded:
  assumes  $P: P\ bot \wedge x. P\ x \implies P\ (f\ x) \wedge M. (\wedge i. P\ (M\ i)) \implies P\ (SUP\ i::nat. M\ i)$ 
  assumes  $\alpha: \wedge M. mono\ M \implies (\wedge i::nat. P\ (M\ i)) \implies \alpha\ (SUP\ i. M\ i) = (SUP\ i. \alpha\ (M\ i))$ 
  assumes  $f: sup\text{-}continuous\ f$  and  $g: sup\text{-}continuous\ g$ 
  assumes [simp]:  $\wedge x. P\ x \implies x \leq lfp\ f \implies \alpha\ (f\ x) = g\ (\alpha\ x)$ 
  assumes  $g\text{-}bound: \wedge x. \alpha\ bot \leq g\ x$ 
  shows  $\alpha\ (lfp\ f) = lfp\ g$ 
proof (rule antisym)
  note mono-g = sup-continuous-mono[OF g]
  note mono-f = sup-continuous-mono[OF f]
  have lfp-bound:  $\alpha\ bot \leq lfp\ g$ 
    by (subst lfp-unfold[OF mono-g]) (rule g-bound)

  have P-pow:  $P\ ((f \rightsquigarrow i)\ bot)$  for  $i$ 
    by (induction i) (auto intro!: P)
  have incseq-pow:  $mono\ (\lambda i. (f \rightsquigarrow i)\ bot)$ 
    unfolding mono-iff-le-Suc
  proof
    fix i show  $(f \rightsquigarrow i)\ bot \leq (f \rightsquigarrow (Suc\ i))\ bot$ 
    proof (induct i)
      case Suc thus ?case using monoD[OF sup-continuous-mono[OF f] Suc] by
        auto
    qed (simp add: le-fun-def)
  qed
  have P-lfp:  $P\ (lfp\ f)$ 
    using P-pow unfolding sup-continuous-lfp[OF f] by (auto intro!: P)

  have iter-le-lfp:  $(f \rightsquigarrow n)\ bot \leq lfp\ f$  for  $n$ 
    apply (induction n)
    apply simp
    apply (subst lfp-unfold[OF mono-f])
    apply (auto intro!: monoD[OF mono-f])
  done

```

```

have  $\alpha$  (lfp f) = (SUP i.  $\alpha$  ((f~i) bot))
  unfolding sup-continuous-lfp[OF f] using incseq-pow P-pow by (rule  $\alpha$ )
also have ...  $\leq$  lfp g
proof (rule SUP-least)
  fix i show  $\alpha$  ((f~i) bot)  $\leq$  lfp g
  proof (induction i)
    case (Suc n) then show ?case
      by (subst lfp-unfold[OF mono-g]) (simp add: monoD[OF mono-g] P-pow
iter-le-lfp)
  qed (simp add: lfp-bound)
qed
finally show  $\alpha$  (lfp f)  $\leq$  lfp g .

```

```

show lfp g  $\leq$   $\alpha$  (lfp f)
proof (induction rule: lfp-ordinal-induct[OF mono-g])
  case (1 S) then show ?case
    by (subst lfp-unfold[OF sup-continuous-mono[OF f]])
      (simp add: monoD[OF mono-g] P-lfp)
  qed (auto intro: Sup-least)
qed

```

lemma lfp-transfer:

```

sup-continuous  $\alpha \implies$  sup-continuous f  $\implies$  sup-continuous g  $\implies$ 
  ( $\bigwedge x. \alpha$  bot  $\leq$  g x)  $\implies$  ( $\bigwedge x. x \leq$  lfp f  $\implies \alpha$  (f x) = g ( $\alpha$  x))  $\implies \alpha$  (lfp f) =
lfp g
  by (rule lfp-transfer-bounded[where P=top]) (auto dest: sup-continuousD)

```

definition

```

inf-continuous :: ('a::countable-complete-lattice  $\Rightarrow$  'b::countable-complete-lattice)
 $\Rightarrow$  bool

```

where

```

inf-continuous F  $\longleftrightarrow$  ( $\forall M::nat \Rightarrow 'a. \text{antimono } M \longrightarrow F$  (INF i. M i) = (INF
i. F (M i)))

```

lemma inf-continuousD: inf-continuous F \implies antimono M \implies F (INF i::nat. M i) = (INF i. F (M i))

by (auto simp: inf-continuous-def)

lemma inf-continuous-mono:

mono F **if** inf-continuous F

proof

fix A B :: 'a

assume A \leq B

let ?f = $\lambda n::nat. \text{if } n = 0 \text{ then } B \text{ else } A$

from $\langle A \leq B \rangle$ have decseq ?f

by (auto intro: antimonoI)

with $\langle \text{inf-continuous } F \rangle$ have *: F (INF i. ?f i) = (INF i. F (?f i))

by (auto dest: inf-continuousD)

```

from  $\langle A \leq B \rangle$  have  $A = \inf B A$ 
  by (simp add: inf.absorb-iff2)
then have  $F A = F (\inf B A)$ 
  by simp
also have  $\dots = \inf (F B) (F A)$ 
  using * by (simp add: if-distrib INF-nat-binary cong del: INF-cong)
finally show  $F A \leq F B$ 
  by (simp add: inf.absorb-iff2)
qed

```

```

lemma [order-continuous-intros]:
  shows inf-continuous-const: inf-continuous  $(\lambda x. c)$ 
    and inf-continuous-id: inf-continuous  $(\lambda x. x)$ 
    and inf-continuous-apply: inf-continuous  $(\lambda f. f x)$ 
    and inf-continuous-fun:  $(\bigwedge s. \text{inf-continuous } (\lambda x. P x s)) \implies \text{inf-continuous } P$ 
    and inf-continuous-If: inf-continuous  $F \implies \text{inf-continuous } G \implies \text{inf-continuous}$ 
       $(\lambda f. \text{if } C \text{ then } F f \text{ else } G f)$ 
    by (auto simp: inf-continuous-def image-comp)

```

```

lemma inf-continuous-inf[order-continuous-intros]:
  inf-continuous  $f \implies \text{inf-continuous } g \implies \text{inf-continuous } (\lambda x. \inf (f x) (g x))$ 
  by (simp add: inf-continuous-def ccINF-inf-distrib)

```

```

lemma inf-continuous-sup[order-continuous-intros]:
  fixes  $P Q :: 'a :: \text{countable-complete-lattice} \Rightarrow 'b :: \text{countable-complete-distrib-lattice}$ 
  assumes  $P$ : inf-continuous  $P$  and  $Q$ : inf-continuous  $Q$ 
  shows inf-continuous  $(\lambda x. \sup (P x) (Q x))$ 
  unfolding inf-continuous-def
proof (safe intro!: antisym)
  fix  $M :: \text{nat} \Rightarrow 'a$  assume  $M$ : decseq  $M$ 
  show  $\sup (P (INF i. M i)) (Q (INF i. M i)) \leq (INF i. \sup (P (M i)) (Q (M i)))$ 
  unfolding inf-continuousD[OF P M] inf-continuousD[OF Q M] by (intro
    ccINF-greatest sup-mono ccINF-lower) auto

```

```

  have  $(INF i. \sup (P (M i)) (Q (M i))) \leq (INF j i. \sup (P (M i)) (Q (M j)))$ 
  proof (intro ccINF-greatest)
    fix  $i j$  from  $M$  assms[THEN inf-continuous-mono] show  $\sup (P (M i)) (Q (M j)) \geq (INF i. \sup (P (M i)) (Q (M i)))$ 
    by (intro ccINF-lower2[of - sup i j] sup-mono) (auto simp: mono-def anti-mono-def)
  qed auto
  also have  $\dots \leq \sup (P (INF i. M i)) (Q (INF i. M i))$ 
  by (simp add: inf-continuousD[OF P M] inf-continuousD[OF Q M] ccINF-sup sup-ccINF)
  finally show  $\sup (P (INF i. M i)) (Q (INF i. M i)) \geq (INF i. \sup (P (M i)) (Q (M i)))$ 
  qed

```

lemma *inf-continuous-and*[*order-continuous-intros*]:

inf-continuous $P \implies \text{inf-continuous } Q \implies \text{inf-continuous } (\lambda x. P\ x \wedge Q\ x)$

using *inf-continuous-inf*[*of P Q*] **by** *simp*

lemma *inf-continuous-or*[*order-continuous-intros*]:

inf-continuous $P \implies \text{inf-continuous } Q \implies \text{inf-continuous } (\lambda x. P\ x \vee Q\ x)$

using *inf-continuous-sup*[*of P Q*] **by** *simp*

lemma *inf-continuous-compose*:

assumes f : *inf-continuous* f **and** g : *inf-continuous* g

shows *inf-continuous* $(\lambda x. f\ (g\ x))$

unfolding *inf-continuous-def*

proof *safe*

fix $M :: \text{nat} \Rightarrow 'c$

assume M : *antimono* M

then have *antimono* $(\lambda i. g\ (M\ i))$

using *inf-continuous-mono*[*OF g*] **by** (*auto simp: mono-def antimono-def*)

with M **show** $f\ (g\ (\text{Inf}\ (M\ ' \text{UNIV}))) = (\text{INF}\ i. f\ (g\ (M\ i)))$

by (*auto simp: inf-continuous-def g[THEN inf-continuousD] f[THEN inf-continuousD]*)

qed

lemma *inf-continuous-gfp*:

assumes *inf-continuous* F **shows** $\text{gfp } F = (\text{INF}\ i. (F\ \frown\ i)\ \text{top})$ (**is** $\text{gfp } F = ?U$)

proof (*rule antisym*)

note $\text{mono} = \text{inf-continuous-mono}$ [*OF* $\langle \text{inf-continuous } F \rangle$]

show $\text{gfp } F \leq ?U$

proof (*rule INF-greatest*)

fix i **show** $\text{gfp } F \leq (F\ \frown\ i)\ \text{top}$

proof (*induct i*)

case (*Suc i*)

have $\text{gfp } F = F\ (\text{gfp } F)$ **by** (*simp add: gfp-fixpoint*[*OF mono*])

also have $\dots \leq F\ ((F\ \frown\ i)\ \text{top})$ **by** (*rule monoD*[*OF mono Suc*])

also have $\dots = (F\ \frown\ \text{Suc } i)\ \text{top}$ **by** *simp*

finally show $?case$.

qed *simp*

qed

show $?U \leq \text{gfp } F$

proof (*rule gfp-upperbound*)

have $*$: *antimono* $(\lambda i::\text{nat}. (F\ \frown\ i)\ \text{top})$

proof –

have $(F\ \frown\ \text{Suc } i)\ \text{top} \leq (F\ \frown\ i)\ \text{top}$ **for** $i::\text{nat}$

proof (*induct i*)

case 0

show $?case$ **by** *simp*

next

case *Suc*

thus $?case$ **using** *monoD*[*OF mono Suc*] **by** *auto*

qed

thus $?thesis$ **by** (*auto simp add: antimono-iff-le-Suc*)

```

qed
have ?U ≤ (INF i. (F ~ Suc i) top)
  by (fast intro: INF-greatest INF-lower)
also have ... ≤ F ?U
  by (simp add: inf-continuousD ⟨inf-continuous F⟩ *)
finally show ?U ≤ F ?U .
qed
qed

```

lemma *gfp-transfer*:

```

assumes α: inf-continuous α and f: inf-continuous f and g: inf-continuous g
assumes [simp]: α top = top ∧ x. α (f x) = g (α x)
shows α (gfp f) = GFP g
proof -
  have α (gfp f) = (INF i. α ((f ~ i) top))
  unfolding inf-continuous-gfp[OF f] by (intro f α inf-continuousD antimono-funpow
inf-continuous-mono)
  moreover have α ((f ~ i) top) = (g ~ i) top for i
    by (induction i; simp)
  ultimately show ?thesis
    unfolding inf-continuous-gfp[OF g] by simp
qed

```

lemma *gfp-transfer-bounded*:

```

assumes P: P (f top) ∧ x. P x ⇒ P (f x) ∧ M. antimono M ⇒ (∧ i. P (M
i)) ⇒ P (INF i::nat. M i)
assumes α: ∧ M. antimono M ⇒ (∧ i::nat. P (M i)) ⇒ α (INF i. M i) =
(INF i. α (M i))
assumes f: inf-continuous f and g: inf-continuous g
assumes [simp]: ∧ x. P x ⇒ α (f x) = g (α x)
assumes g-bound: ∧ x. g x ≤ α (f top)
shows α (gfp f) = GFP g
proof (rule antisym)
  note mono-g = inf-continuous-mono[OF g]

```

```

  have P-pow: P ((f ~ i) (f top)) for i
    by (induction i) (auto intro!: P)

```

```

  have antimono-pow: antimono (λ i. (f ~ i) top)
  unfolding antimono-iff-le-Suc

```

proof

```

  fix i show (f ~ Suc i) top ≤ (f ~ i) top

```

```

  proof (induct i)

```

```

    case Suc thus ?case using monoD[OF inf-continuous-mono[OF f] Suc] by

```

auto

```

    qed (simp add: le-fun-def)

```

qed

```

  have antimono-pow2: antimono (λ i. (f ~ i) (f top))

```

proof

```

show  $x \leq y \implies (f \frown y) (f \text{ top}) \leq (f \frown x) (f \text{ top})$  for  $x \ y$ 
  using antimono-pow[THEN antimonoD, of Suc x Suc y]
  unfolding funpow-Suc-right by simp
qed

have gfp-f:  $\text{gfp } f = (\text{INF } i. (f \frown i) (f \text{ top}))$ 
  unfolding inf-continuous-gfp[OF f]
proof (rule INF-eq)
  show  $\exists j \in \text{UNIV}. (f \frown j) (f \text{ top}) \leq (f \frown i) \text{ top}$  for  $i$ 
    by (intro bexI[of - i - 1]) (auto simp: diff-Suc funpow-Suc-right simp del:
funpow.simps(2) split: nat.split)
  show  $\exists j \in \text{UNIV}. (f \frown j) \text{ top} \leq (f \frown i) (f \text{ top})$  for  $i$ 
    by (intro bexI[of - Suc i]) (auto simp: funpow-Suc-right simp del: fun-
pow.simps(2))
qed

have P-lfp:  $P (\text{gfp } f)$ 
  unfolding gfp-f by (auto intro!: P P-pow antimono-pow2)

have  $\alpha (\text{gfp } f) = (\text{INF } i. \alpha ((f \frown i) (f \text{ top})))$ 
  unfolding gfp-f by (rule  $\alpha$ ) (auto intro!: P-pow antimono-pow2)
also have  $\dots \geq \text{gfp } g$ 
proof (rule INF-greatest)
  fix  $i$  show  $\text{gfp } g \leq \alpha ((f \frown i) (f \text{ top}))$ 
  proof (induction i)
    case (Suc n) then show ?case
      by (subst gfp-unfold[OF mono-g]) (simp add: monoD[OF mono-g] P-pow)
  next
    case 0
    have  $\text{gfp } g \leq \alpha (f \text{ top})$ 
      by (subst gfp-unfold[OF mono-g]) (rule g-bound)
    then show ?case
      by simp
  qed
qed
finally show  $\text{gfp } g \leq \alpha (\text{gfp } f)$  .

show  $\alpha (\text{gfp } f) \leq \text{gfp } g$ 
proof (induction rule: gfp-ordinal-induct[OF mono-g])
  case (1 S) then show ?case
    by (subst gfp-unfold[OF inf-continuous-mono[OF f]])
      (simp add: monoD[OF mono-g] P-lfp)
qed (auto intro: Inf-greatest)
qed

```

35.1.1 Least fixed points in countable complete lattices

definition (*in countable-complete-lattice*) *cclfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where $\text{cclfp } f = (\text{SUP } i. (f \frown i) \text{ bot})$

```

lemma cclfp-unfold:
  assumes sup-continuous F shows cclfp F = F (cclfp F)
proof -
  have cclfp F = (SUP i. F ((F  $\sim$  i) bot))
    unfolding cclfp-def
    by (subst UNIV-nat-eq) (simp add: image-comp)
  also have ... = F (cclfp F)
    unfolding cclfp-def
    by (intro sup-continuousD[symmetric] assms mono-funpow sup-continuous-mono)
  finally show ?thesis .
qed

lemma cclfp-lowerbound: assumes f: mono f and A: f A  $\leq$  A shows cclfp f  $\leq$  A
  unfolding cclfp-def
proof (intro ccSUP-least)
  fix i show (f  $\sim$  i) bot  $\leq$  A
  proof (induction i)
    case (Suc i) from monoD[OF f this] A show ?case
      by auto
  qed simp
qed simp

lemma cclfp-transfer:
  assumes sup-continuous  $\alpha$  mono f
  assumes  $\alpha$  bot = bot  $\wedge$   $x. \alpha (f x) = g (\alpha x)$ 
  shows  $\alpha (cclfp f) = cclfp g$ 
proof -
  have  $\alpha (cclfp f) = (SUP i. \alpha ((f \sim i) bot))$ 
    unfolding cclfp-def by (intro sup-continuousD assms mono-funpow sup-continuous-mono)
  moreover have  $\alpha ((f \sim i) bot) = (g \sim i) bot$  for i
    by (induction i) (simp-all add: assms)
  ultimately show ?thesis
    by (simp add: cclfp-def)
qed

end

```

36 Extended natural numbers (i.e. with infinity)

theory *Extended-Nat*

imports *Main Countable Order-Continuity*

begin

class *infinity* =

fixes *infinity* :: 'a ($\langle \infty \rangle$)

context

fixes *f* :: *nat* \Rightarrow 'a:: $\{canonically-ordered-monoid-add, linorder-topology, com-$


```

plete-linorder}
begin

lemma sums-SUP[simp, intro]: f sums (SUP n.  $\sum i < n. f i$ )
  unfolding sums-def by (intro LIMSEQ-SUP monoI sum-mono2 zero-le) auto

lemma suminf-eq-SUP: suminf f = (SUP n.  $\sum i < n. f i$ )
  using sums-SUP by (rule sums-unique[symmetric])

end

```

36.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

```
typedef enat = UNIV :: nat option set ..
```

TODO: introduce enat as coinductive datatype, enat is just *of-nat*

```

definition enat :: nat  $\Rightarrow$  enat where
  enat n = Abs-enat (Some n)

```

```

instantiation enat :: infinity
begin

```

```

definition  $\infty$  = Abs-enat None
instance ..

```

```
end
```

```
instance enat :: countable
```

```
proof
```

```
  show  $\exists$  to-nat::enat  $\Rightarrow$  nat. inj to-nat
```

```
  by (rule exI[of - to-nat  $\circ$  Rep-enat]) (simp add: inj-on-def Rep-enat-inject)
```

```
qed
```

```
old-rep-datatype enat  $\infty$  :: enat
```

```
proof -
```

```
  fix P i assume  $\bigwedge j. P (enat j) P \infty$ 
```

```
  then show P i
```

```
  proof induct
```

```
    case (Abs-enat y) then show ?case
```

```
    by (cases y rule: option.exhaust)
```

```
      (auto simp: enat-def infinity-enat-def)
```

```
  qed
```

```
qed (auto simp add: enat-def infinity-enat-def Abs-enat-inject)
```

```
declare [[coercion enat::nat $\Rightarrow$ enat]]
```

```
lemmas enat2-cases = enat.exhaust[case-product enat.exhaust]
```

lemmas *enat3-cases* = *enat.exhaust*[*case-product enat.exhaust enat.exhaust*]

lemma *not-infinity-eq* [*iff*]: $(x \neq \infty) = (\exists i. x = \text{enat } i)$
by (*cases x*) *auto*

lemma *not-enat-eq* [*iff*]: $(\forall y. x \neq \text{enat } y) = (x = \infty)$
by (*cases x*) *auto*

lemma *enat-ex-split*: $(\exists c::\text{enat}. P \ c) \longleftrightarrow P \ \infty \vee (\exists c::\text{nat}. P \ c)$
by (*metis enat.exhaust*)

primrec *the-enat* :: *enat* \Rightarrow *nat*
where *the-enat* (*enat n*) = *n*

36.2 Constructors and numbers

instantiation *enat* :: *zero-neq-one*
begin

definition
 $0 = \text{enat } 0$

definition
 $1 = \text{enat } 1$

instance
proof **qed** (*simp add: zero-enat-def one-enat-def*)

end

definition *eSuc* :: *enat* \Rightarrow *enat* **where**
 $eSuc \ i = (\text{case } i \text{ of } \text{enat } n \Rightarrow \text{enat } (\text{Suc } n) \mid \infty \Rightarrow \infty)$

lemma *enat-0* [*code-post*]: *enat 0* = *0*
by (*simp add: zero-enat-def*)

lemma *enat-1* [*code-post*]: *enat 1* = *1*
by (*simp add: one-enat-def*)

lemma *enat-0-iff*: *enat x* = *0* $\longleftrightarrow x = 0$ *0* = *enat x* $\longleftrightarrow x = 0$
by (*auto simp add: zero-enat-def*)

lemma *enat-1-iff*: *enat x* = *1* $\longleftrightarrow x = 1$ *1* = *enat x* $\longleftrightarrow x = 1$
by (*auto simp add: one-enat-def*)

lemma *one-eSuc*: *1* = *eSuc 0*
by (*simp add: zero-enat-def one-enat-def eSuc-def*)

lemma *infinity-ne-i0* [*simp*]: $(\infty::\text{enat}) \neq 0$

```

by (simp add: zero-enat-def)

lemma i0-ne-infinity [simp]: 0 ≠ (∞::enat)
by (simp add: zero-enat-def)

lemma zero-one-enat-neq:
  ¬ 0 = (1::enat)
  ¬ 1 = (0::enat)
unfolding zero-enat-def one-enat-def by simp-all

lemma infinity-ne-i1 [simp]: (∞::enat) ≠ 1
by (simp add: one-enat-def)

lemma i1-ne-infinity [simp]: 1 ≠ (∞::enat)
by (simp add: one-enat-def)

lemma eSuc-enat: eSuc (enat n) = enat (Suc n)
by (simp add: eSuc-def)

lemma eSuc-infinity [simp]: eSuc ∞ = ∞
by (simp add: eSuc-def)

lemma eSuc-ne-0 [simp]: eSuc n ≠ 0
by (simp add: eSuc-def zero-enat-def split: enat.splits)

lemma zero-ne-eSuc [simp]: 0 ≠ eSuc n
by (rule eSuc-ne-0 [symmetric])

lemma eSuc-inject [simp]: eSuc m = eSuc n ⟷ m = n
by (simp add: eSuc-def split: enat.splits)

lemma eSuc-enat-iff: eSuc x = enat y ⟷ (∃ n. y = Suc n ∧ x = enat n)
by (cases y) (auto simp: enat-0 eSuc-enat[symmetric])

lemma enat-eSuc-iff: enat y = eSuc x ⟷ (∃ n. y = Suc n ∧ enat n = x)
by (cases y) (auto simp: enat-0 eSuc-enat[symmetric])

```

36.3 Addition

```

instantiation enat :: comm-monoid-add
begin

```

```

definition [nitpick-simp]:
  m + n = (case m of ∞ ⇒ ∞ | enat m ⇒ (case n of ∞ ⇒ ∞ | enat n ⇒ enat
(m + n)))

```

```

lemma plus-enat-simps [simp, code]:
  fixes q :: enat
  shows enat m + enat n = enat (m + n)

```

```

and  $\infty + q = \infty$ 
and  $q + \infty = \infty$ 
by (simp-all add: plus-enat-def split: enat.splits)

```

instance

proof

```

  fix  $n\ m\ q :: \text{enat}$ 
  show  $n + m + q = n + (m + q)$ 
    by (cases n m q rule: enat3-cases) auto
  show  $n + m = m + n$ 
    by (cases n m rule: enat2-cases) auto
  show  $0 + n = n$ 
    by (cases n) (simp-all add: zero-enat-def)

```

qed

end

lemma *eSuc-plus-1*:

```

   $e\text{Suc } n = n + 1$ 
  by (cases n) (simp-all add: eSuc-enat one-enat-def)

```

lemma *plus-1-eSuc*:

```

   $1 + q = e\text{Suc } q$ 
   $q + 1 = e\text{Suc } q$ 
  by (simp-all add: eSuc-plus-1 ac-simps)

```

lemma *iadd-Suc*: $e\text{Suc } m + n = e\text{Suc } (m + n)$

```

  by (simp add: eSuc-plus-1 ac-simps)

```

lemma *iadd-Suc-right*: $m + e\text{Suc } n = e\text{Suc } (m + n)$

```

  by (metis add.commute iadd-Suc)

```

36.4 Multiplication

instantiation *enat* :: {*comm-semiring-1*, *semiring-no-zero-divisors*}

begin

definition *times-enat-def* [*nitpick-simp*]:

```

   $m * n = (\text{case } m \text{ of } \infty \Rightarrow \text{if } n = 0 \text{ then } 0 \text{ else } \infty \mid \text{enat } m \Rightarrow$ 
     $(\text{case } n \text{ of } \infty \Rightarrow \text{if } m = 0 \text{ then } 0 \text{ else } \infty \mid \text{enat } n \Rightarrow \text{enat } (m * n)))$ 

```

lemma *times-enat-simps* [*simp*, *code*]:

```

   $\text{enat } m * \text{enat } n = \text{enat } (m * n)$ 
   $\infty * \infty = (\infty :: \text{enat})$ 
   $\infty * \text{enat } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \infty)$ 
   $\text{enat } m * \infty = (\text{if } m = 0 \text{ then } 0 \text{ else } \infty)$ 
  unfolding times-enat-def zero-enat-def
  by (simp-all split: enat.split)

```

```

instance
proof
  fix a b c :: enat
  show distr: (a + b) * c = a * c + b * c
    unfolding times-enat-def zero-enat-def
    by (simp split: enat.split add: distrib-right)
  show a * (b + c) = a * b + a * c
    by (cases a b c rule: enat3-cases) (auto simp: times-enat-def zero-enat-def
distrib-left)
qed (auto simp: times-enat-def zero-enat-def one-enat-def split: enat.split)

end

lemma mult-eSuc: eSuc m * n = n + m * n
  unfolding eSuc-plus-1 by (simp add: algebra-simps)

lemma mult-eSuc-right: m * eSuc n = m + m * n
  by (metis mult.commute mult-eSuc)

lemma of-nat-eq-enat: of-nat n = enat n
  by (induct n) (auto simp: enat-0 plus-1-eSuc eSuc-enat)

instance enat :: semiring-char-0
proof
  have inj enat by (rule injI) simp
  then show inj (λn. of-nat n :: enat) by (simp add: of-nat-eq-enat)
qed

lemma imult-is-infinity: ((a::enat) * b = ∞) = (a = ∞ ∧ b ≠ 0 ∨ b = ∞ ∧ a ≠
0)
  by (auto simp add: times-enat-def zero-enat-def split: enat.split)

```

36.5 Numerals

```

lemma numeral-eq-enat:
  numeral k = enat (numeral k)
  by (metis of-nat-eq-enat of-nat-numeral)

lemma enat-numeral [code-abbrev]:
  enat (numeral k) = numeral k
  using numeral-eq-enat ..

lemma infinity-ne-numeral [simp]: (∞::enat) ≠ numeral k
  by (simp add: numeral-eq-enat)

lemma numeral-ne-infinity [simp]: numeral k ≠ (∞::enat)
  by (simp add: numeral-eq-enat)

lemma eSuc-numeral [simp]: eSuc (numeral k) = numeral (k + Num.One)

```

by (simp only: eSuc-plus-1 numeral-plus-one)

36.6 Subtraction

instantiation enat :: minus
begin

definition diff-enat-def:

$$a - b = (\text{case } a \text{ of } (\text{enat } x) \Rightarrow (\text{case } b \text{ of } (\text{enat } y) \Rightarrow \text{enat } (x - y) \mid \infty \Rightarrow 0) \mid \infty \Rightarrow \infty)$$

instance ..

end

lemma idiff-enat-enat [simp, code]: enat $a - \text{enat } b = \text{enat } (a - b)$
by (simp add: diff-enat-def)

lemma idiff-infinity [simp, code]: $\infty - n = (\infty :: \text{enat})$
by (simp add: diff-enat-def)

lemma idiff-infinity-right [simp, code]: enat $a - \infty = 0$
by (simp add: diff-enat-def)

lemma idiff-0 [simp]: $(0 :: \text{enat}) - n = 0$
by (cases n, simp-all add: zero-enat-def)

lemmas idiff-enat-0 [simp] = idiff-0 [unfolded zero-enat-def]

lemma idiff-0-right [simp]: $(n :: \text{enat}) - 0 = n$
by (cases n) (simp-all add: zero-enat-def)

lemmas idiff-enat-0-right [simp] = idiff-0-right [unfolded zero-enat-def]

lemma idiff-self [simp]: $n \neq \infty \implies (n :: \text{enat}) - n = 0$
by (auto simp: zero-enat-def)

lemma eSuc-minus-eSuc [simp]: $e\text{Suc } n - e\text{Suc } m = n - m$
by (simp add: eSuc-def split: enat.split)

lemma eSuc-minus-1 [simp]: $e\text{Suc } n - 1 = n$
by (simp add: one-enat-def flip: eSuc-enat zero-enat-def)

36.7 Ordering

instantiation enat :: linordered-ab-semigroup-add
begin

definition [nitpick-simp]:

$$m \leq n = (\text{case } n \text{ of } \text{enat } n1 \Rightarrow (\text{case } m \text{ of } \text{enat } m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow \text{False}))$$

| $\infty \Rightarrow \text{True}$)

definition [nitpick-simp]:

$m < n = (\text{case } m \text{ of } \text{enat } m1 \Rightarrow (\text{case } n \text{ of } \text{enat } n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow \text{True})$
 $\mid \infty \Rightarrow \text{False})$

lemma enat-ord-simps [simp]:

$\text{enat } m \leq \text{enat } n \longleftrightarrow m \leq n$

$\text{enat } m < \text{enat } n \longleftrightarrow m < n$

$q \leq (\infty :: \text{enat})$

$q < (\infty :: \text{enat}) \longleftrightarrow q \neq \infty$

$(\infty :: \text{enat}) \leq q \longleftrightarrow q = \infty$

$(\infty :: \text{enat}) < q \longleftrightarrow \text{False}$

by (simp-all add: less-eq-enat-def less-enat-def split: enat.splits)

lemma numeral-le-enat-iff [simp]:

shows numeral $m \leq \text{enat } n \longleftrightarrow \text{numeral } m \leq n$

by (auto simp: numeral-eq-enat)

lemma numeral-less-enat-iff [simp]:

shows numeral $m < \text{enat } n \longleftrightarrow \text{numeral } m < n$

by (auto simp: numeral-eq-enat)

lemma enat-ord-code [code]:

$\text{enat } m \leq \text{enat } n \longleftrightarrow m \leq n$

$\text{enat } m < \text{enat } n \longleftrightarrow m < n$

$q \leq (\infty :: \text{enat}) \longleftrightarrow \text{True}$

$\text{enat } m < \infty \longleftrightarrow \text{True}$

$\infty \leq \text{enat } n \longleftrightarrow \text{False}$

$(\infty :: \text{enat}) < q \longleftrightarrow \text{False}$

by simp-all

instance

by standard (auto simp add: less-eq-enat-def less-enat-def plus-enat-def split: enat.splits)

end

instance enat :: dioid

proof

fix $a \ b :: \text{enat}$ **show** $(a \leq b) = (\exists c. b = a + c)$

by (cases $a \ b$ rule: enat2-cases) (auto simp: le-iff-add enat-ex-split)

qed

instance enat :: {linordered-nonzero-semiring, strict-ordered-comm-monoid-add}

proof

fix $a \ b \ c :: \text{enat}$

show $a \leq b \implies 0 \leq c \implies c * a \leq c * b$

unfolding times-enat-def less-eq-enat-def zero-enat-def

```

  by (simp split: enat.splits)
show  $a < b \implies c < d \implies a + c < b + d$  for  $a\ b\ c\ d :: \text{enat}$ 
  by (cases a b c d rule: enat2-cases[case-product enat2-cases]) auto
show  $a < b \implies a + 1 < b + 1$ 
  by (metis add-right-mono eSuc-minus-1 eSuc-plus-1 less-le)
qed (simp add: zero-enat-def one-enat-def)

```

```

lemma add-diff-assoc-enat:  $z \leq y \implies x + (y - z) = x + y - (z :: \text{enat})$ 
by (cases x) (auto simp add: diff-enat-def split: enat.split)

```

```

lemma enat-ord-number [simp]:
  (numeral m :: enat)  $\leq$  numeral n  $\longleftrightarrow$  (numeral m :: nat)  $\leq$  numeral n
  (numeral m :: enat)  $<$  numeral n  $\longleftrightarrow$  (numeral m :: nat)  $<$  numeral n
  by (simp-all add: numeral-eq-enat)

```

```

lemma infinity-ileE [elim!]:  $\infty \leq \text{enat } m \implies R$ 
  by (simp add: zero-enat-def less-eq-enat-def split: enat.splits)

```

```

lemma infinity-ilessE [elim!]:  $\infty < \text{enat } m \implies R$ 
  by simp

```

```

lemma eSuc-ile-mono [simp]:  $e\text{Suc } n \leq e\text{Suc } m \longleftrightarrow n \leq m$ 
  by (simp add: eSuc-def less-eq-enat-def split: enat.splits)

```

```

lemma eSuc-mono [simp]:  $e\text{Suc } n < e\text{Suc } m \longleftrightarrow n < m$ 
  by (simp add: eSuc-def less-enat-def split: enat.splits)

```

```

lemma ile-eSuc [simp]:  $n \leq e\text{Suc } n$ 
  by (simp add: eSuc-def less-eq-enat-def split: enat.splits)

```

```

lemma not-eSuc-ilei0 [simp]:  $\neg e\text{Suc } n \leq 0$ 
  by (simp add: zero-enat-def eSuc-def less-eq-enat-def split: enat.splits)

```

```

lemma i0-iless-eSuc [simp]:  $0 < e\text{Suc } n$ 
  by (simp add: zero-enat-def eSuc-def less-enat-def split: enat.splits)

```

```

lemma illess-eSuc0 [simp]:  $(n < e\text{Suc } 0) = (n = 0)$ 
  by (simp add: zero-enat-def eSuc-def less-enat-def split: enat.split)

```

```

lemma ileI1:  $m < n \implies e\text{Suc } m \leq n$ 
  by (simp add: eSuc-def less-eq-enat-def less-enat-def split: enat.splits)

```

```

lemma Suc-ile-eq:  $\text{enat } (\text{Suc } m) \leq n \longleftrightarrow \text{enat } m < n$ 
  by (cases n) auto

```

```

lemma illess-Suc-eq [simp]:  $\text{enat } m < e\text{Suc } n \longleftrightarrow \text{enat } m \leq n$ 
  by (auto simp add: eSuc-def less-enat-def split: enat.splits)

```



```

lemma imult-infinity:  $(0::\text{enat}) < n \implies \infty * n = \infty$ 
  by (simp add: zero-enat-def less-enat-def split: enat.splits)

lemma imult-infinity-right:  $(0::\text{enat}) < n \implies n * \infty = \infty$ 
  by (simp add: zero-enat-def less-enat-def split: enat.splits)

lemma enat-0-less-mult-iff:  $(0 < (m::\text{enat}) * n) = (0 < m \wedge 0 < n)$ 
  by (simp only: zero-less-iff-neq-zero mult-eq-0-iff, simp)

lemma mono-eSuc: mono eSuc
  by (simp add: mono-def)

lemma min-enat-simps [simp]:
  min (enat m) (enat n) = enat (min m n)
  min q 0 = 0
  min 0 q = 0
  min q ( $\infty::\text{enat}$ ) = q
  min ( $\infty::\text{enat}$ ) q = q
  by (auto simp add: min-def)

lemma max-enat-simps [simp]:
  max (enat m) (enat n) = enat (max m n)
  max q 0 = q
  max 0 q = q
  max q  $\infty$  = ( $\infty::\text{enat}$ )
  max  $\infty$  q = ( $\infty::\text{enat}$ )
  by (simp-all add: max-def)

lemma enat-ile:  $n \leq \text{enat } m \implies \exists k. n = \text{enat } k$ 
  by (cases n) simp-all

lemma enat-iless:  $n < \text{enat } m \implies \exists k. n = \text{enat } k$ 
  by (cases n) simp-all

lemma iadd-le-enat-iff:
   $x + y \leq \text{enat } n \iff (\exists y' x'. x = \text{enat } x' \wedge y = \text{enat } y' \wedge x' + y' \leq n)$ 
by(cases x y rule: enat.exhaust[case-product enat.exhaust]) simp-all

lemma chain-incr:  $\forall i. \exists j. Y i < Y j \implies \exists j. \text{enat } k < Y j$ 
proof (induction k)
  case 0
  then show ?case
    using enat-0 zero-less-iff-neq-zero by fastforce
  next
  case (Suc k)
  then show ?case
    by (meson Suc-ile-eq order-le-less-trans)
qed

```

lemma *eSuc-max*: $eSuc (max\ x\ y) = max\ (eSuc\ x)\ (eSuc\ y)$
by (*simp add: eSuc-def split: enat.split*)

lemma *eSuc-Max*:
assumes *finite A* $A \neq \{\}$
shows $eSuc\ (Max\ A) = Max\ (eSuc\ 'A)$
by (*simp add: asms mono-Max-commute mono-eSuc*)

instantiation *enat* :: $\{order-bot, order-top\}$
begin

definition *bot-enat* :: *enat* **where** *bot-enat* = 0
definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
by *standard (simp-all add: bot-enat-def top-enat-def)*

end

lemma *finite-enat-bounded*:
assumes *le-fin*: $\bigwedge y. y \in A \implies y \leq enat\ n$
shows *finite A*
proof (*rule finite-subset*)
show *finite (enat ' {..n})* **by** *blast*
have $A \subseteq enat\ ' \{..n\}$
using *enat-ile le-fin* **by** *fastforce*
then show $A \subseteq enat\ ' \{..n\}$.
qed

36.8 Cancellation simprocs

lemma *add-diff-cancel-enat[simp]*: $x \neq \infty \implies x + y - x = (y::enat)$
by (*metis add.commute add.right-neutral add-diff-assoc-enat idiff-self order-refl*)

lemma *enat-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::enat) \vee b = c$
unfolding *plus-enat-def* **by** (*simp split: enat.split*)

lemma *enat-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::enat) \vee b \leq c$
unfolding *plus-enat-def* **by** (*simp split: enat.split*)

lemma *enat-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::enat) \wedge b < c$
unfolding *plus-enat-def* **by** (*simp split: enat.split*)

lemma *plus-eq-infty-iff-enat*: $(m::enat) + n = \infty \longleftrightarrow m = \infty \vee n = \infty$
using *enat-add-left-cancel* **by** *fastforce*

ML \langle
structure Cancel-Enat-Common =

```

struct
  (* copied from src/HOL/Tools/nat-numeral-simprocs.ML *)
  fun find-first-t - [] = raise TERM(find-first-t, [])
    | find-first-t past u (t::terms) =
        if u aconv t then (rev past @ terms)
        else find-first-t (t::past) u terms

  fun dest-summing (Const (const-name <Groups.plus>, -) $ t $ u, ts) =
        dest-summing (t, dest-summing (u, ts))
    | dest-summing (t, ts) = t :: ts

  val mk-sum = Arith-Data.long-mk-sum
  fun dest-sum t = dest-summing (t, [])
  val find-first = find-first-t []
  val trans-tac = Numeral-Simprocs.trans-tac
  val norm-ss =
    simpset-of (put-simpset HOL-basic-ss context
      |> Simplifier.add-simps @ {thms ac-simps add-0-left add-0-right})
  fun norm-tac ctxt = ALLGOALS (simp-tac (put-simpset norm-ss ctxt))
  fun simplify-meta-eq ctxt cancel-th th =
    Arith-Data.simplify-meta-eq [] ctxt
    ([th, cancel-th] MRS trans)
  fun mk-eq (a, b) = HOLogic.mk-Trueprop (HOLogic.mk-eq (a, b))
end

structure Eq-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-eq
  val dest-bal = HOLogic.dest-bin const-name <HOL.eq> typ <enat>
  fun simp-conv - - = SOME @ {thm enat-add-left-cancel}
)

structure Le-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel const-name <Orderings.less-eq>
  val dest-bal = HOLogic.dest-bin const-name <Orderings.less-eq> typ <enat>
  fun simp-conv - - = SOME @ {thm enat-add-left-cancel-le}
)

structure Less-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel const-name <Orderings.less>
  val dest-bal = HOLogic.dest-bin const-name <Orderings.less> typ <enat>
  fun simp-conv - - = SOME @ {thm enat-add-left-cancel-less}
)
>

simproc-setup enat-eq-cancel
((l::enat) + m = n | (l::enat) = m + n) =

```

$\langle K \text{ (fn } \textit{etxt} \Rightarrow \textit{fn } \textit{ct} \Rightarrow \textit{Eq-Enat-Cancel.proc } \textit{etxt} \text{ (Thm.term-of } \textit{ct})) \rangle$

simproc-setup *enat-le-cancel*

$((l::\textit{enat}) + m \leq n \mid (l::\textit{enat}) \leq m + n) =$

$\langle K \text{ (fn } \textit{etxt} \Rightarrow \textit{fn } \textit{ct} \Rightarrow \textit{Le-Enat-Cancel.proc } \textit{etxt} \text{ (Thm.term-of } \textit{ct})) \rangle$

simproc-setup *enat-less-cancel*

$((l::\textit{enat}) + m < n \mid (l::\textit{enat}) < m + n) =$

$\langle K \text{ (fn } \textit{etxt} \Rightarrow \textit{fn } \textit{ct} \Rightarrow \textit{Less-Enat-Cancel.proc } \textit{etxt} \text{ (Thm.term-of } \textit{ct})) \rangle$

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

36.9 Well-ordering

lemma *less-enatE*:

$\llbracket n < \textit{enat } m; \bigwedge k. \llbracket n = \textit{enat } k; k < m \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$

using *enat-less enat-ord-simps(2)* **by** *blast*

lemma *less-infinityE*:

$\llbracket n < \infty; \bigwedge k. n = \textit{enat } k \Longrightarrow P \rrbracket \Longrightarrow P$

by *auto*

lemma *enat-less-induct*:

assumes $\bigwedge n. \forall m::\textit{enat}. m < n \longrightarrow P \text{ } m \Longrightarrow P \text{ } n$

shows $P \text{ } n$

proof –

have $P \text{ (enat } k) \text{ for } k$

by (*induction k rule: less-induct*) (*metis less-enatE assms*)

then show *?thesis*

by (*metis enat.exhaust less-infinityE assms*)

qed

instance *enat :: wellorder*

proof

fix P **and** n

assume *hyp*: $(\bigwedge n::\textit{enat}. (\bigwedge m::\textit{enat}. m < n \Longrightarrow P \text{ } m) \Longrightarrow P \text{ } n)$

show $P \text{ } n$ **by** (*blast intro: enat-less-induct hyp*)

qed

36.10 Complete Lattice

instantiation *enat :: complete-lattice*

begin

definition *inf-enat* :: $\textit{enat} \Rightarrow \textit{enat} \Rightarrow \textit{enat}$ **where**

inf-enat = *min*

definition *sup-enat* :: $\textit{enat} \Rightarrow \textit{enat} \Rightarrow \textit{enat}$ **where**

$\text{sup-enat} = \text{max}$

definition $\text{Inf-enat} :: \text{enat set} \Rightarrow \text{enat}$ **where**

$\text{Inf-enat } A = (\text{if } A = \{\} \text{ then } \infty \text{ else } (\text{LEAST } x. x \in A))$

definition $\text{Sup-enat} :: \text{enat set} \Rightarrow \text{enat}$ **where**

$\text{Sup-enat } A = (\text{if } A = \{\} \text{ then } 0 \text{ else if finite } A \text{ then Max } A \text{ else } \infty)$

instance

proof

fix $x :: \text{enat}$ **and** $A :: \text{enat set}$

show $x \in A \Longrightarrow \text{Inf } A \leq x$

unfolding Inf-enat-def **by** $(\text{auto intro: Least-le})$

show $(\bigwedge y. y \in A \Longrightarrow x \leq y) \Longrightarrow x \leq \text{Inf } A$

unfolding Inf-enat-def

by $(\text{cases } A = \{\}) (\text{auto intro: LeastI2-ex})$

show $x \in A \Longrightarrow x \leq \text{Sup } A$

unfolding Sup-enat-def **by** $(\text{cases finite } A) \text{ auto}$

show $(\bigwedge y. y \in A \Longrightarrow y \leq x) \Longrightarrow \text{Sup } A \leq x$

unfolding Sup-enat-def **using** $\text{finite-enat-bounded}$ **by** auto

qed $(\text{simp-all add: inf-enat-def sup-enat-def bot-enat-def top-enat-def Inf-enat-def Sup-enat-def})$

end

instance $\text{enat} :: \text{complete-linorder ..}$

lemma $\text{eSuc-Sup: } A \neq \{\} \Longrightarrow \text{eSuc } (\text{Sup } A) = \text{Sup } (\text{eSuc } ` A)$

by $(\text{auto simp add: Sup-enat-def eSuc-Max inj-on-def dest: finite-imageD})$

lemma $\text{sup-continuous-eSuc: sup-continuous } f \Longrightarrow \text{sup-continuous } (\lambda x. \text{eSuc } (f x))$

using $\text{eSuc-Sup [of - ` UNIV]}$ **by** $(\text{auto simp: sup-continuous-def image-comp})$

36.11 Traditional theorem names

lemmas $\text{enat-defs} = \text{zero-enat-def one-enat-def eSuc-def}$

$\text{plus-enat-def less-eq-enat-def less-enat-def}$

lemma $\text{iadd-is-0: } (m + n = (0 :: \text{enat})) = (m = 0 \wedge n = 0)$

by $(\text{rule add-eq-0-iff-both-eq-0})$

lemma $\text{i0-lb: } (0 :: \text{enat}) \leq n$

by (rule zero-le)

lemma $\text{ile0-eq: } n \leq (0 :: \text{enat}) \longleftrightarrow n = 0$

by (rule le-zero-eq)

lemma $\text{not-iless0: } \neg n < (0 :: \text{enat})$

by $(\text{rule not-less-zero})$

```

lemma i0-less[simp]:  $(0::\text{enat}) < n \longleftrightarrow n \neq 0$ 
  by (rule zero-less-iff-neq-zero)

lemma imult-is-0:  $((m::\text{enat}) * n = 0) = (m = 0 \vee n = 0)$ 
  by (rule mult-eq-0-iff)

end

```

37 Liminf and Limsup on conditionally complete lattices

```

theory Liminf-Limsup
imports Complex-Main
begin

```

```

lemma (in conditionally-complete-linorder) le-cSup-iff:
  assumes  $A \neq \{\}$  bdd-above A
  shows  $x \leq \text{Sup } A \longleftrightarrow (\forall y < x. \exists a \in A. y < a)$ 
proof safe
  fix  $y$  assume  $x \leq \text{Sup } A$   $y < x$ 
  then have  $y < \text{Sup } A$  by auto
  then show  $\exists a \in A. y < a$ 
    unfolding less-cSup-iff[OF assms] .
qed (auto elim!: allE[of - Sup A] simp add: not-le[symmetric] cSup-upper assms)

```

```

lemma (in conditionally-complete-linorder) le-cSUP-iff:
   $A \neq \{\} \implies \text{bdd-above } (f'A) \implies x \leq \text{Sup } (f' A) \longleftrightarrow (\forall y < x. \exists i \in A. y < f i)$ 
  using le-cSup-iff [of f' A] by simp

```

```

lemma le-cSup-iff-less:
  fixes  $x :: 'a :: \{\text{conditionally-complete-linorder}, \text{dense-linorder}\}$ 
  shows  $A \neq \{\} \implies \text{bdd-above } (f'A) \implies x \leq (\text{SUP } i \in A. f i) \longleftrightarrow (\forall y < x. \exists i \in A. y \leq f i)$ 
  by (simp add: le-cSUP-iff)
  (blast intro: less-imp-le less-trans less-le-trans dest: dense)

```

```

lemma le-Sup-iff-less:
  fixes  $x :: 'a :: \{\text{complete-linorder}, \text{dense-linorder}\}$ 
  shows  $x \leq (\text{SUP } i \in A. f i) \longleftrightarrow (\forall y < x. \exists i \in A. y \leq f i)$  (is ?lhs = ?rhs)
  unfolding le-SUP-iff
  by (blast intro: less-imp-le less-trans less-le-trans dest: dense)

```

```

lemma (in conditionally-complete-linorder) cInf-le-iff:
  assumes  $A \neq \{\}$  bdd-below A
  shows  $\text{Inf } A \leq x \longleftrightarrow (\forall y > x. \exists a \in A. y > a)$ 
proof safe
  fix  $y$  assume  $x \geq \text{Inf } A$   $y > x$ 

```

then have $y > \text{Inf } A$ **by** *auto*
then show $\exists a \in A. y > a$
unfolding *cInf-less-iff* [*OF assms*] .
qed (*auto elim!*: *allE*[*of - Inf A*] *simp add*: *not-le[symmetric]* *cInf-lower assms*)

lemma (*in conditionally-complete-linorder*) *cINF-le-iff*:
 $A \neq \{\}$ $\implies \text{bdd-below } (f'A) \implies \text{Inf } (f' A) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. y > f i)$
using *cInf-le-iff* [*of f' A*] **by** *simp*

lemma *cInf-le-iff-less*:
fixes $x :: 'a :: \{\text{conditionally-complete-linorder}, \text{dense-linorder}\}$
shows $A \neq \{\} \implies \text{bdd-below } (f'A) \implies (\text{INF } i \in A. f i) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. f i \leq y)$
by (*simp add*: *cINF-le-iff*)
(blast intro: less-imp-le less-trans le-less-trans dest: dense)

lemma *Inf-le-iff-less*:
fixes $x :: 'a :: \{\text{complete-linorder}, \text{dense-linorder}\}$
shows $(\text{INF } i \in A. f i) \leq x \longleftrightarrow (\forall y > x. \exists i \in A. f i \leq y)$
unfolding *INF-le-iff*
by (*blast intro: less-imp-le less-trans le-less-trans dest: dense*)

lemma *SUP-pair*:
fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$
shows $(\text{SUP } i \in A. \text{SUP } j \in B. f i j) = (\text{SUP } p \in A \times B. f (\text{fst } p) (\text{snd } p))$
by (*rule antisym*) (*auto intro!*: *SUP-least SUP-upper2*)

lemma *INF-pair*:
fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$
shows $(\text{INF } i \in A. \text{INF } j \in B. f i j) = (\text{INF } p \in A \times B. f (\text{fst } p) (\text{snd } p))$
by (*rule antisym*) (*auto intro!*: *INF-greatest INF-lower2*)

lemma *INF-Sigma*:
fixes $f :: - \Rightarrow - \Rightarrow - :: \text{complete-lattice}$
shows $(\text{INF } i \in A. \text{INF } j \in B i. f i j) = (\text{INF } p \in \text{Sigma } A B. f (\text{fst } p) (\text{snd } p))$
by (*rule antisym*) (*auto intro!*: *INF-greatest INF-lower2*)

37.0.1 Liminf and Limsup

definition *Liminf* $:: 'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$ **where**
 $\text{Liminf } F f = (\text{SUP } P \in \{P. \text{eventually } P F\}. \text{INF } x \in \{x. P x\}. f x)$

definition *Limsup* $:: 'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b :: \text{complete-lattice}$ **where**
 $\text{Limsup } F f = (\text{INF } P \in \{P. \text{eventually } P F\}. \text{SUP } x \in \{x. P x\}. f x)$

abbreviation *liminf* $\equiv \text{Liminf sequentially}$

abbreviation *limsup* $\equiv \text{Limsup sequentially}$

lemma *Liminf-eqI*:

$(\bigwedge P. \text{eventually } P \ F \implies \text{Inf } (f \text{ ' } (\text{Collect } P)) \leq x) \implies$
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \implies \text{Inf } (f \text{ ' } (\text{Collect } P)) \leq y) \implies x \leq y) \implies \text{Liminf}$
 $F \ f = x$
unfolding *Liminf-def* **by** (*auto intro!*: *SUP-eqI*)

lemma *Limsup-eqI*:

$(\bigwedge P. \text{eventually } P \ F \implies x \leq \text{Sup } (f \text{ ' } (\text{Collect } P))) \implies$
 $(\bigwedge y. (\bigwedge P. \text{eventually } P \ F \implies y \leq \text{Sup } (f \text{ ' } (\text{Collect } P))) \implies y \leq x) \implies$
 $\text{Limsup } F \ f = x$
unfolding *Limsup-def* **by** (*auto intro!*: *INF-eqI*)

lemma *liminf-SUP-INF*: $\text{liminf } f = (\text{SUP } n. \text{INF } m \in \{n..\}. f \ m)$

unfolding *Liminf-def* *eventually-sequentially*
by (*rule SUP-eq*) (*auto simp: atLeast-def intro!*: *INF-mono*)

lemma *limsup-INF-SUP*: $\text{limsup } f = (\text{INF } n. \text{SUP } m \in \{n..\}. f \ m)$

unfolding *Limsup-def* *eventually-sequentially*
by (*rule INF-eq*) (*auto simp: atLeast-def intro!*: *SUP-mono*)

lemma *mem-limsup-iff*: $x \in \text{limsup } A \longleftrightarrow (\exists_F n \text{ in sequentially. } x \in A \ n)$

by (*simp add: Limsup-def*) (*metis (mono-tags) eventually-mono not-frequently*)

lemma *mem-liminf-iff*: $x \in \text{liminf } A \longleftrightarrow (\forall_F n \text{ in sequentially. } x \in A \ n)$

by (*simp add: Liminf-def*) (*metis (mono-tags) eventually-mono*)

lemma *Limsup-const*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Limsup } F \ (\lambda x. \ c) = c$

proof –

have *: $\bigwedge P. \text{Ex } P \longleftrightarrow P \neq (\lambda x. \text{False})$ **by** *auto*

have $\bigwedge P. \text{eventually } P \ F \implies (\text{SUP } x \in \{x. \ P \ x\}. \ c) = c$

using *ntriv* **by** (*intro SUP-const*) (*auto simp: eventually-False **)

then show *?thesis*

apply (*auto simp add: Limsup-def*)

apply (*rule INF-const*)

apply *auto*

using *eventually-True* **apply** *blast*

done

qed

lemma *Liminf-const*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Liminf } F \ (\lambda x. \ c) = c$

proof –

have *: $\bigwedge P. \text{Ex } P \longleftrightarrow P \neq (\lambda x. \text{False})$ **by** *auto*

have $\bigwedge P. \text{eventually } P \ F \implies (\text{INF } x \in \{x. \ P \ x\}. \ c) = c$

using *ntriv* **by** (*intro INF-const*) (*auto simp: eventually-False **)

then show *?thesis*


```

apply (auto simp add: Liminf-def)
apply (rule SUP-const)
apply auto
using eventually-True apply blast
done
qed

```

lemma *Liminf-mono*:

```

assumes ev: eventually ( $\lambda x. f\ x \leq g\ x$ ) F
shows Liminf F f  $\leq$  Liminf F g
unfolding Liminf-def
proof (safe intro!: SUP-mono)
  fix P assume eventually P F
  with ev have eventually ( $\lambda x. f\ x \leq g\ x \wedge P\ x$ ) F (is eventually ?Q F) by (rule
    eventually-conj)
  then show  $\exists Q \in \{P. \text{eventually } P\ F\}. \text{Inf } (f \text{ ' (Collect } P)) \leq \text{Inf } (g \text{ ' (Collect } Q))$ 
    by (intro bexI[of - ?Q]) (auto intro!: INF-mono)
qed

```

lemma *Liminf-eq*:

```

assumes eventually ( $\lambda x. f\ x = g\ x$ ) F
shows Liminf F f = Liminf F g
by (intro antisym Liminf-mono eventually-mono[OF assms]) auto

```

lemma *Limsup-mono*:

```

assumes ev: eventually ( $\lambda x. f\ x \leq g\ x$ ) F
shows Limsup F f  $\leq$  Limsup F g
unfolding Limsup-def
proof (safe intro!: INF-mono)
  fix P assume eventually P F
  with ev have eventually ( $\lambda x. f\ x \leq g\ x \wedge P\ x$ ) F (is eventually ?Q F) by (rule
    eventually-conj)
  then show  $\exists Q \in \{P. \text{eventually } P\ F\}. \text{Sup } (f \text{ ' (Collect } Q)) \leq \text{Sup } (g \text{ ' (Collect } P))$ 
    by (intro bexI[of - ?Q]) (auto intro!: SUP-mono)
qed

```

lemma *Limsup-eq*:

```

assumes eventually ( $\lambda x. f\ x = g\ x$ ) net
shows Limsup net f = Limsup net g
by (intro antisym Limsup-mono eventually-mono[OF assms]) auto

```

lemma *Liminf-bot[simp]*: $\text{Liminf bot } f = \text{top}$

```

unfolding Liminf-def top-unique[symmetric]
by (rule SUP-upper2[where  $i = \lambda x. \text{False}$ ]) simp-all

```

lemma *Limsup-bot[simp]*: $\text{Limsup bot } f = \text{bot}$

```

unfolding Limsup-def bot-unique[symmetric]

```

by (rule INF-lower2[where $i=\lambda x. \text{False}$]) simp-all

lemma *Liminf-le-Limsup*:

assumes *ntriv*: $\neg \text{trivial-limit } F$

shows $\text{Liminf } F f \leq \text{Limsup } F f$

unfolding *Limsup-def* *Liminf-def*

apply (rule SUP-least)

apply (rule INF-greatest)

proof safe

fix *P Q* assume *eventually P F eventually Q F*

then have *eventually* $(\lambda x. P x \wedge Q x) F$ (is *eventually ?C F*) by (rule *eventually-conj*)

then have *not-False*: $(\lambda x. P x \wedge Q x) \neq (\lambda x. \text{False})$

using *ntriv* by (auto simp add: *eventually-False*)

have *Inf* $(f ' (\text{Collect } P)) \leq \text{Inf } (f ' (\text{Collect } ?C))$

by (rule INF-mono) auto

also have $\dots \leq \text{Sup } (f ' (\text{Collect } ?C))$

using *not-False* by (intro INF-le-SUP) auto

also have $\dots \leq \text{Sup } (f ' (\text{Collect } Q))$

by (rule SUP-mono) auto

finally show $\text{Inf } (f ' (\text{Collect } P)) \leq \text{Sup } (f ' (\text{Collect } Q))$.

qed

lemma *Liminf-bounded*:

assumes *le*: *eventually* $(\lambda n. C \leq X n) F$

shows $C \leq \text{Liminf } F X$

using *Liminf-mono*[OF *le*] *Liminf-const*[of *F C*]

by (cases $F = \text{bot}$) simp-all

lemma *Limsup-bounded*:

assumes *le*: *eventually* $(\lambda n. X n \leq C) F$

shows $\text{Limsup } F X \leq C$

using *Limsup-mono*[OF *le*] *Limsup-const*[of *F C*]

by (cases $F = \text{bot}$) simp-all

lemma *le-Limsup*:

assumes *F*: $F \neq \text{bot}$ and *x*: $\forall_F x \text{ in } F. l \leq f x$

shows $l \leq \text{Limsup } F f$

using *F* *Liminf-bounded*[of *l f F*] *Liminf-le-Limsup*[of *F f*] *order.trans x* by blast

lemma *Liminf-le*:

assumes *F*: $F \neq \text{bot}$ and *x*: $\forall_F x \text{ in } F. f x \leq l$

shows $\text{Liminf } F f \leq l$

using *F* *Liminf-le-Limsup* *Limsup-bounded* *order.trans x* by blast

lemma *le-Liminf-iff*:

fixes *X* :: $- \Rightarrow - :: \text{complete-linorder}$

shows $C \leq \text{Liminf } F X \longleftrightarrow (\forall y < C. \text{eventually } (\lambda x. y < X x) F)$

proof –

```

have eventually ( $\lambda x. y < X x$ ) F
  if eventually P F  $y < \text{Inf } (X \text{ ' } (\text{Collect } P))$  for  $y$  P
  using that by (auto elim!: eventually-mono dest: less-INF-D)
moreover
have  $\exists P. \text{eventually } P F \wedge y < \text{Inf } (X \text{ ' } (\text{Collect } P))$ 
  if  $y < C$  and  $y: \forall y < C. \text{eventually } (\lambda x. y < X x) F$  for  $y$  P
proof (cases  $\exists z. y < z \wedge z < C$ )
  case True
    then obtain  $z$  where  $z: y < z \wedge z < C ..$ 
    moreover from  $z$  have  $z \leq \text{Inf } (X \text{ ' } \{x. z < X x\})$ 
      by (auto intro!: INF-greatest)
    ultimately show ?thesis
      using  $y$  by (intro exI[of -  $\lambda x. z < X x$ ]) auto
  next
  case False
    then have  $C \leq \text{Inf } (X \text{ ' } \{x. y < X x\})$ 
      by (intro INF-greatest) auto
    with  $\langle y < C \rangle$  show ?thesis
      using  $y$  by (intro exI[of -  $\lambda x. y < X x$ ]) auto
qed
ultimately show ?thesis
  unfolding Liminf-def le-SUP-iff by auto
qed

lemma Limsup-le-iff:
  fixes  $X :: - \Rightarrow - :: \text{complete-linorder}$ 
  shows  $C \geq \text{Limsup } F X \longleftrightarrow (\forall y > C. \text{eventually } (\lambda x. y > X x) F)$ 
proof -
  { fix  $y$  P assume eventually P F  $y > \text{Sup } (X \text{ ' } (\text{Collect } P))$ 
    then have eventually ( $\lambda x. y > X x$ ) F
      by (auto elim!: eventually-mono dest: SUP-lessD) }
  moreover
  { fix  $y$  P assume  $y > C$  and  $y: \forall y > C. \text{eventually } (\lambda x. y > X x) F$ 
    have  $\exists P. \text{eventually } P F \wedge y > \text{Sup } (X \text{ ' } (\text{Collect } P))$ 
    proof (cases  $\exists z. C < z \wedge z < y$ )
      case True
        then obtain  $z$  where  $z: C < z \wedge z < y ..$ 
        moreover from  $z$  have  $z \geq \text{Sup } (X \text{ ' } \{x. X x < z\})$ 
          by (auto intro!: SUP-least)
        ultimately show ?thesis
          using  $y$  by (intro exI[of -  $\lambda x. z > X x$ ]) auto
      next
      case False
        then have  $C \geq \text{Sup } (X \text{ ' } \{x. X x < y\})$ 
          by (intro SUP-least) (auto simp: not-less)
        with  $\langle y > C \rangle$  show ?thesis
          using  $y$  by (intro exI[of -  $\lambda x. y > X x$ ]) auto
    qed }
  ultimately show ?thesis

```

unfolding *Limsup-def INF-le-iff* **by** *auto*
qed

lemma *less-LiminfD*:

$y < \text{Liminf } F \ (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f\ x > y) \ F$
using *le-Liminf-iff*[*of Liminf F f F f*] **by** *simp*

lemma *Limsup-lessD*:

$y > \text{Limsup } F \ (f :: - \Rightarrow 'a :: \text{complete-linorder}) \implies \text{eventually } (\lambda x. f\ x < y) \ F$
using *Limsup-le-iff*[*of F f Limsup F f*] **by** *simp*

lemma *lim-imp-Liminf*:

fixes $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$
assumes *ntriv*: $\neg \text{trivial-limit } F$
assumes *lim*: $(f \longrightarrow f0) \ F$
shows $\text{Liminf } F\ f = f0$

proof (*intro Liminf-eqI*)

fix P **assume** P : *eventually P F*
then have $\text{eventually } (\lambda x. \text{Inf } (f \ ' (\text{Collect } P)) \leq f\ x) \ F$
by *eventually-elim (auto intro!: INF-lower)*
then show $\text{Inf } (f \ ' (\text{Collect } P)) \leq f0$
by (*rule tendsto-le*[*OF ntriv lim tendsto-const*])

next

fix y **assume** *upper*: $\bigwedge P. \text{eventually } P\ F \implies \text{Inf } (f \ ' (\text{Collect } P)) \leq y$
show $f0 \leq y$

proof *cases*

assume $\exists z. y < z \wedge z < f0$
then obtain z **where** $y < z \wedge z < f0$..
moreover have $z \leq \text{Inf } (f \ ' \{x. z < f\ x\})$
by (*rule INF-greatest*) *simp*
ultimately show *?thesis*

using *lim*[*THEN topological-tendstoD, THEN upper, of {z <..}*] **by** *auto*

next

assume *discrete*: $\neg (\exists z. y < z \wedge z < f0)$

show *?thesis*

proof (*rule classical*)

assume $\neg f0 \leq y$
then have $\text{eventually } (\lambda x. y < f\ x) \ F$
using *lim*[*THEN topological-tendstoD, of {y <..}*] **by** *auto*
then have $\text{eventually } (\lambda x. f0 \leq f\ x) \ F$
using *discrete* **by** (*auto elim!: eventually-mono*)
then have $\text{Inf } (f \ ' \{x. f0 \leq f\ x\}) \leq y$
by (*rule upper*)
moreover have $f0 \leq \text{Inf } (f \ ' \{x. f0 \leq f\ x\})$
by (*intro INF-greatest*) *simp*
ultimately show $f0 \leq y$ **by** *simp*

qed

qed

qed

```

lemma lim-imp-Limsup:
  fixes  $f :: 'a \Rightarrow - :: \{\text{complete-linorder}, \text{linorder-topology}\}$ 
  assumes ntriv:  $\neg \text{trivial-limit } F$ 
  assumes lim:  $(f \longrightarrow f0) F$ 
  shows  $\text{Limsup } F f = f0$ 
proof (intro Limsup-eqI)
  fix  $P$  assume  $P$ : eventually P F
  then have eventually  $(\lambda x. f x \leq \text{Sup } (f \text{ ` } (\text{Collect } P))) F$ 
    by eventually-elim (auto intro!: SUP-upper)
  then show  $f0 \leq \text{Sup } (f \text{ ` } (\text{Collect } P))$ 
    by (rule tendsto-le[OF ntriv tendsto-const lim])
next
  fix  $y$  assume lower:  $\bigwedge P. \text{eventually } P F \implies y \leq \text{Sup } (f \text{ ` } (\text{Collect } P))$ 
  show  $y \leq f0$ 
  proof (cases  $\exists z. f0 < z \wedge z < y$ )
    case True
    then obtain  $z$  where  $f0 < z \wedge z < y$  ..
    moreover have  $\text{Sup } (f \text{ ` } \{x. f x < z\}) \leq z$ 
      by (rule SUP-least) simp
    ultimately show ?thesis
      using lim[THEN topological-tendstoD, THEN lower, of  $\{.. < z\}$ ] by auto
  next
  case False
  show ?thesis
  proof (rule classical)
    assume  $\neg y \leq f0$ 
    then have eventually  $(\lambda x. f x < y) F$ 
      using lim[THEN topological-tendstoD, of  $\{.. < y\}$ ] by auto
    then have eventually  $(\lambda x. f x \leq f0) F$ 
      using False by (auto elim!: eventually-mono simp: not-less)
    then have  $y \leq \text{Sup } (f \text{ ` } \{x. f x \leq f0\})$ 
      by (rule lower)
    moreover have  $\text{Sup } (f \text{ ` } \{x. f x \leq f0\}) \leq f0$ 
      by (intro SUP-least) simp
    ultimately show  $y \leq f0$  by simp
  qed
qed
qed

lemma Liminf-eq-Limsup:
  fixes  $f0 :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$ 
  assumes ntriv:  $\neg \text{trivial-limit } F$ 
  and lim:  $\text{Liminf } F f = f0 \text{ Limsup } F f = f0$ 
  shows  $(f \longrightarrow f0) F$ 
proof (rule order-tendstoI)
  fix  $a$  assume  $f0 < a$ 
  with assms have  $\text{Limsup } F f < a$  by simp
  then obtain  $P$  where eventually P F  $\text{Sup } (f \text{ ` } (\text{Collect } P)) < a$ 

```

```

    unfolding Limsup-def INF-less-iff by auto
  then show eventually ( $\lambda x. f\ x < a$ ) F
    by (auto elim!: eventually-mono dest: SUP-lessD)
next
  fix a assume a < f0
  with assms have a < Liminf F f by simp
  then obtain P where eventually P F a < Inf (f ‘ (Collect P))
    unfolding Liminf-def less-SUP-iff by auto
  then show eventually ( $\lambda x. a < f\ x$ ) F
    by (auto elim!: eventually-mono dest: less-INF-D)
qed

lemma tendsto-iff-Liminf-eq-Limsup:
  fixes f0 :: 'a :: {complete-linorder, linorder-topology}
  shows  $\neg \text{trivial-limit } F \implies (f \longrightarrow f0) \iff (Liminf F f = f0 \wedge Limsup F f = f0)$ 
  by (metis Liminf-eq-Limsup lim-imp-Limsup lim-imp-Liminf)

lemma liminf-subseq-mono:
  fixes X :: nat  $\Rightarrow$  'a :: complete-linorder
  assumes strict-mono r
  shows  $\text{liminf } X \leq \text{liminf } (X \circ r)$ 
proof-
  have  $\bigwedge n. (\text{INF } m \in \{n..\}. X\ m) \leq (\text{INF } m \in \{n..\}. (X \circ r)\ m)$ 
  proof (safe intro!: INF-mono)
    fix n m :: nat assume n  $\leq$  m then show  $\exists ma \in \{n..\}. X\ ma \leq (X \circ r)\ m$ 
      using seq-suble[OF <strict-mono r>, of m] by (intro bexI[of - r m]) auto
  qed
  then show ?thesis by (auto intro!: SUP-mono simp: liminf-SUP-INF comp-def)
qed

lemma limsup-subseq-mono:
  fixes X :: nat  $\Rightarrow$  'a :: complete-linorder
  assumes strict-mono r
  shows  $\text{limsup } (X \circ r) \leq \text{limsup } X$ 
proof-
  have  $(\text{SUP } m \in \{n..\}. (X \circ r)\ m) \leq (\text{SUP } m \in \{n..\}. X\ m)$  for n
  proof (safe intro!: SUP-mono)
    fix m :: nat
    assume n  $\leq$  m
    then show  $\exists ma \in \{n..\}. (X \circ r)\ m \leq X\ ma$ 
      using seq-suble[OF <strict-mono r>, of m] by (intro bexI[of - r m]) auto
  qed
  then show ?thesis
    by (auto intro!: INF-mono simp: limsup-INF-SUP comp-def)
qed

lemma continuous-on-imp-continuous-within:
  continuous-on s f  $\implies t \subseteq s \implies x \in s \implies \text{continuous (at } x \text{ within } t) f$ 

```

unfolding *continuous-on-eq-continuous-within*
by (*auto simp: continuous-within intro: tendsto-within-subset*)

lemma *Liminf-compose-continuous-mono*:

fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b::\{\text{complete-linorder}, \text{linorder-topology}\}$
assumes c : *continuous-on UNIV* f **and** am : *mono* f **and** F : $F \neq \text{bot}$
shows $\text{Liminf } F (\lambda n. f (g \ n)) = f (\text{Liminf } F g)$

proof –

have $*$: $\exists x. P \ x$ **if** *eventually* $P \ F$ **for** P

proof (*rule ccontr*)

assume $\neg ?thesis$

then have $P = (\lambda x. \text{False})$

by *auto*

with $\langle \text{eventually } P \ F \rangle F$ **show** *False*

by *auto*

qed

have $f (\text{SUP } P \in \{P. \text{eventually } P \ F\}. \text{Inf } (g \ ' \ \text{Collect } P)) =$

$\text{Sup } (f \ ' \ (\lambda P. \text{Inf } (g \ ' \ \text{Collect } P)) \ ' \ \{P. \text{eventually } P \ F\})$

using am *continuous-on-imp-continuous-within* [*OF c*]

by (*rule continuous-at-Sup-mono*) (*auto intro: eventually-True*)

then have $f (\text{Liminf } F g) = (\text{SUP } P \in \{P. \text{eventually } P \ F\}. f (\text{Inf } (g \ ' \ \text{Collect } P)))$

by (*simp add: Liminf-def image-comp*)

also have $\dots = (\text{SUP } P \in \{P. \text{eventually } P \ F\}. \text{Inf } (f \ ' \ (g \ ' \ \text{Collect } P)))$

using $*$ *continuous-at-Inf-mono* [*OF am continuous-on-imp-continuous-within* [*OF c*]]

by *auto*

finally show $?thesis$ **by** (*auto simp: Liminf-def image-comp*)

qed

lemma *Limsup-compose-continuous-mono*:

fixes $f :: 'a::\{\text{complete-linorder}, \text{linorder-topology}\} \Rightarrow 'b::\{\text{complete-linorder}, \text{linorder-topology}\}$
assumes c : *continuous-on UNIV* f **and** am : *mono* f **and** F : $F \neq \text{bot}$
shows $\text{Limsup } F (\lambda n. f (g \ n)) = f (\text{Limsup } F g)$

proof –

have $*$: $\exists x. P \ x$ **if** *eventually* $P \ F$ **for** P

proof (*rule ccontr*)

assume $\neg ?thesis$

then have $P = (\lambda x. \text{False})$

by *auto*

with $\langle \text{eventually } P \ F \rangle F$ **show** *False*

by *auto*

qed

have $f (\text{INF } P \in \{P. \text{eventually } P \ F\}. \text{Sup } (g \ ' \ \text{Collect } P)) =$

$\text{Inf } (f \ ' \ (\lambda P. \text{Sup } (g \ ' \ \text{Collect } P)) \ ' \ \{P. \text{eventually } P \ F\})$

using am *continuous-on-imp-continuous-within* [*OF c*]

by (*rule continuous-at-Inf-mono*) (*auto intro: eventually-True*)

then have $f (\text{Limsup } F g) = (\text{INF } P \in \{P. \text{eventually } P \ F\}. f (\text{Sup } (g \ ' \ \text{Collect } P)))$

```

    by (simp add: Limsup-def image-comp)
  also have ... = (INF P ∈ {P. eventually P F}. Sup (f ‘ (g ‘ Collect P)))
    using * continuous-at-Sup-mono [OF am continuous-on-imp-continuous-within
[OF c]]
    by auto
  finally show ?thesis by (auto simp: Limsup-def image-comp)
qed

```

lemma *Liminf-compose-continuous-antimono:*

```

fixes f :: 'a::{complete-linorder,linorder-topology} ⇒ 'b::{complete-linorder,linorder-topology}
assumes c: continuous-on UNIV f
and am: antimono f
and F: F ≠ bot
shows Liminf F (λn. f (g n)) = f (Limsup F g)
proof –
  have *: ∃ x. P x if eventually P F for P
  proof (rule ccontr)
    assume ¬ (∃ x. P x) then have P = (λx. False)
    by auto
  with ⟨eventually P F⟩ F show False
  by auto
qed

```

```

have f (INF P ∈ {P. eventually P F}. Sup (g ‘ Collect P)) =
  Sup (f ‘ (λP. Sup (g ‘ Collect P)) ‘ {P. eventually P F})
  using am continuous-on-imp-continuous-within [OF c]
  by (rule continuous-at-Inf-antimono) (auto intro: eventually-True)
then have f (Limsup F g) = (SUP P ∈ {P. eventually P F}. f (Sup (g ‘ Collect
P)))
  by (simp add: Limsup-def image-comp)
also have ... = (SUP P ∈ {P. eventually P F}. Inf (f ‘ (g ‘ Collect P)))
  using * continuous-at-Sup-antimono [OF am continuous-on-imp-continuous-within
[OF c]]
  by auto
finally show ?thesis
  by (auto simp: Liminf-def image-comp)
qed

```

lemma *Limsup-compose-continuous-antimono:*

```

fixes f :: 'a::{complete-linorder, linorder-topology} ⇒ 'b::{complete-linorder, linorder-topology}
assumes c: continuous-on UNIV f and am: antimono f and F: F ≠ bot
shows Limsup F (λn. f (g n)) = f (Liminf F g)
proof –
  have *: ∃ x. P x if eventually P F for P
  proof (rule ccontr)
    assume ¬ (∃ x. P x) then have P = (λx. False)
    by auto
  with ⟨eventually P F⟩ F show False
  by auto

```



```

qed
have  $f (SUP P \in \{P. \text{eventually } P F\}. Inf (g \text{ ‘ Collect } P)) =$ 
   $Inf (f \text{ ‘ } (\lambda P. Inf (g \text{ ‘ Collect } P)) \text{ ‘ } \{P. \text{eventually } P F\})$ 
  using am continuous-on-imp-continuous-within [OF c]
  by (rule continuous-at-Sup-antimono) (auto intro: eventually-True)
then have  $f (Liminf F g) = (INF P \in \{P. \text{eventually } P F\}. f (Inf (g \text{ ‘ Collect } P)))$ 
by (simp add: Liminf-def image-comp)
also have  $\dots = (INF P \in \{P. \text{eventually } P F\}. Sup (f \text{ ‘ } (g \text{ ‘ Collect } P)))$ 
  using * continuous-at-Inf-antimono [OF am continuous-on-imp-continuous-within [OF c]]
  by auto
finally show ?thesis
  by (auto simp: Limsup-def image-comp)
qed

```

```

lemma Liminf-filtermap-le:  $Liminf (filtermap f F) g \leq Liminf F (\lambda x. g (f x))$ 
apply (cases F = bot, simp)
by (subst Liminf-def)
  (auto simp add: INF-lower Liminf-bounded eventually-filtermap eventually-mono intro!: SUP-least)

```

```

lemma Limsup-filtermap-ge:  $Limsup (filtermap f F) g \geq Limsup F (\lambda x. g (f x))$ 
apply (cases F = bot, simp)
by (subst Limsup-def)
  (auto simp add: SUP-upper Limsup-bounded eventually-filtermap eventually-mono intro!: INF-greatest)

```

```

lemma Liminf-least:  $(\bigwedge P. \text{eventually } P F \implies (INF x \in Collect P. f x) \leq x) \implies$ 
 $Liminf F f \leq x$ 
by (auto intro!: SUP-least simp: Liminf-def)

```

```

lemma Limsup-greatest:  $(\bigwedge P. \text{eventually } P F \implies x \leq (SUP x \in Collect P. f x)) \implies$ 
 $Limsup F f \geq x$ 
by (auto intro!: INF-greatest simp: Limsup-def)

```

```

lemma Liminf-filtermap-ge:  $inj f \implies Liminf (filtermap f F) g \geq Liminf F (\lambda x. g (f x))$ 
apply (cases F = bot, simp)
apply (rule Liminf-least)
subgoal for P
  by (auto simp: eventually-filtermap the-inv-f-f intro!: Liminf-bounded INF-lower2 eventually-mono[of P])
done

```

```

lemma Limsup-filtermap-le:  $inj f \implies Limsup (filtermap f F) g \leq Limsup F (\lambda x. g (f x))$ 
apply (cases F = bot, simp)
apply (rule Limsup-greatest)

```

subgoal for P
 by (auto simp: eventually-filtermap the-inv-f-f
 intro!: Limsup-bounded SUP-upper2 eventually-mono[of P])
 done

lemma *Liminf-filtermap-eq*: $\text{inj } f \implies \text{Liminf } (\text{filtermap } f \ F) \ g = \text{Liminf } F \ (\lambda x. g \ (f \ x))$
 using *Liminf-filtermap-le*[of $f \ F \ g$] *Liminf-filtermap-ge*[of $f \ F \ g$]
 by *simp*

lemma *Limsup-filtermap-eq*: $\text{inj } f \implies \text{Limsup } (\text{filtermap } f \ F) \ g = \text{Limsup } F \ (\lambda x. g \ (f \ x))$
 using *Limsup-filtermap-le*[of $f \ F \ g$] *Limsup-filtermap-ge*[of $F \ g \ f$]
 by *simp*

37.1 More Limits

lemma *convergent-limsup-cl*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
 shows $\text{convergent } X \implies \text{limsup } X = \text{lim } X$
 by (auto simp: convergent-def limI lim-imp-Limsup)

lemma *convergent-liminf-cl*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
 shows $\text{convergent } X \implies \text{liminf } X = \text{lim } X$
 by (auto simp: convergent-def limI lim-imp-Liminf)

lemma *lim-increasing-cl*:
 assumes $\bigwedge n \ m. n \geq m \implies f \ n \geq f \ m$
 obtains l where $f \longrightarrow (l :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\})$
proof
 show $f \longrightarrow (\text{SUP } n. f \ n)$
 using *assms*
 by (intro increasing-tendsto)
 (auto simp: SUP-upper eventually-sequentially less-SUP-iff intro: less-le-trans)
qed

lemma *lim-decreasing-cl*:
 assumes $\bigwedge n \ m. n \geq m \implies f \ n \leq f \ m$
 obtains l where $f \longrightarrow (l :: 'a :: \{\text{complete-linorder}, \text{linorder-topology}\})$
proof
 show $f \longrightarrow (\text{INF } n. f \ n)$
 using *assms*
 by (intro decreasing-tendsto)
 (auto simp: INF-lower eventually-sequentially INF-less-iff intro: le-less-trans)
qed

lemma *compact-complete-linorder*:
 fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$

```

shows  $\exists l r. \text{strict-mono } r \wedge (X \circ r) \longrightarrow l$ 
proof -
  obtain  $r$  where  $\text{strict-mono } r$  and  $\text{mono: monoseq } (X \circ r)$ 
  using  $\text{seq-monosub[of } X]$ 
  unfolding  $\text{comp-def}$ 
  by  $\text{auto}$ 
  then have  $(\forall n m. m \leq n \longrightarrow (X \circ r) m \leq (X \circ r) n) \vee (\forall n m. m \leq n \longrightarrow$ 
 $(X \circ r) n \leq (X \circ r) m)$ 
  by  $(\text{auto simp add: monoseq-def})$ 
  then obtain  $l$  where  $(X \circ r) \longrightarrow l$ 
  using  $\text{lim-increasing-cl[of } X \circ r]$   $\text{lim-decreasing-cl[of } X \circ r]$ 
  by  $\text{auto}$ 
  then show  $?thesis$ 
  using  $\langle \text{strict-mono } r \rangle$  by  $\text{auto}$ 
qed

```

```

lemma  $\text{tendsto-Limsup}$ :
  fixes  $f :: - \Rightarrow 'a :: \{\text{complete-linorder, linorder-topology}\}$ 
  shows  $F \neq \text{bot} \implies \text{Limsup } F f = \text{Liminf } F f \implies (f \longrightarrow \text{Limsup } F f) F$ 
  by  $(\text{subst tendsto-iff-Liminf-eq-Limsup}) \text{ auto}$ 

```

```

lemma  $\text{tendsto-Liminf}$ :
  fixes  $f :: - \Rightarrow 'a :: \{\text{complete-linorder, linorder-topology}\}$ 
  shows  $F \neq \text{bot} \implies \text{Limsup } F f = \text{Liminf } F f \implies (f \longrightarrow \text{Liminf } F f) F$ 
  by  $(\text{subst tendsto-iff-Liminf-eq-Limsup}) \text{ auto}$ 

```

end

38 Extended real number line

```

theory  $\text{Extended-Real}$ 
imports  $\text{Complex-Main}$   $\text{Extended-Nat}$   $\text{Liminf-Limsup}$ 
begin

```

This should be part of *HOL–Library.Extended-Nat* or *HOL–Library.Order-Continuity*, but then the AFP-entry *Jinja-Thread* fails, as it does overload certain named from *Complex-Main*.

```

lemma  $\text{incseq-sumI2}$ :
  fixes  $f :: 'i \Rightarrow \text{nat} \Rightarrow 'a :: \text{ordered-comm-monoid-add}$ 
  shows  $(\bigwedge n. n \in A \implies \text{mono } (f n)) \implies \text{mono } (\lambda i. \sum_{n \in A} f n i)$ 
  unfolding  $\text{incseq-def}$  by  $(\text{auto intro: sum-mono})$ 

```

```

lemma  $\text{incseq-sumI}$ :
  fixes  $f :: \text{nat} \Rightarrow 'a :: \text{ordered-comm-monoid-add}$ 
  assumes  $\bigwedge i. 0 \leq f i$ 
  shows  $\text{incseq } (\lambda i. \text{sum } f \{..< i\})$ 
proof (intro  $\text{incseq-SucI}$ )
  fix  $n$ 
  have  $\text{sum } f \{..< n\} + 0 \leq \text{sum } f \{..< n\} + f n$ 

```

```

    using assms by (rule add-left-mono)
  then show  $\sum f \{.. $n\} \leq \sum f \{.. $Suc\ n\}$ 
    by auto
qed$$ 
```

```

lemma continuous-at-left-imp-sup-continuous:
  fixes  $f :: 'a::\{complete-linorder, linorder-topology\} \Rightarrow 'b::\{complete-linorder, linorder-topology\}$ 
  assumes  $\text{mono } f \wedge x. \text{continuous } (\text{at-left } x) f$ 
  shows  $\text{sup-continuous } f$ 
  unfolding sup-continuous-def
proof safe
  fix  $M :: \text{nat} \Rightarrow 'a$  assume incseq  $M$  then show  $f (\text{SUP } i. M\ i) = (\text{SUP } i. f (M\ i))$ 
    using continuous-at-Sup-mono [OF assms, of range  $M$ ] by (simp add: image-comp)
qed

```

```

lemma sup-continuous-at-left:
  fixes  $f :: 'a::\{complete-linorder, linorder-topology, first-countable-topology\} \Rightarrow 'b::\{complete-linorder, linorder-topology\}$ 
  assumes  $f: \text{sup-continuous } f$ 
  shows  $\text{continuous } (\text{at-left } x) f$ 
proof cases
  assume  $x = \text{bot}$  then show ?thesis
    by (simp add: trivial-limit-at-left-bot)
next
  assume  $x: x \neq \text{bot}$ 
  show ?thesis
    unfolding continuous-within
  proof (intro tendsto-at-left-sequentially[of bot])
    fix  $S :: \text{nat} \Rightarrow 'a$  assume  $S: \text{incseq } S$  and  $S\ x: S \longrightarrow x$ 
    from  $S\ x$  have  $x\text{-eq}: x = (\text{SUP } i. S\ i)$ 
      by (rule LIMSEQ-unique) (intro LIMSEQ-SUP  $S$ )
    show  $(\lambda n. f (S\ n)) \longrightarrow f\ x$ 
      unfolding  $x\text{-eq}$  sup-continuousD[OF  $f\ S$ ]
      using  $S$  sup-continuous-mono[OF  $f$ ] by (intro LIMSEQ-SUP) (auto simp: mono-def)
  qed (insert  $x$ , auto simp: bot-less)
qed

```

```

lemma sup-continuous-iff-at-left:
  fixes  $f :: 'a::\{complete-linorder, linorder-topology, first-countable-topology\} \Rightarrow 'b::\{complete-linorder, linorder-topology\}$ 
  shows  $\text{sup-continuous } f \iff (\forall x. \text{continuous } (\text{at-left } x) f) \wedge \text{mono } f$ 
  using continuous-at-left-imp-sup-continuous sup-continuous-at-left sup-continuous-mono
  by blast

```

```

lemma continuous-at-right-imp-inf-continuous:
  fixes  $f :: 'a::\{complete-linorder, linorder-topology\} \Rightarrow 'b::\{complete-linorder, linorder-topology\}$ 

```

```

assumes mono f  $\wedge x.$  continuous (at-right x) f
shows inf-continuous f
unfolding inf-continuous-def
proof safe
  fix M :: nat  $\Rightarrow$  'a
  assume decseq M
  then show f (INF i. M i) = (INF i. f (M i))
    using continuous-at-Inf-mono [OF assms, of range M]
    by (simp add: image-comp)
qed

lemma inf-continuous-at-right:
  fixes f :: 'a::\{complete-linorder, linorder-topology, first-countable-topology\}  $\Rightarrow$ 
    'b::\{complete-linorder, linorder-topology\}
  assumes f: inf-continuous f
  shows continuous (at-right x) f
proof cases
  assume x = top then show ?thesis
    by (simp add: trivial-limit-at-right-top)
next
  assume x: x  $\neq$  top
  show ?thesis
    unfolding continuous-within
  proof (intro tendsto-at-right-sequentially[of - top])
    fix S :: nat  $\Rightarrow$  'a
    assume S: decseq S and S-x: S  $\longrightarrow$  x
    then have x-eq: x = (INF i. S i)
      using INF-Lim by blast
    show  $(\lambda n. f (S n)) \longrightarrow f x$ 
      unfolding x-eq inf-continuousD[OF f S]
      using S inf-continuous-mono[OF f] by (intro LIMSEQ-INF) (auto simp:
mono-def antimono-def)
    qed (insert x, auto simp: less-top)
  qed

lemma inf-continuous-iff-at-right:
  fixes f :: 'a::\{complete-linorder, linorder-topology, first-countable-topology\}  $\Rightarrow$ 
    'b::\{complete-linorder, linorder-topology\}
  shows inf-continuous f  $\longleftrightarrow (\forall x. \text{continuous (at-right x) f}) \wedge \text{mono f}$ 
  using continuous-at-right-imp-inf-continuous inf-continuous-at-right inf-continuous-mono
  by blast

instantiation enat :: linorder-topology
begin

definition open-enat :: enat set  $\Rightarrow$  bool where
  open-enat = generate-topology (range lessThan  $\cup$  range greaterThan)

instance

```

```

proof qed (rule open-enat-def)

end

lemma open-enat: open {enat n}
proof (cases n)
  case 0
  then have {enat n} = {..< eSuc 0}
    by (auto simp: enat-0)
  then show ?thesis
    by simp
next
  case (Suc n')
  then have {enat n} = {enat n' <..< enat (Suc n)}
    using enat-iless by (fastforce simp: set-eq-iff)
  then show ?thesis
    by simp
qed

lemma open-enat-iff:
  fixes A :: enat set
  shows open A  $\longleftrightarrow$  ( $\infty \in A \longrightarrow (\exists n::nat. \{n <..\} \subseteq A)$ )
proof safe
  assume  $\infty \notin A$ 
  then have A = ( $\bigcup n \in \{n. \text{enat } n \in A\}. \{enat\ } n\}$ )
    by (simp add: set-eq-iff) (metis not-enat-eq)
  moreover have open ...
    by (auto intro: open-enat)
  ultimately show open A
    by simp
next
  fix n assume {enat n <..<}  $\subseteq$  A
  then have A = ( $\bigcup n \in \{n. \text{enat } n \in A\}. \{enat\ } n\} \cup \{enat\ } n <..\}$ )
    using enat-ile leI by (simp add: set-eq-iff) blast
  moreover have open ...
    by (intro open-Un open-UN ballI open-enat open-greaterThan)
  ultimately show open A
    by simp
next
  assume open A  $\infty \in A$ 
  then have generate-topology (range lessThan  $\cup$  range greaterThan) A  $\infty \in A$ 
    unfolding open-enat-def by auto
  then show  $\exists n::nat. \{n <..\} \subseteq A$ 
proof induction
  case (Int A B)
  then obtain n m where {enat n <..<}  $\subseteq$  A {enat m <..<}  $\subseteq$  B
    by auto
  then have {enat (max n m) <..<}  $\subseteq$  A  $\cap$  B
    by (auto simp: subset-eq Ball-def max-def simp flip: enat-ord-code(1))

```

```

    then show ?case
      by auto
  next
    case (UN K)
    then obtain k where  $k \in K$   $\infty \in k$ 
      by auto
    with UN.IH[OF this] show ?case
      by auto
  qed auto
qed

lemma nhds-enat:  $nhds\ x = (if\ x = \infty\ then\ INF\ i.\ principal\ \{enat\ i..\}\ else\ principal\ \{x\})$ 
proof auto
  show  $nhds\ \infty = (INF\ i.\ principal\ \{enat\ i..\})$ 
  proof (rule antisym)
    show  $nhds\ \infty \leq (INF\ i.\ principal\ \{enat\ i..\})$ 
      unfolding nhds-def
      using Ioi-le-Ico by (intro INF-greatest INF-lower) (auto simp: open-enat-iff)
    show  $(INF\ i.\ principal\ \{enat\ i..\}) \leq nhds\ \infty$ 
      unfolding nhds-def
      by (intro INF-greatest) (force intro: INF-lower2[of Suc -] simp add: open-enat-iff
Suc-ile-eq)
  qed
  show  $nhds\ (enat\ i) = principal\ \{enat\ i\}$  for  $i$ 
    by (simp add: nhds-discrete-open open-enat)
  qed

instance enat :: topological-comm-monoid-add
proof
  have [simp]:  $enat\ i \leq aa \implies enat\ i \leq aa + ba$  for  $aa\ ba\ i$ 
    by (rule order-trans[OF - add-mono[of aa aa 0 ba]]) auto
  then have [simp]:  $enat\ i \leq ba \implies enat\ i \leq aa + ba$  for  $aa\ ba\ i$ 
    by (metis add.commute)
  fix  $a\ b :: enat$ 
  have  $\forall_F\ x\ in\ INF\ m\ n.\ principal\ (\{enat\ n..\} \times \{enat\ m..\}).\ enat\ i \leq fst\ x +$ 
 $snd\ x$ 
     $\forall_F\ x\ in\ INF\ n.\ principal\ (\{enat\ n..\} \times \{enat\ j\}).\ enat\ i \leq fst\ x + snd\ x$ 
     $\forall_F\ x\ in\ INF\ n.\ principal\ (\{enat\ j\} \times \{enat\ n..\}).\ enat\ i \leq fst\ x + snd\ x$ 
    for  $i\ j$ 
    by (auto intro!: eventually-INF1[of i] simp: eventually-principal)
  then show  $((\lambda x.\ fst\ x + snd\ x) \longrightarrow a + b)$  ( $nhds\ a \times_F nhds\ b$ )
    by (auto simp: nhds-enat filterlim-INF prod-filter-INF1 prod-filter-INF2
filterlim-principal principal-prod-principal eventually-principal)
  qed

```

For more lemmas about the extended real numbers see `~~/src/HOL/Analysis/Extended_Real_Limits.thy`.

38.1 Definition and basic properties

datatype *ereal* = *ereal real* | *PInfty* | *MInfty*

instantiation *ereal* :: *uminus*
begin

fun *uminus-ereal* **where**
 – (*ereal* *r*) = *ereal* (– *r*)
 | – *PInfty* = *MInfty*
 | – *MInfty* = *PInfty*

instance ..

end

instantiation *ereal* :: *infinity*
begin

definition ($\infty :: \text{ereal}$) = *PInfty*
instance ..

end

declare [[*coercion* *ereal* :: *real* \Rightarrow *ereal*]]

lemma *ereal-uminus-uminus*[*simp*]:
fixes *a* :: *ereal*
shows – (– *a*) = *a*
by (*cases* *a*) *simp-all*

lemma
shows *PInfty-eq-infinity*[*simp*]: *PInfty* = ∞
and *MInfty-eq-minfinity*[*simp*]: *MInfty* = $-\infty$
and *MInfty-neq-PInfty*[*simp*]: $\infty \neq -(\infty :: \text{ereal})$ $-\infty \neq (\infty :: \text{ereal})$
and *MInfty-neq-ereal*[*simp*]: *ereal* *r* $\neq -\infty$ $-\infty \neq \text{ereal } r$
and *PInfty-neq-ereal*[*simp*]: *ereal* *r* $\neq \infty$ $\infty \neq \text{ereal } r$
and *PInfty-cases*[*simp*]: (*case* ∞ of *ereal* *r* \Rightarrow *f* *r* | *PInfty* \Rightarrow *y* | *MInfty* \Rightarrow *z*)
 = *y*
and *MInfty-cases*[*simp*]: (*case* $-\infty$ of *ereal* *r* \Rightarrow *f* *r* | *PInfty* \Rightarrow *y* | *MInfty* \Rightarrow *z*) = *z*
by (*simp-all* *add*: *infinity-ereal-def*)

declare

PInfty-eq-infinity[*code-post*]
MInfty-eq-minfinity[*code-post*]

lemma [*code-unfold*]:
 $\infty = \text{PInfty}$
 $-\text{PInfty} = \text{MInfty}$

by *simp-all*

lemma *inj-ereal[simp]*: *inj-on ereal A*
unfolding *inj-on-def* **by** *auto*

lemma *ereal-cases[cases type: ereal]*:
obtains (*real*) *r* **where** *x = ereal r*
| (*PInf*) *x = ∞*
| (*MInf*) *x = -∞*
by (*cases x*) *auto*

lemmas *ereal2-cases = ereal-cases[case-product ereal-cases]*
lemmas *ereal3-cases = ereal2-cases[case-product ereal-cases]*

lemma *ereal-all-split*: $\bigwedge P. (\forall x::ereal. P\ x) \longleftrightarrow P\ \infty \wedge (\forall x. P\ (ereal\ x)) \wedge P\ (-\infty)$
by (*metis ereal-cases*)

lemma *ereal-ex-split*: $\bigwedge P. (\exists x::ereal. P\ x) \longleftrightarrow P\ \infty \vee (\exists x. P\ (ereal\ x)) \vee P\ (-\infty)$
by (*metis ereal-cases*)

lemma *ereal-uminus-eq-iff[simp]*:
fixes *a b :: ereal*
shows $-a = -b \longleftrightarrow a = b$
by (*cases rule: ereal2-cases[of a b]*) *simp-all*

function *real-of-ereal* :: *ereal* \Rightarrow *real* **where**
real-of-ereal (*ereal r*) = *r*
| *real-of-ereal* ∞ = 0
| *real-of-ereal* $(-\infty)$ = 0
by (*auto intro: ereal-cases*)
termination **by** *standard* (*rule wf-on-bot*)

lemma *real-of-ereal[simp]*:
real-of-ereal ($-x :: ereal$) = $-(real-of-ereal\ x)$
by (*cases x*) *simp-all*

lemma *range-ereal[simp]*: *range ereal* = *UNIV* - $\{\infty, -\infty\}$
proof *safe*
fix *x*
assume $x \notin range\ ereal\ x \neq \infty$
then show $x = -\infty$
by (*cases x*) *auto*
qed *auto*

lemma *ereal-range-uminus[simp]*: *range uminus* = (*UNIV::ereal set*)
proof *safe*
fix *x :: ereal*

```

show  $x \in \text{range } \text{uminus}$ 
  by (intro image-eqI[of - - -x]) auto
qed auto

```

```

instantiation ereal :: abs
begin

```

```

function abs-ereal where
   $|\text{ereal } r| = \text{ereal } |r|$ 
   $|- \infty| = (\infty :: \text{ereal})$ 
   $|\infty| = (\infty :: \text{ereal})$ 
by (auto intro: ereal-cases)
termination proof qed (rule wf-on-bot)

```

```

instance ..

```

```

end

```

```

lemma abs-eq-infinity-cases[elim!]:
  fixes  $x :: \text{ereal}$ 
  assumes  $|x| = \infty$ 
  obtains  $x = \infty \mid x = -\infty$ 
  using assms by (cases x) auto

```

```

lemma abs-neq-infinity-cases[elim!]:
  fixes  $x :: \text{ereal}$ 
  assumes  $|x| \neq \infty$ 
  obtains  $r$  where  $x = \text{ereal } r$ 
  using assms by (cases x) auto

```

```

lemma abs-ereal-uminus[simp]:
  fixes  $x :: \text{ereal}$ 
  shows  $|- x| = |x|$ 
  by (cases x) auto

```

```

lemma ereal-infinity-cases:
  fixes  $a :: \text{ereal}$ 
  shows  $a \neq \infty \implies a \neq -\infty \implies |a| \neq \infty$ 
  by auto

```

38.1.1 Addition

```

instantiation ereal :: {one,comm-monoid-add,zero-neq-one}
begin

```

```

definition  $0 = \text{ereal } 0$ 

```

```

definition  $1 = \text{ereal } 1$ 

```

```

function plus-ereal where

```

```

    ereal r + ereal p = ereal (r + p)
|  $\infty + a = (\infty :: \text{ereal})$ 
|  $a + \infty = (\infty :: \text{ereal})$ 
|  $\text{ereal } r + -\infty = -\infty$ 
|  $-\infty + \text{ereal } p = -(\infty :: \text{ereal})$ 
|  $-\infty + -\infty = -(\infty :: \text{ereal})$ 
proof goal-cases
  case prems: (1 P x)
  then obtain a b where x = (a, b)
    by (cases x) auto
  with prems show P
    by (cases rule: ereal2-cases[of a b]) auto
qed auto
termination by standard (rule wf-on-bot)

```

```

lemma Infty-neq-0[simp]:
   $(\infty :: \text{ereal}) \neq 0$   $0 \neq (\infty :: \text{ereal})$ 
   $-(\infty :: \text{ereal}) \neq 0$   $0 \neq -(\infty :: \text{ereal})$ 
  by (simp-all add: zero-ereal-def)

```

```

lemma ereal-eq-0[simp]:
   $\text{ereal } r = 0 \longleftrightarrow r = 0$ 
   $0 = \text{ereal } r \longleftrightarrow r = 0$ 
  unfolding zero-ereal-def by simp-all

```

```

lemma ereal-eq-1[simp]:
   $\text{ereal } r = 1 \longleftrightarrow r = 1$ 
   $1 = \text{ereal } r \longleftrightarrow r = 1$ 
  unfolding one-ereal-def by simp-all

```

instance

```

proof
  fix a b c :: ereal
  show 0 + a = a
    by (cases a) (simp-all add: zero-ereal-def)
  show a + b = b + a
    by (cases rule: ereal2-cases[of a b]) simp-all
  show a + b + c = a + (b + c)
    by (cases rule: ereal3-cases[of a b c]) simp-all
  show 0  $\neq$  (1 :: ereal)
    by (simp add: one-ereal-def zero-ereal-def)
qed

```

end

```

lemma ereal-0-plus [simp]:  $\text{ereal } 0 + x = x$ 
and plus-ereal-0 [simp]:  $x + \text{ereal } 0 = x$ 
  by (simp-all flip: zero-ereal-def)

```

instance *ereal* :: *numeral* ..

lemma *real-of-ereal-0*[*simp*]: *real-of-ereal* (*0*::*ereal*) = *0*
unfolding *zero-ereal-def* **by** *simp*

lemma *abs-ereal-zero*[*simp*]: $|0| = (0::ereal)$
unfolding *zero-ereal-def* *abs-ereal.simps* **by** *simp*

lemma *ereal-uminus-zero*[*simp*]: $- 0 = (0::ereal)$
by (*simp add: zero-ereal-def*)

lemma *ereal-uminus-zero-iff*[*simp*]:
fixes *a* :: *ereal*
shows $-a = 0 \longleftrightarrow a = 0$
by (*cases a*) *simp-all*

lemma *ereal-plus-eq-PIfty*[*simp*]:
fixes *a b* :: *ereal*
shows $a + b = \infty \longleftrightarrow a = \infty \vee b = \infty$
by (*cases rule: ereal2-cases*[*of a b*]) *auto*

lemma *ereal-plus-eq-MIfty*[*simp*]:
fixes *a b* :: *ereal*
shows $a + b = -\infty \longleftrightarrow (a = -\infty \vee b = -\infty) \wedge a \neq \infty \wedge b \neq \infty$
by (*cases rule: ereal2-cases*[*of a b*]) *auto*

lemma *ereal-add-cancel-left*:
fixes *a b* :: *ereal*
assumes $a \neq -\infty$
shows $a + b = a + c \longleftrightarrow a = \infty \vee b = c$
using *assms* **by** (*cases rule: ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-add-cancel-right*:
fixes *a b* :: *ereal*
assumes $a \neq -\infty$
shows $b + a = c + a \longleftrightarrow a = \infty \vee b = c$
using *assms* **by** (*cases rule: ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-real*: *ereal* (*real-of-ereal* *x*) = (*if* $|x| = \infty$ *then* *0* *else* *x*)
by *auto*

lemma *real-of-ereal-add*:
fixes *a b* :: *ereal*
shows *real-of-ereal* (*a* + *b*) =
 (*if* ($|a| = \infty$) \wedge ($|b| = \infty$) \vee ($|a| \neq \infty$) \wedge ($|b| \neq \infty$) *then* *real-of-ereal* *a* +
real-of-ereal *b* *else* *0*)
by *auto*

38.1.2 Linear order on ereal

instantiation *ereal* :: *linorder*
begin

function *less-ereal*

where

$ereal\ x < ereal\ y \iff x < y$
 $(\infty::ereal) < a \iff False$
 $a < -(\infty::ereal) \iff False$
 $ereal\ x < \infty \iff True$
 $-\infty < ereal\ r \iff True$
 $-\infty < (\infty::ereal) \iff True$

proof *goal-cases*

case *prems*: (1 *P x*)

then obtain *a b* **where** $x = (a, b)$ **by** (*cases x*) *auto*

with *prems* **show** *P* **by** (*cases rule: ereal2-cases[of a b]*) *auto*

qed *simp-all*

termination by (*relation {}*) *simp*

definition $x \leq (y::ereal) \iff x < y \vee x = y$

lemma *ereal-infity-less[simp]*:

fixes $x :: ereal$

shows $x < \infty \iff (x \neq \infty)$

$-\infty < x \iff (x \neq -\infty)$

by (*cases x, simp-all*)**+**

lemma *ereal-infity-less-eq[simp]*:

fixes $x :: ereal$

shows $\infty \leq x \iff x = \infty$

and $x \leq -\infty \iff x = -\infty$

by (*auto simp: less-eq-ereal-def*)

lemma *ereal-less[simp]*:

$ereal\ r < 0 \iff (r < 0)$

$0 < ereal\ r \iff (0 < r)$

$ereal\ r < 1 \iff (r < 1)$

$1 < ereal\ r \iff (1 < r)$

$0 < (\infty::ereal)$

$-(\infty::ereal) < 0$

by (*simp-all add: zero-ereal-def one-ereal-def*)

lemma *ereal-less-eq[simp]*:

$x \leq (\infty::ereal)$

$-(\infty::ereal) \leq x$

$ereal\ r \leq ereal\ p \iff r \leq p$

$ereal\ r \leq 0 \iff r \leq 0$

$0 \leq ereal\ r \iff 0 \leq r$

$ereal\ r \leq 1 \iff r \leq 1$

$1 \leq \text{ereal } r \longleftrightarrow 1 \leq r$
by (*auto simp: less-eq-ereal-def zero-ereal-def one-ereal-def*)

lemma *ereal-infity-less-eq2*:
 $a \leq b \implies a = \infty \implies b = (\infty::\text{ereal})$
 $a \leq b \implies b = -\infty \implies a = -(\infty::\text{ereal})$
by *simp-all*

instance

proof

fix $x\ y\ z :: \text{ereal}$
show $x \leq x$
by (*cases x*) *simp-all*
show $x < y \longleftrightarrow x \leq y \wedge \neg y \leq x$
by (*cases rule: ereal2-cases[of x y]*) *auto*
show $x \leq y \vee y \leq x$
by (*cases rule: ereal2-cases[of x y]*) *auto*
assume $x \leq y$
then show $y \leq x \implies x = y$
by (*cases rule: ereal2-cases[of x y]*) *auto*
show $y \leq z \implies x \leq z$
using $\langle x \leq y \rangle$
by (*cases rule: ereal3-cases[of x y z]*) *auto*

qed

end

lemma *ereal-dense2*: $x < y \implies \exists z. x < \text{ereal } z \wedge \text{ereal } z < y$
using *lt-ex gt-ex dense* **by** (*cases x y rule: ereal2-cases*) *auto*

instance *ereal* :: *dense-linorder*

by *standard (blast dest: ereal-dense2)*

instance *ereal* :: *ordered-comm-monoid-add*

proof

fix $a\ b\ c :: \text{ereal}$
assume $a \leq b$
then show $c + a \leq c + b$
by (*cases rule: ereal3-cases[of a b c]*) *auto*

qed

lemma *ereal-one-not-less-zero-ereal*[*simp*]: $\neg 1 < (0::\text{ereal})$
by (*simp add: zero-ereal-def*)

lemma *real-of-ereal-positive-mono*:

fixes $x\ y :: \text{ereal}$

shows $0 \leq x \implies x \leq y \implies y \neq \infty \implies \text{real-of-ereal } x \leq \text{real-of-ereal } y$

by (*cases rule: ereal2-cases[of x y]*) *auto*

lemma *ereal-MInfty-lessI*[*intro, simp*]:

fixes $a :: \text{ereal}$
shows $a \neq -\infty \implies -\infty < a$
by *simp*

lemma *ereal-less-PInfty*[*intro, simp*]:

fixes $a :: \text{ereal}$
shows $a \neq \infty \implies a < \infty$
by *simp*

lemma *ereal-less-ereal-Ex*:

fixes $a \ b :: \text{ereal}$
shows $x < \text{ereal } r \longleftrightarrow x = -\infty \vee (\exists p. p < r \wedge x = \text{ereal } p)$
by (*cases x*) *auto*

lemma *less-PInf-Ex-of-nat*: $x \neq \infty \longleftrightarrow (\exists n::\text{nat}. x < \text{ereal } (\text{real } n))$

proof (*cases x*)

case (*real r*)

then show *?thesis*

using *reals-Archimedean2*[*of r*] **by** *simp*

qed *simp-all*

lemma *ereal-add-strict-mono2*:

fixes $a \ b \ c \ d :: \text{ereal}$
assumes $a < b$ **and** $c < d$
shows $a + c < b + d$
using *assms*
by (*cases a*; *force simp: elim: less-ereal.elims*)

lemma *ereal-minus-le-minus*[*simp*]:

fixes $a \ b :: \text{ereal}$
shows $-a \leq -b \longleftrightarrow b \leq a$
by (*cases rule: ereal2-cases*[*of a b*]) *auto*

lemma *ereal-minus-less-minus*[*simp*]:

fixes $a \ b :: \text{ereal}$
shows $-a < -b \longleftrightarrow b < a$
by (*cases rule: ereal2-cases*[*of a b*]) *auto*

lemma *ereal-le-real-iff*:

$x \leq \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$
by (*cases y*) *auto*

lemma *real-le-ereal-iff*:

$\text{real-of-ereal } y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$
by (*cases y*) *auto*

lemma *ereal-less-real-iff*:

$x < \text{real-of-ereal } y \longleftrightarrow (|y| \neq \infty \longrightarrow \text{ereal } x < y) \wedge (|y| = \infty \longrightarrow x < 0)$

by (*cases y*) *auto*

lemma *real-less-ereal-iff*:

real-of-ereal $y < x \longleftrightarrow (|y| \neq \infty \longrightarrow y < \text{ereal } x) \wedge (|y| = \infty \longrightarrow 0 < x)$

by (*cases y*) *auto*

To help with inferences like $\llbracket a < \text{ereal } x; x < y \rrbracket \Longrightarrow a < \text{ereal } y$, where x and y are real.

lemma *le-ereal-le*: $a \leq \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a \leq \text{ereal } y$

using *ereal-less-eq(3)* *order.trans* **by** *blast*

lemma *le-ereal-less*: $a \leq \text{ereal } x \Longrightarrow x < y \Longrightarrow a < \text{ereal } y$

by (*simp add: le-less-trans*)

lemma *less-ereal-le*: $a < \text{ereal } x \Longrightarrow x \leq y \Longrightarrow a < \text{ereal } y$

using *ereal-less-ereal-Ex* **by** *auto*

lemma *ereal-le-le*: $\text{ereal } y \leq a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x \leq a$

by (*simp add: order-subst2*)

lemma *ereal-le-less*: $\text{ereal } y \leq a \Longrightarrow x < y \Longrightarrow \text{ereal } x < a$

by (*simp add: dual-order.strict-trans1*)

lemma *ereal-less-le*: $\text{ereal } y < a \Longrightarrow x \leq y \Longrightarrow \text{ereal } x < a$

using *ereal-less-eq(3)* *le-less-trans* **by** *blast*

lemma *real-of-ereal-pos*:

fixes $x :: \text{ereal}$

shows $0 \leq x \Longrightarrow 0 \leq \text{real-of-ereal } x$

by (*cases x*) *auto*

lemmas *real-of-ereal-ord-simps* =

ereal-le-real-iff *real-le-ereal-iff* *ereal-less-real-iff* *real-less-ereal-iff*

lemma *abs-ereal-ge0[simp]*: $0 \leq x \Longrightarrow |x :: \text{ereal}| = x$

by (*cases x*) *auto*

lemma *abs-ereal-less0[simp]*: $x < 0 \Longrightarrow |x :: \text{ereal}| = -x$

by (*cases x*) *auto*

lemma *abs-ereal-pos[simp]*: $0 \leq |x :: \text{ereal}|$

by (*cases x*) *auto*

lemma *ereal-abs-leI*:

fixes $x y :: \text{ereal}$

shows $\llbracket x \leq y; -x \leq y \rrbracket \Longrightarrow |x| \leq y$

by(*cases x y rule: ereal2-cases*)(*simp-all*)

lemma *ereal-abs-add*:


```

fixes  $a b :: ereal$ 
shows  $abs(a+b) \leq abs\ a + abs\ b$ 
by (cases rule: ereal2-cases[of  $a\ b$ ]) (auto)

lemma real-of-ereal-le-0[simp]:  $real-of-ereal\ (x :: ereal) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$ 
by (cases  $x$ ) auto

lemma abs-real-of-ereal[simp]:  $|real-of-ereal\ (x :: ereal)| = real-of-ereal\ |x|$ 
by (cases  $x$ ) auto

lemma zero-less-real-of-ereal:
  fixes  $x :: ereal$ 
  shows  $0 < real-of-ereal\ x \longleftrightarrow 0 < x \wedge x \neq \infty$ 
  by (cases  $x$ ) auto

lemma ereal-0-le-uminus-iff[simp]:
  fixes  $a :: ereal$ 
  shows  $0 \leq -a \longleftrightarrow a \leq 0$ 
  by (cases rule: ereal2-cases[of  $a$ ]) auto

lemma ereal-uminus-le-0-iff[simp]:
  fixes  $a :: ereal$ 
  shows  $-a \leq 0 \longleftrightarrow 0 \leq a$ 
  by (cases rule: ereal2-cases[of  $a$ ]) auto

lemma ereal-add-strict-mono:
  fixes  $a\ b\ c\ d :: ereal$ 
  assumes  $a \leq b$ 
  and  $0 \leq a$ 
  and  $a \neq \infty$ 
  and  $c < d$ 
  shows  $a + c < b + d$ 
  using assms
  by (cases rule: ereal3-cases[case-product ereal-cases, of  $a\ b\ c\ d$ ]) auto

lemma ereal-less-add:
  fixes  $a\ b\ c :: ereal$ 
  shows  $|a| \neq \infty \implies c < b \implies a + c < a + b$ 
  by (cases rule: ereal2-cases[of  $b\ c$ ]) auto

lemma ereal-uminus-eq-reorder:  $-a = b \longleftrightarrow a = (-b :: ereal)$ 
by auto

lemma ereal-uminus-less-reorder:  $-a < b \longleftrightarrow -b < a$ 
and ereal-less-uminus-reorder:  $a < -b \longleftrightarrow b < -a$ 
and ereal-uminus-le-reorder:  $-a \leq b \longleftrightarrow -b \leq a$  for  $a :: ereal$ 
using ereal-minus-le-minus ereal-minus-less-minus by fastforce+

lemmas ereal-uminus-reorder =

```

ereal-uminus-eq-reorder eréal-uminus-less-reorder eréal-uminus-le-reorder

```

lemma ereal-bot:
  fixes  $x :: \text{ereal}$ 
  assumes  $\bigwedge B. x \leq \text{ereal } B$ 
  shows  $x = -\infty$ 
proof (cases  $x$ )
  case (real  $r$ )
    with assms[of  $r - 1$ ] show ?thesis
    by auto
  next
    case PInf
    with assms[of  $0$ ] show ?thesis
    by auto
  next
    case MInf
    then show ?thesis
    by simp
qed

```

```

lemma ereal-top:
  fixes  $x :: \text{ereal}$ 
  assumes  $\bigwedge B. x \geq \text{ereal } B$ 
  shows  $x = \infty$ 
proof (cases  $x$ )
  case (real  $r$ )
    with assms[of  $r + 1$ ] show ?thesis
    by auto
  next
    case MInf
    with assms[of  $0$ ] show ?thesis
    by auto
  next
    case PInf
    then show ?thesis
    by simp
qed

```

```

lemma
  shows ereal-max[simp]:  $\text{ereal } (\max x y) = \max (\text{ereal } x) (\text{ereal } y)$ 
  and ereal-min[simp]:  $\text{ereal } (\min x y) = \min (\text{ereal } x) (\text{ereal } y)$ 
  by (simp-all add: min-def max-def)

```

```

lemma ereal-max-0:  $\max 0 (\text{ereal } r) = \text{ereal } (\max 0 r)$ 
  by (auto simp: zero-ereal-def)

```

```

lemma
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  shows ereal-incseq-uminus[simp]:  $\text{incseq } (\lambda x. - f x) \longleftrightarrow \text{decseq } f$ 

```

```

and ereal-decseq-uminus[simp]:  $\text{decseq } (\lambda x. - f x) \longleftrightarrow \text{incseq } f$ 
unfolding decseq-def incseq-def by auto

lemma incseq-ereal:  $\text{incseq } f \implies \text{incseq } (\lambda x. \text{ereal } (f x))$ 
unfolding incseq-def by auto

lemma sum-ereal[simp]:  $(\sum_{x \in A}. \text{ereal } (f x)) = \text{ereal } (\sum_{x \in A}. f x)$ 
by (induction A rule: infinite-finite-induct) auto

lemma sum-list-ereal [simp]:  $\text{sum-list } (\text{map } (\lambda x. \text{ereal } (f x)) \text{ xs}) = \text{ereal } (\text{sum-list } (\text{map } f \text{ xs}))$ 
by (induction xs) simp-all

lemma sum-Pinf:
  fixes  $f :: 'a \Rightarrow \text{ereal}$ 
  shows  $(\sum_{x \in P}. f x) = \infty \longleftrightarrow \text{finite } P \wedge (\exists i \in P. f i = \infty)$ 
proof safe
  assume *:  $\text{sum } f P = \infty$ 
  show finite P
    by (metis * Infty-neg-0(2) sum.infinite)
  show  $\exists i \in P. f i = \infty$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then have  $\bigwedge i. i \in P \implies f i \neq \infty$ 
    by auto
    with  $\langle \text{finite } P \rangle$  have  $\text{sum } f P \neq \infty$ 
    by induct auto
    with * show False
    by auto
  qed
next
  fix  $i$ 
  assume finite P and i ∈ P and f i = ∞
  then show  $\text{sum } f P = \infty$ 
  proof induct
    case (insert x A)
    show ?case using insert by (cases x = i) auto
  qed simp
qed

lemma sum-Inf:
  fixes  $f :: 'a \Rightarrow \text{ereal}$ 
  shows  $|\text{sum } f A| = \infty \longleftrightarrow \text{finite } A \wedge (\exists i \in A. |f i| = \infty)$ 
proof
  assume *:  $|\text{sum } f A| = \infty$ 
  have finite A
    by (rule ccontr) (insert *, auto)
  moreover have  $\exists i \in A. |f i| = \infty$ 
  proof (rule ccontr)

```

```

    assume  $\neg ?thesis$ 
    then have  $\forall i \in A. \exists r. f\ i = ereal\ r$ 
      by auto
    then obtain  $r$  where  $\forall x \in A. f\ x = ereal\ (r\ x)$ 
      by metis
    with * show False
      by auto
  qed
  ultimately show  $finite\ A \wedge (\exists i \in A. |f\ i| = \infty)$ 
    by auto
next
  assume  $finite\ A \wedge (\exists i \in A. |f\ i| = \infty)$ 
  then obtain  $i$  where  $finite\ A\ i \in A$  and  $|f\ i| = \infty$ 
    by auto
  then show  $|sum\ f\ A| = \infty$ 
  proof induct
    case (insert  $j\ A$ )
    then show ?case
      by (cases rule: ereal3-cases[of  $f\ i\ f\ j\ sum\ f\ A$ ]) auto
  qed simp
qed

```

lemma *sum-real-of-ereal*:

```

  fixes  $f :: 'i \Rightarrow ereal$ 
  assumes  $\bigwedge x. x \in S \implies |f\ x| \neq \infty$ 
  shows  $(\sum x \in S. real-of-ereal\ (f\ x)) = real-of-ereal\ (sum\ f\ S)$ 
  proof -
    have  $\forall x \in S. \exists r. f\ x = ereal\ r$ 
      using assms by blast
    then obtain  $r$  where  $\forall x \in S. f\ x = ereal\ (r\ x)$ 
      by metis
    then show ?thesis
      by simp
  qed

```

38.1.3 Multiplication

instantiation *ereal* :: {comm-monoid-mult,sgn}
begin

function *sgn-ereal* :: *ereal* \Rightarrow *ereal* **where**
 $sgn\ (ereal\ r) = ereal\ (sgn\ r)$
 $|sgn\ (\infty::ereal) = 1$
 $|sgn\ (-\infty::ereal) = -1$
by (auto intro: ereal-cases)
termination **by** standard (rule wf-on-bot)

function *times-ereal* **where**
 $ereal\ r * ereal\ p = ereal\ (r * p)$

```

| ereal r * ∞ = (if r = 0 then 0 else if r > 0 then ∞ else -∞)
| ∞ * ereal r = (if r = 0 then 0 else if r > 0 then ∞ else -∞)
| ereal r * -∞ = (if r = 0 then 0 else if r > 0 then -∞ else ∞)
| -∞ * ereal r = (if r = 0 then 0 else if r > 0 then -∞ else ∞)
| (∞::ereal) * ∞ = ∞
| -(∞::ereal) * ∞ = -∞
| (∞::ereal) * -∞ = -∞
| -(∞::ereal) * -∞ = ∞

```

proof *goal-cases*

```

  case prems: (1 P x)
  then obtain a b where x = (a, b)
    by (cases x) auto
  with prems show P
    by (cases rule: ereal2-cases[of a b]) auto

```

qed *simp-all*

termination by (relation {}) *simp*

instance

proof

```

  fix a b c :: ereal
  show 1 * a = a
    by (cases a) (simp-all add: one-ereal-def)
  show a * b = b * a
    by (cases rule: ereal2-cases[of a b]) simp-all
  show a * b * c = a * (b * c)
    by (cases rule: ereal3-cases[of a b c])
      (simp-all add: zero-ereal-def zero-less-mult-iff)

```

qed

end

lemma [*simp*]:

```

  shows ereal-1-times: ereal 1 * x = x
  and times-ereal-1: x * ereal 1 = x
  by (simp-all flip: one-ereal-def)

```

lemma *one-not-le-zero-ereal*[*simp*]: $\neg (1 \leq (0::ereal))$

by (*simp* add: *one-ereal-def* *zero-ereal-def*)

lemma *real-ereal-1*[*simp*]: *real-of-ereal* (1::ereal) = 1

unfolding *one-ereal-def* by *simp*

lemma *real-of-ereal-le-1*:

```

  fixes a :: ereal
  shows a ≤ 1 ⇒ real-of-ereal a ≤ 1
  by (cases a) (auto simp: one-ereal-def)

```

lemma *abs-ereal-one*[*simp*]: $|1| = (1::ereal)$

unfolding *one-ereal-def* by *simp*

```

lemma ereal-mult-zero[simp]:
  fixes a :: ereal
  shows a * 0 = 0
  by (cases a) (simp-all add: zero-ereal-def)

lemma ereal-zero-mult[simp]:
  fixes a :: ereal
  shows 0 * a = 0
  by (metis ereal-mult-zero mult.commute)

lemma ereal-m1-less-0[simp]:  $-(1::\text{ereal}) < 0$ 
  by (simp add: zero-ereal-def one-ereal-def)

lemma ereal-times[simp]:
   $1 \neq (\infty::\text{ereal})$   $(\infty::\text{ereal}) \neq 1$ 
   $1 \neq -(\infty::\text{ereal})$   $-(\infty::\text{ereal}) \neq 1$ 
  by (auto simp: one-ereal-def)

lemma ereal-plus-1[simp]:
   $1 + \text{ereal } r = \text{ereal } (r + 1)$ 
   $\text{ereal } r + 1 = \text{ereal } (r + 1)$ 
   $1 + -(\infty::\text{ereal}) = -\infty$ 
   $-(\infty::\text{ereal}) + 1 = -\infty$ 
  unfolding one-ereal-def by auto

lemma ereal-zero-times[simp]:
  fixes a b :: ereal
  shows  $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$ 
  by (cases rule: ereal2-cases[of a b]) auto

lemma ereal-mult-eq-PInfty[simp]:
   $a * b = (\infty::\text{ereal}) \longleftrightarrow$ 
   $(a = \infty \wedge b > 0) \vee (a > 0 \wedge b = \infty) \vee (a = -\infty \wedge b < 0) \vee (a < 0 \wedge b =$ 
   $-\infty)$ 
  by (cases rule: ereal2-cases[of a b]) auto

lemma ereal-mult-eq-MInfty[simp]:
   $a * b = -(\infty::\text{ereal}) \longleftrightarrow$ 
   $(a = \infty \wedge b < 0) \vee (a < 0 \wedge b = \infty) \vee (a = -\infty \wedge b > 0) \vee (a > 0 \wedge b =$ 
   $-\infty)$ 
  by (cases rule: ereal2-cases[of a b]) auto

lemma ereal-abs-mult:  $|x * y :: \text{ereal}| = |x| * |y|$ 
  by (cases x y rule: ereal2-cases) (auto simp: abs-mult)

lemma ereal-0-less-1[simp]:  $0 < (1::\text{ereal})$ 
  by (simp add: zero-ereal-def one-ereal-def)

```

```

lemma ereal-mult-minus-left[simp]:
  fixes a b :: ereal
  shows  $-a * b = -(a * b)$ 
  by (cases rule: ereal2-cases[of a b]) auto

lemma ereal-mult-minus-right[simp]:
  fixes a b :: ereal
  shows  $a * -b = -(a * b)$ 
  by (cases rule: ereal2-cases[of a b]) auto

lemma ereal-mult-infy[simp]:
   $a * (\infty::ereal) = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$ 
  by (cases a) auto

lemma ereal-infy-mult[simp]:
   $(\infty::ereal) * a = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } \infty \text{ else } -\infty)$ 
  by (cases a) auto

lemma ereal-mult-strict-right-mono:
  assumes  $a < b$ 
  and  $0 < c$ 
  and  $c < (\infty::ereal)$ 
  shows  $a * c < b * c$ 
  using assms
  by (cases rule: ereal3-cases[of a b c]) (auto simp: zero-le-mult-iff)

lemma ereal-mult-strict-left-mono:
   $a < b \implies 0 < c \implies c < (\infty::ereal) \implies c * a < c * b$ 
  using ereal-mult-strict-right-mono
  by (simp add: mult.commute[of c])

lemma ereal-mult-right-mono:
  fixes a b c :: ereal
  assumes  $a \leq b$   $0 \leq c$ 
  shows  $a * c \leq b * c$ 
proof (cases c = 0)
  case False
  with assms show ?thesis
  by (cases rule: ereal3-cases[of a b c]) auto
qed auto

lemma ereal-mult-left-mono:
  fixes a b c :: ereal
  shows  $a \leq b \implies 0 \leq c \implies c * a \leq c * b$ 
  by (simp add: ereal-mult-right-mono mult.commute)

lemma ereal-mult-mono:
  fixes a b c d :: ereal
  assumes  $b \geq 0$   $c \geq 0$   $a \leq b$   $c \leq d$ 

```

shows $a * c \leq b * d$
by (*metis ereal-mult-right-mono mult.commute order-trans assms*)

lemma *ereal-mult-mono'*:
fixes $a\ b\ c\ d::ereal$
assumes $a \geq 0\ c \geq 0\ a \leq b\ c \leq d$
shows $a * c \leq b * d$
by (*metis ereal-mult-right-mono mult.commute order-trans assms*)

lemma *ereal-mult-mono-strict*:
fixes $a\ b\ c\ d::ereal$
assumes $b > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
proof –
have $c < \infty$ **using** $\langle c < d \rangle$
by *auto*
then have $a * c < b * c$
by (*metis ereal-mult-strict-left-mono[OF assms(3) assms(2)] mult.commute*)
moreover have $b * c \leq b * d$
using *assms(1,4) ereal-mult-left-mono* **by** *force*
ultimately show *?thesis* **by** *simp*
qed

lemma *ereal-mult-mono-strict'*:
fixes $a\ b\ c\ d::ereal$
assumes $a > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
using *assms ereal-mult-mono-strict* **by** *auto*

lemma *zero-less-one-ereal[simp]*: $0 \leq (1::ereal)$
by (*simp add: one-ereal-def zero-ereal-def*)

lemma *ereal-0-le-mult[simp]*: $0 \leq a \implies 0 \leq b \implies 0 \leq a * (b::ereal)$
by (*cases rule: ereal2-cases[of a b]*) *auto*

lemma *ereal-right-distrib*:
fixes $r\ a\ b::ereal$
shows $0 \leq a \implies 0 \leq b \implies r * (a + b) = r * a + r * b$
by (*cases rule: ereal3-cases[of r a b]*) (*simp-all add: field-simps*)

lemma *ereal-left-distrib*:
fixes $r\ a\ b::ereal$
shows $0 \leq a \implies 0 \leq b \implies (a + b) * r = a * r + b * r$
by (*cases rule: ereal3-cases[of r a b]*) (*simp-all add: field-simps*)

lemma *ereal-mult-le-0-iff*:
fixes $a\ b::ereal$
shows $a * b \leq 0 \iff (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$
by (*cases rule: ereal2-cases[of a b]*) (*simp-all add: mult-le-0-iff*)

lemma *ereal-zero-le-0-iff*:

fixes $a\ b :: \text{ereal}$

shows $0 \leq a * b \longleftrightarrow (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$

by (*cases rule: ereal2-cases[of a b]*) (*simp-all add: zero-le-mult-iff*)

lemma *ereal-mult-less-0-iff*:

fixes $a\ b :: \text{ereal}$

shows $a * b < 0 \longleftrightarrow (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$

by (*cases rule: ereal2-cases[of a b]*) (*simp-all add: mult-less-0-iff*)

lemma *ereal-zero-less-0-iff*:

fixes $a\ b :: \text{ereal}$

shows $0 < a * b \longleftrightarrow (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$

by (*cases rule: ereal2-cases[of a b]*) (*simp-all add: zero-less-mult-iff*)

lemma *ereal-left-mult-cong*:

fixes $a\ b\ c :: \text{ereal}$

shows $c = d \implies (d \neq 0 \implies a = b) \implies a * c = b * d$

by (*cases c = 0*) *simp-all*

lemma *ereal-right-mult-cong*:

fixes $a\ b\ c :: \text{ereal}$

shows $c = d \implies (d \neq 0 \implies a = b) \implies c * a = d * b$

by (*cases c = 0*) *simp-all*

lemma *ereal-distrib*:

fixes $a\ b\ c :: \text{ereal}$

assumes $a \neq \infty \vee b \neq -\infty$

and $a \neq -\infty \vee b \neq \infty$

and $|c| \neq \infty$

shows $(a + b) * c = a * c + b * c$

using *assms*

by (*cases rule: ereal3-cases[of a b c]*) (*simp-all add: field-simps*)

lemma *numeral-eq-ereal* [*simp*]: *numeral w = ereal (numeral w)*

proof (*induct w rule: num-induct*)

case *One*

then show *?case*

by *simp*

next

case (*inc x*)

then show *?case*

by (*simp add: inc numeral-inc*)

qed

lemma *m1-ereal-less-iff* [*simp*]:

$((-1::\text{ereal}) < \text{numeral } a) \longleftrightarrow ((-1::\text{real}) < \text{numeral } a)$

by (*simp add: one-ereal-def*)

lemma *m1-ereal-le-iff* [simp]:
 $((-1::ereal) \leq numeral\ a) \longleftrightarrow ((-1::real) \leq numeral\ a)$
by (simp add: one-ereal-def)

lemma *m1-ereal-eq-iff* [simp]:
 $((-1::ereal) = numeral\ a) \longleftrightarrow ((-1::real) = numeral\ a)$
by (simp add: one-ereal-def)

lemma *ereal-less-m1-iff* [simp]:
 $(numeral\ a < (-1::ereal)) \longleftrightarrow (numeral\ a < (-1::real))$
by (simp add: one-ereal-def)

lemma *ereal-le-m1-iff* [simp]:
 $(numeral\ a \leq (-1::ereal)) \longleftrightarrow (numeral\ a \leq (-1::real))$
by (simp add: one-ereal-def)

lemma *ereal-eq-m1-iff* [simp]:
 $(numeral\ a = (-1::ereal)) \longleftrightarrow (numeral\ a = (-1::real))$
by (simp add: one-ereal-def)

lemma *distrib-left-ereal-nn*:
 $c \geq 0 \implies (x + y) * ereal\ c = x * ereal\ c + y * ereal\ c$
by(cases x y rule: ereal2-cases)(simp-all add: ring-distrib)

lemma *sum-ereal-right-distrib*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\bigwedge i. i \in A \implies 0 \leq f\ i) \implies r * \text{sum } f\ A = (\sum n \in A. r * f\ n)$
by (induct A rule: infinite-finite-induct) (auto simp: ereal-right-distrib sum-nonneg)

lemma *sum-ereal-left-distrib*:
 $(\bigwedge i. i \in A \implies 0 \leq f\ i) \implies \text{sum } f\ A * r = (\sum n \in A. f\ n * r :: ereal)$
using *sum-ereal-right-distrib*[of A f r] **by** (simp add: mult-ac)

lemma *sum-distrib-right-ereal*:
 $c \geq 0 \implies \text{sum } f\ A * ereal\ c = (\sum x \in A. f\ x * c :: ereal)$
by(subst sum-comp-morphism[where h= $\lambda x. x * ereal\ c$, symmetric])(simp-all add: distrib-left-ereal-nn)

lemma *ereal-le-epsilon*:
fixes $x\ y :: ereal$
assumes $\bigwedge e. 0 < e \implies x \leq y + e$
shows $x \leq y$
proof (cases $x = -\infty \vee x = \infty \vee y = -\infty \vee y = \infty$)
case True
then show ?thesis
using *assms*[of 1] **by** auto
next
case False

then obtain $p\ q$ where $x = \text{ereal } p\ y = \text{ereal } q$
 by (metis *MInfty-eq-minfinity ereal.distinct(3) uminus-ereal.elims*)
 then show ?thesis
 by (metis *assms field-le-epsilon ereal-less(2) ereal-less-eq(3) plus-ereal.simps(1)*)
 qed

lemma *ereal-le-epsilon2*:
 fixes $x\ y :: \text{ereal}$
 assumes $\bigwedge e :: \text{real}. 0 < e \implies x \leq y + \text{ereal } e$
 shows $x \leq y$
proof (rule *ereal-le-epsilon*)
 show $\bigwedge \varepsilon :: \text{real}. 0 < \varepsilon \implies x \leq y + \varepsilon$
 using *assms less-ereal.elims(2) zero-less-real-of-ereal* by *fastforce*
 qed

lemma *ereal-le-real*:
 fixes $x\ y :: \text{ereal}$
 assumes $\bigwedge z. x \leq \text{ereal } z \implies y \leq \text{ereal } z$
 shows $y \leq x$
 by (metis *assms ereal-bot ereal-cases ereal-infty-less-eq(2) ereal-less-eq(1) linorder-le-cases*)

lemma *prod-ereal-0*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 shows $(\prod i \in A. f\ i) = 0 \longleftrightarrow \text{finite } A \wedge (\exists i \in A. f\ i = 0)$
 by (induction A rule: *infinite-finite-induct*) auto

lemma *prod-ereal-pos*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 assumes $\bigwedge i. i \in I \implies 0 \leq f\ i$
 shows $0 \leq (\prod i \in I. f\ i)$
 using *assms*
 by (induction I rule: *infinite-finite-induct*) auto

lemma *prod-PInf*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 assumes $\bigwedge i. i \in I \implies 0 \leq f\ i$
 shows $(\prod i \in I. f\ i) = \infty \longleftrightarrow \text{finite } I \wedge (\exists i \in I. f\ i = \infty) \wedge (\forall i \in I. f\ i \neq 0)$
 using *assms*
proof (induction I rule: *infinite-finite-induct*)
 case (insert $i\ I$)
 then have *pos*: $0 \leq f\ i \leq \text{prod } f\ I$
 by (auto intro!: *prod-ereal-pos*)
 from insert have $(\prod j \in \text{insert } i\ I. f\ j) = \infty \longleftrightarrow \text{prod } f\ I * f\ i = \infty$
 by auto
 also have $\dots \longleftrightarrow (\text{prod } f\ I = \infty \vee f\ i = \infty) \wedge f\ i \neq 0 \wedge \text{prod } f\ I \neq 0$
 using *prod-ereal-pos[of I f] pos*
 by (cases rule: *ereal2-cases[of f i prod f I]*) auto
 also have $\dots \longleftrightarrow \text{finite } (\text{insert } i\ I) \wedge (\exists j \in \text{insert } i\ I. f\ j = \infty) \wedge (\forall j \in \text{insert } i\ I. f\ j \neq 0)$

using insert by (auto simp: prod-ereal-0)
 finally show ?case .
 qed auto

lemma prod-ereal: $(\prod_{i \in A} \text{ereal } (f i)) = \text{ereal } (\text{prod } f A)$
 by (induction A rule: infinite-finite-induct) (auto simp: one-ereal-def)

38.1.4 Power

lemma ereal-power[simp]: $(\text{ereal } x) ^ n = \text{ereal } (x ^ n)$
 by (induct n) (auto simp: one-ereal-def)

lemma ereal-power-PInf[simp]: $(\infty :: \text{ereal}) ^ n = (\text{if } n = 0 \text{ then } 1 \text{ else } \infty)$
 by (induct n) (auto simp: one-ereal-def)

lemma ereal-power-uminus[simp]:
 fixes $x :: \text{ereal}$
 shows $(- x) ^ n = (\text{if even } n \text{ then } x ^ n \text{ else } - (x ^ n))$
 by (induct n) (auto simp: one-ereal-def)

lemma ereal-power-numeral[simp]:
 $(\text{numeral } num :: \text{ereal}) ^ n = \text{ereal } (\text{numeral } num ^ n)$
 by (induct n) (auto simp: one-ereal-def)

lemma zero-le-power-ereal[simp]:
 fixes $a :: \text{ereal}$
 assumes $0 \leq a$
 shows $0 \leq a ^ n$
 using assms by (induct n) (auto simp: ereal-zero-le-0-iff)

38.1.5 Subtraction

lemma ereal-minus-minus-image[simp]:
 fixes $S :: \text{ereal set}$
 shows $\text{uminus } ' \text{uminus } ' S = S$
 by (auto simp: image-iff)

lemma ereal-uminus-lessThan[simp]:
 fixes $a :: \text{ereal}$
 shows $\text{uminus } ' \{.. < a\} = \{-a < ..\}$
 by (force simp: ereal-uminus-less-reorder)

lemma ereal-uminus-greaterThan[simp]: $\text{uminus } ' \{(a :: \text{ereal}) < ..\} = \{.. < -a\}$
 by (metis ereal-uminus-lessThan ereal-uminus-uminus ereal-minus-minus-image)

instantiation ereal :: minus
 begin

definition $x - y = x + -(y :: \text{ereal})$
 instance ..

end

lemma *ereal-minus[simp]*:
 $\text{ereal } r - \text{ereal } p = \text{ereal } (r - p)$
 $-\infty - \text{ereal } r = -\infty$
 $\text{ereal } r - \infty = -\infty$
 $(\infty::\text{ereal}) - x = \infty$
 $-(\infty::\text{ereal}) - \infty = -\infty$
 $x - -y = x + y$
 $x - 0 = x$
 $0 - x = -x$
by (*simp-all add: minus-ereal-def*)

lemma *ereal-x-minus-x[simp]*: $x - x = (\text{if } |x| = \infty \text{ then } \infty \text{ else } 0::\text{ereal})$
by *auto*

lemma *ereal-eq-minus-iff*:
fixes $x \ y \ z :: \text{ereal}$
shows $x = z - y \longleftrightarrow$
 $(|y| \neq \infty \longrightarrow x + y = z) \wedge$
 $(y = -\infty \longrightarrow x = \infty) \wedge$
 $(y = \infty \longrightarrow z = \infty \longrightarrow x = \infty) \wedge$
 $(y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty)$
by (*cases rule: ereal3-cases[of x y z]*) *auto*

lemma *ereal-eq-minus*:
fixes $x \ y \ z :: \text{ereal}$
shows $|y| \neq \infty \implies x = z - y \longleftrightarrow x + y = z$
by (*auto simp: ereal-eq-minus-iff*)

lemma *ereal-less-minus-iff*:
fixes $x \ y \ z :: \text{ereal}$
shows $x < z - y \longleftrightarrow$
 $(y = \infty \longrightarrow z = \infty \wedge x \neq \infty) \wedge$
 $(y = -\infty \longrightarrow x \neq \infty) \wedge$
 $(|y| \neq \infty \longrightarrow x + y < z)$
by (*cases rule: ereal3-cases[of x y z]*) *auto*

lemma *ereal-less-minus*:
fixes $x \ y \ z :: \text{ereal}$
shows $|y| \neq \infty \implies x < z - y \longleftrightarrow x + y < z$
by (*auto simp: ereal-less-minus-iff*)

lemma *ereal-le-minus-iff*:
fixes $x \ y \ z :: \text{ereal}$
shows $x \leq z - y \longleftrightarrow (y = \infty \longrightarrow z \neq \infty \longrightarrow x = -\infty) \wedge (|y| \neq \infty \longrightarrow x + y \leq z)$
by (*cases rule: ereal3-cases[of x y z]*) *auto*

lemma *ereal-le-minus*:

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x \leq z - y \longleftrightarrow x + y \leq z$

by (*auto simp: ereal-le-minus-iff*)

lemma *ereal-minus-less-iff*:

fixes $x\ y\ z :: \text{ereal}$

shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$

by (*cases rule: ereal3-cases[of x y z]*) *auto*

lemma *ereal-minus-less*:

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x - y < z \longleftrightarrow x < z + y$

by (*auto simp: ereal-minus-less-iff*)

lemma *ereal-minus-le-iff*:

fixes $x\ y\ z :: \text{ereal}$

shows $x - y \leq z \longleftrightarrow$

$(y = -\infty \longrightarrow z = \infty) \wedge$

$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$

$(|y| \neq \infty \longrightarrow x \leq z + y)$

by (*cases rule: ereal3-cases[of x y z]*) *auto*

lemma *ereal-minus-le*:

fixes $x\ y\ z :: \text{ereal}$

shows $|y| \neq \infty \implies x - y \leq z \longleftrightarrow x \leq z + y$

by (*auto simp: ereal-minus-le-iff*)

lemma *ereal-minus-eq-minus-iff*:

fixes $a\ b\ c :: \text{ereal}$

shows $a - b = a - c \longleftrightarrow$

$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$

by (*cases rule: ereal3-cases[of a b c]*) *auto*

lemma *ereal-add-le-add-iff*:

fixes $a\ b\ c :: \text{ereal}$

shows $c + a \leq c + b \longleftrightarrow$

$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

by (*cases rule: ereal3-cases[of a b c]*) (*simp-all add: field-simps*)

lemma *ereal-add-le-add-iff2*:

fixes $a\ b\ c :: \text{ereal}$

shows $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$

by (*metis (no-types, lifting) add.commute ereal-add-le-add-iff*)

lemma *ereal-mult-le-mult-iff*:

fixes $a\ b\ c :: \text{ereal}$

shows $|c| \neq \infty \implies c * a \leq c * b \iff (0 < c \implies a \leq b) \wedge (c < 0 \implies b \leq a)$
by (cases rule: ereal3-cases[of a b c]) (simp-all add: mult-le-cancel-left)

lemma *ereal-minus-mono*:

fixes $A B C D :: \text{ereal}$ **assumes** $A \leq B$ $D \leq C$

shows $A - C \leq B - D$

using *assms*

by (cases rule: ereal3-cases[case-product ereal-cases, of A B C D]) simp-all

lemma *ereal-mono-minus-cancel*:

fixes $a b c :: \text{ereal}$

shows $c - a \leq c - b \implies 0 \leq c \implies c < \infty \implies b \leq a$

by (cases a b c rule: ereal3-cases) auto

lemma *real-of-ereal-minus*:

fixes $a b :: \text{ereal}$

shows $\text{real-of-ereal } (a - b) = (\text{if } |a| = \infty \vee |b| = \infty \text{ then } 0 \text{ else } \text{real-of-ereal } a - \text{real-of-ereal } b)$

by (cases rule: ereal2-cases[of a b]) auto

lemma *real-of-ereal-minus'*: $|x| = \infty \iff |y| = \infty \implies \text{real-of-ereal } x - \text{real-of-ereal } y = \text{real-of-ereal } (x - y :: \text{ereal})$

by(subst real-of-ereal-minus) auto

lemma *ereal-diff-positive*:

fixes $a b :: \text{ereal}$ **shows** $a \leq b \implies 0 \leq b - a$

by (cases rule: ereal2-cases[of a b]) auto

lemma *ereal-between*:

fixes $x e :: \text{ereal}$

assumes $|x| \neq \infty$ **and** $0 < e$

shows $x - e < x$

and $x < x + e$

using *assms* **by** (cases x, cases e, auto)+

lemma *ereal-minus-eq-PIfty-iff*:

fixes $x y :: \text{ereal}$

shows $x - y = \infty \iff y = -\infty \vee x = \infty$

by (cases x y rule: ereal2-cases) simp-all

lemma *ereal-diff-add-eq-diff-diff-swap*:

fixes $x y z :: \text{ereal}$

shows $|y| \neq \infty \implies x - (y + z) = x - y - z$

by(cases x y z rule: ereal3-cases) simp-all

lemma *ereal-diff-add-assoc2*:

fixes $x y z :: \text{ereal}$

shows $x + y - z = x - z + y$

by(cases x y z rule: ereal3-cases) simp-all

lemma *ereal-add-uminus-conv-diff*: **fixes** $x\ y\ z :: \text{ereal}$ **shows** $-x + y = y - x$
by (*simp add: add.commute minus-ereal-def*)

lemma *ereal-minus-diff-eq*:
fixes $x\ y :: \text{ereal}$
shows $\llbracket x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq -\infty \rrbracket \Longrightarrow -(x - y) = y - x$
by(*cases x y rule: ereal2-cases*) *simp-all*

lemma *ediff-le-self* [*simp*]: $x - y \leq (x :: \text{enat})$
by(*cases x y rule: enat.exhaust[case-product enat.exhaust]*) *simp-all*

lemma *ereal-abs-diff*:
fixes $a\ b :: \text{ereal}$
shows $\text{abs}(a - b) \leq \text{abs } a + \text{abs } b$
by (*cases rule: ereal2-cases[of a b]*) (*auto*)

38.1.6 Division

instantiation *ereal* :: *inverse*
begin

function *inverse-ereal* **where**
 $\text{inverse } (\text{ereal } r) = (\text{if } r = 0 \text{ then } \infty \text{ else } \text{ereal } (\text{inverse } r))$
 $\mid \text{inverse } (\infty :: \text{ereal}) = 0$
 $\mid \text{inverse } (-\infty :: \text{ereal}) = 0$
by (*auto intro: ereal-cases*)
termination **by** (*relation {}*) *simp*

definition $x \text{ div } y = x * \text{inverse } (y :: \text{ereal})$

instance ..

end

lemma *real-of-ereal-inverse*[*simp*]:
fixes $a :: \text{ereal}$
shows $\text{real-of-ereal } (\text{inverse } a) = 1 / \text{real-of-ereal } a$
by (*cases a*) (*auto simp: inverse-eq-divide*)

lemma *ereal-inverse*[*simp*]:
 $\text{inverse } (0 :: \text{ereal}) = \infty$
 $\text{inverse } (1 :: \text{ereal}) = 1$
by (*simp-all add: one-ereal-def zero-ereal-def*)

lemma *ereal-divide*[*simp*]:
 $\text{ereal } r / \text{ereal } p = (\text{if } p = 0 \text{ then } \text{ereal } r * \infty \text{ else } \text{ereal } (r / p))$
unfolding *divide-ereal-def* **by** (*auto simp: divide-real-def*)

lemma *ereal-divide-same*[simp]:

fixes $x :: \text{ereal}$

shows $x / x = (\text{if } |x| = \infty \vee x = 0 \text{ then } 0 \text{ else } 1)$

by (cases x) (simp-all add: divide-real-def divide-ereal-def one-ereal-def)

lemma *ereal-inv-inv*[simp]:

fixes $x :: \text{ereal}$

shows $\text{inverse} (\text{inverse } x) = (\text{if } x \neq -\infty \text{ then } x \text{ else } \infty)$

by (cases x) auto

lemma *ereal-inverse-minus*[simp]:

fixes $x :: \text{ereal}$

shows $\text{inverse} (-x) = (\text{if } x = 0 \text{ then } \infty \text{ else } -\text{inverse } x)$

by (cases x) simp-all

lemma *ereal-uminus-divide*[simp]:

fixes $x \ y :: \text{ereal}$

shows $-x / y = -(x / y)$

unfolding divide-ereal-def **by** simp

lemma *ereal-divide-Infty*[simp]:

fixes $x :: \text{ereal}$

shows $x / \infty = 0 \ x / -\infty = 0$

unfolding divide-ereal-def **by** simp-all

lemma *ereal-divide-one*[simp]: $x / 1 = (x :: \text{ereal})$

unfolding divide-ereal-def **by** simp

lemma *ereal-divide-ereal*[simp]: $\infty / \text{ereal } r = (\text{if } 0 \leq r \text{ then } \infty \text{ else } -\infty)$

unfolding divide-ereal-def **by** simp

lemma *ereal-inverse-nonneg-iff*: $0 \leq \text{inverse } (x :: \text{ereal}) \longleftrightarrow 0 \leq x \vee x = -\infty$

by (cases x) auto

lemma *inverse-ereal-ge0I*: $0 \leq (x :: \text{ereal}) \implies 0 \leq \text{inverse } x$

by(cases x) simp-all

lemma *zero-le-divide-ereal*[simp]:

fixes $a :: \text{ereal}$

assumes $0 \leq a$ and $0 \leq b$

shows $0 \leq a / b$

by (simp add: assms divide-ereal-def ereal-inverse-nonneg-iff)

lemma *ereal-le-divide-pos*:

fixes $x \ y \ z :: \text{ereal}$

shows $x > 0 \implies x \neq \infty \implies y \leq z / x \longleftrightarrow x * y \leq z$

by (cases rule: ereal3-cases[of $x \ y \ z$]) (auto simp: field-simps)

lemma *ereal-divide-le-pos*:

fixes $x\ y\ z :: \text{ereal}$
shows $x > 0 \implies x \neq \infty \implies z / x \leq y \longleftrightarrow z \leq x * y$
by (cases rule: *ereal3-cases*[of $x\ y\ z$]) (auto simp: *field-simps*)

lemma *ereal-le-divide-neg*:

fixes $x\ y\ z :: \text{ereal}$
shows $x < 0 \implies x \neq -\infty \implies y \leq z / x \longleftrightarrow z \leq x * y$
by (cases rule: *ereal3-cases*[of $x\ y\ z$]) (auto simp: *field-simps*)

lemma *ereal-divide-le-neg*:

fixes $x\ y\ z :: \text{ereal}$
shows $x < 0 \implies x \neq -\infty \implies z / x \leq y \longleftrightarrow x * y \leq z$
by (cases rule: *ereal3-cases*[of $x\ y\ z$]) (auto simp: *field-simps*)

lemma *ereal-inverse-antimono-strict*:

fixes $x\ y :: \text{ereal}$
shows $0 \leq x \implies x < y \implies \text{inverse } y < \text{inverse } x$
by (cases rule: *ereal2-cases*[of $x\ y$]) auto

lemma *ereal-inverse-antimono*:

fixes $x\ y :: \text{ereal}$
shows $0 \leq x \implies x \leq y \implies \text{inverse } y \leq \text{inverse } x$
by (cases rule: *ereal2-cases*[of $x\ y$]) auto

lemma *inverse-inverse-Pinfy-iff*[simp]:

fixes $x :: \text{ereal}$
shows $\text{inverse } x = \infty \longleftrightarrow x = 0$
by (cases x) auto

lemma *ereal-inverse-eq-0*:

fixes $x :: \text{ereal}$
shows $\text{inverse } x = 0 \longleftrightarrow x = \infty \vee x = -\infty$
by (cases x) auto

lemma *ereal-0-gt-inverse*:

fixes $x :: \text{ereal}$
shows $0 < \text{inverse } x \longleftrightarrow x \neq \infty \wedge 0 \leq x$
by (cases x) auto

lemma *ereal-inverse-le-0-iff*:

fixes $x :: \text{ereal}$
shows $\text{inverse } x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$
by(cases x) auto

lemma *ereal-divide-eq-0-iff*: $x / y = 0 \longleftrightarrow x = 0 \vee |y :: \text{ereal}| = \infty$
by(cases $x\ y$ rule: *ereal2-cases*) simp-all

lemma *ereal-mult-less-right*:

fixes $a\ b\ c :: \text{ereal}$

```

assumes  $b * a < c * a$   $0 < a$   $a < \infty$ 
shows  $b < c$ 
using assms
by (metis order.asym ereal-mult-strict-left-mono linorder-neqE mult.commute)

```

```

lemma ereal-mult-divide:
  fixes  $a\ b :: \text{ereal}$ 
  shows  $0 < b \implies b < \infty \implies b * (a / b) = a$ 
  by (cases a b rule: ereal2-cases) auto

```

```

lemma ereal-power-divide:
  fixes  $x\ y :: \text{ereal}$ 
  shows  $y \neq 0 \implies (x / y) ^ n = x ^ n / y ^ n$ 
  by (cases rule: ereal2-cases [of x y])
    (auto simp: one-ereal-def zero-ereal-def power-divide zero-le-power-eq)

```

```

lemma ereal-le-mult-one-interval:
  fixes  $x\ y :: \text{ereal}$ 
  assumes  $y: y \neq -\infty$ 
  assumes  $z: \bigwedge z. 0 < z \implies z < 1 \implies z * x \leq y$ 
  shows  $x \leq y$ 
proof (cases x)
  case PInf
    with  $z[\text{of } 1 / 2]$  show  $x \leq y$ 
    by (simp add: one-ereal-def)
  next
    case  $r: (\text{real } r)$ 
    show  $x \leq y$ 
    proof (cases y)
      case  $p: (\text{real } p)$ 
      have  $r \leq p$ 
      proof (rule field-le-mult-one-interval)
        fix  $z :: \text{real}$ 
        assume  $0 < z$  and  $z < 1$ 
        with  $z[\text{of } \text{ereal } z]$  show  $z * r \leq p$ 
        using  $p\ r$  by (auto simp: zero-le-mult-iff one-ereal-def)
      qed
    then show  $x \leq y$ 
    using  $p\ r$  by simp
    qed (use y in simp-all)
qed simp

```

```

lemma ereal-divide-right-mono[simp]:
  fixes  $x\ y\ z :: \text{ereal}$ 
  assumes  $x \leq y$ 
  and  $0 < z$ 
  shows  $x / z \leq y / z$ 
  using assms by (cases x y z rule: ereal3-cases) (auto intro: divide-right-mono)

```

lemma *ereal-divide-left-mono*[simp]:
 fixes $x\ y\ z :: \text{ereal}$
 assumes $y \leq x$
 and $0 < z$
 and $0 < x * y$
 shows $z / x \leq z / y$
 using *assms*
 by (*cases* $x\ y\ z$ *rule: ereal3-cases*)
 (*auto intro: divide-left-mono simp: field-simps zero-less-mult-iff mult-less-0-iff*
split: if-split-asm)

lemma *ereal-divide-zero-left*[simp]:
 fixes $a :: \text{ereal}$
 shows $0 / a = 0$
 using *ereal-divide-eq-0-iff* **by** *blast*

lemma *ereal-times-divide-eq-left*[simp]:
 fixes $a\ b\ c :: \text{ereal}$
 shows $b / c * a = b * a / c$
by (*metis divide-ereal-def mult.assoc mult.commute*)

lemma *ereal-times-divide-eq*: $a * (b / c :: \text{ereal}) = a * b / c$
by (*metis ereal-times-divide-eq-left mult.commute*)

lemma *ereal-inverse-real* [simp]: $|z| \neq \infty \implies z \neq 0 \implies \text{ereal} (\text{inverse} (\text{real-of-ereal } z)) = \text{inverse } z$
by *auto*

lemma *ereal-inverse-mult*:
 $a \neq 0 \implies b \neq 0 \implies \text{inverse} (a * (b :: \text{ereal})) = \text{inverse } a * \text{inverse } b$
by (*cases* a ; *cases* b) *auto*

lemma *inverse-eq-infinity-iff-eq-zero* [simp]:
 $1/(x :: \text{ereal}) = \infty \longleftrightarrow x = 0$
by (*simp add: divide-ereal-def*)

lemma *ereal-distrib-left*:
 fixes $a\ b\ c :: \text{ereal}$
 assumes $a \neq \infty \vee b \neq -\infty$
 and $a \neq -\infty \vee b \neq \infty$
 and $|c| \neq \infty$
 shows $c * (a + b) = c * a + c * b$
by (*metis assms ereal-distrib mult.commute*)

lemma *ereal-distrib-minus-left*:
 fixes $a\ b\ c :: \text{ereal}$
 assumes $a \neq \infty \vee b \neq \infty$
 and $a \neq -\infty \vee b \neq -\infty$
 and $|c| \neq \infty$

shows $c * (a - b) = c * a - c * b$
using *assms ereal-distrib-left ereal-uminus-eq-reorder minus-ereal-def* **by** *auto*

lemma *ereal-distrib-minus-right*:
fixes $a\ b\ c :: \text{ereal}$
assumes $a \neq \infty \vee b \neq \infty$
and $a \neq -\infty \vee b \neq -\infty$
and $|c| \neq \infty$
shows $(a - b) * c = a * c - b * c$
by (*metis assms ereal-distrib-minus-left mult.commute*)

38.2 Complete lattice

instantiation *ereal* :: *lattice*
begin

definition [*simp*]: $\text{sup } x\ y = (\text{max } x\ y :: \text{ereal})$
definition [*simp*]: $\text{inf } x\ y = (\text{min } x\ y :: \text{ereal})$
instance **by** *standard simp-all*

end

instantiation *ereal* :: *complete-lattice*
begin

definition *bot* = $(-\infty :: \text{ereal})$
definition *top* = $(\infty :: \text{ereal})$

definition *Sup* $S = (\text{SOME } x :: \text{ereal}. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z))$
definition *Inf* $S = (\text{SOME } x :: \text{ereal}. (\forall y \in S. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x))$

lemma *ereal-complete-Sup*:
fixes $S :: \text{ereal set}$
shows $\exists x. (\forall y \in S. y \leq x) \wedge (\forall z. (\forall y \in S. y \leq z) \longrightarrow x \leq z)$
proof (*cases* $\exists x. \forall a \in S. a \leq \text{ereal } x$)
case *True*
then obtain y **where** $y: a \leq \text{ereal } y$ **if** $a \in S$ **for** a
by *auto*
then have $\infty \notin S$
by *force*
show *?thesis*
proof (*cases* $S \neq \{-\infty\} \wedge S \neq \{\}$)
case *True*
with $\langle \infty \notin S \rangle$ **obtain** x **where** $x: x \in S \wedge |x| \neq \infty$
by *auto*
obtain s **where** $s: \forall x \in \text{ereal} - \{-\infty\}. x \leq s \wedge (\forall x \in \text{ereal} - \{-\infty\}. x \leq z) \implies s \leq z$
for z

```

proof (atomize-elim, rule complete-real)
  show  $\exists x. x \in \text{ereal} - 'S$ 
    using  $x$  by auto
  show  $\exists z. \forall x \in \text{ereal} - 'S. x \leq z$ 
    by (auto dest: y intro!: exI[of - y])
qed
show ?thesis
proof (safe intro!: exI[of - ereal s])
  fix  $y$ 
  assume  $y \in S$ 
  with  $s \infty \notin S$  show  $y \leq \text{ereal } s$ 
    by (cases y) auto
next
  fix  $z$ 
  assume  $\forall y \in S. y \leq z$ 
  with  $\langle S \neq \{-\infty\} \wedge S \neq \{\} \rangle$  show  $\text{ereal } s \leq z$ 
    by (cases z) (auto intro!: s)
qed
next
  case False
  then show ?thesis
    by (auto intro!: exI[of -  $-\infty$ ])
qed
next
  case False
  then show ?thesis
    by (fastforce intro!: exI[of -  $\infty$ ] ereal-top intro: order-trans dest: less-imp-le simp: not-le)
qed

lemma ereal-complete-uminus-eq:
  fixes  $S :: \text{ereal set}$ 
  shows  $(\forall y \in \text{uminus}'S. y \leq x) \wedge (\forall z. (\forall y \in \text{uminus}'S. y \leq z) \longrightarrow x \leq z)$ 
     $\longleftrightarrow (\forall y \in S. -x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq -x)$ 
  by simp (metis ereal-minus-le-minus ereal-uminus-uminus)

lemma ereal-complete-Inf:
   $\exists x. (\forall y \in S :: \text{ereal set}. x \leq y) \wedge (\forall z. (\forall y \in S. z \leq y) \longrightarrow z \leq x)$ 
  using ereal-complete-Sup[of uminus 'S]
  unfolding ereal-complete-uminus-eq
  by auto

instance
proof
  show Sup  $\{\}$  = (bot::ereal)
    using ereal-bot by (auto simp: bot-ereal-def Sup-ereal-def)
  show Inf  $\{\}$  = (top::ereal)
    unfolding top-ereal-def Inf-ereal-def
    using ereal-infty-less-eq(1) ereal-less-eq(1) by blast

```

```

show  $\bigwedge x::ereal. \bigwedge A. x \in A \implies \text{Inf } A \leq x$ 
   $\bigwedge A z. (\bigwedge x::ereal. x \in A \implies z \leq x) \implies z \leq \text{Inf } A$ 
  by (auto intro: someI2-ex ereal-complete-Inf simp: Inf-ereal-def)
show  $\bigwedge x::ereal. \bigwedge A. x \in A \implies x \leq \text{Sup } A$ 
   $\bigwedge A z. (\bigwedge x::ereal. x \in A \implies x \leq z) \implies \text{Sup } A \leq z$ 
  by (auto intro: someI2-ex ereal-complete-Sup simp: Sup-ereal-def)
qed

```

```
end
```

```
instance ereal :: complete-linorder ..
```

```
instance ereal :: linear-continuum
```

```
proof
```

```
  show  $\exists a b::ereal. a \neq b$ 
```

```
  using zero-neq-one by blast
```

```
qed
```

```
lemma min-PInf [simp]:  $\min (\infty::ereal) x = x$ 
  by (metis min-top top-ereal-def)
```

```
lemma min-PInf2 [simp]:  $\min x (\infty::ereal) = x$ 
  by (metis min-top2 top-ereal-def)
```

```
lemma max-PInf [simp]:  $\max (\infty::ereal) x = \infty$ 
  by (metis max-top top-ereal-def)
```

```
lemma max-PInf2 [simp]:  $\max x (\infty::ereal) = \infty$ 
  by (metis max-top2 top-ereal-def)
```

```
lemma min-MInf [simp]:  $\min (-\infty::ereal) x = -\infty$ 
  by (metis min-bot bot-ereal-def)
```

```
lemma min-MInf2 [simp]:  $\min x (-\infty::ereal) = -\infty$ 
  by (metis min-bot2 bot-ereal-def)
```

```
lemma max-MInf [simp]:  $\max (-\infty::ereal) x = x$ 
  by (metis max-bot bot-ereal-def)
```

```
lemma max-MInf2 [simp]:  $\max x (-\infty::ereal) = x$ 
  by (metis max-bot2 bot-ereal-def)
```

38.3 Extended real intervals

```
lemma real-greaterThanLessThan-infinity-eq:
```

```
  real-of-ereal ‘ $\{N::ereal <..<\infty\}$ ’ =
```

```
  (if  $N = \infty$  then  $\{\}$  else if  $N = -\infty$  then UNIV else  $\{\text{real-of-ereal } N <..\}$ )
```

```
  by (force simp: real-less-ereal-iff intro!: image-eqI[where  $x=ereal \ -$ ] elim!: less-ereal.elims)
```

lemma *real-greaterThanLessThan-minus-infinity-eq*:

real-of-ereal ‘ $\{-\infty < .. < N :: \text{ereal}\}$ =
 (if $N = \infty$ then UNIV else if $N = -\infty$ then $\{\}$ else $\{.. < \text{real-of-ereal } N\}$)

proof –

have *real-of-ereal* ‘ $\{-\infty < .. < N :: \text{ereal}\}$ = *uminus* ‘ *real-of-ereal* ‘ $\{-N < .. < \infty\}$
by (*auto simp*: *ereal-uminus-less-reorder intro!*: *image-eqI*[**where** $x = -x$ **for** x])
also note *real-greaterThanLessThan-minus-infinity-eq*
finally show ?thesis **by** (*auto intro!*: *image-eqI*[**where** $x = -x$ **for** x])

qed

lemma *real-greaterThanLessThan-inter*:

real-of-ereal ‘ $\{N < .. < M :: \text{ereal}\}$ = *real-of-ereal* ‘ $\{-\infty < .. < M\} \cap \text{real-of-ereal}$ ‘
 $\{N < .. < \infty\}$
by (*force elim!*: *less-ereal.elims*)

lemma *real-atLeastGreaterThan-eq*: *real-of-ereal* ‘ $\{N < .. < M :: \text{ereal}\}$ =

(if $N = \infty$ then $\{\}$ else
 if $N = -\infty$ then
 (if $M = \infty$ then UNIV
 else if $M = -\infty$ then $\{\}$
 else $\{.. < \text{real-of-ereal } M\}$)
 else if $M = -\infty$ then $\{\}$
 else if $M = \infty$ then $\{\text{real-of-ereal } N < ..\}$
 else $\{\text{real-of-ereal } N < .. < \text{real-of-ereal } M\}$)

proof (*cases* $M = -\infty \vee M = \infty \vee N = -\infty \vee N = \infty$)

case *True*

then show ?thesis

by (*auto simp*: *real-greaterThanLessThan-minus-infinity-eq real-greaterThanLessThan-minus-infinity-eq*
)

next

case *False*

then obtain p q **where** $M = \text{ereal } p$ $N = \text{ereal } q$

by (*metis MInfty-eq-minfinity ereal.distinct(3) uminus-ereal.elims*)

moreover have $\bigwedge x. \llbracket q < x; x < p \rrbracket \implies x \in \text{real-of-ereal}$ ‘ $\{\text{ereal } q < .. < \text{ereal } p\}$

by (*metis greaterThanLessThan-iff imageI less-ereal.simps(1) real-of-ereal.simps(1)*)

ultimately show ?thesis

by (*auto elim!*: *less-ereal.elims*)

qed

lemma *real-image-ereal-ivl*:

fixes a $b :: \text{ereal}$

shows

real-of-ereal ‘ $\{a < .. < b\}$ =

(if $a < b$ then (if $a = -\infty$ then if $b = \infty$ then UNIV else $\{.. < \text{real-of-ereal } b\}$

else if $b = \infty$ then $\{\text{real-of-ereal } a < ..\}$ else $\{\text{real-of-ereal } a < .. < \text{real-of-ereal } b\}$
 else $\{\}$)

by (*cases* a ; *cases* b ; *simp add*: *real-atLeastGreaterThan-eq not-less*)

lemma **fixes** a b $c :: \text{ereal}$


```

shows not-infityI:  $a < b \implies b < c \implies \text{abs } b \neq \infty$ 
by force

context
  fixes  $r \ s \ t :: \text{real}$ 
begin

lemma interval-Ioo-neq-Ioi:  $\{r < .. < s\} \neq \{t < ..\}$ 
  by (simp add: set-eq-iff) (meson linordered-field-no-ub nless-le order-less-trans)

lemma interval-Ioo-neq-Iio:  $\{r < .. < s\} \neq \{.. < t\}$ 
  by (simp add: set-eq-iff) (meson linordered-field-no-lb order-less-irrefl order-less-trans)

lemma interval-neq-ioo-UNIV:  $\{r < .. < s\} \neq \text{UNIV}$ 
  and interval-Ioi-neq-UNIV:  $\{r < ..\} \neq \text{UNIV}$ 
  and interval-Iio-neq-UNIV:  $\{.. < r\} \neq \text{UNIV}$ 
  by auto

lemma interval-Ioi-neq-Iio:  $\{r < ..\} \neq \{.. < s\}$ 
  by (simp add: set-eq-iff) (meson lt-ex order-less-irrefl order-less-trans)

lemma interval-empty-neq-Ioi:  $\{\} \neq \{r < ..\}$ 
  and interval-empty-neq-Iio:  $\{\} \neq \{.. < r\}$ 
  by (auto simp: set-eq-iff linordered-field-no-ub linordered-field-no-lb)

end

lemmas interval-neqs = interval-Ioo-neq-Ioi interval-Ioo-neq-Iio
  interval-neq-ioo-UNIV interval-Ioi-neq-Iio
  interval-Ioi-neq-UNIV interval-Iio-neq-UNIV
  interval-empty-neq-Ioi interval-empty-neq-Iio

lemma greaterThanLessThan-eq-iff:
  fixes  $r \ s \ t \ u :: \text{real}$ 
  shows  $(\{r < .. < s\} = \{t < .. < u\}) = (r \geq s \wedge u \leq t \vee r = t \wedge s = u)$ 
  by (metis cInf-greaterThanLessThan cSup-greaterThanLessThan greaterThanLessThan-empty-iff not-le)

lemma real-of-ereal-image-greaterThanLessThan-iff:
   $\text{real-of-ereal } ' \{a < .. < b\} = \text{real-of-ereal } ' \{c < .. < d\} \longleftrightarrow (a \geq b \wedge c \geq d \vee a = c \wedge b = d)$ 
  unfolding real-atLeastGreaterThan-eq
  by (cases a; cases b; cases c; cases d;
    simp add: greaterThanLessThan-eq-iff interval-neqs interval-neqs[symmetric])

lemma uminus-image-real-of-ereal-image-greaterThanLessThan:
   $\text{uminus } ' \text{real-of-ereal } ' \{l < .. < u\} = \text{real-of-ereal } ' \{-u < .. < -l\}$ 
  by (force simp: algebra-simps ereal-less-uminus-reorder
    ereal-uminus-less-reorder intro: image-eqI[where  $x = -x$  for  $x$ ])

```

lemma *add-image-real-of-ereal-image-greaterThanLessThan*:

(+) $c \text{ ‘ } \text{real-of-ereal ‘ } \{l <..< u\} = \text{real-of-ereal ‘ } \{c + l <..< c + u\}$

apply *safe*

subgoal for x

using *ereal-less-add[of c]*

by (*force simp: real-of-ereal-add add commute*)

subgoal for $-x$

by (*force simp: add commute real-of-ereal-minus ereal-minus-less ereal-less-minus*

intro: image-eqI[where $x=x - c$])

done

lemma *add2-image-real-of-ereal-image-greaterThanLessThan*:

($\lambda x. x + c$) $\text{ ‘ } \text{real-of-ereal ‘ } \{l <..< u\} = \text{real-of-ereal ‘ } \{l + c <..< u + c\}$

using *add-image-real-of-ereal-image-greaterThanLessThan[of c l u]*

by (*metis add commute image-cong*)

lemma *minus-image-real-of-ereal-image-greaterThanLessThan*:

(-) $c \text{ ‘ } \text{real-of-ereal ‘ } \{l <..< u\} = \text{real-of-ereal ‘ } \{c - u <..< c - l\}$

(**is** $?l = ?r$)

proof –

have $?l = (+) c \text{ ‘ } \text{uminus ‘ } \text{real-of-ereal ‘ } \{l <..< u\}$ **by** *auto*

also note *uminus-image-real-of-ereal-image-greaterThanLessThan*

also note *add-image-real-of-ereal-image-greaterThanLessThan*

finally show $?thesis$ **by** (*simp add: minus-ereal-def*)

qed

lemma *real-ereal-bound-lemma-up*:

assumes $s \in \text{real-of-ereal ‘ } \{a <..< b\}$

assumes $t \notin \text{real-of-ereal ‘ } \{a <..< b\}$

assumes $s \leq t$

shows $b \neq \infty$

proof (*cases b*)

case *PInf*

then show $?thesis$

using *assms*

by (*metis UNIV-I empty-iff greaterThan-iff order-less-le-trans real-image-ereal-ivl*)

qed *auto*

lemma *real-ereal-bound-lemma-down*:

assumes $s: s \in \text{real-of-ereal ‘ } \{a <..< b\}$

and $t: t \notin \text{real-of-ereal ‘ } \{a <..< b\}$

and $t \leq s$

shows $a \neq -\infty$

by (*metis UNIV-I assms empty-iff lessThan-iff order-le-less-trans*

real-greaterThanLessThan-minus-infinity-eq)

38.4 Topological space

instantiation *ereal* :: *linear-continuum-topology*
begin

definition *open-ereal* :: *ereal set* \Rightarrow *bool* **where**

open-ereal-generated: *open-ereal* = *generate-topology* (*range lessThan* \cup *range greaterThan*)

instance

by *standard* (*simp add: open-ereal-generated*)

end

lemma *continuous-on-ereal*[*continuous-intros*]:

assumes *f*: *continuous-on s f* **shows** *continuous-on s* ($\lambda x. \text{ereal } (f x)$)

by (*rule continuous-on-compose2* [*OF continuous-onI-mono*[*of ereal UNIV*] *f*])
auto

lemma *tendsto-ereal*[*tendsto-intros, simp, intro*]: ($f \longrightarrow x$) $F \Longrightarrow ((\lambda x. \text{ereal } (f x)) \longrightarrow \text{ereal } x) F$

using *isCont-tendsto-compose*[*of x ereal f F*] *continuous-on-ereal*[*of UNIV* $\lambda x. x$]

by (*simp add: continuous-on-eq-continuous-at*)

lemma *tendsto-uminus-ereal*[*tendsto-intros, simp, intro*]:

assumes ($f \longrightarrow x$) F

shows ($(\lambda x. - f x :: \text{ereal}) \longrightarrow - x$) F

proof (*rule tendsto-compose*[*OF order-tendstoI assms*])

show $\bigwedge a. a < - x \Longrightarrow \forall_F x \text{ in at } x. a < - x$

by (*metis ereal-less-uminus-reorder eventually-at-topological lessThan-iff open-lessThan*)

show $\bigwedge a. - x < a \Longrightarrow \forall_F x \text{ in at } x. - x < a$

by (*metis ereal-uminus-reorder(2) eventually-at-topological greaterThan-iff open-greaterThan*)

qed

lemma *at-infty-ereal-eq-at-top*: *at* ∞ = *filtermap ereal at-top*

proof –

have $\bigwedge P b. \forall z. b \leq z \wedge b \neq z \longrightarrow P (\text{ereal } z) \Longrightarrow \exists N. \forall n \geq N. P (\text{ereal } n)$

by (*metis gt-ex order-less-le order-less-le-trans*)

then show *?thesis*

unfolding *filter-eq-iff eventually-at-filter eventually-at-top-linorder eventually-filtermap top-ereal-def*[*symmetric*]

apply (*subst eventually-nhds-top*[*of 0*])

apply (*auto simp: top-ereal-def less-le ereal-all-split ereal-ex-split*)

done

qed

lemma *ereal-Lim-uminus*: ($f \longrightarrow f0$) *net* $\longleftrightarrow ((\lambda x. - f x :: \text{ereal}) \longrightarrow - f0)$

net

using *tendsto-uminus-ereal*[of *f f0 net*] *tendsto-uminus-ereal*[of $\lambda x. - f x - f0$ *net*]

by *auto*

lemma *ereal-divide-less-iff*: $0 < (c::ereal) \implies c < \infty \implies a / c < b \longleftrightarrow a < b * c$

by (*cases a b c rule: ereal3-cases*) (*auto simp: field-simps*)

lemma *ereal-less-divide-iff*: $0 < (c::ereal) \implies c < \infty \implies a < b / c \longleftrightarrow a * c < b$

by (*cases a b c rule: ereal3-cases*) (*auto simp: field-simps*)

lemma *tendsto-cmult-ereal*[*tendsto-intros, simp, intro*]:

assumes *c*: $|c| \neq \infty$ **and** *f*: $(f \longrightarrow x) F$

shows $((\lambda x. c * f x::ereal) \longrightarrow c * x) F$

proof –

have *: $((\lambda x. c * f x::ereal) \longrightarrow c * x) F$ **if** $0 < c < \infty$ **for** *c* :: *ereal*

using *that*

apply (*intro tendsto-compose*[*OF - f*])

apply (*auto intro!*: *order-tendstoI simp: eventually-at-topological*)

apply (*rule-tac* *x*= $\{a/c <..\}$ **in** *exI*)

apply (*auto split: ereal.split simp: ereal-divide-less-iff mult.commute*) []

apply (*rule-tac* *x*= $\{.. < a/c\}$ **in** *exI*)

apply (*auto split: ereal.split simp: ereal-less-divide-iff mult.commute*) []

done

have $((0 < c \wedge c < \infty) \vee (-\infty < c \wedge c < 0) \vee c = 0)$

using *c* **by** (*cases c*) *auto*

then show *?thesis*

proof (*elim disjE conjE*)

assume $-\infty < c < 0$

then have $0 < -c - c < \infty$

by (*auto simp: ereal-uminus-reorder ereal-less-uminus-reorder*[of *0*])

then have $((\lambda x. (-c) * f x) \longrightarrow (-c) * x) F$

by (*rule **)

from *tendsto-uminus-ereal*[*OF this*] **show** *?thesis*

by *simp*

qed (*auto intro!*: *)

qed

lemma *tendsto-cmult-ereal-not-0*[*tendsto-intros, simp, intro*]:

assumes *x* $\neq 0$ **and** *f*: $(f \longrightarrow x) F$

shows $((\lambda x. c * f x::ereal) \longrightarrow c * x) F$

proof *cases*

assume $|c| = \infty$

show *?thesis*

proof (*rule filterlim-cong*[*THEN iffD1, OF refl refl - tendsto-const*])

have $0 < x \vee x < 0$

using $\langle x \neq 0 \rangle$ **by** (*auto simp: neg-iff*)

then show *eventually* $(\lambda x'. c * x = c * f x') F$

```

proof
  assume  $0 < x$  from order-tendstoD(1)[OF f this] show ?thesis
  by eventually-elim (use  $\langle 0 < x \rangle \langle |c| = \infty \rangle$  in auto)
next
  assume  $x < 0$  from order-tendstoD(2)[OF f this] show ?thesis
  by eventually-elim (use  $\langle x < 0 \rangle \langle |c| = \infty \rangle$  in auto)
qed
qed
qed (rule tendsto-cmult-ereal[OF - f])

lemma tendsto-cadd-ereal[tendsto-intros, simp, intro]:
  assumes  $c: y \neq -\infty$   $x \neq -\infty$  and  $f: (f \longrightarrow x) F$ 
  shows  $((\lambda x. f x + y)::ereal) \longrightarrow x + y$   $F$ 
  apply (intro tendsto-compose[OF - f])
  apply (auto intro!: order-tendstoI simp: eventually-at-topological)
  apply (rule-tac  $x=\{a - y < ..\}$  in exI)
  apply (auto split: ereal.split simp: ereal-minus-less-iff  $c$ ) []
  apply (rule-tac  $x=\{.. < a - y\}$  in exI)
  apply (auto split: ereal.split simp: ereal-less-minus-iff  $c$ ) []
  done

lemma tendsto-add-left-ereal[tendsto-intros, simp, intro]:
  assumes  $c: |y| \neq \infty$  and  $f: (f \longrightarrow x) F$ 
  shows  $((\lambda x. f x + y)::ereal) \longrightarrow x + y$   $F$ 
  apply (intro tendsto-compose[OF - f])
  apply (auto intro!: order-tendstoI simp: eventually-at-topological)
  apply (rule-tac  $x=\{a - y < ..\}$  in exI)
  apply (insert  $c$ , auto split: ereal.split simp: ereal-minus-less-iff) []
  apply (rule-tac  $x=\{.. < a - y\}$  in exI)
  apply (auto split: ereal.split simp: ereal-less-minus-iff  $c$ ) []
  done

lemma continuous-at-ereal[continuous-intros]: continuous  $F f \implies$  continuous  $F$ 
  ( $\lambda x. ereal (f x)$ )
  unfolding continuous-def by auto

lemma ereal-Sup:
  assumes  $*$ :  $|\text{SUP } a \in A. ereal a| \neq \infty$ 
  shows  $ereal (\text{Sup } A) = (\text{SUP } a \in A. ereal a)$ 
proof (rule continuous-at-Sup-mono)
  obtain  $r$  where  $r: ereal r = (\text{SUP } a \in A. ereal a)$   $A \neq \{\}$ 
  using  $*$  by (force simp: bot-ereal-def)
  then show bdd-above  $A$   $A \neq \{\}$ 
  by (auto intro!: SUP-upper bdd-aboveI[of - r] simp flip: ereal-less-eq)
qed (auto simp: mono-def continuous-at-imp-continuous-at-within continuous-at-ereal)

lemma ereal-SUP:  $|\text{SUP } a \in A. ereal (f a)| \neq \infty \implies ereal (\text{SUP } a \in A. f a) = (\text{SUP } a \in A. ereal (f a))$ 
  by (simp add: ereal-Sup image-comp)

```

lemma *ereal-Inf*:
 assumes *: $|\text{INF } a \in A. \text{ereal } a| \neq \infty$
 shows $\text{ereal } (\text{Inf } A) = (\text{INF } a \in A. \text{ereal } a)$
proof (rule *continuous-at-Inf-mono*)
 obtain r where $r: \text{ereal } r = (\text{INF } a \in A. \text{ereal } a) \ A \neq \{\}$
 using * by (force simp: *top-ereal-def*)
 then show $\text{bdd-below } A \ A \neq \{\}$
 by (auto intro!: *INF-lower bdd-belowI[of - r] simp flip: ereal-less-eq*)
qed (auto simp: *mono-def continuous-at-imp-continuous-at-within continuous-at-ereal*)

lemma *ereal-Inf'*:
 assumes *: $\text{bdd-below } A \ A \neq \{\}$
 shows $\text{ereal } (\text{Inf } A) = (\text{INF } a \in A. \text{ereal } a)$
proof (rule *ereal-Inf*)
 from * obtain $l \ u$ where $x \in A \implies l \leq x \ u \in A$ for x
 by (auto simp: *bdd-below-def*)
 then have $l \leq (\text{INF } x \in A. \text{ereal } x) (\text{INF } x \in A. \text{ereal } x) \leq u$
 by (auto intro!: *INF-greatest INF-lower*)
 then show $|\text{INF } a \in A. \text{ereal } a| \neq \infty$
 by auto
qed

lemma *ereal-INF*: $|\text{INF } a \in A. \text{ereal } (f \ a)| \neq \infty \implies \text{ereal } (\text{INF } a \in A. f \ a) = (\text{INF } a \in A. \text{ereal } (f \ a))$
 by (simp add: *ereal-Inf image-comp*)

lemma *ereal-Sup-uminus-image-eq*: $\text{Sup } (\text{uminus } 'S::\text{ereal set}) = - \text{Inf } S$
 by (auto intro!: *SUP-eqI*
 simp: *Ball-def[symmetric] ereal-uminus-le-reorder le-Inf-iff*
 intro!: *complete-lattice-class.Inf-lower2*)

lemma *ereal-SUP-uminus-eq*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 shows $(\text{SUP } x \in S. \text{uminus } (f \ x)) = - (\text{INF } x \in S. f \ x)$
 using *ereal-Sup-uminus-image-eq* [of $f \ 'S$] by (simp add: *image-comp*)

lemma *ereal-inj-on-uminus*[*intro, simp*]: $\text{inj-on } \text{uminus } (A :: \text{ereal set})$
 by (auto intro!: *inj-onI*)

lemma *ereal-Inf-uminus-image-eq*: $\text{Inf } (\text{uminus } 'S::\text{ereal set}) = - \text{Sup } S$
 using *ereal-Sup-uminus-image-eq* [of $\text{uminus } 'S$] by simp

lemma *ereal-INF-uminus-eq*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 shows $(\text{INF } x \in S. - f \ x) = - (\text{SUP } x \in S. f \ x)$
 using *ereal-Inf-uminus-image-eq* [of $f \ 'S$] by (simp add: *image-comp*)

lemma *ereal-SUP-not-infty*:

```

fixes  $f :: - \Rightarrow \text{ereal}$ 
shows  $A \neq \{\}$   $\implies l \neq -\infty \implies u \neq \infty \implies \forall a \in A. l \leq f\ a \wedge f\ a \leq u \implies |Sup$ 
 $(f\ 'A)| \neq \infty$ 
using  $SUP\text{-}upper2[of\ -\ A\ l\ f]$   $SUP\text{-}least[of\ A\ f\ u]$ 
by  $(cases\ Sup\ (f\ 'A))\ auto$ 

```

lemma *ereal-INF-not-infty*:

```

fixes  $f :: - \Rightarrow \text{ereal}$ 
shows  $A \neq \{\}$   $\implies l \neq -\infty \implies u \neq \infty \implies \forall a \in A. l \leq f\ a \wedge f\ a \leq u \implies |Inf$ 
 $(f\ 'A)| \neq \infty$ 
using  $INF\text{-}lower2[of\ -\ A\ f\ u]$   $INF\text{-}greatest[of\ A\ l\ f]$ 
by  $(cases\ Inf\ (f\ 'A))\ auto$ 

```

lemma *ereal-image-uminus-shift*:

```

fixes  $X\ Y :: \text{ereal set}$ 
shows  $uminus\ 'X = Y \longleftrightarrow X = uminus\ 'Y$ 
by  $(metis\ \text{ereal-minus-minus-image})$ 

```

lemma *Sup-eq-MInfty*:

```

fixes  $S :: \text{ereal set}$ 
shows  $Sup\ S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$ 
unfolding  $bot\text{-ereal-def}[symmetric]$  by auto

```

lemma *Inf-eq-PInfty*:

```

fixes  $S :: \text{ereal set}$ 
shows  $Inf\ S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$ 
using  $Sup\text{-eq-MInfty}[of\ uminus\ 'S]$ 
unfolding  $ereal\text{-}Sup\text{-uminus-image-eq}\ \text{ereal-image-uminus-shift}$  by simp

```

lemma *Inf-eq-MInfty*:

```

fixes  $S :: \text{ereal set}$ 
shows  $-\infty \in S \implies Inf\ S = -\infty$ 
unfolding  $bot\text{-ereal-def}[symmetric]$  by auto

```

lemma *Sup-eq-PInfty*:

```

fixes  $S :: \text{ereal set}$ 
shows  $\infty \in S \implies Sup\ S = \infty$ 
unfolding  $top\text{-ereal-def}[symmetric]$  by auto

```

lemma *not-MInfty-nonneg[simp]*: $0 \leq (x::\text{ereal}) \implies x \neq -\infty$

by *auto*

lemma *Sup-ereal-close*:

```

fixes  $e :: \text{ereal}$ 
assumes  $0 < e$ 
and  $S: |Sup\ S| \neq \infty\ S \neq \{\}$ 
shows  $\exists x \in S. Sup\ S - e < x$ 
using assms by  $(cases\ e)\ (auto\ intro!\ \text{less-Sup-iff}[THEN\ iffD1])$ 

```

lemma *Inf-ereal-close*:

fixes $e :: \text{ereal}$
assumes $| \text{Inf } X | \neq \infty$
and $0 < e$
shows $\exists x \in X. x < \text{Inf } X + e$
by (*meson Inf-less-iff assms ereal-between(2)*)

lemma *SUP-PInfy*:

$(\bigwedge n :: \text{nat}. \exists i \in A. \text{ereal } (\text{real } n) \leq f i) \implies (\text{SUP } i \in A. f i :: \text{ereal}) = \infty$
by (*meson SUP-upper2 less-PInf-Ex-of-nat linorder-not-less*)

lemma *SUP-nat-Infy*: $(\text{SUP } i. \text{ereal } (\text{real } i)) = \infty$

by (*rule SUP-PInfy auto*)

lemma *SUP-ereal-add-left*:

assumes $I \neq \{\}$ $c \neq -\infty$
shows $(\text{SUP } i \in I. f i + c :: \text{ereal}) = (\text{SUP } i \in I. f i) + c$
proof (*cases* $(\text{SUP } i \in I. f i) = -\infty$)
case *True*
then have $\bigwedge i. i \in I \implies f i = -\infty$
unfolding *Sup-eq-MInfy* **by** *auto*
with *True* **show** *?thesis*
by (*cases* c) (*auto simp: I ≠ {}*)
next
case *False*
then show *?thesis*
by (*subst continuous-at-Sup-mono[where f=λx. x + c]*)
(auto simp: continuous-at-imp-continuous-at-within continuous-at mono-def
add-mono I ≠ {}
c ≠ -∞ image-comp)
qed

lemma *SUP-ereal-add-right*:

fixes $c :: \text{ereal}$
shows $I \neq \{\} \implies c \neq -\infty \implies (\text{SUP } i \in I. c + f i) = c + (\text{SUP } i \in I. f i)$
using *SUP-ereal-add-left[of I c f]* **by** (*simp add: add commute*)

lemma *SUP-ereal-minus-right*:

assumes $I \neq \{\}$ $c \neq -\infty$
shows $(\text{SUP } i \in I. c - f i :: \text{ereal}) = c - (\text{INF } i \in I. f i)$
using *SUP-ereal-add-right[OF assms, of λi. - f i]*
by (*simp add: ereal-SUP-uminus-eq minus-ereal-def*)

lemma *SUP-ereal-minus-left*:

assumes $I \neq \{\}$ $c \neq \infty$
shows $(\text{SUP } i \in I. f i - c :: \text{ereal}) = (\text{SUP } i \in I. f i) - c$
using *SUP-ereal-add-left[OF I ≠ {}, of -c f]* **by** (*simp add: c ≠ ∞ minus-ereal-def*)

lemma *INF-ereal-minus-right*:
 assumes $I \neq \{\}$ and $|c| \neq \infty$
 shows $(\text{INF } i \in I. c - f i) = c - (\text{SUP } i \in I. f i :: \text{ereal})$
proof –
 have *: $(-c) + b = -(c - b)$ for b
 using $\langle |c| \neq \infty \rangle$ by (cases c b rule: *ereal2-cases*) auto
 show ?thesis
 using *SUP-ereal-add-right*[OF $\langle I \neq \{\} \rangle$, of $-c$ f] $\langle |c| \neq \infty \rangle$
 by (auto simp: * *ereal-SUP-uminus-eq*)
qed

lemma *SUP-ereal-le-addI*:
 fixes $f :: 'i \Rightarrow \text{ereal}$
 assumes $\bigwedge i. f i + y \leq z$ and $y \neq -\infty$
 shows $\text{Sup } (f \text{ ‘ UNIV}) + y \leq z$
 by (metis *SUP-ereal-add-left SUP-least UNIV-not-empty assms*)

lemma *SUP-combine*:
 fixes $f :: 'a :: \text{semilattice-sup} \Rightarrow 'a :: \text{semilattice-sup} \Rightarrow 'b :: \text{complete-lattice}$
 assumes *mono*: $\bigwedge a b c d. a \leq b \implies c \leq d \implies f a c \leq f b d$
 shows $(\text{SUP } i \in \text{UNIV}. \text{SUP } j \in \text{UNIV}. f i j) = (\text{SUP } i. f i i)$
proof (rule *antisym*)
 show $(\text{SUP } i j. f i j) \leq (\text{SUP } i. f i i)$
 by (rule *SUP-least SUP-upper2*[where $i = \text{sup } i j$ for $i j$] *UNIV-I mono sup-ge1 sup-ge2*) +
 show $(\text{SUP } i. f i i) \leq (\text{SUP } i j. f i j)$
 by (rule *SUP-least SUP-upper2 UNIV-I mono order-refl*) +
qed

lemma *SUP-ereal-add*:
 fixes $f g :: \text{nat} \Rightarrow \text{ereal}$
 assumes *inc*: *incseq* f *incseq* g
 and *pos*: $\bigwedge i. f i \neq -\infty \bigwedge i. g i \neq -\infty$
 shows $(\text{SUP } i. f i + g i) = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$
proof –
 have $\bigwedge i j k l. \llbracket i \leq j; k \leq l \rrbracket \implies f i + g k \leq f j + g l$
 by (meson *add-mono inc incseq-def*)
 then have $(\text{SUP } i. f i + g i) = (\text{SUP } i j. f i + g j)$
 by (simp add: *SUP-combine*)
 also have $\dots = (\text{SUP } i j. g j + f i)$
 by (simp add: *add commute*)
 also have $\dots = (\text{SUP } i. \text{Sup } (\text{range } g) + f i)$
 by (simp add: *SUP-ereal-add-left pos(1)*)
 also have $\dots = (\text{SUP } i. f i + \text{Sup } (\text{range } g))$
 by (simp add: *add commute*)
 also have $\dots = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$
 by (simp add: *SUP-eq-iff SUP-ereal-add-left pos(2)*)
 finally show ?thesis .
qed

lemma *INF-eq-minf*: $(\text{INF } i \in I. f \ i :: \text{ereal}) \neq -\infty \longleftrightarrow (\exists b > -\infty. \forall i \in I. b \leq f \ i)$
unfolding *bot-ereal-def[symmetric]* *INF-eq-bot-iff* **by** (*auto simp: not-less*)

lemma *INF-ereal-add-left*:

assumes $I \neq \{\}$ $c \neq -\infty \wedge x. x \in I \implies 0 \leq f \ x$

shows $(\text{INF } i \in I. f \ i + c :: \text{ereal}) = (\text{INF } i \in I. f \ i) + c$

proof –

have $(\text{INF } i \in I. f \ i) \neq -\infty$

unfolding *INF-eq-minf* **using** *assms* **by** (*intro exI[of - 0]*) *auto*

then show *?thesis*

by (*subst continuous-at-Inf-mono[where f= $\lambda x. x + c$]*)

(*auto simp: mono-def add-mono* $\langle I \neq \{\} \rangle \langle c \neq -\infty \rangle$ *continuous-at-imp-continuous-at-within*
continuous-at image-comp)

qed

lemma *INF-ereal-add-right*:

assumes $I \neq \{\}$ $c \neq -\infty \wedge x. x \in I \implies 0 \leq f \ x$

shows $(\text{INF } i \in I. c + f \ i :: \text{ereal}) = c + (\text{INF } i \in I. f \ i)$

using *INF-ereal-add-left[OF assms]* **by** (*simp add: ac-simps*)

lemma *INF-ereal-add-directed*:

fixes $f \ g :: 'a \Rightarrow \text{ereal}$

assumes *nonneg*: $\bigwedge i. i \in I \implies 0 \leq f \ i \wedge i. i \in I \implies 0 \leq g \ i$

assumes *directed*: $\bigwedge i \ j. i \in I \implies j \in I \implies \exists k \in I. f \ i + g \ j \geq f \ k + g \ k$

shows $(\text{INF } i \in I. f \ i + g \ i) = (\text{INF } i \in I. f \ i) + (\text{INF } i \in I. g \ i)$

proof (*cases* $I = \{\}$)

case *False*

show *?thesis*

proof (*rule antisym*)

show $(\text{INF } i \in I. f \ i) + (\text{INF } i \in I. g \ i) \leq (\text{INF } i \in I. f \ i + g \ i)$

by (*rule INF-greatest; intro add-mono INF-lower*)

next

have $(\text{INF } i \in I. f \ i + g \ i) \leq (\text{INF } i \in I. (\text{INF } j \in I. f \ i + g \ j))$

using *directed* **by** (*intro INF-greatest*) (*blast intro: INF-lower2*)

also have $\dots = (\text{INF } i \in I. f \ i + (\text{INF } i \in I. g \ i))$

using *nonneg* $\langle I \neq \{\} \rangle$ **by** (*auto simp: INF-ereal-add-right*)

also have $\dots = (\text{INF } i \in I. f \ i) + (\text{INF } i \in I. g \ i)$

using *nonneg* **by** (*intro INF-ereal-add-left* $\langle I \neq \{\} \rangle$) (*auto simp: INF-eq-minf*
intro!: exI[of - 0])

finally show $(\text{INF } i \in I. f \ i + g \ i) \leq (\text{INF } i \in I. f \ i) + (\text{INF } i \in I. g \ i)$.

qed

qed (*simp add: top-ereal-def*)

lemma *INF-ereal-add*:

fixes $f :: \text{nat} \Rightarrow \text{ereal}$

assumes *decseq* f *decseq* g

and *fin*: $\bigwedge i. f \ i \neq \infty \wedge i. g \ i \neq \infty$

shows $(\text{INF } i. f \ i + g \ i) = \text{Inf } (f \ ' \text{UNIV}) + \text{Inf } (g \ ' \text{UNIV})$

proof –

have *INF-less*: $(\text{INF } i. f \ i) < \infty \ (\text{INF } i. g \ i) < \infty$
 using *assms* **unfolding** *INF-less-iff* **by** *auto*
 have *: $- ((- a) + (- b)) = a + b$ **if** $a \neq \infty \ b \neq \infty$ **for** $a \ b :: \text{ereal}$
 using *that* **by** (*cases a b rule: ereal2-cases*) *auto*
 have $(\text{INF } i. f \ i + g \ i) = (\text{INF } i. - ((- f \ i) + (- g \ i)))$
by (*simp add: fin **)
 also have $\dots = \text{Inf } (f \text{ ‘ UNIV}) + \text{Inf } (g \text{ ‘ UNIV})$
unfolding *ereal-INF-uminus-eq*
 using *assms* *INF-less*
by (*subst SUP-ereal-add*) (*auto simp: ereal-SUP-uminus-eq fin **)
 finally **show** *?thesis* .

qed

lemma *SUP-ereal-add-pos*:

fixes $f \ g :: \text{nat} \Rightarrow \text{ereal}$
assumes *incseq f incseq g*
and $\bigwedge i. 0 \leq f \ i \ \bigwedge i. 0 \leq g \ i$
shows $(\text{SUP } i. f \ i + g \ i) = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$
by (*simp add: SUP-ereal-add assms*)

lemma *SUP-ereal-sum*:

fixes $f \ g :: 'a \Rightarrow \text{nat} \Rightarrow \text{ereal}$
assumes $\bigwedge n. n \in A \Longrightarrow \text{incseq } (f \ n)$
and $\text{pos}: \bigwedge n \ i. n \in A \Longrightarrow 0 \leq f \ n \ i$
shows $(\text{SUP } i. \sum_{n \in A} f \ n \ i) = (\sum_{n \in A} \text{Sup } ((f \ n) \text{ ‘ UNIV}))$
using *assms*
by (*induction A rule: infinite-finite-induct*) (*auto simp: incseq-sumI2 sum-nonneg SUP-ereal-add-pos*)

lemma *SUP-ereal-mult-left*:

fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $I \neq \{\}$
assumes $f: \bigwedge i. i \in I \Longrightarrow 0 \leq f \ i$ **and** $c: 0 \leq c$
shows $(\text{SUP } i \in I. c * f \ i) = c * (\text{SUP } i \in I. f \ i)$

proof (*cases (SUP i ∈ I. f i) = 0*)

case *True*

then have $\bigwedge i. i \in I \Longrightarrow f \ i = 0$

by (*metis SUP-upper f antisym*)

with *True* **show** *?thesis*

by *simp*

next

case *False*

then show *?thesis*

by (*subst continuous-at-Sup-mono[where f=λx. c * x]*)

(*auto simp: mono-def continuous-at continuous-at-imp-continuous-at-within*

$I \neq \{\}$) *image-comp*

intro!: ereal-mult-left-mono c)

qed

```

lemma countable-approach:
  fixes  $x :: \text{ereal}$ 
  assumes  $x \neq -\infty$ 
  shows  $\exists f. \text{incseq } f \wedge (\forall i :: \text{nat}. f\ i < x) \wedge (f \longrightarrow x)$ 
proof (cases  $x$ )
  case (real  $r$ )
    moreover have  $(\lambda n. r - \text{inverse } (\text{real } (\text{Suc } n))) \longrightarrow r - 0$ 
      by (intro tendsto-intros LIMSEQ-inverse-real-of-nat)
    ultimately show ?thesis
      by (intro exI[of -  $\lambda n. x - \text{inverse } (\text{Suc } n)$ ]) (auto simp: incseq-def)
  next
    case PInf with LIMSEQ-SUP[of  $\lambda n :: \text{nat}. \text{ereal } (\text{real } n)$ ] show ?thesis
      by (intro exI[of -  $\lambda n. \text{ereal } (\text{real } n)$ ]) (auto simp: incseq-def SUP-nat-Infty)
  qed (simp add: assms)

lemma Sup-countable-SUP:
  assumes  $A \neq \{\}$ 
  shows  $\exists f :: \text{nat} \Rightarrow \text{ereal}. \text{incseq } f \wedge \text{range } f \subseteq A \wedge \text{Sup } A = (\text{SUP } i. f\ i)$ 
proof cases
  assume  $\text{Sup } A = -\infty$ 
  with  $\langle A \neq \{\} \rangle$  have  $A = \{-\infty\}$ 
    by (auto simp: Sup-eq-MInfty)
  then show ?thesis
    by (auto intro!: exI[of -  $\lambda \cdot. -\infty$ ] simp: bot-ereal-def)
  next
    assume  $\text{Sup } A \neq -\infty$ 
    then obtain  $l$  where  $\text{incseq } l$  and  $l: l\ i < \text{Sup } A$  and  $l\text{-Sup}: l \longrightarrow \text{Sup } A$ 
  for  $i :: \text{nat}$ 
    by (auto dest: countable-approach)

  have  $\exists f. \forall n. (f\ n \in A \wedge l\ n \leq f\ n) \wedge (f\ n \leq f\ (\text{Suc } n))$  (is  $\exists f. ?P\ f$ )
proof (rule dependent-nat-choice)
  show  $\exists x. x \in A \wedge l\ 0 \leq x$ 
    using  $l[0]$  by (auto simp: less-Sup-iff)
  next
    fix  $x\ n$  assume  $x \in A \wedge l\ n \leq x$ 
    moreover from  $l[0\ \text{Suc } n]$  obtain  $y$  where  $y \in A \wedge l\ (\text{Suc } n) < y$ 
      by (auto simp: less-Sup-iff)
    ultimately show  $\exists y. (y \in A \wedge l\ (\text{Suc } n) \leq y) \wedge x \leq y$ 
      by (auto intro!: exI[of -  $\max\ x\ y$ ] split: split-max)
  qed
  then obtain  $f$  where  $f: ?P\ f ..$ 
  then have  $\text{range } f \subseteq A$   $\text{incseq } f$ 
    by (auto simp: incseq-Suc-iff)
  then have  $(\text{SUP } i. f\ i) = \text{Sup } A$ 
    by (meson LIMSEQ-SUP LIMSEQ-le Sup-subset-mono f  $l\text{-Sup}$ 
      order-class.order-eq-iff)
  then show ?thesis

```

by (metis ‹incseq f› ‹range f ⊆ A›)
qed

lemma *Inf-countable-INF*:

assumes $A \neq \{\}$ shows $\exists f::nat \Rightarrow ereal. decseq f \wedge range f \subseteq A \wedge Inf A = (INF i. f i)$

proof –

obtain f where $incseq f \wedge range f \subseteq uminus 'A \ Sup (uminus 'A) = (SUP i. f i)$
using *Sup-countable-SUP* [of $uminus 'A$] ‹ $A \neq \{\}$ › by auto
then show ?thesis
by (intro exI [of - $\lambda x. - f x$])
(auto simp: *ereal-Sup-uminus-image-eq* *ereal-INF-uminus-eq* *eq-commute* [of - -])
qed

lemma *SUP-countable-SUP*:

$A \neq \{\} \implies \exists f::nat \Rightarrow ereal. range f \subseteq g 'A \wedge Sup (g 'A) = Sup (f 'UNIV)$
using *Sup-countable-SUP* [of $g 'A$] by auto

38.5 Relation to enat

definition *ereal-of-enat* $n = (case\ n\ of\ enat\ n \Rightarrow ereal\ (real\ n) \mid \infty \Rightarrow \infty)$

declare [[*coercion* *ereal-of-enat* :: *enat* \Rightarrow *ereal*]]

declare [[*coercion* ($\lambda n. ereal\ (real\ n)$) :: *nat* \Rightarrow *ereal*]]

lemma *ereal-of-enat-simps*[*simp*]:

ereal-of-enat (*enat* n) = *ereal* n

ereal-of-enat $\infty = \infty$

by (*simp-all* add: *ereal-of-enat-def*)

lemma *ereal-of-enat-le-iff*[*simp*]: *ereal-of-enat* $m \leq ereal\ of\ enat\ n \longleftrightarrow m \leq n$

by (*cases* $m\ n$ rule: *enat2-cases*) auto

lemma *ereal-of-enat-less-iff*[*simp*]: *ereal-of-enat* $m < ereal\ of\ enat\ n \longleftrightarrow m < n$

by (*cases* $m\ n$ rule: *enat2-cases*) auto

lemma *numeral-le-ereal-of-enat-iff*[*simp*]: *numeral* $m \leq ereal\ of\ enat\ n \longleftrightarrow numeral\ m \leq n$

by (*cases* n) (*auto*)

lemma *numeral-less-ereal-of-enat-iff*[*simp*]: *numeral* $m < ereal\ of\ enat\ n \longleftrightarrow numeral\ m < n$

by (*cases* n) *auto*

lemma *ereal-of-enat-ge-zero-cancel-iff*[*simp*]: $0 \leq ereal\ of\ enat\ n \longleftrightarrow 0 \leq n$

by (*cases* n) (*auto* *simp* flip: *enat-0*)

lemma *ereal-of-enat-gt-zero-cancel-iff*[*simp*]: $0 < ereal\ of\ enat\ n \longleftrightarrow 0 < n$

```

by (cases n) (auto simp flip: enat-0)

lemma ereal-of-enat-zero[simp]: ereal-of-enat 0 = 0
  by (auto simp flip: enat-0)

lemma ereal-of-enat-inf[simp]: ereal-of-enat n =  $\infty$   $\longleftrightarrow$  n =  $\infty$ 
  by (cases n) auto

lemma ereal-of-enat-add: ereal-of-enat (m + n) = ereal-of-enat m + ereal-of-enat n
  by (cases m n rule: enat2-cases) auto

lemma ereal-of-enat-sub:
  assumes n  $\leq$  m
  shows ereal-of-enat (m - n) = ereal-of-enat m - ereal-of-enat n
  using assms by (cases m n rule: enat2-cases) auto

lemma ereal-of-enat-mult:
  ereal-of-enat (m * n) = ereal-of-enat m * ereal-of-enat n
  by (cases m n rule: enat2-cases) auto

lemmas ereal-of-enat-pushin = ereal-of-enat-add ereal-of-enat-sub ereal-of-enat-mult
lemmas ereal-of-enat-pushout = ereal-of-enat-pushin[symmetric]

lemma ereal-of-enat-nonneg: ereal-of-enat n  $\geq$  0
  by simp

lemma ereal-of-enat-Sup:
  assumes A  $\neq$  {} shows ereal-of-enat (Sup A) = (SUP a  $\in$  A. ereal-of-enat a)
proof (intro antisym mono-Sup)
  show ereal-of-enat (Sup A)  $\leq$  (SUP a  $\in$  A. ereal-of-enat a)
proof cases
  assume finite A
  with  $\langle A \neq \{\} \rangle$  obtain a where a  $\in$  A ereal-of-enat (Sup A) = ereal-of-enat a
  using Max-in[of A] by (auto simp: Sup-enat-def simp del: Max-in)
  then show ?thesis
  by (auto intro: SUP-upper)
next
  assume  $\neg$  finite A
  have [simp]: (SUP a  $\in$  A. ereal-of-enat a) = top
  unfolding SUP-eq-top-iff
proof safe
  fix x :: ereal assume x < top
  then obtain n :: nat where x < n
  using less-PIInf-Ex-of-nat top-ereal-def by auto
  obtain a where a  $\in$  A - enat ‘ {.. n}
  by (metis  $\langle \neg$  finite A  $\rangle$  all-not-in-conv finite-Diff2 finite-atMost finite-imageI
finite.emptyI)
  then have a  $\in$  A ereal n  $\leq$  ereal-of-enat a

```

```

    by (auto simp: image-iff Ball-def)
      (metis enat-iless enat-ord-simps(1) ereal-of-enat-less-iff ereal-of-enat-simps(1)
less-le not-less)
  with ⟨x < n⟩ show ∃ i ∈ A. x < ereal-of-enat i
    by (auto intro!: bexI[of - a])
  qed
  show ?thesis
    by simp
  qed
qed (simp add: mono-def)

```

lemma *ereal-of-enat-SUP*:

$A \neq \{\}$ $\implies \text{ereal-of-enat } (\text{SUP } a \in A. f a) = (\text{SUP } a \in A. \text{ereal-of-enat } (f a))$
 by (simp add: ereal-of-enat-Sup image-comp)

38.6 Limits on *ereal*

lemma *open-PInfy*: $\text{open } A \implies \infty \in A \implies (\exists x. \{\text{ereal } x < ..\} \subseteq A)$

unfolding *open-ereal-generated*

proof (*induct rule: generate-topology.induct*)

case (*Int A B*)

then obtain $x z$ where $\infty \in A \implies \{\text{ereal } x < ..\} \subseteq A$ $\infty \in B \implies \{\text{ereal } z < ..\} \subseteq B$

by *auto*

with *Int* show ?case

by (intro exI[of - max x z]) fastforce

next

case (*Basis S*)

moreover have $x \neq \infty \implies \exists t. x \leq \text{ereal } t$ for x

by (cases x) auto

ultimately show ?case

by (auto split: ereal.split)

qed (*fastforce simp: vimage-Union*) +

lemma *open-MInfy*: $\text{open } A \implies -\infty \in A \implies (\exists x. \{.. < \text{ereal } x\} \subseteq A)$

unfolding *open-ereal-generated*

proof (*induct rule: generate-topology.induct*)

case (*Int A B*)

then obtain $x z$ where $-\infty \in A \implies \{.. < \text{ereal } x\} \subseteq A$ $-\infty \in B \implies \{.. < \text{ereal } z\} \subseteq B$

by *auto*

with *Int* show ?case

by (intro exI[of - min x z]) fastforce

next

case (*Basis S*)

moreover have $x \neq -\infty \implies \exists t. \text{ereal } t \leq x$ for x

by (cases x) auto

ultimately show ?case

by (auto split: ereal.split)

qed (*fastforce simp: vimage-Union*)⁺

lemma *open-ereal-vimage*: $\text{open } S \implies \text{open } (\text{ereal} - ' S)$
by (*intro open-vimage continuous-intros*)

lemma *open-ereal*: $\text{open } S \implies \text{open } (\text{ereal} - ' S)$
unfolding *open-generated-order* [**where** *'a=real*]
proof (*induct rule: generate-topology.induct*)
case (*Basis S*)
moreover **have** $\bigwedge x. \text{ereal} - ' \{.. < x\} = \{ -\infty < .. < \text{ereal } x \}$
using *ereal-less-ereal-Ex* **by** *auto*
moreover **have** $\bigwedge x. \text{ereal} - ' \{x < ..\} = \{ \text{ereal } x < .. < \infty \}$
using *less-ereal.elims(2)* **by** *fastforce*
ultimately show *?case*
by *auto*
qed (*auto simp: image-Union image-Int*)

lemma *open-image-real-of-ereal*:
fixes *X::ereal set*
assumes *open X*
assumes *infty: $\infty \notin X$ $-\infty \notin X$*
shows $\text{open } (\text{real-of-ereal} - ' X)$
proof –
have $\text{real-of-ereal} - ' X = \text{ereal} - ' X$
using *infty_ereal-real* **by** (*force simp: set-eq-iff*)
thus *?thesis*
by (*auto intro!: open-ereal-vimage assms*)
qed

lemma *eventually-finite*:
fixes *x::ereal*
assumes $|x| \neq \infty$ ($f \longrightarrow x$) *F*
shows *eventually* ($\lambda x. |f x| \neq \infty$) *F*
proof –
have ($f \longrightarrow \text{ereal } (\text{real-of-ereal } x)$) *F*
using *assms* **by** (*cases x*) *auto*
then have *eventually* ($\lambda x. f x \in \text{ereal} - ' \text{UNIV}$) *F*
by (*rule topological-tendstoD*) (*auto intro: open-ereal*)
also have ($\lambda x. f x \in \text{ereal} - ' \text{UNIV}$) = ($\lambda x. |f x| \neq \infty$)
by *auto*
finally show *?thesis* .
qed

lemma *open-ereal-def*:
 $\text{open } A \iff \text{open } (\text{ereal} - ' A) \wedge (\infty \in A \longrightarrow (\exists x. \{ \text{ereal } x < .. \} \subseteq A)) \wedge (-\infty \in A \longrightarrow (\exists x. \{ .. < \text{ereal } x \} \subseteq A))$
(is $\text{open } A \iff ?rhs$ **)**
proof


```

assume open A
then show ?rhs
  using open-PInfty open-MInfty open-ereal-vimage by auto
next
  assume ?rhs
  then obtain x y where A: open (ereal -‘ A)  $\infty \in A \implies \{ereal x < ..\} \subseteq A - \infty$ 
   $\in A \implies \{.. < ereal y\} \subseteq A$ 
  by auto
  have *: A = ereal -‘ (ereal -‘ A)  $\cup$  (if  $\infty \in A$  then  $\{ereal x < ..\}$  else  $\{\}$ )  $\cup$  (if
   $-\infty \in A$  then  $\{.. < ereal y\}$  else  $\{\}$ )
  using A(2,3) by auto
  from open-ereal[OF A(1)] show open A
  by (subst *) (auto simp: open-Un)
qed

```

```

lemma open-PInfty2:
  assumes open A and  $\infty \in A$ 
  obtains x where  $\{ereal x < ..\} \subseteq A$ 
  using open-PInfty[OF assms] by auto

```

```

lemma open-MInfty2:
  assumes open A and  $-\infty \in A$ 
  obtains x where  $\{.. < ereal x\} \subseteq A$ 
  using open-MInfty[OF assms] by auto

```

```

lemma ereal-openE:
  assumes open A
  obtains x y where open (ereal -‘ A)
  and  $\infty \in A \implies \{ereal x < ..\} \subseteq A$ 
  and  $-\infty \in A \implies \{.. < ereal y\} \subseteq A$ 
  using assms open-ereal-def by auto

```

```

lemmas open-ereal-lessThan = open-lessThan[where 'a=ereal]
lemmas open-ereal-greaterThan = open-greaterThan[where 'a=ereal]
lemmas ereal-open-greaterThanLessThan = open-greaterThanLessThan[where 'a=ereal]
lemmas closed-ereal-atLeast = closed-atLeast[where 'a=ereal]
lemmas closed-ereal-atMost = closed-atMost[where 'a=ereal]
lemmas closed-ereal-atLeastAtMost = closed-atLeastAtMost[where 'a=ereal]
lemmas closed-ereal-singleton = closed-singleton[where 'a=ereal]

```

```

lemma ereal-open-cont-interval:
  fixes S :: ereal set
  assumes open S
  and x  $\in$  S
  and |x|  $\neq \infty$ 
  obtains e where e > 0 and  $\{x-e < .. < x+e\} \subseteq S$ 
proof –
  from ⟨open S⟩
  have open (ereal -‘ S)

```

by (rule ereal-openE)
 then obtain e where $e > 0$ and e : $\text{dist } y \text{ (real-of-ereal } x) < e \implies \text{ereal } y \in S$
 for y
 using assms unfolding open-dist by force
 show thesis
 proof (intro that subsetI)
 show $0 < \text{ereal } e$
 using $\langle 0 < e \rangle$ by auto
 fix y
 assume $y \in \{x - \text{ereal } e .. x + \text{ereal } e\}$
 with assms obtain t where $y = \text{ereal } t$ dist $t \text{ (real-of-ereal } x) < e$
 by (cases y) (auto simp: dist-real-def)
 then show $y \in S$
 using $e[\text{of } t]$ by auto
 qed
 qed

lemma ereal-open-cont-interval2:
 fixes $S :: \text{ereal set}$
 assumes open S and $x \in S$ and $|x| \neq \infty$
 obtains $a \ b$ where $a < x$ and $x < b$ and $\{a .. b\} \subseteq S$
 by (meson assms ereal-between ereal-open-cont-interval)

38.6.1 Convergent sequences

lemma lim-real-of-ereal[simp]:
 assumes $\text{lim}: (f \longrightarrow \text{ereal } x) \text{ net}$
 shows $((\lambda x. \text{real-of-ereal } (f x)) \longrightarrow x) \text{ net}$
 proof (intro topological-tendstoI)
 fix S
 assume open S and $x \in S$
 then have S : open S ereal $x \in \text{ereal } S$
 by (simp-all add: inj-image-mem-iff)
 show eventually $(\lambda x. \text{real-of-ereal } (f x) \in S) \text{ net}$
 by (auto intro: eventually-mono [OF lim [THEN topological-tendstoD, OF open-ereal, OF S]])
 qed

lemma lim-ereal[simp]: $((\lambda n. \text{ereal } (f n)) \longrightarrow \text{ereal } x) \text{ net} \longleftrightarrow (f \longrightarrow x) \text{ net}$
 by (auto dest!: lim-real-of-ereal)

lemma convergent-real-imp-convergent-ereal:
 assumes convergent a
 shows convergent $(\lambda n. \text{ereal } (a n))$ and $\text{lim } (\lambda n. \text{ereal } (a n)) = \text{ereal } (\text{lim } a)$
 proof –
 from assms obtain L where L : $a \longrightarrow L$ unfolding convergent-def ..
 hence $\text{lim}: (\lambda n. \text{ereal } (a n)) \longrightarrow \text{ereal } L$ using lim-ereal by auto
 thus convergent $(\lambda n. \text{ereal } (a n))$ unfolding convergent-def ..
 thus $\text{lim } (\lambda n. \text{ereal } (a n)) = \text{ereal } (\text{lim } a)$ using $\text{lim } L \text{ limI}$ by metis

qed

lemma *tendsto-PInfty*: $(f \longrightarrow \infty) F \longleftrightarrow (\forall r. \text{eventually } (\lambda x. \text{ereal } r < f x) F)$

proof –

```
{ fix l :: ereal
  assume  $\forall r. \text{eventually } (\lambda x. \text{ereal } r < f x) F$ 
  from this[THEN spec, of real-of-ereal l]
  have  $l \neq \infty \implies \text{eventually } (\lambda x. l < f x) F$ 
    by (cases l) (auto elim: eventually-mono)
}
```

then show *?thesis*
by (auto simp: order-tendsto-iff)

qed

lemma *tendsto-PInfty'*: $(f \longrightarrow \infty) F = (\forall r > c. \text{eventually } (\lambda x. \text{ereal } r < f x) F)$

proof –

```
{ fix r :: real
  assume  $\forall r > c. \text{eventually } (\lambda x. \text{ereal } r < f x) F$ 
  then have  $\text{eventually } (\lambda x. \text{ereal } r < f x) F$ 
    if  $r > c$  for r using that by blast
  then have  $\text{eventually } (\lambda x. \text{ereal } r < f x) F$ 
    by (smt (verit, del-Insts) ereal-less-le eventually-mono gt-ex)
}
```

then show *?thesis*
using *tendsto-PInfty* by blast

qed

lemma *tendsto-PInfty-eq-at-top*:

$((\lambda z. \text{ereal } (f z)) \longrightarrow \infty) F \longleftrightarrow (\text{LIM } z F. f z :> \text{at-top})$
unfolding *tendsto-PInfty filterlim-at-top-dense* by simp

lemma *tendsto-MInfty*: $(f \longrightarrow -\infty) F \longleftrightarrow (\forall r. \text{eventually } (\lambda x. f x < \text{ereal } r) F)$

unfolding *tendsto-def*

proof safe

```
fix S :: ereal set
assume open S  $-\infty \in S$ 
from open-MInfty[OF this] obtain B where  $\{..<\text{ereal } B\} \subseteq S$  ..
moreover
assume  $\forall r::\text{real}. \text{eventually } (\lambda z. f z < r) F$ 
then have  $\text{eventually } (\lambda z. f z \in \{..< B\}) F$ 
  by auto
ultimately show  $\text{eventually } (\lambda z. f z \in S) F$ 
  by (auto elim!: eventually-mono)
```

next

```
fix x
assume  $\forall S. \text{open } S \longrightarrow -\infty \in S \longrightarrow \text{eventually } (\lambda x. f x \in S) F$ 
from this[rule-format, of  $\{..<\text{ereal } x\}$ ] show  $\text{eventually } (\lambda y. f y < \text{ereal } x) F$ 
  by auto
```

qed

lemma *tendsto-MInfty'*: $(f \longrightarrow -\infty) \ F = (\forall r < c. \text{eventually } (\lambda x. \text{ereal } r > f x) \ F)$

proof (*subst tendsto-MInfty, intro iffI allI impI*)

assume $A: \forall r < c. \text{eventually } (\lambda x. \text{ereal } r > f x) \ F$

fix $r :: \text{real}$

from A **have** $A: \text{eventually } (\lambda x. \text{ereal } r > f x) \ F$ **if** $r < c$ **for** r **using** *that* **by** *blast*

show $\text{eventually } (\lambda x. \text{ereal } r > f x) \ F$

proof (*cases* $r < c$)

case *False*

hence $B: \text{ereal } r \geq \text{ereal } (c - 1)$ **by** *simp*

have $c > c - 1$ **by** *simp*

from $A[OF \text{ this}]$ **show** $\text{eventually } (\lambda x. \text{ereal } r > f x) \ F$

by *eventually-elim* (*erule less-le-trans*[$OF - B$])

qed (*simp add: A*)

qed *simp*

lemma *Lim-PInfty*: $f \longrightarrow \infty \iff (\forall B. \exists N. \forall n \geq N. f n \geq \text{ereal } B)$

unfolding *tendsto-PInfty eventually-sequentially*

proof *safe*

fix r

assume $\forall r. \exists N. \forall n \geq N. \text{ereal } r \leq f n$

then obtain N **where** $\forall n \geq N. \text{ereal } (r + 1) \leq f n$

by *blast*

moreover have $\text{ereal } r < \text{ereal } (r + 1)$

by *auto*

ultimately show $\exists N. \forall n \geq N. \text{ereal } r < f n$

by (*blast intro: less-le-trans*)

qed (*blast intro: less-imp-le*)

lemma *Lim-MInfty*: $f \longrightarrow -\infty \iff (\forall B. \exists N. \forall n \geq N. \text{ereal } B \geq f n)$

unfolding *tendsto-MInfty eventually-sequentially*

proof *safe*

fix r

assume $\forall r. \exists N. \forall n \geq N. f n \leq \text{ereal } r$

then obtain N **where** $\forall n \geq N. f n \leq \text{ereal } (r - 1)$

by *blast*

moreover have $\text{ereal } (r - 1) < \text{ereal } r$

by *auto*

ultimately show $\exists N. \forall n \geq N. f n < \text{ereal } r$

by (*blast intro: le-less-trans*)

qed (*blast intro: less-imp-le*)

lemma *Lim-bounded-PInfty*: $f \longrightarrow l \implies (\bigwedge n. f n \leq \text{ereal } B) \implies l \neq \infty$

using *LIMSEQ-le-const2*[*of f l eréal B*] **by** *auto*

lemma *Lim-bounded-MInfty*: $f \longrightarrow l \implies (\bigwedge n. \text{ereal } B \leq f n) \implies l \neq -\infty$

using *LIMSEQ-le-const*[of f l *ereal* B] by *auto*

lemma *tendsto-zero-erealI*:

assumes $\bigwedge e. e > 0 \implies \text{eventually } (\lambda x. |f x| < \text{ereal } e) F$

shows $(f \longrightarrow 0) F$

proof (*subst filterlim-cong*[*OF refl refl*])

from *assms*[*OF zero-less-one*] **show** $\text{eventually } (\lambda x. f x = \text{ereal } (\text{real-of-ereal } (f x))) F$

by *eventually-elim* (*auto simp: ereal-real*)

hence $\text{eventually } (\lambda x. \text{abs } (\text{real-of-ereal } (f x)) < e) F$ **if** $e > 0$ **for** e **using** *assms*[*OF that*]

by *eventually-elim* (*simp add: real-less-ereal-iff that*)

hence $((\lambda x. \text{real-of-ereal } (f x)) \longrightarrow 0) F$ **unfolding** *tendsto-iff*

by (*auto simp: tendsto-iff dist-real-def*)

thus $((\lambda x. \text{ereal } (\text{real-of-ereal } (f x))) \longrightarrow 0) F$ **by** (*simp add: zero-ereal-def*)

qed

lemma *Lim-bounded-PInfy2*: $f \longrightarrow l \implies \forall n \geq N. f n \leq \text{ereal } B \implies l \neq \infty$

using *LIMSEQ-le-const2*[of f l *ereal* B] by *fastforce*

lemma *real-of-ereal-mult*[*simp*]:

fixes $a b :: \text{ereal}$

shows $\text{real-of-ereal } (a * b) = \text{real-of-ereal } a * \text{real-of-ereal } b$

by (*cases rule: ereal2-cases*[of a b]) *auto*

lemma *real-of-ereal-eq-0*:

fixes $x :: \text{ereal}$

shows $\text{real-of-ereal } x = 0 \iff x = \infty \vee x = -\infty \vee x = 0$

by (*cases x*) *auto*

lemma *tendsto-ereal-realD*:

fixes $f :: 'a \Rightarrow \text{ereal}$

assumes $x \neq 0$

and *tendsto*: $((\lambda x. \text{ereal } (\text{real-of-ereal } (f x))) \longrightarrow x) \text{ net}$

shows $(f \longrightarrow x) \text{ net}$

proof (*intro topological-tendstoI*)

fix S

assume S : *open* S $x \in S$

with $\langle x \neq 0 \rangle$ **have** *open* $(S - \{0\})$ $x \in S - \{0\}$

by *auto*

from *tendsto*[*THEN topological-tendstoD*, *OF this*]

show $\text{eventually } (\lambda x. f x \in S) \text{ net}$

by (*rule eventually-conv-mp*) (*auto simp: ereal-real*)

qed

lemma *tendsto-ereal-realI*:

fixes $f :: 'a \Rightarrow \text{ereal}$

assumes $x: |x| \neq \infty$ **and** *tendsto*: $(f \longrightarrow x) \text{ net}$

shows $((\lambda x. \text{ereal } (\text{real-of-ereal } (f x))) \longrightarrow x) \text{ net}$

proof (*intro topological-tendstoI*)
fix S
assume *open* S **and** $x \in S$
with x **have** *open* $(S - \{\infty, -\infty\})$ $x \in S - \{\infty, -\infty\}$
by *auto*
from *tendsto*[*THEN topological-tendstoD, OF this*]
show *eventually* $(\lambda x. \text{ereal} (\text{real-of-ereal} (f x)) \in S)$ *net*
by (*elim eventually-mono*) (*auto simp: ereal-real*)
qed

lemma *ereal-mult-cancel-left*:

fixes $a b c :: \text{ereal}$
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
by (*cases rule: ereal3-cases[of a b c]*) (*simp-all add: zero-less-mult-iff*)

lemma *tendsto-add-ereal*:

fixes $x y :: \text{ereal}$
assumes $x: |x| \neq \infty$ **and** $y: |y| \neq \infty$
assumes $f: (f \longrightarrow x) F$ **and** $g: (g \longrightarrow y) F$
shows $((\lambda x. f x + g x) \longrightarrow x + y) F$
proof –
from x **obtain** r **where** $x' : x = \text{ereal } r$ **by** (*cases x*) *auto*
with f **have** $((\lambda i. \text{real-of-ereal} (f i)) \longrightarrow r) F$ **by** *simp*
moreover
from y **obtain** p **where** $y' : y = \text{ereal } p$ **by** (*cases y*) *auto*
with g **have** $((\lambda i. \text{real-of-ereal} (g i)) \longrightarrow p) F$ **by** *simp*
ultimately have $((\lambda i. \text{real-of-ereal} (f i) + \text{real-of-ereal} (g i)) \longrightarrow r + p) F$
by (*rule tendsto-add*)
moreover
from *eventually-finite*[*OF x f*] *eventually-finite*[*OF y g*]
have *eventually* $(\lambda x. f x + g x = \text{ereal} (\text{real-of-ereal} (f x) + \text{real-of-ereal} (g x)))$
 F
by *eventually-elim auto*
ultimately show *?thesis*
by (*simp add: x' y' cong: filterlim-cong*)
qed

lemma *tendsto-add-ereal-nonneg*:

fixes $x y :: \text{ereal}$
assumes $x \neq -\infty$ $y \neq -\infty$ $(f \longrightarrow x) F$ $(g \longrightarrow y) F$
shows $((\lambda x. f x + g x) \longrightarrow x + y) F$
proof (*cases x = ∞ \vee y = ∞*)
case *True*
moreover
{ fix $y :: \text{ereal}$ **and** $f g :: 'a \Rightarrow \text{ereal}$ **assume** $y \neq -\infty$ $(f \longrightarrow \infty) F$ $(g \longrightarrow$
 $y) F$
then obtain y' **where** $-\infty < y' < y$
using *dense*[*of $-\infty$ y*] **by** *auto*
have $((\lambda x. f x + g x) \longrightarrow \infty) F$

```

proof (rule tendsto-sandwich)
  have  $\forall_F x \text{ in } F. y' < g \ x$ 
    using order-tendstoD(1)[OF  $\langle (g \longrightarrow y) \ F \rangle \langle y' < y \rangle$ ] by auto
  then show  $\forall_F x \text{ in } F. f \ x + y' \leq f \ x + g \ x$ 
    by eventually-elim (auto intro!: add-mono)
  show  $\forall_F n \text{ in } F. f \ n + g \ n \leq \infty \ ((\lambda n. \infty) \longrightarrow \infty) \ F$ 
    by auto
  show  $((\lambda x. f \ x + y') \longrightarrow \infty) \ F$ 
    using tendsto-cadd-ereal[of  $y' \infty f \ F$ ]  $\langle (f \longrightarrow \infty) \ F \rangle \langle -\infty < y' \rangle$  by auto
qed }
note this[of  $y \ f \ g$ ] this[of  $x \ g \ f$ ]
ultimately show ?thesis
  using assms by (auto simp: add-ac)
next
  case False
  with assms tendsto-add-ereal[of  $x \ y \ f \ F \ g$ ]
  show ?thesis
    by auto
qed

lemma ereal-inj-affinity:
  fixes  $m \ t :: \text{ereal}$ 
  assumes  $|m| \neq \infty$ 
    and  $m \neq 0$ 
    and  $|t| \neq \infty$ 
  shows inj-on  $(\lambda x. m * x + t) \ A$ 
  using assms
  by (cases rule: ereal2-cases[of  $m \ t$ ])
    (auto intro!: inj-onI simp: ereal-add-cancel-right ereal-mult-cancel-left)

lemma ereal-PInfty-eq-plus[simp]:
  fixes  $a \ b :: \text{ereal}$ 
  shows  $\infty = a + b \longleftrightarrow a = \infty \vee b = \infty$ 
  by (cases rule: ereal2-cases[of  $a \ b$ ]) auto

lemma ereal-MInfty-eq-plus[simp]:
  fixes  $a \ b :: \text{ereal}$ 
  shows  $-\infty = a + b \longleftrightarrow (a = -\infty \wedge b \neq \infty) \vee (b = -\infty \wedge a \neq \infty)$ 
  by (cases rule: ereal2-cases[of  $a \ b$ ]) auto

lemma ereal-less-divide-pos:
  fixes  $x \ y :: \text{ereal}$ 
  shows  $x > 0 \implies x \neq \infty \implies y < z / x \longleftrightarrow x * y < z$ 
  by (simp add: ereal-less-divide-iff mult.commute)

lemma ereal-divide-less-pos:
  fixes  $x \ y \ z :: \text{ereal}$ 
  shows  $x > 0 \implies x \neq \infty \implies y / x < z \longleftrightarrow y < x * z$ 
  by (simp add: ereal-divide-less-iff mult.commute)

```

lemma *ereal-divide-eq*:
fixes $a\ b\ c :: \text{ereal}$
shows $b \neq 0 \implies |b| \neq \infty \implies a / b = c \longleftrightarrow a = b * c$
by (*metis* *ereal-divide-same* *ereal-times-divide-eq* *mult.commute* *mult.right-neutral*)

lemma *ereal-inverse-not-MInfty[simp]*: $\text{inverse } (a :: \text{ereal}) \neq -\infty$
by (*cases* a) *auto*

lemma *ereal-mult-m1[simp]*: $x * \text{ereal } (-1) = -x$
by (*cases* x) *auto*

lemma *ereal-real'*:
assumes $|x| \neq \infty$
shows $\text{ereal } (\text{real-of-ereal } x) = x$
using *assms* **by** *auto*

lemma *real-ereal-id*: $\text{real-of-ereal} \circ \text{ereal} = \text{id}$
by *auto*

lemma *open-image-ereal*: $\text{open}(UNIV - \{\infty, (-\infty :: \text{ereal})\})$
by (*metis* *range-ereal* *open-ereal* *open-UNIV*)

lemma *ereal-le-distrib*:
fixes $a\ b\ c :: \text{ereal}$
shows $c * (a + b) \leq c * a + c * b$
by (*cases* *rule*: *ereal3-cases*[*of* $a\ b\ c$])
(auto simp: field-simps not-le mult-le-0-iff mult-less-0-iff)

lemma *ereal-pos-distrib*:
fixes $a\ b\ c :: \text{ereal}$
assumes $0 \leq c$
and $c \neq \infty$
shows $c * (a + b) = c * a + c * b$
using *assms*
by (*cases* *rule*: *ereal3-cases*[*of* $a\ b\ c$])
(auto simp: field-simps not-le mult-le-0-iff mult-less-0-iff)

lemma *ereal-LimI-finite*:
fixes $x :: \text{ereal}$
assumes $|x| \neq \infty$
and $\bigwedge r. 0 < r \implies \exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r$
shows $u \longrightarrow x$
proof (*rule* *topological-tendstoI*, *unfold* *eventually-sequentially*)
obtain rx **where** $rx: x = \text{ereal } rx$
using *assms* **by** (*cases* x) *auto*
fix S
assume *open* S **and** $x \in S$


```

then have open (ereal - ' S)
  unfolding open-ereal-def by auto
with ⟨x ∈ S⟩ obtain r where 0 < r and dist: dist y rx < r ⟹ ereal y ∈ S
for y
  unfolding open-dist rx by auto
then obtain n
  where upper: u N < x + ereal r
    and lower: x < u N + ereal r
    if n ≤ N for N
  using assms(2)[of ereal r] by auto
show ∃ N. ∀ n ≥ N. u n ∈ S
proof (safe intro!: exI[of - n])
  fix N
  assume n ≤ N
  from upper[OF this] lower[OF this] assms ⟨0 < r⟩
  have u N ∉ {∞, (-∞)}
    by auto
  then obtain ra where ra-def: (u N) = ereal ra
    by (cases u N) auto
  then have rx < ra + r and ra < rx + r
    using rx assms ⟨0 < r⟩ lower[OF ⟨n ≤ N⟩] upper[OF ⟨n ≤ N⟩]
    by auto
  then have dist (real-of-ereal (u N)) rx < r
    using rx ra-def
    by (auto simp: dist-real-def abs-diff-less-iff field-simps)
  from dist[OF this] show u N ∈ S
    using ⟨u N ∉ {∞, -∞}⟩
    by (auto simp: ereal-real split: if-split-asm)
qed
qed

```

```

lemma tendsto-obtains-N:
  assumes f ⟶ f0 open S f0 ∈ S
  obtains N where ∀ n ≥ N. f n ∈ S
  using assms lim-explicit by blast

```

```

lemma ereal-LimI-finite-iff:
  fixes x :: ereal
  assumes |x| ≠ ∞
  shows u ⟶ x ⟷ (∀ r. 0 < r ⟶ (∃ N. ∀ n ≥ N. u n < x + r ∧ x < u n
+ r))
  (is ?lhs ⟷ ?rhs)
proof
  assume lim: u ⟶ x
  {
    fix r :: ereal
    assume r > 0
    then obtain N where ∀ n ≥ N. u n ∈ {x - r <..< x + r}
      using lim ereal-between[of x r] assms ⟨r > 0⟩ tendsto-obtains-N[of u x {x -

```

```

r <.. $x + r$ }]
  by auto
  then have  $\exists N. \forall n \geq N. u\ n < x + r \wedge x < u\ n + r$ 
    using ereal-minus-less[of r x]
    by (cases r) auto
  }
  then show ?rhs
    by auto
next
  assume ?rhs
  then show  $u \longrightarrow x$ 
    using ereal-LimI-finite[of x] assms by auto
qed

```

lemma *ereal-Limsup-uminus*:
 fixes $f :: 'a \Rightarrow \text{ereal}$
 shows $\text{Limsup } \text{net } (\lambda x. - (f\ x)) = - \text{Liminf } \text{net } f$
 unfolding *Limsup-def Liminf-def ereal-SUP-uminus-eq ereal-INF-uminus-eq* ..

lemma *liminf-bounded-iff*:
 fixes $x :: \text{nat} \Rightarrow \text{ereal}$
 shows $C \leq \text{liminf } x \longleftrightarrow (\forall B < C. \exists N. \forall n \geq N. B < x\ n)$
 (is ?lhs \longleftrightarrow ?rhs)
 unfolding *le-Liminf-iff eventually-sequentially* ..

lemma *Liminf-add-le*:
 fixes $f\ g :: - \Rightarrow \text{ereal}$
 assumes $F: F \neq \text{bot}$
 assumes *ev*: *eventually* $(\lambda x. 0 \leq f\ x)\ F$ *eventually* $(\lambda x. 0 \leq g\ x)\ F$
 shows $\text{Liminf } F\ f + \text{Liminf } F\ g \leq \text{Liminf } F\ (\lambda x. f\ x + g\ x)$
 unfolding *Liminf-def*
proof (*subst SUP-ereal-add-left[symmetric]*)
 let $?F = \{P. \text{eventually } P\ F\}$
 let $?INF = \lambda P\ g. \text{Inf } (g\ ` (\text{Collect } P))$
 show $?F \neq \{\}$
 by (*auto intro: eventually-True*)
 show $(\text{SUP } P \in ?F. ?INF\ P\ g) \neq -\infty$
 unfolding *bot-ereal-def[symmetric] SUP-bot-conv INF-eq-bot-iff*
 by (*auto intro!: exI[of - 0] ev simp: bot-ereal-def*)
 have $(\text{SUP } P \in ?F. ?INF\ P\ f + (\text{SUP } P \in ?F. ?INF\ P\ g)) \leq (\text{SUP } P \in ?F. (\text{SUP } P' \in ?F. ?INF\ P\ f + ?INF\ P'\ g))$
proof (*safe intro!: SUP-mono bexI[of - $\lambda x. P\ x \wedge 0 \leq f\ x$ for P]*)
 fix P let $?P' = \lambda x. P\ x \wedge 0 \leq f\ x$
 assume *eventually* $P\ F$
 with *ev* show *eventually* $?P'\ F$
 by *eventually-elim auto*
 have $?INF\ P\ f + (\text{SUP } P \in ?F. ?INF\ P\ g) \leq ?INF\ ?P'\ f + (\text{SUP } P \in ?F. ?INF\ P\ g)$
 by (*intro add-mono INF-mono*) *auto*

```

also have ... = (SUP P'∈?F. ?INF ?P' f + ?INF P' g)
proof (rule SUP-ereal-add-right[symmetric])
  show Inf (f ' {x. P x ∧ 0 ≤ f x}) ≠ -∞
    unfolding bot-ereal-def[symmetric] INF-eq-bot-iff
    by (auto intro!: exI[of - 0] ev simp: bot-ereal-def)
  qed fact
finally show ?INF P f + (SUP P∈?F. ?INF P g) ≤ (SUP P'∈?F. ?INF ?P'
f + ?INF P' g) .
qed
also have ... ≤ (SUP P∈?F. INF x∈Collect P. f x + g x)
proof (safe intro!: SUP-least)
  fix P Q assume *: eventually P F eventually Q F
  show ?INF P f + ?INF Q g ≤ (SUP P∈?F. INF x∈Collect P. f x + g x)
  proof (rule SUP-upper2)
    show (λx. P x ∧ Q x) ∈ ?F
    using * by (auto simp: eventually-conj)
    show ?INF P f + ?INF Q g ≤ (INF x∈{x. P x ∧ Q x}. f x + g x)
    by (intro INF-greatest add-mono) (auto intro: INF-lower)
  qed
qed
finally show (SUP P∈?F. ?INF P f + (SUP P∈?F. ?INF P g)) ≤ (SUP P∈?F.
INF x∈Collect P. f x + g x) .
qed

lemma Sup-ereal-mult-right':
  assumes nonempty: Y ≠ {}
  and x: x ≥ 0
  shows (SUP i∈Y. f i) * ereal x = (SUP i∈Y. f i * ereal x) (is ?lhs = ?rhs)
proof(cases x = 0)
  case True thus ?thesis by(auto simp: nonempty zero-ereal-def[symmetric])
next
  case False
  show ?thesis
  proof(rule antisym)
    show ?rhs ≤ ?lhs
    by(rule SUP-least)(simp add: ereal-mult-right-mono SUP-upper x)
  next
    have ?lhs / ereal x = (SUP i∈Y. f i) * (ereal x / ereal x) by(simp only:
ereal-times-divide-eq)
    also have ... = (SUP i∈Y. f i) using False by simp
    also have ... ≤ ?rhs / x
    proof(rule SUP-least)
      fix i
      assume i ∈ Y
      have f i = f i * (ereal x / ereal x) using False by simp
      also have ... = f i * x / x by(simp only: ereal-times-divide-eq)
      also from ⟨i ∈ Y⟩ have f i * x ≤ ?rhs by(rule SUP-upper)
      hence f i * x / x ≤ ?rhs / x using x False by simp
      finally show f i ≤ ?rhs / x .

```

```

qed
finally have (?lhs / x) * x ≤ (?rhs / x) * x
  by(rule ereal-mult-right-mono)(simp add: x)
also have ... = ?rhs using False ereal-divide-eq mult.commute by force
also have (?lhs / x) * x = ?lhs using False ereal-divide-eq mult.commute by
force
finally show ?lhs ≤ ?rhs .
qed
qed

```

lemma *Sup-ereal-mult-left'*:

```

[[ Y ≠ {} ; x ≥ 0 ]] ⇒ ereal x * (SUP i∈Y. f i) = (SUP i∈Y. ereal x * f i)
by (smt (verit) Sup.SUP-cong Sup-ereal-mult-right' mult.commute)

```

lemma *sup-continuous-add[order-continuous-intros]*:

```

fixes f g :: 'a::complete-lattice ⇒ ereal
assumes nn: ∧x. 0 ≤ f x ∧x. 0 ≤ g x and cont: sup-continuous f sup-continuous
g
shows sup-continuous (λx. f x + g x)
unfolding sup-continuous-def
proof safe
fix M :: nat ⇒ 'a assume incseq M
then show f (SUP i. M i) + g (SUP i. M i) = (SUP i. f (M i) + g (M i))
  using SUP-ereal-add-pos[of λi. f (M i) λi. g (M i)] nn
  cont[THEN sup-continuous-mono] cont[THEN sup-continuousD]
  by (auto simp: mono-def)
qed

```

lemma *sup-continuous-mult-right[order-continuous-intros]*:

```

0 ≤ c ⇒ c < ∞ ⇒ sup-continuous f ⇒ sup-continuous (λx. f x * c :: ereal)
by (cases c) (auto simp: sup-continuous-def fun-eq-iff Sup-ereal-mult-right')

```

lemma *sup-continuous-mult-left[order-continuous-intros]*:

```

0 ≤ c ⇒ c < ∞ ⇒ sup-continuous f ⇒ sup-continuous (λx. c * f x :: ereal)
using sup-continuous-mult-right[of c f] by (simp add: mult-ac)

```

lemma *sup-continuous-ereal-of-enat[order-continuous-intros]*:

```

assumes f: sup-continuous f
shows sup-continuous (λx. ereal-of-enat (f x))
by (metis UNIV-not-empty ereal-of-enat-SUP f sup-continuous-compose
sup-continuous-def)

```

38.6.2 Sums

lemma *sums-ereal-positive*:

```

fixes f :: nat ⇒ ereal
assumes ∧i. 0 ≤ f i
shows f sums (SUP n. ∑ i<n. f i)
by (simp add: LIMSEQ-SUP assms incseq-sumI sums-def)

```

```

lemma summable-ereal-pos:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  assumes  $\bigwedge i. 0 \leq f\ i$ 
  shows summable  $f$ 
  using sums-ereal-positive[of  $f$ , OF assms]
  unfolding summable-def
  by auto

lemma sums-ereal:  $(\lambda x. \text{ereal } (f\ x)) \text{ sums } \text{ereal } x \longleftrightarrow f \text{ sums } x$ 
  unfolding sums-def by simp

lemma suminf-ereal-eq-SUP:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  assumes  $\bigwedge i. 0 \leq f\ i$ 
  shows  $(\sum x. f\ x) = (\text{SUP } n. \sum i < n. f\ i)$ 
  using sums-ereal-positive[of  $f$ , OF assms, THEN sums-unique]
  by simp

lemma suminf-bound:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  assumes  $\forall N. (\sum n < N. f\ n) \leq x \wedge n. 0 \leq f\ n$ 
  shows  $\text{suminf } f \leq x$ 
  by (simp add: SUP-least assms suminf-ereal-eq-SUP)

lemma suminf-bound-add:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  assumes  $\forall N. (\sum n < N. f\ n) + y \leq x$ 
  and  $\bigwedge n. 0 \leq f\ n$ 
  and  $y \neq -\infty$ 
  shows  $\text{suminf } f + y \leq x$ 
  by (simp add: SUP-ereal-le-addI assms suminf-ereal-eq-SUP)

lemma suminf-upper:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  assumes  $\bigwedge n. 0 \leq f\ n$ 
  shows  $(\sum n < N. f\ n) \leq (\sum n. f\ n)$ 
  unfolding suminf-ereal-eq-SUP [OF assms]
  by (auto intro: complete-lattice-class.SUP-upper)

lemma suminf-0-le:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  assumes  $\bigwedge n. 0 \leq f\ n$ 
  shows  $0 \leq (\sum n. f\ n)$ 
  using suminf-upper[of  $f\ 0$ , OF assms]
  by simp

lemma suminf-le-pos:
  fixes  $f\ g :: \text{nat} \Rightarrow \text{ereal}$ 

```

```

assumes  $\bigwedge N. f\ N \leq g\ N$ 
and  $\bigwedge N. 0 \leq f\ N$ 
shows  $\text{suminf } f \leq \text{suminf } g$ 
by (meson assms order-trans suminf-le summable-ereal-pos)

lemma suminf-half-series-ereal:  $(\sum n. (1/2 :: \text{ereal}) \wedge \text{Suc } n) = 1$ 
using sums-ereal[THEN iffD2, OF power-half-series, THEN sums-unique, symmetric]
by (simp add: one-ereal-def)

lemma suminf-add-ereal:
  fixes  $f\ g :: \text{nat} \Rightarrow \text{ereal}$ 
assumes  $\bigwedge i. 0 \leq f\ i \wedge i. 0 \leq g\ i$ 
shows  $(\sum i. f\ i + g\ i) = \text{suminf } f + \text{suminf } g$ 
proof –
  have  $(\text{SUP } n. \sum i < n. f\ i + g\ i) = (\text{SUP } n. \text{sum } f\ \{..<n\}) + (\text{SUP } n. \text{sum } g\ \{..<n\})$ 
  unfolding sum.distrib
  by (intro assms add-nonneg-nonneg SUP-ereal-add-pos incseq-sumI sum-nonneg ballI)
  with assms show ?thesis
  by (simp add: suminf-ereal-eq-SUP)
qed

lemma suminf-cmult-ereal:
  fixes  $f\ g :: \text{nat} \Rightarrow \text{ereal}$ 
assumes  $\bigwedge i. 0 \leq f\ i$  and  $0 \leq a$ 
shows  $(\sum i. a * f\ i) = a * \text{suminf } f$ 
by (simp add: assms sum-nonneg suminf-ereal-eq-SUP sum-ereal-right-distrib flip: SUP-ereal-mult-left)

lemma suminf-PInf:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
assumes  $\bigwedge i. 0 \leq f\ i$ 
and  $\text{suminf } f \neq \infty$ 
shows  $f\ i \neq \infty$ 
proof –
  from suminf-upper[of f Suc i, OF assms(1)] assms(2)
  have  $(\sum i < \text{Suc } i. f\ i) \neq \infty$ 
  by auto
  then show ?thesis
  unfolding sum-PInf by simp
qed

lemma suminf-PInf-fun:
  assumes  $\bigwedge i. 0 \leq f\ i$ 
and  $\text{suminf } f \neq \infty$ 
shows  $\exists f'. f = (\lambda x. \text{ereal } (f'\ x))$ 
proof –

```

```

have  $\forall i. \exists r. f\ i = \text{ereal } r$ 
  by (metis abs-ereal-ge0 abs-neq-infinity-cases assms suminf-PInfy)
then show ?thesis
  by metis
qed

```

```

lemma summable-ereal:
  assumes  $\bigwedge i. 0 \leq f\ i$ 
  and  $(\sum i. \text{ereal } (f\ i)) \neq \infty$ 
  shows summable f
proof -
  have  $0 \leq (\sum i. \text{ereal } (f\ i))$ 
  using assms by (intro suminf-0-le) auto
  with assms obtain r where  $r: (\sum i. \text{ereal } (f\ i)) = \text{ereal } r$ 
  by (cases  $\sum i. \text{ereal } (f\ i)$ ) auto
  from summable-ereal-pos[of  $\lambda x. \text{ereal } (f\ x)$ ]
  have summable  $(\lambda x. \text{ereal } (f\ x))$ 
  using assms by auto
  from summable-sums[OF this]
  have  $(\lambda x. \text{ereal } (f\ x)) \text{ sums } (\sum x. \text{ereal } (f\ x))$ 
  by auto
  then show summable f
    unfolding r sums-ereal summable-def ..
qed

```

```

lemma suminf-ereal:
  assumes  $\bigwedge i. 0 \leq f\ i$ 
  and  $(\sum i. \text{ereal } (f\ i)) \neq \infty$ 
  shows  $(\sum i. \text{ereal } (f\ i)) = \text{ereal } (\text{suminf } f)$ 
proof (rule sums-unique[symmetric])
  from summable-ereal[OF assms]
  show  $(\lambda x. \text{ereal } (f\ x)) \text{ sums } (\text{ereal } (\text{suminf } f))$ 
  unfolding sums-ereal
  using assms
  by (intro summable-sums summable-ereal)
qed

```

```

lemma suminf-ereal-minus:
  fixes f g :: nat  $\Rightarrow$  ereal
  assumes ord:  $\bigwedge i. g\ i \leq f\ i \wedge i. 0 \leq g\ i$ 
  and fin:  $\text{suminf } f \neq \infty \wedge \text{suminf } g \neq \infty$ 
  shows  $(\sum i. f\ i - g\ i) = \text{suminf } f - \text{suminf } g$ 
proof -
  have 0:  $0 \leq f\ i$  for i
  using ord order-trans by blast
  moreover
  obtain f' where [simp]:  $f = (\lambda x. \text{ereal } (f'\ x))$ 
  using 0 fin(1) suminf-PInfy-fun by presburger
  obtain g' where [simp]:  $g = (\lambda x. \text{ereal } (g'\ x))$ 

```

```

    using fin(2) ord(2) suminf-PInfty-fun by presburger
  have  $0 \leq f\ i - g\ i$  for  $i$ 
    using ord(1) by auto
  moreover
  have suminf  $(\lambda i. f\ i - g\ i) \leq \text{suminf } f$ 
    using assms by (auto intro!: suminf-le-pos simp: field-simps)
  then have suminf  $(\lambda i. f\ i - g\ i) \neq \infty$ 
    using fin by auto
  ultimately show ?thesis
    using assms  $\langle \bigwedge i. 0 \leq f\ i \rangle$ 
    apply simp
    apply (subst (1 2 3) suminf-ereal)
    apply (auto intro!: suminf-diff[symmetric] summable-ereal)
    done
qed

```

```

lemma suminf-ereal-PInf [simp]:  $(\sum x. \infty::ereal) = \infty$ 
  by (metis ereal-less-eq(1) suminf-PInfty)

```

```

lemma summable-real-of-ereal:

```

```

  fixes  $f :: nat \Rightarrow ereal$ 
  assumes  $f: \bigwedge i. 0 \leq f\ i$ 
    and  $\text{fin}: (\sum i. f\ i) \neq \infty$ 
  shows summable  $(\lambda i. \text{real-of-ereal } (f\ i))$ 
proof (rule summable-def[THEN iffD2])
  have  $0 \leq (\sum i. f\ i)$ 
    using assms by (auto intro: suminf-0-le)
  with fin obtain  $r$  where  $r: ereal\ r = (\sum i. f\ i)$ 
    by (cases  $(\sum i. f\ i)$ ) auto
  have  $\text{fin}: |f\ i| \neq \infty$  for  $i$ 
    by (simp add: assms(2) f suminf-PInfty)
  have  $(\lambda i. ereal\ (\text{real-of-ereal } (f\ i))) \text{ sums } (\sum i. ereal\ (\text{real-of-ereal } (f\ i)))$ 
    using f
    by (auto intro!: summable-ereal-pos simp: ereal-le-real-iff zero-ereal-def)
  also have  $\dots = ereal\ r$ 
    using fin r by (auto simp: ereal-real)
  finally show  $\exists r. (\lambda i. \text{real-of-ereal } (f\ i)) \text{ sums } r$ 
    by (auto simp: sums-ereal)
qed

```

```

lemma suminf-SUP-eq:

```

```

  fixes  $f :: nat \Rightarrow nat \Rightarrow ereal$ 
  assumes  $\bigwedge i. \text{incseq } (\lambda n. f\ n\ i)$ 
    and  $\bigwedge n\ i. 0 \leq f\ n\ i$ 
  shows  $(\sum i. \text{SUP } n. f\ n\ i) = (\text{SUP } n. \sum i. f\ n\ i)$ 
proof -
  have *:  $\bigwedge n. (\sum i < n. \text{SUP } k. f\ k\ i) = (\text{SUP } k. \sum i < n. f\ k\ i)$ 
    using assms
    by (auto intro!: SUP-ereal-sum [symmetric])

```



```

show ?thesis
  using assms
  by (auto simp: suminf-ereal-eq-SUP SUP-upper2 * intro!: SUP-commute)
qed

```

```

lemma suminf-sum-ereal:
  fixes f ::  $\alpha \Rightarrow \text{ereal}$ 
  assumes nonneg:  $\bigwedge i. a \in A \implies 0 \leq f\ i\ a$ 
  shows  $(\sum i. \sum a \in A. f\ i\ a) = (\sum a \in A. \sum i. f\ i\ a)$ 
  using nonneg
  by (induction A rule: infinite-finite-induct; simp add: suminf-add-ereal sum-nonneg)

```

```

lemma suminf-ereal-eq-0:
  fixes f ::  $\text{nat} \Rightarrow \text{ereal}$ 
  assumes nneg:  $\bigwedge i. 0 \leq f\ i$ 
  shows  $(\sum i. f\ i) = 0 \iff (\forall i. f\ i = 0)$ 
proof
  assume  $(\sum i. f\ i) = 0$ 
  {
    fix i
    assume  $f\ i \neq 0$ 
    with nneg have  $0 < f\ i$ 
    by (auto simp: less-le)
    also have  $f\ i = (\sum j. \text{if } j = i \text{ then } f\ i \text{ else } 0)$ 
    by (subst suminf-finite[where N={i}]) auto
    also have  $\dots \leq (\sum i. f\ i)$ 
    using nneg
    by (auto intro!: suminf-le-pos)
    finally have False
    using  $\langle (\sum i. f\ i) = 0 \rangle$  by auto
  }
  then show  $\forall i. f\ i = 0$ 
  by auto
qed simp

```

```

lemma suminf-ereal-offset-le:
  fixes f ::  $\text{nat} \Rightarrow \text{ereal}$ 
  assumes f:  $\bigwedge i. 0 \leq f\ i$ 
  shows  $(\sum i. f\ (i + k)) \leq \text{suminf } f$ 
proof -
  have  $(\lambda n. \sum i < n. f\ (i + k)) \longrightarrow (\sum i. f\ (i + k))$ 
  using summable-sums[OF summable-ereal-pos]
  by (simp add: sums-def atLeast0LessThan f)
  moreover have  $(\lambda n. \sum i < n. f\ i) \longrightarrow (\sum i. f\ i)$ 
  using summable-sums[OF summable-ereal-pos]
  by (simp add: sums-def atLeast0LessThan f)
  then have  $(\lambda n. \sum i < n + k. f\ i) \longrightarrow (\sum i. f\ i)$ 
  by (rule LIMSEQ-ignore-initial-segment)
  ultimately show ?thesis

```

```

proof (rule LIMSEQ-le, safe intro!: exI[of - k])
  fix  $n$  assume  $k \leq n$ 
  have  $(\sum i < n. f (i + k)) = (\sum i < n. (f \circ \text{plus } k) i)$ 
    by (simp add: ac-simps)
  also have  $\dots = (\sum i \in (\text{plus } k) ' \{.. < n\}. f i)$ 
    by (rule sum.reindex [symmetric]) simp
  also have  $\dots \leq \text{sum } f \{.. < n + k\}$ 
    by (intro sum-mono2) (auto simp: f)
  finally show  $(\sum i < n. f (i + k)) \leq \text{sum } f \{.. < n + k\} .$ 
qed
qed

lemma sums-suminf-ereal:  $f \text{ sums } x \implies (\sum i. \text{ereal } (f i)) = \text{ereal } x$ 
  by (metis sums-ereal sums-unique)

lemma suminf-ereal':  $\text{summable } f \implies (\sum i. \text{ereal } (f i)) = \text{ereal } (\sum i. f i)$ 
  by (metis sums-ereal sums-unique summable-def)

lemma suminf-ereal-finite:  $\text{summable } f \implies (\sum i. \text{ereal } (f i)) \neq \infty$ 
  by (auto simp: summable-def simp flip: sums-ereal sums-unique)

lemma suminf-ereal-finite-neg:
  assumes summable  $f$ 
  shows  $(\sum x. \text{ereal } (f x)) \neq -\infty$ 
  by (simp add: asms suminf-ereal')

lemma SUP-ereal-add-directed:
  fixes  $f g :: 'a \Rightarrow \text{ereal}$ 
  assumes nonneg:  $\bigwedge i. i \in I \implies 0 \leq f i \wedge i \in I \implies 0 \leq g i$ 
  assumes directed:  $\bigwedge i j. i \in I \implies j \in I \implies \exists k \in I. f i + g j \leq f k + g k$ 
  shows  $(\text{SUP } i \in I. f i + g i) = (\text{SUP } i \in I. f i) + (\text{SUP } i \in I. g i)$ 
proof cases
  assume  $I = \{\}$  then show ?thesis
    by (simp add: bot-ereal-def)
next
  assume  $I \neq \{\}$ 
  show ?thesis
  proof (rule antisym)
    show  $(\text{SUP } i \in I. f i + g i) \leq (\text{SUP } i \in I. f i) + (\text{SUP } i \in I. g i)$ 
      by (rule SUP-least; intro add-mono SUP-upper)
    next
      have  $\text{bot} < (\text{SUP } i \in I. g i)$ 
        using  $\langle I \neq \{\} \rangle$  nonneg(2) by (auto simp: bot-ereal-def less-SUP-iff)
      then have  $(\text{SUP } i \in I. f i) + (\text{SUP } i \in I. g i) = (\text{SUP } i \in I. f i + (\text{SUP } i \in I. g i))$ 
        by (intro SUP-ereal-add-left[symmetric]  $\langle I \neq \{\} \rangle$ ) auto
      also have  $\dots = (\text{SUP } i \in I. (\text{SUP } j \in I. f i + g j))$ 
        using nonneg(1)  $\langle I \neq \{\} \rangle$  by (simp add: SUP-ereal-add-right)
      also have  $\dots \leq (\text{SUP } i \in I. f i + g i)$ 

```

```

    using directed by (intro SUP-least) (blast intro: SUP-upper2)
    finally show  $(\text{SUP } i \in I. f \ i) + (\text{SUP } i \in I. g \ i) \leq (\text{SUP } i \in I. f \ i + g \ i)$  .
  qed
qed

lemma SUP-ereal-sum-directed:
  fixes f g :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ereal
  assumes I  $\neq \{\}$ 
  assumes directed:  $\bigwedge N \ i \ j. N \subseteq A \implies i \in I \implies j \in I \implies \exists k \in I. \forall n \in N. f \ n \ i$ 
 $\leq f \ n \ k \wedge f \ n \ j \leq f \ n \ k$ 
  assumes nonneg:  $\bigwedge n \ i. i \in I \implies n \in A \implies 0 \leq f \ n \ i$ 
  shows  $(\text{SUP } i \in I. \sum n \in A. f \ n \ i) = (\sum n \in A. \text{SUP } i \in I. f \ n \ i)$ 
proof -
  have  $N \subseteq A \implies (\text{SUP } i \in I. \sum n \in N. f \ n \ i) = (\sum n \in N. \text{SUP } i \in I. f \ n \ i)$  for N
  proof (induction N rule: infinite-finite-induct)
    case (insert n N)
    have  $(\text{SUP } i \in I. f \ n \ i + (\sum l \in N. f \ l \ i)) = (\text{SUP } i \in I. f \ n \ i) + (\text{SUP } i \in I.$ 
 $\sum l \in N. f \ l \ i)$ 
  proof (rule SUP-ereal-add-directed)
    fix i assume  $i \in I$  then show  $0 \leq f \ n \ i \ 0 \leq (\sum l \in N. f \ l \ i)$ 
    using insert by (auto intro!: sum-nonneg nonneg)
  next
    fix i j assume  $i \in I \ j \in I$ 
    from directed[OF insert(4) this]
    show  $\exists k \in I. f \ n \ i + (\sum l \in N. f \ l \ j) \leq f \ n \ k + (\sum l \in N. f \ l \ k)$ 
    by (auto intro!: add-mono sum-mono)
  qed
  with insert show ?case
  by simp
qed (simp-all add: SUP-constant  $\langle I \neq \{\} \rangle$ )
from this[of A] show ?thesis by simp
qed

```

```

lemma suminf-SUP-eq-directed:
  fixes f :: -  $\Rightarrow$  nat  $\Rightarrow$  ereal
  assumes I  $\neq \{\}$ 
  assumes directed:  $\bigwedge N \ i \ j. i \in I \implies j \in I \implies \text{finite } N \implies \exists k \in I. \forall n \in N. f \ i \ n$ 
 $\leq f \ k \ n \wedge f \ j \ n \leq f \ k \ n$ 
  assumes nonneg:  $\bigwedge n \ i. 0 \leq f \ n \ i$ 
  shows  $(\sum i. \text{SUP } n \in I. f \ n \ i) = (\text{SUP } n \in I. \sum i. f \ n \ i)$ 
proof (subst (1 2) suminf-ereal-eq-SUP)
  show  $\bigwedge n \ i. 0 \leq f \ n \ i \wedge i. 0 \leq (\text{SUP } n \in I. f \ n \ i)$ 
  using  $\langle I \neq \{\} \rangle$  nonneg by (auto intro: SUP-upper2)
  show  $(\text{SUP } n. \sum i < n. \text{SUP } n \in I. f \ n \ i) = (\text{SUP } n \in I. \text{SUP } j. \sum i < j. f \ n \ i)$ 
  by (auto simp: finite-subset SUP-commute SUP-ereal-sum-directed assms)
qed

```

```

lemma ereal-dense3:
  fixes x y :: ereal

```

```

shows  $x < y \implies \exists r :: \text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$ 
proof (cases  $x \ y$  rule: ereal2-cases, simp-all)
  fix  $r \ q :: \text{real}$ 
  assume  $r < q$ 
  from Rats-dense-in-real[OF this] show  $\exists x. r < \text{real-of-rat } x \wedge \text{real-of-rat } x < q$ 
    by (fastforce simp: Rats-def)
next
  fix  $r :: \text{real}$ 
  show  $\exists x. r < \text{real-of-rat } x \ \exists x. \text{real-of-rat } x < r$ 
    using gt-ex[of  $r$ ] lt-ex[of  $r$ ] Rats-dense-in-real
    by (auto simp: Rats-def)
qed

lemma continuous-within-ereal[intro, simp]:  $x \in A \implies \text{continuous (at } x \text{ within } A)$ 
  ereal
  using continuous-on-eq-continuous-within[of  $A$  ereal]
  by (auto intro: continuous-on-ereal continuous-on-id)

lemma ereal-open-uminus:
  fixes  $S :: \text{ereal set}$ 
  assumes open  $S$ 
  shows open (uminus ‘  $S$ )
  using ‘open  $S$ ’[unfolded open-generated-order]
proof induct
  have range uminus = (UNIV :: ereal set)
    by (auto simp: image-iff ereal-uminus-eq-reorder)
  then show open (range uminus :: ereal set)
    by simp
qed (auto simp: image-Union image-Int)

lemma ereal-uminus-complement:
  fixes  $S :: \text{ereal set}$ 
  shows uminus ‘ ( $- \ S$ ) =  $- \ \text{uminus} \ ' S$ 
  by (auto intro!: bij-image-Compl-eq surjI[of  $- \ \text{uminus}$ ] simp: bij-betw-def)

lemma ereal-closed-uminus:
  fixes  $S :: \text{ereal set}$ 
  assumes closed  $S$ 
  shows closed (uminus ‘  $S$ )
  using assms
  unfolding closed-def ereal-uminus-complement[symmetric]
  by (rule ereal-open-uminus)

lemma ereal-open-affinity-pos:
  fixes  $S :: \text{ereal set}$ 
  assumes open  $S$ 
  and  $m: m \neq \infty \ 0 < m$ 
  and  $t: |t| \neq \infty$ 
  shows open ( $(\lambda x. m * x + t) \ ' S$ )

```

```

proof –
  have continuous-on UNIV ( $\lambda x. \text{inverse } m * (x + - t)$ )
    using m t
    by (intro continuous-at-imp-continuous-on ballI continuous-at[THEN iffD2];
force)
  then have open ( $(\lambda x. \text{inverse } m * (x + - t)) - ' S$ )
    using  $\langle \text{open } S \rangle \text{ open-vimage}$  by blast
  also have  $(\lambda x. \text{inverse } m * (x + - t)) - ' S = (\lambda x. (x - t) / m) - ' S$ 
    using m t by (auto simp: divide-ereal-def mult.commute minus-ereal-def
simp flip: uminus-ereal.simps)
  also have  $(\lambda x. (x - t) / m) - ' S = (\lambda x. m * x + t) - ' S$ 
    using m t
    by (simp add: set-eq-iff image-iff)
    (metis abs-ereal-less0 abs-ereal-uminus ereal-divide-eq ereal-eq-minus ereal-minus(7,8)
ereal-minus-less-minus ereal-mult-eq-PIfty ereal-uminus-uminus
ereal-zero-mult)
  finally show ?thesis .
qed

```

```

lemma ereal-open-affinity:
  fixes S :: ereal set
  assumes open S
  and m:  $|m| \neq \infty$  m  $\neq 0$ 
  and t:  $|t| \neq \infty$ 
  shows open ( $(\lambda x. m * x + t) - ' S$ )
proof cases
  assume  $0 < m$ 
  then show ?thesis
    using ereal-open-affinity-pos[OF  $\langle \text{open } S \rangle$  - - t, of m] m
    by auto
next
  assume  $\neg 0 < m$  then
  have  $0 < -m$ 
    using  $\langle m \neq 0 \rangle$ 
    by (cases m) auto
  then have m:  $-m \neq \infty$   $0 < -m$ 
    using  $\langle |m| \neq \infty \rangle$ 
    by (auto simp: ereal-uminus-eq-reorder)
  from ereal-open-affinity-pos[OF ereal-open-uminus[OF  $\langle \text{open } S \rangle$ ] m t] show ?thesis
    unfolding image-image by simp
qed

```

```

lemma open-uminus-iff:
  fixes S :: ereal set
  shows open (uminus - ' S)  $\longleftrightarrow$  open S
  using ereal-open-uminus[of S] ereal-open-uminus[of uminus - ' S]
  by auto

```

```

lemma ereal-Liminf-uminus:

```

fixes $f :: 'a \Rightarrow \text{ereal}$
shows $\text{Liminf net } (\lambda x. - (f x)) = - \text{Limsup net } f$
using $\text{ereal-Limsup-uminus}[of - (\lambda x. - (f x))]$ **by** *auto*

lemma *Liminf-PInfy*:
fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $\neg \text{trivial-limit net}$
shows $(f \longrightarrow \infty) \text{ net} \longleftrightarrow \text{Liminf net } f = \infty$
unfolding $\text{tendsto-iff-Liminf-eq-Limsup}[OF \text{ assms}]$
using $\text{Liminf-le-Limsup}[OF \text{ assms}, of f]$
by *auto*

lemma *Limsup-MInfy*:
fixes $f :: 'a \Rightarrow \text{ereal}$
assumes $\neg \text{trivial-limit net}$
shows $(f \longrightarrow -\infty) \text{ net} \longleftrightarrow \text{Limsup net } f = -\infty$
unfolding $\text{tendsto-iff-Liminf-eq-Limsup}[OF \text{ assms}]$
using $\text{Liminf-le-Limsup}[OF \text{ assms}, of f]$
by *auto*

lemma *convergent-ereal*: — RENAME
fixes $X :: \text{nat} \Rightarrow 'a :: \{\text{complete-linorder}, \text{linorder-topology}\}$
shows $\text{convergent } X \longleftrightarrow \text{limsup } X = \text{liminf } X$
using $\text{tendsto-iff-Liminf-eq-Limsup}[of \text{ sequentially}]$
by $(\text{auto simp: convergent-def})$

lemma *limsup-le-liminf-real*:
fixes $X :: \text{nat} \Rightarrow \text{real}$ **and** $L :: \text{real}$
assumes $1: \text{limsup } X \leq L$ **and** $2: L \leq \text{liminf } X$
shows $X \longrightarrow L$
proof —
from $1\ 2$ **have** $\text{limsup } X \leq \text{liminf } X$ **by** *auto*
hence $3: \text{limsup } X = \text{liminf } X$
by $(\text{simp add: Liminf-le-Limsup order-class.order.antisym})$
hence $4: \text{convergent } (\lambda n. \text{ereal } (X n))$
by $(\text{subst convergent-ereal})$
hence $\text{limsup } X = \lim (\lambda n. \text{ereal } (X n))$
by $(\text{rule convergent-limsup-cl})$
also from $1\ 2\ 3$ **have** $\text{limsup } X = L$ **by** *auto*
finally have $\lim (\lambda n. \text{ereal } (X n)) = L$ **..**
hence $(\lambda n. \text{ereal } (X n)) \longrightarrow L$
using 4 $\text{convergent-LIMSEQ-iff}$ **by** *force*
thus $?thesis$ **by** *simp*
qed

lemma *liminf-PInfy*:
fixes $X :: \text{nat} \Rightarrow \text{ereal}$
shows $X \longrightarrow \infty \longleftrightarrow \text{liminf } X = \infty$
by $(\text{metis Liminf-PInfy trivial-limit-sequentially})$

```

lemma limsup-MInfty:
  fixes  $X :: \text{nat} \Rightarrow \text{ereal}$ 
  shows  $X \longrightarrow -\infty \iff \text{limsup } X = -\infty$ 
  by (metis Limsup-MInfty trivial-limit-sequentially)

lemma SUP-eq-LIMSEQ:
  assumes mono f
  shows  $(\text{SUP } n. \text{ereal } (f \ n)) = \text{ereal } x \iff f \longrightarrow x$ 
proof
  have inc: incseq ( $\lambda i. \text{ereal } (f \ i)$ )
    using  $\langle \text{mono } f \rangle$  unfolding mono-def incseq-def by auto
  {
    assume  $f \longrightarrow x$ 
    then have  $(\lambda i. \text{ereal } (f \ i)) \longrightarrow \text{ereal } x$ 
      by auto
    from SUP-Lim[OF inc this] show  $(\text{SUP } n. \text{ereal } (f \ n)) = \text{ereal } x$  .
  }
  next
    assume  $(\text{SUP } n. \text{ereal } (f \ n)) = \text{ereal } x$ 
    with LIMSEQ-SUP[OF inc] show  $f \longrightarrow x$  by auto
  }
qed

lemma liminf-ereal-cminus:
  fixes  $f :: \text{nat} \Rightarrow \text{ereal}$ 
  assumes  $c \neq -\infty$ 
  shows  $\text{liminf } (\lambda x. c - f \ x) = c - \text{limsup } f$ 
proof (cases c)
  case PIInf
  then show ?thesis
    by (simp add: Liminf-const)
  next
    case (real r)
    then show ?thesis
      by (simp add: liminf-SUP-INF limsup-INF-SUP INF-ereal-minus-right SUP-ereal-minus-right)
qed (use  $\langle c \neq -\infty \rangle$  in simp)

```

38.6.3 Continuity

```

lemma continuous-at-of-ereal:
   $|x0 :: \text{ereal}| \neq \infty \implies \text{continuous } (\text{at } x0) \text{ real-of-ereal}$ 
  unfolding continuous-at
  by (rule lim-real-of-ereal) (simp add: ereal-real)

lemma nhds-ereal:  $\text{nhds } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{nhds } r)$ 
  by (simp add: filtermap-nhds-open-map open-ereal continuous-at-of-ereal)

lemma at-ereal:  $\text{at } (\text{ereal } r) = \text{filtermap } \text{ereal } (\text{at } r)$ 
  by (simp add: filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap)

```

lemma *at-left-ereal*: $at_left\ (ereal\ r) = filtermap\ ereal\ (at_left\ r)$
by (*simp add: filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma *at-right-ereal*: $at_right\ (ereal\ r) = filtermap\ ereal\ (at_right\ r)$
by (*simp add: filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma
shows *at-left-PInf*: $at_left\ \infty = filtermap\ ereal\ at_top$
and *at-right-MInf*: $at_right\ (-\infty) = filtermap\ ereal\ at_bot$
unfolding *filter-eq-iff eventually-filtermap eventually-at-top-dense eventually-at-bot-dense eventually-at-left[OF ereal-less(5)] eventually-at-right[OF ereal-less(6)]*
by (*auto simp: ereal-all-split ereal-ex-split*)

lemma *ereal-tendsto-simps1*:
 $((f \circ real_of_ereal) \longrightarrow y)\ (at_left\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at_left\ x)$
 $((f \circ real_of_ereal) \longrightarrow y)\ (at_right\ (ereal\ x)) \longleftrightarrow (f \longrightarrow y)\ (at_right\ x)$
 $((f \circ real_of_ereal) \longrightarrow y)\ (at_left\ (\infty::ereal)) \longleftrightarrow (f \longrightarrow y)\ at_top$
 $((f \circ real_of_ereal) \longrightarrow y)\ (at_right\ (-\infty::ereal)) \longleftrightarrow (f \longrightarrow y)\ at_bot$
unfolding *tendsto-compose-filtermap at-left-ereal at-right-ereal at-left-PInf at-right-MInf*
by (*auto simp: filtermap-filtermap filtermap-ident*)

lemma *ereal-tendsto-simps2*:
 $((ereal \circ f) \longrightarrow ereal\ a)\ F \longleftrightarrow (f \longrightarrow a)\ F$
 $((ereal \circ f) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x\ :=>\ at_top)$
 $((ereal \circ f) \longrightarrow -\infty)\ F \longleftrightarrow (LIM\ x\ F.\ f\ x\ :=>\ at_bot)$
unfolding *tendsto-PInfy filterlim-at-top-dense tendsto-MInfy filterlim-at-bot-dense*
using *lim-ereal* **by** (*simp-all add: comp-def*)

lemma *inverse-infty-ereal-tendsto-0*: $inverse\ -\infty \rightarrow (0::ereal)$

proof –

have **: $((\lambda x.\ ereal\ (inverse\ x)) \longrightarrow ereal\ 0)\ at_infinity$

by (*intro tendsto-intros tendsto-inverse-0*)

then have $((\lambda x.\ if\ x = 0\ then\ \infty\ else\ ereal\ (inverse\ x)) \longrightarrow 0)\ at_top$

proof (*rule filterlim-mono-eventually*)

show $nhds\ (ereal\ 0) \leq nhds\ 0$

by (*simp add: zero-ereal-def*)

show $(at_top::real\ filter) \leq at_infinity$

by (*simp add: at-top-le-at-infinity*)

qed *auto*

then show *?thesis*

unfolding *at-infty-ereal-eq-at-top tendsto-compose-filtermap[symmetric] comp-def*

by *auto*

qed

lemma *inverse-ereal-tendsto-pos*:

fixes $x :: ereal$ **assumes** $0 < x$

shows $inverse\ -x \rightarrow inverse\ x$

proof (*cases x*)


```

case (real r)
with  $\langle 0 < x \rangle$  have **:  $(\lambda x. \text{ereal } (\text{inverse } x)) -r \rightarrow \text{ereal } (\text{inverse } r)$ 
  by (auto intro!: tendsto-inverse)
from real  $\langle 0 < x \rangle$  show ?thesis
  by (auto simp: at-ereal tendsto-compose-filtermap[symmetric] eventually-at-filter
    intro!: Lim-transform-eventually[OF **] t1-space-nhds)
qed (insert  $\langle 0 < x \rangle$ , auto intro!: inverse-infty-ereal-tendsto-0)

lemma inverse-ereal-tendsto-at-right-0:  $(\text{inverse} \longrightarrow \infty) (\text{at-right } (0::\text{ereal}))$ 
  unfolding tendsto-compose-filtermap[symmetric] at-right-ereal zero-ereal-def
  by (subst filterlim-cong[OF refl refl, where  $g = \lambda x. \text{ereal } (\text{inverse } x)$ ])
  (auto simp: eventually-at-filter tendsto-PInfty-eq-at-top filterlim-inverse-at-top-right)

lemmas ereal-tendsto-simps = ereal-tendsto-simps1 ereal-tendsto-simps2

lemma continuous-at-iff-ereal:
  fixes  $f :: 'a::t2\text{-space} \Rightarrow \text{real}$ 
  shows  $\text{continuous } (\text{at } x0 \text{ within } s) f \longleftrightarrow \text{continuous } (\text{at } x0 \text{ within } s) (\text{ereal} \circ f)$ 
  unfolding continuous-within comp-def lim-ereal ..

lemma continuous-on-iff-ereal:
  fixes  $f :: 'a::t2\text{-space} \Rightarrow \text{real}$ 
  assumes open A
  shows  $\text{continuous-on } A f \longleftrightarrow \text{continuous-on } A (\text{ereal} \circ f)$ 
  unfolding continuous-on-def comp-def lim-ereal ..

lemma continuous-on-real:  $\text{continuous-on } (\text{UNIV} - \{\infty, -\infty::\text{ereal}\}) \text{real-of-ereal}$ 
  using continuous-at-of-ereal continuous-on-eq-continuous-at open-image-ereal
  by auto

lemma continuous-on-iff-real:
  fixes  $f :: 'a::t2\text{-space} \Rightarrow \text{ereal}$ 
  assumes  $\bigwedge x. x \in A \implies |f x| \neq \infty$ 
  shows  $\text{continuous-on } A f \longleftrightarrow \text{continuous-on } A (\text{real-of-ereal} \circ f)$ 
proof
  assume L:  $\text{continuous-on } A f$ 
  have  $f' : A \subseteq \text{UNIV} - \{\infty, -\infty\}$ 
    using assms by force
  then show  $\text{continuous-on } A (\text{real-of-ereal} \circ f)$ 
    by (meson L continuous-on-compose continuous-on-real continuous-on-subset)
next
  assume R:  $\text{continuous-on } A (\text{real-of-ereal} \circ f)$ 
  then have  $\text{continuous-on } A (\text{ereal} \circ (\text{real-of-ereal} \circ f))$ 
    by (meson continuous-at-iff-ereal continuous-on-eq-continuous-within)
  then show  $\text{continuous-on } A f$ 
    using assms ereal-real' by auto
qed

lemma continuous-uminus-ereal [continuous-intros]:  $\text{continuous-on } (A :: \text{ereal set})$ 

```

uminus

unfolding *continuous-on-def*
by (*intro ballI tendsto-uminus-ereal*[*of* $\lambda x. x :: \text{ereal}$]) *simp*

lemma *ereal-uminus-atMost* [*simp*]: *uminus* ‘ $\{..(a :: \text{ereal})\} = \{-a..\}$

proof (*intro equalityI subsetI*)

fix $x :: \text{ereal}$ **assume** $x \in \{-a..\}$

hence $-(-x) \in \text{uminus } \{..a\}$ **by** (*intro imageI*) (*simp add: ereal-uminus-le-reorder*)

thus $x \in \text{uminus } \{..a\}$ **by** *simp*

qed *auto*

lemma *continuous-on-inverse-ereal* [*continuous-intros*]:

continuous-on $\{0 :: \text{ereal} ..\}$ *inverse*

unfolding *continuous-on-def*

proof *clarsimp*

fix $x :: \text{ereal}$ **assume** $0 \leq x$

moreover **have** *at* 0 *within* $\{0 ..\} = \text{at-right } (0 :: \text{ereal})$

by (*auto simp: filter-eq-iff eventually-at-filter le-less*)

moreover **have** *at* x *within* $\{0 ..\} = \text{at } x$ **if** $0 < x$

using *that* **by** (*intro at-within-nhd*[*of* $\{0 < ..\}$]) *auto*

ultimately **show** (*inverse* \longrightarrow *inverse* x) (*at* x *within* $\{0..\}$)

by (*auto simp: le-less inverse-ereal-tendsto-at-right-0 inverse-ereal-tendsto-pos*)

qed

lemma *continuous-inverse-ereal-nonpos*: *continuous-on* $(\{..<0\} :: \text{ereal set})$ *inverse*

proof (*subst continuous-on-cong*[*OF refl*])

have *continuous-on* $\{(0 :: \text{ereal}) < ..\}$ *inverse*

by (*rule continuous-on-subset*[*OF continuous-on-inverse-ereal*]) *auto*

thus *continuous-on* $\{..<(0 :: \text{ereal})\}$ (*uminus* \circ *inverse* \circ *uminus*)

by (*intro continuous-intros*) *simp-all*

qed *simp*

lemma *tendsto-inverse-ereal*:

assumes ($f \longrightarrow (c :: \text{ereal})$) F

assumes *eventually* $(\lambda x. f x \geq 0)$ F

shows $((\lambda x. \text{inverse } (f x)) \longrightarrow \text{inverse } c)$ F

by (*cases* $F = \text{bot}$)

(*auto intro!: tendsto-lowerbound assms*

continuous-on-tendsto-compose[*OF continuous-on-inverse-ereal*])

38.6.4 liminf and limsup

lemma *Limsup-ereal-mult-right*:

assumes $F \neq \text{bot}$ $(c :: \text{real}) \geq 0$

shows $\text{Limsup } F (\lambda n. f n * \text{ereal } c) = \text{Limsup } F f * \text{ereal } c$

proof (*rule Limsup-compose-continuous-mono*)

from *assms* **show** *continuous-on UNIV* $(\lambda a. a * \text{ereal } c)$

using *tendsto-cmult-ereal*[*of* *ereal* c $\lambda x. x$]

by (force simp: continuous-on-def mult-ac)
 qed (use assms in ⟨auto simp: mono-def ereal-mult-right-mono⟩)

lemma *Liminf-ereal-mult-right*:
 assumes $F \neq \text{bot } (c::\text{real}) \geq 0$
 shows $\text{Liminf } F (\lambda n. f n * \text{ereal } c) = \text{Liminf } F f * \text{ereal } c$
proof (rule *Liminf-compose-continuous-mono*)
 from assms show continuous-on UNIV $(\lambda a. a * \text{ereal } c)$
 using tendsto-cmult-ereal[of $\text{ereal } c \lambda x. x$]
 by (force simp: continuous-on-def mult-ac)
 qed (use assms in ⟨auto simp: mono-def ereal-mult-right-mono⟩)

lemma *Liminf-ereal-mult-left*:
 assumes $F \neq \text{bot } (c::\text{real}) \geq 0$
 shows $\text{Liminf } F (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{Liminf } F f$
 using *Liminf-ereal-mult-right*[OF assms] by (subst (1 2) mult.commute)

lemma *Limsup-ereal-mult-left*:
 assumes $F \neq \text{bot } (c::\text{real}) \geq 0$
 shows $\text{Limsup } F (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{Limsup } F f$
 using *Limsup-ereal-mult-right*[OF assms] by (subst (1 2) mult.commute)

lemma *limsup-ereal-mult-right*:
 $(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. f n * \text{ereal } c) = \text{limsup } f * \text{ereal } c$
 by (rule *Limsup-ereal-mult-right*) simp-all

lemma *limsup-ereal-mult-left*:
 $(c::\text{real}) \geq 0 \implies \text{limsup } (\lambda n. \text{ereal } c * f n) = \text{ereal } c * \text{limsup } f$
 by (simp add: *Limsup-ereal-mult-left*)

lemma *Limsup-add-ereal-right*:
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. g n + (c :: \text{ereal})) = \text{Limsup } F g + c$
 by (rule *Limsup-compose-continuous-mono*) (auto simp: mono-def add-mono continuous-on-def)

lemma *Limsup-add-ereal-left*:
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Limsup } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Limsup } F g$
 by (subst (1 2) add.commute) (rule *Limsup-add-ereal-right*)

lemma *Liminf-add-ereal-right*:
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. g n + (c :: \text{ereal})) = \text{Liminf } F g + c$
 by (rule *Liminf-compose-continuous-mono*) (auto simp: mono-def add-mono continuous-on-def)

lemma *Liminf-add-ereal-left*:
 $F \neq \text{bot} \implies \text{abs } c \neq \infty \implies \text{Liminf } F (\lambda n. (c :: \text{ereal}) + g n) = c + \text{Liminf } F g$
 by (subst (1 2) add.commute) (rule *Liminf-add-ereal-right*)

lemma
assumes $F \neq \text{bot}$
assumes *nonneg*: *eventually* $(\lambda x. f\ x \geq (0 :: \text{ereal}))\ F$
shows *Liminf-inverse-ereal*: $\text{Liminf}\ F\ (\lambda x. \text{inverse}\ (f\ x)) = \text{inverse}\ (\text{Limsup}\ F\ f)$
and *Limsup-inverse-ereal*: $\text{Limsup}\ F\ (\lambda x. \text{inverse}\ (f\ x)) = \text{inverse}\ (\text{Liminf}\ F\ f)$
proof –
define *inv* **where** [*abs-def*]: $\text{inv}\ x = (\text{if } x \leq 0 \text{ then } \infty \text{ else } \text{inverse}\ x)$ **for** $x :: \text{ereal}$
have *continuous-on* $(\{..0\} \cup \{0..\})$ *inv* **unfolding** *inv-def*
by (*intro continuous-on-If*) (*auto intro!*: *continuous-intros*)
also have $\{..0\} \cup \{0..\} = (\text{UNIV} :: \text{ereal set})$ **by** *auto*
finally have *cont*: *continuous-on UNIV inv* .
have *antimono*: *antimono inv* **unfolding** *inv-def antimono-def*
by (*auto intro!*: *ereal-inverse-antimono*)

have $\text{Liminf}\ F\ (\lambda x. \text{inverse}\ (f\ x)) = \text{Liminf}\ F\ (\lambda x. \text{inv}\ (f\ x))$ **using** *nonneg*
by (*auto intro!*: *Liminf-eq elim!*: *eventually-mono simp: inv-def*)
also have $\dots = \text{inv}\ (\text{Limsup}\ F\ f)$
by (*simp add: assms(1) Liminf-compose-continuous-antimono[OF cont anti-mono]*)
also from *assms* **have** $\text{Limsup}\ F\ f \geq 0$ **by** (*intro le-Limsup*) *simp-all*
hence $\text{inv}\ (\text{Limsup}\ F\ f) = \text{inverse}\ (\text{Limsup}\ F\ f)$ **by** (*simp add: inv-def*)
finally show $\text{Liminf}\ F\ (\lambda x. \text{inverse}\ (f\ x)) = \text{inverse}\ (\text{Limsup}\ F\ f)$.

have $\text{Limsup}\ F\ (\lambda x. \text{inverse}\ (f\ x)) = \text{Limsup}\ F\ (\lambda x. \text{inv}\ (f\ x))$ **using** *nonneg*
by (*auto intro!*: *Limsup-eq elim!*: *eventually-mono simp: inv-def*)
also have $\dots = \text{inv}\ (\text{Liminf}\ F\ f)$
by (*simp add: assms(1) Limsup-compose-continuous-antimono[OF cont anti-mono]*)
also from *assms* **have** $\text{Liminf}\ F\ f \geq 0$ **by** (*intro Liminf-bounded*) *simp-all*
hence $\text{inv}\ (\text{Liminf}\ F\ f) = \text{inverse}\ (\text{Liminf}\ F\ f)$ **by** (*simp add: inv-def*)
finally show $\text{Limsup}\ F\ (\lambda x. \text{inverse}\ (f\ x)) = \text{inverse}\ (\text{Liminf}\ F\ f)$.
qed

lemma *ereal-diff-le-mono-left*: $\llbracket x \leq z; 0 \leq y \rrbracket \implies x - y \leq (z :: \text{ereal})$
by(*cases x y z rule: ereal3-cases*) *simp-all*

lemma *neg-0-less-iff-less-erea* [*simp*]: $0 < -a \longleftrightarrow (a :: \text{ereal}) < 0$
by(*cases a*) *simp-all*

lemma *not-infty-ereal*: $|x| \neq \infty \longleftrightarrow (\exists x'. x = \text{ereal } x')$
by *auto*

lemma *neg-PInf-trans*: **fixes** $x\ y :: \text{ereal}$ **shows** $\llbracket y \neq \infty; x \leq y \rrbracket \implies x \neq \infty$
by *auto*

lemma *mult-2-ereal*: $\text{ereal } 2 * x = x + x$
by(*cases x simp-all*)

lemma *ereal-diff-le-self*: $0 \leq y \implies x - y \leq (x :: \text{ereal})$
by(*cases x y rule: ereal2-cases simp-all*)

lemma *ereal-le-add-self*: $0 \leq y \implies x \leq x + (y :: \text{ereal})$
by(*cases x y rule: ereal2-cases simp-all*)

lemma *ereal-le-add-self2*: $0 \leq y \implies x \leq y + (x :: \text{ereal})$
by(*cases x y rule: ereal2-cases simp-all*)

lemma *ereal-diff-nonpos*:
fixes $a\ b :: \text{ereal}$ **shows** $\llbracket a \leq b; a = \infty \implies b \neq \infty; a = -\infty \implies b \neq -\infty \rrbracket$
 $\implies a - b \leq 0$
by (*cases rule: ereal2-cases[of a b] auto*)

lemma *minus-ereal-0* [*simp*]: $x - \text{ereal } 0 = x$
by(*simp flip: zero-ereal-def*)

lemma *ereal-diff-eq-0-iff*: **fixes** $a\ b :: \text{ereal}$
shows $(|a| = \infty \implies |b| \neq \infty) \implies a - b = 0 \longleftrightarrow a = b$
by(*cases a b rule: ereal2-cases simp-all*)

lemma *SUP-ereal-eq-0-iff-nonneg*:
fixes $f :: - \Rightarrow \text{ereal}$ **and** A
assumes *nonneg*: $\forall x \in A. f\ x \geq 0$
and $A:A \neq \{\}$
shows $(\text{SUP } x \in A. f\ x) = 0 \longleftrightarrow (\forall x \in A. f\ x = 0)$ (**is** *?lhs \longleftrightarrow -*)
proof(*intro iffI ballI*)
fix x
assume *?lhs* $x \in A$
from $\langle x \in A \rangle$ **have** $f\ x \leq (\text{SUP } x \in A. f\ x)$ **by**(*rule SUP-upper*)
with $\langle ?lhs \rangle$ **show** $f\ x = 0$ **using** *nonneg* $\langle x \in A \rangle$ **by** *auto*
qed (*simp add: A*)

lemma *ereal-divide-le-posI*:
fixes $x\ y\ z :: \text{ereal}$
shows $x > 0 \implies z \neq -\infty \implies z \leq x * y \implies z / x \leq y$
by (*cases rule: ereal3-cases[of x y z](auto simp: field-simps split: if-split-asm)*)

lemma *add-diff-eq-ereal*:
fixes $x\ y\ z :: \text{ereal}$
shows $x + (y - z) = x + y - z$
by(*cases x y z rule: ereal3-cases simp-all*)

lemma *ereal-diff-gr0*:
fixes $a\ b :: \text{ereal}$
shows $a < b \implies 0 < b - a$

by (*cases rule: ereal2-cases*[of *a b*]) *auto*

lemma *ereal-minus-minus*:

fixes *x y z :: ereal* **shows**

$(|y| = \infty \implies |z| \neq \infty) \implies x - (y - z) = x + z - y$

by(*cases x y z rule: ereal3-cases*) *simp-all*

lemma *diff-diff-commute-ereal*:

fixes *x y z :: ereal*

shows $x - y - z = x - z - y$

by (*metis add-diff-eq-ereal ereal-add-uminus-conv-diff*)

lemma *ereal-diff-eq-MInfty-iff*:

fixes *x y :: ereal*

shows $x - y = -\infty \longleftrightarrow x = -\infty \wedge y \neq -\infty \vee y = \infty \wedge |x| \neq \infty$

by(*cases x y rule: ereal2-cases*) *simp-all*

lemma *ereal-diff-add-inverse*:

fixes *x y :: ereal*

shows $|x| \neq \infty \implies x + y - x = y$

by(*cases x y rule: ereal2-cases*) *simp-all*

lemma *tendsto-diff-ereal*:

fixes *x y :: ereal*

assumes *x*: $|x| \neq \infty$ **and** *y*: $|y| \neq \infty$

assumes *f*: $(f \longrightarrow x) F$ **and** *g*: $(g \longrightarrow y) F$

shows $((\lambda x. f x - g x) \longrightarrow x - y) F$

proof –

from *x* **obtain** *r* **where** *x'*: $x = \text{ereal } r$ **by** (*cases x*) *auto*

with *f* **have** $((\lambda i. \text{real-of-ereal } (f i)) \longrightarrow r) F$ **by** *simp*

moreover

from *y* **obtain** *p* **where** *y'*: $y = \text{ereal } p$ **by** (*cases y*) *auto*

with *g* **have** $((\lambda i. \text{real-of-ereal } (g i)) \longrightarrow p) F$ **by** *simp*

ultimately have $((\lambda i. \text{real-of-ereal } (f i) - \text{real-of-ereal } (g i)) \longrightarrow r - p) F$

by (*rule tendsto-diff*)

moreover

from *eventually-finite*[*OF x f*] *eventually-finite*[*OF y g*]

have *eventually* $(\lambda x. f x - g x = \text{ereal } (\text{real-of-ereal } (f x) - \text{real-of-ereal } (g x)))$

F

by *eventually-elim auto*

ultimately show *?thesis*

by (*simp add: x' y' cong: filterlim-cong*)

qed

lemma *continuous-on-diff-ereal*:

continuous-on A f \implies *continuous-on A g* \implies $(\bigwedge x. x \in A \implies |f x| \neq \infty) \implies$

$(\bigwedge x. x \in A \implies |g x| \neq \infty) \implies \text{continuous-on } A (\lambda z. f z - g z :: \text{ereal})$

by (*auto simp: tendsto-diff-ereal continuous-on-def*)

38.6.5 Tests for code generator

A small list of simple arithmetic expressions.

```

value  $-\infty$  :: ereal
value  $|\infty|$  :: ereal
value  $4 + 5 / 4 - \text{ereal } 2$  :: ereal
value ereal 3 <  $\infty$ 
value real-of-ereal ( $\infty$ ::ereal) = 0

```

end

39 Indicator Function

```

theory Indicator-Function
imports Complex-Main Disjoint-Sets
begin

```

definition *indicator* $S\ x = \text{of-bool } (x \in S)$

Type constrained version

abbreviation *indicat-real* :: $'a\ \text{set} \Rightarrow 'a \Rightarrow \text{real}$ **where** *indicat-real* $S \equiv \text{indicator } S$

```

lemma indicator-simps[simp]:
   $x \in S \implies \text{indicator } S\ x = 1$ 
   $x \notin S \implies \text{indicator } S\ x = 0$ 
unfolding indicator-def by auto

```

```

lemma indicator-pos-le[intro, simp]:  $(0::'a::\text{linordered-semidom}) \leq \text{indicator } S\ x$ 
and indicator-le-1[intro, simp]:  $\text{indicator } S\ x \leq (1::'a::\text{linordered-semidom})$ 
unfolding indicator-def by auto

```

```

lemma indicator-abs-le-1:  $|\text{indicator } S\ x| \leq (1::'a::\text{linordered-idom})$ 
unfolding indicator-def by auto

```

```

lemma indicator-eq-0-iff:  $\text{indicator } A\ x = (0::'a::\text{zero-neq-one}) \longleftrightarrow x \notin A$ 
by (auto simp: indicator-def)

```

```

lemma indicator-eq-1-iff:  $\text{indicator } A\ x = (1::'a::\text{zero-neq-one}) \longleftrightarrow x \in A$ 
by (auto simp: indicator-def)

```

```

lemma indicator-UNIV [simp]:  $\text{indicator } \text{UNIV} = (\lambda x. 1)$ 
by auto

```

```

lemma indicator-leI:
   $(x \in A \implies y \in B) \implies (\text{indicator } A\ x :: 'a::\text{linordered-nonzero-semiring}) \leq$ 
   $\text{indicator } B\ y$ 
by (auto simp: indicator-def)

```

lemma *split-indicator*: $P \text{ (indicator } S \text{ } x) \longleftrightarrow ((x \in S \longrightarrow P \text{ } 1) \wedge (x \notin S \longrightarrow P \text{ } 0))$

unfolding *indicator-def* **by** *auto*

lemma *split-indicator-asm*: $P \text{ (indicator } S \text{ } x) \longleftrightarrow (\neg (x \in S \wedge \neg P \text{ } 1 \vee x \notin S \wedge \neg P \text{ } 0))$

unfolding *indicator-def* **by** *auto*

lemma *indicator-inter-arith*: $\text{indicator } (A \cap B) \text{ } x = \text{indicator } A \text{ } x * (\text{indicator } B \text{ } x :: 'a::\text{semiring-1})$

unfolding *indicator-def* **by** (*auto simp: min-def max-def*)

lemma *indicator-union-arith*:

$\text{indicator } (A \cup B) \text{ } x = \text{indicator } A \text{ } x + \text{indicator } B \text{ } x - \text{indicator } A \text{ } x * (\text{indicator } B \text{ } x :: 'a::\text{ring-1})$

unfolding *indicator-def* **by** (*auto simp: min-def max-def*)

lemma *indicator-inter-min*: $\text{indicator } (A \cap B) \text{ } x = \min (\text{indicator } A \text{ } x) (\text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

and *indicator-union-max*: $\text{indicator } (A \cup B) \text{ } x = \max (\text{indicator } A \text{ } x) (\text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

unfolding *indicator-def* **by** (*auto simp: min-def max-def*)

lemma *indicator-disj-union*:

$A \cap B = \{\} \implies \text{indicator } (A \cup B) \text{ } x = (\text{indicator } A \text{ } x + \text{indicator } B \text{ } x :: 'a::\text{linordered-semidom})$

by (*auto split: split-indicator*)

lemma *indicator-compl*: $\text{indicator } (\neg A) \text{ } x = 1 - (\text{indicator } A \text{ } x :: 'a::\text{ring-1})$

and *indicator-diff*: $\text{indicator } (A - B) \text{ } x = \text{indicator } A \text{ } x * (1 - \text{indicator } B \text{ } x :: 'a::\text{ring-1})$

unfolding *indicator-def* **by** (*auto simp: min-def max-def*)

lemma *indicator-times*:

$\text{indicator } (A \times B) \text{ } x = \text{indicator } A \text{ } (\text{fst } x) * (\text{indicator } B \text{ } (\text{snd } x) :: 'a::\text{semiring-1})$

unfolding *indicator-def* **by** (*cases x auto*)

lemma *indicator-sum*:

$\text{indicator } (A <+> B) \text{ } x = (\text{case } x \text{ of } \text{Inl } x \Rightarrow \text{indicator } A \text{ } x \mid \text{Inr } x \Rightarrow \text{indicator } B \text{ } x)$

unfolding *indicator-def* **by** (*cases x auto*)

lemma *indicator-image*: $\text{inj } f \implies \text{indicator } (f ' X) \text{ } (f \text{ } x) = (\text{indicator } X \text{ } x :: \text{zero-neq-one})$

by (*auto simp: indicator-def inj-def*)

lemma *indicator-vimage*: $\text{indicator } (f - ' A) \text{ } x = \text{indicator } A \text{ } (f \text{ } x)$

by (*auto split: split-indicator*)

lemma *mult-indicator-cong*:

fixes $f\ g :: - \Rightarrow 'a :: \text{semiring-1}$
shows $(\bigwedge x. x \in A \implies f\ x = g\ x) \implies \text{indicator } A\ x * f\ x = \text{indicator } A\ x * g\ x$
by $(\text{auto simp: indicator-def})$

lemma

fixes $f :: 'a \Rightarrow 'b :: \text{semiring-1}$
assumes $\text{finite } A$
shows $\text{sum-mult-indicator[simp]}: (\sum x \in A. f\ x * \text{indicator } B\ x) = (\sum x \in A \cap B. f\ x)$
and $\text{sum-indicator-mult[simp]}: (\sum x \in A. \text{indicator } B\ x * f\ x) = (\sum x \in A \cap B. f\ x)$
unfolding indicator-def
using $\text{assms by (auto intro!: sum.mono-neutral-cong-right split: if-split-asm)}$

lemma $\text{sum-indicator-eq-card}$:

assumes $\text{finite } A$
shows $(\sum x \in A. \text{indicator } B\ x) = \text{card } (A \text{ Int } B)$
using $\text{sum-mult-indicator [OF assms, of } \lambda x. 1::\text{nat}]}$
unfolding $\text{card-eq-sum by simp}$

lemma $\text{sum-indicator-scaleR[simp]}$:

$\text{finite } A \implies$
 $(\sum x \in A. \text{indicator } (B\ x)\ (g\ x) *_R f\ x) = (\sum x \in \{x \in A. g\ x \in B\ x\}. f\ x :: 'a :: \text{real-vector})$
by $(\text{auto intro!: sum.mono-neutral-cong-right split: if-split-asm simp: indicator-def})$

lemma $\text{LIMSEQ-indicator-incseq}$:

assumes $\text{incseq } A$
shows $(\lambda i. \text{indicator } (A\ i)\ x :: 'a :: \{\text{topological-space, zero-neq-one}\}) \longrightarrow \text{indicator } (\bigcup i. A\ i)\ x$
proof $(\text{cases } \exists i. x \in A\ i)$
case True
then obtain i **where** $x \in A\ i$
by auto
then have $*$:
 $\bigwedge n. (\text{indicator } (A\ (n + i))\ x :: 'a) = 1$
 $(\text{indicator } (\bigcup i. A\ i)\ x :: 'a) = 1$
using $\text{incseqD[OF } \langle \text{incseq } A \rangle, \text{ of } i\ n + i \text{ for } n] \langle x \in A\ i \rangle \text{ by (auto simp: indicator-def)}$
show $?thesis$
by $(\text{rule LIMSEQ-offset[of } -\ i])\ (\text{use } * \text{ in simp})$
next
case False
then show $?thesis$ **by** $(\text{simp add: indicator-def})$
qed

lemma $\text{LIMSEQ-indicator-UN}$:

$(\lambda k. \text{indicator } (\bigcup i < k. A\ i)\ x :: 'a :: \{\text{topological-space, zero-neq-one}\}) \longrightarrow \text{in-}$

indicator ($\bigcup i. A\ i$) x

proof –

have ($\lambda k. \text{indicator } (\bigcup i < k. A\ i) \ x :: 'a$) $\longrightarrow \text{indicator } (\bigcup k. \bigcup i < k. A\ i) \ x$
 by (*intro LIMSEQ-indicator-incseq*) (*auto simp: incseq-def intro: less-le-trans*)
 also have ($\bigcup k. \bigcup i < k. A\ i = \bigcup i. A\ i$)
 by *auto*
 finally show *?thesis* .

qed

lemma *LIMSEQ-indicator-decseq*:

assumes *decseq A*

shows ($\lambda i. \text{indicator } (A\ i) \ x :: 'a :: \{\text{topological-space, zero-neq-one}\}$) $\longrightarrow \text{indicator } (\bigcap i. A\ i) \ x$

proof (*cases* $\exists i. x \notin A\ i$)

case *True*

then obtain i **where** $x \notin A\ i$

by *auto*

then have $*$:

$\bigwedge n. (\text{indicator } (A\ (n + i)) \ x :: 'a) = 0$

$(\text{indicator } (\bigcap i. A\ i) \ x :: 'a) = 0$

using *decseqD[OF <decseq A>, of i n + i for n]* $\langle x \notin A\ i \rangle$ **by** (*auto simp: indicator-def*)

show *?thesis*

by (*rule LIMSEQ-offset[of - i]*) (*use * in simp*)

next

case *False*

then show *?thesis* **by** (*simp add: indicator-def*)

qed

lemma *LIMSEQ-indicator-INT*:

 ($\lambda k. \text{indicator } (\bigcap i < k. A\ i) \ x :: 'a :: \{\text{topological-space, zero-neq-one}\}$) $\longrightarrow \text{indicator } (\bigcap i. A\ i) \ x$

proof –

have ($\lambda k. \text{indicator } (\bigcap i < k. A\ i) \ x :: 'a$) $\longrightarrow \text{indicator } (\bigcap k. \bigcap i < k. A\ i) \ x$

by (*intro LIMSEQ-indicator-decseq*) (*auto simp: decseq-def intro: less-le-trans*)

also have ($\bigcap k. \bigcap i < k. A\ i = \bigcap i. A\ i$)

by *auto*

finally show *?thesis* .

qed

lemma *indicator-add*:

$A \cap B = \{\} \implies (\text{indicator } A \ x :: \text{monoid-add}) + \text{indicator } B \ x = \text{indicator } (A \cup B) \ x$

unfolding *indicator-def* **by** *auto*

lemma *of-real-indicator*: *of-real* (*indicator A x*) = *indicator A x*

by (*simp split: split-indicator*)

lemma *real-of-nat-indicator*: *real* (*indicator A x :: nat*) = *indicator A x*

```

    by (simp split: split-indicator)

lemma abs-indicator: |indicator A x :: 'a::linordered-idom| = indicator A x
  by (simp split: split-indicator)

lemma mult-indicator-subset:
  A ⊆ B ⇒ indicator A x * indicator B x = (indicator A x :: 'a::comm-semiring-1)
  by (auto split: split-indicator simp: fun-eq-iff)

lemma indicator-times-eq-if:
  fixes f :: 'a ⇒ 'b::comm-ring-1
  shows indicator S x * f x = (if x ∈ S then f x else 0) f x * indicator S x = (if x
  ∈ S then f x else 0)
  by auto

lemma indicator-scaleR-eq-if:
  fixes f :: 'a ⇒ 'b::real-vector
  shows indicator S x *R f x = (if x ∈ S then f x else 0)
  by simp

lemma indicator-sums:
  assumes ⋀i j. i ≠ j ⇒ A i ∩ A j = {}
  shows (λi. indicator (A i) x::real) sums indicator (⋃i. A i) x
proof (cases ∃i. x ∈ A i)
  case True
  then obtain i where i: x ∈ A i ..
  with assms have (λi. indicator (A i) x::real) sums (∑ i∈{i}. indicator (A i) x)
    by (intro sums-finite) (auto split: split-indicator)
  also have (∑ i∈{i}. indicator (A i) x) = indicator (⋃i. A i) x
    using i by (auto split: split-indicator)
  finally show ?thesis .
next
  case False
  then show ?thesis by simp
qed

  The indicator function of the union of a disjoint family of sets is the sum
  over all the individual indicators.

lemma indicator-UN-disjoint:
  finite A ⇒ disjoint-family-on f A ⇒ indicator (⋃(f ` A)) x = (∑ y∈A. indicator
  (f y) x)
  by (induct A rule: finite-induct)
    (auto simp: disjoint-family-on-def indicator-def split: if-splits split-of-bool-asm)

end

```

40 The type of non-negative extended real numbers

```

theory Extended-Nonnegative-Real
  imports Extended-Real Indicator-Function
begin

lemma ereal-ineq-diff-add:
  assumes  $b \neq (-\infty :: \text{ereal})$   $a \geq b$ 
  shows  $a = b + (a - b)$ 
by (metis add.commute assms ereal-eq-minus-iff ereal-minus-le-iff ereal-plus-eq-PInf)

lemma Limsup-const-add:
  fixes  $c :: 'a :: \{\text{complete-linorder, linorder-topology, topological-monoid-add, ordered-ab-semigroup-add}\}$ 
  shows  $F \neq \text{bot} \implies \text{Limsup } F (\lambda x. c + f x) = c + \text{Limsup } F f$ 
by (intro Limsup-compose-continuous-mono monoI add-mono continuous-intros)
  auto

lemma Liminf-const-add:
  fixes  $c :: 'a :: \{\text{complete-linorder, linorder-topology, topological-monoid-add, ordered-ab-semigroup-add}\}$ 
  shows  $F \neq \text{bot} \implies \text{Liminf } F (\lambda x. c + f x) = c + \text{Liminf } F f$ 
by (intro Liminf-compose-continuous-mono monoI add-mono continuous-intros)
  auto

lemma Liminf-add-const:
  fixes  $c :: 'a :: \{\text{complete-linorder, linorder-topology, topological-monoid-add, ordered-ab-semigroup-add}\}$ 
  shows  $F \neq \text{bot} \implies \text{Liminf } F (\lambda x. f x + c) = \text{Liminf } F f + c$ 
by (intro Liminf-compose-continuous-mono monoI add-mono continuous-intros)
  auto

lemma sums-offset:
  fixes  $f g :: \text{nat} \Rightarrow 'a :: \{\text{t2-space, topological-comm-monoid-add}\}$ 
  assumes  $(\lambda n. f (n + i)) \text{ sums } l$  shows  $f \text{ sums } (l + (\sum j < i. f j))$ 
proof -
  have  $(\lambda k. (\sum n < k. f (n + i)) + (\sum j < i. f j)) \longrightarrow l + (\sum j < i. f j)$ 
  using assms by (auto intro!: tendsto-add simp: sums-def)
  moreover have  $(\sum j < k + i. f j) = (\sum n < k. f (n + i)) + (\sum j < i. f j)$  for  $k :: \text{nat}$ 
  proof -
  have  $(\sum j < k + i. f j) = (\sum j = i..< k + i. f j) + (\sum j = 0..< i. f j)$ 
  by (subst sum.union-disjoint[symmetric]) (auto intro!: sum.cong)
  also have  $(\sum j = i..< k + i. f j) = (\sum j \in (\lambda n. n + i) \{0..< k\}. f j)$ 
  unfolding image-add-atLeastLessThan by simp
  finally show ?thesis
  by (auto simp: inj-on-def atLeast0LessThan sum.reindex)
qed

```

ultimately have $(\lambda k. (\sum n < k + i. f\ n)) \longrightarrow l + (\sum j < i. f\ j)$
 by *simp*
 then show *?thesis*
 unfolding *sums-def* by (rule *LIMSEQ-offset*)
 qed

lemma *suminf-offset*:
 fixes $f\ g :: \text{nat} \Rightarrow 'a :: \{\text{t2-space, topological-comm-monoid-add}\}$
 shows $\text{summable } (\lambda j. f\ (j + i)) \implies \text{suminf } f = (\sum j. f\ (j + i)) + (\sum j < i. f\ j)$
 by (intro *sums-unique[symmetric]* *sums-offset* *summable-sums*)

lemma *eventually-at-left-1*: $(\bigwedge z :: \text{real}. 0 < z \implies z < 1 \implies P\ z) \implies \text{eventually } P\ (\text{at-left } 1)$
 by (subst *eventually-at-left[of 0]*) (auto intro: *exI[of - 0]*)

lemma *mult-eq-1*:
 fixes $a\ b :: 'a :: \{\text{ordered-semiring, comm-monoid-mult}\}$
 shows $0 \leq a \implies a \leq 1 \implies b \leq 1 \implies a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$
 by (metis *mult.left-neutral eq-iff mult.commute mult-right-mono*)

lemma *ereal-add-diff-cancel*:
 fixes $a\ b :: \text{ereal}$
 shows $|b| \neq \infty \implies (a + b) - b = a$
 by (cases *a b* rule: *ereal2-cases*) auto

lemma *add-top*:
 fixes $x :: 'a :: \{\text{order-top, ordered-comm-monoid-add}\}$
 shows $0 \leq x \implies x + \text{top} = \text{top}$
 by (intro *top-le add-increasing order-refl*)

lemma *top-add*:
 fixes $x :: 'a :: \{\text{order-top, ordered-comm-monoid-add}\}$
 shows $0 \leq x \implies \text{top} + x = \text{top}$
 by (intro *top-le add-increasing2 order-refl*)

lemma *le-lfp*: $\text{mono } f \implies x \leq \text{lfp } f \implies f\ x \leq \text{lfp } f$
 by (subst *lfp-unfold*) (auto dest: *monoD*)

lemma *lfp-transfer*:
 assumes α : *sup-continuous* α and f : *sup-continuous* f and mg : *mono* g
 assumes *bot*: $\alpha\ \text{bot} \leq \text{lfp } g$ and *eq*: $\bigwedge x. x \leq \text{lfp } f \implies \alpha\ (f\ x) = g\ (\alpha\ x)$
 shows $\alpha\ (\text{lfp } f) = \text{lfp } g$
 proof (rule *antisym*)
 note $mf = \text{sup-continuous-mono}[OF\ f]$
 have *f-le-lfp*: $(f \frown i)\ \text{bot} \leq \text{lfp } f$ for i
 by (induction i) (auto intro: *le-lfp mf*)

have $\alpha\ ((f \frown i)\ \text{bot}) \leq \text{lfp } g$ for i
 by (induction i) (auto simp: *bot eq f-le-lfp intro!*: *le-lfp mg*)

```

then show  $\alpha$  (lfp  $f$ )  $\leq$   $\text{lfp } g$ 
  unfolding sup-continuous-lfp[OF f]
  by (simp add: SUP-least  $\alpha$ [THEN sup-continuousD] mf mono-funpow)
show  $\text{lfp } g \leq \alpha$  (lfp  $f$ )
  by (rule lfp-lowerbound) (simp add: eq[symmetric] lfp-fixpoint[OF mf])
qed

```

```

lemma sup-continuous-applyD: sup-continuous  $f \implies \text{sup-continuous } (\lambda x. f\ x\ h)$ 
  using sup-continuous-apply[THEN sup-continuous-compose] .

```

```

lemma sup-continuous-SUP[order-continuous-intros]:
  fixes  $M :: - \Rightarrow - \Rightarrow 'a::\text{complete-lattice}$ 
  assumes  $M: \bigwedge i. i \in I \implies \text{sup-continuous } (M\ i)$ 
  shows sup-continuous (SUP  $i \in I. M\ i$ )
  unfolding sup-continuous-def by (auto simp add: sup-continuousD [OF M] image-comp intro: SUP-commute)

```

```

lemma sup-continuous-apply-SUP[order-continuous-intros]:
  fixes  $M :: - \Rightarrow - \Rightarrow 'a::\text{complete-lattice}$ 
  shows  $(\bigwedge i. i \in I \implies \text{sup-continuous } (M\ i)) \implies \text{sup-continuous } (\lambda x. \text{SUP } i \in I. M\ i\ x)$ 
  unfolding SUP-apply[symmetric] by (rule sup-continuous-SUP)

```

```

lemma sup-continuous-lfp'[order-continuous-intros]:
  assumes  $1: \text{sup-continuous } f$ 
  assumes  $2: \bigwedge g. \text{sup-continuous } g \implies \text{sup-continuous } (f\ g)$ 
  shows sup-continuous (lfp  $f$ )
proof –
  have sup-continuous  $((f \rightsquigarrow i)\ \text{bot})$  for  $i$ 
  proof (induction i)
    case (Suc i) then show ?case
      by (auto intro!: 2)
  qed (simp add: bot-fun-def sup-continuous-const)
  then show ?thesis
  unfolding sup-continuous-lfp[OF 1] by (intro order-continuous-intros)
qed

```

```

lemma sup-continuous-lfp''[order-continuous-intros]:
  assumes  $1: \bigwedge s. \text{sup-continuous } (f\ s)$ 
  assumes  $2: \bigwedge g. \text{sup-continuous } g \implies \text{sup-continuous } (\lambda s. f\ s\ (g\ s))$ 
  shows sup-continuous  $(\lambda x. \text{lfp } (f\ x))$ 
proof –
  have sup-continuous  $(\lambda x. (f\ x \rightsquigarrow i)\ \text{bot})$  for  $i$ 
  proof (induction i)
    case (Suc i) then show ?case
      by (auto intro!: 2)
  qed (simp add: bot-fun-def sup-continuous-const)
  then show ?thesis
  unfolding sup-continuous-lfp[OF 1] by (intro order-continuous-intros)

```

qed

lemma *mono-INF-fun*:

$(\bigwedge x y. \text{mono } (F x y)) \implies \text{mono } (\lambda z x. \text{INF } y \in X x. F x y z :: 'a :: \text{complete-lattice})$

by (auto intro!: INF-mono[OF bexI] simp: le-fun-def mono-def)

lemma *continuous-on-cmult-ereal*:

$|c::ereal| \neq \infty \implies \text{continuous-on } A f \implies \text{continuous-on } A (\lambda x. c * f x)$

using tendsto-cmult-ereal[of c f f x at x within A for x]

by (auto simp: continuous-on-def simp del: tendsto-cmult-ereal)

lemma *real-of-nat-Sup*:

assumes $A \neq \{\}$ *bdd-above A*

shows $\text{of-nat } (\text{Sup } A) = (\text{SUP } a \in A. \text{of-nat } a :: \text{real})$

proof (intro antisym)

show $(\text{SUP } a \in A. \text{of-nat } a :: \text{real}) \leq \text{of-nat } (\text{Sup } A)$

using *assms* **by** (intro cSUP-least of-nat-mono) (auto intro: cSup-upper)

have $\text{Sup } A \in A$

using *assms* **by** (auto simp: Sup-nat-def bdd-above-nat)

then show $\text{of-nat } (\text{Sup } A) \leq (\text{SUP } a \in A. \text{of-nat } a :: \text{real})$

by (intro cSUP-upper bdd-above-image-mono *assms*) (auto simp: mono-def)

qed

lemma (in *complete-lattice*) *SUP-sup-const1*:

$I \neq \{\} \implies (\text{SUP } i \in I. \text{sup } c (f i)) = \text{sup } c (\text{SUP } i \in I. f i)$

using SUP-sup-distrib[of $\lambda \cdot. c$ I f] **by** simp

lemma (in *complete-lattice*) *SUP-sup-const2*:

$I \neq \{\} \implies (\text{SUP } i \in I. \text{sup } (f i) c) = \text{sup } (\text{SUP } i \in I. f i) c$

using SUP-sup-distrib[of f I $\lambda \cdot. c$] **by** simp

lemma *one-less-of-natD*:

assumes $(1::'a::\text{linordered-semidom}) < \text{of-nat } n$ **shows** $1 < n$

by (cases n) (use *assms* in auto)

40.1 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = $\{x :: \text{ereal}. 0 \leq x\}$

morphisms *enn2ereal* *e2ennreal'*

by auto

definition *e2ennreal* $x = e2ennreal' (\max 0 x)$

lemma *enn2ereal-range*: $e2ennreal \text{ ' } \{0..\} = \text{UNIV}$

proof –

have $\exists y \geq 0. x = e2ennreal y$ **for** x

by (cases x) (auto simp: e2ennreal-def max-absorb2)

```

then show ?thesis
  by (auto simp: image-iff Bex-def)
qed

lemma type-definition-ennreal': type-definition enn2ereal e2ennreal {x. 0 ≤ x}
  using type-definition-ennreal
  by (auto simp: type-definition-def e2ennreal-def max-absorb2)

setup-lifting type-definition-ennreal'

declare [[coercion e2ennreal]]

instantiation ennreal :: complete-linorder
begin

lift-definition top-ennreal :: ennreal is top by (rule top-greatest)
lift-definition bot-ennreal :: ennreal is 0 by (rule order-refl)
lift-definition sup-ennreal :: ennreal ⇒ ennreal ⇒ ennreal is sup by (rule le-supI1)
lift-definition inf-ennreal :: ennreal ⇒ ennreal ⇒ ennreal is inf by (rule le-infI)

lift-definition Inf-ennreal :: ennreal set ⇒ ennreal is Inf
  by (rule Inf-greatest)

lift-definition Sup-ennreal :: ennreal set ⇒ ennreal is sup 0 ∘ Sup
  by auto

lift-definition less-eq-ennreal :: ennreal ⇒ ennreal ⇒ bool is (≤) .
lift-definition less-ennreal :: ennreal ⇒ ennreal ⇒ bool is (<) .

instance
  by standard
  (transfer; auto simp: Inf-lower Inf-greatest Sup-upper Sup-least le-max-iff-disj
max.absorb1)+

end

lemma pcr-ennreal-enn2ereal[simp]: pcr-ennreal (enn2ereal x) x
  by (simp add: ennreal.pcr-cr-eq cr-ennreal-def)

lemma rel-fun-eq-pcr-ennreal: rel-fun (=) pcr-ennreal f g ⟷ f = enn2ereal ∘ g
  by (auto simp: rel-fun-def ennreal.pcr-cr-eq cr-ennreal-def)

instantiation ennreal :: infinity
begin

definition infinity-ennreal :: ennreal
  where [simp]: ∞ = (top::ennreal)

instance ..

```


end

instantiation *ennreal* :: {*semiring-1-no-zero-divisors*, *comm-semiring-1*}
begin

lift-definition *one-ennreal* :: *ennreal* **is** 1 **by** *simp*

lift-definition *zero-ennreal* :: *ennreal* **is** 0 **by** *simp*

lift-definition *plus-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** (+) **by** *simp*

lift-definition *times-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** (*) **by** *simp*

instance

by *standard* (*transfer*; *auto simp: field-simps ereal-right-distrib*)+

end

instantiation *ennreal* :: *minus*

begin

lift-definition *minus-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal* **is** $\lambda a\ b. \max\ 0\ (a - b)$

by *simp*

instance ..

end

instance *ennreal* :: *numeral* ..

instantiation *ennreal* :: *inverse*

begin

lift-definition *inverse-ennreal* :: *ennreal* \Rightarrow *ennreal* **is** *inverse*

by (*rule inverse-ereal-ge0I*)

definition *divide-ennreal* :: *ennreal* \Rightarrow *ennreal* \Rightarrow *ennreal*

where $x \text{ div } y = x * \text{inverse } (y :: \text{ennreal})$

instance ..

end

lemma *ennreal-zero-less-one*: $0 < (1 :: \text{ennreal})$ — TODO: remove

by *transfer auto*

instance *ennreal* :: *dioid*

proof (*standard*; *transfer*)

fix $a\ b :: \text{ereal}$ **assume** $0 \leq a\ 0 \leq b$ **then show** $(a \leq b) = (\exists c \in \text{Collect } ((\leq) 0). b = a + c)$

```

unfolding ereal-ex-split Bex-def
by (cases a b rule: ereal2-cases) (auto intro!: exI[of - real-of-ereal (b - a)])
qed

instance ennreal :: ordered-comm-semiring
by standard
  (transfer; auto intro: add-mono mult-mono mult-ac ereal-left-distrib ereal-mult-left-mono)+

instance ennreal :: linordered-nonzero-semiring
proof
  fix a b :: ennreal
  show a < b  $\implies$  a + 1 < b + 1
    by transfer (simp add: add-right-mono ereal-add-cancel-right less-le)
qed (transfer; simp)

instance ennreal :: strict-ordered-ab-semigroup-add
proof
  fix a b c d :: ennreal show a < b  $\implies$  c < d  $\implies$  a + c < b + d
    by transfer (auto intro!: ereal-add-strict-mono)
qed

declare [[coercion of-nat :: nat  $\Rightarrow$  ennreal]]

lemma e2ennreal-neg:  $x \leq 0 \implies e2ennreal\ x = 0$ 
  unfolding zero-ennreal-def e2ennreal-def by (simp add: max-absorb1)

lemma e2ennreal-mono:  $x \leq y \implies e2ennreal\ x \leq e2ennreal\ y$ 
  by (cases 0  $\leq$  x 0  $\leq$  y rule: bool.exhaust[case-product bool.exhaust])
    (auto simp: e2ennreal-neg less-eq-ennreal.abs-eq eq-onp-def)

lemma enn2ereal-nonneg[simp]:  $0 \leq enn2ereal\ x$ 
  using ennreal.enn2ereal[of x] by simp

lemma ereal-ennreal-cases:
  obtains b where  $0 \leq a$  a = enn2ereal b | a < 0
  using e2ennreal'-inverse[of a, symmetric] by (cases 0  $\leq$  a) (auto intro: enn2ereal-nonneg)

lemma rel-fun-liminf[transfer-rule]: rel-fun (rel-fun (=) pcr-ennreal) pcr-ennreal
  liminf liminf
proof -
  have  $\forall x\ y. \text{rel-fun } (=) \text{ pcr-ennreal } x\ y \longrightarrow \text{pcr-ennreal } (\sup 0 (\text{liminf } x)) (\text{liminf } y)$ 
   $\implies$ 
   $\forall x\ y. \text{rel-fun } (=) \text{ pcr-ennreal } x\ y \longrightarrow \text{pcr-ennreal } (\text{liminf } x) (\text{liminf } y)$ 
  by (auto simp: comp-def Liminf-bounded rel-fun-eq-pcr-ennreal)
  moreover have rel-fun (rel-fun (=) pcr-ennreal) pcr-ennreal  $(\lambda x. \sup 0 (\text{liminf } x)) \text{ liminf}$ 
  unfolding liminf-SUP-INF[abs-def] by (transfer-prover-start, transfer-step+;
  simp)
  ultimately show ?thesis

```

by (*simp add: rel-fun-def*)
qed

lemma *rel-fun-limsup[transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal* *limsup* *limsup*

proof –

have [*simp*]: $\max 0 \ (SUP\ x \in \{n.. \}. \text{enn2ereal } (y\ x)) = (SUP\ x \in \{n.. \}. \text{enn2ereal } (y\ x))$ for *n::nat* and *y*

by (*meson SUP-upper atLeast-iff enn2ereal-nonneg max.absorb2 nle-le order-trans*)

have *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal* ($\lambda x. \text{INF } n. \text{sup } 0 \ (SUP\ i \in \{n.. \}. x\ i)$) *limsup*

unfolding *limsup-INF-SUP[abs-def]* by (*transfer-prover-start, transfer-step+; simp*)

moreover

have $\bigwedge x\ y. \llbracket \text{rel-fun } (=) \text{ pcr-ennreal } x\ y;$

$\text{pcr-ennreal } (\text{INF } n::\text{nat}. \max 0 \ (\text{Sup } (x\ ' \ \{n.. \}))) \ (\text{INF } n. \text{Sup } (y\ ' \ \{n.. \})) \rrbracket$

$\implies \text{pcr-ennreal } (\text{INF } n. \text{Sup } (x\ ' \ \{n.. \})) \ (\text{INF } n. \text{Sup } (y\ ' \ \{n.. \}))$

by (*auto simp: comp-def rel-fun-eq-pcr-ennreal*)

ultimately show ?thesis

by (*simp add: limsup-INF-SUP rel-fun-def*)

qed

lemma *sum-enn2ereal[simp]*: $(\bigwedge i. i \in I \implies 0 \leq f\ i) \implies (\sum i \in I. \text{enn2ereal } (f\ i)) = \text{enn2ereal } (\text{sum } f\ I)$

by (*induction I rule: infinite-finite-induct*) (*auto simp: sum-nonneg zero-ennreal.rep-eq plus-ennreal.rep-eq*)

lemma *transfer-e2ennreal-sum [transfer-rule]*:

rel-fun (*rel-fun* (=) *pcr-ennreal*) (*rel-fun* (=) *pcr-ennreal*) *sum* *sum*

by (*auto intro!: rel-funI simp: rel-fun-eq-pcr-ennreal comp-def*)

lemma *enn2ereal-of-nat[simp]*: *enn2ereal* (*of-nat* *n*) = *ereal* *n*

by (*induction n*) (*auto simp: zero-ennreal.rep-eq one-ennreal.rep-eq plus-ennreal.rep-eq*)

lemma *enn2ereal-numeral[simp]*: *enn2ereal* (*numeral* *a*) = *numeral* *a*

by (*metis enn2ereal-of-nat numeral-eq-ereal of-nat-numeral*)

lemma *transfer-numeral[transfer-rule]*: *pcr-ennreal* (*numeral* *a*) (*numeral* *a*)

by (*metis enn2ereal-numeral pcr-ennreal-enn2ereal*)

40.2 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::\text{ennreal}) \vee b = c$

unfolding *infinity-ennreal-def* by *transfer* (*simp add: top-ereal-def ereal-add-cancel-left*)

lemma *ennreal-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::\text{ennreal}) \vee b \leq c$

unfolding *infinity-ennreal-def* by *transfer* (*simp add: ereal-add-le-add-iff top-ereal-def*)

disj-commute)

lemma *ereal-add-left-cancel-less*:

fixes $a\ b\ c :: \text{ereal}$

shows $0 \leq a \implies 0 \leq b \implies a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$

by (*cases a b c rule: ereal3-cases*) *auto*

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty :: \text{ennreal}) \wedge b < c$

unfolding *infinity-ennreal-def*

by *transfer (simp add: top-ereal-def ereal-add-left-cancel-less)*

ML \langle

structure Cancel-Ennreal-Common =

struct

(copied from src/HOL/Tools/nat-numeral-simprocs.ML *)*

fun find-first-t - - [] = raise TERM (find-first-t, [])

| find-first-t past u (t::terms) =

if u aconv t then (rev past @ terms)

else find-first-t (t::past) u terms

fun dest-summing (Const (const-name <Groups.plus>, -) \$ t \$ u, ts) =

dest-summing (t, dest-summing (u, ts))

| dest-summing (t, ts) = t :: ts

val mk-sum = Arith-Data.long-mk-sum

fun dest-sum t = dest-summing (t, [])

val find-first = find-first-t []

val trans-tac = Numeral-Simprocs.trans-tac

val norm-ss =

simpset-of (put-simpset HOL-basic-ss context

|> Simplifier.add-simps @{thms ac-simps add-0-left add-0-right})

fun norm-tac ctxt = ALLGOALS (simp-tac (put-simpset norm-ss ctxt))

fun simplify-meta-eq ctxt cancel-th th =

Arith-Data.simplify-meta-eq [] ctxt

([th, cancel-th] MRS trans)

fun mk-eq (a, b) = HOLogic.mk-Trueprop (HOLogic.mk-eq (a, b))

end

structure Eq-Ennreal-Cancel = ExtractCommonTermFun

(open Cancel-Ennreal-Common

val mk-bal = HOLogic.mk-eq

val dest-bal = HOLogic.dest-bin const-name <HOL.eq> typ <ennreal>

fun simp-conv - - = SOME @{thm ennreal-add-left-cancel}

)

structure Le-Ennreal-Cancel = ExtractCommonTermFun

(open Cancel-Ennreal-Common

val mk-bal = HOLogic.mk-binrel const-name <Orderings.less-eq>

val dest-bal = HOLogic.dest-bin const-name <Orderings.less-eq> typ <ennreal>

```

  fun simp-conv - - = SOME @{thm ennreal-add-left-cancel-le}
)

```

```

structure Less-Ennreal-Cancel = ExtractCommonTermFun
(open Cancel-Ennreal-Common
  val mk-bal = HOLogic.mk-binrel const-name ⟨Orderings.less⟩
  val dest-bal = HOLogic.dest-bin const-name ⟨Orderings.less⟩ typ ⟨ennreal⟩
  fun simp-conv - - = SOME @{thm ennreal-add-left-cancel-less}
)
>

```

```

simproc-setup ennreal-eq-cancel
  ((l::ennreal) + m = n | (l::ennreal) = m + n) =
  ⟨K (fn ctxt => fn ct => Eq-Ennreal-Cancel.proc ctxt (Thm.term-of ct))⟩

```

```

simproc-setup ennreal-le-cancel
  ((l::ennreal) + m ≤ n | (l::ennreal) ≤ m + n) =
  ⟨K (fn ctxt => fn ct => Le-Ennreal-Cancel.proc ctxt (Thm.term-of ct))⟩

```

```

simproc-setup ennreal-less-cancel
  ((l::ennreal) + m < n | (l::ennreal) < m + n) =
  ⟨K (fn ctxt => fn ct => Less-Ennreal-Cancel.proc ctxt (Thm.term-of ct))⟩

```

40.3 Order with top

```

lemma ennreal-zero-less-top[simp]: 0 < (top::ennreal)
  by transfer (simp add: top-ereal-def)

```

```

lemma ennreal-one-less-top[simp]: 1 < (top::ennreal)
  by transfer (simp add: top-ereal-def)

```

```

lemma ennreal-zero-neq-top[simp]: 0 ≠ (top::ennreal)
  by transfer (simp add: top-ereal-def)

```

```

lemma ennreal-top-neq-zero[simp]: (top::ennreal) ≠ 0
  by transfer (simp add: top-ereal-def)

```

```

lemma ennreal-top-neq-one[simp]: top ≠ (1::ennreal)
  by transfer (simp add: top-ereal-def one-ereal-def flip: ereal-max)

```

```

lemma ennreal-one-neq-top[simp]: 1 ≠ (top::ennreal)
  by transfer (simp add: top-ereal-def one-ereal-def flip: ereal-max)

```

```

lemma ennreal-add-less-top[simp]:
  fixes a b :: ennreal
  shows a + b < top ⟷ a < top ∧ b < top
  by transfer (auto simp: top-ereal-def)

```

```

lemma ennreal-add-eq-top[simp]:

```

fixes $a\ b :: \text{ennreal}$
shows $a + b = \text{top} \longleftrightarrow a = \text{top} \vee b = \text{top}$
by *transfer (auto simp: top-ereal-def)*

lemma *ennreal-sum-less-top[simp]:*
fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $\text{finite } I \implies (\sum i \in I. f\ i) < \text{top} \longleftrightarrow (\forall i \in I. f\ i < \text{top})$
by (*induction I rule: finite-induct*) *auto*

lemma *ennreal-sum-eq-top[simp]:*
fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $\text{finite } I \implies (\sum i \in I. f\ i) = \text{top} \longleftrightarrow (\exists i \in I. f\ i = \text{top})$
by (*induction I rule: finite-induct*) *auto*

lemma *ennreal-mult-eq-top-iff:*
fixes $a\ b :: \text{ennreal}$
shows $a * b = \text{top} \longleftrightarrow (a = \text{top} \wedge b \neq 0) \vee (b = \text{top} \wedge a \neq 0)$
by *transfer (auto simp: top-ereal-def)*

lemma *ennreal-top-eq-mult-iff:*
fixes $a\ b :: \text{ennreal}$
shows $\text{top} = a * b \longleftrightarrow (a = \text{top} \wedge b \neq 0) \vee (b = \text{top} \wedge a \neq 0)$
using *ennreal-mult-eq-top-iff[of a b]* **by** *auto*

lemma *ennreal-mult-less-top:*
fixes $a\ b :: \text{ennreal}$
shows $a * b < \text{top} \longleftrightarrow (a = 0 \vee b = 0 \vee (a < \text{top} \wedge b < \text{top}))$
by *transfer (auto simp add: top-ereal-def)*

lemma *top-power-ennreal:* $\text{top}^n = (\text{if } n = 0 \text{ then } 1 \text{ else } \text{top} :: \text{ennreal})$
by (*induction n*) (*simp-all add: ennreal-mult-eq-top-iff*)

lemma *ennreal-prod-eq-0[simp]:*
fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $(\text{prod } f\ A = 0) = (\text{finite } A \wedge (\exists i \in A. f\ i = 0))$
by (*induction A rule: infinite-finite-induct*) *auto*

lemma *ennreal-prod-eq-top:*
fixes $f :: 'a \Rightarrow \text{ennreal}$
shows $(\prod i \in I. f\ i) = \text{top} \longleftrightarrow (\text{finite } I \wedge ((\forall i \in I. f\ i \neq 0) \wedge (\exists i \in I. f\ i = \text{top})))$
by (*induction I rule: infinite-finite-induct*) (*auto simp: ennreal-mult-eq-top-iff*)

lemma *ennreal-top-mult:* $\text{top} * a = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$
by (*simp add: ennreal-mult-eq-top-iff*)

lemma *ennreal-mult-top:* $a * \text{top} = (\text{if } a = 0 \text{ then } 0 \text{ else } \text{top} :: \text{ennreal})$
by (*simp add: ennreal-mult-eq-top-iff*)

lemma *enn2ereal-eq-top-iff[simp]:* $\text{enn2ereal } x = \infty \longleftrightarrow x = \text{top}$

by *transfer (simp add: top-ereal-def)*

lemma *enn2ereal-top[simp]: enn2ereal top = ∞*
 by *transfer (simp add: top-ereal-def)*

lemma *e2ennreal-infty[simp]: e2ennreal ∞ = top*
 by *(simp add: top-ennreal.abs-eq top-ereal-def)*

lemma *ennreal-top-minus[simp]: top - x = (top::ennreal)*
 by *transfer (auto simp: top-ereal-def max-def)*

lemma *minus-top-ennreal: x - top = (if x = top then top else 0::ennreal)*
 by *transfer (use ereal-eq-minus-iff top-ereal-def in force)*

lemma *bot-ennreal: bot = (0::ennreal)*
 by *transfer rule*

lemma *ennreal-of-nat-neq-top[simp]: of-nat i \neq (top::ennreal)*
 by *(induction i) auto*

lemma *numeral-eq-of-nat: (numeral a::ennreal) = of-nat (numeral a)*
 by *simp*

lemma *of-nat-less-top: of-nat i < (top::ennreal)*
 using *less-le-trans[of of-nat i of-nat (Suc i) top::ennreal]*
 by *simp*

lemma *top-neq-numeral[simp]: top \neq (numeral i::ennreal)*
 using *of-nat-less-top[of numeral i] by simp*

lemma *ennreal-numeral-less-top[simp]: numeral i < (top::ennreal)*
 using *of-nat-less-top[of numeral i] by simp*

lemma *ennreal-add-bot[simp]: bot + x = (x::ennreal)*
 by *transfer simp*

lemma *add-top-right-ennreal [simp]: x + top = (top :: ennreal)*
 by *(cases x) auto*

lemma *add-top-left-ennreal [simp]: top + x = (top :: ennreal)*
 by *(cases x) auto*

lemma *ennreal-top-mult-left [simp]: x \neq 0 \implies x * top = (top :: ennreal)*
 by *(subst ennreal-mult-eq-top-iff) auto*

lemma *ennreal-top-mult-right [simp]: x \neq 0 \implies top * x = (top :: ennreal)*
 by *(subst ennreal-mult-eq-top-iff) auto*

lemma *power-top-ennreal* [*simp*]: $n > 0 \implies \text{top} \wedge n = (\text{top} :: \text{ennreal})$
by (*induction n*) *auto*

lemma *power-eq-top-ennreal-iff*: $x \wedge n = \text{top} \iff x = (\text{top} :: \text{ennreal}) \wedge n > 0$
by (*induction n*) (*auto simp: ennreal-mult-eq-top-iff*)

lemma *ennreal-mult-le-mult-iff*: $c \neq 0 \implies c \neq \text{top} \implies c * a \leq c * b \iff a \leq (b :: \text{ennreal})$
including *ennreal.lifting*
by (*transfer, subst ereal-mult-le-mult-iff*) (*auto simp: top-ereal-def*)

lemma *power-mono-ennreal*: $x \leq y \implies x \wedge n \leq (y \wedge n :: \text{ennreal})$
by (*induction n*) (*auto intro!: mult-mono*)

instance *ennreal* :: *semiring-char-0*

proof (*standard, safe intro!: linorder-injI*)

have $1 + \text{of-nat } k \neq (0 :: \text{ennreal})$ **for** k

using *add-pos-nonneg* [*OF zero-less-one, of of-nat k :: ennreal*] **by** *auto*

fix $x y :: \text{nat}$ **assume** $x < y$ *of-nat* $x = (\text{of-nat } y :: \text{ennreal})$ **then show** *False*

by (*auto simp add: less-iff-Suc-add **)

qed

40.4 Arithmetic

lemma *ennreal-minus-zero* [*simp*]: $a - (0 :: \text{ennreal}) = a$
by *transfer (auto simp: max-def)*

lemma *ennreal-add-diff-cancel-right* [*simp*]:
fixes $x y z :: \text{ennreal}$ **shows** $y \neq \text{top} \implies (x + y) - y = x$
by *transfer (metis ereal-eq-minus-iff max-absorb2 not-MInfty-nonneg top-ereal-def)*

lemma *ennreal-add-diff-cancel-left* [*simp*]:
fixes $x y z :: \text{ennreal}$ **shows** $y \neq \text{top} \implies (y + x) - y = x$
by (*simp add: add.commute*)

lemma
fixes $a b :: \text{ennreal}$
shows $a - b = 0 \implies a \leq b$
by *transfer (metis ereal-diff-gr0 le-cases max.absorb2 not-less)*

lemma *ennreal-minus-cancel*:
fixes $a b c :: \text{ennreal}$
shows $c \neq \text{top} \implies a \leq c \implies b \leq c \implies c - a = c - b \implies a = b$
by (*metis ennreal-add-diff-cancel-left ennreal-add-diff-cancel-right ennreal-add-eq-top less-eqE*)

lemma *sup-const-add-ennreal*:
fixes $a b c :: \text{ennreal}$
shows $\sup (c + a) (c + b) = c + \sup a b$

by *transfer (metis add-left-mono le-cases sup.absorb2 sup.orderE)*

lemma *ennreal-diff-add-assoc:*

fixes $a\ b\ c :: \text{ennreal}$

shows $a \leq b \implies c + b - a = c + (b - a)$

by (*metis add.left-commute ennreal-add-diff-cancel-left ennreal-add-eq-top ennreal-top-minus less-eqE*)

lemma *mult-divide-eq-ennreal:*

fixes $a\ b :: \text{ennreal}$

shows $b \neq 0 \implies b \neq \text{top} \implies (a * b) / b = a$

unfolding *divide-ennreal-def*

apply *transfer*

by (*metis abs-ereal-ge0 divide-ereal-def ereal-divide-eq ereal-times-divide-eq top-ereal-def*)

lemma *divide-mult-eq:* $a \neq 0 \implies a \neq \infty \implies x * a / (b * a) = x / (b :: \text{ennreal})$

unfolding *divide-ennreal-def infinity-ennreal-def*

apply *transfer*

subgoal for $a\ b\ c$

by (*cases a b c rule: ereal3-cases (auto simp: top-ereal-def)*)

done

lemma *ennreal-mult-divide-eq:*

fixes $a\ b :: \text{ennreal}$

shows $b \neq 0 \implies b \neq \text{top} \implies (a * b) / b = a$

by (*fact mult-divide-eq-ennreal*)

lemma *ennreal-add-diff-cancel:*

fixes $a\ b :: \text{ennreal}$

shows $b \neq \infty \implies (a + b) - b = a$

by *simp*

lemma *ennreal-minus-eq-0:*

$a - b = 0 \implies a \leq (b :: \text{ennreal})$

by *transfer (metis ereal-diff-gr0 le-cases max.absorb2 not-less)*

lemma *ennreal-mono-minus-cancel:*

fixes $a\ b\ c :: \text{ennreal}$

shows $a - b \leq a - c \implies a < \text{top} \implies b \leq a \implies c \leq a \implies c \leq b$

by *transfer*

(*auto simp add: ereal-diff-positive top-ereal-def dest: ereal-mono-minus-cancel*)

lemma *ennreal-mono-minus:*

fixes $a\ b\ c :: \text{ennreal}$

shows $c \leq b \implies a - b \leq a - c$

by *transfer (meson ereal-minus-mono max.mono order-reft)*

lemma *ennreal-minus-pos-iff:*

fixes $a\ b :: \text{ennreal}$

shows $a < \text{top} \vee b < \text{top} \implies 0 < a - b \implies b < a$
by *transfer* (use *add.left-neutral ereal-minus-le-iff less-irrefl not-less in fastforce*)

lemma *ennreal-inverse-top[simp]*: $\text{inverse } \text{top} = (0::\text{ennreal})$
by *transfer* (*simp add: top-ereal-def ereal-inverse-eq-0*)

lemma *ennreal-inverse-zero[simp]*: $\text{inverse } 0 = (\text{top}::\text{ennreal})$
by *transfer* (*simp add: top-ereal-def ereal-inverse-eq-0*)

lemma *ennreal-top-divide*: $\text{top} / (x::\text{ennreal}) = (\text{if } x = \text{top} \text{ then } 0 \text{ else } \text{top})$
unfolding *divide-ennreal-def*
by *transfer* (*simp add: top-ereal-def ereal-inverse-eq-0 ereal-0-gt-inverse*)

lemma *ennreal-zero-divide[simp]*: $0 / (x::\text{ennreal}) = 0$
by (*simp add: divide-ennreal-def*)

lemma *ennreal-divide-zero[simp]*: $x / (0::\text{ennreal}) = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{top})$
by (*simp add: divide-ennreal-def ennreal-mult-top*)

lemma *ennreal-divide-top[simp]*: $x / (\text{top}::\text{ennreal}) = 0$
by (*simp add: divide-ennreal-def ennreal-top-mult*)

lemma *ennreal-times-divide*: $a * (b / c) = a * b / (c::\text{ennreal})$
unfolding *divide-ennreal-def*
by *transfer* (*simp add: divide-ereal-def[symmetric] ereal-times-divide-eq*)

lemma *ennreal-zero-less-divide*: $0 < a / b \longleftrightarrow (0 < a \wedge b < (\text{top}::\text{ennreal}))$
unfolding *divide-ennreal-def*
by *transfer* (*auto simp: ereal-zero-less-0-iff top-ereal-def ereal-0-gt-inverse*)

lemma *add-divide-distrib-ennreal*: $(a + b) / c = a / c + b / (c::\text{ennreal})$
by (*simp add: divide-ennreal-def ring-distrib*)

lemma *divide-right-mono-ennreal*:
fixes $a \ b \ c :: \text{ennreal}$
shows $a \leq b \implies a / c \leq b / c$
unfolding *divide-ennreal-def* **by** (*intro mult-mono*) *auto*

lemma *ennreal-mult-strict-right-mono*: $(a::\text{ennreal}) < c \implies 0 < b \implies b < \text{top} \implies a * b < c * b$
by *transfer* (*auto intro!: ereal-mult-strict-right-mono*)

lemma *ennreal-indicator-less[simp]*:
indicator $A \ x \leq (\text{indicator } B \ x::\text{ennreal}) \longleftrightarrow (x \in A \longrightarrow x \in B)$
by (*simp add: indicator-def not-le*)

lemma *ennreal-inverse-positive*: $0 < \text{inverse } x \longleftrightarrow (x::\text{ennreal}) \neq \text{top}$
by *transfer* (*simp add: ereal-0-gt-inverse top-ereal-def*)

lemma *ennreal-inverse-mult'*: $((0 < b \vee a < \text{top}) \wedge (0 < a \vee b < \text{top})) \implies$
 $\text{inverse } (a * b::\text{ennreal}) = \text{inverse } a * \text{inverse } b$
apply *transfer*
subgoal for $a \ b$
by (*cases a b rule: ereal2-cases*) (*auto simp: top-ereal-def*)
done

lemma *ennreal-inverse-mult*: $a < \text{top} \implies b < \text{top} \implies \text{inverse } (a * b::\text{ennreal}) =$
 $\text{inverse } a * \text{inverse } b$
by (*simp add: ennreal-inverse-mult'*)

lemma *ennreal-inverse-1[simp]*: $\text{inverse } (1::\text{ennreal}) = 1$
by *transfer simp*

lemma *ennreal-inverse-eq-0-iff[simp]*: $\text{inverse } (a::\text{ennreal}) = 0 \longleftrightarrow a = \text{top}$
by (*metis ennreal-inverse-positive not-gr-zero*)

lemma *ennreal-inverse-eq-top-iff[simp]*: $\text{inverse } (a::\text{ennreal}) = \text{top} \longleftrightarrow a = 0$
by *transfer (simp add: top-ereal-def)*

lemma *ennreal-divide-eq-0-iff[simp]*: $(a::\text{ennreal}) / b = 0 \longleftrightarrow (a = 0 \vee b = \text{top})$
by (*simp add: divide-ennreal-def*)

lemma *ennreal-divide-eq-top-iff*: $(a::\text{ennreal}) / b = \text{top} \longleftrightarrow ((a \neq 0 \wedge b = 0) \vee$
 $(a = \text{top} \wedge b \neq \text{top}))$
by (*auto simp add: divide-ennreal-def ennreal-mult-eq-top-iff*)

lemma *one-divide-one-divide-ennreal[simp]*: $1 / (1 / c) = (c::\text{ennreal})$
including *ennreal.lifting*
unfolding *divide-ennreal-def*
by *transfer auto*

lemma *ennreal-mult-left-cong*:
 $((a::\text{ennreal}) \neq 0 \implies b = c) \implies a * b = a * c$
by (*cases a = 0*) *simp-all*

lemma *ennreal-mult-right-cong*:
 $((a::\text{ennreal}) \neq 0 \implies b = c) \implies b * a = c * a$
by (*cases a = 0*) *simp-all*

lemma *ennreal-zero-less-mult-iff*: $0 < a * b \longleftrightarrow 0 < a \wedge 0 < (b::\text{ennreal})$
using *not-gr-zero by fastforce*

lemma *less-diff-eq-ennreal*:
fixes $a \ b \ c :: \text{ennreal}$
shows $b < \text{top} \vee c < \text{top} \implies a < b - c \longleftrightarrow a + c < b$
apply *transfer*
subgoal for $a \ b \ c$
by (*cases a b c rule: ereal3-cases*) (*auto split: split-max*)

done

lemma *diff-add-cancel-ennreal*:

fixes $a\ b :: \text{ennreal}$ **shows** $a \leq b \implies b - a + a = b$

unfolding *infinity-ennreal-def*

by *transfer (metis (no-types) add.commute ereal-diff-positive ereal-ineq-diff-add max-def not-MInfty-nonneg)*

lemma *ennreal-diff-self[simp]*: $a \neq \text{top} \implies a - a = (0 :: \text{ennreal})$

by *(meson ennreal-minus-pos-iff less-imp-neq not-gr-zero top.not-eq-extremum)*

lemma *ennreal-minus-mono*:

fixes $a\ b\ c :: \text{ennreal}$

shows $a \leq c \implies d \leq b \implies a - b \leq c - d$

by *transfer (meson ereal-minus-mono max.mono order-refl)*

lemma *ennreal-minus-eq-top[simp]*: $a - (b :: \text{ennreal}) = \text{top} \longleftrightarrow a = \text{top}$

by *(metis add-top diff-add-cancel-ennreal ennreal-mono-minus ennreal-top-minus zero-le)*

lemma *ennreal-divide-self[simp]*: $a \neq 0 \implies a < \text{top} \implies a / a = (1 :: \text{ennreal})$

by *(metis mult-1 mult-divide-eq-ennreal top.not-eq-extremum)*

40.5 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: *real* \Rightarrow *ennreal* **is** $\text{sup } 0 \circ \text{ereal}$

by *simp*

declare *[[coercion ennreal]]*

lemma *ennreal-cong*: $x = y \implies \text{ennreal } x = \text{ennreal } y$

by *simp*

lemma *ennreal-cases*[*cases type: ennreal*]:

fixes $x :: \text{ennreal}$

obtains (*real*) $r :: \text{real}$ **where** $0 \leq r$ $x = \text{ennreal } r \mid (\text{top})\ x = \text{top}$

apply *transfer*

subgoal for x *thesis*

by *(cases x) (auto simp: top-ereal-def)*

done

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]

lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neq-top[simp]*: $\text{ennreal } r \neq \text{top}$

by *transfer (simp add: top-ereal-def zero-ereal-def flip: ereal-max)*

lemma *top-neq-ennreal[simp]*: $\text{top} \neq \text{ennreal } r$

using *ennreal-neq-top[of r]* **by** *(auto simp del: ennreal-neq-top)*

lemma *ennreal-less-top*[simp]: $\text{ennreal } x < \text{top}$
by *transfer* (*simp add: top-ereal-def max-def*)

lemma *ennreal-neg*: $x \leq 0 \implies \text{ennreal } x = 0$
by *transfer* (*simp add: max.absorb1*)

lemma *ennreal-inj*[simp]:
 $0 \leq a \implies 0 \leq b \implies \text{ennreal } a = \text{ennreal } b \longleftrightarrow a = b$
by (*transfer fixing: a b*) (*auto simp: max.absorb2*)

lemma *ennreal-le-iff*[simp]: $0 \leq y \implies \text{ennreal } x \leq \text{ennreal } y \longleftrightarrow x \leq y$
by (*auto simp: ennreal-def zero-ereal-def less-eq-ennreal.abs-eq eq-onp-def split: split-max*)

lemma *le-ennreal-iff*: $0 \leq r \implies x \leq \text{ennreal } r \longleftrightarrow (\exists q \geq 0. x = \text{ennreal } q \wedge q \leq r)$
by (*cases x*) (*auto simp: top-unique*)

lemma *ennreal-less-iff*: $0 \leq r \implies \text{ennreal } r < \text{ennreal } q \longleftrightarrow r < q$
unfolding *not-le[symmetric]* **by** *auto*

lemma *ennreal-eq-zero-iff*[simp]: $0 \leq x \implies \text{ennreal } x = 0 \longleftrightarrow x = 0$
by *transfer* (*auto simp: max.absorb2*)

lemma *ennreal-less-zero-iff*[simp]: $0 < \text{ennreal } x \longleftrightarrow 0 < x$
by *transfer* (*auto simp: max-def*)

lemma *ennreal-lessI*: $0 < q \implies r < q \implies \text{ennreal } r < \text{ennreal } q$
by (*cases* $0 \leq r$) (*auto simp: ennreal-less-iff ennreal-neg*)

lemma *ennreal-leI*: $x \leq y \implies \text{ennreal } x \leq \text{ennreal } y$
by (*cases* $0 \leq y$) (*auto simp: ennreal-neg*)

lemma *enn2ereal-ennreal*[simp]: $0 \leq x \implies \text{enn2ereal } (\text{ennreal } x) = x$
by *transfer* (*simp add: max.absorb2*)

lemma *e2ennreal-enn2ereal*[simp]: $e2ennreal (\text{enn2ereal } x) = x$
by (*simp add: e2ennreal-def max.absorb2 ennreal.enn2ereal-inverse*)

lemma *enn2ereal-e2ennreal*: $x \geq 0 \implies \text{enn2ereal } (e2ennreal x) = x$
by (*metis e2ennreal-enn2ereal ereal-ennreal-cases not-le*)

lemma *e2ennreal-ereal* [simp]: $e2ennreal (\text{ereal } x) = \text{ennreal } x$
by (*metis e2ennreal-def enn2ereal-inverse ennreal.rep-eq sup-ereal-def*)

lemma *ennreal-0*[simp]: $\text{ennreal } 0 = 0$
by (*simp add: ennreal-def zero-ennreal.abs-eq*)

lemma *ennreal-1[simp]*: $\text{ennreal } 1 = 1$
by *transfer (simp add: max-absorb2)*

lemma *ennreal-eq-0-iff*: $\text{ennreal } x = 0 \longleftrightarrow x \leq 0$
by *(cases 0 ≤ x) (auto simp: ennreal-neg)*

lemma *ennreal-le-iff2*: $\text{ennreal } x \leq \text{ennreal } y \longleftrightarrow ((0 \leq y \wedge x \leq y) \vee (x \leq 0 \wedge y \leq 0))$
by *(cases 0 ≤ y) (auto simp: ennreal-eq-0-iff ennreal-neg)*

lemma *ennreal-eq-1[simp]*: $\text{ennreal } x = 1 \longleftrightarrow x = 1$
by *(cases 0 ≤ x) (auto simp: ennreal-neg simp flip: ennreal-1)*

lemma *ennreal-le-1[simp]*: $\text{ennreal } x \leq 1 \longleftrightarrow x \leq 1$
by *(cases 0 ≤ x) (auto simp: ennreal-neg simp flip: ennreal-1)*

lemma *ennreal-ge-1[simp]*: $\text{ennreal } x \geq 1 \longleftrightarrow x \geq 1$
by *(cases 0 ≤ x) (auto simp: ennreal-neg simp flip: ennreal-1)*

lemma *one-less-ennreal[simp]*: $1 < \text{ennreal } x \longleftrightarrow 1 < x$
by *(meson ennreal-le-1 linorder-not-le)*

lemma *ennreal-plus[simp]*:
 $0 \leq a \implies 0 \leq b \implies \text{ennreal } (a + b) = \text{ennreal } a + \text{ennreal } b$
by *(transfer fixing: a b) (auto simp: max-absorb2)*

lemma *add-mono-ennreal*: $x < \text{ennreal } y \implies x' < \text{ennreal } y' \implies x + x' < \text{ennreal } (y + y')$
by *(metis (full-types) add-strict-mono ennreal-less-zero-iff ennreal-plus less-le not-less zero-le)*

lemma *sum-ennreal[simp]*: $(\bigwedge i. i \in I \implies 0 \leq f i) \implies (\sum_{i \in I}. \text{ennreal } (f i)) = \text{ennreal } (\sum f I)$
by *(induction I rule: infinite-finite-induct) (auto simp: sum-nonneg)*

lemma *sum-list-ennreal[simp]*:
assumes $\bigwedge x. x \in \text{set } xs \implies f x \geq 0$
shows $\text{sum-list } (\text{map } (\lambda x. \text{ennreal } (f x)) xs) = \text{ennreal } (\text{sum-list } (\text{map } f xs))$
using *assms*
proof *(induction xs)*
case *(Cons x xs)*
from *Cons* **have** $(\sum_{x \leftarrow x \# xs}. \text{ennreal } (f x)) = \text{ennreal } (f x) + \text{ennreal } (\text{sum-list } (\text{map } f xs))$
by *simp*
also from *Cons.prem*s **have** $\dots = \text{ennreal } (f x + \text{sum-list } (\text{map } f xs))$
by *(intro ennreal-plus [symmetric] sum-list-nonneg) auto*
finally show *?case* **by** *simp*
qed *simp-all*

lemma *ennreal-of-nat-eq-real-of-nat*: $\text{of-nat } i = \text{ennreal } (\text{of-nat } i)$
by (*induction i*) *simp-all*

lemma *of-nat-le-ennreal-iff[simp]*: $0 \leq r \implies \text{of-nat } i \leq \text{ennreal } r \longleftrightarrow \text{of-nat } i \leq r$
by (*simp add: ennreal-of-nat-eq-real-of-nat*)

lemma *ennreal-le-of-nat-iff[simp]*: $\text{ennreal } r \leq \text{of-nat } i \longleftrightarrow r \leq \text{of-nat } i$
by (*simp add: ennreal-of-nat-eq-real-of-nat*)

lemma *ennreal-indicator*: $\text{ennreal } (\text{indicator } A \ x) = \text{indicator } A \ x$
by (*auto split: split-indicator*)

lemma *ennreal-numeral[simp]*: $\text{ennreal } (\text{numeral } n) = \text{numeral } n$
using *ennreal-of-nat-eq-real-of-nat[of numeral n]* **by** *simp*

lemma *ennreal-less-numeral-iff [simp]*: $\text{ennreal } n < \text{numeral } w \longleftrightarrow n < \text{numeral } w$
by (*metis ennreal-less-iff ennreal-numeral less-le not-less zero-less-numeral*)

lemma *numeral-less-ennreal-iff [simp]*: $\text{numeral } w < \text{ennreal } n \longleftrightarrow \text{numeral } w < n$
using *ennreal-less-iff zero-le-numeral* **by** *fastforce*

lemma *numeral-le-ennreal-iff [simp]*: $\text{numeral } n \leq \text{ennreal } m \longleftrightarrow \text{numeral } n \leq m$
by (*metis not-le ennreal-less-numeral-iff*)

lemma *min-ennreal*: $0 \leq x \implies 0 \leq y \implies \min (\text{ennreal } x) (\text{ennreal } y) = \text{ennreal } (\min x \ y)$
by (*auto split: split-min*)

lemma *ennreal-half[simp]*: $\text{ennreal } (1/2) = \text{inverse } 2$
by *transfer auto*

lemma *ennreal-minus*: $0 \leq q \implies \text{ennreal } r - \text{ennreal } q = \text{ennreal } (r - q)$
by *transfer (simp add: zero-ereal-def flip: ereal-max)*

lemma *ennreal-minus-top[simp]*: $\text{ennreal } a - \text{top} = 0$
by (*simp add: minus-top-ennreal*)

lemma *e2ennreal-enn2ereal-diff [simp]*:
 $e2ennreal(\text{enn2ereal } x - \text{enn2ereal } y) = x - y$ **for** $x \ y$
by (*cases x, cases y, auto simp add: ennreal-minus e2ennreal-neg*)

lemma *ennreal-mult*: $0 \leq a \implies 0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
by *transfer (simp add: max-absorb2)*

lemma *ennreal-mult'*: $0 \leq a \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$

by (cases $0 \leq b$) (auto simp: ennreal-mult ennreal-neg mult-nonneg-nonpos)

lemma indicator-mult-ennreal: indicator A $x * \text{ennreal } r = \text{ennreal } (\text{indicator } A \ x * r)$
by (simp split: split-indicator)

lemma ennreal-mult'': $0 \leq b \implies \text{ennreal } (a * b) = \text{ennreal } a * \text{ennreal } b$
by (cases $0 \leq a$) (auto simp: ennreal-mult ennreal-neg mult-nonneg-nonneg)

lemma numeral-mult-ennreal: $0 \leq x \implies \text{numeral } b * \text{ennreal } x = \text{ennreal } (\text{numeral } b * x)$
by (simp add: ennreal-mult)

lemma ennreal-power: $0 \leq r \implies \text{ennreal } r \wedge n = \text{ennreal } (r \wedge n)$
by (induction n) (auto simp: ennreal-mult)

lemma power-eq-top-ennreal: $x \wedge n = \text{top} \longleftrightarrow (n \neq 0 \wedge (x :: \text{ennreal}) = \text{top})$
using not-gr-zero power-eq-top-ennreal-iff **by** force

lemma inverse-ennreal: $0 < r \implies \text{inverse } (\text{ennreal } r) = \text{ennreal } (\text{inverse } r)$
by transfer (simp add: max.absorb2)

lemma divide-ennreal: $0 \leq r \implies 0 < q \implies \text{ennreal } r / \text{ennreal } q = \text{ennreal } (r / q)$
by (simp add: divide-ennreal-def inverse-ennreal ennreal-mult[symmetric] inverse-eq-divide)

lemma ennreal-inverse-power: $\text{inverse } (x \wedge n :: \text{ennreal}) = \text{inverse } x \wedge n$

proof (cases x rule: ennreal-cases)

case top **with** power-eq-top-ennreal[of x n] **show** ?thesis

by (cases $n = 0$) auto

next

case (real r) **then show** ?thesis

proof (cases $x = 0$)

case False **then show** ?thesis

by (smt (verit, best) ennreal-0 ennreal-power inverse-ennreal
inverse-nonnegative-iff-nonnegative power-inverse real zero-less-power)

qed (simp add: top-power-ennreal)

qed

lemma power-divide-distrib-ennreal [algebra-simps]:

$(x / y) \wedge n = x \wedge n / (y \wedge n :: \text{ennreal})$

by (simp add: divide-ennreal-def ennreal-inverse-power power-mult-distrib)

lemma ennreal-divide-numeral: $0 \leq x \implies \text{ennreal } x / \text{numeral } b = \text{ennreal } (x / \text{numeral } b)$

by (subst divide-ennreal[symmetric]) auto

lemma prod-ennreal: $(\bigwedge i. i \in A \implies 0 \leq f i) \implies (\prod_{i \in A} \text{ennreal } (f i)) = \text{ennreal } (\prod_{i \in A} f i)$

(*prod f A*)
by (*induction A rule: infinite-finite-induct*)
 (*auto simp: ennreal-mult prod-nonneg*)

lemma *prod-mono-ennreal*:
assumes $\bigwedge x. x \in A \implies f\ x \leq (g\ x :: \text{ennreal})$
shows $\text{prod } f\ A \leq \text{prod } g\ A$
using *assms* **by** (*induction A rule: infinite-finite-induct*) (*auto intro!: mult-mono*)

lemma *mult-right-ennreal-cancel*: $a * \text{ennreal } c = b * \text{ennreal } c \longleftrightarrow (a = b \vee c \leq 0)$
by (*metis ennreal-eq-0-iff mult-divide-eq-ennreal mult-eq-0-iff top-neq-ennreal*)

lemma *ennreal-le-epsilon*:
 $(\bigwedge e :: \text{real}. y < \text{top} \implies 0 < e \implies x \leq y + \text{ennreal } e) \implies x \leq y$
apply (*cases y rule: ennreal-cases*)
apply (*cases x rule: ennreal-cases*)
apply (*auto simp flip: ennreal-plus simp add: top-unique intro: zero-less-one field-le-epsilon*)
done

lemma *ennreal-rat-dense*:
fixes $x\ y :: \text{ennreal}$
shows $x < y \implies \exists r :: \text{rat}. x < \text{real-of-rat } r \wedge \text{real-of-rat } r < y$
proof *transfer*
fix $x\ y :: \text{ereal}$ **assume** $xy: 0 \leq x\ 0 \leq y\ x < y$
moreover
from *ereal-dense3*[*OF* $\langle x < y \rangle$]
obtain r **where** $r: x < \text{ereal } (\text{real-of-rat } r) \wedge \text{ereal } (\text{real-of-rat } r) < y$
by *auto*
then have $0 \leq r$
using *le-less-trans*[*OF* $\langle 0 \leq x \rangle \langle x < \text{ereal } (\text{real-of-rat } r) \rangle$] **by** *auto*
with r **show** $\exists r. x < (\text{sup } 0 \circ \text{ereal}) (\text{real-of-rat } r) \wedge (\text{sup } 0 \circ \text{ereal}) (\text{real-of-rat } r) < y$
by (*intro exI[of - r]*) (*auto simp: max-absorb2*)
qed

lemma *ennreal-Ex-less-of-nat*: $(x :: \text{ennreal}) < \text{top} \implies \exists n. x < \text{of-nat } n$
by (*cases x rule: ennreal-cases*)
 (*auto simp: ennreal-of-nat-eq-real-of-nat ennreal-less-iff reals-Archimedean2*)

40.6 Coercion from *ennreal* to *real*

definition *enn2real* $x = \text{real-of-ereal } (\text{enn2ereal } x)$

lemma *enn2real-nonneg*[*simp*]: $0 \leq \text{enn2real } x$
by (*auto simp: enn2real-def intro!: real-of-ereal-pos enn2ereal-nonneg*)

lemma *enn2real-mono*: $a \leq b \implies b < \text{top} \implies \text{enn2real } a \leq \text{enn2real } b$

by (*auto simp add: enn2real-def less-eq-ennreal.rep-eq intro!: real-of-ereal-positive-mono enn2ereal-nonneg*)

lemma *enn2real-of-nat[simp]*: $\text{enn2real } (\text{of-nat } n) = n$
by (*auto simp: enn2real-def*)

lemma *enn2real-ennreal[simp]*: $0 \leq r \implies \text{enn2real } (\text{ennreal } r) = r$
by (*simp add: enn2real-def*)

lemma *ennreal-enn2real[simp]*: $r < \text{top} \implies \text{ennreal } (\text{enn2real } r) = r$
by (*cases r rule: ennreal-cases*) *auto*

lemma *real-of-ereal-enn2ereal[simp]*: $\text{real-of-ereal } (\text{enn2ereal } x) = \text{enn2real } x$
by (*simp add: enn2real-def*)

lemma *enn2real-top[simp]*: $\text{enn2real } \text{top} = 0$
unfolding *enn2real-def top-ennreal.rep-eq top-ereal-def* **by** *simp*

lemma *enn2real-0[simp]*: $\text{enn2real } 0 = 0$
unfolding *enn2real-def zero-ennreal.rep-eq* **by** *simp*

lemma *enn2real-1[simp]*: $\text{enn2real } 1 = 1$
unfolding *enn2real-def one-ennreal.rep-eq* **by** *simp*

lemma *enn2real-numeral[simp]*: $\text{enn2real } (\text{numeral } n) = (\text{numeral } n)$
unfolding *enn2real-def* **by** *simp*

lemma *enn2real-mult*: $\text{enn2real } (a * b) = \text{enn2real } a * \text{enn2real } b$
unfolding *enn2real-def*
by (*simp del: real-of-ereal-enn2ereal add: times-ennreal.rep-eq*)

lemma *enn2real-leI*: $0 \leq B \implies x \leq \text{ennreal } B \implies \text{enn2real } x \leq B$
by (*cases x rule: ennreal-cases*) (*auto simp: top-unique*)

lemma *enn2real-positive-iff*: $0 < \text{enn2real } x \longleftrightarrow (0 < x \wedge x < \text{top})$
by (*cases x rule: ennreal-cases*) *auto*

lemma *enn2real-eq-posreal-iff[simp]*: $c > 0 \implies \text{enn2real } x = c \longleftrightarrow x = c$
by (*cases x*) *auto*

lemma *ennreal-enn2real-if*: $\text{ennreal } (\text{enn2real } r) = (\text{if } r = \text{top} \text{ then } 0 \text{ else } r)$
by (*auto intro!: ennreal-enn2real simp add: less-top*)

40.7 Coercion from *enat* to *ennreal*

definition *ennreal-of-enat* :: *enat* \Rightarrow *ennreal*

where

ennreal-of-enat $n = (\text{case } n \text{ of } \infty \Rightarrow \text{top} \mid \text{enat } n \Rightarrow \text{of-nat } n)$

```

declare [[coercion ennreal-of-enat]]
declare [[coercion of-nat :: nat  $\Rightarrow$  ennreal]]

lemma ennreal-of-enat-infty[simp]: ennreal-of-enat  $\infty$  =  $\infty$ 
  by (simp add: ennreal-of-enat-def)

lemma ennreal-of-enat-enat[simp]: ennreal-of-enat (enat n) = of-nat n
  by (simp add: ennreal-of-enat-def)

lemma ennreal-of-enat-0[simp]: ennreal-of-enat 0 = 0
  using ennreal-of-enat-enat[of 0] unfolding enat-0 by simp

lemma ennreal-of-enat-1[simp]: ennreal-of-enat 1 = 1
  using ennreal-of-enat-enat[of 1] unfolding enat-1 by simp

lemma ennreal-top-neq-of-nat[simp]: (top::ennreal)  $\neq$  of-nat i
  using ennreal-of-nat-neq-top[of i] by metis

lemma ennreal-of-enat-inj[simp]: ennreal-of-enat i = ennreal-of-enat j  $\longleftrightarrow$  i = j
  by (cases i j rule: enat.exhaust[case-product enat.exhaust]) auto

lemma ennreal-of-enat-le-iff[simp]: ennreal-of-enat m  $\leq$  ennreal-of-enat n  $\longleftrightarrow$  m
 $\leq$  n
  by (auto simp: ennreal-of-enat-def top-unique split: enat.split)

lemma of-nat-less-ennreal-of-nat[simp]: of-nat n  $\leq$  ennreal-of-enat x  $\longleftrightarrow$  of-nat
n  $\leq$  x
  by (cases x) (auto simp: of-nat-eq-enat)

lemma ennreal-of-enat-Sup: ennreal-of-enat (Sup X) = (SUP x  $\in$  X. ennreal-of-enat
x)
proof –
  have ennreal-of-enat (Sup X)  $\leq$  (SUP x  $\in$  X. ennreal-of-enat x)
    unfolding Sup-enat-def
  proof (clarsimp, intro conjI impI)
    fix x assume finite X X  $\neq$  {}
    then show ennreal-of-enat (Max X)  $\leq$  (SUP x  $\in$  X. ennreal-of-enat x)
      by (intro SUP-upper Max-in)
  next
    assume infinite X X  $\neq$  {}
    have  $\exists y \in X. r < \text{ennreal-of-enat } y$  if r: r < top for r
    proof –
      obtain n where n: r < of-nat n
        using ennreal-Ex-less-of-nat[OF r] ..
      have  $\neg (X \subseteq \text{enat } \{.. n\})$ 
        using  $\langle \text{infinite } X \rangle$  by (auto dest: finite-subset)
      then obtain x where x: x  $\in$  X  $\wedge$  x  $\notin$  enat ‘{..n}’
        by blast
      then have of-nat n  $\leq$  x

```

```

    by (cases x) (auto simp: of-nat-eq-enat)
  with x show ?thesis
    by (auto intro!: bexI[of - x] less-le-trans[OF n])
qed
then have (SUP x ∈ X. ennreal-of-enat x) = top
  by simp
then show top ≤ (SUP x ∈ X. ennreal-of-enat x)
  unfolding top-unique by simp
qed
then show ?thesis
  by (auto intro!: antisym Sup-least intro: Sup-upper)
qed

```

lemma *ennreal-of-enat-eSuc[simp]: ennreal-of-enat (eSuc x) = 1 + ennreal-of-enat x*

```

  by (cases x) (auto simp: eSuc-enat)

```

lemma *ennreal-of-enat-plus[simp]: <ennreal-of-enat (a+b) = ennreal-of-enat a + ennreal-of-enat b>*

proof (*induct a*)

```

  case (enat nat)
  with enat.simps show ?case
    by (smt (verit, del-Inst) add.commute add-top-left-ennreal enat.exhaust enat-defs(4)
ennreal-of-enat-def of-nat-add)
qed auto

```

lemma *sum-ennreal-of-enat[simp]: (∑ i ∈ I. ennreal-of-enat (f i)) = ennreal-of-enat (sum f I)*

```

  by (induct I rule: infinite-finite-induct) (auto simp: sum-nonneg)

```

40.8 Topology on *ennreal*

lemma *enn2ereal-Iio: enn2ereal - ‘{..} = (if 0 ≤ a then {..*

```

  using enn2ereal-nonneg
  by (cases a rule: ereal-ennreal-cases)
    (auto simp add: vimage-def set-eq-iff ennreal.enn2ereal-inverse less-ennreal.rep-eq
e2ennreal-def
      simp del: enn2ereal-nonneg
      intro: le-less-trans less-imp-le)

```

lemma *enn2ereal-Ioi: enn2ereal - ‘{a <..} = (if 0 ≤ a then {e2ennreal a <..} else UNIV)*

```

  by (cases a rule: ereal-ennreal-cases)
    (auto simp add: vimage-def set-eq-iff ennreal.enn2ereal-inverse less-ennreal.rep-eq
e2ennreal-def
      intro: less-le-trans)

```

instantiation *ennreal* :: *linear-continuum-topology*
begin

definition *open-ennreal* :: *ennreal set* \Rightarrow *bool*
where (*open* :: *ennreal set* \Rightarrow *bool*) = *generate-topology* (*range lessThan* \cup *range greaterThan*)

instance

proof

show $\exists a b :: \text{ennreal}. a \neq b$
using *zero-neg-one* **by** (*intro exI*)
show $\bigwedge x y :: \text{ennreal}. x < y \implies \exists z > x. z < y$
proof *transfer*
fix *x y* :: *ereal*
assume *: $0 \leq x$
assume $x < y$
from *dense[OF this]* **obtain** *z* **where** $x < z \wedge z < y$..
with * **show** $\exists z \in \text{Collect } ((\leq) 0). x < z \wedge z < y$
by (*intro bexI[of - z]*) *auto*
qed
qed (*rule open-ennreal-def*)

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*

proof (*rule continuous-on-subset*)

show *continuous-on* ($\{0..\} \cup \{..0\}$) *e2ennreal*
proof (*rule continuous-on-closed-Un*)
show *continuous-on* $\{0 ..\}$ *e2ennreal*
by (*simp add: continuous-onI-mono e2ennreal-mono enn2ereal-range*)
show *continuous-on* $\{.. 0\}$ *e2ennreal*
by (*metis atMost-iff continuous-on-cong continuous-on-const e2ennreal-neg*)
qed *auto*
qed *auto*

lemma *continuous-at-e2ennreal*: *continuous (at x within A) e2ennreal*

using *continuous-on-e2ennreal continuous-on-imp-continuous-within top.extremum*
by *blast*

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*

by (*rule continuous-on-generate-topology[OF open-generated-order]*)
(auto simp add: enn2ereal-Iio enn2ereal-Ioi)

lemma *continuous-at-enn2ereal*: *continuous (at x within A) enn2ereal*

by (*meson UNIV-I continuous-at-imp-continuous-at-within*
continuous-on-enn2ereal continuous-on-eq-continuous-within)

lemma *sup-continuous-e2ennreal[order-continuous-intros]*:

```

assumes  $f$ : sup-continuous  $f$  shows sup-continuous  $(\lambda x. e2ennreal (f x))$ 
proof (rule sup-continuous-compose[OF -  $f$ ])
  show sup-continuous  $e2ennreal$ 
    by (simp add: continuous-at-e2ennreal continuous-at-left-imp-sup-continuous
e2ennreal-mono mono-def)
qed

```

```

lemma sup-continuous-enn2ereal[order-continuous-intros]:
  assumes  $f$ : sup-continuous  $f$  shows sup-continuous  $(\lambda x. enn2ereal (f x))$ 
proof (rule sup-continuous-compose[OF -  $f$ ])
  show sup-continuous  $enn2ereal$ 
    by (simp add: continuous-at-enn2ereal continuous-at-left-imp-sup-continuous less-eq-ennreal.rep-eq
mono-def)
qed

```

```

lemma sup-continuous-mult-left-ennreal':
  fixes  $c :: ennreal$ 
  shows sup-continuous  $(\lambda x. c * x)$ 
  unfolding sup-continuous-def
  by transfer (auto simp: SUP-ereal-mult-left max.absorb2 SUP-upper2)

```

```

lemma sup-continuous-mult-left-ennreal[order-continuous-intros]:
  sup-continuous  $f \implies \text{sup-continuous } (\lambda x. c * f x :: ennreal)$ 
  by (rule sup-continuous-compose[OF sup-continuous-mult-left-ennreal'])

```

```

lemma sup-continuous-mult-right-ennreal[order-continuous-intros]:
  sup-continuous  $f \implies \text{sup-continuous } (\lambda x. f x * c :: ennreal)$ 
  using sup-continuous-mult-left-ennreal[of f c] by (simp add: mult.commute)

```

```

lemma sup-continuous-divide-ennreal[order-continuous-intros]:
  fixes  $f g :: 'a :: complete\_lattice \Rightarrow ennreal$ 
  shows sup-continuous  $f \implies \text{sup-continuous } (\lambda x. f x / c)$ 
  unfolding divide-ennreal-def by (rule sup-continuous-mult-right-ennreal)

```

```

lemma transfer-enn2ereal-continuous-on [transfer-rule]:
  rel-fun  $(=)$  (rel-fun (rel-fun  $(=)$  pcr-ennreal)  $(=)$ ) continuous-on continuous-on
proof –
  have continuous-on  $A f$  if continuous-on  $A (\lambda x. enn2ereal (f x))$  for  $A$  and  $f ::$ 
   $'a \Rightarrow ennreal$ 
    using continuous-on-compose2[OF continuous-on-e2ennreal[of {0..}] that]
    by (auto simp: ennreal.enn2ereal-inverse subset-eq e2ennreal-def max-absorb2)
  moreover
  have continuous-on  $A (\lambda x. enn2ereal (f x))$  if continuous-on  $A f$  for  $A$  and  $f ::$ 
   $'a \Rightarrow ennreal$ 
    using continuous-on-compose2[OF continuous-on-enn2ereal that] by auto
  ultimately
  show ?thesis
    by (auto simp add: rel-fun-def ennreal.pcr-cr-eq cr-ennreal-def)
qed

```

lemma *transfer-sup-continuous*[*transfer-rule*]:
 (*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *sup-continuous sup-continuous*
proof (*safe intro!*: *rel-funI* *dest!*: *rel-fun-eq-pcr-ennreal*[*THEN iffD1*])
 show *sup-continuous* (*enn2ereal* \circ *f*) \implies *sup-continuous* *f* **for** *f* :: '*a* \Rightarrow -
 using *sup-continuous-e2ennreal*[*of* *enn2ereal* \circ *f*] **by** *simp*
 show *sup-continuous* *f* \implies *sup-continuous* (*enn2ereal* \circ *f*) **for** *f* :: '*a* \Rightarrow -
 using *sup-continuous-enn2ereal*[*of* *f*] **by** (*simp add: comp-def*)
qed

lemma *continuous-on-ennreal*[*tendsto-intros*]:
continuous-on *A* *f* \implies *continuous-on* *A* ($\lambda x.$ *ennreal* (*f* *x*))
by *transfer* (*auto intro!*: *continuous-on-max continuous-on-const continuous-on-ereal*)

lemma *tendsto-ennrealD*:
 assumes *lim*: ($\lambda x.$ *ennreal* (*f* *x*)) \longrightarrow *ennreal* *x*) *F*
 assumes *: $\forall_F x$ in *F*. $0 \leq f\ x$ **and** *x*: $0 \leq x$
 shows (*f* \longrightarrow *x*) *F*
proof -
 have ($\lambda x.$ *enn2ereal* (*ennreal* (*f* *x*))) \longrightarrow *enn2ereal* (*ennreal* *x*) *F*
 \longleftrightarrow (*f* \longrightarrow *enn2ereal* (*ennreal* *x*)) *F*
 using * *eventually-mono*
by (*intro tendsto-cong*) *fastforce*
 then show ?thesis
 using *assms*(1) *continuous-at-enn2ereal isCont-tendsto-compose* *x* **by** *fastforce*
qed

lemma *tendsto-ennreal-iff* [*simp*]:
 $\langle (\lambda x.$ *ennreal* (*f* *x*)) \longrightarrow *ennreal* *x*) *F* \longleftrightarrow (*f* \longrightarrow *x*) *F* **(is** $\langle ?P \longleftrightarrow ?Q \rangle$)
if $\langle \forall_F x$ in *F*. $0 \leq f\ x \rangle \langle 0 \leq x \rangle$
proof
 assume $\langle ?P \rangle$
 then show $\langle ?Q \rangle$
 using *that* **by** (*rule tendsto-ennrealD*)
next
 assume $\langle ?Q \rangle$
 have \langle *continuous-on* *UNIV* *ereal* \rangle
 using *continuous-on-ereal* [*of - id*] **by** *simp*
 then have \langle *continuous-on* *UNIV* (*e2ennreal* \circ *ereal*) \rangle
by (*rule continuous-on-compose*) (*simp-all add: continuous-on-e2ennreal*)
 then have $\langle (\lambda x.$ (*e2ennreal* \circ *ereal*) (*f* *x*)) \longrightarrow (*e2ennreal* \circ *ereal*) *x*) *F* \rangle
 using $\langle ?Q \rangle$ **by** (*rule continuous-on-tendsto-compose*) *simp-all*
 then show $\langle ?P \rangle$
by (*simp flip: e2ennreal-ereal*)
qed

lemma *tendsto-enn2ereal-iff*[*simp*]: ($\lambda i.$ *enn2ereal* (*f* *i*)) \longrightarrow *enn2ereal* *x*) *F* \longleftrightarrow
 (*f* \longrightarrow *x*) *F*
 using *continuous-on-enn2ereal*[*THEN continuous-on-tendsto-compose, of f x F*]

continuous-on-e2ennreal[*THEN continuous-on-tendsto-compose*, of $\lambda x. \text{enn2ereal } (f x) \text{ enn2ereal } x \text{ F UNIV}$]

by *auto*

lemma *ennreal-tendsto-0-iff*: $(\bigwedge n. f n \geq 0) \implies ((\lambda n. \text{ennreal } (f n)) \longrightarrow 0)$
 $\iff (f \longrightarrow 0)$

by (*metis (mono-tags) ennreal-0 eventuallyI order-refl tendsto-ennreal-iff*)

lemma *continuous-on-add-ennreal*:

fixes $f g :: 'a::\text{topological-space} \Rightarrow \text{ennreal}$

shows $\text{continuous-on } A f \implies \text{continuous-on } A g \implies \text{continuous-on } A (\lambda x. f x + g x)$

by (*transfer fixing: A (auto intro!: tendsto-add-ereal-nonneg simp: continuous-on-def)*)

lemma *continuous-on-inverse-ennreal*[*continuous-intros*]:

fixes $f :: 'a::\text{topological-space} \Rightarrow \text{ennreal}$

shows $\text{continuous-on } A f \implies \text{continuous-on } A (\lambda x. \text{inverse } (f x))$

proof (*transfer fixing: A*)

show $\text{pred-fun top } ((\leq) 0) f \implies \text{continuous-on } A (\lambda x. \text{inverse } (f x))$ **if** *continuous-on A f*

for $f :: 'a \Rightarrow \text{ereal}$

using *continuous-on-compose2[OF continuous-on-inverse-ereal that]* **by** (*auto simp: subset-eq*)

qed

instance *ennreal :: topological-comm-monoid-add*

proof

show $((\lambda x. \text{fst } x + \text{snd } x) \longrightarrow a + b) (\text{nhds } a \times_F \text{nhds } b)$ **for** $a b :: \text{ennreal}$

using *continuous-on-add-ennreal[of UNIV fst snd]*

using *tendsto-at-iff-tendsto-nhds[symmetric, of $\lambda x::(\text{ennreal} \times \text{ennreal}). \text{fst } x + \text{snd } x$]*

by (*auto simp: continuous-on-eq-continuous-at*)

(*simp add: isCont-def nhds-prod[symmetric]*)

qed

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:

fixes $f g :: 'a::\text{complete-lattice} \Rightarrow \text{ennreal}$

shows $\text{sup-continuous } f \implies \text{sup-continuous } g \implies \text{sup-continuous } (\lambda x. f x + g x)$

by *transfer (auto intro!: sup-continuous-add)*

lemma *ennreal-suminf-lessD*: $(\sum i. f i :: \text{ennreal}) < x \implies f i < x$

using *le-less-trans[OF sum-le-suminf[OF summableI, of $\{i\} f$]]* **by** *simp*

lemma *sums-ennreal*[*simp*]: $(\bigwedge i. 0 \leq f i) \implies 0 \leq x \implies (\lambda i. \text{ennreal } (f i)) \text{ sums } \text{ennreal } x \iff f \text{ sums } x$

unfolding *sums-def* **by** (*simp add: always-eventually sum-nonneg*)

lemma *summable-suminf-not-top*: $(\bigwedge i. 0 \leq f i) \implies (\sum i. \text{ennreal } (f i)) \neq \text{top} \implies$

summable f

using *summable-sums*[*OF summableI*, *of* $\lambda i. \text{ennreal } (f\ i)$]
by (*cases* $\sum i. \text{ennreal } (f\ i)$ *rule: ennreal-cases*)
 (*auto simp: summable-def*)

lemma *suminf-ennreal[simp]*:

$(\bigwedge i. 0 \leq f\ i) \implies (\sum i. \text{ennreal } (f\ i)) \neq \text{top} \implies (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } (\sum i. f\ i)$
by (*rule sums-unique[symmetric]*) (*simp add: summable-suminf-not-top suminf-nonneg summable-sums*)

lemma *sums-enn2ereal[simp]*: $(\lambda i. \text{enn2ereal } (f\ i)) \text{ sums } \text{enn2ereal } x \longleftrightarrow f \text{ sums } x$

unfolding *sums-def* **by** (*simp add: always-eventually sum-nonneg*)

lemma *suminf-enn2ereal[simp]*: $(\sum i. \text{enn2ereal } (f\ i)) = \text{enn2ereal } (\text{suminf } f)$

by (*metis summableI summable-sums sums-enn2ereal sums-unique*)

lemma *transfer-e2ennreal-suminf [transfer-rule]*: *rel-fun* (*rel-fun* (=) *pcr-ennreal*)
pcr-ennreal suminf suminf

by (*auto simp: rel-funI rel-fun-eq-pcr-ennreal comp-def*)

lemma *ennreal-suminf-cmult[simp]*: $(\sum i. r * f\ i) = r * (\sum i. f\ i::\text{ennreal})$

by *transfer* (*auto intro!: suminf-cmult-ereal*)

lemma *ennreal-suminf-multc[simp]*: $(\sum i. f\ i * r) = (\sum i. f\ i::\text{ennreal}) * r$

using *ennreal-suminf-cmult[of r f]* **by** (*simp add: ac-simps*)

lemma *ennreal-suminf-divide[simp]*: $(\sum i. f\ i / r) = (\sum i. f\ i::\text{ennreal}) / r$

by (*simp add: divide-ennreal-def*)

lemma *ennreal-suminf-neq-top*: *summable f* $\implies (\bigwedge i. 0 \leq f\ i) \implies (\sum i. \text{ennreal } (f\ i)) \neq \text{top}$

using *sums-ennreal[of f suminf f]*

by (*simp add: suminf-nonneg flip: sums-unique summable-sums-iff del: sums-ennreal*)

lemma *suminf-ennreal-eq*:

$(\bigwedge i. 0 \leq f\ i) \implies f \text{ sums } x \implies (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } x$

using *suminf-nonneg[of f]* *sums-unique[of f x]*

by (*intro sums-unique[symmetric]*) (*auto simp: summable-sums-iff*)

lemma *ennreal-suminf-bound-add*:

fixes *f* :: *nat* \Rightarrow *ennreal*

shows $(\bigwedge N. (\sum n < N. f\ n) + y \leq x) \implies \text{suminf } f + y \leq x$

by *transfer* (*auto intro!: suminf-bound-add*)

lemma *ennreal-suminf-SUP-eq-directed*:

fixes *f* :: '*a* \Rightarrow *nat* \Rightarrow *ennreal*

assumes *: $\bigwedge N\ i\ j. i \in I \implies j \in I \implies \text{finite } N \implies \exists k \in I. \forall n \in N. f\ i\ n \leq f\ k$

```

 $n \wedge f j n \leq f k n$ 
shows  $(\sum n. \text{SUP } i \in I. f i n) = (\text{SUP } i \in I. \sum n. f i n)$ 
proof cases
  assume  $I \neq \{\}$ 
  then obtain  $i$  where  $i \in I$  by auto
  from  $*$  show ?thesis
    by (transfer fixing: I)
      (auto simp: max-absorb2 SUP-upper2[OF  $\langle i \in I \rangle$ ] suminf-nonneg summable-ereal-pos
 $\langle I \neq \{\} \rangle$ 
        intro!: suminf-SUP-eq-directed)
qed (simp add: bot-ennreal)

lemma INF-ennreal-add-const:
  fixes  $f g :: \text{nat} \Rightarrow \text{ennreal}$ 
  shows  $(\text{INF } i. f i + c) = (\text{INF } i. f i) + c$ 
  using continuous-at-Inf-mono[of  $\lambda x. x + c$  f'UNIV]
  using continuous-add[of at-right (Inf (range f)), of  $\lambda x. x \lambda x. c$ ]
  by (auto simp: mono-def image-comp)

lemma INF-ennreal-const-add:
  fixes  $f g :: \text{nat} \Rightarrow \text{ennreal}$ 
  shows  $(\text{INF } i. c + f i) = c + (\text{INF } i. f i)$ 
  using INF-ennreal-add-const[of f c] by (simp add: ac-simps)

lemma SUP-mult-left-ennreal:  $c * (\text{SUP } i \in I. f i) = (\text{SUP } i \in I. c * f i :: \text{ennreal})$ 
proof cases
  assume  $I \neq \{\}$  then show ?thesis
    by transfer (auto simp add: SUP-ereal-mult-left max-absorb2 SUP-upper2)
qed (simp add: bot-ennreal)

lemma SUP-mult-right-ennreal:  $(\text{SUP } i \in I. f i) * c = (\text{SUP } i \in I. f i * c :: \text{ennreal})$ 
  using SUP-mult-left-ennreal by (simp add: mult.commute)

lemma SUP-divide-ennreal:  $(\text{SUP } i \in I. f i) / c = (\text{SUP } i \in I. f i / c :: \text{ennreal})$ 
  using SUP-mult-right-ennreal by (simp add: divide-ennreal-def)

lemma ennreal-SUP-of-nat-eq-top:  $(\text{SUP } x. \text{of-nat } x :: \text{ennreal}) = \text{top}$ 
proof (intro antisym top-greatest le-SUP-iff[THEN iffD2] allI impI)
  fix  $y :: \text{ennreal}$  assume  $y < \text{top}$ 
  then obtain  $r$  where  $y = \text{ennreal } r$ 
  by (cases y rule: ennreal-cases) auto
  then show  $\exists i \in \text{UNIV}. y < \text{of-nat } i$ 
    using reals-Archimedean2[of max 1 r] zero-less-one
    by (simp add: ennreal-Ex-less-of-nat)
qed

lemma ennreal-SUP-eq-top:
  fixes  $f :: 'a \Rightarrow \text{ennreal}$ 
  assumes  $\bigwedge n. \exists i \in I. \text{of-nat } n \leq f i$ 

```

```

  shows  $(\text{SUP } i \in I. f \ i) = \text{top}$ 
proof -
  have  $(\text{SUP } x. \text{ of-nat } x :: \text{ennreal}) \leq (\text{SUP } i \in I. f \ i)$ 
    using assms by (auto intro!: SUP-least intro: SUP-upper2)
  then show ?thesis
    by (auto simp: ennreal-SUP-of-nat-eq-top top-unique)
qed

lemma ennreal-INF-const-minus:
  fixes  $f :: 'a \Rightarrow \text{ennreal}$ 
  shows  $I \neq \{\} \implies (\text{SUP } x \in I. c - f \ x) = c - (\text{INF } x \in I. f \ x)$ 
  by (transfer fixing: I)
    (simp add: sup-max[symmetric] SUP-sup-const1 SUP-ereal-minus-right del:
sup-ereal-def)

lemma of-nat-Sup-ennreal:
  assumes  $A \neq \{\}$  bdd-above  $A$ 
  shows  $\text{of-nat } (\text{Sup } A) = (\text{SUP } a \in A. \text{ of-nat } a :: \text{ennreal})$ 
proof (intro antisym)
  show  $(\text{SUP } a \in A. \text{ of-nat } a :: \text{ennreal}) \leq \text{of-nat } (\text{Sup } A)$ 
    by (intro SUP-least of-nat-mono) (auto intro: cSup-upper assms)
  have  $\text{Sup } A \in A$ 
    using assms by (auto simp: Sup-nat-def bdd-above-nat)
  then show  $\text{of-nat } (\text{Sup } A) \leq (\text{SUP } a \in A. \text{ of-nat } a :: \text{ennreal})$ 
    by (intro SUP-upper)
qed

lemma ennreal-tendsto-const-minus:
  fixes  $g :: 'a \Rightarrow \text{ennreal}$ 
  assumes ae:  $\forall_F x \text{ in } F. g \ x \leq c$ 
  assumes g:  $((\lambda x. c - g \ x) \longrightarrow 0) \ F$ 
  shows  $(g \longrightarrow c) \ F$ 
proof (cases c rule: ennreal-cases)
  case top with tendsto-unique[OF - g, of top] show ?thesis
    by (cases  $F = \text{bot}$ ) auto
next
  case (real r)
  then have  $\forall x. \exists q \geq 0. g \ x \leq c \longrightarrow (g \ x = \text{ennreal } q \wedge q \leq r)$ 
    by (auto simp: le-ennreal-iff)
  then obtain f where  $0 \leq f \ x \wedge g \ x = \text{ennreal } (f \ x) \wedge f \ x \leq r$  if  $g \ x \leq c$  for x
    by metis
  from ae have ae2:  $\forall_F x \text{ in } F. c - g \ x = \text{ennreal } (r - f \ x) \wedge f \ x \leq r \wedge g \ x =$ 
 $\text{ennreal } (f \ x) \wedge 0 \leq f \ x$ 
  proof eventually-elim
    fix x assume  $g \ x \leq c$  with  $*[of \ x] \langle 0 \leq r \rangle$  show  $c - g \ x = \text{ennreal } (r - f \ x)$ 
 $\wedge f \ x \leq r \wedge g \ x = \text{ennreal } (f \ x) \wedge 0 \leq f \ x$ 
    by (auto simp: real ennreal-minus)
  qed
  with g have  $((\lambda x. \text{ennreal } (r - f \ x)) \longrightarrow \text{ennreal } 0) \ F$ 

```

```

  by (auto simp add: tendsto-cong eventually-conj-iff)
with ae2 have  $((\lambda x. r - f x) \longrightarrow 0) F$ 
  by (subst (asm) tendsto-ennreal-iff) (auto elim: eventually-mono)
then have  $(f \longrightarrow r) F$ 
  by (rule Lim-transform2[OF tendsto-const])
with ae2 have  $((\lambda x. \text{ennreal } (f x)) \longrightarrow \text{ennreal } r) F$ 
  by (subst tendsto-ennreal-iff) (auto elim: eventually-mono simp: real)
with ae2 show ?thesis
  by (auto simp: real tendsto-cong eventually-conj-iff)
qed

```

lemma *ennreal-SUP-add*:

```

  fixes  $f g :: \text{nat} \Rightarrow \text{ennreal}$ 
  shows  $\text{incseq } f \Longrightarrow \text{incseq } g \Longrightarrow (\text{SUP } i. f i + g i) = \text{Sup } (f \text{ ‘ UNIV}) + \text{Sup } (g \text{ ‘ UNIV})$ 
  unfolding incseq-def le-fun-def
  by transfer
  (simp add: SUP-ereal-add incseq-def le-fun-def max-absorb2 SUP-upper2)

```

lemma *ennreal-SUP-sum*:

```

  fixes  $f :: 'a \Rightarrow \text{nat} \Rightarrow \text{ennreal}$ 
  shows  $(\bigwedge i. i \in I \Longrightarrow \text{incseq } (f i)) \Longrightarrow (\text{SUP } n. \sum_{i \in I} f i n) = (\sum_{i \in I} \text{SUP } n. f i n)$ 
  unfolding incseq-def
  by transfer
  (simp add: SUP-ereal-sum incseq-def SUP-upper2 max-absorb2 sum-nonneg)

```

lemma *ennreal-liminf-minus*:

```

  fixes  $f :: \text{nat} \Rightarrow \text{ennreal}$ 
  shows  $(\bigwedge n. f n \leq c) \Longrightarrow \text{liminf } (\lambda n. c - f n) = c - \text{limsup } f$ 
  apply transfer
  apply (simp add: ereal-diff-positive liminf-ereal-cminus)
  by (metis max.absorb2 ereal-diff-positive Limsup-bounded eventually-sequentiallyI)

```

lemma *ennreal-continuous-on-cmult*:

```

   $(c :: \text{ennreal}) < \text{top} \Longrightarrow \text{continuous-on } A f \Longrightarrow \text{continuous-on } A (\lambda x. c * f x)$ 
  by (transfer fixing: A) (auto intro: continuous-on-cmult-ereal)

```

lemma *ennreal-tendsto-cmult*:

```

   $(c :: \text{ennreal}) < \text{top} \Longrightarrow (f \longrightarrow x) F \Longrightarrow ((\lambda x. c * f x) \longrightarrow c * x) F$ 
  by (rule continuous-on-tendsto-compose[where g=f, OF ennreal-continuous-on-cmult,
  where s=UNIV])
  (auto simp: continuous-on-id)

```

lemma *tendsto-ennrealI*[intro, simp, tendsto-intros]:

```

   $(f \longrightarrow x) F \Longrightarrow ((\lambda x. \text{ennreal } (f x)) \longrightarrow \text{ennreal } x) F$ 
  by (auto simp: ennreal-def
    intro!: continuous-on-tendsto-compose[OF continuous-on-e2ennreal[of UNIV]] tendsto-max)

```

```

lemma tendsto-enn2erealI [tendsto-intros]:
  assumes ( $f \longrightarrow l$ )  $F$ 
  shows  $((\lambda i. \text{enn2ereal}(f\ i)) \longrightarrow \text{enn2ereal } l) F$ 
using tendsto-enn2ereal-iff assms by auto

lemma tendsto-e2ennrealI [tendsto-intros]:
  assumes ( $f \longrightarrow l$ )  $F$ 
  shows  $((\lambda i. \text{e2ennreal}(f\ i)) \longrightarrow \text{e2ennreal } l) F$ 
proof –
  have  $*$ :  $\text{e2ennreal} (\max x\ 0) = \text{e2ennreal } x$  for  $x$ 
    by (simp add: e2ennreal-def max.commute)
  have  $((\lambda i. \max (f\ i)\ 0) \longrightarrow \max l\ 0) F$ 
    using assms by (intro tendsto-intros) auto
  then have  $((\lambda i. \text{enn2ereal}(\text{e2ennreal} (\max (f\ i)\ 0))) \longrightarrow \text{enn2ereal} (\text{e2ennreal} (\max l\ 0))) F$ 
    by (subst enn2ereal-e2ennreal, auto) +
  then have  $((\lambda i. \text{e2ennreal} (\max (f\ i)\ 0)) \longrightarrow \text{e2ennreal} (\max l\ 0)) F$ 
    using tendsto-enn2ereal-iff by auto
  then show ?thesis
    unfolding  $*$  by auto
qed

lemma ennreal-suminf-minus:
  fixes  $f\ g :: \text{nat} \Rightarrow \text{ennreal}$ 
  shows  $(\bigwedge i. g\ i \leq f\ i) \implies \text{suminf } f \neq \text{top} \implies \text{suminf } g \neq \text{top} \implies (\sum i. f\ i - g\ i) = \text{suminf } f - \text{suminf } g$ 
    by transfer
    (auto simp add: ereal-diff-positive suminf-le-pos top-ereal-def intro!: suminf-ereal-minus)

lemma ennreal-Sup-countable-SUP:
   $A \neq \{\}$   $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{incseq } f \wedge \text{range } f \subseteq A \wedge \text{Sup } A = (\text{SUP } i. f\ i)$ 
  unfolding incseq-def
  apply transfer
  subgoal for  $A$ 
    using Sup-countable-SUP[of A]
    by (force simp add: incseq-def[symmetric] SUP-upper2 image-subset-iff Sup-upper2 cong: conj-cong)
  done

lemma ennreal-Inf-countable-INF:
   $A \neq \{\}$   $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{decseq } f \wedge \text{range } f \subseteq A \wedge \text{Inf } A = (\text{INF } i. f\ i)$ 
  unfolding decseq-def
  apply transfer
  subgoal for  $A$ 
    using Inf-countable-INF[of A] by (simp flip: decseq-def) blast
  done

```

lemma *ennreal-SUP-countable-SUP*:

$A \neq \{\}$ $\implies \exists f :: \text{nat} \Rightarrow \text{ennreal}. \text{range } f \subseteq g'A \wedge \text{Sup } (g \text{ ' } A) = \text{Sup } (f \text{ ' } \text{UNIV})$
using *ennreal-Sup-countable-SUP* [of $g'A$] **by** *auto*

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n :: \text{nat}. \text{of-nat } n :: \text{ennreal}) \longrightarrow \text{top}$

using *LIMSEQ-SUP*[of $\text{of-nat} :: \text{nat} \Rightarrow \text{ennreal}$]
by (*simp add: ennreal-SUP-of-nat-eq-top incseq-def*)

lemma *SUP-sup-continuous-ennreal*:

fixes $f :: \text{ennreal} \Rightarrow 'a :: \text{complete-lattice}$

assumes f : *sup-continuous* f **and** $I \neq \{\}$

shows $(\text{SUP } i \in I. f (g i)) = f (\text{SUP } i \in I. g i)$

proof (*rule antisym*)

show $(\text{SUP } i \in I. f (g i)) \leq f (\text{SUP } i \in I. g i)$

by (*rule mono-SUP*[*OF sup-continuous-mono*[*OF f*]])

from *ennreal-Sup-countable-SUP*[of $g'I$] $\langle I \neq \{\} \rangle$

obtain $M :: \text{nat} \Rightarrow \text{ennreal}$ **where** *incseq* M **and** M : *range* $M \subseteq g \text{ ' } I$ **and** *eq*:
 $(\text{SUP } i \in I. g i) = (\text{SUP } i. M i)$

by *auto*

have $f (\text{SUP } i \in I. g i) = f (\text{SUP } i \in \text{range } M. f i)$

unfolding *eq sup-continuousD*[*OF f* $\langle \text{mono } M \rangle$] **by** (*simp add: image-comp*)

also have $\dots \leq (\text{SUP } i \in I. f (g i))$

by (*smt (verit) M SUP-le-iff dual-order.refl image-iff subsetD*)

finally show $f (\text{SUP } i \in I. g i) \leq (\text{SUP } i \in I. f (g i))$.

qed

lemma *ennreal-suminf-SUP-eq*:

fixes $f :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{ennreal}$

shows $(\bigwedge i. \text{incseq } (\lambda n. f n i)) \implies (\sum i. \text{SUP } n. f n i) = (\text{SUP } n. \sum i. f n i)$

by (*metis ennreal-suminf-SUP-eq-directed incseqD nat-le-linear*)

lemma *ennreal-SUP-add-left*:

fixes $c :: \text{ennreal}$

shows $I \neq \{\} \implies (\text{SUP } i \in I. f i + c) = (\text{SUP } i \in I. f i) + c$

apply *transfer*

apply (*simp add: SUP-ereal-add-left*)

by (*metis SUP-upper all-not-in-conv add-increasing2 max.absorb2 max.bounded-iff*)

lemma *ennreal-SUP-const-minus*:

fixes $f :: 'a \Rightarrow \text{ennreal}$

shows $I \neq \{\} \implies c < \text{top} \implies (\text{INF } x \in I. c - f x) = c - (\text{SUP } x \in I. f x)$

apply (*transfer fixing: I*)

unfolding *ex-in-conv*[*symmetric*]

apply (*auto simp add: SUP-upper2 sup-absorb2 simp flip: sup-ereal-def*)

apply (*subst INF-ereal-minus-right*[*symmetric*])

apply (*auto simp del: sup-ereal-def simp add: sup-INF*)

done

```

lemma isCont-ennreal[simp]:  $\langle \text{isCont } \text{ennreal } x \rangle$ 
  unfolding continuous-within tendsto-def
  using tendsto-ennrealI topological-tendstoD
  by (blast intro: sequentially-imp-eventually-within)

lemma isCont-ennreal-of-enat[simp]:  $\langle \text{isCont } \text{ennreal-of-enat } x \rangle$ 
proof –
  have continuous-at-open:
    — Copied lemma from HOL-Analysis to avoid dependency.
     $\text{continuous } (\text{at } x) f \iff (\forall t. \text{open } t \wedge f x \in t \implies (\exists s. \text{open } s \wedge x \in s \wedge$ 
     $(\forall x' \in s. (f x') \in t)))$  for  $f :: \langle \text{enat} \Rightarrow 'z :: \text{topological-space} \rangle$ 
    unfolding continuous-within-topological [of x UNIV f]
    unfolding imp-conjL
    by (intro all-cong imp-cong ex-cong conj-cong refl) auto
  show ?thesis
proof (subst continuous-at-open, intro allI impI, cases  $\langle x = \infty \rangle$ )
  case True

    fix  $t$  assume  $\langle \text{open } t \wedge \text{ennreal-of-enat } x \in t \rangle$ 
    then have  $\langle \exists y < \infty. \{y < .. \infty\} \subseteq t \rangle$ 
      by (rule-tac open-left[where y=0]) (auto simp: True)
    then obtain  $y$  where  $\langle \{y < ..\} \subseteq t \rangle$  and  $\langle y \neq \infty \rangle$ 
      by fastforce
    from  $\langle y \neq \infty \rangle$ 
    obtain  $x'$  where  $x'y: \langle \text{ennreal-of-enat } x' > y \rangle$  and  $\langle x' \neq \infty \rangle$ 
      by (metis enat.simps(3) ennreal-Ex-less-of-nat ennreal-of-enat-enat infinity-ennreal-def top.not-eq-extremum)
    define  $s$  where  $\langle s = \{x' < ..\} \rangle$ 
    have  $\langle \text{open } s \rangle$ 
      by (simp add: s-def)
    moreover have  $\langle x \in s \rangle$ 
      by (simp add:  $\langle x' \neq \infty \rangle$  s-def True)
    moreover have  $\langle \text{ennreal-of-enat } z \in t \rangle$  if  $\langle z \in s \rangle$  for  $z$ 
      by (metis x'y  $\langle \{y < ..\} \subseteq t \rangle$  ennreal-of-enat-le-iff greaterThan-iff le-less-trans less-imp-le not-less s-def subsetD that)
    ultimately show  $\langle \exists s. \text{open } s \wedge x \in s \wedge (\forall z \in s. \text{ennreal-of-enat } z \in t) \rangle$ 
      by auto
  next
  case False
    fix  $t$  assume asm:  $\langle \text{open } t \wedge \text{ennreal-of-enat } x \in t \rangle$ 
    define  $s$  where  $\langle s = \{x\} \rangle$ 
    have  $\langle \text{open } s \rangle$ 
      using False open-enat-iff s-def by blast
    then show  $\langle \exists s. \text{open } s \wedge x \in s \wedge (\forall z \in s. \text{ennreal-of-enat } z \in t) \rangle$ 
      using asm s-def by blast
  qed
qed

```

40.9 Approximation lemmas

lemma *INF-approx-ennreal*:
fixes $x::ennreal$ **and** $e::real$
assumes $e > 0$
assumes $x = (INF\ i \in A. f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A. f\ i < x + e$
using *INF-less-iff* **assms** **by** *fastforce*

lemma *SUP-approx-ennreal*:
fixes $x::ennreal$ **and** $e::real$
assumes $e > 0$ $A \neq \{\}$
assumes *SUP*: $x = (SUP\ i \in A. f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A. x < f\ i + e$
proof –
have $x < x + e$
using $\langle 0 < e \rangle$ $\langle x \neq \infty \rangle$ **by** (*cases* x) *auto*
also have $x + e = (SUP\ i \in A. f\ i + e)$
unfolding *SUP ennreal-SUP-add-left*[*OF* $\langle A \neq \{\} \rangle$] ..
finally show *?thesis*
unfolding *less-SUP-iff* .
qed

lemma *ennreal-approx-SUP*:
fixes $x::ennreal$
assumes *f-bound*: $\bigwedge i. i \in A \implies f\ i \leq x$
assumes *approx*: $\bigwedge e. (e::real) > 0 \implies \exists i \in A. x \leq f\ i + e$
shows $x = (SUP\ i \in A. f\ i)$
proof (*rule antisym*)
show $x \leq (SUP\ i \in A. f\ i)$
proof (*rule ennreal-le-epsilon*)
fix $e :: real$ **assume** $0 < e$
from *approx*[*OF this*] **obtain** i **where** $i \in A$ **and** $*$: $x \leq f\ i + ennreal\ e$
by *blast*
from $*$ **have** $x \leq f\ i + e$
by *simp*
also have $\dots \leq (SUP\ i \in A. f\ i) + e$
by (*intro add-mono* $\langle i \in A \rangle$ *SUP-upper order-reft*)
finally show $x \leq (SUP\ i \in A. f\ i) + e$.
qed
qed (*intro SUP-least f-bound*)

lemma *ennreal-approx-INF*:
fixes $x::ennreal$
assumes *f-bound*: $\bigwedge i. i \in A \implies x \leq f\ i$
assumes *approx*: $\bigwedge e. (e::real) > 0 \implies \exists i \in A. f\ i \leq x + e$
shows $x = (INF\ i \in A. f\ i)$
proof (*rule antisym*)


```

show ( $\text{INF } i \in A. f\ i \leq x$ )
proof (rule ennreal-le-epsilon)
  fix  $e :: \text{real}$  assume  $0 < e$ 
  from approx[OF this] obtain  $i$  where  $i \in A$   $f\ i \leq x + \text{ennreal } e$ 
  by blast
  then have ( $\text{INF } i \in A. f\ i \leq f\ i$ )
  by (intro INF-lower)
  also have  $\dots \leq x + e$ 
  by fact
  finally show ( $\text{INF } i \in A. f\ i \leq x + e$ ) .
qed
qed (intro INF-greatest f-bound)

```

```

lemma ennreal-approx-unit:
  ( $\bigwedge a :: \text{ennreal}. 0 < a \implies a < 1 \implies a * z \leq y \implies z \leq y$ )
  using SUP-mult-right-ennreal[of  $\lambda x. x \{0 < .. < 1\} z$ ]
  by (smt (verit) SUP-least Sup-greaterThanLessThan greaterThanLessThan-iff
    image-ident mult-1 zero-less-one)

```

```

lemma suminf-ennreal2:
  ( $\bigwedge i. 0 \leq f\ i \implies \text{summable } f \implies (\sum i. \text{ennreal } (f\ i)) = \text{ennreal } (\sum i. f\ i)$ )
  using suminf-ennreal-eq by blast

```

```

lemma less-top-ennreal:  $x < \text{top} \longleftrightarrow (\exists r \geq 0. x = \text{ennreal } r)$ 
by (cases x) auto

```

```

lemma enn2real-less-iff[simp]:  $x < \text{top} \implies \text{enn2real } x < c \longleftrightarrow x < c$ 
using ennreal-less-iff less-top-ennreal by auto

```

```

lemma enn2real-le-iff[simp]:  $\llbracket x < \text{top}; c > 0 \rrbracket \implies \text{enn2real } x \leq c \longleftrightarrow x \leq c$ 
by (cases x) auto

```

```

lemma enn2real-less:
  assumes  $\text{enn2real } e < r \ e \neq \text{top}$  shows  $e < \text{ennreal } r$ 
  using enn2real-less-iff assms top.not-eq-extremum by blast

```

```

lemma enn2real-le:
  assumes  $\text{enn2real } e \leq r \ e \neq \text{top}$  shows  $e \leq \text{ennreal } r$ 
  by (metis assms enn2real-less ennreal-enn2real-if eq-iff less-le)

```

```

lemma tendsto-top-iff-ennreal:
  fixes  $f :: 'a \Rightarrow \text{ennreal}$ 
  shows ( $f \longrightarrow \text{top}$ )  $F \longleftrightarrow (\forall l \geq 0. \text{eventually } (\lambda x. \text{ennreal } l < f\ x) F)$ 
  by (auto simp: less-top-ennreal order-tendsto-iff)

```

```

lemma ennreal-tendsto-top-eq-at-top:
  ( $(\lambda z. \text{ennreal } (f\ z)) \longrightarrow \text{top}$ )  $F \longleftrightarrow (\text{LIM } z\ F. f\ z :> \text{at-top})$ 
  unfolding filterlim-at-top-dense tendsto-top-iff-ennreal
  using ennreal-less-iff eventually-mono allE[of - max 0 -]

```

by (smt (verit) linorder-not-less order-refl order-trans)

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:

fixes $f :: - \Rightarrow \text{ennreal}$

shows $\text{Limsup } F f = 0 \implies (f \longrightarrow 0) F$

using *Liminf-le-Limsup*[of $F f$] *tendsto-iff-Liminf-eq-Limsup*[of $F f 0$]

by (cases $F = \text{bot}$) auto

lemma *diff-le-self-ennreal[simp]*: $a - b \leq (a::\text{ennreal})$

by (cases $a b$ rule: *ennreal2-cases*) (auto simp: *ennreal-minus*)

lemma *ennreal-ineq-diff-add*: $b \leq a \implies a = b + (a - b::\text{ennreal})$

by transfer (auto simp: *ereal-diff-positive max.absorb2 ereal-ineq-diff-add*)

lemma *ennreal-mult-strict-left-mono*: $(a::\text{ennreal}) < c \implies 0 < b \implies b < \text{top} \implies b * a < b * c$

by transfer (auto intro!: *ereal-mult-strict-left-mono*)

lemma *ennreal-between*: $0 < e \implies 0 < x \implies x < \text{top} \implies x - e < (x::\text{ennreal})$

by transfer (auto intro!: *ereal-between*)

lemma *minus-less-iff-ennreal*: $b < \text{top} \implies b \leq a \implies a - b < c \longleftrightarrow a < c + (b::\text{ennreal})$

by transfer

(auto simp: *top-ereal-def ereal-minus-less le-less*)

lemma *tendsto-zero-ennreal*:

assumes $ev: \bigwedge r. 0 < r \implies \forall_F x \text{ in } F. f x < \text{ennreal } r$

shows $(f \longrightarrow 0) F$

proof (rule *order-tendstoI*)

fix $e::\text{ennreal}$ assume $e > 0$

obtain $e'::\text{real}$ where $e' > 0$ *ennreal* $e' < e$

using $\langle 0 < e \rangle$ *dense*[of 0 if $e = \text{top}$ then 1 else (*enn2real* e)]

by (cases e) (auto simp: *ennreal-less-iff*)

from $ev[OF \langle e' > 0 \rangle]$ show $\forall_F x \text{ in } F. f x < e$

by *eventually-elim* (insert $\langle \text{ennreal } e' < e \rangle$, auto)

qed simp

lifting-update *ennreal.lifting*

lifting-forget *ennreal.lifting*

40.10 *ennreal* theorems

lemma *neg-top-trans*: fixes $x y :: \text{ennreal}$ shows $\llbracket y \neq \text{top}; x \leq y \rrbracket \implies x \neq \text{top}$

by (auto simp: *top-unique*)

lemma *diff-diff-ennreal*: fixes $a b :: \text{ennreal}$ shows $a \leq b \implies b \neq \infty \implies b - (b - a) = a$

by (cases $a b$ rule: *ennreal2-cases*) (auto simp: *ennreal-minus top-unique*)

lemma *ennreal-less-one-iff*[simp]: $\text{ennreal } x < 1 \longleftrightarrow x < 1$
by (cases $0 \leq x$) (auto simp: *ennreal-neg ennreal-less-iff simp flip: ennreal-1*)

lemma *SUP-const-minus-ennreal*:
fixes $f :: 'a \Rightarrow \text{ennreal}$ **shows** $I \neq \{\}$ $\implies (\text{SUP } x \in I. c - f x) = c - (\text{INF } x \in I. f x)$
including *ennreal.lifting*
by (*transfer fixing: I*)
 (*simp add: SUP-sup-distrib[symmetric] SUP-ereal-minus-right flip: sup-ereal-def*)

lemma *zero-minus-ennreal*[simp]: $0 - (a :: \text{ennreal}) = 0$
including *ennreal.lifting*
by *transfer (simp split: split-max)*

lemma *diff-diff-commute-ennreal*:
fixes $a \ b \ c :: \text{ennreal}$ **shows** $a - b - c = a - c - b$
by (cases $a \ b \ c$ rule: *ennreal3-cases*) (*simp-all add: ennreal-minus field-simps*)

lemma *diff-gr0-ennreal*: $b < (a :: \text{ennreal}) \implies 0 < a - b$
including *ennreal.lifting* **by** *transfer (auto simp: ereal-diff-gr0 ereal-diff-positive split: split-max)*

lemma *divide-le-posI-ennreal*:
fixes $x \ y \ z :: \text{ennreal}$
shows $x > 0 \implies z \leq x * y \implies z / x \leq y$
by (cases $x \ y \ z$ rule: *ennreal3-cases*)
 (*auto simp: divide-ennreal ennreal-mult[symmetric] field-simps top-unique*)

lemma *add-diff-eq-ennreal*:
fixes $x \ y \ z :: \text{ennreal}$
shows $z \leq y \implies x + (y - z) = x + y - z$
using *ennreal-diff-add-assoc* **by** *auto*

lemma *add-diff-inverse-ennreal*:
fixes $x \ y :: \text{ennreal}$ **shows** $x \leq y \implies x + (y - x) = y$
by (cases x) (*simp-all add: top-unique add-diff-eq-ennreal*)

lemma *add-diff-eq-iff-ennreal*[simp]:
fixes $x \ y :: \text{ennreal}$ **shows** $x + (y - x) = y \longleftrightarrow x \leq y$
by (*metis ennreal-ineq-diff-add le-iff-add*)

lemma *add-diff-le-ennreal*: $a + b - c \leq a + (b - c :: \text{ennreal})$
apply (cases $a \ b \ c$ rule: *ennreal3-cases*)
subgoal for $a' \ b' \ c'$
by (cases $0 \leq b' - c'$) (*simp-all add: ennreal-minus top-add ennreal-neg flip: ennreal-plus*)
apply (*simp-all add: top-add flip: ennreal-plus*)

done

lemma *diff-eq-0-ennreal*: $a < \text{top} \implies a \leq b \implies a - b = (0 :: \text{ennreal})$
using *ennreal-minus-pos-iff gr-zeroI not-less* **by** *blast*

lemma *diff-diff-ennreal'*: **fixes** $x\ y\ z :: \text{ennreal}$ **shows** $z \leq y \implies y - z \leq x \implies x - (y - z) = x + z - y$
by (*cases x*; *cases y*; *cases z*)
 (*auto simp add: top-add add-top minus-top-ennreal ennreal-minus top-unique simp flip: ennreal-plus*)

lemma *diff-diff-ennreal''*: **fixes** $x\ y\ z :: \text{ennreal}$
shows $z \leq y \implies x - (y - z) = (\text{if } y - z \leq x \text{ then } x + z - y \text{ else } 0)$
by (*cases x*; *cases y*; *cases z*)
 (*auto simp add: top-add add-top minus-top-ennreal ennreal-minus top-unique ennreal-neg simp flip: ennreal-plus*)

lemma *power-less-top-ennreal*: **fixes** $x :: \text{ennreal}$ **shows** $x \wedge n < \text{top} \longleftrightarrow x < \text{top} \vee n = 0$
using *power-eq-top-ennreal top.not-eq-extremum*
by *blast*

lemma *ennreal-divide-times*: $(a / b) * c = a * (c / b :: \text{ennreal})$
by (*simp add: mult.commute ennreal-times-divide*)

lemma *diff-less-top-ennreal*: $a - b < \text{top} \longleftrightarrow a < (\text{top} :: \text{ennreal})$
by (*cases a*; *cases b*) (*auto simp: ennreal-minus*)

lemma *divide-less-ennreal*: $b \neq 0 \implies b < \text{top} \implies a / b < c \longleftrightarrow a < (c * b :: \text{ennreal})$
by (*cases a*; *cases b*; *cases c*)
 (*auto simp: divide-ennreal ennreal-mult[symmetric] ennreal-less-iff field-simps ennreal-top-mult ennreal-top-divide*)

lemma *one-less-numeral[simp]*: $1 < (\text{numeral } n :: \text{ennreal}) \longleftrightarrow (\text{num.One} < n)$
by *simp*

lemma *divide-eq-1-ennreal*: $a / b = (1 :: \text{ennreal}) \longleftrightarrow (b \neq \text{top} \wedge b \neq 0 \wedge b = a)$
by (*cases a*; *cases b*; *cases b = 0*) (*auto simp: ennreal-top-divide divide-ennreal split: if-split-asm*)

lemma *ennreal-mult-cancel-left*: $(a * b = a * c) = (a = \text{top} \wedge b \neq 0 \wedge c \neq 0 \vee a = 0 \vee b = (c :: \text{ennreal}))$
by (*cases a*; *cases b*; *cases c*) (*auto simp: ennreal-mult[symmetric] ennreal-mult-top ennreal-top-mult*)

lemma *ennreal-minus-if*: $\text{ennreal } a - \text{ennreal } b = \text{ennreal } (\text{if } 0 \leq b \text{ then } (\text{if } b \leq a \text{ then } a - b \text{ else } 0) \text{ else } a)$

by (*auto simp: ennreal-minus ennreal-neg*)

lemma *ennreal-plus-if*: $\text{ennreal } a + \text{ennreal } b = \text{ennreal } (\text{if } 0 \leq a \text{ then } (\text{if } 0 \leq b \text{ then } a + b \text{ else } a) \text{ else } b)$
by (*auto simp: ennreal-neg*)

lemma *ennreal-diff-le-mono-left*: $a \leq b \implies a - c \leq (b::\text{ennreal})$
using *ennreal-mono-minus*[of 0 c a, THEN *order-trans*, of b] **by** *simp*

lemma *ennreal-minus-le-iff*: $a - b \leq c \iff (a \leq b + (c::\text{ennreal}) \wedge (a = \text{top} \wedge b = \text{top} \implies c = \text{top}))$
by (*cases a; cases b; cases c*)
(auto simp: top-unique top-add add-top ennreal-minus simp flip: ennreal-plus)

lemma *ennreal-le-minus-iff*: $a \leq b - c \iff (a + c \leq (b::\text{ennreal}) \vee (a = 0 \wedge b \leq c))$
by (*cases a; cases b; cases c*)
(auto simp: top-unique top-add add-top ennreal-minus ennreal-le-iff2 simp flip: ennreal-plus)

lemma *diff-add-eq-diff-diff-swap-ennreal*: $x - (y + z :: \text{ennreal}) = x - y - z$
by (*cases x; cases y; cases z*)
(auto simp: ennreal-minus-if add-top top-add simp flip: ennreal-plus)

lemma *diff-add-assoc2-ennreal*: $b \leq a \implies (a - b + c::\text{ennreal}) = a + c - b$
by (*cases a; cases b; cases c*)
(auto simp add: ennreal-minus-if ennreal-plus-if add-top top-add top-unique simp del: ennreal-plus)

lemma *diff-gt-0-iff-gt-ennreal*: $0 < a - b \iff (a = \text{top} \wedge b = \text{top} \vee b < (a::\text{ennreal}))$
by (*cases a; cases b*) *(auto simp: ennreal-minus-if ennreal-less-iff)*

lemma *diff-eq-0-iff-ennreal*: $(a - b::\text{ennreal}) = 0 \iff (a < \text{top} \wedge a \leq b)$
by (*cases a*) *(auto simp: ennreal-minus-eq-0 diff-eq-0-ennreal)*

lemma *add-diff-self-ennreal*: $a + (b - a::\text{ennreal}) = (\text{if } a \leq b \text{ then } b \text{ else } a)$
by (*auto simp: diff-eq-0-iff-ennreal less-top*)

lemma *diff-add-self-ennreal*: $(b - a + a::\text{ennreal}) = (\text{if } a \leq b \text{ then } b \text{ else } a)$
by (*auto simp: diff-add-cancel-ennreal diff-eq-0-iff-ennreal less-top*)

lemma *ennreal-minus-cancel-iff*:
fixes $a \ b \ c :: \text{ennreal}$
shows $a - b = a - c \iff (b = c \vee (a \leq b \wedge a \leq c) \vee a = \text{top})$
by (*cases a; cases b; cases c*) *(auto simp: ennreal-minus-if)*

The next lemma is wrong for $a = \text{top}$, for $b = c = 1$ for instance.

lemma *ennreal-right-diff-distrib*:
fixes $a \ b \ c :: \text{ennreal}$

```

assumes  $a \neq \text{top}$ 
shows  $a * (b - c) = a * b - a * c$ 
apply (cases  $a$ ; cases  $b$ ; cases  $c$ )
      apply (use assms in  $\langle \text{auto simp add: ennreal-mult-top ennreal-minus}$ 
ennreal-mult' [symmetric]  $\rangle$ )
      apply (simp add: algebra-simps)
done

```

lemma *SUP-diff-ennreal*:

```

 $c < \text{top} \implies (\text{SUP } i \in I. f\ i - c :: \text{ennreal}) = (\text{SUP } i \in I. f\ i) - c$ 
by (auto intro!: SUP-eqI ennreal-minus-mono SUP-least intro: SUP-upper
      simp: ennreal-minus-cancel-iff ennreal-minus-le-iff less-top[symmetric])

```

lemma *ennreal-SUP-add-right*:

```

fixes  $c :: \text{ennreal}$  shows  $I \neq \{\}$   $\implies c + (\text{SUP } i \in I. f\ i) = (\text{SUP } i \in I. c + f\ i)$ 
using ennreal-SUP-add-left[of  $I\ f\ c$ ] by (simp add: add commute)

```

lemma *SUP-add-directed-ennreal*:

```

fixes  $f\ g :: - \Rightarrow \text{ennreal}$ 
assumes directed:  $\bigwedge i\ j. i \in I \implies j \in I \implies \exists k \in I. f\ i + g\ j \leq f\ k + g\ k$ 
shows  $(\text{SUP } i \in I. f\ i + g\ i) = (\text{SUP } i \in I. f\ i) + (\text{SUP } i \in I. g\ i)$ 
proof (cases  $I = \{\}$ )
  case False
    show ?thesis
    proof (rule antisym)
      show  $(\text{SUP } i \in I. f\ i + g\ i) \leq (\text{SUP } i \in I. f\ i) + (\text{SUP } i \in I. g\ i)$ 
        by (rule SUP-least; intro add-mono SUP-upper)
      next
        have  $(\text{SUP } i \in I. f\ i) + (\text{SUP } i \in I. g\ i) = (\text{SUP } i \in I. f\ i + (\text{SUP } i \in I. g\ i))$ 
          by (intro ennreal-SUP-add-left[symmetric]  $\langle I \neq \{\} \rangle$ )
        also have  $\dots = (\text{SUP } i \in I. (\text{SUP } j \in I. f\ i + g\ j))$ 
          using  $\langle I \neq \{\} \rangle$  by (simp add: ennreal-SUP-add-right)
        also have  $\dots \leq (\text{SUP } i \in I. f\ i + g\ i)$ 
          using directed by (intro SUP-least) (blast intro: SUP-upper2)
        finally show  $(\text{SUP } i \in I. f\ i) + (\text{SUP } i \in I. g\ i) \leq (\text{SUP } i \in I. f\ i + g\ i)$  .
    qed
qed (simp add: bot-ereal-def)

```

lemma *enn2real-eq-0-iff*: $\text{enn2real } x = 0 \longleftrightarrow x = 0 \vee x = \text{top}$

```

by (cases  $x$ ) auto

```

lemma *continuous-on-diff-ennreal*:

```

 $\text{continuous-on } A\ f \implies \text{continuous-on } A\ g \implies (\bigwedge x. x \in A \implies f\ x \neq \text{top}) \implies$ 
 $(\bigwedge x. x \in A \implies g\ x \neq \text{top}) \implies \text{continuous-on } A\ (\lambda z. f\ z - g\ z :: \text{ennreal})$ 
including ennreal.lifting
proof (transfer fixing:  $A$ , simp add: top-ereal-def)
  fix  $f\ g :: 'a \Rightarrow \text{ereal}$  assume  $\forall x. 0 \leq f\ x \ \forall x. 0 \leq g\ x$  continuous-on  $A\ f$ 
continuous-on  $A\ g$ 

```

moreover assume $f\ x \neq \infty\ g\ x \neq \infty$ if $x \in A$ for x
ultimately show continuous-on $A\ (\lambda z. \max\ 0\ (f\ z - g\ z))$
by (intro continuous-on-max continuous-on-const continuous-on-diff-ereal) auto
qed

lemma tendsto-diff-ennreal:

$(f \longrightarrow x)\ F \implies (g \longrightarrow y)\ F \implies x \neq \text{top} \implies y \neq \text{top} \implies ((\lambda z. f\ z - g\ z)::\text{ennreal}) \longrightarrow x - y)\ F$

using continuous-on-tendsto-compose[where $f=\lambda x. \text{fst}\ x - \text{snd}\ x::\text{ennreal}$ and
 $s=\{(x, y). x \neq \text{top} \wedge y \neq \text{top}\}$ and $g=\lambda x. (f\ x, g\ x)$ and $l=(x, y)$ and $F=F$,
OF continuous-on-diff-ennreal]

by (auto simp: tendsto-Pair eventually-conj-iff less-top order-tendstoD continuous-on-fst continuous-on-snd continuous-on-id)

declare lim-real-of-ereal [tendsto-intros]

lemma tendsto-enn2real [tendsto-intros]:

assumes $(u \longrightarrow \text{ennreal}\ l)\ F\ l \geq 0$

shows $((\lambda n. \text{enn2real}\ (u\ n)) \longrightarrow l)\ F$

unfolding enn2real-def

by (metis assms enn2ereal-ennreal lim-real-of-ereal tendsto-enn2erealI)

end

41 Logarithm of Natural Numbers

theory Log-Nat

imports Complex-Main

begin

41.1 Preliminaries

lemma divide-nat-diff-div-nat-less-one:

$\text{real}\ x / \text{real}\ b - \text{real}\ (x\ \text{div}\ b) < 1$ for $x\ b :: \text{nat}$

proof (cases $b = 0$)

case True

then show ?thesis

by simp

next

case False

then have $\text{real}\ (x\ \text{div}\ b) + \text{real}\ (x\ \text{mod}\ b) / \text{real}\ b - \text{real}\ (x\ \text{div}\ b) < 1$

by (simp add: field-simps)

then show ?thesis

by (metis of-nat-of-nat-div-aux)

qed

41.2 Floorlog

definition floorlog :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$

where $\text{floorlog } b \ a = (\text{if } a > 0 \wedge b > 1 \text{ then } \text{nat } \lfloor \log b \ a \rfloor + 1 \text{ else } 0)$

lemma *floorlog-mono*: $x \leq y \implies \text{floorlog } b \ x \leq \text{floorlog } b \ y$
by (*auto simp: floorlog-def floor-mono nat-mono*)

lemma *floorlog-bounds*:

$b \wedge (\text{floorlog } b \ x - 1) \leq x \wedge x < b \wedge (\text{floorlog } b \ x) \text{ if } x > 0 \ b > 1$

proof

show $b \wedge (\text{floorlog } b \ x - 1) \leq x$

proof –

have $b \wedge \text{nat } \lfloor \log b \ x \rfloor = b \ \text{powr } \lfloor \log b \ x \rfloor$

using *powr-realpow[symmetric, of b nat ⌊log b x⌋] <x> 0 1*

by *simp*

also have $\dots \leq b \ \text{powr } \log b \ x$ **using** * 1* **by** *simp*

also have $\dots = \text{real-of-int } x$ **using** *<0< x> 1* **by** *simp*

finally have $b \wedge \text{nat } \lfloor \log b \ x \rfloor \leq \text{real-of-int } x$ **by** *simp*

then show *?thesis*

using *<0< x> 1> of-nat-le-iff*

by (*fastforce simp add: floorlog-def*)

qed

show $x < b \wedge (\text{floorlog } b \ x)$

proof –

have $x \leq b \ \text{powr } (\log b \ x)$ **using** *<x> 0 1* **by** *simp*

also have $\dots < b \ \text{powr } (\lfloor \log b \ x \rfloor + 1)$

using *that* **by** (*intro powr-less-mono*) *auto*

also have $\dots = b \wedge \text{nat } (\lfloor \log b \ (\text{real-of-int } x) \rfloor + 1)$

using *that* **by** (*simp flip: powr-realpow*)

finally

have $x < b \wedge \text{nat } (\lfloor \log b \ (\text{int } x) \rfloor + 1)$

by (*rule of-nat-less-imp-less*)

then show *?thesis*

using *<x> 0 1* **by** (*simp add: floorlog-def nat-add-distrib*)

qed

qed

lemma *floorlog-power [simp]*:

$\text{floorlog } b \ (a * b \wedge c) = \text{floorlog } b \ a + c \text{ if } a > 0 \ b > 1$

proof –

have $\lfloor \log b \ a + \text{real } c \rfloor = \lfloor \log b \ a \rfloor + c$ **by** *arith*

then show *?thesis* **using** *that*

by (*auto simp: floorlog-def log-mult powr-realpow[symmetric] nat-add-distrib*)

qed

lemma *floor-log-add-eq1*:

$\lfloor \log b \ (a + r) \rfloor = \lfloor \log b \ a \rfloor \text{ if } b > 1 \ a \geq 1 \ 0 \leq r \ r < 1$

for $a \ b :: \text{nat}$ **and** $r :: \text{real}$

proof (*rule floor-eq2*)

have $\log b \ a \leq \log b \ (a + r)$ **using** *that* **by** *force*

then show $\lfloor \log b \ a \rfloor \leq \log b \ (a + r)$ **by** *arith*


```

next
  define l::int where l = int b ^ (nat ⌊log b a⌋ + 1)
  have l-def-real: l = b powr (⌊log b a⌋ + 1)
    using that by (simp add: l-def powr-add powr-real-of-int)
  have a < l
  proof -
    have a = b powr (log b a) using that by simp
    also have ... < b powr floor ((log b a) + 1)
      using that(1) by auto
    also have ... = l
      using that by (simp add: l-def powr-real-of-int powr-add)
    finally show ?thesis by simp
  qed
  then have a + r < l using that by simp
  then have log b (a + r) < log b l using that by simp
  also have ... = real-of-int ⌊log b a⌋ + 1
    using that by (simp add: l-def-real)
  finally show log b (a + r) < real-of-int ⌊log b a⌋ + 1 .
qed

lemma floor-log-div:
  ⌊log b x⌋ = ⌊log b (x div b)⌋ + 1 if b > 1 x > 0 x div b > 0
  for b x :: nat
proof -
  have ⌊log b x⌋ = ⌊log b (x / b * b)⌋ using that by simp
  also have ... = ⌊log b (x / b) + log b b⌋
    using that by (subst log-mult) auto
  also have ... = ⌊log b (x / b)⌋ + 1 using that by simp
  also have ⌊log b (x / b)⌋ = ⌊log b (x div b + (x / b - x div b))⌋ by simp
  also have ... = ⌊log b (x div b)⌋
    using that of-nat-div-le-of-nat divide-nat-diff-div-nat-less-one
    by (intro floor-log-add-eqI) auto
  finally show ?thesis .
qed

lemma compute-floorlog [code]:
  floorlog b x = (if x > 0 ∧ b > 1 then floorlog b (x div b) + 1 else 0)
  by (auto simp: floorlog-def floor-log-div[of b x] div-eq-0-iff nat-add-distrib
    intro!: floor-eq2)

lemma floor-log-eq-if:
  ⌊log b x⌋ = ⌊log b y⌋ if x div b = y div b b > 1 x > 0 x div b ≥ 1
  for b x y :: nat
proof -
  have y > 0 using that by (auto intro: ccontr)
  thus ?thesis using that by (simp add: floor-log-div)
qed

lemma floorlog-eq-if:

```

$\text{floorlog } b \ x = \text{floorlog } b \ y$ **if** $x \text{ div } b = y \text{ div } b \ b > 1 \ x > 0 \ x \text{ div } b \geq 1$
for $b \ x \ y :: \text{nat}$
proof –
have $y > 0$ **using** *that* **by** (*auto intro: ccontr*)
then show *?thesis* **using** *that*
by (*auto simp add: floorlog-def eq-nat-nat-iff intro: floor-log-eq-if*)
qed

lemma *floorlog-leD*:
 $\text{floorlog } b \ x \leq w \implies b > 1 \implies x < b \wedge w$
by (*metis floorlog-bounds leD linorder-negE-nat order.strict-trans power-strict-increasing-iff zero-less-one zero-less-power*)

lemma *floorlog-leI*:
 $x < b \wedge w \implies 0 \leq w \implies b > 1 \implies \text{floorlog } b \ x \leq w$
by (*drule less-imp-of-nat-less[where 'a=real]*)
(auto simp: floorlog-def Suc-le-eq nat-less-iff floor-less-iff log-of-power-less)

lemma *floorlog-eq-zero-iff*:
 $\text{floorlog } b \ x = 0 \iff b \leq 1 \vee x \leq 0$
by (*auto simp: floorlog-def*)

lemma *floorlog-le-iff*:
 $\text{floorlog } b \ x \leq w \iff b \leq 1 \vee b > 1 \wedge 0 \leq w \wedge x < b \wedge w$
using *floorlog-leD*[*of* $b \ x \ w$] *floorlog-leI*[*of* $x \ b \ w$]
by (*auto simp: floorlog-eq-zero-iff[THEN iffD2]*)

lemma *floorlog-ge-SucI*:
 $\text{Suc } w \leq \text{floorlog } b \ x$ **if** $b \wedge w \leq x \ b > 1$
using *that* *le-log-of-power*[*of* $b \ w \ x$] *power-not-zero*
by (*force simp: floorlog-def Suc-le-eq powr-realpow not-less Suc-nat-eq-nat-zadd1 zless-nat-eq-int-zless int-add-floor less-floor-iff simp del: floor-add2*)

lemma *floorlog-geI*:
 $w \leq \text{floorlog } b \ x$ **if** $b \wedge (w - 1) \leq x \ b > 1$
using *floorlog-ge-SucI*[*of* $b \ w - 1 \ x$] *that*
by *auto*

lemma *floorlog-geD*:
 $b \wedge (w - 1) \leq x$ **if** $w \leq \text{floorlog } b \ x \ w > 0$
proof –
have $b > 1 \ 0 < x$
using *that* **by** (*auto simp: floorlog-def split: if-splits*)
have $b \wedge (w - 1) \leq x$ **if** $b \wedge w \leq x$
proof –
have $b \wedge (w - 1) \leq b \wedge w$
using $\langle b > 1 \rangle$
by (*auto intro!: power-increasing*)

also note that
 finally show ?thesis .
 qed
 moreover have $b \wedge \text{nat } \lfloor \log (\text{real } b) (\text{real } x) \rfloor \leq x$ (is ?l ≤ -)
 proof -
 have $0 \leq \log (\text{real } b) (\text{real } x)$
 using $\langle b > 1 \rangle \langle 0 < x \rangle$
 by auto
 then have $?l \leq b \text{ powr } \log (\text{real } b) (\text{real } x)$
 using $\langle b > 1 \rangle$
 by (auto simp flip: powr-realpow intro!: powr-mono of-nat-floor)
 also have $\dots = x$ using $\langle b > 1 \rangle \langle 0 < x \rangle$
 by auto
 finally show ?thesis
 unfolding of-nat-le-iff .
 qed
 ultimately show ?thesis
 using that
 by (auto simp: floorlog-def le-nat-iff le-floor-iff le-log-iff powr-realpow
 split: if-splits elim!: le-SucE)
 qed

41.3

definition ceillog2 :: $\text{nat} \Rightarrow \text{nat}$ where
 $\text{ceillog2 } n = (\text{if } n = 0 \text{ then } 0 \text{ else } \text{nat } \lceil \log 2 (\text{real } n) \rceil)$

lemma ceillog2-0 [simp]: $\text{ceillog2 } 0 = 0$
and ceillog2-Suc-0 [simp]: $\text{ceillog2 } (\text{Suc } 0) = 0$
and ceillog2-2 [simp]: $\text{ceillog2 } 2 = 1$
 by (auto simp: ceillog2-def)

lemma ceillog2-le1-eq-0 [simp]: $n \leq 1 \implies \text{ceillog2 } n = 0$
 by (cases n) auto

lemma ceillog2-2-power [simp]: $\text{ceillog2 } (2 \wedge n) = n$
 by (auto simp: ceillog2-def)

lemma ceillog2-ge-log:
 assumes $n > 0$
 shows $\text{real } (\text{ceillog2 } n) \geq \log 2 (\text{real } n)$
 proof -
 have $\text{real-of-int } \lceil \log 2 (\text{real } n) \rceil \geq \log 2 (\text{real } n)$
 by linarith
 thus ?thesis
 using assms unfolding ceillog2-def by auto
 qed

lemma ceillog2-less-log:

```

  assumes  $n > 0$ 
  shows  $\text{real } (\text{ceillog2 } n) < \log 2 (\text{real } n) + 1$ 
proof -
  have  $\text{real-of-int } \lceil \log 2 (\text{real } n) \rceil < \log 2 (\text{real } n) + 1$ 
    by linarith
  thus ?thesis
    using assms unfolding ceillog2-def by auto
qed

```

```

lemma ceillog2-le-iff:
  assumes  $n > 0$ 
  shows  $\text{ceillog2 } n \leq l \iff n \leq 2^l$ 
proof -
  have  $\text{ceillog2 } n \leq l \iff \text{real } n \leq 2^l$ 
    unfolding ceillog2-def using assms by (auto simp: log-le-iff powr-realpow)
  also have  $2^l = \text{real } (2^l)$ 
    by simp
  also have  $\text{real } n \leq \text{real } (2^l) \iff n \leq 2^l$ 
    by linarith
  finally show ?thesis .
qed

```

```

lemma ceillog2-ge-iff:
  assumes  $n > 0$ 
  shows  $\text{ceillog2 } n \geq l \iff 2^l < 2 * n$ 
proof -
  have  $-1 < (0 :: \text{real})$ 
    by auto
  also have  $\dots \leq \log 2 (\text{real } n)$ 
    using assms by auto
  finally have  $\text{ceillog2 } n \geq l \iff \text{real } l - 1 < \log 2 (\text{real } n)$ 
    unfolding ceillog2-def using assms by (auto simp: le-nat-iff le-ceiling-iff)
  also have  $\dots \iff \text{real } l < \log 2 (\text{real } (2 * n))$ 
    using assms by (auto simp: log-mult)
  also have  $\dots \iff 2^l < \text{real } (2 * n)$ 
    using assms by (subst less-log-iff) (auto simp: powr-realpow)
  also have  $2^l = \text{real } (2^l)$ 
    by simp
  also have  $\text{real } (2^l) < \text{real } (2 * n) \iff 2^l < 2 * n$ 
    by linarith
  finally show ?thesis .
qed

```

```

lemma le-two-power-ceillog2:  $n \leq 2^{\text{ceillog2 } n}$ 
  using neq0-conv ceillog2-le-iff by blast

```

```

lemma two-power-ceillog2-gt:
  assumes  $n > 0$ 
  shows  $2 * n > 2^{\text{ceillog2 } n}$ 

```

using *ceillog2-ge-iff*[of n *ceillog2* n] *assms* by *simp*

lemma *ceillog2-eqI*:

assumes $n \leq 2^l$ $2^l < 2 * n$

shows *ceillog2* $n = l$

by (metis *Suc-leI* *assms* *bot-nat-0.not-eq-extremum* *ceillog2-ge-iff* *ceillog2-le-iff*
le-antisym *mult-is-0*
not-less-eq-eq)

lemma *ceillog2-rec-even*:

assumes $k > 0$

shows *ceillog2* $(2 * k) = \text{Suc}(\text{ceillog2 } k)$

by (rule *ceillog2-eqI*) (auto simp: *le-two-power-ceillog2* *two-power-ceillog2-gt* *assms*)

lemma *ceillog2-mono*:

assumes $m \leq n$

shows *ceillog2* $m \leq \text{ceillog2 } n$

proof (cases $m = 0$)

case *False*

have $\lceil \log 2 (\text{real } m) \rceil \leq \lceil \log 2 (\text{real } n) \rceil$

by (intro *ceiling-mono*) (use *False* *assms* in auto)

hence $\text{nat } \lceil \log 2 (\text{real } m) \rceil \leq \text{nat } \lceil \log 2 (\text{real } n) \rceil$

by *linarith*

thus ?thesis using *False* *assms*

unfolding *ceillog2-def* by *simp*

qed auto

lemma *ceillog2-rec-odd*:

assumes $k > 0$

shows *ceillog2* $(\text{Suc}(2 * k)) = \text{Suc}(\text{ceillog2}(\text{Suc } k))$

proof –

have $2^{\text{ceillog2}(\text{Suc}(2 * k))} > \text{Suc}(2 * k)$

by (metis *assms* *diff-Suc-1* *dvd-triv-left* *le-two-power-ceillog2* *mult-pos-pos* *nat-power-eq-Suc-0-iff*

order-less-le *pos2* *semiring-parity-class.even-mask-iff*)

then have *ceillog2* $(2 * k + 2) \leq \text{ceillog2}(2 * k + 1)$

by (simp add: *ceillog2-le-iff*)

moreover have *ceillog2* $(2 * k + 2) \geq \text{ceillog2}(2 * k + 1)$

by (rule *ceillog2-mono*) auto

ultimately have *ceillog2* $(2 * k + 2) = \text{ceillog2}(2 * k + 1)$

by (rule *antisym*)

also have $2 * k + 2 = 2 * \text{Suc } k$

by *simp*

also have *ceillog2* $(2 * \text{Suc } k) = \text{Suc}(\text{ceillog2}(\text{Suc } k))$

by (rule *ceillog2-rec-even*) auto

finally show ?thesis

by *simp*

qed

lemma *ceillog2-rec*:

ceillog2 *n* = (if *n* ≤ 1 then 0 else 1 + *ceillog2* ((*n* + 1) div 2))

proof (*cases n* ≤ 1)

case *True*

thus ?*thesis*

by (*cases n*) *auto*

next

case *False*

thus ?*thesis*

by (*cases even n*) (*auto elim!*: *evenE oddE simp: ceillog2-rec-even ceillog2-rec-odd*)

qed

lemma *funpow-div2-ceillog2-le-1*:

((λ*n*. (*n* + 1) div 2) ^{^^} *ceillog2 n*) *n* ≤ 1

proof (*induction n rule: less-induct*)

case (*less n*)

show ?*case*

proof (*cases n* ≤ 1)

case *True*

thus ?*thesis* **by** (*cases n*) *auto*

next

case *False*

have ((λ*n*. (*n* + 1) div 2) ^{^^} *Suc (ceillog2 ((*n* + 1) div 2)))* *n* ≤ 1

using *less.IH*[*of (n+1) div 2*] *False* **by** (*subst funpow-Suc-right*) *auto*

also have *Suc (ceillog2 ((*n* + 1) div 2)) = ceillog2 n*

using *False* **by** (*subst ceillog2-rec[of n]*) *auto*

finally show ?*thesis* .

qed

qed

fun *ceillog2-aux* :: *nat* ⇒ *nat* ⇒ *nat* **where**

ceillog2-aux *acc n* = (if *n* ≤ 1 then *acc* else *ceillog2-aux* (*acc* + 1) ((*n* + 1) div 2))

lemmas [*simp del*] = *ceillog2-aux.simps*

lemma *ceillog2-aux-correct*: *ceillog2-aux* *acc n* = *ceillog2 n* + *acc*

proof (*induction acc n rule: ceillog2-aux.induct*)

case (*1 acc n*)

show ?*case*

proof (*cases n* ≤ 1)

case *False*

thus ?*thesis* **using** *ceillog2-rec[of n]* 1.*IH*

by (*auto simp: ceillog2-aux.simps[of acc n]*)

qed (*auto simp: ceillog2-aux.simps[of acc n]*)

qed

lemma *ceillog2-code* [code]: $\text{ceillog2 } n = \text{ceillog2-aux } 0 \ n$
by (*simp add: ceillog2-aux-correct*)

41.4 Bitlen

definition *bitlen* :: $\text{int} \Rightarrow \text{int}$
where *bitlen* *a* = *floorlog 2 (nat a)*

lemma *bitlen-alt-def*:
 $\text{bitlen } a = (\text{if } a > 0 \text{ then } \lfloor \log 2 a \rfloor + 1 \text{ else } 0)$
by (*simp add: bitlen-def floorlog-def*)

lemma *bitlen-zero* [simp]:
 $\text{bitlen } 0 = 0$
by (*auto simp: bitlen-def floorlog-def*)

lemma *bitlen-nonneg*:
 $0 \leq \text{bitlen } x$
by (*simp add: bitlen-def*)

lemma *bitlen-bounds*:
 $2^{\text{nat } (\text{bitlen } x - 1)} \leq x \wedge x < 2^{\text{nat } (\text{bitlen } x)} \text{ if } x > 0$
proof –
from *that have bitlen* $x \geq 1$ **by** (*auto simp: bitlen-alt-def*)
with *that floorlog-bounds* [of $\text{nat } x \ 2$] **show** *?thesis*
by (*auto simp add: bitlen-def le-nat-iff nat-less-iff nat-diff-distrib*)
qed

lemma *bitlen-pow2* [simp]:
 $\text{bitlen } (b * 2^c) = \text{bitlen } b + c \text{ if } b > 0$
using *that* **by** (*simp add: bitlen-def nat-mult-distrib nat-power-eq*)

lemma *compute-bitlen* [code]:
 $\text{bitlen } x = (\text{if } x > 0 \text{ then } \text{bitlen } (x \text{ div } 2) + 1 \text{ else } 0)$
by (*simp add: bitlen-def nat-div-distrib compute-floorlog*)

lemma *bitlen-eq-zero-iff*:
 $\text{bitlen } x = 0 \iff x \leq 0$
by (*auto simp add: bitlen-alt-def*)
(metis compute-bitlen add commute bitlen-alt-def bitlen-nonneg less-add-same-cancel2 not-less zero-less-one)

lemma *bitlen-div*:
 $1 \leq \text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)}$
and $\text{real-of-int } m / 2^{\text{nat } (\text{bitlen } m - 1)} < 2 \text{ if } 0 < m$
proof –
let $?B = 2^{\text{nat } (\text{bitlen } m - 1)}$

```

have ?B ≤ m using bitlen-bounds[OF ‹0 < m›] ..
then have 1 * ?B ≤ real-of-int m
  unfolding of-int-le-iff[symmetric] by auto
then show 1 ≤ real-of-int m / ?B by auto

from that have 0 ≤ bitlen m - 1 by (auto simp: bitlen-alt-def)

have m < 2nat(bitlen m) using bitlen-bounds[OF that] ..
also from that have ... = 2nat(bitlen m - 1 + 1)
  by (auto simp: bitlen-def)
also have ... = ?B * 2
  unfolding nat-add-distrib[OF ‹0 ≤ bitlen m - 1› zero-le-one] by auto
finally have real-of-int m < 2 * ?B
  by (metis (full-types) mult.commute power.simps(2) of-int-less-numeral-power-cancel-iff)
then have real-of-int m / ?B < 2 * ?B / ?B
  by (rule divide-strict-right-mono) auto
then show real-of-int m / ?B < 2 by auto
qed

lemma bitlen-le-iff-floorlog:
  bitlen x ≤ w ⟷ w ≥ 0 ∧ floorlog 2 (nat x) ≤ nat w
  by (auto simp: bitlen-def)

lemma bitlen-le-iff-power:
  bitlen x ≤ w ⟷ w ≥ 0 ∧ x < 2nat w
  by (auto simp: bitlen-le-iff-floorlog floorlog-le-iff)

lemma less-power-nat-iff-bitlen:
  x < 2w ⟷ bitlen (int x) ≤ w
  using bitlen-le-iff-power[of x w]
  by auto

lemma bitlen-ge-iff-power:
  w ≤ bitlen x ⟷ w ≤ 0 ∨ 2(nat w - 1) ≤ x
  unfolding bitlen-def
  by (auto simp flip: nat-le-iff intro: floorlog-geI dest: floorlog-geD)

lemma bitlen-twopow-add-eq:
  bitlen (2w + b) = w + 1 if 0 ≤ b < 2w
  by (auto simp: that nat-add-distrib bitlen-le-iff-power bitlen-ge-iff-power intro!:
    antisym)

end

```

42 Various algebraic structures combined with a lattice

theory *Lattice-Algebras*


```

imports Complex-Main
begin

class semilattice-inf-ab-group-add = ordered-ab-group-add + semilattice-inf
begin

lemma add-inf-distrib-left:  $a + \inf b\ c = \inf (a + b)\ (a + c)$  (is  $?L=?R$ )
proof (intro order.antisym)
  show  $?R \leq ?L$ 
  by (metis add-commute diff-le-eq inf-greatest inf-le1 inf-le2)
qed simp

lemma add-inf-distrib-right:  $\inf a\ b + c = \inf (a + c)\ (b + c)$ 
  using add-commute add-inf-distrib-left by presburger

end

class semilattice-sup-ab-group-add = ordered-ab-group-add + semilattice-sup
begin

lemma add-sup-distrib-left:  $a + \sup b\ c = \sup (a + b)\ (a + c)$  (is  $?L = ?R$ )
proof (rule order.antisym)
  show  $?L \leq ?R$ 
  by (metis add-commute le-diff-eq sup.bounded-iff sup-ge1 sup-ge2)
qed simp

lemma add-sup-distrib-right:  $\sup a\ b + c = \sup (a + c)\ (b + c)$ 
proof –
  have  $c + \sup a\ b = \sup (c+a)\ (c+b)$ 
  by (simp add: add-sup-distrib-left)
  then show  $?thesis$ 
  by (simp add: add.commute)
qed

end

class lattice-ab-group-add = ordered-ab-group-add + lattice
begin

subclass semilattice-inf-ab-group-add ..
subclass semilattice-sup-ab-group-add ..

lemmas add-sup-inf-distribs =
  add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

lemma inf-eq-neg-sup:  $\inf a\ b = - \sup (- a)\ (- b)$ 
proof (rule inf-unique)
  fix  $a\ b\ c :: 'a$ 
  show  $- \sup (- a)\ (- b) \leq a$ 

```

```

    by (rule add-le-imp-le-right [of - sup (uminus a) (uminus b)])
      (simp, simp add: add-sup-distrib-left)
  show  $- \sup (-a) (-b) \leq b$ 
    by (rule add-le-imp-le-right [of - sup (uminus a) (uminus b)])
      (simp, simp add: add-sup-distrib-left)
  assume  $a \leq b$   $a \leq c$ 
  then show  $a \leq - \sup (-b) (-c)$ 
    by (subst neg-le-iff-le [symmetric]) (simp add: le-supI)
qed

```

```

lemma sup-eq-neg-inf:  $\sup a b = - \inf (- a) (- b)$ 
proof (rule sup-unique)
  fix a b c :: 'a
  show  $a \leq - \inf (- a) (- b)$ 
    by (rule add-le-imp-le-right [of - inf (uminus a) (uminus b)])
      (simp, simp add: add-inf-distrib-left)
  show  $b \leq - \inf (- a) (- b)$ 
    by (rule add-le-imp-le-right [of - inf (uminus a) (uminus b)])
      (simp, simp add: add-inf-distrib-left)
  show  $- \inf (- a) (- b) \leq c$  if  $a \leq c$   $b \leq c$ 
    using that by (subst neg-le-iff-le [symmetric]) (simp add: le-infI)
qed

```

```

lemma neg-inf-eq-sup:  $- \inf a b = \sup (- a) (- b)$ 
  by (simp add: inf-eq-neg-sup)

```

```

lemma diff-inf-eq-sup:  $a - \inf b c = a + \sup (- b) (- c)$ 
  using neg-inf-eq-sup [of b c, symmetric] by simp

```

```

lemma neg-sup-eq-inf:  $- \sup a b = \inf (- a) (- b)$ 
  by (simp add: sup-eq-neg-inf)

```

```

lemma diff-sup-eq-inf:  $a - \sup b c = a + \inf (- b) (- c)$ 
  using neg-sup-eq-inf [of b c, symmetric] by simp

```

```

lemma add-eq-inf-sup:  $a + b = \sup a b + \inf a b$ 
proof -
  have  $0 = - \inf 0 (a - b) + \inf (a - b) 0$ 
    by (simp add: inf-commute)
  then have  $0 = \sup 0 (b - a) + \inf (a - b) 0$ 
    by (simp add: inf-eq-neg-sup)
  then have  $0 = (- a + \sup a b) + (\inf a b + (- b))$ 
    by (simp only: add-sup-distrib-left add-inf-distrib-right) simp
  then show ?thesis
    by (simp add: algebra-simps)
qed

```

42.1 Positive Part, Negative Part, Absolute Value

definition $\text{nprt} :: 'a \Rightarrow 'a$
where $\text{nprt } x = \inf x 0$

definition $\text{pprt} :: 'a \Rightarrow 'a$
where $\text{pprt } x = \sup x 0$

lemma pprt-neg : $\text{pprt } (- x) = - \text{nprt } x$

proof –

have $\sup (- x) 0 = \sup (- x) (- 0)$

by (*simp only: minus-zero*)

also have $\dots = - \inf x 0$

by (*simp only: neg-inf-eq-sup*)

finally have $\sup (- x) 0 = - \inf x 0$.

then show *?thesis*

by (*simp only: pprt-def nprt-def*)

qed

lemma nprt-neg : $\text{nprt } (- x) = - \text{pprt } x$

proof –

from pprt-neg **have** $\text{pprt } (- (- x)) = - \text{nprt } (- x)$.

then have $\text{pprt } x = - \text{nprt } (- x)$ **by** *simp*

then show *?thesis* **by** *simp*

qed

lemma prts : $a = \text{pprt } a + \text{nprt } a$

by (*simp add: pprt-def nprt-def flip: add-eq-inf-sup*)

lemma zero-le-pprt [*simp*]: $0 \leq \text{pprt } a$

by (*simp add: pprt-def*)

lemma nprt-le-zero [*simp*]: $\text{nprt } a \leq 0$

by (*simp add: nprt-def*)

lemma le-eq-neg : $a \leq - b \longleftrightarrow a + b \leq 0$

(**is** *?lhs = ?rhs*)

proof

assume *?lhs*

show *?rhs*

by (*rule add-le-imp-le-right*[*of - uminus b -*]) (*simp add: add.assoc* $\langle ?lhs \rangle$)

next

assume *?rhs*

show *?lhs*

by (*rule add-le-imp-le-right*[*of - b -*]) (*simp add:* $\langle ?rhs \rangle$)

qed

lemma pprt-0 [*simp*]: $\text{pprt } 0 = 0$ **by** (*simp add: pprt-def*)

lemma nprt-0 [*simp*]: $\text{nprt } 0 = 0$ **by** (*simp add: nprt-def*)

lemma *pprt-eq-id* [*simp*, *no-atp*]: $0 \leq x \implies \text{pprt } x = x$
by (*simp add: pprt-def sup-absorb1*)

lemma *nprrt-eq-id* [*simp*, *no-atp*]: $x \leq 0 \implies \text{nprrt } x = x$
by (*simp add: nprrt-def inf-absorb1*)

lemma *pprt-eq-0* [*simp*, *no-atp*]: $x \leq 0 \implies \text{pprt } x = 0$
by (*simp add: pprt-def sup-absorb2*)

lemma *nprrt-eq-0* [*simp*, *no-atp*]: $0 \leq x \implies \text{nprrt } x = 0$
by (*simp add: nprrt-def inf-absorb2*)

lemma *sup-0-imp-0*:
assumes *sup a (- a) = 0*
shows *a = 0*
proof –
have *pos: 0 ≤ a if sup a (- a) = 0 for a :: 'a*
proof –
from *that* **have** *sup a (- a) + a = a*
by *simp*
then **have** *sup (a + a) 0 = a*
by (*simp add: add-sup-distrib-right*)
then **have** *sup (a + a) 0 ≤ a*
by *simp*
then **show** *?thesis*
by (*blast intro: order-trans inf-sup-ord*)
qed
from *assms* **have** *** : sup (-a) (-(-a)) = 0*
by (*simp add: sup-commute*)
from *pos[OF assms] pos[OF **]* **show** *a = 0*
by *simp*
qed

lemma *inf-0-imp-0*: $\text{inf } a (- a) = 0 \implies a = 0$
by (*metis local.neg-0-equal-iff-equal neg-inf-eq-sup sup-0-imp-0*)

lemma *inf-0-eq-0* [*simp*]: $\text{inf } a (- a) = 0 \longleftrightarrow a = 0$
by (*metis inf-0-imp-0 inf.idem minus-zero*)

lemma *sup-0-eq-0* [*simp*]: $\text{sup } a (- a) = 0 \longleftrightarrow a = 0$
by (*metis minus-zero sup.idem sup-0-imp-0*)

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: $0 \leq a + a \longleftrightarrow 0 \leq a$
(is ?lhs \longleftrightarrow ?rhs)

proof
show *?rhs if ?lhs*
proof –
from *that* **have** *a: inf (a + a) 0 = 0*
by (*simp add: inf-commute inf-absorb1*)

```

have inf a 0 + inf a 0 = inf (inf (a + a) 0) a (is ?l = -)
  by (simp add: add-sup-inf-distrib inf-aci)
then have ?l = 0 + inf a 0
  by (simp add: a, simp add: inf-commute)
then have inf a 0 = 0
  by (simp only: add-right-cancel)
then show ?thesis
  unfolding le-iff-inf by (simp add: inf-commute)
qed
show ?lhs if ?rhs
  by (simp add: add-mono[OF that that, simplified])
qed

```

```

lemma double-zero [simp]: a + a = 0  $\longleftrightarrow$  a = 0
  using add-nonneg-eq-0-iff order.eq-iff by auto

```

```

lemma zero-less-double-add-iff-zero-less-single-add [simp]: 0 < a + a  $\longleftrightarrow$  0 < a
  by (meson le-less-trans less-add-same-cancel2 less-le-not-le
    zero-le-double-add-iff-zero-le-single-add)

```

```

lemma double-add-le-zero-iff-single-add-le-zero [simp]: a + a  $\leq$  0  $\longleftrightarrow$  a  $\leq$  0
proof -
  have a + a  $\leq$  0  $\longleftrightarrow$  0  $\leq$  - (a + a)
    by (subst le-minus-iff) simp
  moreover have ...  $\longleftrightarrow$  a  $\leq$  0
    by (simp only: minus-add-distrib zero-le-double-add-iff-zero-le-single-add) simp
  ultimately show ?thesis
    by blast
qed

```

```

lemma double-add-less-zero-iff-single-less-zero [simp]: a + a < 0  $\longleftrightarrow$  a < 0
proof -
  have a + a < 0  $\longleftrightarrow$  0 < - (a + a)
    by (subst less-minus-iff) simp
  moreover have ...  $\longleftrightarrow$  a < 0
    by (simp only: minus-add-distrib zero-less-double-add-iff-zero-less-single-add)
  simp
  ultimately show ?thesis
    by blast
qed

```

```

declare neg-inf-eq-sup [simp]
and neg-sup-eq-inf [simp]
and diff-inf-eq-sup [simp]
and diff-sup-eq-inf [simp]

```

```

lemma le-minus-self-iff: a  $\leq$  - a  $\longleftrightarrow$  a  $\leq$  0
proof -
  from add-le-cancel-left [of uminus a plus a a zero]

```

```

have  $a \leq -a \longleftrightarrow a + a \leq 0$ 
  by (simp flip: add.assoc)
then show ?thesis
  by simp
qed

```

```

lemma minus-le-self-iff:  $-a \leq a \longleftrightarrow 0 \leq a$ 
proof -
  have  $-a \leq a \longleftrightarrow 0 \leq a + a$ 
    using add-le-cancel-left [of uminus a zero plus a]
    by (simp flip: add.assoc)
  then show ?thesis
    by simp
qed

```

```

lemma zero-le-iff-zero-nprt:  $0 \leq a \longleftrightarrow \text{nprt } a = 0$ 
  unfolding le-iff-inf by (simp add: nprt-def inf-commute)

```

```

lemma le-zero-iff-zero-pprt:  $a \leq 0 \longleftrightarrow \text{pprt } a = 0$ 
  unfolding le-iff-sup by (simp add: pprt-def sup-commute)

```

```

lemma le-zero-iff-pprt-id:  $0 \leq a \longleftrightarrow \text{pprt } a = a$ 
  unfolding le-iff-sup by (simp add: pprt-def sup-commute)

```

```

lemma zero-le-iff-nprt-id:  $a \leq 0 \longleftrightarrow \text{nprt } a = a$ 
  unfolding le-iff-inf by (simp add: nprt-def inf-commute)

```

```

lemma pprt-mono [simp, no-atp]:  $a \leq b \implies \text{pprt } a \leq \text{pprt } b$ 
  unfolding le-iff-sup by (simp add: pprt-def sup-aci sup-assoc [symmetric, of a])

```

```

lemma nprt-mono [simp, no-atp]:  $a \leq b \implies \text{nprt } a \leq \text{nprt } b$ 
  unfolding le-iff-inf by (simp add: nprt-def inf-aci inf-assoc [symmetric, of a])

```

```

end

```

```

lemmas add-sup-inf-distrib =
  add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

```

```

class lattice-ab-group-add-abs = lattice-ab-group-add + abs +
  assumes abs-lattice:  $|a| = \text{sup } a \text{ } (-a)$ 
begin

```

```

lemma abs-prts:  $|a| = \text{pprt } a - \text{nprt } a$ 
proof -
  have  $0 \leq |a|$ 
  proof -
    have  $a: a \leq |a|$  and  $b: -a \leq |a|$ 
      by (auto simp add: abs-lattice)

```

```

    show ?thesis
      by (rule add-mono [OF a b, simplified])
  qed
  then have  $0 \leq \sup a (- a)$ 
    unfolding abs-lattice .
  then have  $\sup (\sup a (- a)) 0 = \sup a (- a)$ 
    by (rule sup-absorb1)
  then show ?thesis
    by (simp add: add-sup-inf-distrib ac-simps ppri-def npri-def abs-lattice)
qed

subclass ordered-ab-group-add-abs
proof
  have abs-ge-zero [simp]:  $0 \leq |a|$  for  $a$ 
  proof -
    have  $a: a \leq |a|$  and  $b: - a \leq |a|$ 
      by (auto simp add: abs-lattice)
    show  $0 \leq |a|$ 
      by (rule add-mono [OF a b, simplified])
  qed
  have abs-leI:  $a \leq b \implies - a \leq b \implies |a| \leq b$  for  $a b$ 
    by (simp add: abs-lattice le-supI)
  fix  $a b$ 
  show  $0 \leq |a|$ 
    by simp
  show  $a \leq |a|$ 
    by (auto simp add: abs-lattice)
  show  $|-a| = |a|$ 
    by (simp add: abs-lattice sup-commute)
  show  $- a \leq b \implies |a| \leq b$  if  $a \leq b$ 
    using that by (rule abs-leI)
  show  $|a + b| \leq |a| + |b|$ 
  proof -
    have  $g: |a| + |b| = \sup (a + b) (\sup (- a - b) (\sup (- a + b) (a + (- b))))$ 
      (is  $- = \sup ?m ?n$ )
      by (simp add: abs-lattice add-sup-inf-distrib ac-simps)
    have  $a: a + b \leq \sup ?m ?n$ 
      by simp
    have  $b: - a - b \leq ?n$ 
      by simp
    have  $c: ?n \leq \sup ?m ?n$ 
      by simp
    from  $b c$  have  $d: - a - b \leq \sup ?m ?n$ 
      by (rule order-trans)
    have  $e: - a - b = - (a + b)$ 
      by simp
    from  $a d e$  have  $|a + b| \leq \sup ?m ?n$ 
      by (metis abs-leI)
    with  $g[symmetric]$  show ?thesis by simp
  qed

```

qed
qed

end

lemma *sup-eq-if*:

fixes $a :: 'a::\{\text{lattice-ab-group-add,linorder}\}$
 shows $\text{sup } a \ (-a) = (\text{if } a < 0 \text{ then } -a \text{ else } a)$
 using *add-le-cancel-right* [of $a \ a \ -a$, *symmetric, simplified*]
 and *add-le-cancel-right* [of $-a \ a \ a$, *symmetric, simplified*]
 by (*auto simp: sup-max max.absorb1 max.absorb2*)

lemma *abs-if-lattice*:

fixes $a :: 'a::\{\text{lattice-ab-group-add-abs,linorder}\}$
 shows $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$
 by *auto*

lemma *estimate-by-abs*:

fixes $a \ b \ c :: 'a::\text{lattice-ab-group-add-abs}$
 assumes $a + b \leq c$
 shows $a \leq c + |b|$

proof –

from *assms* have $a \leq c + (-b)$
 by (*simp add: algebra-simps*)
 have $-b \leq |b|$
 by (*rule abs-ge-minus-self*)
 then have $c + (-b) \leq c + |b|$
 by (*rule add-left-mono*)
 with $\langle a \leq c + (-b) \rangle$ show ?thesis
 by (*rule order-trans*)

qed

class *lattice-ring* = *ordered-ring* + *lattice-ab-group-add-abs*
 begin

subclass *semilattice-inf-ab-group-add* ..

subclass *semilattice-sup-ab-group-add* ..

end

lemma *abs-le-mult*:

fixes $a \ b :: 'a::\text{lattice-ring}$
 shows $|a * b| \leq |a| * |b|$

proof –

let $?x = \text{pprt } a * \text{pprt } b - \text{pprt } a * \text{nprt } b - \text{nprt } a * \text{pprt } b + \text{nprt } a * \text{nprt } b$
 let $?y = \text{pprt } a * \text{pprt } b + \text{pprt } a * \text{nprt } b + \text{nprt } a * \text{pprt } b + \text{nprt } a * \text{nprt } b$
 have $a: |a| * |b| = ?x$
 by (*simp only: abs-prts[of a] abs-prts[of b] algebra-simps*)
 have $bh: u = a \implies v = b \implies$


```

      u * v = pprt a * pprt b + pprt a * nprt b +
              nprt a * pprt b + nprt a * nprt b for u v :: 'a
    by (metis add.commute combine-common-factor distrib-left prts)
  note b = this[OF refl[of a] refl[of b]]
  have xy: - ?x ≤ ?y
    apply simp
    by (meson add-increasing2 diff-le-eq neg-le-0-iff-le nprt-le-zero order.trans split-mult-pos-le
      zero-le-pprt)
  have yx: ?y ≤ ?x
    apply simp
    by (metis add-decreasing2 diff-0 diff-mono diff-zero mult-nonpos-nonneg mult-right-mono-neg
      mult-zero-left nprt-le-zero zero-le-pprt)
  show ?thesis
  proof (rule abs-leI)
    show a * b ≤ |a| * |b|
      by (simp only: a b yx)
    show - (a * b) ≤ |a| * |b|
      by (metis a bh minus-le-iff xy)
  qed
qed

```

instance *lattice-ring* \subseteq *ordered-ring-abs*

```

proof
  fix a b :: 'a::lattice-ring
  assume a: (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  show |a * b| = |a| * |b|
  proof -
    have s: (0 ≤ a * b) ∨ (a * b ≤ 0)
      by (metis a split-mult-neg-le split-mult-pos-le)
    have mulprts: a * b = (pprt a + nprt a) * (pprt b + nprt b)
      by (simp flip: prts)
    show ?thesis
    proof (cases 0 ≤ a * b)
      case True
      then show ?thesis
        using a split-mult-neg-le by fastforce
    next
      case False
      with s have a * b ≤ 0
        by simp
      then show ?thesis
        using a split-mult-pos-le by fastforce
    qed
  qed
qed

```

lemma *mult-le-prts*:

```

  fixes a b :: 'a::lattice-ring
  assumes a1 ≤ a

```

```

    and  $a \leq a2$ 
    and  $b1 \leq b$ 
    and  $b \leq b2$ 
  shows  $a * b \leq$ 
     $pprt\ a2 * pprt\ b2 + pprt\ a1 * nprt\ b2 + nprt\ a2 * pprt\ b1 + nprt\ a1 * nprt$ 
 $b1$ 
  proof -
    have  $a * b = (pprt\ a + nprt\ a) * (pprt\ b + nprt\ b)$ 
    by (subst prts[symmetric])+ simp
    then have  $a * b = pprt\ a * pprt\ b + pprt\ a * nprt\ b + nprt\ a * pprt\ b + nprt$ 
 $a * nprt\ b$ 
    by (simp add: algebra-simps)
    moreover have  $pprt\ a * pprt\ b \leq pprt\ a2 * pprt\ b2$ 
    by (simp-all add: assms mult-mono)
    moreover have  $pprt\ a * nprt\ b \leq pprt\ a1 * nprt\ b2$ 
  proof -
    have  $pprt\ a * nprt\ b \leq pprt\ a * nprt\ b2$ 
    by (simp add: mult-left-mono assms)
    moreover have  $pprt\ a * nprt\ b2 \leq pprt\ a1 * nprt\ b2$ 
    by (simp add: mult-right-mono-neg assms)
    ultimately show ?thesis
    by simp
  qed
  moreover have  $nprt\ a * pprt\ b \leq nprt\ a2 * pprt\ b1$ 
  proof -
    have  $nprt\ a * pprt\ b \leq nprt\ a2 * pprt\ b$ 
    by (simp add: mult-right-mono assms)
    moreover have  $nprt\ a2 * pprt\ b \leq nprt\ a2 * pprt\ b1$ 
    by (simp add: mult-left-mono-neg assms)
    ultimately show ?thesis
    by simp
  qed
  moreover have  $nprt\ a * nprt\ b \leq nprt\ a1 * nprt\ b1$ 
  proof -
    have  $nprt\ a * nprt\ b \leq nprt\ a * nprt\ b1$ 
    by (simp add: mult-left-mono-neg assms)
    moreover have  $nprt\ a * nprt\ b1 \leq nprt\ a1 * nprt\ b1$ 
    by (simp add: mult-right-mono-neg assms)
    ultimately show ?thesis
    by simp
  qed
  ultimately show ?thesis
  by - (rule add-mono | simp)+
  qed

lemma mult-ge-prts:
  fixes  $a\ b :: 'a::lattice-ring$ 
  assumes  $a1 \leq a$ 
  and  $a \leq a2$ 

```

```

    and b1 ≤ b
    and b ≤ b2
  shows a * b ≥
    npert a1 * ppert b2 + npert a2 * npert b2 + ppert a1 * ppert b1 + ppert a2 * npert
b1
proof -
  from assms have a1: - a2 ≤ -a
    by auto
  from assms have a2: - a ≤ -a1
    by auto
  from mult-le-prts[of - a2 - a - a1 b1 b b2,
    OF a1 a2 assms(3) assms(4), simplified npert-neg ppert-neg]
  have le: - (a * b) ≤
    - npert a1 * ppert b2 + - npert a2 * npert b2 +
    - ppert a1 * ppert b1 + - ppert a2 * npert b1
    by simp
  then have - (- npert a1 * ppert b2 + - npert a2 * npert b2 +
    - ppert a1 * ppert b1 + - ppert a2 * npert b1) ≤ a * b
    by (simp only: minus-le-iff)
  then show ?thesis
    by (simp add: algebra-simps)
qed

instance int :: lattice-ring
proof
  show |k| = sup k (- k) for k :: int
    by (auto simp add: sup-int-def)
qed

instance real :: lattice-ring
proof
  show |a| = sup a (- a) for a :: real
    by (auto simp add: sup-real-def)
qed

end

```

43 Floating-Point Numbers

```

theory Float
imports Log-Nat Lattice-Algebras
begin

definition float = {m * 2 powr e | (m :: int) (e :: int). True}

typedef float = float
  morphisms real-of-float float-of
  unfolding float-def by auto

```

setup-lifting *type-definition-float*

declare *real-of-float* [*code-unfold*]

lemmas *float-of-inject*[*simp*]

declare [[*coercion* *real-of-float* :: *float* \Rightarrow *real*]]

lemma *real-of-float-eq*: $f1 = f2 \longleftrightarrow \text{real-of-float } f1 = \text{real-of-float } f2$ **for** $f1\ f2 :: \text{float}$
unfolding *real-of-float-inject* ..

declare *real-of-float-inverse*[*simp*] *float-of-inverse* [*simp*]

declare *real-of-float* [*simp*]

43.1 Real operations preserving the representation as floating point number

lemma *floatI*: $m * 2^{\text{powr } e} = x \implies x \in \text{float}$ **for** $m\ e :: \text{int}$
by (*auto simp: float-def*)

lemma *zero-float*[*simp*]: $0 \in \text{float}$
by (*auto simp: float-def*)

lemma *one-float*[*simp*]: $1 \in \text{float}$
by (*intro floatI[of 1 0]*) *simp*

lemma *numeral-float*[*simp*]: *numeral* $i \in \text{float}$
by (*intro floatI[of numeral i 0]*) *simp*

lemma *neg-numeral-float*[*simp*]: $- \text{numeral } i \in \text{float}$
by (*intro floatI[of - numeral i 0]*) *simp*

lemma *real-of-int-float*[*simp*]: *real-of-int* $x \in \text{float}$ **for** $x :: \text{int}$
by (*intro floatI[of x 0]*) *simp*

lemma *real-of-nat-float*[*simp*]: *real* $x \in \text{float}$ **for** $x :: \text{nat}$
by (*intro floatI[of x 0]*) *simp*

lemma *two-powr-int-float*[*simp*]: $2^{\text{powr } (\text{real-of-int } i)} \in \text{float}$ **for** $i :: \text{int}$
by (*intro floatI[of 1 i]*) *simp*

lemma *two-powr-nat-float*[*simp*]: $2^{\text{powr } (\text{real } i)} \in \text{float}$ **for** $i :: \text{nat}$
by (*intro floatI[of 1 i]*) *simp*

lemma *two-powr-minus-int-float*[*simp*]: $2^{\text{powr } - (\text{real-of-int } i)} \in \text{float}$ **for** $i :: \text{int}$
by (*intro floatI[of 1 -i]*) *simp*

lemma *two-powr-minus-nat-float*[*simp*]: $2^{\text{powr } - (\text{real } i)} \in \text{float}$ **for** $i :: \text{nat}$

```

by (intro floatI[of 1 -i]) simp

lemma two-powr-numeral-float[simp]: 2 powr numeral i ∈ float
  by (intro floatI[of 1 numeral i]) simp

lemma two-powr-neg-numeral-float[simp]: 2 powr - numeral i ∈ float
  by (intro floatI[of 1 - numeral i]) simp

lemma two-pow-float[simp]: 2 ^ n ∈ float
  by (intro floatI[of 1 n]) (simp add: powr-realpow)

lemma plus-float[simp]: r ∈ float ⟹ p ∈ float ⟹ r + p ∈ float
  unfolding float-def
proof (safe, simp)
  have *: ∃ (m::int) (e::int). m1 * 2 powr e1 + m2 * 2 powr e2 = m * 2 powr e
    if e1 ≤ e2 for e1 m1 e2 m2 :: int
  proof -
    from that have m1 * 2 powr e1 + m2 * 2 powr e2 = (m1 + m2 * 2 ^ nat
      (e2 - e1)) * 2 powr e1
    by (simp add: powr-diff field-simps flip: powr-realpow)
    then show ?thesis
    by blast
  qed
fix e1 m1 e2 m2 :: int
consider e2 ≤ e1 | e1 ≤ e2 by (rule linorder-le-cases)
then show ∃ (m::int) (e::int). m1 * 2 powr e1 + m2 * 2 powr e2 = m * 2 powr
e
proof cases
case 1
from *[OF this, of m2 m1] show ?thesis
by (simp add: ac-simps)
next
case 2
then show ?thesis by (rule *)
qed
qed

lemma uminus-float[simp]: x ∈ float ⟹ -x ∈ float
  by (simp add: float-def) (metis mult-minus-left of-int-minus)

lemma times-float[simp]: x ∈ float ⟹ y ∈ float ⟹ x * y ∈ float
  apply (clarsimp simp: float-def mult-ac)
  by (metis mult.assoc of-int-mult of-int-add powr-add)

lemma minus-float[simp]: x ∈ float ⟹ y ∈ float ⟹ x - y ∈ float
  using plus-float [of x - y] by simp

lemma abs-float[simp]: x ∈ float ⟹ |x| ∈ float

```

```

    by (cases x rule: linorder-cases[of 0]) auto

lemma sgn-of-float[simp]:  $x \in \text{float} \implies \text{sgn } x \in \text{float}$ 
  by (simp add: sgn-real-def)

lemma div-power-2-float[simp]:  $x \in \text{float} \implies x / 2^d \in \text{float}$ 
  by (simp add: float-def) (metis of-int-diff of-int-of-nat-eq powr-diff powr-realpow
    zero-less-numeral times-divide-eq-right)

lemma div-power-2-int-float[simp]:  $x \in \text{float} \implies x / (2::\text{int})^d \in \text{float}$ 
  by simp

lemma div-numeral-Bit0-float[simp]:
  assumes  $x / \text{numeral } n \in \text{float}$ 
  shows  $x / (\text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$ 
proof -
  have  $(x / \text{numeral } n) / 2^1 \in \text{float}$ 
    by (intro assms div-power-2-float)
  also have  $(x / \text{numeral } n) / 2^1 = x / (\text{numeral } (\text{Num.Bit0 } n))$ 
    by (induct n) auto
  finally show ?thesis .
qed

lemma div-neg-numeral-Bit0-float[simp]:
  assumes  $x / \text{numeral } n \in \text{float}$ 
  shows  $x / (- \text{numeral } (\text{Num.Bit0 } n)) \in \text{float}$ 
  using assms by force

lemma power-float[simp]:
  assumes  $a \in \text{float}$ 
  shows  $a ^ b \in \text{float}$ 
proof -
  from assms obtain  $m e :: \text{int}$  where  $a = m * 2^{\text{powr } e}$ 
    by (auto simp: float-def)
  then show ?thesis
    by (intro floatI[where  $m=m$   $b=b$  and  $e=e$ ])
      (auto simp: powr-powr power-mult-distrib simp flip: powr-realpow)
qed

lift-definition Float ::  $\text{int} \Rightarrow \text{int} \Rightarrow \text{float}$  is  $\lambda(m::\text{int}) (e::\text{int}). m * 2^{\text{powr } e}$ 
  by simp
declare Float.rep-eq[simp]

code-datatype Float

lemma compute-real-of-float[code]:
  real-of-float (Float  $m e$ ) = (if  $e \geq 0$  then  $m * 2^{\text{nat } e}$  else  $m / 2^{\text{nat } (-e)}$ )
  by (simp add: powr-int)

```

43.2 Arithmetic operations on floating point numbers

instantiation *float* :: {*ring-1*, *linorder*, *linordered-ring*, *linordered-idom*, *numeral*, *equal*}
begin

lift-definition *zero-float* :: *float* **is** 0 **by** *simp*
declare *zero-float.rep-eq*[*simp*]

lift-definition *one-float* :: *float* **is** 1 **by** *simp*
declare *one-float.rep-eq*[*simp*]

lift-definition *plus-float* :: *float* \Rightarrow *float* \Rightarrow *float* **is** (+) **by** *simp*
declare *plus-float.rep-eq*[*simp*]

lift-definition *times-float* :: *float* \Rightarrow *float* \Rightarrow *float* **is** (*) **by** *simp*
declare *times-float.rep-eq*[*simp*]

lift-definition *minus-float* :: *float* \Rightarrow *float* \Rightarrow *float* **is** (−) **by** *simp*
declare *minus-float.rep-eq*[*simp*]

lift-definition *uminus-float* :: *float* \Rightarrow *float* **is** *uminus* **by** *simp*
declare *uminus-float.rep-eq*[*simp*]

lift-definition *abs-float* :: *float* \Rightarrow *float* **is** *abs* **by** *simp*
declare *abs-float.rep-eq*[*simp*]

lift-definition *sgn-float* :: *float* \Rightarrow *float* **is** *sgn* **by** *simp*
declare *sgn-float.rep-eq*[*simp*]

lift-definition *equal-float* :: *float* \Rightarrow *float* \Rightarrow *bool* **is** (=) :: *real* \Rightarrow *real* \Rightarrow *bool* .

lift-definition *less-eq-float* :: *float* \Rightarrow *float* \Rightarrow *bool* **is** (\leq) .
declare *less-eq-float.rep-eq*[*simp*]

lift-definition *less-float* :: *float* \Rightarrow *float* \Rightarrow *bool* **is** (<) .
declare *less-float.rep-eq*[*simp*]

instance

by *standard* (*transfer*; *fastforce simp add: field-simps intro: mult-left-mono mult-right-mono*) +

end

lemma *real-of-float* [*simp*]: *real-of-float* (*of-nat* *n*) = *of-nat* *n*
by (*induct* *n*) *simp-all*

lemma *real-of-float-of-int-eq* [*simp*]: *real-of-float* (*of-int* *z*) = *of-int* *z*
by (*cases* *z* *rule: int-diff-cases*) (*simp-all add: of-rat-diff*)

lemma *Float-0-eq-0*[*simp*]: *Float* 0 *e* = 0
by *transfer simp*

lemma *real-of-float-power*[simp]: *real-of-float* (f^n) = *real-of-float* f^n **for** $f :: \text{float}$
by (*induct* n) *simp-all*

lemma *real-of-float-min*: *real-of-float* ($\min x y$) = *min* (*real-of-float* x) (*real-of-float* y)
and *real-of-float-max*: *real-of-float* ($\max x y$) = *max* (*real-of-float* x) (*real-of-float* y)
for $x y :: \text{float}$
by (*simp-all add: min-def max-def*)

instance *float* :: *unbounded-dense-linorder*

proof

fix $a b :: \text{float}$
show $\exists c. a < c$
by (*metis* *Float.real-of-float less-float.rep-eq reals-Archimedean2*)
show $\exists c. c < a$
by (*metis* *add-0 add-strict-right-mono neg-less-0-iff-less zero-less-one*)
show $\exists c. a < c \wedge c < b$ **if** $a < b$
apply (*rule* *exI*[*of* - ($a + b$) * *Float* 1 ($- 1$)])
using *that*
apply *transfer*
apply (*simp add: powr-minus*)
done

qed

instantiation *float* :: *lattice-ab-group-add*

begin

definition *inf-float* :: *float* \Rightarrow *float* \Rightarrow *float*
where *inf-float* $a b = \min a b$

definition *sup-float* :: *float* \Rightarrow *float* \Rightarrow *float*
where *sup-float* $a b = \max a b$

instance

by *standard* (*transfer*; *simp add: inf-float-def sup-float-def real-of-float-min real-of-float-max*) +

end

lemma *float-numeral*[simp]: *real-of-float* (*numeral* $x :: \text{float}$) = *numeral* x
by (*metis* *of-int-numeral real-of-float-of-int-eq*)

lemma *transfer-numeral* [*transfer-rule*]:

rel-fun ($=$) *pcr-float* (*numeral* :: $- \Rightarrow \text{real}$) (*numeral* :: $- \Rightarrow \text{float}$)
by (*simp add: rel-fun-def float.pcr-cr-eq cr-float-def*)

lemma *float-neg-numeral*[simp]: *real-of-float* ($- \text{numeral } x :: \text{float}$) = $- \text{numeral } x$

by *simp*

lemma *transfer-neg-numeral* [*transfer-rule*]:

rel-fun (*=*) *pcr-float* (*- numeral :: - \Rightarrow real*) (*- numeral :: - \Rightarrow float*)

by (*simp add: rel-fun-def float.pcr-cr-eq cr-float-def*)

lemma *float-of-numeral*: *numeral k = float-of (numeral k)*

and *float-of-neg-numeral*: *- numeral k = float-of (- numeral k)*

unfolding *real-of-float-eq* **by** *simp-all*

43.3 Quickcheck

instantiation *float :: exhaustive*

begin

definition *exhaustive-float* **where**

exhaustive-float f d =

Quickcheck-Exhaustive.exhaustive ($\lambda x.$ *Quickcheck-Exhaustive.exhaustive* ($\lambda y.$ *f* (*Float x y*)) *d*) *d*

instance ..

end

context

includes *term-syntax*

begin

definition [*code-unfold*]:

valtermify-float x y = Code-Evaluation.valtermify Float {·} x {·} y

end

instantiation *float :: full-exhaustive*

begin

definition

full-exhaustive-float f d =

Quickcheck-Exhaustive.full-exhaustive

($\lambda x.$ *Quickcheck-Exhaustive.full-exhaustive* ($\lambda y.$ *f* (*valtermify-float x y*)) *d*) *d*

instance ..

end

instantiation *float :: random*

begin

definition *Quickcheck-Random.random i =*

```

    scomp (Quickcheck-Random.random (2 ^ nat-of-natural i))
      (λman. scomp (Quickcheck-Random.random i) (λexp. Pair (valtermify-float
man exp)))

```

```
instance ..
```

```
end
```

43.4 Represent floats as unique mantissa and exponent

lemma *int-induct-abs*[*case-names less*]:

```

  fixes j :: int
  assumes H:  $\bigwedge n. (\bigwedge i. |i| < |n| \implies P i) \implies P n$ 
  shows P j
proof (induct nat |j| arbitrary: j rule: less-induct)
  case less
  show ?case by (rule H[OF less]) simp
qed

```

lemma *int-cancel-factors*:

```

  fixes n :: int
  assumes 1 < r
  shows  $n = 0 \vee (\exists k i. n = k * r ^ i \wedge \neg r \text{ dvd } k)$ 
proof (induct n rule: int-induct-abs)
  case (less n)
  have  $\exists k i. n = k * r ^ \text{Suc } i \wedge \neg r \text{ dvd } k$  if  $n \neq 0$  n  $n = m * r$  for m
  proof –
    from that have  $|m| < |n|$ 
    using  $\langle 1 < r \rangle$  by (simp add: abs-mult)
    from less[OF this] that show ?thesis by auto
  qed
  then show ?case
    by (metis dvd-def monoid-mult-class.mult.right-neutral mult.commute power-0)
qed

```

lemma *mult-powr-eq-mult-powr-iff-asym*:

```

  fixes m1 m2 e1 e2 :: int
  assumes m1:  $\neg 2 \text{ dvd } m1$ 
    and e1 ≤ e2
  shows  $m1 * 2^{\text{powr } e1} = m2 * 2^{\text{powr } e2} \longleftrightarrow m1 = m2 \wedge e1 = e2$ 
    (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  show ?rhs if eq: ?lhs
  proof –
    have  $m1 \neq 0$ 
    using m1 unfolding dvd-def by auto
    from  $\langle e1 \leq e2 \rangle$  eq have  $m1 = m2 * 2^{\text{powr } \text{nat } (e2 - e1)}$ 
    by (simp add: powr-diff field-simps)
    also have  $\dots = m2 * 2^{\text{nat } (e2 - e1)}$ 

```

```

    by (simp add: powr-realpow)
  finally have m1-eq:  $m1 = m2 * 2^{\text{nat } (e2 - e1)}$ 
    by linarith
  with m1 have m1 = m2
    by (cases nat (e2 - e1)) (auto simp add: dvd-def)
  then show ?thesis
    using eq ⟨ $m1 \neq 0$ ⟩ by (simp add: powr-inj)
qed
show ?lhs if ?rhs
  using that by simp
qed

```

```

lemma mult-powr-eq-mult-powr-iff:
   $\neg 2 \text{ dvd } m1 \implies \neg 2 \text{ dvd } m2 \implies m1 * 2^{\text{powr } e1} = m2 * 2^{\text{powr } e2} \longleftrightarrow m1 = m2 \wedge e1 = e2$ 
  for  $m1\ m2\ e1\ e2 :: \text{int}$ 
  using mult-powr-eq-mult-powr-iff-asm[of  $m1\ e1\ e2\ m2$ ]
  using mult-powr-eq-mult-powr-iff-asm[of  $m2\ e2\ e1\ m1$ ]
  by (cases e1 e2 rule: linorder-le-cases) auto

```

```

lemma floatE-normed:
  assumes  $x: x \in \text{float}$ 
  obtains (zero)  $x = 0$ 
  | (powr)  $m\ e :: \text{int}$  where  $x = m * 2^{\text{powr } e} \wedge \neg 2 \text{ dvd } m\ x \neq 0$ 
proof -
  have  $\exists (m::\text{int})\ (e::\text{int}). x = m * 2^{\text{powr } e} \wedge \neg (2::\text{int}) \text{ dvd } m$  if  $x \neq 0$ 
  proof -
    from x obtain  $m\ e :: \text{int}$  where  $x: x = m * 2^{\text{powr } e}$ 
    by (auto simp: float-def)
    with ⟨ $x \neq 0$ ⟩ int-cancel-factors[of  $2\ m$ ] obtain  $k\ i$  where  $m = k * 2^i \wedge \neg 2 \text{ dvd } k$ 
    by auto
    with ⟨ $\neg 2 \text{ dvd } k$ ⟩ x have  $x = \text{real-of-int } k * 2^{\text{powr } \text{real-of-int } (e + \text{int } i)} \wedge$ 
    odd  $k$ 
    by (simp add: powr-add powr-realpow)
    then show ?thesis
    by blast
  qed
  with that show thesis by blast
qed

```

```

lemma float-normed-cases:
  fixes  $f :: \text{float}$ 
  obtains (zero)  $f = 0$ 
  | (powr)  $m\ e :: \text{int}$  where  $\text{real-of-float } f = m * 2^{\text{powr } e} \wedge \neg 2 \text{ dvd } m\ f \neq 0$ 
proof (atomize-elim, induct f)
  case (float-of y)
  then show ?case
    by (cases rule: floatE-normed) (auto simp: zero-float-def)

```

qed

definition *mantissa* :: float \Rightarrow int

where *mantissa* *f* =
 $\text{fst } (\text{SOME } p::\text{int} \times \text{int}. (f = 0 \wedge \text{fst } p = 0 \wedge \text{snd } p = 0) \vee$
 $(f \neq 0 \wedge \text{real-of-float } f = \text{real-of-int } (\text{fst } p) * 2^{\text{powr real-of-int } (\text{snd } p)} \wedge \neg$
 $2 \text{ dvd fst } p))$

definition *exponent* :: float \Rightarrow int

where *exponent* *f* =
 $\text{snd } (\text{SOME } p::\text{int} \times \text{int}. (f = 0 \wedge \text{fst } p = 0 \wedge \text{snd } p = 0) \vee$
 $(f \neq 0 \wedge \text{real-of-float } f = \text{real-of-int } (\text{fst } p) * 2^{\text{powr real-of-int } (\text{snd } p)} \wedge \neg$
 $2 \text{ dvd fst } p))$

lemma *exponent-0[simp]*: *exponent* 0 = 0 (is ?E)

and *mantissa-0[simp]*: *mantissa* 0 = 0 (is ?M)

proof –

have $\bigwedge p::\text{int} \times \text{int}. \text{fst } p = 0 \wedge \text{snd } p = 0 \longleftrightarrow p = (0, 0)$

by *auto*

then show ?E ?M

by (*auto simp add: mantissa-def exponent-def zero-float-def*)

qed

lemma *mantissa-exponent*: *real-of-float* *f* = *mantissa* *f* * 2^{*powr exponent f*} (is ?E)

and *mantissa-not-dvd*: $f \neq 0 \implies \neg 2 \text{ dvd mantissa } f$ (is - \implies ?D)

proof *cases*

assume [*simp*]: $f \neq 0$

have $f = \text{mantissa } f * 2^{\text{powr exponent } f} \wedge \neg 2 \text{ dvd mantissa } f$

proof (*cases f rule: float-normed-cases*)

case *zero*

then show ?thesis **by** *simp*

next

case (*powr m e*)

then have $\exists p::\text{int} \times \text{int}. (f = 0 \wedge \text{fst } p = 0 \wedge \text{snd } p = 0) \vee$

$(f \neq 0 \wedge \text{real-of-float } f = \text{real-of-int } (\text{fst } p) * 2^{\text{powr real-of-int } (\text{snd } p)} \wedge \neg$
 $2 \text{ dvd fst } p)$

by *auto*

then show ?thesis

unfolding *exponent-def mantissa-def*

by (*rule someI2-ex simp*)

qed

then show ?E ?D **by** *auto*

qed *simp*

lemma *mantissa-noteq-0*: $f \neq 0 \implies \text{mantissa } f \neq 0$

using *mantissa-not-dvd[of f]* **by** *auto*

lemma *mantissa-eq-zero-iff*: $\text{mantissa } x = 0 \longleftrightarrow x = 0$

```

  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  show ?rhs if ?lhs
  proof -
    from that have z: 0 = real-of-float x
    using mantissa-exponent by simp
    show ?thesis
    by (simp add: zero-float-def z)
  qed
  show ?lhs if ?rhs
  using that by simp
qed

lemma mantissa-pos-iff: 0 < mantissa x  $\longleftrightarrow$  0 < x
  by (auto simp: mantissa-exponent algebra-split-simps)

lemma mantissa-nonneg-iff: 0  $\leq$  mantissa x  $\longleftrightarrow$  0  $\leq$  x
  by (auto simp: mantissa-exponent algebra-split-simps)

lemma mantissa-neg-iff: 0 > mantissa x  $\longleftrightarrow$  0 > x
  by (auto simp: mantissa-exponent algebra-split-simps)

lemma
  fixes m e :: int
  defines f  $\equiv$  float-of (m * 2 powr e)
  assumes dvd:  $\neg$  2 dvd m
  shows mantissa-float: mantissa f = m (is ?M)
    and exponent-float: m  $\neq$  0  $\implies$  exponent f = e (is -  $\implies$  ?E)
proof cases
  assume m = 0
  with dvd show mantissa f = m by auto
next
  assume m  $\neq$  0
  then have f-not-0: f  $\neq$  0 by (simp add: f-def zero-float-def)
  from mantissa-exponent[of f] have m * 2 powr e = mantissa f * 2 powr exponent
  f
  by (auto simp add: f-def)
  then show ?M ?E
  using mantissa-not-dvd[OF f-not-0] dvd
  by (auto simp: mult-powr-eq-mult-powr-iff)
qed

```

43.5 Compute arithmetic operations

```

lemma Float-mantissa-exponent: Float (mantissa f) (exponent f) = f
  unfolding real-of-float-eq mantissa-exponent[of f] by simp

```

```

lemma Float-cases [cases type: float]:
  fixes f :: float

```

obtains $(\text{Float})\ m\ e :: \text{int}$ **where** $f = \text{Float}\ m\ e$
using *Float-mantissa-exponent[symmetric]*
by *(atomize-elim) auto*

lemma *denormalize-shift:*

assumes *f-def*: $f = \text{Float}\ m\ e$
and *not-0*: $f \neq 0$

obtains i **where** $m = \text{mantissa}\ f * 2^i$ $e = \text{exponent}\ f - i$

proof

from *mantissa-exponent[of f] f-def*

have $m * 2^{\text{powr}\ e} = \text{mantissa}\ f * 2^{\text{powr}\ \text{exponent}\ f}$

by *simp*

then have *eq*: $m = \text{mantissa}\ f * 2^{\text{powr}\ (\text{exponent}\ f - e)}$

by *(simp add: powr-diff field-simps)*

moreover

have $e \leq \text{exponent}\ f$

proof *(rule ccontr)*

assume $\neg e \leq \text{exponent}\ f$

then have *pos*: $\text{exponent}\ f < e$ **by** *simp*

then have $2^{\text{powr}\ (\text{exponent}\ f - e)} = 2^{\text{powr}\ - \text{real-of-int}\ (e - \text{exponent}\ f)}$

by *simp*

also have $\dots = 1 / 2^{\text{nat}\ (e - \text{exponent}\ f)}$

using *pos* **by** *(simp flip: powr-realpow add: powr-diff)*

finally have $m * 2^{\text{nat}\ (e - \text{exponent}\ f)} = \text{real-of-int}\ (\text{mantissa}\ f)$

using *eq* **by** *simp*

then have $\text{mantissa}\ f = m * 2^{\text{nat}\ (e - \text{exponent}\ f)}$

by *linarith*

with $\langle \text{exponent}\ f < e \rangle$ **have** $2 \text{ dvd } \text{mantissa}\ f$

by *(force intro: dvdI[where k=m * 2^(nat (e-exponent f)) div 2])*

then show *False* **using** *mantissa-not-dvd[OF not-0]* **by** *simp*

qed

ultimately have $\text{real-of-int}\ m = \text{mantissa}\ f * 2^{\text{nat}\ (\text{exponent}\ f - e)}$

by *(simp flip: powr-realpow)*

with $\langle e \leq \text{exponent}\ f \rangle$

show $m = \text{mantissa}\ f * 2^{\text{nat}\ (\text{exponent}\ f - e)}$

by *linarith*

show $e = \text{exponent}\ f - \text{nat}\ (\text{exponent}\ f - e)$

using $\langle e \leq \text{exponent}\ f \rangle$ **by** *auto*

qed

context

begin

qualified lemma *compute-float-zero[code-unfold, code]*: $0 = \text{Float}\ 0\ 0$

by *transfer simp*

qualified lemma *compute-float-one[code-unfold, code]*: $1 = \text{Float}\ 1\ 0$

by *transfer simp*

lift-definition *normfloat* :: *float* \Rightarrow *float* **is** $\lambda x. x$.

lemma *normloat-id[simp]*: *normfloat* $x = x$ **by** *transfer rule*

qualified lemma *compute-normfloat[code]*:

normfloat (*Float* $m\ e$) =
 (if $m \bmod 2 = 0 \wedge m \neq 0$ then *normfloat* (*Float* ($m \div 2$) ($e + 1$))
 else if $m = 0$ then 0 else *Float* $m\ e$)
by *transfer (auto simp add: powr-add zmod-eq-0-iff)*

qualified lemma *compute-float-numeral[code-abbrev]*: *Float* (*numeral* k) 0 = *numeral* k

by *transfer simp*

qualified lemma *compute-float-neg-numeral[code-abbrev]*: *Float* ($-$ *numeral* k) 0 = $-$ *numeral* k

by *transfer simp*

qualified lemma *compute-float-uminus[code]*: $-$ *Float* $m1\ e1 =$ *Float* ($-$ $m1$) $e1$

by *transfer simp*

qualified lemma *compute-float-times[code]*: *Float* $m1\ e1 *$ *Float* $m2\ e2 =$ *Float* ($m1 * m2$) ($e1 + e2$)

by *transfer (simp add: field-simps powr-add)*

qualified lemma *compute-float-plus[code]*:

Float $m1\ e1 +$ *Float* $m2\ e2 =$
 (if $m1 = 0$ then *Float* $m2\ e2$
 else if $m2 = 0$ then *Float* $m1\ e1$
 else if $e1 \leq e2$ then *Float* ($m1 + m2 * 2^{\text{nat}} (e2 - e1)$) $e1$
 else *Float* ($m2 + m1 * 2^{\text{nat}} (e1 - e2)$) $e2$)
by *transfer (simp add: field-simps powr-realpow[symmetric] powr-diff)*

qualified lemma *compute-float-minus[code]*: $f - g = f + (-g)$ **for** $f\ g :: \text{float}$

by *simp*

qualified lemma *compute-float-sgn[code]*:

sgn (*Float* $m1\ e1$) = (if $0 < m1$ then 1 else if $m1 < 0$ then -1 else 0)
by *transfer (simp add: sgn-mult)*

lift-definition *is-float-pos* :: *float* \Rightarrow *bool* **is** ($<$) 0 :: *real* \Rightarrow *bool* .

qualified lemma *compute-is-float-pos[code]*: *is-float-pos* (*Float* $m\ e$) $\longleftrightarrow 0 < m$

by *transfer (auto simp add: zero-less-mult-iff not-le[symmetric, of - 0])*

lift-definition *is-float-nonneg* :: *float* \Rightarrow *bool* **is** (\leq) 0 :: *real* \Rightarrow *bool* .

qualified lemma *compute-is-float-nonneg[code]*: *is-float-nonneg* (*Float* $m\ e$) $\longleftrightarrow 0 \leq m$

by *transfer (auto simp add: zero-le-mult-iff not-less[symmetric, of - 0])*

lift-definition *is-float-zero* :: *float* \Rightarrow *bool* **is** (=) 0 :: *real* \Rightarrow *bool* .

qualified lemma *compute-is-float-zero*[code]: *is-float-zero* (*Float* *m* *e*) \longleftrightarrow 0 = *m*
by *transfer* (*auto simp add: is-float-zero-def*)

qualified lemma *compute-float-abs*[code]: |*Float* *m* *e*| = *Float* |*m*| *e*
by *transfer* (*simp add: abs-mult*)

qualified lemma *compute-float-eq*[code]: *equal-class.equal* *f* *g* = *is-float-zero* (*f* – *g*)
by *transfer simp*

end

43.6 Lemmas for types *real*, *nat*, *int*

lemmas *real-of-ints* =
of-int-add
of-int-minus
of-int-diff
of-int-mult
of-int-power
of-int-numeral of-int-neg-numeral

lemmas *int-of-reals* = *real-of-ints*[*symmetric*]

43.7 Rounding Real Numbers

definition *round-down* :: *int* \Rightarrow *real* \Rightarrow *real*
where *round-down* *prec* *x* = $\lfloor x * 2^{\text{powr } \text{prec}} \rfloor * 2^{\text{powr } -\text{prec}}$

definition *round-up* :: *int* \Rightarrow *real* \Rightarrow *real*
where *round-up* *prec* *x* = $\lceil x * 2^{\text{powr } \text{prec}} \rceil * 2^{\text{powr } -\text{prec}}$

lemma *round-down-float*[*simp*]: *round-down* *prec* *x* \in *float*
unfolding *round-down-def*
by (*auto intro!: times-float simp flip: of-int-minus*)

lemma *round-up-float*[*simp*]: *round-up* *prec* *x* \in *float*
unfolding *round-up-def*
by (*auto intro!: times-float simp flip: of-int-minus*)

lemma *round-up*: $x \leq \text{round-up } \text{prec } x$
by (*simp add: powr-minus-divide le-divide-eq round-up-def ceiling-correct*)

lemma *round-down*: $\text{round-down } \text{prec } x \leq x$
by (*simp add: powr-minus-divide divide-le-eq round-down-def*)

lemma *round-up-0*[*simp*]: *round-up* *p* 0 = 0

unfolding *round-up-def* **by** *simp*

lemma *round-down-0[simp]*: *round-down* *p* 0 = 0

unfolding *round-down-def* **by** *simp*

lemma *round-up-diff-round-down*: *round-up* *prec* *x* − *round-down* *prec* *x* ≤ 2 *powr* −*prec*

proof −

have *round-up* *prec* *x* − *round-down* *prec* *x* = ($\lceil x * 2^{\text{powr } \text{prec}} \rceil - \lfloor x * 2^{\text{powr } \text{prec}} \rfloor$) * 2 *powr* −*prec*

by (*simp* *add*: *round-up-def* *round-down-def* *field-simps*)

also have ... ≤ 1 * 2 *powr* −*prec*

by (*rule* *mult-mono*)

(*auto* *simp* *flip*: *of-int-diff* *simp*: *ceiling-diff-floor-le-1*)

finally show ?*thesis* **by** *simp*

qed

lemma *round-down-shift*: *round-down* *p* (*x* * 2 *powr* *k*) = 2 *powr* *k* * *round-down* (*p* + *k*) *x*

unfolding *round-down-def*

by (*simp* *add*: *powr-add* *powr-mult* *field-simps* *powr-diff*)

(*simp* *flip*: *powr-add*)

lemma *round-up-shift*: *round-up* *p* (*x* * 2 *powr* *k*) = 2 *powr* *k* * *round-up* (*p* + *k*) *x*

unfolding *round-up-def*

by (*simp* *add*: *powr-add* *powr-mult* *field-simps* *powr-diff*)

(*simp* *flip*: *powr-add*)

lemma *round-up-uminus-eq*: *round-up* *p* (−*x*) = − *round-down* *p* *x*

and *round-down-uminus-eq*: *round-down* *p* (−*x*) = − *round-up* *p* *x*

by (*auto* *simp*: *round-up-def* *round-down-def* *ceiling-def*)

lemma *round-up-mono*: *x* ≤ *y* ⇒ *round-up* *p* *x* ≤ *round-up* *p* *y*

by (*auto* *intro*!: *ceiling-mono* *simp*: *round-up-def*)

lemma *round-up-le1*:

assumes *x* ≤ 1 *prec* ≥ 0

shows *round-up* *prec* *x* ≤ 1

proof −

have *real-of-int* $\lceil x * 2^{\text{powr } \text{prec}} \rceil \leq \text{real-of-int } \lceil 2^{\text{powr } \text{prec}} \rceil$

using *assms* **by** (*auto* *intro*!: *ceiling-mono*)

also have ... = 2 *powr* *prec* **using** *assms* **by** (*auto* *simp*: *powr-int* *intro*!: *exI*[*where* *x*=2^{nat} *prec*])

finally show ?*thesis*

by (*simp* *add*: *round-up-def*) (*simp* *add*: *powr-minus* *inverse-eq-divide*)

qed

lemma *round-up-less1*:

```

assumes  $x < 1 / 2 \wedge p > 0$ 
shows  $\text{round-up } p \ x < 1$ 
proof –
  have  $x * 2^{\text{powr } p} < 1 / 2 * 2^{\text{powr } p}$ 
    using assms by simp
  also have  $\dots \leq 2^{\text{powr } p} - 1$  using  $\langle p > 0 \rangle$ 
    by (auto simp: powr-diff powr-int field-simps self-le-power)
  finally show ?thesis using  $\langle p > 0 \rangle$ 
    by (simp add: round-up-def field-simps powr-minus powr-int ceiling-less-iff)
qed

```

```

lemma round-down-ge1:
  assumes  $x: x \geq 1$ 
  assumes prec:  $p \geq -\log 2 \ x$ 
  shows  $1 \leq \text{round-down } p \ x$ 
proof cases
  assume nonneg:  $0 \leq p$ 
  have  $2^{\text{powr } p} = \text{real-of-int } \lfloor 2^{\text{powr } \text{real-of-int } p} \rfloor$ 
    using nonneg by (auto simp: powr-int)
  also have  $\dots \leq \text{real-of-int } \lfloor x * 2^{\text{powr } p} \rfloor$ 
    using assms by (auto intro!: floor-mono)
  finally show ?thesis
    by (simp add: round-down-def) (simp add: powr-minus inverse-eq-divide)
next
  assume neg:  $\neg 0 \leq p$ 
  have  $x = 2^{\text{powr } (\log 2 \ x)}$ 
    using x by simp
  also have  $2^{\text{powr } (\log 2 \ x)} \geq 2^{\text{powr } -p}$ 
    using prec by auto
  finally have  $x \text{-le: } x \geq 2^{\text{powr } -p}$  .

```

```

from neg have  $2^{\text{powr } \text{real-of-int } p} \leq 2^{\text{powr } 0}$ 
  by (intro powr-mono) auto
also have  $\dots \leq \lfloor 2^{\text{powr } 0::\text{real}} \rfloor$  by simp
also have  $\dots \leq \lfloor x * 2^{\text{powr } (\text{real-of-int } p)} \rfloor$ 
  unfolding of-int-le-iff
  using x x-le by (intro floor-mono) (simp add: powr-minus-divide field-simps)
finally show ?thesis
  using prec x
  by (simp add: round-down-def powr-minus-divide pos-le-divide-eq)
qed

```

```

lemma round-up-le0:  $x \leq 0 \implies \text{round-up } p \ x \leq 0$ 
  unfolding round-up-def
  by (auto simp: field-simps mult-le-0-iff zero-le-mult-iff)

```

43.8 Rounding Floats

```

definition div-twopow ::  $\text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$ 

```

where $[simp]: \text{div-twoPow } x \ n = x \text{ div } (2 \wedge n)$

definition $\text{mod-twoPow} :: \text{int} \Rightarrow \text{nat} \Rightarrow \text{int}$
where $[simp]: \text{mod-twoPow } x \ n = x \text{ mod } (2 \wedge n)$

lemma $\text{compute-div-twoPow}[code]:$
 $\text{div-twoPow } x \ n = (\text{if } x = 0 \vee x = -1 \vee n = 0 \text{ then } x \text{ else } \text{div-twoPow } (x \text{ div } 2) \ (n - 1))$
by $(\text{cases } n) \ (\text{auto simp: zdiv-zmult2-eq div-eq-minus1})$

lemma $\text{compute-mod-twoPow}[code]:$
 $\text{mod-twoPow } x \ n = (\text{if } n = 0 \text{ then } 0 \text{ else } x \text{ mod } 2 + 2 * \text{mod-twoPow } (x \text{ div } 2) \ (n - 1))$
by $(\text{cases } n) \ (\text{auto simp: zmod-zmult2-eq})$

lift-definition $\text{float-up} :: \text{int} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** round-up **by** simp
declare $\text{float-up.rep-eq}[simp]$

lemma $\text{round-up-correct}: \text{round-up } e \ f - f \in \{0..2^{\text{powr } -e}\}$
unfolding atLeastAtMost-iff
proof
have $\text{round-up } e \ f - f \leq \text{round-up } e \ f - \text{round-down } e \ f$
using round-down **by** simp
also have $\dots \leq 2^{\text{powr } -e}$
using $\text{round-up-diff-round-down}$ **by** simp
finally show $\text{round-up } e \ f - f \leq 2^{\text{powr } -e} - (\text{real-of-int } e)$
by simp
qed $(\text{simp add: algebra-simps round-up})$

lemma $\text{float-up-correct}: \text{real-of-float } (\text{float-up } e \ f) - \text{real-of-float } f \in \{0..2^{\text{powr } -e}\}$
by $\text{transfer (rule round-up-correct)}$

lift-definition $\text{float-down} :: \text{int} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** round-down **by** simp
declare $\text{float-down.rep-eq}[simp]$

lemma $\text{round-down-correct}: f - (\text{round-down } e \ f) \in \{0..2^{\text{powr } -e}\}$
unfolding atLeastAtMost-iff
proof
have $f - \text{round-down } e \ f \leq \text{round-up } e \ f - \text{round-down } e \ f$
using round-up **by** simp
also have $\dots \leq 2^{\text{powr } -e}$
using $\text{round-up-diff-round-down}$ **by** simp
finally show $f - \text{round-down } e \ f \leq 2^{\text{powr } -e} - (\text{real-of-int } e)$
by simp
qed $(\text{simp add: algebra-simps round-down})$

lemma $\text{float-down-correct}: \text{real-of-float } f - \text{real-of-float } (\text{float-down } e \ f) \in \{0..2^{\text{powr } -e}\}$

```

    by transfer (rule round-down-correct)

context
begin

qualified lemma compute-float-down[code]:
  float-down p (Float m e) =
    (if p + e < 0 then Float (div-twopow m (nat (-(p + e)))) (-p) else Float m e)
proof (cases p + e < 0)
  case True
  then have real-of-int ((2::int) ^ nat (-(p + e))) = 2 powr (-(p + e))
    using powr-realpow[of 2 nat (-(p + e))] by simp
  also have ... = 1 / 2 powr p / 2 powr e
    unfolding powr-minus-divide of-int-minus by (simp add: powr-add)
  finally show ?thesis
    using ⟨p + e < 0⟩
    apply transfer
  apply (simp add: round-down-def field-simps flip: floor-divide-of-int-eq powr-add)
  apply (metis (no-types, opaque-lifting) Float.rep-eq
    add.inverse-inverse compute-real-of-float diff-minus-eq-add
    floor-divide-of-int-eq int-of-reals(1) linorder-not-le
    minus-add-distrib of-int-eq-numeral-power-cancel-iff)
  done
next
  case False
  then have r: real-of-int e + real-of-int p = real (nat (e + p))
    by simp
  have r: ⌊(m * 2 powr e) * 2 powr real-of-int p⌋ = (m * 2 powr e) * 2 powr
    real-of-int p
    by (auto intro: exI[where x=m*2^nat (e+p)]
      simp add: ac-simps powr-add[symmetric] r powr-realpow)
  with ⟨¬ p + e < 0⟩ show ?thesis
    by transfer (auto simp add: round-down-def field-simps powr-add powr-minus)
qed

lemma abs-round-down-le: |f - (round-down e f)| ≤ 2 powr -e
  using round-down-correct[of f e] by simp

lemma abs-round-up-le: |f - (round-up e f)| ≤ 2 powr -e
  using round-up-correct[of e f] by simp

lemma round-down-nonneg: 0 ≤ s ⟹ 0 ≤ round-down p s
  by (auto simp: round-down-def)

lemma ceil-divide-floor-conv:
  assumes b ≠ 0
  shows ⌈real-of-int a / real-of-int b⌉ =
    (if b dvd a then a div b else ⌊real-of-int a / real-of-int b⌋ + 1)
proof (cases b dvd a)

```

```

case True
then show ?thesis
  by (simp add: ceiling-def floor-divide-of-int-eq dvd-neg-div
      flip: of-int-minus divide-minus-left)
next
case False
then have  $a \bmod b \neq 0$ 
  by auto
then have  $ne: \text{real-of-int } (a \bmod b) / \text{real-of-int } b \neq 0$ 
  using  $\langle b \neq 0 \rangle$  by auto
have  $\lceil \text{real-of-int } a / \text{real-of-int } b \rceil = \lfloor \text{real-of-int } a / \text{real-of-int } b \rfloor + 1$ 
  by (metis add-cancel-left-right ceiling-altdef floor-divide-of-int-eq ne of-int-div-aux)
then show ?thesis
  using  $\langle \neg b \text{ dvd } a \rangle$  by simp
qed

```

```

qualified lemma compute-float-up[code]: float-up p x = - float-down p (-x)
by transfer (simp add: round-down-uminus-eq)

```

end

lemma *bitlen-Float:*

```

  fixes  $m\ e$ 
  defines [THEN meta-eq-to-obj-eq]:  $f \equiv \text{Float } m\ e$ 
  shows  $\text{bitlen } |\text{mantissa } f| + \text{exponent } f = (\text{if } m = 0 \text{ then } 0 \text{ else } \text{bitlen } |m| + e)$ 
proof (cases m = 0)
  case True
  then show ?thesis by (simp add: f-def bitlen-alt-def)
next
  case False
  then have  $f \neq 0$ 
    unfolding real-of-float-eq by (simp add: f-def)
  then have  $\text{mantissa } f \neq 0$ 
    by (simp add: mantissa-eq-zero-iff)
  moreover
  obtain  $i$  where  $m = \text{mantissa } f * 2^i$   $e = \text{exponent } f - \text{int } i$ 
    by (rule f-def[THEN denormalize-shift, OF <f ≠ 0>])
  ultimately show ?thesis by (simp add: abs-mult)
qed

```

lemma *float-gt1-scale:*

```

  assumes  $1 \leq \text{Float } m\ e$ 
  shows  $0 \leq e + (\text{bitlen } m - 1)$ 
proof –
  have  $0 < \text{Float } m\ e$  using assms by auto
  then have  $0 < m$  using powr-gt-zero[of 2 e]
    by (auto simp: zero-less-mult-iff)
  then have  $m \neq 0$  by auto

```

```

show ?thesis
proof (cases 0 ≤ e)
  case True
  then show ?thesis
    using ⟨0 < m⟩ by (simp add: bitlen-alt-def)
next
  case False
  have (1::int) < 2 by simp
  let ?S = 2^(nat (-e))
  have inverse (2 ^ nat (- e)) = 2 powr e
    using assms False powr-realpow[of 2 nat (-e)]
    by (auto simp: powr-minus field-simps)
  then have 1 ≤ real-of-int m * inverse ?S
    using assms False powr-realpow[of 2 nat (-e)]
    by (auto simp: powr-minus)
  then have 1 * ?S ≤ real-of-int m * inverse ?S * ?S
    by (rule mult-right-mono) auto
  then have ?S ≤ real-of-int m
    unfolding mult.assoc by auto
  then have ?S ≤ m
    unfolding of-int-le-iff[symmetric] by auto
  from this bitlen-bounds[OF ⟨0 < m⟩, THEN conjunct2]
  have nat (-e) < (nat (bitlen m))
    unfolding power-strict-increasing-iff[OF ⟨1 < 2⟩, symmetric]
    by (rule order-le-less-trans)
  then have -e < bitlen m
    using False by auto
  then show ?thesis
    by auto
qed
qed

```

43.9 Truncating Real Numbers

definition *truncate-down::nat ⇒ real ⇒ real*
 where *truncate-down prec x = round-down (prec - ⌊log 2 |x|⌋) x*

lemma *truncate-down: truncate-down prec x ≤ x*
 using *round-down* by (simp add: truncate-down-def)

lemma *truncate-down-le: x ≤ y ⇒ truncate-down prec x ≤ y*
 by (rule order-trans[OF truncate-down])

lemma *truncate-down-zero[simp]: truncate-down prec 0 = 0*
 by (simp add: truncate-down-def)

lemma *truncate-down-float[simp]: truncate-down p x ∈ float*
 by (auto simp: truncate-down-def)

definition *truncate-up*::*nat* \Rightarrow *real* \Rightarrow *real*
where *truncate-up* *prec* *x* = *round-up* (*prec* - $\lfloor \log 2 \mid x \rfloor$) *x*

lemma *truncate-up*: $x \leq \text{truncate-up } \text{prec } x$
using *round-up* **by** (*simp* *add*: *truncate-up-def*)

lemma *truncate-up-le*: $x \leq y \implies x \leq \text{truncate-up } \text{prec } y$
by (*rule* *order-trans*[*OF* - *truncate-up*])

lemma *truncate-up-zero*[*simp*]: *truncate-up* *prec* 0 = 0
by (*simp* *add*: *truncate-up-def*)

lemma *truncate-up-uminus-eq*: *truncate-up* *prec* (-*x*) = - *truncate-down* *prec* *x*
and *truncate-down-uminus-eq*: *truncate-down* *prec* (-*x*) = - *truncate-up* *prec* *x*
by (*auto* *simp*: *truncate-up-def* *round-up-def* *truncate-down-def* *round-down-def* *ceiling-def*)

lemma *truncate-up-float*[*simp*]: *truncate-up* *p* *x* \in *float*
by (*auto* *simp*: *truncate-up-def*)

lemma *mult-powr-eq*: $0 < b \implies b \neq 1 \implies 0 < x \implies x * b^{\text{powr } y} = b^{\text{powr } (y + \log b \ x)}$
by (*simp*-*all* *add*: *powr-add*)

lemma *truncate-down-pos*:
assumes $x > 0$
shows *truncate-down* *p* *x* > 0
proof -
have $0 \leq \log 2 \ x - \text{real-of-int } \lfloor \log 2 \ x \rfloor$
by (*simp* *add*: *algebra-simps*)
moreover **have** $0 \leq \text{real } p - \text{real-of-int } \lfloor \log 2 \ x \rfloor + \log 2 \ x$
by *linarith*
ultimately **show** ?*thesis*
using *assms*
by (*auto* *simp*: *truncate-down-def* *round-down-def* *mult-powr-eq* *intro!*: *ge-one-powr-ge-zero* *mult-pos-pos*)
qed

lemma *truncate-down-nonneg*: $0 \leq y \implies 0 \leq \text{truncate-down } \text{prec } y$
by (*auto* *simp*: *truncate-down-def* *round-down-def*)

lemma *truncate-down-ge1*: $1 \leq x \implies 1 \leq \text{truncate-down } p \ x$
apply (*auto* *simp*: *truncate-down-def* *algebra-simps* *intro!*: *round-down-ge1*)
apply *linarith*
done

lemma *truncate-up-nonpos*: $x \leq 0 \implies \text{truncate-up } \text{prec } x \leq 0$
by (*auto* *simp*: *truncate-up-def* *round-up-def* *intro!*: *mult-nonpos-nonneg*)

```

lemma truncate-up-le1:
  assumes  $x \leq 1$ 
  shows truncate-up  $p$   $x \leq 1$ 
proof –
  consider  $x \leq 0 \mid x > 0$ 
    by arith
  then show ?thesis
proof cases
  case 1
    with truncate-up-nonpos[OF this, of p] show ?thesis
    by simp
  next
  case 2
    then have  $le: \lfloor \log 2 \mid x \rfloor \leq 0$ 
      using assms by (auto simp: log-less-iff)
    from assms have  $0 \leq \text{int } p$  by simp
    from add-mono[OF this le]
    show ?thesis
    using assms by (simp add: truncate-up-def round-up-le1 add-mono)
qed
qed

```

```

lemma truncate-down-shift-int:
  truncate-down  $p$  ( $x * 2^{\text{powr real-of-int } k}$ ) = truncate-down  $p$   $x * 2^{\text{powr } k}$ 
  by (cases  $x = 0$ )
  (simp-all add: algebra-simps abs-mult log-mult truncate-down-def
    round-down-shift[of - - k, simplified])

```

```

lemma truncate-down-shift-nat: truncate-down  $p$  ( $x * 2^{\text{powr real } k}$ ) = truncate-down  $p$   $x * 2^{\text{powr } k}$ 
  by (metis of-int-of-nat-eq truncate-down-shift-int)

```

```

lemma truncate-up-shift-int: truncate-up  $p$  ( $x * 2^{\text{powr real-of-int } k}$ ) = truncate-up
 $p$   $x * 2^{\text{powr } k}$ 
  by (cases  $x = 0$ )
  (simp-all add: algebra-simps abs-mult log-mult truncate-up-def
    round-up-shift[of - - k, simplified])

```

```

lemma truncate-up-shift-nat: truncate-up  $p$  ( $x * 2^{\text{powr real } k}$ ) = truncate-up  $p$   $x * 2^{\text{powr } k}$ 
  by (metis of-int-of-nat-eq truncate-up-shift-int)

```

43.10 Truncating Floats

```

lift-definition float-round-up :: nat  $\Rightarrow$  float  $\Rightarrow$  float is truncate-up
  by (simp add: truncate-up-def)

```

```

lemma float-round-up: real-of-float  $x \leq \text{real-of-float } (\text{float-round-up } \text{prec } x)$ 
  using truncate-up by transfer simp

```


lemma *float-round-up-zero*[simp]: *float-round-up prec 0 = 0*
by *transfer simp*

lift-definition *float-round-down* :: *nat* \Rightarrow *float* \Rightarrow *float* **is** *truncate-down*
by (*simp add: truncate-down-def*)

lemma *float-round-down*: *real-of-float (float-round-down prec x) \leq real-of-float x*
using *truncate-down* **by** *transfer simp*

lemma *float-round-down-zero*[simp]: *float-round-down prec 0 = 0*
by *transfer simp*

lemmas *float-round-up-le* = *order-trans*[*OF - float-round-up*]
and *float-round-down-le* = *order-trans*[*OF float-round-down*]

lemma *minus-float-round-up-eq*: *- float-round-up prec x = float-round-down prec (- x)*
and *minus-float-round-down-eq*: *- float-round-down prec x = float-round-up prec (- x)*
by (*transfer; simp add: truncate-down-uminus-eq truncate-up-uminus-eq*)**+**

context
begin

qualified lemma *compute-float-round-down*[code]:
float-round-down prec (Float m e) =
 (*let d = bitlen |m| - int prec - 1 in*
 if 0 < d then Float (div-twopow m (nat d)) (e + d)
 else Float m e)
using *Float.compute-float-down*[*of Suc prec - bitlen |m| - e m e, symmetric*]
by *transfer*
 (*simp add: field-simps abs-mult log-mult bitlen-alt-def truncate-down-def*
 cong del: if-weak-cong)

qualified lemma *compute-float-round-up*[code]:
float-round-up prec x = - float-round-down prec (-x)
by *transfer (simp add: truncate-down-uminus-eq)*

end

lemma *truncate-up-nonneg-mono*:
assumes *0 \leq x x \leq y*
shows *truncate-up prec x \leq truncate-up prec y*
proof –
consider [*log 2 x*] = [*log 2 y*] | [*log 2 x*] \neq [*log 2 y*] *0 < x | x \leq 0*
by *arith*
then show *?thesis*
proof *cases*

```

case 1
then show ?thesis
  using assms
  by (auto simp: truncate-up-def round-up-def intro!: ceiling-mono)
next
case 2
from assms  $\langle 0 < x \rangle$  have  $\log 2\ x \leq \log 2\ y$ 
  by auto
with  $\langle \lfloor \log 2\ x \rfloor \neq \lfloor \log 2\ y \rfloor \rangle$ 
have logless:  $\log 2\ x < \log 2\ y$ 
  by linarith
have flogless:  $\lfloor \log 2\ x \rfloor < \lfloor \log 2\ y \rfloor$ 
  using  $\langle \lfloor \log 2\ x \rfloor \neq \lfloor \log 2\ y \rfloor \rangle \langle \log 2\ x \leq \log 2\ y \rangle$  by linarith
have truncate-up prec  $x =$ 
   $\text{real-of-int } [x * 2^{\text{powr real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ x \rfloor)}] * 2^{\text{powr} - \text{real-of-int}}$ 
 $(\text{int } \text{prec} - \lfloor \log 2\ x \rfloor)$ 
  using assms by (simp add: truncate-up-def round-up-def)
also have  $[x * 2^{\text{powr real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ x \rfloor)}] \leq (2^{\wedge (\text{Suc } \text{prec})})$ 
proof (simp only: ceiling-le-iff)
  have  $x * 2^{\text{powr real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ x \rfloor)} \leq$ 
 $x * (2^{\text{powr real } (\text{Suc } \text{prec})} / (2^{\text{powr } \log 2\ x}))$ 
  using real-of-int-floor-add-one-ge[of  $\log 2\ x$ ] assms
  by (auto simp: algebra-simps simp flip: powr-diff intro!: mult-left-mono)
  then show  $x * 2^{\text{powr real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ x \rfloor)} \leq \text{real-of-int } ((2^{\text{int}})^{\wedge (\text{Suc } \text{prec})})$ 
  using  $\langle 0 < x \rangle$  by (simp add: powr-realpow powr-add)
qed
then have  $\text{real-of-int } [x * 2^{\text{powr real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ x \rfloor)}] \leq 2^{\text{powr}}$ 
 $\text{int } (\text{Suc } \text{prec})$ 
  by (auto simp: powr-realpow powr-add)
  (metis power-Suc of-int-le-numeral-power-cancel-iff)
also
  have  $2^{\text{powr} - \text{real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ x \rfloor)} \leq 2^{\text{powr} - \text{real-of-int } (\text{int}}$ 
 $\text{prec} - \lfloor \log 2\ y \rfloor + 1)$ 
  using logless flogless by (auto intro!: floor-mono)
also have  $2^{\text{powr real-of-int } (\text{int } (\text{Suc } \text{prec}))} \leq$ 
 $2^{\text{powr } (\log 2\ y + \text{real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ y \rfloor + 1))}$ 
  using assms  $\langle 0 < x \rangle$ 
  by (auto simp: algebra-simps)
finally have truncate-up prec  $x \leq$ 
 $2^{\text{powr } (\log 2\ y + \text{real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ y \rfloor + 1))} * 2^{\text{powr} - \text{real-of-int}}$ 
 $(\text{int } \text{prec} - \lfloor \log 2\ y \rfloor + 1)$ 
  by simp
also have  $\dots = 2^{\text{powr } (\log 2\ y + \text{real-of-int } (\text{int } \text{prec} - \lfloor \log 2\ y \rfloor) - \text{real-of-int}$ 
 $(\text{int } \text{prec} - \lfloor \log 2\ y \rfloor))}$ 
  by (subst powr-add[symmetric]) simp
also have  $\dots = y$ 
  using  $\langle 0 < x \rangle$  assms
  by (simp add: powr-add)

```

```

    also have ... ≤ truncate-up prec y
    by (rule truncate-up)
    finally show ?thesis .
next
case 3
then show ?thesis
using assms
by (auto intro!: truncate-up-le)
qed
qed

```

```

lemma truncate-up-switch-sign-mono:
  assumes  $x \leq 0$   $0 \leq y$ 
  shows  $\text{truncate-up prec } x \leq \text{truncate-up prec } y$ 
proof -
  note truncate-up-nonpos[OF ‹ $x \leq 0$ ›]
  also note truncate-up-le[OF ‹ $0 \leq y$ ›]
  finally show ?thesis .
qed

```

```

lemma truncate-down-switch-sign-mono:
  assumes  $x \leq 0$ 
  and  $0 \leq y$ 
  and  $x \leq y$ 
  shows  $\text{truncate-down prec } x \leq \text{truncate-down prec } y$ 
proof -
  note truncate-down-le[OF ‹ $x \leq 0$ ›]
  also note truncate-down-nonneg[OF ‹ $0 \leq y$ ›]
  finally show ?thesis .
qed

```

```

lemma truncate-down-nonneg-mono:
  assumes  $0 \leq x$   $x \leq y$ 
  shows  $\text{truncate-down prec } x \leq \text{truncate-down prec } y$ 
proof -
  consider  $x \leq 0$  |  $\lfloor \log 2 |x| \rfloor = \lfloor \log 2 |y| \rfloor$  |
     $0 < x$   $\lfloor \log 2 |x| \rfloor \neq \lfloor \log 2 |y| \rfloor$ 
  by arith
  then show ?thesis
  proof cases
    case 1
    with assms have  $x = 0$   $0 \leq y$  by simp-all
    then show ?thesis
    by (auto intro!: truncate-down-nonneg)
  next
    case 2
    then show ?thesis
    using assms
    by (auto simp: truncate-down-def round-down-def intro!: floor-mono)
  next
    case 3
    then show ?thesis
    using assms
    by (auto simp: truncate-down-def round-down-def intro!: floor-mono)
  qed
qed

```

```

next
case 3
from ⟨0 < x⟩ have log 2 x ≤ log 2 y 0 < y 0 ≤ y
  using assms by auto
with ⟨⌊log 2 |x|⌋ ≠ ⌊log 2 |y|⌋⟩
have logless: log 2 x < log 2 y and flogless: ⌊log 2 x⌋ < ⌊log 2 y⌋
  unfolding atomize-conj abs-of-pos[OF ⟨0 < x⟩] abs-of-pos[OF ⟨0 < y⟩]
  by (metis floor-less-cancel linorder-cases not-le)
have 2 powr prec ≤ y * 2 powr real prec / (2 powr log 2 y)
  using ⟨0 < y⟩ by simp
also have ... ≤ y * 2 powr real (Suc prec) / (2 powr (real-of-int ⌊log 2 y⌋ +
1))
  using ⟨0 ≤ y⟩ ⟨0 ≤ x⟩ assms(2)
  by (auto intro!: powr-mono divide-left-mono
    simp: of-nat-diff powr-add powr-diff)
also have ... = y * 2 powr real (Suc prec) / (2 powr real-of-int ⌊log 2 y⌋ * 2)
  by (auto simp: powr-add)
finally have (2 ^ prec) ≤ ⌊y * 2 powr real-of-int (int (Suc prec) - ⌊log 2 |y|⌋
- 1)⌋
  using ⟨0 ≤ y⟩
  by (auto simp: powr-diff le-floor-iff powr-realpow powr-add)
then have (2 ^ (prec)) * 2 powr - real-of-int (int prec - ⌊log 2 |y|⌋) ≤
truncate-down prec y
  by (auto simp: truncate-down-def round-down-def)
moreover have x ≤ (2 ^ prec) * 2 powr - real-of-int (int prec - ⌊log 2 |y|⌋)
proof -
  have x = 2 powr (log 2 |x|) using ⟨0 < x⟩ by simp
  also have ... ≤ (2 ^ (Suc prec)) * 2 powr - real-of-int (int prec - ⌊log 2
|x|⌋)
    using real-of-int-floor-add-one-ge[of log 2 |x|] ⟨0 < x⟩
    by (auto simp flip: powr-realpow powr-add simp: algebra-simps powr-mult-base
le-powr-iff)
  also
  have 2 powr - real-of-int (int prec - ⌊log 2 |x|⌋) ≤ 2 powr - real-of-int (int
prec - ⌊log 2 |y|⌋ + 1)
    using logless flogless ⟨x > 0⟩ ⟨y > 0⟩
    by (auto intro!: floor-mono)
  finally show ?thesis
    by (auto simp flip: powr-realpow simp: powr-diff assms)
qed
ultimately show ?thesis
  by (metis dual-order.trans truncate-down)
qed
qed

lemma truncate-down-eq-truncate-up: truncate-down p x = - truncate-up p (-x)
and truncate-up-eq-truncate-down: truncate-up p x = - truncate-down p (-x)
by (auto simp: truncate-up-uminus-eq truncate-down-uminus-eq)

```

lemma *truncate-down-mono*: $x \leq y \implies \text{truncate-down } p \ x \leq \text{truncate-down } p \ y$
by (*smt* (*verit*) *truncate-down-nonneg-mono truncate-up-nonneg-mono truncate-up-uminus-eq*)

lemma *truncate-up-mono*: $x \leq y \implies \text{truncate-up } p \ x \leq \text{truncate-up } p \ y$
by (*simp* *add*: *truncate-up-eq-truncate-down truncate-down-mono*)

lemma *truncate-up-nonneg*: $0 \leq \text{truncate-up } p \ x$ **if** $0 \leq x$
by (*simp* *add*: *that truncate-up-le*)

lemma *truncate-up-pos*: $0 < \text{truncate-up } p \ x$ **if** $0 < x$
by (*meson* *less-le-trans that truncate-up*)

lemma *truncate-up-less-zero-iff*[*simp*]: $\text{truncate-up } p \ x < 0 \longleftrightarrow x < 0$
by (*smt* (*verit*) *truncate-down-pos truncate-down-uminus-eq truncate-up-nonneg*)

lemma *truncate-up-nonneg-iff*[*simp*]: $\text{truncate-up } p \ x \geq 0 \longleftrightarrow x \geq 0$
using *truncate-up-less-zero-iff*[*of* $p \ x$] *truncate-up-nonneg*[*of* x]
by *linarith*

lemma *truncate-down-less-zero-iff*[*simp*]: $\text{truncate-down } p \ x < 0 \longleftrightarrow x < 0$
by (*metis* *le-less-trans not-less-iff-gr-or-eq truncate-down truncate-down-pos truncate-down-zero*)

lemma *truncate-down-nonneg-iff*[*simp*]: $\text{truncate-down } p \ x \geq 0 \longleftrightarrow x \geq 0$
using *truncate-down-less-zero-iff*[*of* $p \ x$] *truncate-down-nonneg*[*of* $x \ p$]
by *linarith*

lemma *truncate-down-eq-zero-iff*[*simp*]: $\text{truncate-down } p \ x = 0 \longleftrightarrow x = 0$
by (*metis* *not-less-iff-gr-or-eq truncate-down-less-zero-iff truncate-down-pos truncate-down-zero*)

lemma *truncate-up-eq-zero-iff*[*simp*]: $\text{truncate-up } p \ x = 0 \longleftrightarrow x = 0$
by (*metis* *not-less-iff-gr-or-eq truncate-up-less-zero-iff truncate-up-pos truncate-up-zero*)

43.11 Approximation of positive rationals

lemma *div-mult-twopow-eq*: $a \text{ div } ((2 :: \text{nat}) \wedge n) \text{ div } b = a \text{ div } (b * 2 \wedge n)$ **for** $a \ b :: \text{nat}$
by (*cases* $b = 0$) (*simp-all* *add*: *div-mult2-eq[symmetric] ac-simps*)

lemma *real-div-nat-eq-floor-of-divide*: $a \text{ div } b = \text{real-of-int } \lfloor a / b \rfloor$ **for** $a \ b :: \text{nat}$
by (*simp* *add*: *floor-divide-of-nat-eq* [*of* $a \ b$])

definition *rat-precision* $\text{prec } x \ y =$
 (*let* $d = \text{bitlen } x - \text{bitlen } y$
in $\text{int } \text{prec} - d + (\text{if } \text{Float } (\text{abs } x) \ 0 < \text{Float } (\text{abs } y) \ d \text{ then } 1 \text{ else } 0))$

lemma *floor-log-divide-eq*:
assumes $i > 0 \ j > 0 \ p > 1$

shows $\lfloor \log p (i / j) \rfloor = \text{floor} (\log p i) - \text{floor} (\log p j) -$
 $(\text{if } i \geq j * p^{\text{powr}} (\text{floor} (\log p i) - \text{floor} (\log p j)) \text{ then } 0 \text{ else } 1)$

proof –

let $?l = \log p$
let $?fl = \lambda x. \text{floor} (?l x)$
have $\lfloor ?l (i / j) \rfloor = \lfloor ?l i - ?l j \rfloor$ **using** *assms*
by (*auto simp: log-divide*)
also have $\dots = \text{floor} (\text{real-of-int} (?fl i - ?fl j) + (?l i - ?fl i - (?l j - ?fl j)))$
 $(\text{is } - = \text{floor} (- + ?r))$
by (*simp add: algebra-simps*)
also note *floor-add2*
also note $\langle p > 1 \rangle$
note *powr = powr-le-cancel-iff[symmetric, OF $\langle 1 < p \rangle$, THEN iffD2]*
note *powr-strict = powr-less-cancel-iff[symmetric, OF $\langle 1 < p \rangle$, THEN iffD2]*
have $\text{floor } ?r = (\text{if } i \geq j * p^{\text{powr}} (?fl i - ?fl j) \text{ then } 0 \text{ else } -1)$ **(is** $- = ?if$ **)**
using *assms*
apply *simp*
by (*smt (verit, ccfv-SIG) floor-less-iff floor-uminus-of-int le-log-iff mult-powr-eq*
of-int-1 real-of-int-floor-add-one-gt zero-le-floor)
finally
show *?thesis* **by** *simp*

qed

lemma *truncate-down-rat-precision:*

$\text{truncate-down prec (real } x / \text{ real } y) = \text{round-down (rat-precision prec } x \ y) \text{ (real } x / \text{ real } y)$

and *truncate-up-rat-precision:*

$\text{truncate-up prec (real } x / \text{ real } y) = \text{round-up (rat-precision prec } x \ y) \text{ (real } x / \text{ real } y)$

unfolding *truncate-down-def truncate-up-def rat-precision-def*

by (*cases x; cases y*) (*auto simp: floor-log-divide-eq algebra-simps bitlen-alt-def*)

lift-definition *lapprox-posrat :: nat \Rightarrow nat \Rightarrow nat \Rightarrow float*

is $\lambda \text{prec } (x::\text{nat}) \ (y::\text{nat}). \text{truncate-down prec } (x / y)$

by *simp*

context

begin

qualified lemma *compute-lapprox-posrat[code]:*

$\text{lapprox-posrat prec } x \ y =$

(*let*

$l = \text{rat-precision prec } x \ y;$

$d = \text{if } 0 \leq l \text{ then } x * 2^{\text{nat } l} \text{ div } y \text{ else } x \text{ div } 2^{\text{nat } (- l)} \text{ div } y$

in normfloat (Float d (- l)))

unfolding *div-mult-twopow-eq*

by *transfer*

(*simp add: round-down-def powr-int real-div-nat-eq-floor-of-divide field-simps*

Let-def

```

    truncate-down-rat-precision del: two-powr-minus-int-float)

end

lift-definition rapprox-posrat :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  float
  is  $\lambda prec (x::nat) (y::nat).$  truncate-up prec (x / y)
  by simp

context
begin

qualified lemma compute-rapprox-posrat[code]:
  fixes prec x y
  defines l  $\equiv$  rat-precision prec x y
  shows rapprox-posrat prec x y =
    (let
      l = l;
      (r, s) = if 0  $\leq$  l then (x * 2nat l, y) else (x, y * 2nat(-l));
      d = r div s;
      m = r mod s
      in normfloat (Float (d + (if m = 0  $\vee$  y = 0 then 0 else 1)) (- l)))
proof (cases y = 0)
  assume y = 0
  then show ?thesis by transfer simp
next
  assume y  $\neq$  0
  show ?thesis
  proof (cases 0  $\leq$  l)
    case True
    define x' where x' = x * 2nat l
    have int x * 2nat l = x'
    by (simp add: x'-def)
    moreover have real x * 2powr l = real x'
    by (simp flip: powr-realpow add:  $\langle 0 \leq l \rangle$  x'-def)
    ultimately show ?thesis
    using ceil-divide-floor-conv[of y x'] powr-realpow[of 2 nat l]  $\langle 0 \leq l \rangle \langle y \neq 0 \rangle$ 
      l-def[symmetric, THEN meta-eq-to-obj-eq]
    apply transfer
    apply (auto simp add: round-up-def truncate-up-rat-precision)
    apply (metis floor-divide-of-int-eq of-int-of-nat-eq)
    done
  next
  case False
  define y' where y' = y * 2nat(-l)
  from  $\langle y \neq 0 \rangle$  have y'  $\neq$  0 by (simp add: y'-def)
  have int y * 2nat(-l) = y'
  by (simp add: y'-def)
  moreover have real x * real-of-int (2::int) powr real-of-int l / real y = x /
    real y'

```

```

    using  $\langle \neg 0 \leq l \rangle$  by (simp flip: powr-realpow add: powr-minus y'-def field-simps)
  ultimately show ?thesis
    using ceil-divide-floor-conv[of y' x]  $\langle \neg 0 \leq l \rangle$   $\langle y' \neq 0 \rangle$   $\langle y \neq 0 \rangle$ 
      l-def[symmetric, THEN meta-eq-to-obj-eq]
    apply transfer
  apply (auto simp add: round-up-def ceil-divide-floor-conv truncate-up-rat-precision)
  apply (metis floor-divide-of-int-eq of-int-of-nat-eq)
  done
qed
qed
end

```

```

lemma rat-precision-pos:
  assumes  $0 \leq x$ 
    and  $0 < y$ 
    and  $2 * x < y$ 
  shows rat-precision n (int x) (int y) > 0
proof -
  have  $0 < x \implies \log 2 x + 1 = \log 2 (2 * x)$ 
    by (simp add: log-mult)
  then have bitlen (int x) < bitlen (int y)
    using assms
    by (simp add: bitlen-alt-def)
    (auto intro!: floor-mono simp add: one-add-floor)
  then show ?thesis
    using assms
    by (auto intro!: pos-add-strict simp add: field-simps rat-precision-def)
qed

```

```

lemma rapprox-posrat-less1:
   $0 \leq x \implies 0 < y \implies 2 * x < y \implies \text{real-of-float } (\text{rapprox-posrat } n \ x \ y) < 1$ 
  by transfer (simp add: rat-precision-pos round-up-less1 truncate-up-rat-precision)

```

```

lift-definition lapprox-rat :: nat  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  float is
   $\lambda \text{prec } (x::\text{int}) \ (y::\text{int}). \text{truncate-down prec } (x / y)$ 
  by simp

```

```

context
begin

```

```

qualified lemma compute-lapprox-rat[code]:
  lapprox-rat prec x y =
    (if y = 0 then 0
     else if  $0 \leq x$  then
       (if  $0 < y$  then lapprox-posrat prec (nat x) (nat y)
        else - (rapprox-posrat prec (nat x) (nat (-y))))
     else
       (if  $0 < y$ 

```



```

      then - (rapprox-posrat prec (nat (-x)) (nat y))
      else lapprox-posrat prec (nat (-x)) (nat (-y)))
  by transfer (simp add: truncate-up-uminus-eq)

lift-definition rapprox-rat :: nat  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  float is
   $\lambda$ prec (x::int) (y::int). truncate-up prec (x / y)
  by simp

lemma rapprox-rat = rapprox-posrat
  by transfer auto

lemma lapprox-rat = lapprox-posrat
  by transfer auto

qualified lemma compute-rapprox-rat[code]:
  rapprox-rat prec x y = - lapprox-rat prec (-x) y
  by transfer (simp add: truncate-down-uminus-eq)

qualified lemma compute-truncate-down[code]:
  truncate-down p (Ratreal r) = (let (a, b) = quotient-of r in lapprox-rat p a b)
  by transfer (auto split: prod.split simp: of-rat-divide dest!: quotient-of-div)

qualified lemma compute-truncate-up[code]:
  truncate-up p (Ratreal r) = (let (a, b) = quotient-of r in rapprox-rat p a b)
  by transfer (auto split: prod.split simp: of-rat-divide dest!: quotient-of-div)

end

43.12 Division

definition real-divl prec a b = truncate-down prec (a / b)

definition real-divr prec a b = truncate-up prec (a / b)

lift-definition float-divl :: nat  $\Rightarrow$  float  $\Rightarrow$  float  $\Rightarrow$  float is real-divl
  by (simp add: real-divl-def)

context
begin

qualified lemma compute-float-divl[code]:
  float-divl prec (Float m1 s1) (Float m2 s2) = lapprox-rat prec m1 m2 * Float 1
  (s1 - s2)
  apply transfer
  unfolding real-divl-def of-int-1 mult-1 truncate-down-shift-int[symmetric]
  apply (simp add: powr-diff powr-minus)
  done

lift-definition float-divr :: nat  $\Rightarrow$  float  $\Rightarrow$  float  $\Rightarrow$  float is real-divr

```

by (*simp add: real-divr-def*)

qualified lemma *compute-float-divr*[*code*]:

float-divr prec x y = - float-divl prec (-x) y

by *transfer (simp add: real-divr-def real-divl-def truncate-down-uminus-eq)*

end

43.13 Approximate Addition

definition *plus-down prec x y = truncate-down prec (x + y)*

definition *plus-up prec x y = truncate-up prec (x + y)*

lemma *float-plus-down-float*[*intro, simp*]: $x \in \text{float} \implies y \in \text{float} \implies \text{plus-down } p \ x \ y \in \text{float}$

by (*simp add: plus-down-def*)

lemma *float-plus-up-float*[*intro, simp*]: $x \in \text{float} \implies y \in \text{float} \implies \text{plus-up } p \ x \ y \in \text{float}$

by (*simp add: plus-up-def*)

lift-definition *float-plus-down* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *plus-down* ..

lift-definition *float-plus-up* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{float} \Rightarrow \text{float}$ **is** *plus-up* ..

lemma *plus-down*: $\text{plus-down } prec \ x \ y \leq x + y$

and *plus-up*: $x + y \leq \text{plus-up } prec \ x \ y$

by (*auto simp: plus-down-def truncate-down plus-up-def truncate-up*)

lemma *float-plus-down*: $\text{real-of-float } (\text{float-plus-down } prec \ x \ y) \leq x + y$

and *float-plus-up*: $x + y \leq \text{real-of-float } (\text{float-plus-up } prec \ x \ y)$

by (*transfer; rule plus-down plus-up*)**+**

lemmas *plus-down-le* = *order-trans*[*OF plus-down*]

and *plus-up-le* = *order-trans*[*OF - plus-up*]

and *float-plus-down-le* = *order-trans*[*OF float-plus-down*]

and *float-plus-up-le* = *order-trans*[*OF - float-plus-up*]

lemma *compute-plus-up*[*code*]: $\text{plus-up } p \ x \ y = - \text{plus-down } p \ (-x) \ (-y)$

using *truncate-down-uminus-eq*[*of p x + y*]

by (*auto simp: plus-down-def plus-up-def*)

lemma *truncate-down-log2-eqI*:

assumes $\lfloor \log 2 \ |x| \rfloor = \lfloor \log 2 \ |y| \rfloor$

assumes $\lfloor x * 2^{\text{powr } (p - \lfloor \log 2 \ |x| \rfloor)} \rfloor = \lfloor y * 2^{\text{powr } (p - \lfloor \log 2 \ |x| \rfloor)} \rfloor$

shows $\text{truncate-down } p \ x = \text{truncate-down } p \ y$

using *assms* **by** (*auto simp: truncate-down-def round-down-def*)

lemma *sum-neq-zeroI*:

$|a| \geq k \implies |b| < k \implies a + b \neq 0$

$|a| > k \implies |b| \leq k \implies a + b \neq 0$

for $a\ k :: \text{real}$

by *auto*

lemma *abs-real-le-2-powr-bitlen[simp]*: $|\text{real-of-int } m2| < 2^{\text{powr real-of-int (bitlen } |m2|)}}$

proof (*cases* $m2 = 0$)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

then have $|m2| < 2^{\text{nat (bitlen } |m2|)}}$

using *bitlen-bounds[of |m2|]*

by (*auto simp: powr-add bitlen-nonneg*)

then show *?thesis*

by (*metis bitlen-nonneg powr-int of-int-abs of-int-less-numeral-power-cancel-iff zero-less-numeral*)

qed

lemma *floor-sum-times-2-powr-sgn-eq*:

fixes $ai\ p\ q :: \text{int}$

and $a\ b :: \text{real}$

assumes $a * 2^{\text{powr } p} = ai$

and *b-le-1*: $|b * 2^{\text{powr } (p + 1)}| \leq 1$

and *leqp*: $q \leq p$

shows $\lfloor (a + b) * 2^{\text{powr } q} \rfloor = \lfloor (2 * ai + \text{sgn } b) * 2^{\text{powr } (q - p - 1)} \rfloor$

proof –

consider $b = 0 \mid b > 0 \mid b < 0$ **by** *arith*

then show *?thesis*

proof *cases*

case *1*

then show *?thesis*

by (*simp flip: assms(1) powr-add add: algebra-simps powr-mult-base*)

next

case *2*

then have $b * 2^{\text{powr } p} < |b * 2^{\text{powr } (p + 1)}|$

by *simp*

also note *b-le-1*

finally have *b-less-1*: $b * 2^{\text{powr real-of-int } p} < 1$.

from *b-less-1* $\langle b > 0 \rangle$ **have** *floor-eq*: $\lfloor b * 2^{\text{powr real-of-int } p} \rfloor = 0 \lfloor \text{sgn } b / 2 \rfloor = 0$

by (*simp-all add: floor-eq-iff*)

have $\lfloor (a + b) * 2^{\text{powr } q} \rfloor = \lfloor (a + b) * 2^{\text{powr } p} * 2^{\text{powr } (q - p)} \rfloor$

by (*simp add: algebra-simps flip: powr-realpow powr-add*)

also have $\dots = \lfloor (ai + b * 2^{\text{powr } p}) * 2^{\text{powr } (q - p)} \rfloor$

```

    by (simp add: assms algebra-simps)
  also have ... =  $\lfloor (ai + b * 2^{\text{powr } p}) / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q)}) \rfloor$ 
    using assms
    by (simp add: algebra-simps divide-powr-uminus flip: powr-realpow powr-add)
  also have ... =  $\lfloor ai / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q)}) \rfloor$ 
    by (simp del: of-int-power add: floor-divide-real-eq-div floor-eq)
  finally have  $\lfloor (a + b) * 2^{\text{powr } \text{real-of-int } q} \rfloor = \lfloor \text{real-of-int } ai / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q)}) \rfloor$  .
  moreover
  have  $\lfloor (2 * ai + (\text{sgn } b)) * 2^{\text{powr } (\text{real-of-int } (q - p) - 1)} \rfloor =$ 
     $\lfloor \text{real-of-int } ai / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q)}) \rfloor$ 
  proof -
    have  $\lfloor (2 * ai + \text{sgn } b) * 2^{\text{powr } (\text{real-of-int } (q - p) - 1)} \rfloor = \lfloor (ai + \text{sgn } b /$ 
     $2) * 2^{\text{powr } (q - p)} \rfloor$ 
    by (subst powr-diff) (simp add: field-simps)
    also have ... =  $\lfloor (ai + \text{sgn } b / 2) / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q)}) \rfloor$ 
    using leqp by (simp flip: powr-realpow add: powr-diff)
    also have ... =  $\lfloor ai / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q)}) \rfloor$ 
    by (simp del: of-int-power add: floor-divide-real-eq-div floor-eq)
    finally show ?thesis .
  qed
  ultimately show ?thesis by simp
next
case 3
then have floor-eq:  $\lfloor b * 2^{\text{powr } (\text{real-of-int } p + 1)} \rfloor = -1$ 
  using b-le-1
  by (auto simp: floor-eq-iff algebra-simps pos-divide-le-eq[symmetric] abs-if
  divide-powr-uminus
  intro!: mult-neg-pos split: if-split-asm)
  have  $\lfloor (a + b) * 2^{\text{powr } q} \rfloor = \lfloor (2*a + 2*b) * 2^{\text{powr } p} * 2^{\text{powr } (q - p - 1)} \rfloor$ 
    by (simp add: algebra-simps powr-mult-base flip: powr-realpow powr-add)
  also have ... =  $\lfloor (2 * (a * 2^{\text{powr } p}) + 2 * b * 2^{\text{powr } p}) * 2^{\text{powr } (q - p - 1)} \rfloor$ 
    by (simp add: algebra-simps)
  also have ... =  $\lfloor (2 * ai + b * 2^{\text{powr } (p + 1)}) / 2^{\text{powr } (1 - q + p)} \rfloor$ 
    using assms by (simp add: algebra-simps powr-mult-base divide-powr-uminus)
  also have ... =  $\lfloor (2 * ai + b * 2^{\text{powr } (p + 1)}) / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q + 1)}) \rfloor$ 
    using assms by (simp add: algebra-simps flip: powr-realpow)
  also have ... =  $\lfloor (2 * ai - 1) / \text{real-of-int } ((2::\text{int})^{\wedge \text{nat } (p - q + 1)}) \rfloor$ 
    using  $\langle b < 0 \rangle$  assms
    by (simp add: floor-divide-of-int-eq floor-eq floor-divide-real-eq-div
    del: of-int-mult of-int-power of-int-diff)
  also have ... =  $\lfloor (2 * ai - 1) * 2^{\text{powr } (q - p - 1)} \rfloor$ 
    using assms by (simp add: algebra-simps divide-powr-uminus flip: powr-realpow)
  finally show ?thesis
    using  $\langle b < 0 \rangle$  by simp
qed
qed

```

```

lemma log2-abs-int-add-less-half-sgn-eq:
  fixes ai :: int
    and b :: real
  assumes  $|b| \leq 1/2$ 
    and  $ai \neq 0$ 
  shows  $\lfloor \log 2 \mid \text{real-of-int } ai + b \rfloor = \lfloor \log 2 \mid ai + \text{sgn } b / 2 \rfloor$ 
proof (cases  $b = 0$ )
  case True
    then show ?thesis by simp
next
  case False
    define k where  $k = \lfloor \log 2 \mid ai \rfloor$ 
    then have  $\lfloor \log 2 \mid ai \rfloor = k$ 
      by simp
    then have  $k: 2^{\text{powr } k} \leq |ai| \mid |ai| < 2^{\text{powr } (k + 1)}$ 
      by (simp-all add: floor-log-eq-powr-iff  $\langle ai \neq 0 \rangle$ )
    have  $k \geq 0$ 
      using assms by (auto simp: k-def)
    define r where  $r = |ai| - 2^{\text{nat } k}$ 
    have  $r: 0 \leq r \mid r < 2^{\text{powr } k}$ 
      using  $\langle k \geq 0 \rangle$  k
      by (auto simp: r-def k-def algebra-simps powr-add abs-if powr-int)
    then have  $r \leq (2::\text{int})^{\text{nat } k} - 1$ 
      using  $\langle k \geq 0 \rangle$  by (auto simp: powr-int)
    from this [simplified of-int-le-iff [symmetric]]  $\langle 0 \leq k \rangle$ 
    have  $r\text{-le}: r \leq 2^{\text{powr } k} - 1$ 
      by (auto simp: algebra-simps powr-int)
      (metis of-int-1 of-int-add of-int-le-numeral-power-cancel-iff)

    have  $|ai| = 2^{\text{powr } k} + r$ 
      using  $\langle k \geq 0 \rangle$  by (auto simp: k-def r-def simp flip: powr-realpow)

    have pos:  $|b| < 1 \implies 0 < 2^{\text{powr } k} + (r + b)$  for  $b :: \text{real}$ 
      using  $\langle 0 \leq k \rangle \mid \langle ai \neq 0 \rangle$ 
      by (auto simp add: r-def powr-realpow [symmetric] abs-if sgn-if algebra-simps split: if-split-asm)
    have less:  $|\text{sgn } ai * b| < 1$ 
      and less':  $|\text{sgn } (\text{sgn } ai * b) / 2| < 1$ 
      using  $\langle |b| \leq \rightarrow \rangle$  by (auto simp: abs-if sgn-if split: if-split-asm)

    have floor-eq:  $\bigwedge b::\text{real}. |b| \leq 1 / 2 \implies$ 
       $\lfloor \log 2 \mid (1 + (r + b) / 2^{\text{powr } k}) \rfloor = (\text{if } r = 0 \wedge b < 0 \text{ then } -1 \text{ else } 0)$ 
      using  $\langle k \geq 0 \rangle$  r r-le
      by (auto simp: floor-log-eq-powr-iff powr-minus-divide field-simps sgn-if)

    from  $\langle \text{real-of-int } |ai| = \rightarrow \rangle$  have  $|ai + b| = 2^{\text{powr } k} + (r + \text{sgn } ai * b)$ 
      using  $\langle |b| \leq \rightarrow \rangle \mid \langle 0 \leq k \rangle$  r
      by (auto simp add: sgn-if abs-if)

```

```

also have  $\lfloor \log 2 \dots \rfloor = \lfloor \log 2 (2^{\text{powr } k} + r + \text{sgn } (\text{sgn } ai * b) / 2) \rfloor$ 
proof –
  have  $2^{\text{powr } k} + (r + (\text{sgn } ai) * b) = 2^{\text{powr } k} * (1 + (r + \text{sgn } ai * b) / 2^{\text{powr } k})$ 
  by (simp add: field-simps)
  also have  $\lfloor \log 2 \dots \rfloor = k + \lfloor \log 2 (1 + (r + \text{sgn } ai * b) / 2^{\text{powr } k}) \rfloor$ 
  using pos[OF less]
  by (subst log-mult) (simp-all add: log-mult powr-mult field-simps)
  also
  let ?if = if  $r = 0 \wedge \text{sgn } ai * b < 0$  then  $-1$  else  $0$ 
  have  $\lfloor \log 2 (1 + (r + \text{sgn } ai * b) / 2^{\text{powr } k}) \rfloor = ?if$ 
  using  $\langle |b| \leq - \rangle$ 
  by (intro floor-eq) (auto simp: abs-mult sgn-if)
  also
  have  $\dots = \lfloor \log 2 (1 + (r + \text{sgn } (\text{sgn } ai * b) / 2) / 2^{\text{powr } k}) \rfloor$ 
  by (subst floor-eq) (auto simp: sgn-if)
  also have  $k + \dots = \lfloor \log 2 (2^{\text{powr } k} * (1 + (r + \text{sgn } (\text{sgn } ai * b) / 2) / 2^{\text{powr } k})) \rfloor$ 
  unfolding int-add-floor
  using pos[OF less]  $\langle |b| \leq - \rangle$ 
  by (simp add: field-simps add-log-eq-powr del: floor-add2)
  also have  $2^{\text{powr } k} * (1 + (r + \text{sgn } (\text{sgn } ai * b) / 2) / 2^{\text{powr } k}) =$ 
     $2^{\text{powr } k} + r + \text{sgn } (\text{sgn } ai * b) / 2$ 
  by (simp add: sgn-if field-simps)
  finally show ?thesis .
qed
also have  $2^{\text{powr } k} + r + \text{sgn } (\text{sgn } ai * b) / 2 = |ai + \text{sgn } b / 2|$ 
  unfolding  $\langle \text{real-of-int } |ai| = - \rangle$  [symmetric] using  $\langle ai \neq 0 \rangle$ 
  by (auto simp: abs-if sgn-if algebra-simps)
  finally show ?thesis .
qed

```

context

begin

qualified lemma *compute-far-float-plus-down:*

```

fixes m1 e1 m2 e2 :: int
and p :: nat
defines k1  $\equiv \text{Suc } p - \text{nat } (\text{bitlen } |m1|)$ 
assumes H:  $\text{bitlen } |m2| \leq e1 - e2 - k1 - 2$   $m1 \neq 0$   $m2 \neq 0$   $e1 \geq e2$ 
shows float-plus-down p (Float m1 e1) (Float m2 e2) =
  float-round-down p (Float (m1 *  $2^{\wedge} (\text{Suc } (\text{Suc } k1))$ ) + sgn m2) (e1 - int k1 - 2))
proof –
  let ?a = real-of-float (Float m1 e1)
  let ?b = real-of-float (Float m2 e2)
  let ?sum = ?a + ?b
  let ?shift = real-of-int e2 - real-of-int e1 + real k1 + 1
  let ?m1 = m1 *  $2^{\wedge} \text{Suc } k1$ 

```

```

let ?m2 = m2 * 2 powr ?shift
let ?m2' = sgn m2 / 2
let ?e = e1 - int k1 - 1

have sum-eq: ?sum = (?m1 + ?m2) * 2 powr ?e
  by (auto simp flip: powr-add powr-mult powr-realpow simp: powr-mult-base
algebra-simps)

have |?m2| * 2 < 2 powr (bitlen |m2| + ?shift + 1)
  by (auto simp: field-simps powr-add powr-mult-base powr-diff abs-mult)
also have ... ≤ 2 powr 0
  using H by (intro powr-mono) auto
finally have abs-m2-less-half: |?m2| < 1 / 2
  by simp

then have |real-of-int m2| < 2 powr -(?shift + 1)
  unfolding powr-minus-divide by (auto simp: bitlen-alt-def field-simps powr-mult-base
abs-mult)
also have ... ≤ 2 powr real-of-int (e1 - e2 - 2)
  by simp
finally have b-less-quarter: |?b| < 1/4 * 2 powr real-of-int e1
  by (simp add: powr-add field-simps powr-diff abs-mult)
also have 1/4 < |real-of-int m1| / 2 using ⟨m1 ≠ 0⟩ by simp
finally have b-less-half-a: |?b| < 1/2 * |?a|
  by (simp add: algebra-simps powr-mult-base abs-mult)
then have a-half-less-sum: |?a| / 2 < |?sum|
  by (auto simp: field-simps abs-if split: if-split-asm)

from b-less-half-a have |?b| < |?a| |?b| ≤ |?a|
  by simp-all

have |real-of-float (Float m1 e1)| ≥ 1/4 * 2 powr real-of-int e1
  using ⟨m1 ≠ 0⟩
  by (auto simp: powr-add powr-int bitlen-nonneg divide-right-mono abs-mult)
then have ?sum ≠ 0 using b-less-quarter
  by (rule sum-neq-zeroI)
then have ?m1 + ?m2 ≠ 0
  unfolding sum-eq by (simp add: abs-mult zero-less-mult-iff)

have |real-of-int ?m1| ≥ 2 ^ Suc k1 |?m2'| < 2 ^ Suc k1
  using ⟨m1 ≠ 0⟩ ⟨m2 ≠ 0⟩ by (auto simp: sgn-if less-1-mult abs-mult simp del:
power.simps)
then have sum'-nz: ?m1 + ?m2' ≠ 0
  by (intro sum-neq-zeroI)

have ⌊log 2 |real-of-float (Float m1 e1) + real-of-float (Float m2 e2)|⌋ = ⌊log 2
|?m1 + ?m2|⌋ + ?e
  using ⟨?m1 + ?m2 ≠ 0⟩
  unfolding floor-add[symmetric] sum-eq

```

```

    by (simp add: abs-mult log-mult) linarith
    also have  $\lfloor \log 2 \mid ?m1 + ?m2 \rfloor = \lfloor \log 2 \mid ?m1 + \text{sgn}(\text{real-of-int } m2 * 2^{\text{powr } ?shift}) / 2 \rfloor$ 
    using abs-m2-less-half  $\langle m1 \neq 0 \rangle$ 
    by (intro log2-abs-int-add-less-half-sgn-eq) (auto simp: abs-mult)
    also have  $\text{sgn}(\text{real-of-int } m2 * 2^{\text{powr } ?shift}) = \text{sgn } m2$ 
    by (auto simp: sgn-if zero-less-mult-iff less-not-sym)
    also
    have  $\lfloor ?m1 + ?m2' \rfloor * 2^{\text{powr } ?e} = \lfloor ?m1 * 2 + \text{sgn } m2 \rfloor * 2^{\text{powr } (?e - 1)}$ 
    by (auto simp: field-simps powr-minus[symmetric] powr-diff powr-mult-base)
    then have  $\lfloor \log 2 \mid ?m1 + ?m2' \rfloor + ?e = \lfloor \log 2 \mid \text{real-of-float}(\text{Float} (?m1 * 2 + \text{sgn } m2) (?e - 1)) \rfloor$ 
    using  $\langle ?m1 + ?m2' \neq 0 \rangle$ 
    unfolding floor-add-int
    by (simp add: log-add-eq-powr abs-mult-pos del: floor-add2)
    finally
    have  $\lfloor \log 2 \mid ?sum \rfloor = \lfloor \log 2 \mid \text{real-of-float}(\text{Float} (?m1 * 2 + \text{sgn } m2) (?e - 1)) \rfloor$ 
    .
    then have plus-down p (Float m1 e1) (Float m2 e2) =
      truncate-down p (Float (?m1 * 2 + sgn m2) (?e - 1))
    unfolding plus-down-def
    proof (rule truncate-down-log2-eqI)
      let ?f = (int p -  $\lfloor \log 2 \mid \text{real-of-float}(\text{Float } m1 \text{ } e1) + \text{real-of-float}(\text{Float } m2 \text{ } e2) \rfloor$ )
      let ?ai =  $m1 * 2^{\wedge}(\text{Suc } k1)$ 
      have  $\lfloor (?a + ?b) * 2^{\text{powr } \text{real-of-int } ?f} \rfloor = \lfloor (\text{real-of-int } (2 * ?ai) + \text{sgn } ?b) * 2^{\text{powr } \text{real-of-int } (?f - ?e - 1)} \rfloor$ 
      proof (rule floor-sum-times-2-powr-sgn-eq)
        show  $?a * 2^{\text{powr } \text{real-of-int } (-?e)} = \text{real-of-int } ?ai$ 
        by (simp add: powr-add powr-realpow[symmetric] powr-diff)
        show  $\lfloor ?b * 2^{\text{powr } \text{real-of-int } (-?e + 1)} \rfloor \leq 1$ 
        using abs-m2-less-half
        by (simp add: abs-mult powr-add[symmetric] algebra-simps powr-mult-base)
      next
      have  $e1 + \lfloor \log 2 \mid \text{real-of-int } m1 \rfloor - 1 = \lfloor \log 2 \mid ?a \rfloor - 1$ 
      using  $\langle m1 \neq 0 \rangle$ 
      by (simp add: int-add-floor algebra-simps log-mult abs-mult del: floor-add2)
      also have  $\dots \leq \lfloor \log 2 \mid ?a + ?b \rfloor$ 
      using a-half-less-sum  $\langle m1 \neq 0 \rangle \langle ?sum \neq 0 \rangle$ 
      unfolding floor-diff-of-int[symmetric]
      by (auto simp add: log-minus-eq-powr powr-minus-divide intro!: floor-mono)
      finally
      have  $\text{int } p - \lfloor \log 2 \mid ?a + ?b \rfloor \leq p - (\text{bitlen } |m1|) - e1 + 2$ 
      by (auto simp: algebra-simps bitlen-alt-def  $\langle m1 \neq 0 \rangle$ )
      also have  $\dots \leq -?e$ 
      using bitlen-nonneg[of |m1|] by (simp add: k1-def)
      finally show  $?f \leq -?e$  by simp
    qed
    also have  $\text{sgn } ?b = \text{sgn } m2$ 

```



```

    using powr-gt-zero[of 2 e2]
    by (auto simp add: sgn-if zero-less-mult-iff simp del: powr-gt-zero)
    also have  $\lfloor (\text{real-of-int } (2 * ?m1) + \text{real-of-int } (\text{sgn } m2)) * 2^{\text{powr real-of-int } (?f - ?e - 1)} \rfloor =$ 
       $\lfloor \text{Float } (?m1 * 2 + \text{sgn } m2) * 2^{(?e - 1)} * 2^{\text{powr } ?f} \rfloor$ 
    by (simp flip: powr-add powr-realpow add: algebra-simps)
    finally
    show  $\lfloor (?a + ?b) * 2^{\text{powr } ?f} \rfloor = \lfloor \text{real-of-float } (\text{Float } (?m1 * 2 + \text{sgn } m2) * 2^{(?e - 1)}) * 2^{\text{powr } ?f} \rfloor$  .
  qed
  then show ?thesis
    by transfer (simp add: plus-down-def ac-simps Let-def)
qed

```

lemma *compute-float-plus-down-naive*: $\text{float-plus-down } p \ x \ y = \text{float-round-down } p \ (x + y)$
 by transfer (auto simp: plus-down-def)

qualified lemma *compute-float-plus-down*[code]:
 fixes $p::\text{nat}$ and $m1 \ e1 \ m2 \ e2::\text{int}$
 shows $\text{float-plus-down } p \ (\text{Float } m1 \ e1) \ (\text{Float } m2 \ e2) =$
 (if $m1 = 0$ then $\text{float-round-down } p \ (\text{Float } m2 \ e2)$
 else if $m2 = 0$ then $\text{float-round-down } p \ (\text{Float } m1 \ e1)$
 else
 (if $e1 \geq e2$ then
 (let $k1 = \text{Suc } p - \text{nat } (\text{bitlen } |m1|)$ in
 if $\text{bitlen } |m2| > e1 - e2 - k1 - 2$
 then $\text{float-round-down } p \ ((\text{Float } m1 \ e1) + (\text{Float } m2 \ e2))$
 else $\text{float-round-down } p \ (\text{Float } (m1 * 2^{(\text{Suc } (\text{Suc } k1))} + \text{sgn } m2) \ (e1 - \text{int } k1 - 2))$)
 else $\text{float-plus-down } p \ (\text{Float } m2 \ e2) \ (\text{Float } m1 \ e1))$)
proof –
 {
 assume $\text{bitlen } |m2| \leq e1 - e2 - (\text{Suc } p - \text{nat } (\text{bitlen } |m1|)) - 2$ $m1 \neq 0 \ m2 \neq 0 \ e1 \geq e2$
 note *compute-far-float-plus-down*[OF this]
 }
 then show ?thesis
 by transfer (simp add: Let-def plus-down-def ac-simps)
qed

qualified lemma *compute-float-plus-up*[code]: $\text{float-plus-up } p \ x \ y = - \text{float-plus-down } p \ (-x) \ (-y)$
 using *truncate-down-uminus-eq*[of $p \ x + y$]
 by transfer (simp add: plus-down-def plus-up-def ac-simps)

lemma *mantissa-zero*: $\text{mantissa } 0 = 0$
 by (fact *mantissa-0*)

qualified lemma *compute-float-less*[code]: $a < b \longleftrightarrow \text{is-float-pos } (\text{float-plus-down } 0 \ b \ (- \ a))$

using *truncate-down*[of $0 \ b - a$] *truncate-down-pos*[of $b - a \ 0$]
by *transfer* (*auto simp: plus-down-def*)

qualified lemma *compute-float-le*[code]: $a \leq b \longleftrightarrow \text{is-float-nonneg } (\text{float-plus-down } 0 \ b \ (- \ a))$

using *truncate-down*[of $0 \ b - a$] *truncate-down-nonneg*[of $b - a \ 0$]
by *transfer* (*auto simp: plus-down-def*)

end

lemma *plus-down-mono*: $\text{plus-down } p \ a \ b \leq \text{plus-down } p \ c \ d$ **if** $a + b \leq c + d$
by (*auto simp: plus-down-def intro!: truncate-down-mono that*)

lemma *plus-up-mono*: $\text{plus-up } p \ a \ b \leq \text{plus-up } p \ c \ d$ **if** $a + b \leq c + d$
by (*auto simp: plus-up-def intro!: truncate-up-mono that*)

43.14 Approximate Multiplication

lemma *mult-mono-nonpos-nonneg*: $a * b \leq c * d$
if $a \leq c \ a \leq 0 \ 0 \leq d \ d \leq b$ **for** $a \ b \ c \ d :: 'a :: \text{ordered-ring}$
by (*meson dual-order.trans mult-left-mono-neg mult-right-mono that*)

lemma *mult-mono-nonneg-nonpos*: $b * a \leq d * c$
if $a \leq c \ c \leq 0 \ 0 \leq d \ d \leq b$ **for** $a \ b \ c \ d :: 'a :: \text{ordered-ring}$
by (*meson dual-order.trans mult-right-mono-neg mult-left-mono that*)

lemma *mult-mono-nonpos-nonpos*: $a * b \leq c * d$
if $a \geq c \ a \leq 0 \ b \geq d \ d \leq 0$ **for** $a \ b \ c \ d :: \text{real}$
by (*meson dual-order.trans mult-left-mono-neg mult-right-mono-neg that*)

lemma *mult-float-mono1*:

shows $a \leq b \implies ab \leq bb \implies$

$aa \leq a \implies$

$b \leq ba \implies$

$ac \leq ab \implies$

$bb \leq bc \implies$

$\text{plus-down prec } (\text{nprt } aa * \text{pprt } bc)$

$(\text{plus-down prec } (\text{nprt } ba * \text{nprt } bc)$

$(\text{plus-down prec } (\text{pprt } aa * \text{pprt } ac)$

$(\text{pprt } ba * \text{nprt } ac)))$

$\leq \text{plus-down prec } (\text{nprt } a * \text{pprt } bb)$

$(\text{plus-down prec } (\text{nprt } b * \text{nprt } bb)$

$(\text{plus-down prec } (\text{pprt } a * \text{pprt } ab)$

$(\text{pprt } b * \text{nprt } ab)))$

by (*smt (verit, best) mult-mono plus-down-mono add-mono nprt-mono nprt-le-zero zero-le-pprt*

pprt-mono mult-mono-nonpos-nonneg mult-mono-nonpos-nonpos mult-mono-nonneg-nonpos)

lemma *mult-float-mono2*:
shows $a \leq b \implies$
 $ab \leq bb \implies$
 $aa \leq a \implies$
 $b \leq ba \implies$
 $ac \leq ab \implies$
 $bb \leq bc \implies$
 $\text{plus-up prec } (\text{pprt } b * \text{pprt } bb)$
 $(\text{plus-up prec } (\text{pprt } a * \text{nprrt } bb))$
 $(\text{plus-up prec } (\text{nprrt } b * \text{pprt } ab))$
 $(\text{nprrt } a * \text{nprrt } ab)))$
 $\leq \text{plus-up prec } (\text{pprt } ba * \text{pprt } bc)$
 $(\text{plus-up prec } (\text{pprt } aa * \text{nprrt } bc))$
 $(\text{plus-up prec } (\text{nprrt } ba * \text{pprt } ac))$
 $(\text{nprrt } aa * \text{nprrt } ac)))$
by (*smt* (*verit*, *best*) *plus-up-mono add-mono mult-mono nprrt-mono nprrt-le-zero*
zero-le-pprrt pprrt-mono
mult-mono-nonpos-nonneg mult-mono-nonpos-nonpos mult-mono-nonneg-nonpos)

43.15 Approximate Power

lemma *div2-less-self*[*termination-simp*]: $\text{odd } n \implies n \text{ div } 2 < n$ **for** $n :: \text{nat}$
by (*simp add: odd-pos*)

fun *power-down* :: $\text{nat} \Rightarrow \text{real} \Rightarrow \text{nat} \Rightarrow \text{real}$
where
 $\text{power-down } p \ x \ 0 = 1$
 $|\ \text{power-down } p \ x \ (\text{Suc } n) =$
 $(\text{if odd } n \text{ then truncate-down } (\text{Suc } p) ((\text{power-down } p \ x \ (\text{Suc } n \text{ div } 2))^2)$
 $\text{else truncate-down } (\text{Suc } p) (x * \text{power-down } p \ x \ n))$

fun *power-up* :: $\text{nat} \Rightarrow \text{real} \Rightarrow \text{nat} \Rightarrow \text{real}$
where
 $\text{power-up } p \ x \ 0 = 1$
 $|\ \text{power-up } p \ x \ (\text{Suc } n) =$
 $(\text{if odd } n \text{ then truncate-up } p ((\text{power-up } p \ x \ (\text{Suc } n \text{ div } 2))^2)$
 $\text{else truncate-up } p (x * \text{power-up } p \ x \ n))$

lift-definition *power-up-fl* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{nat} \Rightarrow \text{float}$ **is** *power-up*
by (*induct-tac rule: power-up.induct*) *simp-all*

lift-definition *power-down-fl* :: $\text{nat} \Rightarrow \text{float} \Rightarrow \text{nat} \Rightarrow \text{float}$ **is** *power-down*
by (*induct-tac rule: power-down.induct*) *simp-all*

lemma *power-float-transfer*[*transfer-rule*]:
 $(\text{rel-fun pcr-float } (\text{rel-fun } (=) \text{ pcr-float})) \ (\sim) \ (\sim)$
unfolding *power-def*
by *transfer-prover*

lemma *compute-power-up-fl[code]*:
 $\text{power-up-fl } p \ x \ 0 = 1$
 $\text{power-up-fl } p \ x \ (\text{Suc } n) =$
 (if odd n then $\text{float-round-up } p \ ((\text{power-up-fl } p \ x \ (\text{Suc } n \ \text{div } 2))^2)$
 else $\text{float-round-up } p \ (x * \text{power-up-fl } p \ x \ n)$)
and *compute-power-down-fl[code]*:
 $\text{power-down-fl } p \ x \ 0 = 1$
 $\text{power-down-fl } p \ x \ (\text{Suc } n) =$
 (if odd n then $\text{float-round-down } (\text{Suc } p) \ ((\text{power-down-fl } p \ x \ (\text{Suc } n \ \text{div } 2))^2)$
 else $\text{float-round-down } (\text{Suc } p) \ (x * \text{power-down-fl } p \ x \ n)$)
unfolding *atomize-conj* **by** *transfer simp*

lemma *power-down-pos*: $0 < x \implies 0 < \text{power-down } p \ x \ n$
by (*induct* $p \ x \ n$ *rule*: *power-down.induct*)
(auto simp del: odd-Suc-div-two intro!: truncate-down-pos)

lemma *power-down-nonneg*: $0 \leq x \implies 0 \leq \text{power-down } p \ x \ n$
by (*induct* $p \ x \ n$ *rule*: *power-down.induct*)
(auto simp del: odd-Suc-div-two intro!: truncate-down-nonneg mult-nonneg-nonneg)

lemma *power-down*: $0 \leq x \implies \text{power-down } p \ x \ n \leq x \wedge n$
proof (*induct* $p \ x \ n$ *rule*: *power-down.induct*)
case ($2 \ p \ x \ n$)
have ?case **if** odd n
proof –
from *that* 2 **have** $(\text{power-down } p \ x \ (\text{Suc } n \ \text{div } 2)) \wedge 2 \leq (x \wedge (\text{Suc } n \ \text{div } 2))$
 $\wedge 2$
by (*auto intro: power-mono power-down-nonneg simp del: odd-Suc-div-two*)
also have $\dots = x \wedge (\text{Suc } n \ \text{div } 2 * 2)$
by (*simp flip: power-mult*)
also have $\text{Suc } n \ \text{div } 2 * 2 = \text{Suc } n$
using $\langle \text{odd } n \rangle$ **by** *presburger*
finally show ?thesis
using *that* **by** (*auto intro!: truncate-down-le simp del: odd-Suc-div-two*)
qed
then show ?case
by (*auto intro!: truncate-down-le mult-left-mono 2 mult-nonneg-nonneg power-down-nonneg*)
qed *simp*

lemma *power-up*: $0 \leq x \implies x \wedge n \leq \text{power-up } p \ x \ n$
proof (*induct* $p \ x \ n$ *rule*: *power-up.induct*)
case ($2 \ p \ x \ n$)
have ?case **if** odd n
proof –
from *that even-Suc* **have** $\text{Suc } n = \text{Suc } n \ \text{div } 2 * 2$
by *presburger*
then have $x \wedge \text{Suc } n \leq (x \wedge (\text{Suc } n \ \text{div } 2))^2$
by (*simp flip: power-mult*)

also from that 2 have $\dots \leq (\text{power-up } p \ x \ (\text{Suc } n \ \text{div } 2))^2$
 by (auto intro: power-mono simp del: odd-Suc-div-two)
 finally show ?thesis
 using that by (auto intro!: truncate-up-le simp del: odd-Suc-div-two)
 qed
 then show ?case
 by (auto intro!: truncate-up-le mult-left-mono 2)
 qed simp

lemmas power-up-le = order-trans[OF - power-up]
 and power-up-less = less-le-trans[OF - power-up]
 and power-down-le = order-trans[OF power-down]

lemma power-down-fl: $0 \leq x \implies \text{power-down-fl } p \ x \ n \leq x \wedge n$
 by transfer (rule power-down)

lemma power-up-fl: $0 \leq x \implies x \wedge n \leq \text{power-up-fl } p \ x \ n$
 by transfer (rule power-up)

lemma real-power-up-fl: $\text{real-of-float } (\text{power-up-fl } p \ x \ n) = \text{power-up } p \ x \ n$
 by transfer simp

lemma real-power-down-fl: $\text{real-of-float } (\text{power-down-fl } p \ x \ n) = \text{power-down } p \ x \ n$
 by transfer simp

lemmas [simp del] = power-down.simps(2) power-up.simps(2)

lemmas power-down-simp = power-down.simps(2)
 lemmas power-up-simp = power-up.simps(2)

lemma power-down-even-nonneg: $\text{even } n \implies 0 \leq \text{power-down } p \ x \ n$
 by (induct p x n rule: power-down.induct)
 (auto simp: power-down-simp simp del: odd-Suc-div-two intro!: truncate-down-nonneg)

lemma power-down-eq-zero-iff[simp]: $\text{power-down } p \ x \ n = 0 \iff b = 0 \wedge n \neq 0$
 proof (induction n arbitrary: b rule: less-induct)
 case (less x)
 then show ?case
 using power-down-simp[of - - x - 1]
 by (cases x) (auto simp add: div2-less-self)
 qed

lemma power-down-nonneg-iff[simp]:
 $\text{power-down } p \ x \ n \geq 0 \iff \text{even } n \vee b \geq 0$
 proof (induction n arbitrary: b rule: less-induct)
 case (less x)

```

show ?case
  using less(1)[of  $x - 1$  b] power-down-simp[of - -  $x - 1$ ]
  by (cases x) (auto simp: algebra-split-simps zero-le-mult-iff)
qed

```

```

lemma power-down-neg-iff[simp]:
  power-down prec b n < 0  $\longleftrightarrow$ 
    b < 0  $\wedge$  odd n
  using power-down-nonneg-iff[of prec b n] by (auto simp del: power-down-nonneg-iff)

```

```

lemma power-down-nonpos-iff[simp]:
  notes [simp del] = power-down-neg-iff power-down-eq-zero-iff
  shows power-down prec b n  $\leq$  0  $\longleftrightarrow$  b < 0  $\wedge$  odd n  $\vee$  b = 0  $\wedge$  n  $\neq$  0
  using power-down-neg-iff[of prec b n] power-down-eq-zero-iff[of prec b n]
  by auto

```

```

lemma power-down-mono:
  power-down prec a n  $\leq$  power-down prec b n
  if ((0  $\leq$  a  $\wedge$  a  $\leq$  b)  $\vee$  (odd n  $\wedge$  a  $\leq$  b)  $\vee$  (even n  $\wedge$  a  $\leq$  0  $\wedge$  b  $\leq$  a))
  using that
proof (induction n arbitrary: a b rule: less-induct)
  case (less i)
  show ?case
  proof (cases i)
    case j: (Suc j)
    note IH = less[unfolded j even-Suc not-not]
    note [simp del] = power-down.simps
    show ?thesis
  proof cases
    assume [simp]: even j
    have a * power-down prec a j  $\leq$  b * power-down prec b j
    by (metis IH(1) IH(2) <even j> lessI linear mult-mono mult-mono' mult-mono-nonpos-nonneg
power-down-even-nonneg)
    then have truncate-down (Suc prec) (a * power-down prec a j)  $\leq$  truncate-down (Suc prec) (b * power-down prec b j)
    by (auto intro!: truncate-down-mono simp: abs-le-square-iff[symmetric]
abs-real-def)
    then show ?thesis
      unfolding j
      by (simp add: power-down-simp)
  next
    assume [simp]: odd j
    have power-down prec 0 (Suc (j div 2))  $\leq$  - power-down prec b (Suc (j div 2))
    if b < 0 even (j div 2)
    by (metis even-Suc le-minus-iff Suc-neg-Zero neg-equal-zero power-down-eq-zero-iff
power-down-nonpos-iff that)
    then have truncate-down (Suc prec) ((power-down prec a (Suc (j div 2)))2)
       $\leq$  truncate-down (Suc prec) ((power-down prec b (Suc (j div 2)))2)

```

```

    by (smt (verit) IH Suc-less-eq (odd j) div2-less-self mult-mono-nonpos-nonpos
        Suc-neq-Zero power2-eq-square power-down-neg-iff power-down-nonpos-iff
        power-mono truncate-down-mono)
    then show ?thesis
    unfolding j by (simp add: power-down-simp)
  qed
qed simp
qed

lemma power-up-even-nonneg: even n  $\implies$  0  $\leq$  power-up p x n
  by (induct p x n rule: power-up.induct)
  (auto simp: power-up.simps simp del: odd-Suc-div-two)

lemma power-up-eq-zero-iff[simp]: power-up prec b n = 0  $\longleftrightarrow$  b = 0  $\wedge$  n  $\neq$  0
proof (induction n arbitrary: b rule: less-induct)
  case (less x)
  then show ?case
    using power-up-simp[of - x - 1]
    by (cases x) (auto simp: algebra-split-simps zero-le-mult-iff div2-less-self)
qed

lemma power-up-nonneg-iff[simp]:
  power-up prec b n  $\geq$  0  $\longleftrightarrow$  even n  $\vee$  b  $\geq$  0
proof (induction n arbitrary: b rule: less-induct)
  case (less x)
  show ?case
    using less(1)[of x - 1 b] power-up-simp[of - x - 1]
    by (cases x) (auto simp: algebra-split-simps zero-le-mult-iff)
qed

lemma power-up-neg-iff[simp]:
  power-up prec b n < 0  $\longleftrightarrow$  b < 0  $\wedge$  odd n
  using power-up-nonneg-iff[of prec b n] by (auto simp del: power-up-nonneg-iff)

lemma power-up-nonpos-iff[simp]:
  notes [simp del] = power-up-neg-iff power-up-eq-zero-iff
  shows power-up prec b n  $\leq$  0  $\longleftrightarrow$  b < 0  $\wedge$  odd n  $\vee$  b = 0  $\wedge$  n  $\neq$  0
  using power-up-neg-iff[of prec b n] power-up-eq-zero-iff[of prec b n]
  by auto

lemma power-up-mono:
  power-up prec a n  $\leq$  power-up prec b n
  if ((0  $\leq$  a  $\wedge$  a  $\leq$  b)  $\vee$  (odd n  $\wedge$  a  $\leq$  b)  $\vee$  (even n  $\wedge$  a  $\leq$  0  $\wedge$  b  $\leq$  a))
  using that
proof (induction n arbitrary: a b rule: less-induct)
  case (less i)
  show ?case
  proof (cases i)

```

```

case  $j$ : (Suc  $j$ )
note  $IH = less[unfolded\ j\ even\text{-}Suc\ not\text{-}not]$ 
note [simp del] = power-up.simps
show ?thesis
proof cases
  assume [simp]: even  $j$ 
  have  $a * power\text{-}up\ prec\ a\ j \leq b * power\text{-}up\ prec\ b\ j$ 
  by (metis  $IH(1)\ IH(2)\ \langle even\ j \rangle\ lessI\ linear\ mult\text{-}mono\ mult\text{-}mono'\ mult\text{-}mono\text{-}nonpos\text{-}nonneg$ 
power-up-even-nonneg)
  then have  $truncate\text{-}up\ prec\ (a * power\text{-}up\ prec\ a\ j) \leq truncate\text{-}up\ prec\ (b * power\text{-}up\ prec\ b\ j)$ 
  by (auto intro!: truncate-up-mono simp: abs-le-square-iff[symmetric] abs-real-def)
  then show ?thesis
    unfolding  $j$ 
    by (simp add: power-up-simp)
next
  assume [simp]: odd  $j$ 
  have  $power\text{-}up\ prec\ 0\ (Suc\ (j\ div\ 2)) \leq -\ power\text{-}up\ prec\ b\ (Suc\ (j\ div\ 2))$ 
  if  $b < 0\ even\ (j\ div\ 2)$ 
  by (metis Suc-neq-Zero even-Suc neg-0-le-iff-le power-up-eq-zero-iff power-up-nonpos-iff
that)
  then have  $truncate\text{-}up\ prec\ ((power\text{-}up\ prec\ a\ (Suc\ (j\ div\ 2)))^2) \leq truncate\text{-}up\ prec\ ((power\text{-}up\ prec\ b\ (Suc\ (j\ div\ 2)))^2)$ 
  using  $IH$ 
  by (auto intro!: truncate-up-mono intro: order-trans[where y=0] simp: abs-le-square-iff[symmetric] abs-real-def div2-less-self)
  then show ?thesis
    unfolding  $j$ 
    by (simp add: power-up-simp)
qed
qed simp
qed

```

43.16 Lemmas needed by Approximate

```

lemma Float-num[simp]:
  real-of-float (Float 1 0) = 1
  real-of-float (Float 1 1) = 2
  real-of-float (Float 1 2) = 4
  real-of-float (Float 1 (- 1)) = 1/2
  real-of-float (Float 1 (- 2)) = 1/4
  real-of-float (Float 1 (- 3)) = 1/8
  real-of-float (Float (- 1) 0) = -1
  real-of-float (Float (numeral  $n$ ) 0) = numeral  $n$ 
  real-of-float (Float (- numeral  $n$ ) 0) = - numeral  $n$ 
using two-powr-int-float[of 2] two-powr-int-float[of -1] two-powr-int-float[of -2]
  two-powr-int-float[of -3]
using powr-realpow[of 2 2] powr-realpow[of 2 3]

```


using *powr-minus*[of 2::real 1] *powr-minus*[of 2::real 2] *powr-minus*[of 2::real 3]
by *auto*

lemma *real-of-Float-int*[simp]: *real-of-float* (Float *n* 0) = *real n*
by *simp*

lemma *float-zero*[simp]: *real-of-float* (Float 0 *e*) = 0
by *simp*

lemma *abs-div-2-less*: $a \neq 0 \implies a \neq -1 \implies |(a::int) \text{ div } 2| < |a|$
by *arith*

lemma *lapprox-rat*: *real-of-float* (*lapprox-rat prec x y*) \leq *real-of-int x / real-of-int y*
by (*simp add: lapprox-rat.rep-eq truncate-down*)

lemma *mult-div-le*:
fixes *a b* :: *int*
assumes $b > 0$
shows $a \geq b * (a \text{ div } b)$
by (*smt (verit, ccfv-threshold) assms minus-div-mult-eq-mod mod-int-pos-iff mult.commute*)

lemma *lapprox-rat-nonneg*:
assumes $0 \leq x$ **and** $0 \leq y$
shows $0 \leq \text{real-of-float} (\text{lapprox-rat } n \ x \ y)$
using *assms*
by *transfer (simp add: truncate-down-nonneg)*

lemma *rapprox-rat*: *real-of-int x / real-of-int y* \leq *real-of-float (rapprox-rat prec x y)*
by (*simp add: rapprox-rat.rep-eq truncate-up*)

lemma *rapprox-rat-le1*:
assumes $0 \leq x$ $0 < y$ $x \leq y$
shows *real-of-float (rapprox-rat n x y)* ≤ 1
using *assms*
by *transfer (simp add: truncate-up-le1)*

lemma *rapprox-rat-nonneg-nonpos*: $0 \leq x \implies y \leq 0 \implies \text{real-of-float} (\text{rapprox-rat } n \ x \ y) \leq 0$
by *transfer (simp add: truncate-up-nonpos divide-nonneg-nonpos)*

lemma *rapprox-rat-nonpos-nonneg*: $x \leq 0 \implies 0 \leq y \implies \text{real-of-float} (\text{rapprox-rat } n \ x \ y) \leq 0$
by *transfer (simp add: truncate-up-nonpos divide-nonpos-nonneg)*

lemma *real-divl*: *real-divl prec x y* $\leq x / y$
by (*simp add: real-divl-def truncate-down*)

lemma *real-divr*: $x / y \leq \text{real-divr } \text{prec } x \ y$
by (*simp add: real-divr-def truncate-up*)

lemma *float-divl*: $\text{real-of-float } (\text{float-divl } \text{prec } x \ y) \leq x / y$
by *transfer (rule real-divl)*

lemma *real-divl-lower-bound*: $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-divl } \text{prec } x \ y$
by (*simp add: real-divl-def truncate-down-nonneg*)

lemma *float-divl-lower-bound*: $0 \leq x \implies 0 \leq y \implies 0 \leq \text{real-of-float } (\text{float-divl } \text{prec } x \ y)$
by *transfer (rule real-divl-lower-bound)*

lemma *exponent-1*: $\text{exponent } 1 = 0$
using *exponent-float[of 1 0]* **by** (*simp add: one-float-def*)

lemma *mantissa-1*: $\text{mantissa } 1 = 1$
using *mantissa-float[of 1 0]* **by** (*simp add: one-float-def*)

lemma *bitlen-1*: $\text{bitlen } 1 = 1$
by (*simp add: bitlen-alt-def*)

lemma *float-upper-bound*: $x \leq 2^{\text{powr } (\text{bitlen } |\text{mantissa } x| + \text{exponent } x)}$
proof (*cases x = 0*)

case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $\text{mantissa } x \neq 0$
using *mantissa-eq-zero-iff* **by** *auto*
have $x = \text{mantissa } x * 2^{\text{powr } (\text{exponent } x)}$
by (*rule mantissa-exponent*)
also have $\text{mantissa } x \leq |\text{mantissa } x|$
by *simp*
also have $\dots \leq 2^{\text{powr } (\text{bitlen } |\text{mantissa } x|)}$
using *bitlen-bounds[of |mantissa x|] bitlen-nonneg* $\langle \text{mantissa } x \neq 0 \rangle$
by (*auto simp del: of-int-abs simp add: powr-int*)
finally show *?thesis* **by** (*simp add: powr-add*)
qed

lemma *real-divl-pos-less1-bound*:
assumes $0 < x \ x \leq 1$
shows $1 \leq \text{real-divl } \text{prec } 1 \ x$
using *assms*
by (*auto intro!: truncate-down-ge1 simp: real-divl-def*)

lemma *float-divl-pos-less1-bound*:
 $0 < \text{real-of-float } x \implies \text{real-of-float } x \leq 1 \implies \text{prec} \geq 1 \implies$
 $1 \leq \text{real-of-float } (\text{float-divl } \text{prec } 1 \ x)$

by *transfer* (rule *real-divl-pos-less1-bound*)

lemma *float-divr*: *real-of-float x / real-of-float y ≤ real-of-float (float-divr prec x y)*
by (*simp add: float-divr.rep-eq real-divr*)

lemma *real-divr-pos-less1-lower-bound*:
assumes $0 < x$
and $x \leq 1$
shows $1 \leq \text{real-divr prec } 1\ x$
proof –
have $1 \leq 1 / x$
using $\langle 0 < x \rangle$ *and* $\langle x \leq 1 \rangle$ **by** *auto*
also have $\dots \leq \text{real-divr prec } 1\ x$
using *real-divr*[**where** $x = 1$ *and* $y = x$] **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *float-divr-pos-less1-lower-bound*: $0 < x \implies x \leq 1 \implies 1 \leq \text{float-divr prec } 1\ x$
by *transfer* (rule *real-divr-pos-less1-lower-bound*)

lemma *real-divr-nonpos-pos-upper-bound*: $x \leq 0 \implies 0 \leq y \implies \text{real-divr prec } x\ y \leq 0$
by (*simp add: real-divr-def truncate-up-nonpos divide-le-0-iff*)

lemma *float-divr-nonpos-pos-upper-bound*:
 $\text{real-of-float } x \leq 0 \implies 0 \leq \text{real-of-float } y \implies \text{real-of-float (float-divr prec } x\ y) \leq 0$
by *transfer* (rule *real-divr-nonpos-pos-upper-bound*)

lemma *real-divr-nonneg-neg-upper-bound*: $0 \leq x \implies y \leq 0 \implies \text{real-divr prec } x\ y \leq 0$
by (*simp add: real-divr-def truncate-up-nonpos divide-le-0-iff*)

lemma *float-divr-nonneg-neg-upper-bound*:
 $0 \leq \text{real-of-float } x \implies \text{real-of-float } y \leq 0 \implies \text{real-of-float (float-divr prec } x\ y) \leq 0$
by *transfer* (rule *real-divr-nonneg-neg-upper-bound*)

lemma *Float-le-zero-iff*: $\text{Float } a\ b \leq 0 \iff a \leq 0$
by (*auto simp: zero-float-def mult-le-0-iff*)

lemma *real-of-float-pprt[simp]*:
fixes $a :: \text{float}$
shows $\text{real-of-float (pprt } a) = \text{pprt (real-of-float } a)$
unfolding *pprt-def sup-float-def max-def sup-real-def* **by** *auto*

lemma *real-of-float-nprt[simp]*:

```

fixes  $a :: \text{float}$ 
shows  $\text{real-of-float } (\text{nprt } a) = \text{nprt } (\text{real-of-float } a)$ 
unfolding  $\text{nprt-def inf-float-def min-def inf-real-def}$  by auto

context
begin

lift-definition  $\text{int-floor-fl} :: \text{float} \Rightarrow \text{int}$  is  $\text{floor}$  .

qualified lemma  $\text{compute-int-floor-fl}[\text{code}]$ :
   $\text{int-floor-fl } (\text{Float } m \ e) = (\text{if } 0 \leq e \text{ then } m * 2^{\text{nat } e} \text{ else } m \text{ div } (2^{\text{nat } (-e)}))$ 
apply transfer
by (smt (verit, ccfv-threshold)  $\text{Float.rep-eq compute-real-of-float floor-divide-of-int-eq}$ 

   $\text{floor-of-int of-int-1 of-int-add of-int-mult of-int-power}$ )

lift-definition  $\text{floor-fl} :: \text{float} \Rightarrow \text{float}$  is  $\lambda x. \text{real-of-int } \lfloor x \rfloor$ 
by simp

qualified lemma  $\text{compute-floor-fl}[\text{code}]$ :
   $\text{floor-fl } (\text{Float } m \ e) = (\text{if } 0 \leq e \text{ then } \text{Float } m \ e \text{ else } \text{Float } (m \text{ div } (2^{\text{nat } (-e)})) \ 0)$ 
apply transfer
using  $\text{compute-int-floor-fl int-floor-fl.rep-eq powr-int}$  by auto

end

lemma  $\text{floor-fl: real-of-float } (\text{floor-fl } x) \leq \text{real-of-float } x$ 
by transfer simp

lemma  $\text{int-floor-fl: real-of-int } (\text{int-floor-fl } x) \leq \text{real-of-float } x$ 
by transfer simp

lemma  $\text{floor-pos-exp: exponent } (\text{floor-fl } x) \geq 0$ 
proof (cases floor-fl x = 0)
  case True
    then show ?thesis
      by (simp add: floor-fl-def)
  next
    case False
      have  $\text{eq: floor-fl } x = \text{Float } \lfloor \text{real-of-float } x \rfloor \ 0$ 
      by transfer simp
      obtain  $i$  where  $\lfloor \text{real-of-float } x \rfloor = \text{mantissa } (\text{floor-fl } x) * 2^i \ 0 = \text{exponent}$ 
       $(\text{floor-fl } x) - \text{int } i$ 
      by (rule denormalize-shift[OF eq False])
      then show ?thesis
      by simp
qed

```

```

lemma compute-mantissa[code]:
  mantissa (Float m e) =
    (if m = 0 then 0 else if 2 dvd m then mantissa (normfloat (Float m e)) else m)
  by (auto simp: mantissa-float Float.abs-eq simp flip: zero-float-def)

```

```

lemma compute-exponent[code]:
  exponent (Float m e) =
    (if m = 0 then 0 else if 2 dvd m then exponent (normfloat (Float m e)) else e)
  by (auto simp: exponent-float Float.abs-eq simp flip: zero-float-def)

```

```

lifting-update Float.float.lifting
lifting-forget Float.float.lifting

```

```

end

```

44 Pointwise instantiation of functions to algebra type classes

```

theory Function-Algebras
imports Main
begin

  Pointwise operations

instantiation fun :: (type, plus) plus
begin

definition  $f + g = (\lambda x. f\ x + g\ x)$ 
instance ..

end

lemma plus-fun-apply [simp]:
   $(f + g)\ x = f\ x + g\ x$ 
  by (simp add: plus-fun-def)

instantiation fun :: (type, zero) zero
begin

definition  $0 = (\lambda x. 0)$ 
instance ..

end

lemma zero-fun-apply [simp]:
   $0\ x = 0$ 
  by (simp add: zero-fun-def)

```

instantiation $fun :: (type, times) \times times$
begin

definition $f * g = (\lambda x. f \ x * g \ x)$
instance ..

end

lemma $times\text{-}fun\text{-}apply$ [simp]:
 $(f * g) \ x = f \ x * g \ x$
by (simp add: times-fun-def)

instantiation $fun :: (type, one) \times one$
begin

definition $1 = (\lambda x. 1)$
instance ..

end

lemma $one\text{-}fun\text{-}apply$ [simp]:
 $1 \ x = 1$
by (simp add: one-fun-def)

Additive structures

instance $fun :: (type, semigroup\text{-}add) \times semigroup\text{-}add$
by standard (simp add: fun-eq-iff add.assoc)

instance $fun :: (type, cancel\text{-}semigroup\text{-}add) \times cancel\text{-}semigroup\text{-}add$
by standard (simp-all add: fun-eq-iff)

instance $fun :: (type, ab\text{-}semigroup\text{-}add) \times ab\text{-}semigroup\text{-}add$
by standard (simp add: fun-eq-iff add.commute)

instance $fun :: (type, cancel\text{-}ab\text{-}semigroup\text{-}add) \times cancel\text{-}ab\text{-}semigroup\text{-}add$
by standard (simp-all add: fun-eq-iff diff-diff-eq)

instance $fun :: (type, monoid\text{-}add) \times monoid\text{-}add$
by standard (simp-all add: fun-eq-iff)

instance $fun :: (type, comm\text{-}monoid\text{-}add) \times comm\text{-}monoid\text{-}add$
by standard simp

instance $fun :: (type, cancel\text{-}comm\text{-}monoid\text{-}add) \times cancel\text{-}comm\text{-}monoid\text{-}add$..

instance $fun :: (type, group\text{-}add) \times group\text{-}add$
by standard (simp-all add: fun-eq-iff)

instance $fun :: (type, ab\text{-}group\text{-}add) \times ab\text{-}group\text{-}add$

by *standard simp-all*

Multiplicative structures

instance *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*
by *standard (simp add: fun-eq-iff mult.assoc)*

instance *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*
by *standard (simp add: fun-eq-iff mult.commute)*

instance *fun* :: (*type*, *monoid-mult*) *monoid-mult*
by *standard (simp-all add: fun-eq-iff)*

instance *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*
by *standard simp*

Misc

instance *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* ..

instance *fun* :: (*type*, *mult-zero*) *mult-zero*
by *standard (simp-all add: fun-eq-iff)*

instance *fun* :: (*type*, *zero-neq-one*) *zero-neq-one*
by *standard (simp add: fun-eq-iff)*

Ring structures

instance *fun* :: (*type*, *semiring*) *semiring*
by *standard (simp-all add: fun-eq-iff algebra-simps)*

instance *fun* :: (*type*, *comm-semiring*) *comm-semiring*
by *standard (simp add: fun-eq-iff algebra-simps)*

instance *fun* :: (*type*, *semiring-0*) *semiring-0* ..

instance *fun* :: (*type*, *comm-semiring-0*) *comm-semiring-0* ..

instance *fun* :: (*type*, *semiring-0-cancel*) *semiring-0-cancel* ..

instance *fun* :: (*type*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel* ..

instance *fun* :: (*type*, *semiring-1*) *semiring-1* ..

lemma *numeral-fun*:

⟨*numeral n* = ($\lambda x::'a$. *numeral n*)⟩

by (*induction n*) (*simp-all only: numeral.simps plus-fun-def, simp-all*)

lemma *numeral-fun-apply* [*simp*]:

⟨*numeral n x* = *numeral n*⟩

by (*simp add: numeral-fun*)

```

lemma of-nat-fun: of-nat n = ( $\lambda x::'a$ . of-nat n)
proof –
  have comp: comp = ( $\lambda f g x$ . f (g x))
    by (rule ext)+ simp
  have plus-fun: plus = ( $\lambda f g x$ . f x + g x)
    by (rule ext, rule ext) (fact plus-fun-def)
  have of-nat n = (comp (plus (1::'b))  $\widetilde{\sim}$  n) ( $\lambda x::'a$ . 0)
    by (simp add: of-nat-def plus-fun zero-fun-def one-fun-def comp)
  also have ... = comp ((plus 1)  $\widetilde{\sim}$  n) ( $\lambda x::'a$ . 0)
    by (simp only: comp-funpow)
  finally show ?thesis by (simp add: of-nat-def comp)
qed

lemma of-nat-fun-apply [simp]:
  of-nat n x = of-nat n
  by (simp add: of-nat-fun)

instance fun :: (type, comm-semiring-1) comm-semiring-1 ..

instance fun :: (type, semiring-1-cancel) semiring-1-cancel ..

instance fun :: (type, comm-semiring-1-cancel) comm-semiring-1-cancel
  by standard (auto simp add: times-fun-def algebra-simps)

instance fun :: (type, semiring-char-0) semiring-char-0
proof
  from inj-of-nat have inj ( $\lambda n$  (x::'a). of-nat n :: 'b)
    by (rule inj-fun)
  then have inj ( $\lambda n$ . of-nat n :: 'a  $\Rightarrow$  'b)
    by (simp add: of-nat-fun)
  then show inj (of-nat :: nat  $\Rightarrow$  'a  $\Rightarrow$  'b) .
qed

instance fun :: (type, ring) ring ..

instance fun :: (type, comm-ring) comm-ring ..

instance fun :: (type, ring-1) ring-1 ..

instance fun :: (type, comm-ring-1) comm-ring-1 ..

instance fun :: (type, ring-char-0) ring-char-0 ..

  Ordered structures

instance fun :: (type, ordered-ab-semigroup-add) ordered-ab-semigroup-add
  by standard (auto simp add: le-fun-def intro: add-left-mono)

instance fun :: (type, ordered-cancel-ab-semigroup-add) ordered-cancel-ab-semigroup-add
  ..

```



```

instance fun :: (type, ordered-ab-semigroup-add-imp-le) ordered-ab-semigroup-add-imp-le
  by standard (simp add: le-fun-def)

instance fun :: (type, ordered-comm-monoid-add) ordered-comm-monoid-add ..

instance fun :: (type, ordered-cancel-comm-monoid-add) ordered-cancel-comm-monoid-add
  ..

instance fun :: (type, ordered-ab-group-add) ordered-ab-group-add ..

instance fun :: (type, ordered-semiring) ordered-semiring
  by standard (auto simp add: le-fun-def intro: mult-left-mono mult-right-mono)

instance fun :: (type, dioid) dioid
proof standard
  fix a b :: 'a  $\Rightarrow$  'b
  show  $a \leq b \iff (\exists c. b = a + c)$ 
    unfolding le-fun-def plus-fun-def fun-eq-iff choice-iff[symmetric, of  $\lambda x c. b x$ 
  =  $a x + c$ ]
    by (intro arg-cong[where f=All] ext canonically-ordered-monoid-add-class.le-iff-add)
qed

instance fun :: (type, ordered-comm-semiring) ordered-comm-semiring
  by standard (fact mult-left-mono)

instance fun :: (type, ordered-cancel-semiring) ordered-cancel-semiring ..

instance fun :: (type, ordered-cancel-comm-semiring) ordered-cancel-comm-semiring
  ..

instance fun :: (type, ordered-ring) ordered-ring ..

instance fun :: (type, ordered-comm-ring) ordered-comm-ring ..

lemmas func-plus = plus-fun-def
lemmas func-zero = zero-fun-def
lemmas func-times = times-fun-def
lemmas func-one = one-fun-def

end

```

45 Pointwise instantiation of functions to division

```

theory Function-Division
imports Function-Algebras
begin

```

45.1 Syntactic with division

instantiation *fun* :: (*type*, *inverse*) *inverse*
begin

definition *inverse f* = *inverse* ∘ *f*

definition *f div g* = ($\lambda x. f\ x \ / \ g\ x$)

instance ..

end

lemma *inverse-fun-apply* [*simp*]:
inverse f x = *inverse* (*f x*)
by (*simp add: inverse-fun-def*)

lemma *divide-fun-apply* [*simp*]:
(*f / g*) *x* = *f x / g x*
by (*simp add: divide-fun-def*)

Unfortunately, we cannot lift this operations to algebraic type classes for division: being different from the constant zero function $f \neq 0$ is too weak as precondition. So we must introduce our own set of lemmas.

abbreviation *zero-free* :: (*'b* \Rightarrow *'a::field*) \Rightarrow *bool* **where**
zero-free f $\equiv \neg (\exists x. f\ x = 0)$

lemma *fun-left-inverse*:
fixes *f* :: *'b* \Rightarrow *'a::field*
shows *zero-free f* \Longrightarrow *inverse f * f* = 1
by (*simp add: fun-eq-iff*)

lemma *fun-right-inverse*:
fixes *f* :: *'b* \Rightarrow *'a::field*
shows *zero-free f* \Longrightarrow *f * inverse f* = 1
by (*simp add: fun-eq-iff*)

lemma *fun-divide-inverse*:
fixes *f g* :: *'b* \Rightarrow *'a::field*
shows *f / g* = *f * inverse g*
by (*simp add: fun-eq-iff divide-inverse*)

Feel free to extend this.

Another possibility would be a reformulation of the division type classes to use a *zero-free* predicate rather than a direct $a \neq 0$ condition.

end

46 Lexicographic order on functions

```
theory Fun-Lexorder
imports Main
begin
```

```
definition less-fun :: ('a::linorder  $\Rightarrow$  'b::linorder)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  bool
where
```

```
  less-fun f g  $\longleftrightarrow$  ( $\exists k. f\ k < g\ k \wedge (\forall k' < k. f\ k' = g\ k')$ )
```

```
lemma less-funI:
```

```
  assumes  $\exists k. f\ k < g\ k \wedge (\forall k' < k. f\ k' = g\ k')$ 
```

```
  shows less-fun f g
```

```
  using assms by (simp add: less-fun-def)
```

```
lemma less-funE:
```

```
  assumes less-fun f g
```

```
  obtains k where  $f\ k < g\ k$  and  $\bigwedge k'. k' < k \implies f\ k' = g\ k'$ 
```

```
  using assms unfolding less-fun-def by blast
```

```
lemma less-fun-asymp:
```

```
  assumes less-fun f g
```

```
  shows  $\neg$  less-fun g f
```

```
proof
```

```
  from assms obtain k1 where  $k1: f\ k1 < g\ k1$   $k' < k1 \implies f\ k' = g\ k'$  for  $k'$ 
```

```
  by (blast elim!: less-funE)
```

```
  assume less-fun g f then obtain k2 where  $k2: g\ k2 < f\ k2$   $k' < k2 \implies g\ k' = f\ k'$  for  $k'$ 
```

```
  by (blast elim!: less-funE)
```

```
  show False proof (cases k1 k2 rule: linorder-cases)
```

```
    case equal with k1 k2 show False by simp
```

```
  next
```

```
    case less with k2 have  $g\ k1 = f\ k1$  by simp
```

```
    with k1 show False by simp
```

```
  next
```

```
    case greater with k1 have  $f\ k2 = g\ k2$  by simp
```

```
    with k2 show False by simp
```

```
qed
```

```
qed
```

```
lemma less-fun-irrefl:
```

```
   $\neg$  less-fun f f
```

```
proof
```

```
  assume less-fun f f
```

```
  then obtain k where  $f\ k < f\ k$ 
```

```
  by (blast elim!: less-funE)
```

```
  then show False by simp
```

```
qed
```

```

lemma less-fun-trans:
  assumes less-fun f g and less-fun g h
  shows less-fun f h
proof (rule less-funI)
  from  $\langle \text{less-fun } f \text{ } g \rangle$  obtain k1 where  $k1: f \text{ } k1 < g \text{ } k1 \text{ } k' < k1 \implies f \text{ } k' = g \text{ } k'$ 
for k'
    by (blast elim!:: less-funE)
  from  $\langle \text{less-fun } g \text{ } h \rangle$  obtain k2 where  $k2: g \text{ } k2 < h \text{ } k2 \text{ } k' < k2 \implies g \text{ } k' = h \text{ } k'$ 
for k'
    by (blast elim!:: less-funE)
  show  $\exists k. f \text{ } k < h \text{ } k \wedge (\forall k' < k. f \text{ } k' = h \text{ } k')$ 
proof (cases k1 k2 rule: linorder-cases)
    case equal with k1 k2 show ?thesis by (auto simp add: exI [of - k2])
  next
    case less with k2 have  $g \text{ } k1 = h \text{ } k1 \wedge k'. k' < k1 \implies g \text{ } k' = h \text{ } k'$  by simp-all
    with k1 show ?thesis by (auto intro: exI [of - k1])
  next
    case greater with k1 have  $f \text{ } k2 = g \text{ } k2 \wedge k'. k' < k2 \implies f \text{ } k' = g \text{ } k'$  by simp-all
    with k2 show ?thesis by (auto intro: exI [of - k2])
qed
qed

```

```

lemma order-less-fun:
  class.order ( $\lambda f g. \text{less-fun } f \text{ } g \vee f = g$ ) less-fun
by (rule order-strictI) (auto intro: less-fun-trans intro!: less-fun-irrefl less-fun-asy)

```

```

lemma less-fun-trichotomy:
  assumes finite {k. f k  $\neq$  g k}
  shows less-fun f g  $\vee$  f = g  $\vee$  less-fun g f
proof –
  { define K where  $K = \{k. f \text{ } k \neq g \text{ } k\}$ 
    assume  $f \neq g$ 
    then obtain k' where  $f \text{ } k' \neq g \text{ } k'$  by auto
    then have [simp]:  $K \neq \{\}$  by (auto simp add: K-def)
    with assms have [simp]: finite K by (simp add: K-def)
    define q where  $q = \text{Min } K$ 
    then have  $q \in K$  and  $\bigwedge k. k \in K \implies k \geq q$  by auto
    then have  $\bigwedge k. \neg k \geq q \implies k \notin K$  by blast
    then have *:  $\bigwedge k. k < q \implies f \text{ } k = g \text{ } k$  by (simp add: K-def)
    from  $\langle q \in K \rangle$  have  $f \text{ } q \neq g \text{ } q$  by (simp add: K-def)
    then have  $f \text{ } q < g \text{ } q \vee f \text{ } q > g \text{ } q$  by auto
    with * have less-fun f g  $\vee$  less-fun g f
      by (auto intro!: less-funI)
    } then show ?thesis by blast
qed

```

end

47 The *going-to* filter

```
theory Going-To-Filter
  imports Complex-Main
begin
```

```
definition going-to-within :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b filter  $\Rightarrow$  'a set  $\Rightarrow$  'a filter
  ( $\langle \langle \text{open-block notation} = \langle \text{mixfix going-to-within} \rangle \rangle (-) / \text{going'-to } (-) / \text{within } (-) \rangle$ 
  [1000,60,60] 60)
  where  $f \text{ going-to } F \text{ within } A = \inf (\text{filtercomap } f F) (\text{principal } A)$ 
```

```
abbreviation going-to :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b filter  $\Rightarrow$  'a filter
  (infix  $\langle \text{going'-to} \rangle$  60)
  where  $f \text{ going-to } F \equiv f \text{ going-to } F \text{ within UNIV}$ 
```

The *going-to* filter is, in a sense, the opposite of *filtermap*. It corresponds to the intuition of, given a function $f : A \rightarrow B$ and a filter F on the range of B , looking at such values of x that $f(x)$ approaches F . This can be written as $f \text{ going-to } F$.

A classic example is the *at-infinity* filter, which describes the neighbourhood of infinity (i. e. all values sufficiently far away from the zero). This can also be written as *norm going-to at-top*.

Additionally, the *going-to* filter can be restricted with an optional ‘within’ parameter. For instance, if one would want to consider the filter of complex numbers near infinity that do not lie on the negative real line, one could write *cmod going-to at-top within – complex-of-real ‘{..0}’*.

A third, less mathematical example lies in the complexity analysis of algorithms. Suppose we wanted to say that an algorithm on lists takes $O(n^2)$ time where n is the length of the input list. We can write this using the Landau symbols from the AFP, where the underlying filter is *length going-to sequentially*. If, on the other hand, we want to look the complexity of the algorithm on sorted lists, we could use the filter *length going-to sequentially within Collect sorted*.

```
lemma going-to-def:  $f \text{ going-to } F = \text{filtercomap } f F$ 
  by (simp add: going-to-within-def)
```

```
lemma eventually-going-toI [intro]:
  assumes eventually  $P F$ 
  shows eventually  $(\lambda x. P (f x)) (f \text{ going-to } F)$ 
  using assms by (auto simp: going-to-def)
```

```
lemma filterlim-going-toI-weak [intro]:  $\text{filterlim } f F (f \text{ going-to } F \text{ within } A)$ 
  unfolding going-to-within-def
  by (meson filterlim-filtercomap filterlim-iff inf-le1 le-filter-def)
```

```
lemma going-to-mono:  $F \leq G \Longrightarrow A \subseteq B \Longrightarrow f \text{ going-to } F \text{ within } A \leq f \text{ going-to } G \text{ within } B$ 
```

unfolding *going-to-within-def* **by** (*intro inf-mono filtercomap-mono simp-all*)

lemma *going-to-inf*:

f going-to (inf F G) within A = inf (f going-to F within A) (f going-to G within A)

by (*simp add: going-to-within-def filtercomap-inf inf-assoc inf-commute inf-left-commute*)

lemma *going-to-sup*:

f going-to (sup F G) within A ≥ sup (f going-to F within A) (f going-to G within A)

by (*auto simp: going-to-within-def intro!: inf.coboundedI1 filtercomap-sup filtercomap-mono*)

lemma *going-to-top* [*simp*]: *f going-to top within A = principal A*

by (*simp add: going-to-within-def*)

lemma *going-to-bot* [*simp*]: *f going-to bot within A = bot*

by (*simp add: going-to-within-def*)

lemma *going-to-principal*:

f going-to principal A within B = principal (f -‘ A ∩ B)

by (*simp add: going-to-within-def*)

lemma *going-to-within-empty* [*simp*]: *f going-to F within {} = bot*

by (*simp add: going-to-within-def*)

lemma *going-to-within-union* [*simp*]:

f going-to F within (A ∪ B) = sup (f going-to F within A) (f going-to F within B)

by (*simp add: going-to-within-def flip: inf-sup-distrib1*)

lemma *eventually-going-to-at-top-linorder*:

fixes *f :: 'a ⇒ 'b :: linorder*

shows *eventually P (f going-to at-top within A) ⟷ (∃ C. ∀ x ∈ A. f x ≥ C ⟶ P x)*

unfolding *going-to-within-def eventually-filtercomap*

eventually-inf-principal eventually-at-top-linorder **by** *fast*

lemma *eventually-going-to-at-bot-linorder*:

fixes *f :: 'a ⇒ 'b :: linorder*

shows *eventually P (f going-to at-bot within A) ⟷ (∃ C. ∀ x ∈ A. f x ≤ C ⟶ P x)*

unfolding *going-to-within-def eventually-filtercomap*

eventually-inf-principal eventually-at-bot-linorder **by** *fast*

lemma *eventually-going-to-at-top-dense*:

fixes *f :: 'a ⇒ 'b :: {linorder, no-top}*

shows *eventually P (f going-to at-top within A) ⟷ (∃ C. ∀ x ∈ A. f x > C ⟶ P x)*

unfolding *going-to-within-def eventually-filtercomap*
eventually-inf-principal eventually-at-top-dense **by** *fast*

lemma *eventually-going-to-at-bot-dense:*
fixes $f :: 'a \Rightarrow 'b :: \{\text{linorder}, \text{no-bot}\}$
shows *eventually* P (*f going-to at-bot within A*) $\longleftrightarrow (\exists C. \forall x \in A. f\ x < C \longrightarrow P\ x)$
unfolding *going-to-within-def eventually-filtercomap*
eventually-inf-principal eventually-at-bot-dense **by** *fast*

lemma *eventually-going-to-nhds:*
eventually P (*f going-to nhds a within A*) \longleftrightarrow
 $(\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f\ x \in S \longrightarrow P\ x))$
unfolding *going-to-within-def eventually-filtercomap eventually-inf-principal*
eventually-nhds **by** *fast*

lemma *eventually-going-to-at:*
eventually P (*f going-to (at a within B) within A*) \longleftrightarrow
 $(\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in A. f\ x \in B \cap S - \{a\} \longrightarrow P\ x))$
unfolding *at-within-def going-to-inf eventually-inf-principal*
eventually-going-to-nhds going-to-principal **by** *fast*

lemma *norm-going-to-at-top-eq: norm going-to at-top = at-infinity*
by (*simp add: eventually-at-infinity eventually-going-to-at-top-linorder filter-eq-iff*)

lemmas *at-infinity-altdef = norm-going-to-at-top-eq* [*symmetric*]

end

48 Big sum and product over function bodies

theory *Groups-Big-Fun*
imports
Main
begin

48.1 Abstract product

locale *comm-monoid-fun = comm-monoid*
begin

definition $G :: ('b \Rightarrow 'a) \Rightarrow 'a$
where
expand-set: G g = comm-monoid-set.F f 1 g {a. g a \neq 1}

interpretation F : *comm-monoid-set f 1*
..

lemma *expand-superset:*

assumes *finite* A and $\{a. g\ a \neq 1\} \subseteq A$
 shows $G\ g = F.F\ g\ A$
 using $F.mono-neutral-right\ assms\ expand-set$ by *fastforce*

lemma *conditionalize*:

assumes *finite* A
 shows $F.F\ g\ A = G\ (\lambda a. \text{if } a \in A \text{ then } g\ a \text{ else } 1)$
 using *assms*
 by (*smt* (*verit*, *ccfu-threshold*) *Diff-iff* $F.mono-neutral-cong-right\ expand-set\ mem-Collect-eq\ subsetI$)

lemma *neutral* [*simp*]:

$G\ (\lambda a. 1) = 1$
 by (*simp add: expand-set*)

lemma *update* [*simp*]:

assumes *finite* $\{a. g\ a \neq 1\}$
 assumes $g\ a = 1$
 shows $G\ (g(a := b)) = b * G\ g$
proof (*cases* $b = 1$)
 case *True* with $\langle g\ a = 1 \rangle$ show *?thesis*
 by (*simp add: expand-set*) (*rule* $F.cong$, *auto*)
next
 case *False*
 moreover have $\{a'. a' \neq a \longrightarrow g\ a' \neq 1\} = \text{insert } a\ \{a. g\ a \neq 1\}$
 by *auto*
 moreover from $\langle g\ a = 1 \rangle$ have $a \notin \{a. g\ a \neq 1\}$
 by *simp*
 moreover have $F.F\ (\lambda a'. \text{if } a' = a \text{ then } b \text{ else } g\ a')\ \{a. g\ a \neq 1\} = F.F\ g\ \{a. g\ a \neq 1\}$
 by (*rule* $F.cong$) (*auto simp add: $\langle g\ a = 1 \rangle$*)
 ultimately show *?thesis* using $\langle \text{finite } \{a. g\ a \neq 1\} \rangle$ by (*simp add: expand-set*)
qed

lemma *infinite* [*simp*]:

$\neg \text{finite } \{a. g\ a \neq 1\} \implies G\ g = 1$
 by (*simp add: expand-set*)

lemma *cong* [*cong*]:

assumes $\bigwedge a. g\ a = h\ a$
 shows $G\ g = G\ h$
 using *assms* by (*simp add: expand-set*)

lemma *not-neutral-obtains-not-neutral*:

assumes $G\ g \neq 1$
 obtains a where $g\ a \neq 1$
 using *assms* by (*auto elim: F.not-neutral-contains-not-neutral simp add: expand-set*)

lemma *reindex-cong*:

assumes *bij l*

assumes $g \circ l = h$

shows $G\ g = G\ h$

proof –

from *assms* have *unfold*: $h = g \circ l$ **by** *simp*

from $\langle \text{bij } l \rangle$ have *inj l* **by** (rule *bij-is-inj*)

then have *inj-on l* $\{a. h\ a \neq 1\}$ **by** (rule *inj-on-subset*) *simp*

moreover from $\langle \text{bij } l \rangle$ have $\{a. g\ a \neq 1\} = l^{-1} \{a. h\ a \neq 1\}$

by (auto *simp add: image-Collect unfold elim: bij-pointE*)

moreover have $\bigwedge x. x \in \{a. h\ a \neq 1\} \implies g\ (l\ x) = h\ x$

by (*simp add: unfold*)

ultimately have $F.F\ g\ \{a. g\ a \neq 1\} = F.F\ h\ \{a. h\ a \neq 1\}$

by (rule *F.reindex-cong*)

then show *?thesis* **by** (*simp add: expand-set*)

qed

lemma *distrib*:

assumes *finite* $\{a. g\ a \neq 1\}$ and *finite* $\{a. h\ a \neq 1\}$

shows $G\ (\lambda a. g\ a * h\ a) = G\ g * G\ h$

proof –

from *assms* have *finite* $(\{a. g\ a \neq 1\} \cup \{a. h\ a \neq 1\})$ **by** *simp*

moreover have $\{a. g\ a * h\ a \neq 1\} \subseteq \{a. g\ a \neq 1\} \cup \{a. h\ a \neq 1\}$

by *auto* (drule *sym, simp*)

ultimately show *?thesis*

using *assms*

by (*simp add: expand-superset [of $\{a. g\ a \neq 1\} \cup \{a. h\ a \neq 1\}$] F.distrib*)

qed

lemma *swap*:

assumes *finite C*

assumes *subset*: $\{a. \exists b. g\ a\ b \neq 1\} \times \{b. \exists a. g\ a\ b \neq 1\} \subseteq C$ (is *?A* \times *?B* $\subseteq C$)

shows $G\ (\lambda a. G\ (g\ a)) = G\ (\lambda b. G\ (\lambda a. g\ a\ b))$

proof –

from $\langle \text{finite } C \rangle$ *subset*

have *finite* $(\{a. \exists b. g\ a\ b \neq 1\} \times \{b. \exists a. g\ a\ b \neq 1\})$

by (rule *rev-finite-subset*)

then have *fins*:

finite $\{b. \exists a. g\ a\ b \neq 1\}$ *finite* $\{a. \exists b. g\ a\ b \neq 1\}$

by (auto *simp add: finite-cartesian-product-iff*)

have *subsets*: $\bigwedge a. \{b. g\ a\ b \neq 1\} \subseteq \{b. \exists a. g\ a\ b \neq 1\}$

$\bigwedge b. \{a. g\ a\ b \neq 1\} \subseteq \{a. \exists b. g\ a\ b \neq 1\}$

$\{a. F.F\ (g\ a)\ \{b. \exists a. g\ a\ b \neq 1\} \neq 1\} \subseteq \{a. \exists b. g\ a\ b \neq 1\}$

$\{a. F.F\ (\lambda aa. g\ aa\ a)\ \{a. \exists b. g\ a\ b \neq 1\} \neq 1\} \subseteq \{b. \exists a. g\ a\ b \neq 1\}$

by (auto *elim: F.not-neutral-contains-not-neutral*)

from *F.swap* have

$F.F\ (\lambda a. F.F\ (g\ a)\ \{b. \exists a. g\ a\ b \neq 1\})\ \{a. \exists b. g\ a\ b \neq 1\} =$

```

    F.F (λb. F.F (λa. g a b) {a. ∃ b. g a b ≠ 1}) {b. ∃ a. g a b ≠ 1} .
  with subsets fins have G (λa. F.F (g a) {b. ∃ a. g a b ≠ 1}) =
    G (λb. F.F (λa. g a b) {a. ∃ b. g a b ≠ 1})
  by (auto simp add: expand-superset [of {b. ∃ a. g a b ≠ 1}]
      expand-superset [of {a. ∃ b. g a b ≠ 1}])
  with subsets fins show ?thesis
  by (auto simp add: expand-superset [of {b. ∃ a. g a b ≠ 1}]
      expand-superset [of {a. ∃ b. g a b ≠ 1}])
qed

```

lemma cartesian-product:

```

  assumes finite C
  assumes subset: {a. ∃ b. g a b ≠ 1} × {b. ∃ a. g a b ≠ 1} ⊆ C (is ?A × ?B ⊆
  C)
  shows G (λa. G (g a)) = G (λ(a, b). g a b)
proof -
  from subset ⟨finite C⟩ have fin-prod: finite (?A × ?B)
  by (rule finite-subset)
  from fin-prod have finite ?A and finite ?B
  by (auto simp add: finite-cartesian-product-iff)
  have *: G (λa. G (g a)) =
    (F.F (λa. F.F (g a) {b. ∃ a. g a b ≠ 1}) {a. ∃ b. g a b ≠ 1})
  using ⟨finite ?A⟩ ⟨finite ?B⟩ expand-superset
  by (smt (verit, del-insts) Collect-mono local.cong not-neutral-obtains-not-neutral)
  have **: {p. (case p of (a, b) ⇒ g a b) ≠ 1} ⊆ ?A × ?B
  by auto
  show ?thesis
  using ⟨finite C⟩ expand-superset
  using * ** F.cartesian-product fin-prod by force
qed

```

lemma cartesian-product2:

```

  assumes fin: finite D
  assumes subset: {(a, b). ∃ c. g a b c ≠ 1} × {c. ∃ a b. g a b c ≠ 1} ⊆ D (is
  ?AB × ?C ⊆ D)
  shows G (λ(a, b). G (g a b)) = G (λ(a, b, c). g a b c)
proof -
  have bij: bij (λ(a, b, c). ((a, b), c))
  by (auto intro!: bijI injI simp add: image-def)
  have {p. ∃ c. g (fst p) (snd p) c ≠ 1} × {c. ∃ p. g (fst p) (snd p) c ≠ 1} ⊆ D
  by auto (insert subset, blast)
  with fin have G (λp. G (g (fst p) (snd p))) = G (λ(p, c). g (fst p) (snd p) c)
  by (rule cartesian-product)
  then have G (λ(a, b). G (g a b)) = G (λ((a, b), c). g a b c)
  by (auto simp add: split-def)
  also have G (λ((a, b), c). g a b c) = G (λ(a, b, c). g a b c)
  using bij by (rule reindex-cong [of λ(a, b, c). ((a, b), c)]) (simp add: fun-eq-iff)
  finally show ?thesis .
qed

```

```

lemma delta [simp]:
   $G (\lambda b. \text{if } b = a \text{ then } g \ b \text{ else } \mathbf{1}) = g \ a$ 
proof -
  have  $\{b. (\text{if } b = a \text{ then } g \ b \text{ else } \mathbf{1}) \neq \mathbf{1}\} \subseteq \{a\}$  by auto
  then show ?thesis by (simp add: expand-superset [of {a}])
qed

lemma delta' [simp]:
   $G (\lambda b. \text{if } a = b \text{ then } g \ b \text{ else } \mathbf{1}) = g \ a$ 
proof -
  have  $(\lambda b. \text{if } a = b \text{ then } g \ b \text{ else } \mathbf{1}) = (\lambda b. \text{if } b = a \text{ then } g \ b \text{ else } \mathbf{1})$ 
    by (simp add: fun-eq-iff)
  then have  $G (\lambda b. \text{if } a = b \text{ then } g \ b \text{ else } \mathbf{1}) = G (\lambda b. \text{if } b = a \text{ then } g \ b \text{ else } \mathbf{1})$ 
    by (simp cong del: cong)
  then show ?thesis by simp
qed

end

```

48.2 Concrete sum

```

context comm-monoid-add
begin

sublocale Sum-any: comm-monoid-fun plus 0
  rewrites comm-monoid-set.F plus 0 = sum
  defines Sum-any = Sum-any.G
proof -
  show comm-monoid-fun plus 0 ..
  then interpret Sum-any: comm-monoid-fun plus 0 .
  from sum-def show comm-monoid-set.F plus 0 = sum by (auto intro: sym)
qed

end

syntax (ASCII)
  -Sum-any :: pttrn  $\Rightarrow$  'a  $\Rightarrow$  'a::comm-monoid-add ( $\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder}$ 
SUM \rangle \rangle \text{SUM -. -} \rangle [0, 10] 10))
syntax
  -Sum-any :: pttrn  $\Rightarrow$  'a  $\Rightarrow$  'a::comm-monoid-add ( $\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{binder}$ 
 $\sum \rangle \rangle \text{SUM -. -} \rangle [0, 10] 10)$ )
syntax-consts
  -Sum-any  $\equiv$  Sum-any
translations
   $\sum a. b \equiv \text{CONST } \text{Sum-any } (\lambda a. b)$ 

lemma Sum-any-left-distrib:
  fixes r :: 'a :: semiring-0

```

```

assumes finite {a. g a ≠ 0}
shows Sum-any g * r = ( $\sum$  n. g n * r)
by (metis (mono-tags, lifting) Collect-mono Sum-any.expand-superset assms mult-zero-left
sum-distrib-right)

```

```

lemma Sum-any-right-distrib:
  fixes r :: 'a :: semiring-0
  assumes finite {a. g a ≠ 0}
  shows r * Sum-any g = ( $\sum$  n. r * g n)
  by (metis (mono-tags, lifting) Collect-mono Sum-any.expand-superset assms mult-zero-right
sum-distrib-left)

```

```

lemma Sum-any-product:
  fixes f g :: 'b ⇒ 'a::semiring-0
  assumes finite {a. f a ≠ 0} and finite {b. g b ≠ 0}
  shows Sum-any f * Sum-any g = ( $\sum$  a.  $\sum$  b. f a * g b)
proof –
  have  $\forall a. (\sum b. a * g b) = a * \text{Sum-any } g$ 
    by (simp add: Sum-any-right-distrib assms(2))
  then show ?thesis
    by (simp add: Sum-any-left-distrib assms(1))
qed

```

```

lemma Sum-any-eq-zero-iff [simp]:
  fixes f :: 'a ⇒ nat
  assumes finite {a. f a ≠ 0}
  shows Sum-any f = 0  $\longleftrightarrow$  f = ( $\lambda$ -. 0)
  using assms by (simp add: Sum-any.expand-set fun-eq-iff)

```

48.3 Concrete product

```

context comm-monoid-mult
begin

```

```

sublocale Prod-any: comm-monoid-fun times 1
  rewrites comm-monoid-set.F times 1 = prod
  defines Prod-any = Prod-any.G
proof –
  show comm-monoid-fun times 1 ..
  then interpret Prod-any: comm-monoid-fun times 1 .
  from prod-def show comm-monoid-set.F times 1 = prod by (auto intro: sym)
qed

```

```

end

```

```

syntax (ASCII)
  -Prod-any :: ptrn ⇒ 'a ⇒ 'a::comm-monoid-mult ( $\langle \langle \text{indent}=4 \text{ notation}=\langle \text{binder}$ 
PROD \rangle \rangle PROD -. - \rangle [0, 10] 10)
syntax

```

```
-Prod-any :: pttm ⇒ 'a ⇒ 'a :: comm-monoid-mult (⟨⟨indent=2 notation=⟨binder
Π⟩⟩Π -. -)⟩ [0, 10] 10)
```

```
syntax-consts
```

```
-Prod-any == Prod-any
```

```
translations
```

```
Π a. b == CONST Prod-any (λa. b)
```

```
lemma Prod-any-zero:
```

```
fixes f :: 'b ⇒ 'a :: comm-semiring-1
```

```
assumes finite {a. f a ≠ 1}
```

```
assumes f a = 0
```

```
shows (Π a. f a) = 0
```

```
proof -
```

```
from ⟨f a = 0⟩ have f a ≠ 1 by simp
```

```
with ⟨f a = 0⟩ have ∃ a. f a ≠ 1 ∧ f a = 0 by blast
```

```
with ⟨finite {a. f a ≠ 1}⟩ show ?thesis
```

```
by (simp add: Prod-any.expand-set prod-zero)
```

```
qed
```

```
lemma Prod-any-not-zero:
```

```
fixes f :: 'b ⇒ 'a :: comm-semiring-1
```

```
assumes finite {a. f a ≠ 1}
```

```
assumes (Π a. f a) ≠ 0
```

```
shows f a ≠ 0
```

```
using assms Prod-any-zero [of f] by blast
```

```
lemma power-Sum-any:
```

```
assumes finite {a. f a ≠ 0}
```

```
shows c ^ (Σ a. f a) = (Π a. c ^ f a)
```

```
proof -
```

```
have {a. c ^ f a ≠ 1} ⊆ {a. f a ≠ 0}
```

```
by (auto intro: ccontr)
```

```
with assms show ?thesis
```

```
by (simp add: Sum-any.expand-set Prod-any.expand-superset power-sum)
```

```
qed
```

```
end
```

49 Infinite Type Class

The type class of infinite sets (originally from the Incredible Proof Machine)

```
theory Infinite-Typeclass
```

```
imports Complex-Main
```

```
begin
```

```
class infinite =
```

```
assumes infinite-UNIV: infinite (UNIV::'a set)
```

begin

lemma *arb-element*: $\text{finite } Y \implies \exists x :: 'a. x \notin Y$
using *ex-new-if-finite infinite-UNIV*
by *blast*

lemma *arb-finite-subset*: $\text{finite } Y \implies \exists X :: 'a \text{ set}. Y \cap X = \{\} \wedge \text{finite } X \wedge n \leq \text{card } X$

proof –

assume *fin*: $\text{finite } Y$
then obtain *X* **where** $X \subseteq \text{UNIV} - Y$ $\text{finite } X$ $n \leq \text{card } X$
using *infinite-UNIV*
by (*metis Compl-eq-Diff-UNIV finite-compl infinite-arbitrarily-large order-refl*)
then show *?thesis*
by *auto*

qed

lemma *arb-inj-on-finite-infinite*: $\text{finite}(A :: 'b \text{ set}) \implies \exists f :: 'b \Rightarrow 'a. \text{inj-on } f \ A$
by (*meson arb-finite-subset card-le-inj infinite-imp-nonempty*)

lemma *arb-countable-map*: $\text{finite } Y \implies \exists f :: (\text{nat} \Rightarrow 'a). \text{inj } f \wedge \text{range } f \subseteq \text{UNIV} - Y$

using *infinite-UNIV*
by (*auto simp: infinite-countable-subset*)

end

instance *nat* :: *infinite*
by (*intro-classes simp*)

instance *int* :: *infinite*
by (*intro-classes simp*)

instance *rat* :: *infinite*
proof
show *infinite* (*UNIV::rat set*)
by (*simp add: infinite-UNIV-char-0*)
qed

instance *real* :: *infinite*
proof
show *infinite* (*UNIV::real set*)
by (*simp add: infinite-UNIV-char-0*)
qed

instance *complex* :: *infinite*
proof
show *infinite* (*UNIV::complex set*)
by (*simp add: infinite-UNIV-char-0*)

qed

instance *option* :: (*infinite*) *infinite*
 by *intro-classes* (*simp add: infinite-UNIV*)

instance *prod* :: (*infinite*, *type*) *infinite*
 by *intro-classes* (*simp add: finite-prod infinite-UNIV*)

instance *list* :: (*type*) *infinite*
 by *intro-classes* (*simp add: infinite-UNIV-listI*)

end

50 Algebraic operations on sets

theory *Set-Algebras*
imports *Main*
begin

This library lifts operations like addition and multiplication to sets. It was designed to support asymptotic calculations for the now-obsolete BigO theory, but has other uses.

instantiation *set* :: (*plus*) *plus*
begin

definition *plus-set* :: '*a*::*plus set* \Rightarrow '*a set* \Rightarrow '*a set*
 where *set-plus-def*: $A + B = \{c. \exists a \in A. \exists b \in B. c = a + b\}$

instance ..

end

instantiation *set* :: (*times*) *times*
begin

definition *times-set* :: '*a*::*times set* \Rightarrow '*a set* \Rightarrow '*a set*
 where *set-times-def*: $A * B = \{c. \exists a \in A. \exists b \in B. c = a * b\}$

instance ..

end

instantiation *set* :: (*zero*) *zero*
begin

definition *set-zero*[*simp*]: $(0::'\textit{a}::\textit{zero set}) = \{0\}$

instance ..

end

instantiation *set* :: (*one*) *one*
begin

definition *set-one*[*simp*]: (*1*::'*a*::*one set*) = {*1*}

instance ..

end

definition *elt-set-plus* :: '*a*::*plus* ⇒ '*a set* ⇒ '*a set* (**infixl** <+o> 70)
 where $a +_o B = \{c. \exists b \in B. c = a + b\}$

definition *elt-set-times* :: '*a*::*times* ⇒ '*a set* ⇒ '*a set* (**infixl** <*o> 80)
 where $a *_o B = \{c. \exists b \in B. c = a * b\}$

abbreviation (*input*) *elt-set-eq* :: '*a* ⇒ '*a set* ⇒ *bool* (**infix** <=o> 50)
 where $x =_o A \equiv x \in A$

instance *set* :: (*semigroup-add*) *semigroup-add*
 by *standard* (*force simp add: set-plus-def add.assoc*)

instance *set* :: (*ab-semigroup-add*) *ab-semigroup-add*
 by *standard* (*force simp add: set-plus-def add.commute*)

instance *set* :: (*monoid-add*) *monoid-add*
 by *standard* (*simp-all add: set-plus-def*)

instance *set* :: (*comm-monoid-add*) *comm-monoid-add*
 by *standard* (*simp-all add: set-plus-def*)

instance *set* :: (*semigroup-mult*) *semigroup-mult*
 by *standard* (*force simp add: set-times-def mult.assoc*)

instance *set* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
 by *standard* (*force simp add: set-times-def mult.commute*)

instance *set* :: (*monoid-mult*) *monoid-mult*
 by *standard* (*simp-all add: set-times-def*)

instance *set* :: (*comm-monoid-mult*) *comm-monoid-mult*
 by *standard* (*simp-all add: set-times-def*)

lemma *sumset-empty* [*simp*]: $A + \{\} = \{\} \{\} + A = \{\}$
 by (*auto simp: set-plus-def*)

lemma *Un-set-plus*: $(A \cup B) + C = (A+C) \cup (B+C)$ **and** *set-plus-Un*: $C + (A \cup B) = (C+A) \cup (C+B)$

by (auto simp: set-plus-def)

lemma

fixes $A :: 'a::comm-monoid-add\ set$

shows $insert\ set\ plus: (insert\ a\ A) + B = (A+B) \cup (((+)\ a)\ 'B)$ and $set-plus-insert:$
 $B + (insert\ a\ A) = (B+A) \cup (((+)\ a)\ 'B)$

using $add.commute$ by (auto simp: set-plus-def)

lemma $set-add-0$ [simp]:

fixes $A :: 'a::comm-monoid-add\ set$

shows $\{0\} + A = A$

by (metis $comm-monoid-add-class.add-0\ set-zero$)

lemma $set-add-0-right$ [simp]:

fixes $A :: 'a::comm-monoid-add\ set$

shows $A + \{0\} = A$

by (metis $add.comm-neutral\ set-zero$)

lemma $card-plus-sing$:

fixes $A :: 'a::ab-group-add\ set$

shows $card\ (A + \{a\}) = card\ A$

proof (rule $bij-betw-same-card$)

show $bij-betw\ ((+)\ (-a))\ (A + \{a\})\ A$

by (fastforce simp: set-plus-def $bij-betw-def\ image-iff$)

qed

lemma $set-plus-intro$ [intro]: $a \in C \implies b \in D \implies a + b \in C + D$

by (auto simp add: set-plus-def)

lemma $set-plus-elim$:

assumes $x \in A + B$

obtains $a\ b$ where $x = a + b$ and $a \in A$ and $b \in B$

using $assms$ unfolding $set-plus-def$ by fast

lemma $set-plus-intro2$ [intro]: $b \in C \implies a + b \in a + o\ C$

by (auto simp add: $elt-set-plus-def$)

lemma $set-plus-rearrange$: $(a + o\ C) + (b + o\ D) = (a + b) + o\ (C + D)$

for $a\ b :: 'a::comm-monoid-add$

by (auto simp: $elt-set-plus-def\ set-plus-def$; metis $group-cancel.add1\ group-cancel.add2$)

lemma $set-plus-rearrange2$: $a + o\ (b + o\ C) = (a + b) + o\ C$

for $a\ b :: 'a::semigroup-add$

by (auto simp add: $elt-set-plus-def\ add.assoc$)

lemma $set-plus-rearrange3$: $(a + o\ B) + C = a + o\ (B + C)$

for $a :: 'a::semigroup-add$

by (auto simp add: $elt-set-plus-def\ set-plus-def$; metis $add.assoc$)

theorem *set-plus-rearrange4*: $C + (a +_o D) = a +_o (C + D)$
for $a :: 'a::comm-monoid-add$
by (*metis add.commute set-plus-rearrange3*)

lemmas *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2 set-plus-rearrange3 set-plus-rearrange4*

lemma *set-plus-mono* [*intro!*]: $C \subseteq D \implies a +_o C \subseteq a +_o D$
by (*auto simp add: elt-set-plus-def*)

lemma *set-plus-mono2* [*intro*]: $C \subseteq D \implies E \subseteq F \implies C + E \subseteq D + F$
for $C D E F :: 'a::plus set$
by (*auto simp add: set-plus-def*)

lemma *set-plus-mono3* [*intro*]: $a \in C \implies a +_o D \subseteq C + D$
by (*auto simp add: elt-set-plus-def set-plus-def*)

lemma *set-plus-mono4* [*intro*]: $a \in C \implies a +_o D \subseteq D + C$
for $a :: 'a::comm-monoid-add$
by (*auto simp add: elt-set-plus-def set-plus-def ac-simps*)

lemma *set-plus-mono5*: $a \in C \implies B \subseteq D \implies a +_o B \subseteq C + D$
using *order-subst2* **by** *blast*

lemma *set-plus-mono-b*: $C \subseteq D \implies x \in a +_o C \implies x \in a +_o D$
using *set-plus-mono* **by** *blast*

lemma *set-zero-plus* [*simp*]: $0 +_o C = C$
for $C :: 'a::comm-monoid-add set$
by (*auto simp add: elt-set-plus-def*)

lemma *set-zero-plus2*: $0 \in A \implies B \subseteq A + B$
for $A B :: 'a::comm-monoid-add set$
using *set-plus-intro* **by** *fastforce*

lemma *set-plus-imp-minus*: $a \in b +_o C \implies a - b \in C$
for $a b :: 'a::ab-group-add$
by (*auto simp add: elt-set-plus-def ac-simps*)

lemma *set-minus-imp-plus*: $a - b \in C \implies a \in b +_o C$
for $a b :: 'a::ab-group-add$
by (*metis add.commute diff-add-cancel set-plus-intro2*)

lemma *set-minus-plus*: $a - b \in C \iff a \in b +_o C$
for $a b :: 'a::ab-group-add$
by (*meson set-minus-imp-plus set-plus-imp-minus*)

lemma *set-times-intro* [*intro*]: $a \in C \implies b \in D \implies a * b \in C * D$
by (*auto simp add: set-times-def*)

lemma *set-times-elim*:

assumes $x \in A * B$

obtains $a \ b$ **where** $x = a * b$ **and** $a \in A$ **and** $b \in B$

using *assms* **unfolding** *set-times-def* **by** *fast*

lemma *set-times-intro2* [*intro!*]: $b \in C \implies a * b \in a *o C$

by (*auto simp add: elt-set-times-def*)

lemma *set-times-rearrange*: $(a *o C) * (b *o D) = (a * b) *o (C * D)$

for $a \ b :: 'a::comm-monoid-mult$

by (*auto simp add: elt-set-times-def set-times-def; metis mult.assoc mult.left-commute*)

lemma *set-times-rearrange2*: $a *o (b *o C) = (a * b) *o C$

for $a \ b :: 'a::semigroup-mult$

by (*auto simp add: elt-set-times-def mult.assoc*)

lemma *set-times-rearrange3*: $(a *o B) * C = a *o (B * C)$

for $a :: 'a::semigroup-mult$

by (*auto simp add: elt-set-times-def set-times-def; metis mult.assoc*)

theorem *set-times-rearrange4*: $C * (a *o D) = a *o (C * D)$

for $a :: 'a::comm-monoid-mult$

by (*metis mult.commute set-times-rearrange3*)

lemmas *set-times-rearranges* = *set-times-rearrange set-times-rearrange2 set-times-rearrange3 set-times-rearrange4*

lemma *set-times-mono* [*intro*]: $C \subseteq D \implies a *o C \subseteq a *o D$

by (*auto simp add: elt-set-times-def*)

lemma *set-times-mono2* [*intro*]: $C \subseteq D \implies E \subseteq F \implies C * E \subseteq D * F$

for $C \ D \ E \ F :: 'a::times \ set$

by (*auto simp add: set-times-def*)

lemma *set-times-mono3* [*intro*]: $a \in C \implies a *o D \subseteq C * D$

by (*auto simp add: elt-set-times-def set-times-def*)

lemma *set-times-mono4* [*intro*]: $a \in C \implies a *o D \subseteq D * C$

for $a :: 'a::comm-monoid-mult$

by (*auto simp add: elt-set-times-def set-times-def ac-simps*)

lemma *set-times-mono5*: $a \in C \implies B \subseteq D \implies a *o B \subseteq C * D$

by (*meson dual-order.trans set-times-mono set-times-mono3*)

lemma *set-one-times* [*simp*]: $1 *o C = C$

for $C :: 'a::comm-monoid-mult \ set$

by (*auto simp add: elt-set-times-def*)

```

lemma set-times-plus-distrib:  $a * o (b + o C) = (a * b) + o (a * o C)$ 
  for  $a b :: 'a::semiring$ 
  by (auto simp add: elt-set-plus-def elt-set-times-def ring-distrib)

lemma set-times-plus-distrib2:  $a * o (B + C) = (a * o B) + (a * o C)$ 
  for  $a :: 'a::semiring$ 
  by (auto simp: set-plus-def elt-set-times-def;metis distrib-left)

lemma set-times-plus-distrib3:  $(a + o C) * D \subseteq a * o D + C * D$ 
  for  $a :: 'a::semiring$ 
  using distrib-right
  by (fastforce simp add: elt-set-plus-def elt-set-times-def set-times-def set-plus-def)

lemmas set-times-plus-distrib =
  set-times-plus-distrib
  set-times-plus-distrib2

lemma set-neg-intro:  $a \in (- 1) * o C \implies - a \in C$ 
  for  $a :: 'a::ring-1$ 
  by (auto simp add: elt-set-times-def)

lemma set-neg-intro2:  $a \in C \implies - a \in (- 1) * o C$ 
  for  $a :: 'a::ring-1$ 
  by (auto simp add: elt-set-times-def)

lemma set-plus-image:  $S + T = (\lambda(x, y). x + y) ` (S \times T)$ 
  by (fastforce simp: set-plus-def image-iff)

lemma set-times-image:  $S * T = (\lambda(x, y). x * y) ` (S \times T)$ 
  by (fastforce simp: set-times-def image-iff)

lemma finite-set-plus:  $finite\ s \implies finite\ t \implies finite\ (s + t)$ 
  by (simp add: set-plus-image)

lemma finite-set-times:  $finite\ s \implies finite\ t \implies finite\ (s * t)$ 
  by (simp add: set-times-image)

lemma set-sum-alt:
  assumes  $fin: finite\ I$ 
  shows  $sum\ S\ I = \{sum\ s\ I \mid s. \forall i \in I. s\ i \in S\ i\}$ 
  (is - = ?sum I)
  using  $fin$ 
proof induct
  case empty
  then show ?case by simp
next
  case (insert x F)
  have  $sum\ S\ (insert\ x\ F) = S\ x + ?sum\ F$ 
  using insert.hyps by auto

```

```

also have ... = {s x + sum s F | s.  $\forall i \in \text{insert } x F. s i \in S i$ }
  unfolding set-plus-def
proof safe
  fix y s
  assume y  $\in S x \forall i \in F. s i \in S i$ 
  then show  $\exists s'. y + \text{sum } s F = s' x + \text{sum } s' F \wedge (\forall i \in \text{insert } x F. s' i \in S i)$ 
    using insert.hyps
    by (intro exI[of -  $\lambda i. \text{if } i \in F \text{ then } s i \text{ else } y$ ]) (auto simp add: set-plus-def)
  qed auto
  finally show ?case
    using insert.hyps by auto
qed

```

lemma *sum-set-cond-linear*:

```

fixes f :: 'a::comm-monoid-add set  $\Rightarrow$  'b::comm-monoid-add set
assumes [intro!]:  $\bigwedge A B. P A \Longrightarrow P B \Longrightarrow P (A + B) P \{0\}$ 
  and f:  $\bigwedge A B. P A \Longrightarrow P B \Longrightarrow f (A + B) = f A + f B f \{0\} = \{0\}$ 
assumes all:  $\bigwedge i. i \in I \Longrightarrow P (S i)$ 
shows f (sum S I) = sum (f  $\circ$  S) I
proof (cases finite I)
  case True
  from this all show ?thesis
  proof induct
    case empty
    then show ?case by (auto intro!: f)
  next
    case (insert x F)
    from (finite F)  $\langle \bigwedge i. i \in \text{insert } x F \Longrightarrow P (S i) \rangle$  have P (sum S F)
      by induct auto
    with insert show ?case
      by (simp, subst f) auto
  qed
next
  case False
  then show ?thesis by (auto intro!: f)
qed

```

lemma *sum-set-linear*:

```

fixes f :: 'a::comm-monoid-add set  $\Rightarrow$  'b::comm-monoid-add set
assumes  $\bigwedge A B. f(A) + f(B) = f(A + B) f \{0\} = \{0\}$ 
shows f (sum S I) = sum (f  $\circ$  S) I
using sum-set-cond-linear[of  $\lambda x. \text{True } f I S$ ] assms by auto

```

lemma *set-times-Un-distrib*:

```

A * (B  $\cup$  C) = A * B  $\cup$  A * C
(A  $\cup$  B) * C = A * C  $\cup$  B * C
by (auto simp: set-times-def)

```

lemma *set-times-UNION-distrib*:

```

  A *  $\bigcup (M \text{ ' } I) = (\bigcup_{i \in I}. A * M \text{ } i)$ 
   $\bigcup (M \text{ ' } I) * A = (\bigcup_{i \in I}. M \text{ } i * A)$ 
  by (auto simp: set-times-def)

end

```

51 Interval Type

```

theory Interval
  imports
    Complex-Main
    Lattice-Algebras
    Set-Algebras
begin

  A type of non-empty, closed intervals.

  typedef (overloaded) 'a interval =
    {(a::'a::preorder, b). a ≤ b}
    morphisms bounds-of-interval Interval
    by auto

  setup-lifting type-definition-interval

  lift-definition lower::('a::preorder) interval  $\Rightarrow$  'a is fst .

  lift-definition upper::('a::preorder) interval  $\Rightarrow$  'a is snd .

  lemma interval-eq-iff: a = b  $\longleftrightarrow$  lower a = lower b  $\wedge$  upper a = upper b
    by transfer auto

  lemma interval-eqI: lower a = lower b  $\implies$  upper a = upper b  $\implies$  a = b
    by (auto simp: interval-eq-iff)

  lemma lower-le-upper[simp]: lower i ≤ upper i
    by transfer auto

  lift-definition set-of :: 'a::preorder interval  $\Rightarrow$  'a set is  $\lambda x. \{fst\ x \ ..\ snd\ x\}$  .

  lemma set-of-eq: set-of x = {lower x .. upper x}
    by transfer simp

  context notes [[typedef-overloaded]] begin

  lift-definition (code-dt) Interval!::'a::preorder  $\Rightarrow$  'a::preorder  $\Rightarrow$  'a interval option
    is  $\lambda a\ b. \text{if } a \leq b \text{ then Some } (a, b) \text{ else None}$ 
    by auto

  lemma Interval'-split:
    P (Interval' a b)  $\longleftrightarrow$ 

```

$(\forall ivl. a \leq b \longrightarrow lower\ ivl = a \longrightarrow upper\ ivl = b \longrightarrow P\ (Some\ ivl)) \wedge (\neg a \leq b \longrightarrow P\ None)$

by *transfer auto*

lemma *Interval'-split-asm:*

$P\ (Interval'\ a\ b) \longleftrightarrow$

$\neg((\exists ivl. a \leq b \wedge lower\ ivl = a \wedge upper\ ivl = b \wedge \neg P\ (Some\ ivl)) \vee (\neg a \leq b \wedge \neg P\ None))$

unfolding *Interval'-split*

by *auto*

lemmas *Interval'-splits = Interval'-split Interval'-split-asm*

lemma *Interval'-eq-Some:* $Interval'\ a\ b = Some\ i \implies lower\ i = a \wedge upper\ i = b$

by (*simp split: Interval'-splits*)

end

instantiation *interval* :: (*{preorder, equal}*) *equal*

begin

definition *equal-class.equal* $a\ b \equiv (lower\ a = lower\ b) \wedge (upper\ a = upper\ b)$

instance proof **qed** (*simp add: equal-interval-def interval-eq-iff*)

end

instantiation *interval* :: (*preorder*) *ord* **begin**

definition *less-eq-interval* :: '*a interval* \Rightarrow '*a interval* \Rightarrow bool

where *less-eq-interval* $a\ b \longleftrightarrow lower\ b \leq lower\ a \wedge upper\ a \leq upper\ b$

definition *less-interval* :: '*a interval* \Rightarrow '*a interval* \Rightarrow bool

where *less-interval* $x\ y = (x \leq y \wedge \neg y \leq x)$

instance proof **qed**

end

instantiation *interval* :: (*lattice*) *semilattice-sup*

begin

lift-definition *sup-interval* :: '*a interval* \Rightarrow '*a interval* \Rightarrow '*a interval*

is $\lambda(a, b)\ (c, d). (\inf\ a\ c, \sup\ b\ d)$

by (*auto simp: le-infI1 le-supI1*)

lemma *lower-sup[simp]:* $lower\ (\sup\ A\ B) = \inf\ (lower\ A)\ (lower\ B)$

by *transfer auto*

lemma *upper-sup[simp]:* $upper\ (\sup\ A\ B) = \sup\ (upper\ A)\ (upper\ B)$

by *transfer auto*

```

instance proof qed (auto simp: less-eq-interval-def less-interval-def interval-eq-iff)
end

lemma set-of-interval-union: set-of  $A \cup$  set-of  $B \subseteq$  set-of ( $\sup A B$ ) for  $A :: 'a :: \text{lattice interval}$ 
by (auto simp: set-of-eq)

lemma interval-union-commute:  $\sup A B = \sup B A$  for  $A :: 'a :: \text{lattice interval}$ 
by (auto simp add: interval-eq-iff inf.commute sup.commute)

lemma interval-union-mono1: set-of  $a \subseteq$  set-of ( $\sup a A$ ) for  $A :: 'a :: \text{lattice interval}$ 
using set-of-interval-union by blast

lemma interval-union-mono2: set-of  $A \subseteq$  set-of ( $\sup a A$ ) for  $A :: 'a :: \text{lattice interval}$ 
using set-of-interval-union by blast

lift-definition interval-of ::  $'a :: \text{preorder} \Rightarrow 'a \text{ interval}$  is  $\lambda x. (x, x)$ 
by auto

lemma lower-interval-of[simp]: lower (interval-of  $a$ ) =  $a$ 
by transfer auto

lemma upper-interval-of[simp]: upper (interval-of  $a$ ) =  $a$ 
by transfer auto

definition width ::  $'a :: \{\text{preorder}, \text{minus}\} \text{ interval} \Rightarrow 'a$ 
where width  $i = \text{upper } i - \text{lower } i$ 

instantiation interval :: (ordered-ab-semigroup-add) ab-semigroup-add
begin

lift-definition plus-interval ::  $'a \text{ interval} \Rightarrow 'a \text{ interval} \Rightarrow 'a \text{ interval}$ 
is  $\lambda(a, b). \lambda(c, d). (a + c, b + d)$ 
by (auto intro!: add-mono)
lemma lower-plus[simp]: lower (plus  $A B$ ) = plus (lower  $A$ ) (lower  $B$ )
by transfer auto
lemma upper-plus[simp]: upper (plus  $A B$ ) = plus (upper  $A$ ) (upper  $B$ )
by transfer auto

instance proof qed (auto simp: interval-eq-iff less-eq-interval-def ac-simps)
end

instance interval :: ( $\{\text{ordered-ab-semigroup-add}, \text{lattice}\}$ ) ordered-ab-semigroup-add
proof qed (auto simp: less-eq-interval-def intro!: add-mono)

```


instantiation *interval* :: (*{preorder, zero}*) *zero*
begin

lift-definition *zero-interval*::'a *interval* **is** (*0, 0*) **by** *auto*

lemma *lower-zero[simp]*: *lower 0 = 0*

by *transfer auto*

lemma *upper-zero[simp]*: *upper 0 = 0*

by *transfer auto*

instance **proof** **qed**

end

instance *interval* :: (*{ordered-comm-monoid-add}*) *comm-monoid-add*

proof **qed** (*auto simp: interval-eq-iff*)

instance *interval* :: (*{ordered-comm-monoid-add, lattice}*) *ordered-comm-monoid-add*

..

instantiation *interval* :: (*{ordered-ab-group-add}*) *uminus*

begin

lift-definition *uminus-interval*::'a *interval* \Rightarrow 'a *interval* **is** $\lambda(a, b). (-b, -a)$ **by** *auto*

lemma *lower-uminus[simp]*: *lower (- A) = - upper A*

by *transfer auto*

lemma *upper-uminus[simp]*: *upper (- A) = - lower A*

by *transfer auto*

instance **..**

end

instantiation *interval* :: (*{ordered-ab-group-add}*) *minus*

begin

definition *minus-interval*::'a *interval* \Rightarrow 'a *interval* \Rightarrow 'a *interval*

where *minus-interval a b* = *a + - b*

lemma *lower-minus[simp]*: *lower (minus A B) = minus (lower A) (upper B)*

by (*auto simp: minus-interval-def*)

lemma *upper-minus[simp]*: *upper (minus A B) = minus (upper A) (lower B)*

by (*auto simp: minus-interval-def*)

instance **..**

end

instantiation *interval* :: (*{times, linorder}*) *times*

begin

lift-definition *times-interval* :: 'a *interval* \Rightarrow 'a *interval* \Rightarrow 'a *interval*

is $\lambda(a1, a2). \lambda(b1, b2).$

(*let* *x1* = *a1 * b1*; *x2* = *a1 * b2*; *x3* = *a2 * b1*; *x4* = *a2 * b2*

in (*min x1 (min x2 (min x3 x4)), max x1 (max x2 (max x3 x4))*)))

by (auto simp: Let-def intro!: min.coboundedI1 max.coboundedI1)

lemma lower-times:

lower (times A B) = Min {lower A * lower B, lower A * upper B, upper A * lower B, upper A * upper B}

by transfer (auto simp: Let-def)

lemma upper-times:

upper (times A B) = Max {lower A * lower B, lower A * upper B, upper A * lower B, upper A * upper B}

by transfer (auto simp: Let-def)

instance ..

end

lemma interval-eq-set-of-iff: $X = Y \longleftrightarrow \text{set-of } X = \text{set-of } Y$ **for** $X\ Y::'a::\text{order interval}$

by (auto simp: set-of-eq interval-eq-iff)

51.1 Membership

abbreviation (in preorder) in-interval ($\langle \langle \text{notation} = \langle \text{infix } \in_i \rangle \rangle - / \in_i - \rangle$) [51, 51] 50)

where in-interval $x\ X \equiv x \in \text{set-of } X$

lemma in-interval-to-interval[intro!]: $a \in_i \text{interval-of } a$

by (auto simp: set-of-eq)

lemma plus-in-intervalI:

fixes $x\ y::'a::\text{ordered-ab-semigroup-add}$

shows $x \in_i X \implies y \in_i Y \implies x + y \in_i X + Y$

by (simp add: add-mono-thms-linordered-semiring(1) set-of-eq)

lemma connected-set-of[intro, simp]:

connected (set-of X) **for** $X::'a::\text{linear-continuum-topology interval}$

by (auto simp: set-of-eq)

lemma ex-sum-in-interval-lemma: $\exists xa \in \{la .. ua\}. \exists xb \in \{lb .. ub\}. x = xa + xb$

if $la \leq ua\ lb \leq ub\ la + lb \leq x\ x \leq ua + ub$

$ua - la \leq ub - lb$

for $la\ b\ c\ d::'a::\text{linordered-ab-group-add}$

proof –

define wa **where** $wa = ua - la$

define wb **where** $wb = ub - lb$

define w **where** $w = wa + wb$

define d **where** $d = x - la - lb$

define da **where** $da = \max\ 0\ (\min\ wa\ (d - wa))$

define db **where** $db = d - da$

from that have nonneg: $0 \leq wa\ 0 \leq wb\ 0 \leq w\ 0 \leq d\ d \leq w$

```

  by (auto simp add: wa-def wb-def w-def d-def add.commute le-diff-eq)
have 0 ≤ db
  by (auto simp: da-def nonneg db-def intro!: min.coboundedI2)
have x = (la + da) + (lb + db)
  by (simp add: da-def db-def d-def)
moreover
have x - la - ub ≤ da
  using that
  unfolding da-def
  by (intro max.coboundedI2) (auto simp: wa-def d-def diff-le-eq diff-add-eq)
then have db ≤ wb
  by (auto simp: db-def d-def wb-def algebra-simps)
with ⟨0 ≤ db⟩ that nonneg have lb + db ∈ {lb..ub}
  by (auto simp: wb-def algebra-simps)
moreover
have da ≤ wa
  by (auto simp: da-def nonneg)
then have la + da ∈ {la..ua}
  by (auto simp: da-def wa-def algebra-simps)
ultimately show ?thesis
  by force
qed

```

```

lemma ex-sum-in-interval:  $\exists xa \geq la. xa \leq ua \wedge (\exists xb \geq lb. xb \leq ub \wedge x = xa + xb)$ 
  if a:  $la \leq ua$  and b:  $lb \leq ub$  and x:  $la + lb \leq x \leq ua + ub$ 
  for la b c d::'a::linordered-ab-group-add
proof -
  from linear consider  $ua - la \leq ub - lb \mid ub - lb \leq ua - la$ 
  by blast
  then show ?thesis
  proof cases
    case 1
    from ex-sum-in-interval-lemma[OF that 1]
    show ?thesis by auto
  next
    case 2
    from x have  $lb + la \leq x \leq ub + ua$  by (simp-all add: ac-simps)
    from ex-sum-in-interval-lemma[OF b a this 2]
    show ?thesis by auto
  qed
qed

```

```

lemma Icc-plus-Icc:
 $\{a .. b\} + \{c .. d\} = \{a + c .. b + d\}$ 
  if  $a \leq b$   $c \leq d$ 
  for a b c d::'a::linordered-ab-group-add
  using ex-sum-in-interval[OF that]
  by (auto intro: add-mono simp: atLeastAtMost-iff Bex-def set-plus-def)

```

lemma *set-of-plus*:

fixes $A :: 'a :: \text{linordered-ab-group-add interval}$
shows $\text{set-of } (A + B) = \text{set-of } A + \text{set-of } B$
using *Icc-plus-Icc*[*of lower A upper A lower B upper B*]
by (*auto simp: set-of-eq*)

lemma *plus-in-intervalE*:

fixes $xy :: 'a :: \text{linordered-ab-group-add}$
assumes $xy \in_i X + Y$
obtains $x \ y$ **where** $xy = x + y \ x \in_i X \ y \in_i Y$
using *assms*
unfolding *set-of-plus set-plus-def*
by *auto*

lemma *set-of-uminus*: $\text{set-of } (-X) = \{-x \mid x. x \in \text{set-of } X\}$

for $X :: 'a :: \text{ordered-ab-group-add interval}$
by (*auto simp: set-of-eq simp: le-minus-iff minus-le-iff*
intro!: exI[where $x = -x$ for x])

lemma *uminus-in-intervalI*:

fixes $x :: 'a :: \text{ordered-ab-group-add}$
shows $x \in_i X \implies -x \in_i -X$
by (*auto simp: set-of-uminus*)

lemma *uminus-in-intervalD*:

fixes $x :: 'a :: \text{ordered-ab-group-add}$
shows $x \in_i -X \implies -x \in_i X$
by (*auto simp: set-of-uminus*)

lemma *minus-in-intervalI*:

fixes $x \ y :: 'a :: \text{ordered-ab-group-add}$
shows $x \in_i X \implies y \in_i Y \implies x - y \in_i X - Y$
by (*metis diff-conv-add-uminus minus-interval-def plus-in-intervalI uminus-in-intervalI*)

lemma *set-of-minus*: $\text{set-of } (X - Y) = \{x - y \mid x \ y. x \in \text{set-of } X \wedge y \in \text{set-of } Y\}$

for $X \ Y :: 'a :: \text{linordered-ab-group-add interval}$
unfolding *minus-interval-def set-of-plus set-of-uminus set-plus-def*
by *force*

lemma *times-in-intervalI*:

fixes $x \ y :: 'a :: \text{linordered-ring}$
assumes $x \in_i X \ y \in_i Y$
shows $x * y \in_i X * Y$

proof –

define $X1$ **where** $X1 \equiv \text{lower } X$
define $X2$ **where** $X2 \equiv \text{upper } X$
define $Y1$ **where** $Y1 \equiv \text{lower } Y$

```

define Y2 where Y2  $\equiv$  upper Y
from assms have assms:  $X1 \leq x \ x \leq X2 \ Y1 \leq y \ y \leq Y2$ 
  by (auto simp: X1-def X2-def Y1-def Y2-def set-of-eq)
have ( $X1 * Y1 \leq x * y \vee X1 * Y2 \leq x * y \vee X2 * Y1 \leq x * y \vee X2 * Y2 \leq$ 
 $x * y$ )  $\wedge$ 
  ( $X1 * Y1 \geq x * y \vee X1 * Y2 \geq x * y \vee X2 * Y1 \geq x * y \vee X2 * Y2 \geq$ 
 $x * y$ )
proof (cases x 0::'a rule: linorder-cases)
  case x0: less
  show ?thesis
  proof (cases y < 0)
    case y0: True
    from y0 x0 assms have  $x * y \leq X1 * y$  by (intro mult-right-mono-neg, auto)
    also from x0 y0 assms have  $X1 * y \leq X1 * Y1$  by (intro mult-left-mono-neg,
auto)
    finally have 1:  $x * y \leq X1 * Y1$ .
    show ?thesis proof(cases X2  $\leq$  0)
      case True
      with assms have  $X2 * Y2 \leq X2 * y$  by (auto intro: mult-left-mono-neg)
      also from assms y0 have  $\dots \leq x * y$  by (auto intro: mult-right-mono-neg)
      finally have  $X2 * Y2 \leq x * y$ .
      with 1 show ?thesis by auto
    next
      case False
      with assms have  $X2 * Y1 \leq X2 * y$  by (auto intro: mult-left-mono)
      also from assms y0 have  $\dots \leq x * y$  by (auto intro: mult-right-mono-neg)
      finally have  $X2 * Y1 \leq x * y$ .
      with 1 show ?thesis by auto
    qed
  next
    case False
    then have y0:  $y \geq 0$  by auto
    from x0 y0 assms have  $X1 * Y2 \leq x * Y2$  by (intro mult-right-mono, auto)
    also from y0 x0 assms have  $\dots \leq x * y$  by (intro mult-left-mono-neg, auto)
    finally have 1:  $X1 * Y2 \leq x * y$ .
    show ?thesis
    proof(cases X2  $\leq$  0)
      case X2: True
      from assms y0 have  $x * y \leq X2 * y$  by (intro mult-right-mono)
      also from assms X2 have  $\dots \leq X2 * Y1$  by (auto intro: mult-left-mono-neg)
      finally have  $x * y \leq X2 * Y1$ .
      with 1 show ?thesis by auto
    next
      case X2: False
      from assms y0 have  $x * y \leq X2 * y$  by (intro mult-right-mono)
      also from assms X2 have  $\dots \leq X2 * Y2$  by (auto intro: mult-left-mono)
      finally have  $x * y \leq X2 * Y2$ .
      with 1 show ?thesis by auto
    qed
  qed

```

```

qed
next
  case [simp]: equal
  with assms show ?thesis by (cases Y2 ≤ 0, auto intro:mult-sign-intros)
next
  case x0: greater
  show ?thesis
  proof (cases y < 0)
    case y0: True
    from x0 y0 assms have X2 * Y1 ≤ X2 * y by (intro mult-left-mono, auto)
    also from y0 x0 assms have X2 * y ≤ x * y by (intro mult-right-mono-neg,
auto)
    finally have 1: X2 * Y1 ≤ x * y.
    show ?thesis
    proof (cases Y2 ≤ 0)
      case Y2: True
      from x0 assms have x * y ≤ x * Y2 by (auto intro: mult-left-mono)
      also from assms Y2 have ... ≤ X1 * Y2 by (auto intro: mult-right-mono-neg)
      finally have x * y ≤ X1 * Y2.
      with 1 show ?thesis by auto
    next
      case Y2: False
      from x0 assms have x * y ≤ x * Y2 by (auto intro: mult-left-mono)
      also from assms Y2 have ... ≤ X2 * Y2 by (auto intro: mult-right-mono)
      finally have x * y ≤ X2 * Y2.
      with 1 show ?thesis by auto
    qed
  next
    case y0: False
    from x0 y0 assms have x * y ≤ X2 * y by (intro mult-right-mono, auto)
    also from y0 x0 assms have ... ≤ X2 * Y2 by (intro mult-left-mono, auto)
    finally have 1: x * y ≤ X2 * Y2.
    show ?thesis
    proof (cases X1 ≤ 0)
      case True
      with assms have X1 * Y2 ≤ X1 * y by (auto intro: mult-left-mono-neg)
      also from assms y0 have ... ≤ x * y by (auto intro: mult-right-mono)
      finally have X1 * Y2 ≤ x * y.
      with 1 show ?thesis by auto
    next
      case False
      with assms have X1 * Y1 ≤ X1 * y by (auto intro: mult-left-mono)
      also from assms y0 have ... ≤ x * y by (auto intro: mult-right-mono)
      finally have X1 * Y1 ≤ x * y.
      with 1 show ?thesis by auto
    qed
  qed
qed
hence min:min (X1 * Y1) (min (X1 * Y2) (min (X2 * Y1) (X2 * Y2))) ≤ x

```

```

* y
  and max:x * y ≤ max (X1 * Y1) (max (X1 * Y2) (max (X2 * Y1) (X2 *
Y2)))
  by (auto simp: min-le-iff-disj le-max-iff-disj)
  show ?thesis using min max
  by (auto simp: Let-def X1-def X2-def Y1-def Y2-def set-of-eq lower-times up-
per-times)
qed

```

lemma *times-in-intervalE*:

```

  fixes xy :: 'a :: {linorder, real-normed-algebra, linear-continuum-topology}
  — TODO: linear continuum topology is pretty strong
  assumes xy ∈i X * Y
  obtains x y where xy = x * y x ∈i X y ∈i Y
proof —
  let ?mult = λ(x, y). x * y
  let ?XY = set-of X × set-of Y
  have cont: continuous-on ?XY ?mult
  by (auto intro!: tendsto-eq-intros simp: continuous-on-def split-beta')
  have conn: connected (?mult ‘ ?XY)
  by (rule connected-continuous-image[OF cont]) auto
  have lower (X * Y) ∈ ?mult ‘ ?XY upper (X * Y) ∈ ?mult ‘ ?XY
  by (auto simp: set-of-eq lower-times upper-times min-def max-def split: if-splits)
  from connectedD-interval[OF conn this, of xy] assms
  obtain x y where xy = x * y x ∈i X y ∈i Y by (auto simp: set-of-eq)
  then show ?thesis ..

```

qed

thm *times-in-intervalE*[of 1::real]

```

lemma set-of-times: set-of (X * Y) = {x * y | x y. x ∈ set-of X ∧ y ∈ set-of Y}
  for X Y :: 'a :: {linordered-ring, real-normed-algebra, linear-continuum-topology}
interval
  by (auto intro!: times-in-intervalI elim!: times-in-intervalE)

```

instance *interval* :: (linordered-idom) cancel-semigroup-add

proof **qed** (auto simp: interval-eq-iff)

lemma *interval-mul-commute*: $A * B = B * A$ **for** $A B :: 'a :: \text{linordered-idom interval}$

by (simp add: interval-eq-iff lower-times upper-times ac-simps)

lemma *interval-times-zero-right*[simp]: $A * 0 = 0$ **for** $A :: 'a :: \text{linordered-ring interval}$

by (simp add: interval-eq-iff lower-times upper-times ac-simps)

lemma *interval-times-zero-left*[simp]:

$0 * A = 0$ **for** $A :: 'a :: \text{linordered-ring interval}$

by (simp add: interval-eq-iff lower-times upper-times ac-simps)

instantiation *interval* :: ($\{\text{preorder}, \text{one}\}$) one

begin

lift-definition *one-interval*::'a *interval* **is** (1, 1) **by** *auto*

lemma *lower-one[simp]*: *lower 1 = 1*

by *transfer auto*

lemma *upper-one[simp]*: *upper 1 = 1*

by *transfer auto*

instance **proof** **qed**

end

instance *interval* :: (*one*, *preorder*, *linorder*, *times*) *power*

proof **qed**

lemma *set-of-one[simp]*: *set-of (1::'a::{one, order} interval) = {1}*

by (*auto simp: set-of-eq*)

instance *interval* ::

(*linordered-idom*, *real-normed-algebra*, *linear-continuum-topology*) *monoid-mult*

apply *standard*

unfolding *interval-eq-set-of-iff set-of-times*

subgoal

by (*auto simp: interval-eq-set-of-iff set-of-times; metis mult.assoc*)

by *auto*

lemma *one-times-ivl-left[simp]*: *1 * A = A* **for** *A :: 'a::linordered-idom interval*

by (*simp add: interval-eq-iff lower-times upper-times ac-simps min-def max-def*)

lemma *one-times-ivl-right[simp]*: *A * 1 = A* **for** *A :: 'a::linordered-idom interval*

by (*metis interval-mul-commute one-times-ivl-left*)

lemma *set-of-power-mono*: *a[^]n ∈ set-of (A[^]n)* **if** *a ∈ set-of A*

for *a :: 'a::linordered-idom*

using *that*

by (*induction n (auto intro!: times-in-intervalI)*)

lemma *set-of-add-cong*:

set-of (A + B) = set-of (A' + B')

if *set-of A = set-of A' set-of B = set-of B'*

for *A :: 'a::linordered-ab-group-add interval*

unfolding *set-of-plus* **that** **..**

lemma *set-of-add-inc-left*:

set-of (A + B) ⊆ set-of (A' + B)

if *set-of A ⊆ set-of A'*

for *A :: 'a::linordered-ab-group-add interval*

unfolding *set-of-plus* **using** *that* **by** (*auto simp: set-plus-def*)

lemma *set-of-add-inc-right*:

set-of (A + B) ⊆ set-of (A + B')


```

if set-of  $B \subseteq \text{set-of } B'$ 
for  $A :: 'a::\text{linordered-ab-group-add interval}$ 
using set-of-add-inc-left[OF that]
by (simp add: add.commute)

```

```

lemma set-of-add-inc:
  set-of  $(A + B) \subseteq \text{set-of } (A' + B')$ 
if set-of  $A \subseteq \text{set-of } A'$  set-of  $B \subseteq \text{set-of } B'$ 
for  $A :: 'a::\text{linordered-ab-group-add interval}$ 
using set-of-add-inc-left[OF that(1)] set-of-add-inc-right[OF that(2)]
by auto

```

```

lemma set-of-neg-inc:
  set-of  $(-A) \subseteq \text{set-of } (-A')$ 
if set-of  $A \subseteq \text{set-of } A'$ 
for  $A :: 'a::\text{ordered-ab-group-add interval}$ 
using that
unfolding set-of-uminus
by auto

```

```

lemma set-of-sub-inc-left:
  set-of  $(A - B) \subseteq \text{set-of } (A' - B)$ 
if set-of  $A \subseteq \text{set-of } A'$ 
for  $A :: 'a::\text{linordered-ab-group-add interval}$ 
using that
unfolding set-of-minus
by auto

```

```

lemma set-of-sub-inc-right:
  set-of  $(A - B) \subseteq \text{set-of } (A - B')$ 
if set-of  $B \subseteq \text{set-of } B'$ 
for  $A :: 'a::\text{linordered-ab-group-add interval}$ 
using that
unfolding set-of-minus
by auto

```

```

lemma set-of-sub-inc:
  set-of  $(A - B) \subseteq \text{set-of } (A' - B')$ 
if set-of  $A \subseteq \text{set-of } A'$  set-of  $B \subseteq \text{set-of } B'$ 
for  $A :: 'a::\text{linordered-idom interval}$ 
using set-of-sub-inc-left[OF that(1)] set-of-sub-inc-right[OF that(2)]
by auto

```

```

lemma set-of-mul-inc-right:
  set-of  $(A * B) \subseteq \text{set-of } (A * B')$ 
if set-of  $B \subseteq \text{set-of } B'$ 
for  $A :: 'a::\text{linordered-ring interval}$ 
using that
apply transfer

```

```

apply (clarsimp simp add: Let-def)
by (smt (verit, best) linorder-le-cases max.coboundedI1 max.coboundedI2 min.absorb1
min.coboundedI2 mult-left-mono mult-left-mono-neg)

```

```

lemma set-of-distrib-left:
  set-of (B * (A1 + A2))  $\subseteq$  set-of (B * A1 + B * A2)
for A1 :: 'a::linordered-ring interval
apply transfer
apply (clarsimp simp: Let-def distrib-left distrib-right)
apply (intro conjI)
  apply (metis add-mono min.cobounded1 min.left-commute)
  apply (metis add-mono min.cobounded1 min.left-commute)
  apply (metis add-mono min.cobounded1 min.left-commute)
  apply (metis add-mono min.assoc min.cobounded2)
  apply (meson add-mono order.trans max.cobounded1 max.cobounded2)
  apply (meson add-mono order.trans max.cobounded1 max.cobounded2)
  apply (meson add-mono order.trans max.cobounded1 max.cobounded2)
  apply (meson add-mono order.trans max.cobounded1 max.cobounded2)
done

```

```

lemma set-of-distrib-right:
  set-of ((A1 + A2) * B)  $\subseteq$  set-of (A1 * B + A2 * B)
for A1 A2 B :: 'a::{linordered-ring, real-normed-algebra, linear-continuum-topology}
interval
  unfolding set-of-times set-of-plus set-plus-def
  using distrib-right by blast

```

```

lemma set-of-mul-inc-left:
  set-of (A * B)  $\subseteq$  set-of (A' * B)
if set-of A  $\subseteq$  set-of A'
for A :: 'a::{linordered-ring, real-normed-algebra, linear-continuum-topology} in-
terval
  using that
  unfolding set-of-times
  by auto

```

```

lemma set-of-mul-inc:
  set-of (A * B)  $\subseteq$  set-of (A' * B')
if set-of A  $\subseteq$  set-of A' set-of B  $\subseteq$  set-of B'
for A :: 'a::{linordered-ring, real-normed-algebra, linear-continuum-topology} in-
terval
  using that unfolding set-of-times by auto

```

```

lemma set-of-pow-inc:
  set-of (A ^ n)  $\subseteq$  set-of (A' ^ n)
if set-of A  $\subseteq$  set-of A'
for A :: 'a::{linordered-idom, real-normed-algebra, linear-continuum-topology} in-
terval
  using that

```

by (induction n, simp-all add: set-of-mul-inc)

lemma set-of-distrib-right-left:

set-of $((A1 + A2) * (B1 + B2)) \subseteq \text{set-of } (A1 * B1 + A1 * B2 + A2 * B1 + A2 * B2)$

for $A1 :: 'a::\{\text{linordered-idom, real-normed-algebra, linear-continuum-topology}\}$
interval

proof–

have set-of $((A1 + A2) * (B1 + B2)) \subseteq \text{set-of } (A1 * (B1 + B2) + A2 * (B1 + B2))$

by (rule set-of-distrib-right)

also have $\dots \subseteq \text{set-of } ((A1 * B1 + A1 * B2) + A2 * (B1 + B2))$

by (rule set-of-add-inc-left[OF set-of-distrib-left])

also have $\dots \subseteq \text{set-of } ((A1 * B1 + A1 * B2) + (A2 * B1 + A2 * B2))$

by (rule set-of-add-inc-right[OF set-of-distrib-left])

finally show ?thesis

by (simp add: add.assoc)

qed

lemma mult-bounds-enclose-zero1:

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$

$0 \leq \max (la * lb) (\max (la * ub) (\max (lb * ua) (ua * ub)))$

if $la \leq 0$ $0 \leq ua$

for $la\ lb\ ua\ ub:: 'a::\text{linordered-idom}$

subgoal by (metis (no-types, opaque-lifting) that eq-iff min-le-iff-disj mult-zero-left mult-zero-right

zero-le-mult-iff)

subgoal by (metis that le-max-iff-disj mult-zero-right order-refl zero-le-mult-iff)

done

lemma mult-bounds-enclose-zero2:

$\min (la * lb) (\min (la * ub) (\min (lb * ua) (ua * ub))) \leq 0$

$0 \leq \max (la * lb) (\max (la * ub) (\max (lb * ua) (ua * ub)))$

if $lb \leq 0$ $0 \leq ub$

for $la\ lb\ ua\ ub:: 'a::\text{linordered-idom}$

using mult-bounds-enclose-zero1[OF that, of la ua]

by (simp-all add: ac-simps)

lemma set-of-mul-contains-zero:

$0 \in \text{set-of } (A * B)$

if $0 \in \text{set-of } A \vee 0 \in \text{set-of } B$

for $A :: 'a::\text{linordered-idom interval}$

using that

by (auto simp: set-of-eq lower-times upper-times algebra-simps mult-le-0-iff mult-bounds-enclose-zero1 mult-bounds-enclose-zero2)

instance interval :: $(\{\text{linordered-semiring, zero, times}\})$ mult-zero

by (standard; transfer; auto)

lift-definition *min-interval*::'a::linorder interval \Rightarrow 'a interval \Rightarrow 'a interval **is**
 $\lambda(l1, u1). \lambda(l2, u2). (\min l1\ l2, \min u1\ u2)$
by (auto simp: min-def)

lemma *lower-min-interval*[simp]: $\text{lower } (\text{min-interval } x\ y) = \min (\text{lower } x) (\text{lower } y)$
by transfer auto

lemma *upper-min-interval*[simp]: $\text{upper } (\text{min-interval } x\ y) = \min (\text{upper } x) (\text{upper } y)$
by transfer auto

lemma *min-intervalI*:
 $a \in_i A \Longrightarrow b \in_i B \Longrightarrow \min a\ b \in_i \text{min-interval } A\ B$
by (auto simp: set-of-eq min-def)

lift-definition *max-interval*::'a::linorder interval \Rightarrow 'a interval \Rightarrow 'a interval **is**
 $\lambda(l1, u1). \lambda(l2, u2). (\max l1\ l2, \max u1\ u2)$
by (auto simp: max-def)

lemma *lower-max-interval*[simp]: $\text{lower } (\text{max-interval } x\ y) = \max (\text{lower } x) (\text{lower } y)$
by transfer auto

lemma *upper-max-interval*[simp]: $\text{upper } (\text{max-interval } x\ y) = \max (\text{upper } x) (\text{upper } y)$
by transfer auto

lemma *max-intervalI*:
 $a \in_i A \Longrightarrow b \in_i B \Longrightarrow \max a\ b \in_i \text{max-interval } A\ B$
by (auto simp: set-of-eq max-def)

lift-definition *abs-interval*::'a::linordered-idom interval \Rightarrow 'a interval **is**
 $(\lambda(l, u). (\text{if } l < 0 \wedge 0 < u \text{ then } 0 \text{ else } \min |l|\ |u|, \max |l|\ |u|))$
by auto

lemma *lower-abs-interval*[simp]:
 $\text{lower } (\text{abs-interval } x) = (\text{if } \text{lower } x < 0 \wedge 0 < \text{upper } x \text{ then } 0 \text{ else } \min |\text{lower } x| |\text{upper } x|)$
by transfer auto

lemma *upper-abs-interval*[simp]: $\text{upper } (\text{abs-interval } x) = \max |\text{lower } x| |\text{upper } x|$
by transfer auto

lemma *in-abs-intervalI1*:
 $lx < 0 \Longrightarrow 0 < ux \Longrightarrow 0 \leq xa \Longrightarrow xa \leq \max (-\ lx)\ (ux) \Longrightarrow xa \in \text{abs } \{lx..ux\}$
for $xa::'a::linordered-idom$
by (metis abs-minus-cancel abs-of-nonneg atLeastAtMost-iff image-eqI le-less le-max-iff-disj
le-minus-iff neg-le-0-iff-le order-trans)

lemma *in-abs-intervalI2*:
 $\min (|lx|)\ |ux| \leq xa \Longrightarrow xa \leq \max |lx|\ |ux| \Longrightarrow lx \leq ux \Longrightarrow 0 \leq lx \vee ux \leq 0$
 \Longrightarrow
 $xa \in \text{abs } \{lx..ux\}$

```

for  $xa :: 'a :: \text{linordered-idom}$ 
by (force intro: image-eqI[where  $x = -xa$ ] image-eqI[where  $x = xa$ ])

lemma set-of-abs-interval: set-of (abs-interval  $x$ ) = abs ‘ set-of  $x$ 
by (auto simp: set-of-eq not-less intro: in-abs-intervalI1 in-abs-intervalI2 cong
del: image-cong-simp)

fun split-domain :: ( $'a :: \text{preorder}$  interval  $\Rightarrow 'a$  interval list)  $\Rightarrow 'a$  interval list  $\Rightarrow$ 
 $'a$  interval list list
where split-domain split [] = [[]]
| split-domain split (I#Is) = (
  let  $S = \text{split } I$ ;
   $D = \text{split-domain split } Is$ 
  in concat (map ( $\lambda d. \text{map } (\lambda s. s \# d) S$ )  $D$ )
)

context notes [[typedef-overloaded]] begin
lift-definition(code-dt) split-interval:: $'a :: \text{linorder}$  interval  $\Rightarrow 'a \Rightarrow ('a$  interval  $\times$ 
 $'a$  interval)
is  $\lambda(l, u) x. ((\min l x, \max l x), (\min u x, \max u x))$ 
by (auto simp: min-def)
end

lemma split-domain-nonempty:
assumes  $\bigwedge I. \text{split } I \neq []$ 
shows split-domain split  $I \neq []$ 
using last-in-set assms
by (induction  $I$ , auto)

lemma lower-split-interval1: lower (fst (split-interval  $X m$ )) = min (lower  $X$ )  $m$ 
and lower-split-interval2: lower (snd (split-interval  $X m$ )) = min (upper  $X$ )  $m$ 
and upper-split-interval1: upper (fst (split-interval  $X m$ )) = max (lower  $X$ )  $m$ 
and upper-split-interval2: upper (snd (split-interval  $X m$ )) = max (upper  $X$ )  $m$ 
subgoal by transfer auto
subgoal by transfer (auto simp: min.commute)
subgoal by transfer auto
subgoal by transfer auto
done

lemma split-intervalD: split-interval  $X x = (A, B) \Longrightarrow \text{set-of } X \subseteq \text{set-of } A \cup \text{set-of } B$ 
unfolding set-of-eq
by transfer (auto simp: min-def max-def split: if-splits)

instantiation interval :: ( $\{\text{topological-space, preorder}\}$ ) topological-space
begin

definition open-interval-def[code del]: open ( $X :: 'a$  interval set) =
( $\forall x \in X.$ 

```

```

     $\exists A B.$ 
       $\text{open } A \wedge$ 
       $\text{open } B \wedge$ 
       $\text{lower } x \in A \wedge \text{upper } x \in B \wedge \text{Interval } ' (A \times B) \subseteq X$ 

instance
proof
  show  $\text{open } (UNIV :: ('a \text{ interval}) \text{ set})$ 
    unfolding  $\text{open-interval-def}$  by  $\text{auto}$ 
next
  fix  $S T :: ('a \text{ interval}) \text{ set}$ 
  assume  $\text{open } S \text{ open } T$ 
  show  $\text{open } (S \cap T)$ 
    unfolding  $\text{open-interval-def}$ 
  proof ( $\text{safe}$ )
    fix  $x$  assume  $x \in S \wedge x \in T$ 
    from  $\langle x \in S \rangle \langle \text{open } S \rangle$  obtain  $Sl Su$  where  $S$ :
       $\text{open } Sl \text{ open } Su \text{ lower } x \in Sl \text{ upper } x \in Su \text{ Interval } ' (Sl \times Su) \subseteq S$ 
      by ( $\text{auto simp: open-interval-def}$ )
    from  $\langle x \in T \rangle \langle \text{open } T \rangle$  obtain  $Tl Tu$  where  $T$ :
       $\text{open } Tl \text{ open } Tu \text{ lower } x \in Tl \text{ upper } x \in Tu \text{ Interval } ' (Tl \times Tu) \subseteq T$ 
      by ( $\text{auto simp: open-interval-def}$ )

    let  $?L = Sl \cap Tl$  and  $?U = Su \cap Tu$ 
    have  $\text{open } ?L \wedge \text{open } ?U \wedge \text{lower } x \in ?L \wedge \text{upper } x \in ?U \wedge \text{Interval } ' (?L \times$ 
 $?U) \subseteq S \cap T$ 
      using  $S T$  by ( $\text{auto simp add: open-Int}$ )
    then show  $\exists A B. \text{open } A \wedge \text{open } B \wedge \text{lower } x \in A \wedge \text{upper } x \in B \wedge \text{Interval } ' (A \times B) \subseteq S \cap T$ 
      by  $\text{fast}$ 
  qed
qed ( $\text{unfold open-interval-def, fast}$ )

end

```

51.2 Quickcheck

lift-definition $\text{Ivl} :: 'a \Rightarrow 'a :: \text{preorder} \Rightarrow 'a \text{ interval}$ **is** $\lambda a b. (\text{min } a b, b)$
by ($\text{auto simp: min-def}$)

instantiation $\text{interval} :: (\{ \text{exhaustive}, \text{preorder} \}) \text{ exhaustive}$
begin

definition $\text{exhaustive-interval} :: ('a \text{ interval} \Rightarrow (\text{bool} \times \text{term list}) \text{ option})$
 $\Rightarrow \text{natural} \Rightarrow (\text{bool} \times \text{term list}) \text{ option}$
where
 $\text{exhaustive-interval } f d =$
 $\text{Quickcheck-Exhaustive.exhaustive } (\lambda x. \text{Quickcheck-Exhaustive.exhaustive } (\lambda y. f$
 $(\text{Ivl } x y)) d) d$

instance ..

end

context

includes *term-syntax*

begin

definition [*code-unfold*]:

valtermify-interval $x\ y = \text{Code-Evaluation.valtermify } (Ivl::'a::\{\text{preorder}, \text{typerep}\} \Rightarrow -)$
 $\{\cdot\} \ x \ \{\cdot\} \ y$

end

instantiation *interval* :: $(\{\text{full-exhaustive}, \text{preorder}, \text{typerep}\}) \ \text{full-exhaustive}$

begin

definition *full-exhaustive-interval*::

$('a \ \text{interval} \times (\text{unit} \Rightarrow \text{term}) \Rightarrow (\text{bool} \times \text{term list}) \ \text{option})$

$\Rightarrow \text{natural} \Rightarrow (\text{bool} \times \text{term list}) \ \text{option}$ **where**

full-exhaustive-interval $f \ d =$

Quickcheck-Exhaustive.full-exhaustive

$(\lambda x. \ \text{Quickcheck-Exhaustive.full-exhaustive } (\lambda y. \ f \ (\text{valtermify-interval } x \ y)) \ d)$

d

instance ..

end

instantiation *interval* :: $(\{\text{random}, \text{preorder}, \text{typerep}\}) \ \text{random}$

begin

definition *random-interval* ::

natural

$\Rightarrow \text{natural} \times \text{natural}$

$\Rightarrow ('a \ \text{interval} \times (\text{unit} \Rightarrow \text{term})) \times \text{natural} \times \text{natural}$ **where**

random-interval $i =$

scomp (*Quickcheck-Random.random* i)

$(\lambda \text{man. } \text{scomp } (\text{Quickcheck-Random.random } i) \ (\lambda \text{exp. } \text{Pair } (\text{valtermify-interval } \text{man } \text{exp})))$

instance ..

end

lifting-update *interval.lifting*

lifting-forget *interval.lifting*

end

52 Approximate Operations on Intervals of Floating Point Numbers

theory *Interval-Float*

imports

Interval

Float

begin

definition *mid* :: *float interval* \Rightarrow *float*

where *mid i* = (*lower i* + *upper i*) * *Float 1* (-1)

lemma *mid-in-interval*: *mid i* \in_i *i*

using *lower-le-upper*[*of i*]

by (*auto simp: mid-def set-of-eq powr-minus*)

lemma *mid-le*: *lower i* \leq *mid i* *mid i* \leq *upper i*

using *mid-in-interval*

by (*auto simp: set-of-eq*)

definition *centered* :: *float interval* \Rightarrow *float interval*

where *centered i* = *i* - *interval-of* (*mid i*)

definition *split-float-interval* *x* = *split-interval* *x* ((*lower x* + *upper x*) * *Float 1* (-1))

lemma *split-float-intervalD*: *split-float-interval X* = (*A*, *B*) \Longrightarrow *set-of X* \subseteq *set-of A* \cup *set-of B*

by (*auto dest!: split-intervalD simp: split-float-interval-def*)

lemma *split-float-interval-bounds*:

shows

lower-split-float-interval1: *lower* (*fst* (*split-float-interval X*)) = *lower X*

and *lower-split-float-interval2*: *lower* (*snd* (*split-float-interval X*)) = *mid X*

and *upper-split-float-interval1*: *upper* (*fst* (*split-float-interval X*)) = *mid X*

and *upper-split-float-interval2*: *upper* (*snd* (*split-float-interval X*)) = *upper X*

using *mid-le*[*of X*]

by (*auto simp: split-float-interval-def mid-def[symmetric] min-def max-def real-of-float-eq*

lower-split-interval1 lower-split-interval2

upper-split-interval1 upper-split-interval2)

lemmas *float-round-down-le*[*intro*] = *order-trans*[*OF float-round-down*]

and *float-round-up-ge*[*intro*] = *order-trans*[*OF - float-round-up*]

TODO: many of the lemmas should move to theories *Float* or *Approximation* (the latter should be based on type *interval*).

52.1 Intervals with Floating Point Bounds

context includes *interval.lifting* begin

lift-definition *round-interval* :: nat \Rightarrow float interval \Rightarrow float interval
 is $\lambda p. \lambda(l, u). (\text{float-round-down } p \ l, \text{float-round-up } p \ u)$
 by (auto simp: intro!: float-round-down-le float-round-up-le)

lemma *lower-round-ivl[simp]*: lower (round-interval $p \ x$) = float-round-down p (lower x)
 by transfer auto

lemma *upper-round-ivl[simp]*: upper (round-interval $p \ x$) = float-round-up p (upper x)
 by transfer auto

lemma *round-ivl-correct*: set-of $A \subseteq$ set-of (round-interval prec A)
 by (auto simp: set-of-eq float-round-down-le float-round-up-le)

lift-definition *truncate-ivl* :: nat \Rightarrow real interval \Rightarrow real interval
 is $\lambda p. \lambda(l, u). (\text{truncate-down } p \ l, \text{truncate-up } p \ u)$
 by (auto intro!: truncate-down-le truncate-up-le)

lemma *lower-truncate-ivl[simp]*: lower (truncate-ivl $p \ x$) = truncate-down p (lower x)
 by transfer auto

lemma *upper-truncate-ivl[simp]*: upper (truncate-ivl $p \ x$) = truncate-up p (upper x)
 by transfer auto

lemma *truncate-ivl-correct*: set-of $A \subseteq$ set-of (truncate-ivl prec A)
 by (auto simp: set-of-eq intro!: truncate-down-le truncate-up-le)

lift-definition *real-interval*::float interval \Rightarrow real interval
 is $\lambda(l, u). (\text{real-of-float } l, \text{real-of-float } u)$
 by auto

lemma *lower-real-interval[simp]*: lower (real-interval x) = lower x
 by transfer auto

lemma *upper-real-interval[simp]*: upper (real-interval x) = upper x
 by transfer auto

definition *set-of'* $x = (\text{case } x \text{ of None } \Rightarrow \text{UNIV} \mid \text{Some } i \Rightarrow \text{set-of } (\text{real-interval } i))$

lemma *real-interval-min-interval[simp]*:
 real-interval (min-interval $a \ b$) = min-interval (real-interval a) (real-interval b)
 by (auto simp: interval-eq-set-of-iff set-of-eq real-of-float-min)

lemma *real-interval-max-interval[simp]*:
 real-interval (max-interval $a \ b$) = max-interval (real-interval a) (real-interval b)

by (*auto simp: interval-eq-set-of-iff set-of-eq real-of-float-max*)

lemma *in-intervalI*:

$x \in_i X$ **if** $\text{lower } X \leq x \leq \text{upper } X$
using *that* **by** (*auto simp: set-of-eq*)

abbreviation *in-real-interval* ($\langle \langle \text{notation} = \langle \text{infix } \in_r \rangle \rangle - / \in_r - \rangle \rangle$ [51, 51] 50)

where $x \in_r X \equiv x \in_i \text{real-interval } X$

lemma *in-real-intervalI*:

$x \in_r X$ **if** $\text{lower } X \leq x \leq \text{upper } X$ **for** $x::\text{real}$ **and** $X::\text{float interval}$
using *that*
by (*intro in-intervalI*) *auto*

52.2 intros for *real-interval*

lemma *in-round-intervalI*: $x \in_r A \implies x \in_r (\text{round-interval prec } A)$

by (*auto simp: set-of-eq float-round-down-le float-round-up-le*)

lemma *zero-in-float-intervalI*: $0 \in_r 0$

by (*auto simp: set-of-eq*)

lemma *plus-in-float-intervalI*: $a + b \in_r A + B$ **if** $a \in_r A$ $b \in_r B$

using *that*
by (*auto simp: set-of-eq*)

lemma *minus-in-float-intervalI*: $a - b \in_r A - B$ **if** $a \in_r A$ $b \in_r B$

using *that*
by (*auto simp: set-of-eq*)

lemma *uminus-in-float-intervalI*: $-a \in_r -A$ **if** $a \in_r A$

using *that*
by (*auto simp: set-of-eq*)

lemma *real-interval-times*: $\text{real-interval } (A * B) = \text{real-interval } A * \text{real-interval } B$

by (*auto simp: interval-eq-iff lower-times upper-times min-def max-def*)

lemma *times-in-float-intervalI*: $a * b \in_r A * B$ **if** $a \in_r A$ $b \in_r B$

using *times-in-intervalI[OF that]*
by (*auto simp: real-interval-times*)

lemma *real-interval-abs*: $\text{real-interval } (\text{abs-interval } A) = \text{abs-interval } (\text{real-interval } A)$

by (*auto simp: interval-eq-iff min-def max-def*)

lemma *abs-in-float-intervalI*: $\text{abs } a \in_r \text{abs-interval } A$ **if** $a \in_r A$

by (*auto simp: set-of-abs-interval real-interval-abs intro!: imageI that*)

lemma *interval-of*[*intro,simp*]: $x \in_r \text{interval-of } x$
by (*auto simp: set-of-eq*)

lemma *split-float-interval-realD*: $\text{split-float-interval } X = (A, B) \implies x \in_r X \implies x \in_r A \vee x \in_r B$
by (*auto simp: set-of-eq prod-eq-iff split-float-interval-bounds*)

52.3 bounds for lists

lemma *lower-Interval*: $\text{lower } (\text{Interval } x) = \text{fst } x$
and *upper-Interval*: $\text{upper } (\text{Interval } x) = \text{snd } x$
if $\text{fst } x \leq \text{snd } x$
using *that*
by (*auto simp: lower-def upper-def Interval-inverse split-beta'*)

definition *all-in-i* :: $'a::\text{preorder list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool}$
(*infix* $\langle \text{all}'\text{-in}_i \rangle$ 50)
where $x \text{ all-in}_i I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_i I ! i))$

definition *all-in* :: $\text{real list} \Rightarrow \text{float interval list} \Rightarrow \text{bool}$
(*infix* $\langle \text{all}'\text{-in} \rangle$ 50)
where $x \text{ all-in } I = (\text{length } x = \text{length } I \wedge (\forall i < \text{length } I. x ! i \in_r I ! i))$

definition *all-subset* :: $'a::\text{order interval list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool}$
(*infix* $\langle \text{all}'\text{-subset} \rangle$ 50)
where $I \text{ all-subset } J = (\text{length } I = \text{length } J \wedge (\forall i < \text{length } I. \text{set-of } (I!i) \subseteq \text{set-of } (J!i)))$

lemmas [*simp*] = *all-in-def all-subset-def*

lemma *all-subsetD*:
assumes $I \text{ all-subset } J$
assumes $x \text{ all-in } I$
shows $x \text{ all-in } J$
using *assms*
by (*auto simp: set-of-eq; fastforce*)

lemma *round-interval-mono*: $\text{set-of } (\text{round-interval prec } X) \subseteq \text{set-of } (\text{round-interval prec } Y)$
if $\text{set-of } X \subseteq \text{set-of } Y$
using *that*
by *transfer*
(*auto simp: float-round-down.rep-eq float-round-up.rep-eq truncate-down-mono truncate-up-mono*)

lemma *Ivl-simps*[*simp*]: $\text{lower } (\text{Ivl } a \ b) = \min a \ b$ $\text{upper } (\text{Ivl } a \ b) = b$
subgoal by *transfer simp*
subgoal by *transfer simp*
done

lemma *set-of-subset-iff*: $\text{set-of } X \subseteq \text{set-of } Y \longleftrightarrow \text{lower } Y \leq \text{lower } X \wedge \text{upper } X \leq \text{upper } Y$

for $X\ Y :: 'a :: \text{linorder interval}$
by (*auto simp: set-of-eq subset-iff*)

lemma *set-of-subset-iff'*:

$\text{set-of } a \subseteq \text{set-of } (b :: 'a :: \text{linorder interval}) \longleftrightarrow a \leq b$
unfolding *less-eq-interval-def set-of-subset-iff* ..

lemma *bounds-of-interval-eq-lower-upper*:

$\text{bounds-of-interval } \text{ivl} = (\text{lower } \text{ivl}, \text{upper } \text{ivl})$ **if** $\text{lower } \text{ivl} \leq \text{upper } \text{ivl}$
using *that*
by (*auto simp: lower.rep-eq upper.rep-eq*)

lemma *real-interval-Ivl*: $\text{real-interval } (\text{Ivl } a\ b) = \text{Ivl } a\ b$

by *transfer (auto simp: min-def)*

lemma *set-of-mul-contains-real-zero*:

$0 \in_r (A * B)$ **if** $0 \in_r A \vee 0 \in_r B$
using *that set-of-mul-contains-zero[of A B]*
by (*auto simp: set-of-eq*)

fun *subdivide-interval* :: $\text{nat} \Rightarrow \text{float interval} \Rightarrow \text{float interval list}$

where *subdivide-interval* $0\ I = [I]$
 $|$ *subdivide-interval* $(\text{Suc } n)\ I =$
 $\quad \text{let } m = \text{mid } I$
 $\quad \text{in } (\text{subdivide-interval } n\ (\text{Ivl } (\text{lower } I)\ m)) @ (\text{subdivide-interval } n\ (\text{Ivl } m\ (\text{upper } I)))$
 $\quad)$

lemma *subdivide-interval-length*:

shows $\text{length } (\text{subdivide-interval } n\ I) = 2^n$
by(*induction n arbitrary: I, simp-all add: Let-def*)

lemma *lower-le-mid*: $\text{lower } x \leq \text{mid } x$ *real-of-float* $(\text{lower } x) \leq \text{mid } x$

and *mid-le-upper*: $\text{mid } x \leq \text{upper } x$ *real-of-float* $(\text{mid } x) \leq \text{upper } x$

unfolding *mid-def*

subgoal by *transfer (auto simp: powr-neg-one)*

subgoal by *transfer (auto simp: powr-neg-one)*

subgoal by *transfer (auto simp: powr-neg-one)*

subgoal by *transfer (auto simp: powr-neg-one)*

done

lemma *subdivide-interval-correct*:

list-ex $(\lambda i. x \in_r i)$ $(\text{subdivide-interval } n\ I)$ **if** $x \in_r I$ **for** $x :: \text{real}$

using *that*

proof(*induction n arbitrary: x I*)

case 0

```

    then show ?case by simp
next
  case (Suc n)
  from  $\langle x \in_r I \rangle$  consider  $x \in_r \text{Ivl } (\text{lower } I) (\text{mid } I) \mid x \in_r \text{Ivl } (\text{mid } I) (\text{upper } I)$ 
  by (cases  $x \leq \text{real-of-float } (\text{mid } I)$ )
    (auto simp: set-of-eq min-def lower-le-mid mid-le-upper)
  from this[case-names lower upper] show ?case
  by cases (use Suc.IH in  $\langle \text{auto simp: Let-def} \rangle$ )
qed

fun interval-list-union :: 'a::lattice interval list  $\Rightarrow$  'a interval
  where interval-list-union [] = undefined
    | interval-list-union [I] = I
    | interval-list-union (I#Is) = sup I (interval-list-union Is)

lemma interval-list-union-correct:
  assumes  $S \neq []$ 
  assumes  $i < \text{length } S$ 
  shows set-of (S!i)  $\subseteq$  set-of (interval-list-union S)
  using assms
proof(induction S arbitrary: i)
  case (Cons a S i)
  thus ?case
  proof(cases S)
    fix b S'
    assume  $S = b \# S'$ 
    hence  $S \neq []$ 
    by simp
    show ?thesis
    proof(cases i)
      case 0
      show ?thesis
      apply(cases S)
      using interval-union-mono1
      by (auto simp add: 0)
    next
      case (Suc i-prev)
      hence  $i\text{-prev} < \text{length } S$ 
      using Cons(3) by simp

      from Cons(1)[OF  $\langle S \neq [] \rangle$  this] Cons(1)
      have set-of ((a # S) ! i)  $\subseteq$  set-of (interval-list-union S)
      by (simp add:  $\langle i = \text{Suc } i\text{-prev} \rangle$ )
      also have ...  $\subseteq$  set-of (interval-list-union (a # S))
      using  $\langle S \neq [] \rangle$ 
      apply(cases S)
      using interval-union-mono2
      by auto
      finally show ?thesis .
    end
  end
end

```

qed
 qed simp
 qed simp

lemma *split-domain-correct*:

fixes $x :: \text{real list}$
 assumes $x \text{ all-in } I$
 assumes *split-correct*: $\bigwedge x a I. x \in_r I \implies \text{list-ex } (\lambda i :: \text{float interval}. x \in_r i) (\text{split } I)$
 shows $\text{list-ex } (\lambda s. x \text{ all-in } s) (\text{split-domain split } I)$
 using *assms*(1)
proof(*induction I arbitrary: x*)
 case (*Cons I Is x*)
 have $x \neq []$
 using *Cons*(2) **by** *auto*
 obtain $x' xs$ **where** *x-decomp*: $x = x' \# xs$
 using $\langle x \neq [] \rangle \text{list.exhaust}$ **by** *auto*
 hence $x' \in_r I \ xs \text{ all-in } Is$
 using *Cons*(2)
by *auto*
 show ?case
 using *Cons*(1)[*OF* $\langle xs \text{ all-in } Is \rangle$]
 split-correct[*OF* $\langle x' \in_r I \rangle$]
apply (*auto simp add: list-ex-iff set-of-eq*)
by (*smt (verit, ccfv-SIG) One-nat-def Suc-pred* $\langle x \neq [] \rangle \text{le-simps}(3) \text{length-greater-0-conv}$
 $\text{length-tl linorder-not-less list.sel}(3) \text{neq0-conv nth-Cons}' x\text{-decomp}$)
 qed simp

lift-definition(*code-dt*) *inverse-float-interval::nat \Rightarrow float interval \Rightarrow float interval option* **is**

$\lambda \text{prec } (l, u). \text{if } (0 < l \vee u < 0) \text{ then Some (float-divl prec 1 } u, \text{float-divr prec 1 } l) \text{ else None}$
by (*auto intro!: order-trans*[*OF* *float-divl*] *order-trans*[*OF* - *float-divr*]
 simp: divide-simps)

lemma *inverse-float-interval-eq-Some-conv*:

defines $\text{one} \equiv (1 :: \text{float})$

shows

$\text{inverse-float-interval } p \ X = \text{Some } R \longleftrightarrow$
 $(\text{lower } X > 0 \vee \text{upper } X < 0) \wedge$
 $\text{lower } R = \text{float-divl } p \ \text{one } (\text{upper } X) \wedge$
 $\text{upper } R = \text{float-divr } p \ \text{one } (\text{lower } X)$

by *clarsimp* (*transfer fixing: one, force simp: one-def split: if-splits*)

lemma *inverse-float-interval*:

inverse ‘*set-of* (*real-interval* X) \subseteq *set-of* (*real-interval* Y)

if $\text{inverse-float-interval } p \ X = \text{Some } Y$

using *that*

```

apply (clarsimp simp: set-of-eq inverse-float-interval-eq-Some-conv)
by (intro order-trans[OF float-divl] order-trans[OF - float-divr] conjI)
    (auto simp: divide-simps)

```

```

lemma inverse-float-intervalI:
   $x \in_r X \implies \text{inverse } x \in \text{set-of}' (\text{inverse-float-interval } p \ X)$ 
using inverse-float-interval[of p X]
by (auto simp: set-of'-def split: option.splits)

```

```

lemma inverse-float-interval-eqI:  $\text{inverse-float-interval } p \ X = \text{Some } IVL \implies x \in_r X \implies \text{inverse } x \in_r IVL$ 
using inverse-float-intervalI[of x X p]
by (auto simp: set-of'-def)

```

```

lemma real-interval-abs-interval[simp]:
   $\text{real-interval } (\text{abs-interval } x) = \text{abs-interval } (\text{real-interval } x)$ 
by (auto simp: interval-eq-set-of-iff set-of-eq real-of-float-max real-of-float-min)

```

```

lift-definition floor-float-interval::float interval  $\Rightarrow$  float interval is
   $\lambda(l, u). (\text{floor-fl } l, \text{floor-fl } u)$ 
by (auto intro!: floor-mono simp: floor-fl.rep-eq)

```

```

lemma lower-floor-float-interval[simp]:  $\text{lower } (\text{floor-float-interval } x) = \text{floor-fl } (\text{lower } x)$ 
by transfer auto
lemma upper-floor-float-interval[simp]:  $\text{upper } (\text{floor-float-interval } x) = \text{floor-fl } (\text{upper } x)$ 
by transfer auto

```

```

lemma floor-float-intervalI:  $\lfloor x \rfloor \in_r \text{floor-float-interval } X$  if  $x \in_r X$ 
using that by (auto simp: set-of-eq floor-fl-def floor-mono)

```

end

52.4 constants for code generation

```

definition lowerF::float interval  $\Rightarrow$  float where lowerF = lower
definition upperF::float interval  $\Rightarrow$  float where upperF = upper

```

end

53 Immutable Arrays with Code Generation

```

theory IArray
imports Main
begin

```

53.1 Fundamental operations

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Could be extended to other target languages and more operations.

context
begin

datatype *'a iarray* = *IArray 'a list*

qualified primrec *list-of* :: *'a iarray* \Rightarrow *'a list* **where**
list-of (*IArray xs*) = *xs*

qualified definition *of-fun* :: (*nat* \Rightarrow *'a*) \Rightarrow *nat* \Rightarrow *'a iarray* **where**
[*simp*]: *of-fun f n* = *IArray (map f [0..*n*])*

qualified definition *sub* :: *'a iarray* \Rightarrow *nat* \Rightarrow *'a* (**infixl** $\langle !! \rangle$ 100) **where**
[*simp*]: *as !! n* = *IArray.list-of as ! n*

qualified definition *length* :: *'a iarray* \Rightarrow *nat* **where**
[*simp*]: *length as* = *List.length (IArray.list-of as)*

qualified definition *all* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a iarray* \Rightarrow *bool* **where**
[*simp*]: *all p as* \longleftrightarrow ($\forall a \in \text{set } (\text{list-of } as). p a$)

qualified definition *exists* :: (*'a* \Rightarrow *bool*) \Rightarrow *'a iarray* \Rightarrow *bool* **where**
[*simp*]: *exists p as* \longleftrightarrow ($\exists a \in \text{set } (\text{list-of } as). p a$)

lemma *of-fun-nth*:
IArray.of-fun f n !! i = *f i* **if** *i* < *n*
using *that* **by** (*simp add: map-nth*)

end

53.2 Generic code equations

lemma [*code*]:
size (as :: 'a iarray) = *Suc (IArray.length as)*
by (*cases as simp*)

lemma [*code*]:
size-iarray f as = *Suc (size-list f (IArray.list-of as))*
by (*cases as simp*)

lemma [*code*]:
rec-iarray f as = *f (IArray.list-of as)*
by (*cases as simp*)


```

lemma [code]:
  case-iarray f as = f (IArray.list-of as)
  by (cases as) simp

lemma [code]:
  set-iarray as = set (IArray.list-of as)
  by (cases as) auto

lemma [code]:
  map-iarray f as = IArray (map f (IArray.list-of as))
  by (cases as) auto

lemma [code]:
  rel-iarray r as bs = list-all2 r (IArray.list-of as) (IArray.list-of bs)
  by (cases as, cases bs) auto

lemma list-of-code [code]:
  IArray.list-of as = map (λn. as !! n) [0 ..< IArray.length as]
  by (cases as) (simp add: map-nth)

lemma [code]:
  HOL.equal as bs  $\longleftrightarrow$  HOL.equal (IArray.list-of as) (IArray.list-of bs)
  by (cases as, cases bs) (simp add: equal)

lemma [code]:
  IArray.all p = Not  $\circ$  IArray.exists (Not  $\circ$  p)
  by (simp add: fun-eq-iff)

context
  includes term-syntax
begin

lemma [code]:
  Code-Evaluation.term-of (as :: 'a::typerep iarray) =
    Code-Evaluation.Const (STR "IArray.iarray.IArray") (TYPEREP('a list  $\Rightarrow$  'a
    iarray)) <.> (Code-Evaluation.term-of (IArray.list-of as))
  by (subst term-of-anything) rule

end

```

53.3 Auxiliary operations for code generation

```

context
begin

```

qualified primrec *tabulate* :: integer \times (integer \Rightarrow 'a) \Rightarrow 'a iarray **where**
tabulate (n, f) = IArray (map (f \circ integer-of-nat) [0.. nat-of-integer n])

```

lemma [code]:

```

IArray.of-fun $f\ n = \text{IArray.tabulate } (\text{integer-of-nat } n, f \circ \text{nat-of-integer})$
by *simp*

qualified primrec $\text{sub}' :: 'a\ \text{iarray} \times \text{integer} \Rightarrow 'a$ **where**
 $\text{sub}'\ (as, n) = as\ !!\ \text{nat-of-integer } n$

lemma [*code*]:
 $\text{IArray.sub}'\ (\text{IArray } as, n) = as\ !\ \text{nat-of-integer } n$
by *simp*

lemma [*code*]:
 $as\ !!\ n = \text{IArray.sub}'\ (as, \text{integer-of-nat } n)$
by *simp*

qualified definition $\text{length}' :: 'a\ \text{iarray} \Rightarrow \text{integer}$ **where**
[*simp*]: $\text{length}'\ as = \text{integer-of-nat } (\text{List.length } (\text{IArray.list-of } as))$

lemma [*code*]:
 $\text{IArray.length}'\ (\text{IArray } as) = \text{integer-of-nat } (\text{List.length } as)$
by *simp*

lemma [*code*]:
 $\text{IArray.length } as = \text{nat-of-integer } (\text{IArray.length}'\ as)$
by *simp*

qualified definition $\text{exists-upto} :: ('a \Rightarrow \text{bool}) \Rightarrow \text{integer} \Rightarrow 'a\ \text{iarray} \Rightarrow \text{bool}$
where
[*simp*]: $\text{exists-upto } p\ k\ as \longleftrightarrow (\exists l. 0 \leq l \wedge l < k \wedge p\ (\text{sub}'\ (as, l)))$

lemma *exists-upto-of-nat*:
 $\text{exists-upto } p\ (\text{of-nat } n)\ as \longleftrightarrow (\exists m < n. p\ (as\ !!\ m))$
including *integer.lifting* **by** (*simp*, *transfer*)
(*metis nat-int nat-less-iff of-nat-0-le-iff*)

lemma [*code*]:
 $\text{exists-upto } p\ k\ as \longleftrightarrow (\text{if } k \leq 0 \text{ then False else}$
 $\text{let } l = k - 1 \text{ in } p\ (\text{sub}'\ (as, l)) \vee \text{exists-upto } p\ l\ as)$

proof (*cases* $k \geq 1$)

case *False*

then have $\langle k \leq 0 \rangle$

including *integer.lifting* **by** *transfer simp*

then show *?thesis*

by *simp*

next

case *True*

then have less: $k \leq 0 \longleftrightarrow \text{False}$

by *simp*

define n **where** $n = \text{nat-of-integer } (k - 1)$

with *True* **have** $k - 1 = \text{of-nat } n\ k = \text{of-nat } (\text{Suc } n)$

```

    by simp-all
  show ?thesis unfolding less Let-def k(1) unfolding k(2) exists-upto-of-nat
    using less-Suc-eq by auto
qed

```

```

lemma [code]:
  IArray.exists p as  $\longleftrightarrow$  exists-upto p (length' as) as
  including integer.lifting by (simp, transfer)
  (auto, metis in-set-conv-nth less-imp-of-nat-less nat-int of-nat-0-le-iff)

end

```

53.4 Code Generation for SML

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

code-reserved (SML) *Vector*

```

code-printing
  type-constructor iarray  $\rightarrow$  (SML) - Vector.vector
| constant IArray  $\rightarrow$  (SML) Vector.fromList
| constant IArray.all  $\rightarrow$  (SML) Vector.all
| constant IArray.exists  $\rightarrow$  (SML) Vector.exists
| constant IArray.tabulate  $\rightarrow$  (SML) Vector.tabulate
| constant IArray.sub'  $\rightarrow$  (SML) Vector.sub
| constant IArray.length'  $\rightarrow$  (SML) Vector.length

```

53.5 Code Generation for Haskell

We map 'a iarrays in Isabelle/HOL to *Data.Array.IArray.array* in Haskell. Performance mapping to *Data.Array.Unboxed.Array* and *Data.Array.Array* is similar.

```

code-printing
  code-module IArray  $\rightarrow$  (Haskell)  $\langle$ 
module IArray(IArray, tabulate, of-list, sub, length) where {

  import Prelude (Bool(True, False), not, Maybe(Nothing, Just),
    Integer, (+), (-), (<), fromInteger, toInteger, map, seq, ());
  import qualified Prelude;
  import qualified Data.Array.IArray;
  import qualified Data.Array.Base;
  import qualified Data.Ix;

  newtype IArray e = IArray (Data.Array.IArray.Array Integer e);

  tabulate :: (Integer, (Integer -> e)) -> IArray e;

```

```

    tabulate (k, f) = IArray (Data.Array.IArray.array (0, k - 1) (map (\i -> let
fi = f i in fi 'seq' (i, fi)) [0..k - 1]));

```

```

    of-list :: [e] -> IArray e;
    of-list l = IArray (Data.Array.IArray.listArray (0, (toInteger . Prelude.length) l
- 1) l);

```

```

    sub :: (IArray e, Integer) -> e;
    sub (IArray v, i) = v 'Data.Array.Base.unsafeAt' fromInteger i;

```

```

    length :: IArray e -> Integer;
    length (IArray v) = toInteger (Data.Ix.rangeSize (Data.Array.IArray.bounds v));

```

```

}› for type-constructor iarray constant IArray IArray.tabulate IArray.sub' IAr-
ray.length'

```

```

code-reserved (Haskell) IArray-Impl

```

```

code-printing

```

```

    type-constructor iarray -> (Haskell) IArray.IArray -
| constant IArray -> (Haskell) IArray.of'-list
| constant IArray.tabulate -> (Haskell) IArray.tabulate
| constant IArray.sub' -> (Haskell) IArray.sub
| constant IArray.length' -> (Haskell) IArray.length

```

```

end

```

54 Definition of Landau symbols

```

theory Landau-Symbols

```

```

imports

```

```

    Complex-Main

```

```

begin

```

```

lemma eventually-subst':

```

```

    eventually (λx. f x = g x) F ==> eventually (λx. P x (f x)) F = eventually (λx.
P x (g x)) F

```

```

    by (rule eventually-subst, erule eventually-rev-mp) simp

```

54.1 Definition of Landau symbols

Our Landau symbols are sign-oblivious, i.e. any function always has the same growth as its absolute. This has the advantage of making some cancelling rules for sums nicer, but introduces some problems in other places. Nevertheless, we found this definition more convenient to work with.

```

definition bigo :: 'a filter => ('a => ('b :: real-normed-field)) => ('a => 'b) set
  (⟨(⟨indent=1 notation=⟨mixfix bigo⟩⟩ O[-]'(-'))⟩)

```

```

  where bigo F g = {f. (∃ c>0. eventually (λx. norm (f x) ≤ c * norm (g x)) F)}

```

definition $\text{smallo} :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
 $(\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix smallo} \rangle o[-]'(-)) \rangle)$
where $\text{smallo } F g = \{f. (\forall c>0. \text{eventually } (\lambda x. \text{norm } (f x) \leq c * \text{norm } (g x)) F)\}$

definition $\text{bigomega} :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
 $(\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix bigomega} \rangle \Omega[-]'(-)) \rangle)$
where $\text{bigomega } F g = \{f. (\exists c>0. \text{eventually } (\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x)) F)\}$

definition $\text{smallomega} :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
 $(\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix smallomega} \rangle \omega[-]'(-)) \rangle)$
where $\text{smallomega } F g = \{f. (\forall c>0. \text{eventually } (\lambda x. \text{norm } (f x) \geq c * \text{norm } (g x)) F)\}$

definition $\text{bigtheta} :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
 $(\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix bigtheta} \rangle \Theta[-]'(-)) \rangle)$
where $\text{bigtheta } F g = \text{bigo } F g \cap \text{bigomega } F g$

abbreviation $\text{bigo-at-top } (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigo} \rangle O'(-)) \rangle)$
where $O(g) \equiv \text{bigo at-top } g$

abbreviation $\text{smallo-at-top } (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{mixfix smallo} \rangle o'(-)) \rangle)$
where $o(g) \equiv \text{smallo at-top } g$

abbreviation $\text{bigomega-at-top } (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigomega} \rangle \Omega'(-)) \rangle)$
where $\Omega(g) \equiv \text{bigomega at-top } g$

abbreviation $\text{smallomega-at-top } (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{mixfix smallomega} \rangle \omega'(-)) \rangle)$
where $\omega(g) \equiv \text{smallomega at-top } g$

abbreviation $\text{bigtheta-at-top } (\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{mixfix bigtheta} \rangle \Theta'(-)) \rangle)$
where $\Theta(g) \equiv \text{bigtheta at-top } g$

The following is a set of properties that all Landau symbols satisfy.

named-theorems *landau-divide-simps*

locale *landau-symbol* =

fixes $L :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
and $L' :: 'c \text{ filter} \Rightarrow ('c \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('c \Rightarrow 'b) \text{ set}$
and $Lr :: 'a \text{ filter} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real}) \text{ set}$
assumes *bot'*: $L \text{ bot } f = \text{UNIV}$
assumes *filter-mono'*: $F1 \leq F2 \implies L F2 f \subseteq L F1 f$
assumes *in-filtermap-iff*:
 $f' \in L (\text{filtermap } h' F') g' \longleftrightarrow (\lambda x. f' (h' x)) \in L' F' (\lambda x. g' (h' x))$
assumes *filtercomap*:
 $f' \in L F'' g' \implies (\lambda x. f' (h' x)) \in L' (\text{filtercomap } h' F'') (\lambda x. g' (h' x))$

assumes *sup*: $f \in L F1\ g \implies f \in L F2\ g \implies f \in L (sup\ F1\ F2)\ g$
assumes *in-cong*: *eventually* $(\lambda x. f\ x = g\ x)\ F \implies f \in L\ F\ (h) \longleftrightarrow g \in L\ F\ (h)$
assumes *cong*: *eventually* $(\lambda x. f\ x = g\ x)\ F \implies L\ F\ (f) = L\ F\ (g)$
assumes *cong-bigtheta*: $f \in \Theta[F](g) \implies L\ F\ (f) = L\ F\ (g)$
assumes *in-cong-bigtheta*: $f \in \Theta[F](g) \implies f \in L\ F\ (h) \longleftrightarrow g \in L\ F\ (h)$
assumes *cmult* [*simp*]: $c \neq 0 \implies L\ F\ (\lambda x. c * f\ x) = L\ F\ (f)$
assumes *cmult-in-iff* [*simp*]: $c \neq 0 \implies (\lambda x. c * f\ x) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
assumes *mult-left* [*simp*]: $f \in L\ F\ (g) \implies (\lambda x. h\ x * f\ x) \in L\ F\ (\lambda x. h\ x * g\ x)$
assumes *inverse*: *eventually* $(\lambda x. f\ x \neq 0)\ F \implies \text{eventually } (\lambda x. g\ x \neq 0)\ F$
 \implies
 $f \in L\ F\ (g) \implies (\lambda x. \text{inverse}\ (g\ x)) \in L\ F\ (\lambda x. \text{inverse}\ (f\ x))$
assumes *subsetI*: $f \in L\ F\ (g) \implies L\ F\ (f) \subseteq L\ F\ (g)$
assumes *plus-subset1*: $f \in o[F](g) \implies L\ F\ (g) \subseteq L\ F\ (\lambda x. f\ x + g\ x)$
assumes *trans*: $f \in L\ F\ (g) \implies g \in L\ F\ (h) \implies f \in L\ F\ (h)$
assumes *compose*: $f \in L\ F\ (g) \implies \text{filterlim}\ h'\ F\ G \implies (\lambda x. f\ (h'\ x)) \in L'\ G\ (\lambda x. g\ (h'\ x))$
assumes *norm-iff* [*simp*]: $(\lambda x. \text{norm}\ (f\ x)) \in Lr\ F\ (\lambda x. \text{norm}\ (g\ x)) \longleftrightarrow f \in L\ F\ (g)$
assumes *abs* [*simp*]: $Lr\ Fr\ (\lambda x. |fr\ x|) = Lr\ Fr\ fr$
assumes *abs-in-iff* [*simp*]: $(\lambda x. |fr\ x|) \in Lr\ Fr\ gr \longleftrightarrow fr \in Lr\ Fr\ gr$
begin
lemma *bot* [*simp*]: $f \in L\ bot\ g\ \text{by}\ (simp\ add:\ bot')$
lemma *filter-mono*: $F1 \leq F2 \implies f \in L\ F2\ g \implies f \in L\ F1\ g$
using *filter-mono'*[*of F1 F2*] **by** *blast*
lemma *cong-ex*:
eventually $(\lambda x. f1\ x = f2\ x)\ F \implies \text{eventually } (\lambda x. g1\ x = g2\ x)\ F \implies$
 $f1 \in L\ F\ (g1) \longleftrightarrow f2 \in L\ F\ (g2)$
by (*subst cong, assumption, subst in-cong, assumption, rule refl*)
lemma *cong-ex-bigtheta*:
 $f1 \in \Theta[F](f2) \implies g1 \in \Theta[F](g2) \implies f1 \in L\ F\ (g1) \longleftrightarrow f2 \in L\ F\ (g2)$
by (*subst cong-bigtheta, assumption, subst in-cong-bigtheta, assumption, rule refl*)
lemma *bigtheta-trans1*:
 $f \in L\ F\ (g) \implies g \in \Theta[F](h) \implies f \in L\ F\ (h)$
by (*subst cong-bigtheta[symmetric]*)
lemma *bigtheta-trans2*:
 $f \in \Theta[F](g) \implies g \in L\ F\ (h) \implies f \in L\ F\ (h)$
by (*subst in-cong-bigtheta*)
lemma *cmult'* [*simp*]: $c \neq 0 \implies L\ F\ (\lambda x. f\ x * c) = L\ F\ (f)$
by (*subst mult.commute*) (*rule cmult*)
lemma *cmult-in-iff'* [*simp*]: $c \neq 0 \implies (\lambda x. f\ x * c) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$

by (*subst mult.commute*) (*rule cmult-in-iff*)

lemma *cdiv [simp]*: $c \neq 0 \implies L\ F\ (\lambda x. f\ x\ /\ c) = L\ F\ (f)$
using *cmult'[of inverse c F f]* **by** (*simp add: field-simps*)

lemma *cdiv-in-iff' [simp]*: $c \neq 0 \implies (\lambda x. f\ x\ /\ c) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
using *cmult-in-iff'[of inverse c f]* **by** (*simp add: field-simps*)

lemma *uminus [simp]*: $L\ F\ (\lambda x. -g\ x) = L\ F\ (g)$ **using** *cmult[of -1]* **by** *simp*

lemma *uminus-in-iff [simp]*: $(\lambda x. -f\ x) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
using *cmult-in-iff[of -1]* **by** *simp*

lemma *const*: $c \neq 0 \implies L\ F\ (\lambda-. c) = L\ F\ (\lambda-. 1)$
by (*subst (2) cmult[symmetric]*) *simp-all*

lemma *const' [simp]*: *NO-MATCH* $1\ c \implies c \neq 0 \implies L\ F\ (\lambda-. c) = L\ F\ (\lambda-. 1)$
by (*rule const*)

lemma *const-in-iff*: $c \neq 0 \implies (\lambda-. c) \in L\ F\ (f) \longleftrightarrow (\lambda-. 1) \in L\ F\ (f)$
using *cmult-in-iff'[of c λ-. 1]* **by** *simp*

lemma *const-in-iff' [simp]*: *NO-MATCH* $1\ c \implies c \neq 0 \implies (\lambda-. c) \in L\ F\ (f) \longleftrightarrow$
 $(\lambda-. 1) \in L\ F\ (f)$
by (*rule const-in-iff*)

lemma *plus-subset2*: $g \in o[F](f) \implies L\ F\ (f) \subseteq L\ F\ (\lambda x. f\ x + g\ x)$
by (*subst add.commute*) (*rule plus-subset1*)

lemma *mult-right [simp]*: $f \in L\ F\ (g) \implies (\lambda x. f\ x * h\ x) \in L\ F\ (\lambda x. g\ x * h\ x)$
using *mult-left* **by** (*simp add: mult.commute*)

lemma *mult*: $f1 \in L\ F\ (g1) \implies f2 \in L\ F\ (g2) \implies (\lambda x. f1\ x * f2\ x) \in L\ F\ (\lambda x.$
 $g1\ x * g2\ x)$
by (*rule trans, erule mult-left, erule mult-right*)

lemma *inverse-cancel*:
assumes *eventually* $(\lambda x. f\ x \neq 0)\ F$
assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$
shows $(\lambda x. \text{inverse}\ (f\ x)) \in L\ F\ (\lambda x. \text{inverse}\ (g\ x)) \longleftrightarrow g \in L\ F\ (f)$
proof
assume $(\lambda x. \text{inverse}\ (f\ x)) \in L\ F\ (\lambda x. \text{inverse}\ (g\ x))$
from *inverse[OF - - this]* *assms* **show** $g \in L\ F\ (f)$ **by** *simp*
qed (*intro inverse assms*)

lemma *divide-right*:
assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$
assumes $f \in L\ F\ (g)$

shows $(\lambda x. f x / h x) \in L F (\lambda x. g x / h x)$
by (*subst* (1 2) *divide-inverse*) (*intro mult-right inverse assms*)

lemma *divide-right-iff*:

assumes *eventually* $(\lambda x. h x \neq 0) F$
shows $(\lambda x. f x / h x) \in L F (\lambda x. g x / h x) \longleftrightarrow f \in L F (g)$

proof

assume $(\lambda x. f x / h x) \in L F (\lambda x. g x / h x)$
from *mult-right*[*OF this, of h*] *assms* **show** $f \in L F (g)$
by (*subst* (*asm*) *cong-ex*[*of - f F - g*]) (*auto elim!*: *eventually-mono*)
qed (*simp add: divide-right assms*)

lemma *divide-left*:

assumes *eventually* $(\lambda x. f x \neq 0) F$
assumes *eventually* $(\lambda x. g x \neq 0) F$
assumes $g \in L F (f)$
shows $(\lambda x. h x / f x) \in L F (\lambda x. h x / g x)$
by (*subst* (1 2) *divide-inverse*) (*intro mult-left inverse assms*)

lemma *divide-left-iff*:

assumes *eventually* $(\lambda x. f x \neq 0) F$
assumes *eventually* $(\lambda x. g x \neq 0) F$
assumes *eventually* $(\lambda x. h x \neq 0) F$
shows $(\lambda x. h x / f x) \in L F (\lambda x. h x / g x) \longleftrightarrow g \in L F (f)$

proof

assume *A*: $(\lambda x. h x / f x) \in L F (\lambda x. h x / g x)$
from *assms* **have** *B*: *eventually* $(\lambda x. h x / f x / h x = \text{inverse } (f x)) F$
by *eventually-elim* (*simp add: divide-inverse*)
from *assms* **have** *C*: *eventually* $(\lambda x. h x / g x / h x = \text{inverse } (g x)) F$
by *eventually-elim* (*simp add: divide-inverse*)
from *divide-right*[*OF assms*(3) *A*] *assms* **show** $g \in L F (f)$
by (*subst* (*asm*) *cong-ex*[*OF B C*]) (*simp add: inverse-cancel*)
qed (*simp add: divide-left assms*)

lemma *divide*:

assumes *eventually* $(\lambda x. g1 x \neq 0) F$
assumes *eventually* $(\lambda x. g2 x \neq 0) F$
assumes $f1 \in L F (f2) \ g2 \in L F (g1)$
shows $(\lambda x. f1 x / g1 x) \in L F (\lambda x. f2 x / g2 x)$
by (*subst* (1 2) *divide-inverse*) (*intro mult inverse assms*)

lemma *divide-eq1*:

assumes *eventually* $(\lambda x. h x \neq 0) F$
shows $f \in L F (\lambda x. g x / h x) \longleftrightarrow (\lambda x. f x * h x) \in L F (g)$

proof—

have $f \in L F (\lambda x. g x / h x) \longleftrightarrow (\lambda x. f x * h x / h x) \in L F (\lambda x. g x / h x)$
using *assms* **by** (*intro in-cong*) (*auto elim: eventually-mono*)
thus *?thesis* **by** (*simp only: divide-right-iff assms*)

qed

lemma *divide-eq2*:

assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$

shows $(\lambda x. f\ x / h\ x) \in L\ F\ (\lambda x. g\ x) \longleftrightarrow f \in L\ F\ (\lambda x. g\ x * h\ x)$

proof–

have $L\ F\ (\lambda x. g\ x) = L\ F\ (\lambda x. g\ x * h\ x / h\ x)$

using *assms* **by** (*intro cong*) (*auto elim: eventually-mono*)

thus *?thesis* **by** (*simp only: divide-right-iff assms*)

qed

lemma *inverse-eq1*:

assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$

shows $f \in L\ F\ (\lambda x. \text{inverse}\ (g\ x)) \longleftrightarrow (\lambda x. f\ x * g\ x) \in L\ F\ (\lambda\cdot. 1)$

using *divide-eq1* [*of g F f λ·. 1*] **by** (*simp add: divide-inverse assms*)

lemma *inverse-eq2*:

assumes *eventually* $(\lambda x. f\ x \neq 0)\ F$

shows $(\lambda x. \text{inverse}\ (f\ x)) \in L\ F\ (g) \longleftrightarrow (\lambda x. 1) \in L\ F\ (\lambda x. f\ x * g\ x)$

using *divide-eq2* [*of f F λ·. 1 g*] **by** (*simp add: divide-inverse assms mult-ac*)

lemma *inverse-flip*:

assumes *eventually* $(\lambda x. g\ x \neq 0)\ F$

assumes *eventually* $(\lambda x. h\ x \neq 0)\ F$

assumes $(\lambda x. \text{inverse}\ (g\ x)) \in L\ F\ (h)$

shows $(\lambda x. \text{inverse}\ (h\ x)) \in L\ F\ (g)$

using *assms* **by** (*simp add: divide-eq1 divide-eq2 inverse-eq-divide mult.commute*)

lemma *lift-trans*:

assumes $f \in L\ F\ (g)$

assumes $(\lambda x. t\ x\ (g\ x)) \in L\ F\ (h)$

assumes $\bigwedge f\ g. f \in L\ F\ (g) \implies (\lambda x. t\ x\ (f\ x)) \in L\ F\ (\lambda x. t\ x\ (g\ x))$

shows $(\lambda x. t\ x\ (f\ x)) \in L\ F\ (h)$

by (*rule trans* [*OF assms(3)*] [*OF assms(1)*] *assms(2)*])

lemma *lift-trans'*:

assumes $f \in L\ F\ (\lambda x. t\ x\ (g\ x))$

assumes $g \in L\ F\ (h)$

assumes $\bigwedge g\ h. g \in L\ F\ (h) \implies (\lambda x. t\ x\ (g\ x)) \in L\ F\ (\lambda x. t\ x\ (h\ x))$

shows $f \in L\ F\ (\lambda x. t\ x\ (h\ x))$

by (*rule trans* [*OF assms(1)*] *assms(3)*] [*OF assms(2)*])

lemma *lift-trans-bigtheta*:

assumes $f \in L\ F\ (g)$

assumes $(\lambda x. t\ x\ (g\ x)) \in \Theta[F](h)$

assumes $\bigwedge f\ g. f \in L\ F\ (g) \implies (\lambda x. t\ x\ (f\ x)) \in L\ F\ (\lambda x. t\ x\ (g\ x))$

shows $(\lambda x. t\ x\ (f\ x)) \in L\ F\ (h)$

using *cong-bigtheta* [*OF assms(2)*] *assms(3)*] [*OF assms(1)*] **by** *simp*

lemma *lift-trans-bigtheta'*:

assumes $f \in L F (\lambda x. t x (g x))$
assumes $g \in \Theta[F](h)$
assumes $\bigwedge g h. g \in \Theta[F](h) \implies (\lambda x. t x (g x)) \in \Theta[F](\lambda x. t x (h x))$
shows $f \in L F (\lambda x. t x (h x))$
using *cong-bigtheta*[*OF assms*(3)[*OF assms*(2)]] *assms*(1) **by** *simp*

lemma (*in landau-symbol*) *mult-in-1*:
assumes $f \in L F (\lambda-. 1) g \in L F (\lambda-. 1)$
shows $(\lambda x. f x * g x) \in L F (\lambda-. 1)$
using *mult*[*OF assms*] **by** *simp*

lemma (*in landau-symbol*) *of-real-cancel*:
 $(\lambda x. \text{of-real } (f x)) \in L F (\lambda x. \text{of-real } (g x)) \implies f \in Lr F g$
by (*subst (asm) norm-iff [symmetric]*, *subst (asm) (1 2) norm-of-real simp-all*)

lemma (*in landau-symbol*) *of-real-iff*:
 $(\lambda x. \text{of-real } (f x)) \in L F (\lambda x. \text{of-real } (g x)) \longleftrightarrow f \in Lr F g$
by (*subst norm-iff [symmetric]*, *subst (1 2) norm-of-real simp-all*)

lemmas [*landau-divide-simps*] =
inverse-cancel divide-left-iff divide-eq1 divide-eq2 inverse-eq1 inverse-eq2

end

The symbols O and o and Ω and ω are dual, so for many rules, replacing O with Ω , o with ω , and \leq with \geq in a theorem yields another valid theorem. The following locale captures this fact.

locale *landau-pair* =
fixes $L l :: 'a \text{ filter} \Rightarrow ('a \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('a \Rightarrow 'b) \text{ set}$
fixes $L' l' :: 'c \text{ filter} \Rightarrow ('c \Rightarrow ('b :: \text{real-normed-field})) \Rightarrow ('c \Rightarrow 'b) \text{ set}$
fixes $Lr lr :: 'a \text{ filter} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real}) \text{ set}$
and $R :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}$
assumes $L\text{-def}: L F g = \{f. \exists c>0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g x))) F\}$
and $l\text{-def}: l F g = \{f. \forall c>0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g x))) F\}$
and $L'\text{-def}: L' F' g' = \{f. \exists c>0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g' x))) F'\}$
and $l'\text{-def}: l' F' g' = \{f. \forall c>0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g' x))) F'\}$
and $Lr\text{-def}: Lr F'' g'' = \{f. \exists c>0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g'' x))) F''\}$
and $lr\text{-def}: lr F'' g'' = \{f. \forall c>0. \text{eventually } (\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g'' x))) F''\}$
and $R: R = (\leq) \vee R = (\geq)$

interpretation *landau-o*:

landau-pair bigo smallo bigo smallo bigo smallo (\leq)
by *unfold-locales (auto simp: bigo-def smallo-def intro!: ext)*

interpretation *landau-omega*:

landau-pair bigomega smallomega bigomega smallomega bigomega smallomega
 (\geq)
 by *unfold-locales (auto simp: bigomega-def smallomega-def intro!: ext)*

context *landau-pair*
begin

lemmas *R-E = disjE [OF R, case-names le ge]*

lemma *bigI*:

*c > 0 \implies eventually $(\lambda x. R (norm (f x)) (c * norm (g x))) F \implies f \in L F (g)$*
 unfolding *L-def* by *blast*

lemma *bigE*:

assumes *f $\in L F (g)$*
obtains *c where c > 0 eventually $(\lambda x. R (norm (f x)) (c * (norm (g x)))) F$*
using *assms* **unfolding** *L-def* **by** *blast*

lemma *smallI*:

$(\bigwedge c. c > 0 \implies \text{eventually } (\lambda x. R (norm (f x)) (c * (norm (g x)))) F) \implies f \in$
 $l F (g)$
 unfolding *l-def* **by** *blast*

lemma *smallD*:

*f $\in l F (g) \implies c > 0 \implies \text{eventually } (\lambda x. R (norm (f x)) (c * (norm (g x)))) F$*
 unfolding *l-def* **by** *blast*

lemma *bigE-nonneg-real*:

assumes *f $\in Lr F (g)$ eventually $(\lambda x. f x \geq 0) F$*
obtains *c where c > 0 eventually $(\lambda x. R (f x) (c * |g x|)) F$*

proof–

from *assms(1)* **obtain** *c where c: c > 0 eventually $(\lambda x. R (norm (f x)) (c * norm (g x))) F$*

by *(auto simp: Lr-def)*

from *c(2) assms(2)* **have** *eventually $(\lambda x. R (f x) (c * |g x|)) F$*

by *eventually-elim simp*

from *c(1)* **and this** **show** *?thesis* **by** *(rule that)*

qed

lemma *smallD-nonneg-real*:

assumes *f $\in lr F (g)$ eventually $(\lambda x. f x \geq 0) F$ c > 0*

shows *eventually $(\lambda x. R (f x) (c * |g x|)) F$*

using *assms* **by** *(auto simp: lr-def dest!: spec[of - c] elim: eventually-elim2)*

lemma *small-imp-big*: *f $\in l F (g) \implies f \in L F (g)$*

by *(rule bigI[OF - smallD, of 1]) simp-all*

```

lemma small-subset-big:  $l\ F\ (g) \subseteq L\ F\ (g)$ 
  using small-imp-big by blast

lemma R-refl [simp]:  $R\ x\ x$  using R by auto

lemma R-linear:  $\neg R\ x\ y \implies R\ y\ x$ 
  using R by auto

lemma R-trans [trans]:  $R\ a\ b \implies R\ b\ c \implies R\ a\ c$ 
  using R by auto

lemma R-mult-left-mono:  $R\ a\ b \implies c \geq 0 \implies R\ (c*a)\ (c*b)$ 
  using R by (auto simp: mult-left-mono)

lemma R-mult-right-mono:  $R\ a\ b \implies c \geq 0 \implies R\ (a*c)\ (b*c)$ 
  using R by (auto simp: mult-right-mono)

lemma big-trans:
  assumes  $f \in L\ F\ (g)\ g \in L\ F\ (h)$ 
  shows  $f \in L\ F\ (h)$ 
proof–
  from assms obtain  $c\ d$  where  $0 < c\ 0 < d$ 
    and  $\forall_F\ x\ in\ F.\ R\ (norm\ (f\ x))\ (c * norm\ (g\ x))$ 
       $\forall_F\ x\ in\ F.\ R\ (norm\ (g\ x))\ (d * norm\ (h\ x))$ 
    by (elim bigE)
  from  $\ast$  have eventually  $(\lambda x.\ R\ (norm\ (f\ x))\ (c * d * (norm\ (h\ x))))\ F$ 
proof eventually-elim
  fix  $x$  assume  $R\ (norm\ (f\ x))\ (c * (norm\ (g\ x)))$ 
  also assume  $R\ (norm\ (g\ x))\ (d * (norm\ (h\ x)))$ 
  with  $\langle 0 < c \rangle$  have  $R\ (c * (norm\ (g\ x)))\ (c * (d * (norm\ (h\ x))))$ 
    by (intro R-mult-left-mono simp-all)
  finally show  $R\ (norm\ (f\ x))\ (c * d * (norm\ (h\ x)))$ 
    by (simp add: algebra-simps)
  qed
  with  $\ast$  show ?thesis by (intro bigI[of c*d] simp-all)
qed

lemma big-small-trans:
  assumes  $f \in L\ F\ (g)\ g \in l\ F\ (h)$ 
  shows  $f \in l\ F\ (h)$ 
proof (rule smallI)
  fix  $c :: real$  assume  $c: c > 0$ 
  from assms(1) obtain  $d$  where  $d: d > 0$  and  $\forall_F\ x\ in\ F.\ R\ (norm\ (f\ x))\ (d$ 
 $*\ norm\ (g\ x))$ 
    by (elim bigE)
  from assms(2)  $c\ d$  have eventually  $(\lambda x.\ R\ (norm\ (g\ x))\ (c * inverse\ d * norm$ 
 $(h\ x)))\ F$ 
    by (intro smallD simp-all)

```

```

with * show eventually  $(\lambda x. R (norm (f x)) (c * (norm (h x)))) F$ 
proof eventually-elim
  case (elim x)
  show ?case
  by (use elim(1) in ‹rule R-trans›) (use elim(2) R d in ‹auto simp: field-simps›)
qed
qed

```

```

lemma small-big-trans:
  assumes  $f \in l F (g) \ g \in L F (h)$ 
  shows  $f \in l F (h)$ 
proof (rule smallI)
  fix c :: real assume c:  $c > 0$ 
  from assms(2) obtain d where  $d > 0$  and *:  $\forall_F x \text{ in } F. R (norm (g x)) (d$ 
  * norm (h x))
  by (elim bigE)
  from assms(1) c d have eventually  $(\lambda x. R (norm (f x)) (c * inverse d * norm$ 
  (g x))) F
  by (intro smallD) simp-all
  with * show eventually  $(\lambda x. R (norm (f x)) (c * norm (h x))) F$ 
  by eventually-elim (rotate-tac 2, erule R-trans, insert R c d, auto simp:
  field-simps)
qed

```

```

lemma small-trans:
   $f \in l F (g) \implies g \in l F (h) \implies f \in l F (h)$ 
  by (rule big-small-trans[OF small-imp-big])

```

```

lemma small-big-trans':
   $f \in l F (g) \implies g \in L F (h) \implies f \in L F (h)$ 
  by (rule small-imp-big[OF small-big-trans])

```

```

lemma big-small-trans':
   $f \in L F (g) \implies g \in l F (h) \implies f \in L F (h)$ 
  by (rule small-imp-big[OF big-small-trans])

```

```

lemma big-subsetI [intro]:  $f \in L F (g) \implies L F (f) \subseteq L F (g)$ 
  by (intro subsetI) (drule (1) big-trans)

```

```

lemma small-subsetI [intro]:  $f \in L F (g) \implies l F (f) \subseteq l F (g)$ 
  by (intro subsetI) (drule (1) small-big-trans)

```

```

lemma big-refl [simp]:  $f \in L F (f)$ 
  by (rule bigI[of 1]) simp-all

```

```

lemma small-refl-iff:  $f \in l F (f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0) F$ 
proof (rule iffI[OF - smallI])
  assume f:  $f \in l F f$ 
  have  $(1/2::real) > 0 \ (2::real) > 0$  by simp-all

```

```

from smallD[OF f this(1)] smallD[OF f this(2)]
  show eventually ( $\lambda x. f\ x = 0$ ) F by eventually-elim (insert R, auto)
next
  fix c :: real assume c > 0 eventually ( $\lambda x. f\ x = 0$ ) F
  from this(2) show eventually ( $\lambda x. R\ (\text{norm}\ (f\ x))\ (c * \text{norm}\ (f\ x)))$  F
    by eventually-elim simp-all
qed

lemma big-small-asymmetric:  $f \in L\ F\ (g) \implies g \in l\ F\ (f) \implies \text{eventually}\ (\lambda x. f\ x = 0)\ F$ 
  by (drule (1) big-small-trans) (simp add: small-refl-iff)

lemma small-big-asymmetric:  $f \in l\ F\ (g) \implies g \in L\ F\ (f) \implies \text{eventually}\ (\lambda x. f\ x = 0)\ F$ 
  by (drule (1) small-big-trans) (simp add: small-refl-iff)

lemma small-asymmetric:  $f \in l\ F\ (g) \implies g \in l\ F\ (f) \implies \text{eventually}\ (\lambda x. f\ x = 0)\ F$ 
  by (drule (1) small-trans) (simp add: small-refl-iff)

lemma plus-aux:
  assumes  $f \in o[F](g)$ 
  shows  $g \in L\ F\ (\lambda x. f\ x + g\ x)$ 
proof (rule R-E)
  assume R:  $R = (\leq)$ 
  have A:  $1/2 > (0::\text{real})$  by simp
  have B:  $1/2 * (\text{norm}\ (g\ x)) \leq \text{norm}\ (f\ x + g\ x)$ 
    if  $\text{norm}\ (f\ x) \leq 1/2 * \text{norm}\ (g\ x)$  for x
  proof -
    from that have  $1/2 * (\text{norm}\ (g\ x)) \leq (\text{norm}\ (g\ x)) - (\text{norm}\ (f\ x))$ 
    by simp
    also have  $\text{norm}\ (g\ x) - \text{norm}\ (f\ x) \leq \text{norm}\ (f\ x + g\ x)$ 
    by (subst add.commute) (rule norm-diff-ineq)
    finally show ?thesis by simp
  qed
  show  $g \in L\ F\ (\lambda x. f\ x + g\ x)$ 
    apply (rule bigI[of 2])
    apply simp
    apply (use landau-o.smallD[OF assms A] in eventually-elim)
    apply (use B in  $\langle \text{simp add: } R\ \text{algebra-simps} \rangle$ )
    done
next
  assume R:  $R = (\lambda x\ y. x \geq y)$ 
  show  $g \in L\ F\ (\lambda x. f\ x + g\ x)$ 
  proof (rule bigI[of 1/2])
    show eventually ( $\lambda x. R\ (\text{norm}\ (g\ x))\ (1/2 * \text{norm}\ (f\ x + g\ x)))$  F
      using landau-o.smallD[OF assms zero-less-one]
    proof eventually-elim

```

```

    case (elim x)
    have norm (f x + g x) ≤ norm (f x) + norm (g x)
      by (rule norm-triangle-ineq)
    also note elim
    finally show ?case by (simp add: R)
  qed
qed simp-all
qed

end

lemma summable-comparison-test-bigo:
  fixes f :: nat ⇒ real
  assumes summable (λn. norm (g n)) f ∈ O(g)
  shows summable f
proof -
  from ⟨f ∈ O(g)⟩ obtain C where C: eventually (λx. norm (f x) ≤ C * norm
(g x)) at-top
    by (auto elim: landau-o.bigE)
  thus ?thesis
    by (rule summable-comparison-test-ev) (insert assms, auto intro: summable-mult)
qed

lemma bigomega-iff-bigo: g ∈ Ω[F](f) ⟷ f ∈ O[F](g)
proof
  assume f ∈ O[F](g)
  then obtain c where 0 < c ∀F x in F. norm (f x) ≤ c * norm (g x)
    by (rule landau-o.bigE)
  then show g ∈ Ω[F](f)
    by (intro landau-omega.bigI[of inverse c]) (simp-all add: field-simps)
next
  assume g ∈ Ω[F](f)
  then obtain c where 0 < c ∀F x in F. c * norm (f x) ≤ norm (g x)
    by (rule landau-omega.bigE)
  then show f ∈ O[F](g)
    by (intro landau-o.bigI[of inverse c]) (simp-all add: field-simps)
qed

lemma smallomega-iff-smallo: g ∈ ω[F](f) ⟷ f ∈ o[F](g)
proof
  assume f ∈ o[F](g)
  from landau-o.smallD[OF this, of inverse c for c]
    show g ∈ ω[F](f) by (intro landau-omega.smallI) (simp-all add: field-simps)
next
  assume g ∈ ω[F](f)
  from landau-omega.smallD[OF this, of inverse c for c]
    show f ∈ o[F](g) by (intro landau-o.smallI) (simp-all add: field-simps)
qed

```

context *landau-pair*

begin

lemma *big-mono*:

eventually $(\lambda x. R (norm (f x)) (norm (g x))) F \implies f \in L F (g)$

by (rule *bigI*[*OF zero-less-one*]) *simp*

lemma *big-mult*:

assumes $f1 \in L F (g1) f2 \in L F (g2)$

shows $(\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x)$

proof –

from *assms* **obtain** $c1 c2$ **where** $∗: c1 > 0 c2 > 0$

and $∗: \forall_F x \text{ in } F. R (norm (f1 x)) (c1 * norm (g1 x))$

$\forall_F x \text{ in } F. R (norm (f2 x)) (c2 * norm (g2 x))$

by (elim *bigE*)

from $∗$ **have** $c1 * c2 > 0$ **by** *simp*

moreover **have** *eventually* $(\lambda x. R (norm (f1 x * f2 x)) (c1 * c2 * norm (g1 x * g2 x))) F$

using $∗∗$

proof *eventually-elim*

case (elim x)

show *?case*

proof (cases rule: *R-E*)

case *le*

have $norm (f1 x) * norm (f2 x) \leq (c1 * norm (g1 x)) * (c2 * norm (g2 x))$

using *elim le* $∗$ **by** (intro *mult-mono mult-nonneg-nonneg*) *auto*

with *le* **show** *?thesis* **by** (*simp add: le norm-mult mult-ac*)

next

case *ge*

have $(c1 * norm (g1 x)) * (c2 * norm (g2 x)) \leq norm (f1 x) * norm (f2 x)$

using *elim ge* $∗$ **by** (intro *mult-mono mult-nonneg-nonneg*) *auto*

with *ge* **show** *?thesis* **by** (*simp-all add: norm-mult mult-ac*)

qed

qed

ultimately **show** *?thesis* **by** (rule *bigI*)

qed

lemma *small-big-mult*:

assumes $f1 \in l F (g1) f2 \in L F (g2)$

shows $(\lambda x. f1 x * f2 x) \in l F (\lambda x. g1 x * g2 x)$

proof (rule *smallI*)

fix $c1 :: real$ **assume** $c1: c1 > 0$

from *assms*(2) **obtain** $c2$ **where** $c2: c2 > 0$

and $∗: \forall_F x \text{ in } F. R (norm (f2 x)) (c2 * norm (g2 x))$ **by** (elim *bigE*)

from *assms*(1) $c1 c2$ **have** *eventually* $(\lambda x. R (norm (f1 x)) (c1 * inverse c2 * norm (g1 x))) F$

by (*auto intro!: smallD*)

with $∗$ **show** *eventually* $(\lambda x. R (norm (f1 x * f2 x)) (c1 * norm (g1 x * g2 x)))$

F
proof *eventually-elim*
 case (*elim x*)
 show *?case*
 proof (*cases rule: R-E*)
 case *le*
 have $\text{norm } (f1\ x) * \text{norm } (f2\ x) \leq (c1 * \text{inverse } c2 * \text{norm } (g1\ x)) * (c2 * \text{norm } (g2\ x))$
 using *elim le c1 c2 by (intro mult-mono mult-nonneg-nonneg) auto*
 with *le c2 show ?thesis by (simp add: le norm-mult field-simps)*
 next
 case *ge*
 have $\text{norm } (f1\ x) * \text{norm } (f2\ x) \geq (c1 * \text{inverse } c2 * \text{norm } (g1\ x)) * (c2 * \text{norm } (g2\ x))$
 using *elim ge c1 c2 by (intro mult-mono mult-nonneg-nonneg) auto*
 with *ge c2 show ?thesis by (simp add: ge norm-mult field-simps)*
qed
qed
qed

lemma *big-small-mult:*

$f1 \in L\ F\ (g1) \implies f2 \in l\ F\ (g2) \implies (\lambda x. f1\ x * f2\ x) \in l\ F\ (\lambda x. g1\ x * g2\ x)$
by (*subst (1 2) mult.commute (rule small-big-mult)*)

lemma *small-mult:* $f1 \in l\ F\ (g1) \implies f2 \in l\ F\ (g2) \implies (\lambda x. f1\ x * f2\ x) \in l\ F\ (\lambda x. g1\ x * g2\ x)$

by (*rule small-big-mult, assumption, rule small-imp-big*)

lemmas *mult = big-mult small-big-mult big-small-mult small-mult*

lemma *big-power:*

assumes $f \in L\ F\ (g)$
shows $(\lambda x. f\ x \wedge^m) \in L\ F\ (\lambda x. g\ x \wedge^m)$
using *assms by (induction m) (auto intro: mult)*

lemma (*in landau-pair*) *small-power:*

assumes $f \in l\ F\ (g)\ m > 0$
shows $(\lambda x. f\ x \wedge^m) \in l\ F\ (\lambda x. g\ x \wedge^m)$

proof –

have $(\lambda x. f\ x * f\ x \wedge^{(m-1)}) \in l\ F\ (\lambda x. g\ x * g\ x \wedge^{(m-1)})$
by (*intro small-big-mult assms big-power[OF small-imp-big]*)

thus *?thesis*

using *assms by (cases m) (simp-all add: mult-ac)*

qed

lemma *big-power-increasing:*

assumes $(\lambda-. 1) \in L\ F\ f\ m \leq n$
shows $(\lambda x. f\ x \wedge^m) \in L\ F\ (\lambda x. f\ x \wedge^n)$

proof –

```

have ( $\lambda x. f\ x \wedge m * 1 \wedge (n - m)$ )  $\in L\ F\ (\lambda x. f\ x \wedge m * f\ x \wedge (n - m))$ 
  using assms by (intro mult big-power) auto
also have ( $\lambda x. f\ x \wedge m * f\ x \wedge (n - m)$ ) = ( $\lambda x. f\ x \wedge (m + (n - m))$ )
  by (subst power-add [symmetric]) (rule refl)
also have  $m + (n - m) = n$ 
  using assms by simp
finally show ?thesis by simp
qed

```

lemma *small-power-increasing*:

```

assumes ( $\lambda -. 1$ )  $\in l\ F\ f\ m < n$ 
shows ( $\lambda x. f\ x \wedge m$ )  $\in l\ F\ (\lambda x. f\ x \wedge n)$ 
proof -
  note [trans] = small-big-trans
  have ( $\lambda x. f\ x \wedge m * 1$ )  $\in l\ F\ (\lambda x. f\ x \wedge m * f\ x)$ 
    using assms by (intro big-small-mult) auto
  also have ( $\lambda x. f\ x \wedge m * f\ x$ ) = ( $\lambda x. f\ x \wedge \text{Suc } m$ )
    by (simp add: mult-ac)
  also have  $\dots \in L\ F\ (\lambda x. f\ x \wedge n)$ 
    using assms by (intro big-power-increasing[OF small-imp-big]) auto
  finally show ?thesis by simp
qed

```

sublocale *big*: *landau-symbol* $L\ L'\ Lr$

proof

```

have  $L: L = \text{bigo} \vee L = \text{bigomega}$ 
  by (rule R-E) (auto simp: bigo-def L-def bigomega-def fun-eq-iff)
have  $A: (\lambda x. c * f\ x) \in L\ F\ f$  if  $c \neq 0$  for  $c :: 'b$  and  $F$  and  $f :: 'a \Rightarrow 'b$ 
  using that by (intro bigI[of norm c]) (simp-all add: norm-mult)
show  $L\ F\ (\lambda x. c * f\ x) = L\ F\ f$  if  $c \neq 0$  for  $c :: 'b$  and  $F$  and  $f :: 'a \Rightarrow 'b$ 
  using  $\langle c \neq 0 \rangle$  and  $A[\text{of } c\ f]$  and  $A[\text{of inverse } c\ \lambda x. c * f\ x]$ 
  by (intro equalityI big-subsetI) (simp-all add: field-simps)
show  $((\lambda x. c * f\ x) \in L\ F\ g) = (f \in L\ F\ g)$  if  $c \neq 0$ 
  for  $c :: 'b$  and  $F$  and  $f\ g :: 'a \Rightarrow 'b$ 
proof -
  from  $\langle c \neq 0 \rangle$  and  $A[\text{of } c\ f]$  and  $A[\text{of inverse } c\ \lambda x. c * f\ x]$ 
  have  $(\lambda x. c * f\ x) \in L\ F\ f\ f \in L\ F\ (\lambda x. c * f\ x)$ 
    by (simp-all add: field-simps)
  then show ?thesis by (intro iffI) (erule (1) big-trans)+
qed
show  $(\lambda x. \text{inverse } (g\ x)) \in L\ F\ (\lambda x. \text{inverse } (f\ x))$ 
  if *:  $f \in L\ F\ (g)$  and **: eventually  $(\lambda x. f\ x \neq 0)\ F$  eventually  $(\lambda x. g\ x \neq 0)\ F$ 
  for  $f\ g :: 'a \Rightarrow 'b$  and  $F$ 
proof -
  from * obtain  $c$  where  $c: c > 0$  and **:  $\forall_F\ x\ \text{in } F. R\ (\text{norm } (f\ x))\ (c * \text{norm } (g\ x))$ 
    by (elim bigE)
  from ** ** have eventually  $(\lambda x. R\ (\text{norm } (\text{inverse } (g\ x)))\ (c * \text{norm } (\text{inverse } (f\ x))))$ 

```

```

(f x)))) F
  by eventually-elim (rule R-E, simp-all add: field-simps norm-inverse norm-divide
c)
  with c show ?thesis by (rule bigI)
qed
show  $L F g \subseteq L F (\lambda x. f x + g x)$  if  $f \in o[F](g)$  for  $f g :: 'a \Rightarrow 'b$  and  $F$ 
  using plus-aux that by (blast intro!: big-subsetI)
show  $L F (f) = L F (g)$  if eventually  $(\lambda x. f x = g x) F$  for  $f g :: 'a \Rightarrow 'b$  and  $F$ 
  unfolding L-def by (subst eventually-subst'[OF that]) (rule refl)
show  $f \in L F (h) \longleftrightarrow g \in L F (h)$  if eventually  $(\lambda x. f x = g x) F$ 
  for  $f g h :: 'a \Rightarrow 'b$  and  $F$ 
  unfolding L-def mem-Collect-eq
  by (subst (1) eventually-subst'[OF that]) (rule refl)
show  $L F f \subseteq L F g$  if  $f \in L F g$  for  $f g :: 'a \Rightarrow 'b$  and  $F$ 
  using that by (rule big-subsetI)
show  $L F (f) = L F (g)$  if  $f \in \Theta[F](g)$  for  $f g :: 'a \Rightarrow 'b$  and  $F$ 
  using L that unfolding bigtheta-def
  by (intro equalityI big-subsetI) (auto simp: bigomega-iff-bigo)
show  $f \in L F (h) \longleftrightarrow g \in L F (h)$  if  $f \in \Theta[F](g)$  for  $f g h :: 'a \Rightarrow 'b$  and  $F$ 
  by (rule disjE[OF L])
  (use that in <auto simp: bigtheta-def bigomega-iff-bigo intro: landau-o.big-trans>)
show  $(\lambda x. h x * f x) \in L F (\lambda x. h x * g x)$  if  $f \in L F g$  for  $f g h :: 'a \Rightarrow 'b$  and
F
  using that by (intro big-mult) simp
show  $f \in L F (h)$  if  $f \in L F g g \in L F h$  for  $f g h :: 'a \Rightarrow 'b$  and  $F$ 
  using that by (rule big-trans)
show  $(\lambda x. f (h x)) \in L' G (\lambda x. g (h x))$ 
  if  $f \in L F g$  and filterlim  $h F G$ 
  for  $f g :: 'a \Rightarrow 'b$  and  $h :: 'c \Rightarrow 'a$  and  $F G$ 
  using that by (auto simp: L-def L'-def filterlim-iff)
show  $f \in L (\sup F G) g$  if  $f \in L F g f \in L G g$ 
  for  $f g :: 'a \Rightarrow 'b$  and  $F G :: 'a \text{ filter}$ 
proof -
  from that [THEN bigE] obtain  $c1 c2$ 
  where *:  $c1 > 0 c2 > 0$ 
  and **:  $\forall_F x \text{ in } F. R (\text{norm } (f x)) (c1 * \text{norm } (g x))$ 
   $\forall_F x \text{ in } G. R (\text{norm } (f x)) (c2 * \text{norm } (g x))$  .
  define  $c$  where  $c = (\text{if } R c1 c2 \text{ then } c2 \text{ else } c1)$ 
  from * have  $c: R c1 c R c2 c c > 0$ 
  by (auto simp: c-def dest: R-linear)
  with ** have eventually  $(\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g x))) F$ 
  eventually  $(\lambda x. R (\text{norm } (f x)) (c * \text{norm } (g x))) G$ 
  by (force elim: eventually-mono intro: R-trans[OF - R-mult-right-mono]) +
  with c show  $f \in L (\sup F G) g$ 
  by (auto simp: L-def eventually-sup)
qed
show  $((\lambda x. f (h x)) \in L' (\text{filtercomap } h F) (\lambda x. g (h x)))$  if  $(f \in L F g)$ 
  for  $f g :: 'a \Rightarrow 'b$  and  $h :: 'c \Rightarrow 'a$  and  $F G :: 'a \text{ filter}$ 
  using that unfolding L-def L'-def by auto

```

qed (*auto simp: L-def Lr-def eventually-filtermap L'-def*
intro: filter-leD exI[of - 1::real])

sublocale *small: landau-symbol l l' lr*

proof

have $A: (\lambda x. c * f x) \in L F f$ **if** $c \neq 0$ **for** $c :: 'b$ **and** $f :: 'a \Rightarrow 'b$ **and** F
using *that by (intro bigI[of norm c]) (simp-all add: norm-mult)*
show $l F (\lambda x. c * f x) = l F f$ **if** $c \neq 0$ **for** $c :: 'b$ **and** $f :: 'a \Rightarrow 'b$ **and** F
using *that and A[of c f] and A[of inverse c $\lambda x. c * f x$]*
by (*intro equalityI small-subsetI (simp-all add: field-simps)*)
show $((\lambda x. c * f x) \in l F g) = (f \in l F g)$ **if** $c \neq 0$ **for** $c :: 'b$ **and** $f g :: 'a \Rightarrow 'b$
and F

proof –

from *that and A[of c f] and A[of inverse c $\lambda x. c * f x$]*

have $(\lambda x. c * f x) \in L F f f \in L F (\lambda x. c * f x)$

by (*simp-all add: field-simps*)

then show *?thesis*

by (*intro iffI (erule (1) big-small-trans)+*)

qed

show $l F g \subseteq l F (\lambda x. f x + g x)$ **if** $f \in o[F](g)$ **for** $f g :: 'a \Rightarrow 'b$ **and** F

using *plus-aux that by (blast intro!: small-subsetI)*

show $(\lambda x. \text{inverse } (g x)) \in l F (\lambda x. \text{inverse } (f x))$

if $A: f \in l F (g)$ **and** $B: \text{eventually } (\lambda x. f x \neq 0) F \text{ eventually } (\lambda x. g x \neq 0) F$
for $f g :: 'a \Rightarrow 'b$ **and** F

proof (*rule smallI*)

fix $c :: \text{real}$ **assume** $c: c > 0$

from B *smallD[OF A c]*

show *eventually* $(\lambda x. R (\text{norm } (\text{inverse } (g x))) (c * \text{norm } (\text{inverse } (f x)))) F$

by *eventually-elim (rule R-E, simp-all add: field-simps norm-inverse norm-divide)*

qed

show $l F (f) = l F (g)$ **if** *eventually* $(\lambda x. f x = g x) F$ **for** $f g :: 'a \Rightarrow 'b$ **and** F

unfolding *l-def by (subst eventually-subst'[OF that]) (rule refl)*

show $f \in l F (h) \longleftrightarrow g \in l F (h)$ **if** *eventually* $(\lambda x. f x = g x) F$ **for** $f g h :: 'a \Rightarrow 'b$ **and** F

unfolding *l-def mem-Collect-eq by (subst (1) eventually-subst'[OF that]) (rule refl)*

show $l F f \subseteq l F g$ **if** $f \in l F g$ **for** $f g :: 'a \Rightarrow 'b$ **and** F

using *that by (intro small-subsetI small-imp-big)*

show $l F (f) = l F (g)$ **if** $f \in \Theta[F](g)$ **for** $f g :: 'a \Rightarrow 'b$ **and** F

proof –

have $L: L = \text{bigo} \vee L = \text{bigomega}$

by (*rule R-E (auto simp: bigo-def L-def bigomega-def fun-eq-iff)*)

with *that show ?thesis unfolding bigtheta-def*

by (*intro equalityI small-subsetI (auto simp: bigomega-iff-bigo)*)

qed

show $f \in l F (h) \longleftrightarrow g \in l F (h)$ **if** $f \in \Theta[F](g)$ **for** $f g h :: 'a \Rightarrow 'b$ **and** F

proof –

have $l: l = \text{smallo} \vee l = \text{smallomega}$

by (*rule R-E (auto simp: smallo-def l-def smallomega-def fun-eq-iff)*)

```

show ?thesis
  by (rule disjE[OF l])
    (use that in ⟨auto simp: bigtheta-def bigomega-iff-bigo smallomega-iff-smallo
      intro: landau-o.big-small-trans landau-o.small-big-trans⟩)
qed
show  $(\lambda x. h\ x * f\ x) \in l\ F\ (\lambda x. h\ x * g\ x)$  if  $f \in l\ F\ g$  for  $f\ g\ h :: 'a \Rightarrow 'b$  and  $F$ 
  using that by (intro big-small-mult) simp
show  $f \in l\ F\ (h)$  if  $f \in l\ F\ g\ g \in l\ F\ h$  for  $f\ g\ h :: 'a \Rightarrow 'b$  and  $F$ 
  using that by (rule small-trans)
show  $(\lambda x. f\ (h\ x)) \in l'\ G\ (\lambda x. g\ (h\ x))$  if  $f \in l\ F\ g$  and filterlim  $h\ F\ G$ 
  for  $f\ g :: 'a \Rightarrow 'b$  and  $h :: 'c \Rightarrow 'a$  and  $F\ G$ 
  using that by (auto simp: l-def l'-def filterlim-iff)
show  $((\lambda x. f\ (h\ x)) \in l'\ (filtercomap\ h\ F)\ (\lambda x. g\ (h\ x)))$  if  $f \in l\ F\ g$ 
  for  $f\ g :: 'a \Rightarrow 'b$  and  $h :: 'c \Rightarrow 'a$  and  $F\ G :: 'a\ filter$ 
  using that unfolding l-def l'-def by auto
qed (auto simp: l-def lr-def eventually-filtermap l'-def eventually-sup intro: filter-leD)

```

These rules allow chaining of Landau symbol propositions in Isar with "also".

```

lemma big-mult-1:  $f \in L\ F\ (g) \Longrightarrow (\lambda-. 1) \in L\ F\ (h) \Longrightarrow f \in L\ F\ (\lambda x. g\ x * h\ x)$ 
and big-mult-1':  $(\lambda-. 1) \in L\ F\ (g) \Longrightarrow f \in L\ F\ (h) \Longrightarrow f \in L\ F\ (\lambda x. g\ x * h\ x)$ 
and small-mult-1:  $f \in l\ F\ (g) \Longrightarrow (\lambda-. 1) \in L\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x. g\ x * h\ x)$ 
and small-mult-1':  $(\lambda-. 1) \in L\ F\ (g) \Longrightarrow f \in l\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x. g\ x * h\ x)$ 
and small-mult-1'':  $f \in L\ F\ (g) \Longrightarrow (\lambda-. 1) \in l\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x. g\ x * h\ x)$ 
and small-mult-1''':  $(\lambda-. 1) \in l\ F\ (g) \Longrightarrow f \in L\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x. g\ x * h\ x)$ 
by (drule (1) big.mult big-small-mult small-big-mult, simp)+

```

```

lemma big-1-mult:  $f \in L\ F\ (g) \Longrightarrow h \in L\ F\ (\lambda-. 1) \Longrightarrow (\lambda x. f\ x * h\ x) \in L\ F\ (g)$ 
and big-1-mult':  $h \in L\ F\ (\lambda-. 1) \Longrightarrow f \in L\ F\ (g) \Longrightarrow (\lambda x. f\ x * h\ x) \in L\ F\ (g)$ 
and small-1-mult:  $f \in l\ F\ (g) \Longrightarrow h \in L\ F\ (\lambda-. 1) \Longrightarrow (\lambda x. f\ x * h\ x) \in l\ F\ (g)$ 
and small-1-mult':  $h \in L\ F\ (\lambda-. 1) \Longrightarrow f \in l\ F\ (g) \Longrightarrow (\lambda x. f\ x * h\ x) \in l\ F\ (g)$ 
and small-1-mult'':  $f \in L\ F\ (g) \Longrightarrow h \in l\ F\ (\lambda-. 1) \Longrightarrow (\lambda x. f\ x * h\ x) \in l\ F\ (g)$ 
and small-1-mult''':  $h \in l\ F\ (\lambda-. 1) \Longrightarrow f \in L\ F\ (g) \Longrightarrow (\lambda x. f\ x * h\ x) \in l\ F\ (g)$ 
by (drule (1) big.mult big-small-mult small-big-mult, simp)+

```

lemmas mult-1-trans =

```

big-mult-1 big-mult-1' small-mult-1 small-mult-1' small-mult-1'' small-mult-1'''
big-1-mult big-1-mult' small-1-mult small-1-mult' small-1-mult'' small-1-mult'''

```

lemma *big-equal-iff-bigtheta*: $L\ F\ (f) = L\ F\ (g) \longleftrightarrow f \in \Theta[F](g)$
proof
 have $L: L = \text{bigo} \vee L = \text{bigomega}$
 by (rule *R-E*) (auto simp: *fun-eq-iff L-def bigo-def bigomega-def*)
 fix $f\ g :: 'a \Rightarrow 'b$ **assume** $L\ F\ (f) = L\ F\ (g)$
 with *big-refl*[*of f F*] *big-refl*[*of g F*] **have** $f \in L\ F\ (g) \ g \in L\ F\ (f)$ **by** *simp-all*
 thus $f \in \Theta[F](g)$ **using** L **unfolding** *bigtheta-def* **by** (auto simp: *bigomega-iff-bigo*)
qed (rule *big.cong-bigtheta*)

lemma *big-prod*:
 assumes $\bigwedge x. x \in A \implies f\ x \in L\ F\ (g\ x)$
 shows $(\lambda y. \prod_{x \in A}. f\ x\ y) \in L\ F\ (\lambda y. \prod_{x \in A}. g\ x\ y)$
 using *assms* **by** (induction A rule: *infinite-finite-induct*) (auto intro!: *big.mult*)

lemma *big-prod-in-1*:
 assumes $\bigwedge x. x \in A \implies f\ x \in L\ F\ (\lambda\cdot. 1)$
 shows $(\lambda y. \prod_{x \in A}. f\ x\ y) \in L\ F\ (\lambda\cdot. 1)$
 using *assms* **by** (induction A rule: *infinite-finite-induct*) (auto intro!: *big.mult-in-1*)

end

context *landau-symbol*
begin

lemma *plus-absorb1*:
 assumes $f \in o[F](g)$
 shows $L\ F\ (\lambda x. f\ x + g\ x) = L\ F\ (g)$
proof (*intro equalityI*)
 from *plus-subset1* **and** *assms* **show** $L\ F\ g \subseteq L\ F\ (\lambda x. f\ x + g\ x)$.
 from *landau-o.small.plus-subset1*[*OF assms*] **and** *assms* **have** $(\lambda x. -f\ x) \in o[F](\lambda x. f\ x + g\ x)$
 by (auto simp: *landau-o.small.uminus-in-iff*)
 from *plus-subset1*[*OF this*] **show** $L\ F\ (\lambda x. f\ x + g\ x) \subseteq L\ F\ (g)$ **by** *simp*
qed

lemma *plus-absorb2*: $g \in o[F](f) \implies L\ F\ (\lambda x. f\ x + g\ x) = L\ F\ (f)$
 using *plus-absorb1*[*of g F f*] **by** (*simp add: add.commute*)

lemma *diff-absorb1*: $f \in o[F](g) \implies L\ F\ (\lambda x. f\ x - g\ x) = L\ F\ (g)$
 by (*simp only: diff-conv-add-uminus plus-absorb1 landau-o.small.uminus uminus*)

lemma *diff-absorb2*: $g \in o[F](f) \implies L\ F\ (\lambda x. f\ x - g\ x) = L\ F\ (f)$
 by (*simp only: diff-conv-add-uminus plus-absorb2 landau-o.small.uminus-in-iff*)

lemmas *absorb* = *plus-absorb1 plus-absorb2 diff-absorb1 diff-absorb2*

end

lemma *bighetaI* [*intro*]: $f \in O[F](g) \implies f \in \Omega[F](g) \implies f \in \Theta[F](g)$
unfolding *bigheta-def bigomega-def* **by** *blast*

lemma *bighetaD1* [*dest*]: $f \in \Theta[F](g) \implies f \in O[F](g)$
and *bighetaD2* [*dest*]: $f \in \Theta[F](g) \implies f \in \Omega[F](g)$
unfolding *bigheta-def bigo-def bigomega-def* **by** *blast+*

lemma *bigheta-refl* [*simp*]: $f \in \Theta[F](f)$
unfolding *bigheta-def* **by** *simp*

lemma *bigheta-sym*: $f \in \Theta[F](g) \longleftrightarrow g \in \Theta[F](f)$
unfolding *bigheta-def* **by** (*auto simp: bigomega-iff-bigo*)

lemmas *landau-flip* =
bigomega-iff-bigo[symmetric] smallomega-iff-smallo[symmetric]
bigomega-iff-bigo smallomega-iff-smallo bigheta-sym

interpretation *landau-theta*: *landau-symbol bigheta bigheta bigheta*
proof

fix *f g* :: '*a* \Rightarrow '*b* **and** *F*
assume $f \in o[F](g)$
hence $O[F](g) \subseteq O[F](\lambda x. f x + g x)$ $\Omega[F](g) \subseteq \Omega[F](\lambda x. f x + g x)$
by (*rule landau-o.big.plus-subset1 landau-omega.big.plus-subset1*) +
thus $\Theta[F](g) \subseteq \Theta[F](\lambda x. f x + g x)$ **unfolding** *bigheta-def* **by** *blast*
next
fix *f g* :: '*a* \Rightarrow '*b* **and** *F*
assume $f \in \Theta[F](g)$
thus $A: \Theta[F](f) = \Theta[F](g)$
apply (*subst (1 2) bigheta-def*)
apply (*subst landau-o.big.cong-bigheta landau-omega.big.cong-bigheta, as-*
sumption) +
apply (*rule refl*)
done
thus $\Theta[F](f) \subseteq \Theta[F](g)$ **by** *simp*
fix *h* :: '*a* \Rightarrow '*b*
show $f \in \Theta[F](h) \longleftrightarrow g \in \Theta[F](h)$ **by** (*subst (1 2) bigheta-sym*) (*simp add: A*)
next
fix *f g h* :: '*a* \Rightarrow '*b* **and** *F*
assume $f \in \Theta[F](g)$ $g \in \Theta[F](h)$
thus $f \in \Theta[F](h)$ **unfolding** *bigheta-def*
by (*blast intro: landau-o.big.trans landau-omega.big.trans*)
next
fix *f* :: '*a* \Rightarrow '*b* **and** *F1 F2* :: '*a* *filter*
assume $F1 \leq F2$
thus $\Theta[F2](f) \subseteq \Theta[F1](f)$
by (*auto simp: bigheta-def intro: landau-o.big.filter-mono landau-omega.big.filter-mono*)
qed (*auto simp: bigheta-def landau-o.big.norm-iff*)

```

landau-o.big.cmult landau-omega.big.cmult
landau-o.big.cmult-in-iff landau-omega.big.cmult-in-iff
landau-o.big.in-cong landau-omega.big.in-cong
landau-o.big.mult landau-omega.big.mult
landau-o.big.inverse landau-omega.big.inverse
landau-o.big.compose landau-omega.big.compose
landau-o.big.bot' landau-omega.big.bot'
landau-o.big.in-filtermap-iff landau-omega.big.in-filtermap-iff
landau-o.big.sup landau-omega.big.sup
landau-o.big.filtercomap landau-omega.big.filtercomap
dest: landau-o.big.cong landau-omega.big.cong)

```

```

lemmas landau-symbols =
  landau-o.big.landau-symbol-axioms landau-o.small.landau-symbol-axioms
  landau-omega.big.landau-symbol-axioms landau-omega.small.landau-symbol-axioms
  landau-theta.landau-symbol-axioms

```

```

lemma bigoI [intro]:
  assumes eventually ( $\lambda x. (\text{norm } (f x)) \leq c * (\text{norm } (g x))$ ) F
  shows  $f \in O[F](g)$ 
proof (rule landau-o.bigI)
  show  $\max 1 c > 0$  by simp
  have  $c * (\text{norm } (g x)) \leq \max 1 c * (\text{norm } (g x))$  for x
  by (simp add: mult-right-mono)
  with assms show eventually ( $\lambda x. (\text{norm } (f x)) \leq \max 1 c * (\text{norm } (g x))$ ) F
  by (auto elim!: eventually-mono dest: order.trans)
qed

```

```

lemma smallomegaD [dest]:
  assumes  $f \in \omega[F](g)$ 
  shows eventually ( $\lambda x. (\text{norm } (f x)) \geq c * (\text{norm } (g x))$ ) F
proof (cases  $c > 0$ )
  case False
  show ?thesis
  by (intro always-eventually allI, rule order.trans[of - 0])
  (insert False, auto intro!: mult-nonpos-nonneg)
qed (blast dest: landau-omega.smallD[OF assms, of c])

```

```

lemma bigthetaI':
  assumes  $c1 > 0$   $c2 > 0$ 
  assumes eventually ( $\lambda x. c1 * (\text{norm } (g x)) \leq (\text{norm } (f x)) \wedge (\text{norm } (f x)) \leq c2$ 
  *  $(\text{norm } (g x))$ ) F
  shows  $f \in \Theta[F](g)$ 
apply (rule bigthetaI)
apply (rule landau-o.bigI[OF assms(2)]) using assms(3) apply (eventually-elim,
simp)
apply (rule landau-omega.bigI[OF assms(1)]) using assms(3) apply (eventually-elim,
simp)

```


done

lemma *bighetaI-cong: eventually* $(\lambda x. f x = g x) F \implies f \in \Theta[F](g)$
by (*intro bighetaI'[of 1 1]*) (*auto elim!: eventually-mono*)

lemma (*in landau-symbol*) *ev-eq-trans1*:
 $f \in L F (\lambda x. g x (h x)) \implies \text{eventually } (\lambda x. h x = h' x) F \implies f \in L F (\lambda x. g x (h' x))$
by (*rule bigheta-trans1[OF - bighetaI-cong]*) (*auto elim!: eventually-mono*)

lemma (*in landau-symbol*) *ev-eq-trans2*:
 $\text{eventually } (\lambda x. f x = f' x) F \implies (\lambda x. g x (f' x)) \in L F (h) \implies (\lambda x. g x (f x)) \in L F (h)$
by (*rule bigheta-trans2[OF bighetaI-cong]*) (*auto elim!: eventually-mono*)

declare *landau-o.smallI landau-omega.bigI landau-omega.smallI* [*intro*]
declare *landau-o.bigE landau-omega.bigE* [*elim*]
declare *landau-o.smallD*

lemma (*in landau-symbol*) *bigheta-trans1'*:
 $f \in L F (g) \implies h \in \Theta[F](g) \implies f \in L F (h)$
by (*subst cong-bigheta[symmetric]*) (*simp add: bigheta-sym*)

lemma (*in landau-symbol*) *bigheta-trans2'*:
 $g \in \Theta[F](f) \implies g \in L F (h) \implies f \in L F (h)$
by (*rule bigheta-trans2, subst bigheta-sym*)

lemma *bigo-bigomega-trans*: $f \in O[F](g) \implies h \in \Omega[F](g) \implies f \in O[F](h)$
and *bigo-smallomega-trans*: $f \in O[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$
and *smallo-bigomega-trans*: $f \in o[F](g) \implies h \in \Omega[F](g) \implies f \in o[F](h)$
and *smallo-smallomega-trans*: $f \in o[F](g) \implies h \in \omega[F](g) \implies f \in o[F](h)$
and *bigomega-bigo-trans*: $f \in \Omega[F](g) \implies h \in O[F](g) \implies f \in \Omega[F](h)$
and *bigomega-smallo-trans*: $f \in \Omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$
and *smallomega-bigo-trans*: $f \in \omega[F](g) \implies h \in O[F](g) \implies f \in \omega[F](h)$
and *smallomega-smallo-trans*: $f \in \omega[F](g) \implies h \in o[F](g) \implies f \in \omega[F](h)$
by (*unfold bigomega-iff-bigo smallomega-iff-smallo*)
(erule (1) landau-o.big-trans landau-o.big-small-trans landau-o.small-big-trans landau-o.big-trans landau-o.small-trans)+

lemmas *landau-trans-lift* [*trans*] =
landau-symbols[THEN landau-symbol.lift-trans]
landau-symbols[THEN landau-symbol.lift-trans']
landau-symbols[THEN landau-symbol.lift-trans-bigheta]
landau-symbols[THEN landau-symbol.lift-trans-bigheta']

lemmas *landau-mult-1-trans* [*trans*] =
landau-o.mult-1-trans landau-omega.mult-1-trans

lemmas *landau-trans* [*trans*] =

```

landau-symbols[THEN landau-symbol.bigheta-trans1]
landau-symbols[THEN landau-symbol.bigheta-trans2]
landau-symbols[THEN landau-symbol.bigheta-trans1 ^]
landau-symbols[THEN landau-symbol.bigheta-trans2 ^]
landau-symbols[THEN landau-symbol.ev-eq-trans1]
landau-symbols[THEN landau-symbol.ev-eq-trans2]

```

```

landau-o.big-trans landau-o.small-trans landau-o.small-big-trans landau-o.big-small-trans
landau-omega.big-trans landau-omega.small-trans
landau-omega.small-big-trans landau-omega.big-small-trans

```

```

bigo-bigomega-trans bigo-smallomega-trans smallo-bigomega-trans smallo-smallomega-trans
bigomega-bigo-trans bigomega-smallo-trans smallomega-bigo-trans smallomega-smallo-trans

```

```

lemma bigheta-inverse [simp]:
  shows  $(\lambda x. \text{inverse } (f x)) \in \Theta[F](\lambda x. \text{inverse } (g x)) \longleftrightarrow f \in \Theta[F](g)$ 
proof -
  have  $(\lambda x. \text{inverse } (f x)) \in O[F](\lambda x. \text{inverse } (g x))$ 
    if  $A: f \in \Theta[F](g)$ 
    for  $f g :: 'a \Rightarrow 'b$  and  $F$ 
  proof -
    from  $A$  obtain  $c1 c2 :: \text{real}$  where  $*$ :  $c1 > 0 \ c2 > 0$ 
    and  $*$ :  $\forall_F x \text{ in } F. \text{norm } (f x) \leq c1 * \text{norm } (g x)$ 
       $\forall_F x \text{ in } F. c2 * \text{norm } (g x) \leq \text{norm } (f x)$ 
    unfolding bigheta-def by (elim landau-o.bigE landau-omega.bigE IntE)
    from  $\langle c2 > 0 \rangle$  have  $c2: \text{inverse } c2 > 0$  by simp
    from  $*$  have eventually  $(\lambda x. \text{norm } (\text{inverse } (f x)) \leq \text{inverse } c2 * \text{norm } (\text{inverse } (g x))) F$ 
  proof eventually-elim
    fix  $x$  assume  $A: \text{norm } (f x) \leq c1 * \text{norm } (g x) \ c2 * \text{norm } (g x) \leq \text{norm } (f x)$ 
  from  $A *$  have  $f x = 0 \longleftrightarrow g x = 0$ 
    by (auto simp: field-simps mult-le-0-iff)
  with  $A *$  show  $\text{norm } (\text{inverse } (f x)) \leq \text{inverse } c2 * \text{norm } (\text{inverse } (g x))$ 
    by (force simp: field-simps norm-inverse norm-divide)
  qed
  with  $c2$  show ?thesis by (rule landau-o.bigI)
qed
then show ?thesis
  unfolding bigheta-def
  by (force simp: bigomega-iff-bigo bigheta-sym)
qed

```

```

lemma bigheta-divide:
  assumes  $f1 \in \Theta(f2) \ g1 \in \Theta(g2)$ 
  shows  $(\lambda x. f1 x / g1 x) \in \Theta(\lambda x. f2 x / g2 x)$ 
  by (subst (1 2) divide-inverse, intro landau-theta.mult) (simp-all add: bigheta-inverse assms)

```

lemma *eventually-nonzero-bigtheta*:
assumes $f \in \Theta[F](g)$
shows $\text{eventually } (\lambda x. f\ x \neq 0)\ F \longleftrightarrow \text{eventually } (\lambda x. g\ x \neq 0)\ F$
proof –
have $\text{eventually } (\lambda x. g\ x \neq 0)\ F$
if $A: f \in \Theta[F](g)$ **and** $B: \text{eventually } (\lambda x. f\ x \neq 0)\ F$
for $f\ g :: 'a \Rightarrow 'b$
proof –
from A **obtain** $c1\ c2$ **where**
 $\forall_F x\ \text{in } F. \text{norm } (f\ x) \leq c1 * \text{norm } (g\ x)$
 $\forall_F x\ \text{in } F. c2 * \text{norm } (g\ x) \leq \text{norm } (f\ x)$
unfolding *bigtheta-def* **by** (*elim landau-o.bigE landau-omega.bigE IntE*)
with B **show** *?thesis* **by** *eventually-elim auto*
qed
with *assms* **show** *?thesis* **by** (*force simp: bigtheta-sym*)
qed

54.2 Landau symbols and limits

lemma *bigoI-tendsto-norm*:
fixes $f\ g$
assumes $((\lambda x. \text{norm } (f\ x / g\ x)) \longrightarrow c)\ F$
assumes $\text{eventually } (\lambda x. g\ x \neq 0)\ F$
shows $f \in O[F](g)$
proof (*rule bigoI*)
from *assms* **have** $\text{eventually } (\lambda x. \text{dist } (\text{norm } (f\ x / g\ x))\ c < 1)\ F$
using *tendstoD* **by** *force*
thus $\text{eventually } (\lambda x. (\text{norm } (f\ x)) \leq (\text{norm } c + 1) * (\text{norm } (g\ x)))\ F$
unfolding *dist-real-def* **using** *assms(2)*
proof *eventually-elim*
case (*elim x*)
have $(\text{norm } (f\ x)) - \text{norm } c * (\text{norm } (g\ x)) \leq \text{norm } ((\text{norm } (f\ x)) - c * (\text{norm } (g\ x)))$
unfolding *norm-mult [symmetric]* **using** *norm-triangle-ineq2[of norm (f x) c * norm (g x)]*
by (*simp add: norm-mult abs-mult*)
also from *elim* **have** $\dots = \text{norm } (\text{norm } (g\ x)) * \text{norm } (\text{norm } (f\ x / g\ x) - c)$
unfolding *norm-mult [symmetric]* **by** (*simp add: algebra-simps norm-divide*)
also from *elim* **have** $\text{norm } (\text{norm } (f\ x / g\ x) - c) \leq 1$ **by** *simp*
hence $\text{norm } (\text{norm } (g\ x)) * \text{norm } (\text{norm } (f\ x / g\ x) - c) \leq \text{norm } (\text{norm } (g\ x)) * 1$
by (*rule mult-left-mono*) *simp-all*
finally show *?case* **by** (*simp add: algebra-simps*)
qed
qed

lemma *bigoI-tendsto*:
assumes $((\lambda x. f\ x / g\ x) \longrightarrow c)\ F$
assumes $\text{eventually } (\lambda x. g\ x \neq 0)\ F$

```

shows  $f \in O[F](g)$ 
using assms by (rule bigO-tendsto-norm[OF tendsto-norm])

lemma bigomegaI-tendsto-norm:
  assumes c-not-0:  $(c::\text{real}) \neq 0$ 
  assumes lim:  $((\lambda x. \text{norm } (f\ x / g\ x)) \longrightarrow c) F$ 
  shows  $f \in \Omega[F](g)$ 
proof (cases  $F = \text{bot}$ )
  case False
  show ?thesis
proof (rule landau-omega.bigI)
  from lim have  $c \geq 0$  by (rule tendsto-lowerbound) (insert False, simp-all)
  with c-not-0 have  $c > 0$  by simp
  with c-not-0 show  $c/2 > 0$  by simp
  from lim have ev:  $\bigwedge \varepsilon. \varepsilon > 0 \implies \text{eventually } (\lambda x. \text{norm } (\text{norm } (f\ x / g\ x) - c) < \varepsilon) F$ 
  by (subst (asm) tendsto-iff) (simp add: dist-real-def)
  from ev[OF  $\langle c/2 > 0 \rangle$ ] show  $\text{eventually } (\lambda x. (\text{norm } (f\ x)) \geq c/2 * (\text{norm } (g\ x))) F$ 
proof (eventually-elim)
  fix x assume B:  $\text{norm } (\text{norm } (f\ x / g\ x) - c) < c / 2$ 
  from B have g:  $g\ x \neq 0$  by auto
  from B have  $-c/2 < -\text{norm } (\text{norm } (f\ x / g\ x) - c)$  by simp
  also have  $\dots \leq \text{norm } (f\ x / g\ x) - c$  by simp
  finally show  $(\text{norm } (f\ x)) \geq c/2 * (\text{norm } (g\ x))$  using g
  by (simp add: field-simps norm-mult norm-divide)
qed
qed
qed simp

lemma bigomegaI-tendsto:
  assumes c-not-0:  $(c::\text{real}) \neq 0$ 
  assumes lim:  $((\lambda x. f\ x / g\ x) \longrightarrow c) F$ 
  shows  $f \in \Omega[F](g)$ 
by (rule bigomegaI-tendsto-norm[OF - tendsto-norm, of c]) (insert assms, simp-all)

lemma smallomegaI-filterlim-at-top-norm:
  assumes lim: filterlim  $(\lambda x. \text{norm } (f\ x / g\ x))$  at-top F
  shows  $f \in \omega[F](g)$ 
proof (rule landau-omega.smallI)
  fix c :: real assume c-pos:  $c > 0$ 
  from lim have ev:  $\text{eventually } (\lambda x. \text{norm } (f\ x / g\ x) \geq c) F$ 
  by (subst (asm) filterlim-at-top) simp
  thus  $\text{eventually } (\lambda x. (\text{norm } (f\ x)) \geq c * (\text{norm } (g\ x))) F$ 
proof eventually-elim
  fix x assume A:  $\text{norm } (f\ x / g\ x) \geq c$ 
  from A c-pos have g:  $g\ x \neq 0$  by auto
  with A show  $(\text{norm } (f\ x)) \geq c * (\text{norm } (g\ x))$  by (simp add: field-simps norm-divide)

```

qed
qed

lemma *smallomegaI-filterlim-at-infinity*:
 assumes *lim*: *filterlim* ($\lambda x. f\ x / g\ x$) *at-infinity* *F*
 shows $f \in \omega[F](g)$
proof (*rule smallomegaI-filterlim-at-top-norm*)
 from *lim* **show** *filterlim* ($\lambda x. \text{norm } (f\ x / g\ x)$) *at-top* *F*
 by (*rule filterlim-at-infinity-imp-norm-at-top*)
 qed

lemma *smallomegaD-filterlim-at-top-norm*:
 assumes $f \in \omega[F](g)$
 assumes *eventually* ($\lambda x. g\ x \neq 0$) *F*
 shows $\text{LIM } x\ F. \text{norm } (f\ x / g\ x) :> \text{at-top}$
proof (*subst filterlim-at-top-gt, clarify*)
 fix *c* :: *real* **assume** *c*: $c > 0$
 from *landau-omega.smallD*[*OF assms(1) this*] *assms(2)*
show *eventually* ($\lambda x. \text{norm } (f\ x / g\ x) \geq c$) *F*
 by *eventually-elim* (*simp add: field-simps norm-divide*)
 qed

lemma *smallomegaD-filterlim-at-infinity*:
 assumes $f \in \omega[F](g)$
 assumes *eventually* ($\lambda x. g\ x \neq 0$) *F*
 shows $\text{LIM } x\ F. f\ x / g\ x :> \text{at-infinity}$
 using *assms* **by** (*intro filterlim-norm-at-top-imp-at-infinity smallomegaD-filterlim-at-top-norm*)

lemma *smallomega-1-conv-filterlim*: $f \in \omega[F](\lambda-. 1) \longleftrightarrow \text{filterlim } f\ \text{at-infinity } F$
by (*auto intro: smallomegaI-filterlim-at-infinity dest: smallomegaD-filterlim-at-infinity*)

lemma *smalloI-tendsto*:
 assumes *lim*: ($\lambda x. f\ x / g\ x \longrightarrow 0$) *F*
 assumes *eventually* ($\lambda x. g\ x \neq 0$) *F*
 shows $f \in o[F](g)$
proof (*rule landau-o.smallI*)
 fix *c* :: *real* **assume** *c-pos*: $c > 0$
 from *c-pos* **and** *lim* **have** *ev*: *eventually* ($\lambda x. \text{norm } (f\ x / g\ x) < c$) *F*
 by (*subst (asm) tendsto-iff*) (*simp add: dist-real-def*)
with *assms(2)* **show** *eventually* ($\lambda x. (\text{norm } (f\ x)) \leq c * (\text{norm } (g\ x))$) *F*
 by *eventually-elim* (*simp add: field-simps norm-divide*)
 qed

lemma *smalloD-tendsto*:
 assumes $f \in o[F](g)$
 shows ($\lambda x. f\ x / g\ x \longrightarrow 0$) *F*
unfolding *tendsto-iff*
proof *clarify*
 fix *e* :: *real* **assume** *e*: $e > 0$

hence $e/2 > 0$ **by** *simp*
 from *landau-o.smallD*[*OF assms this*] **show** eventually $(\lambda x. \text{dist } (f x / g x) 0 < e)$ *F*
proof *eventually-elim*
 fix x **assume** $(\text{norm } (f x)) \leq e/2 * (\text{norm } (g x))$
 with e **have** $\text{dist } (f x / g x) 0 \leq e/2$
 by (*cases* $g x = 0$) (*simp-all add: dist-real-def norm-divide field-simps*)
 also from e **have** $\dots < e$ **by** *simp*
 finally **show** $\text{dist } (f x / g x) 0 < e$ **by** *simp*
qed
qed

lemma *bigthetaI-tendsto-norm*:
 assumes $c\text{-not-0}$: $(c::\text{real}) \neq 0$
 assumes *lim*: $((\lambda x. \text{norm } (f x / g x)) \longrightarrow c)$ *F*
 shows $f \in \Theta[F](g)$
proof (*rule bigthetaI*)
 from $c\text{-not-0}$ **have** $|c| > 0$ **by** *simp*
 with *lim* **have** eventually $(\lambda x. \text{norm } (\text{norm } (f x / g x) - c) < |c|)$ *F*
 by (*subst (asm) tendsto-iff*) (*simp add: dist-real-def*)
 hence g : eventually $(\lambda x. g x \neq 0)$ *F* **by** *eventually-elim (auto simp add: field-simps)*

 from *lim g* **show** $f \in O[F](g)$ **by** (*rule bigoI-tendsto-norm*)
 from $c\text{-not-0}$ and *lim* **show** $f \in \Omega[F](g)$ **by** (*rule bigomegaI-tendsto-norm*)
qed

lemma *bigthetaI-tendsto*:
 assumes $c\text{-not-0}$: $(c::\text{real}) \neq 0$
 assumes *lim*: $((\lambda x. f x / g x) \longrightarrow c)$ *F*
 shows $f \in \Theta[F](g)$
 using *assms* **by** (*intro bigthetaI-tendsto-norm*[*OF - tendsto-norm, of c*]) *simp-all*

lemma *tendsto-add-smallo*:
 assumes $(f1 \longrightarrow a)$ *F*
 assumes $f2 \in o[F](f1)$
 shows $((\lambda x. f1 x + f2 x) \longrightarrow a)$ *F*
proof (*subst filterlim-cong*[*OF refl refl*])
 from *landau-o.smallD*[*OF assms(2) zero-less-one*]
 have eventually $(\lambda x. \text{norm } (f2 x) \leq \text{norm } (f1 x))$ *F* **by** *simp*
 thus eventually $(\lambda x. f1 x + f2 x = f1 x * (1 + f2 x / f1 x))$ *F*
 by *eventually-elim (auto simp: field-simps)*
next
 from *assms(1)* **show** $((\lambda x. f1 x * (1 + f2 x / f1 x)) \longrightarrow a)$ *F*
 by (*force intro: tendsto-eq-intros smalloD-tendsto*[*OF assms(2)*])
qed

lemma *tendsto-diff-smallo*:
 shows $(f1 \longrightarrow a)$ *F* $\implies f2 \in o[F](f1) \implies ((\lambda x. f1 x - f2 x) \longrightarrow a)$ *F*
 using *tendsto-add-smallo*[*of f1 a F $\lambda x. -f2 x$*] **by** *simp*

lemma *tendsto-add-smallo-iff*:

assumes $f2 \in o[F](f1)$

shows $(f1 \longrightarrow a) F \longleftrightarrow ((\lambda x. f1\ x + f2\ x) \longrightarrow a) F$

proof

assume $((\lambda x. f1\ x + f2\ x) \longrightarrow a) F$

hence $((\lambda x. f1\ x + f2\ x - f2\ x) \longrightarrow a) F$

by (*rule tendsto-diff-smallo*) (*simp add: landau-o.small.plus-absorb2 assms*)

thus $(f1 \longrightarrow a) F$ **by** *simp*

qed (*rule tendsto-add-smallo[OF - assms]*)

lemma *tendsto-diff-smallo-iff*:

shows $f2 \in o[F](f1) \implies (f1 \longrightarrow a) F \longleftrightarrow ((\lambda x. f1\ x - f2\ x) \longrightarrow a) F$

using *tendsto-add-smallo-iff* [*of $\lambda x. -f2\ x\ F\ f1\ a$*] **by** *simp*

lemma *tendsto-divide-smallo*:

assumes $((\lambda x. f1\ x / g1\ x) \longrightarrow a) F$

assumes $f2 \in o[F](f1)$ $g2 \in o[F](g1)$

assumes *eventually* $(\lambda x. g1\ x \neq 0) F$

shows $((\lambda x. (f1\ x + f2\ x) / (g1\ x + g2\ x)) \longrightarrow a) F$ (**is** $(?f \longrightarrow -) -$)

proof (*subst tendsto-cong*)

let $?f' = \lambda x. (f1\ x / g1\ x) * (1 + f2\ x / f1\ x) / (1 + g2\ x / g1\ x)$

have $(?f' \longrightarrow a * (1 + 0) / (1 + 0)) F$

by (*rule tendsto-mult tendsto-divide tendsto-add assms tendsto-const*

smalloD-tendsto [*OF assms(2)*] *smalloD-tendsto* [*OF assms(3)*]) **+** *simp-all*

thus $(?f' \longrightarrow a) F$ **by** *simp*

have $(1/2::real) > 0$ **by** *simp*

from *landau-o.smallD* [*OF assms(2) this*] *landau-o.smallD* [*OF assms(3) this*]

have *eventually* $(\lambda x. \text{norm } (f2\ x) \leq \text{norm } (f1\ x)/2) F$

eventually $(\lambda x. \text{norm } (g2\ x) \leq \text{norm } (g1\ x)/2) F$ **by** *simp-all*

with *assms(4)* **show** *eventually* $(\lambda x. ?f\ x = ?f'\ x) F$

proof *eventually-elim*

fix x **assume** $A: \text{norm } (f2\ x) \leq \text{norm } (f1\ x)/2$ **and**

$B: \text{norm } (g2\ x) \leq \text{norm } (g1\ x)/2$ **and** $C: g1\ x \neq 0$

show $?f\ x = ?f'\ x$

proof (*cases* $f1\ x = 0$)

assume $D: f1\ x \neq 0$

from D **have** $f1\ x + f2\ x = f1\ x * (1 + f2\ x/f1\ x)$ **by** (*simp add: field-simps*)

moreover from C **have** $g1\ x + g2\ x = g1\ x * (1 + g2\ x/g1\ x)$ **by** (*simp*

add: field-simps)

ultimately have $?f\ x = (f1\ x * (1 + f2\ x/f1\ x)) / (g1\ x * (1 + g2\ x/g1\ x))$

by (*simp only:*)

also have $\dots = ?f'\ x$ **by** *simp*

finally show *?thesis* .

qed (*insert A, simp*)

qed

qed

lemma *bigo-powr*:
fixes $f :: 'a \Rightarrow \text{real}$
assumes $f \in O[F](g) \ p \geq 0$
shows $(\lambda x. |f x| \text{ powr } p) \in O[F](\lambda x. |g x| \text{ powr } p)$
proof –
from *assms(1)* **obtain** c **where** $c: c > 0$ **and** $*$: $\forall_F x \text{ in } F. \text{ norm } (f x) \leq c * \text{ norm } (g x)$
by (*elim landau-o.bigE landau-omega.bigE IntE*)
from *assms(2)* *** have** *eventually* $(\lambda x. (\text{norm } (f x)) \text{ powr } p \leq (c * \text{ norm } (g x)) \text{ powr } p) \ F$
by (*auto elim!: eventually-mono intro!: powr-mono2*)
with c **show** $(\lambda x. |f x| \text{ powr } p) \in O[F](\lambda x. |g x| \text{ powr } p)$
by (*intro bigOI[of - c powr p] (simp-all add: powr-mult)*)
qed

lemma *smallo-powr*:
fixes $f :: 'a \Rightarrow \text{real}$
assumes $f \in o[F](g) \ p > 0$
shows $(\lambda x. |f x| \text{ powr } p) \in o[F](\lambda x. |g x| \text{ powr } p)$
proof (*rule landau-o.smallI*)
fix $c :: \text{real}$ **assume** $c: c > 0$
hence $c \text{ powr } (1/p) > 0$ **by** *simp*
from *landau-o.smallD[OF assms(1) this]*
show *eventually* $(\lambda x. \text{ norm } (|f x| \text{ powr } p) \leq c * \text{ norm } (|g x| \text{ powr } p)) \ F$
proof *eventually-elim*
fix x **assume** $(\text{ norm } (f x)) \leq c \text{ powr } (1 / p) * (\text{ norm } (g x))$
with *assms(2)* **have** $(\text{ norm } (f x)) \text{ powr } p \leq (c \text{ powr } (1 / p) * (\text{ norm } (g x))) \text{ powr } p$
by (*intro powr-mono2 simp-all*)
also from *assms(2)* c **have** $\dots = c * (\text{ norm } (g x)) \text{ powr } p$
by (*simp add: field-simps powr-mult powr-powr*)
finally show $\text{ norm } (|f x| \text{ powr } p) \leq c * \text{ norm } (|g x| \text{ powr } p)$ **by** *simp*
qed
qed

lemma *smallo-powr-nonneg*:
fixes $f :: 'a \Rightarrow \text{real}$
assumes $f \in o[F](g) \ p > 0$ *eventually* $(\lambda x. f x \geq 0) \ F$ *eventually* $(\lambda x. g x \geq 0) \ F$
shows $(\lambda x. f x \text{ powr } p) \in o[F](\lambda x. g x \text{ powr } p)$
proof –
from *assms(3)* **have** $(\lambda x. f x \text{ powr } p) \in \Theta[F](\lambda x. |f x| \text{ powr } p)$
by (*intro bigthetaI-cong (auto elim!: eventually-mono)*)
also have $(\lambda x. |f x| \text{ powr } p) \in o[F](\lambda x. |g x| \text{ powr } p)$ **by** (*intro smallo-powr*)
fact+
also from *assms(4)* **have** $(\lambda x. |g x| \text{ powr } p) \in \Theta[F](\lambda x. g x \text{ powr } p)$
by (*intro bigthetaI-cong (auto elim!: eventually-mono)*)

finally show *?thesis* .
qed

lemma *bigheta-powr*:
 fixes $f :: 'a \Rightarrow \text{real}$
 shows $f \in \Theta[F](g) \implies (\lambda x. |f\ x| \text{ powr } p) \in \Theta[F](\lambda x. |g\ x| \text{ powr } p)$
apply (*cases* $p < 0$)
apply (*subst bigheta-inverse[symmetric]*, *subst* (*1 2*) *powr-minus[symmetric]*)
unfolding *bigheta-def* **apply** (*auto simp: bigomega-iff-bigo intro!: bigo-powr*)
done

lemma *bigo-powr-nonneg*:
 fixes $f :: 'a \Rightarrow \text{real}$
 assumes $f \in O[F](g)$ $p \geq 0$ *eventually* $(\lambda x. f\ x \geq 0)$ F *eventually* $(\lambda x. g\ x \geq 0)$
 F
 shows $(\lambda x. f\ x \text{ powr } p) \in O[F](\lambda x. g\ x \text{ powr } p)$
proof –
 from *assms*(3) **have** $(\lambda x. f\ x \text{ powr } p) \in \Theta[F](\lambda x. |f\ x| \text{ powr } p)$
 by (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)
 also **have** $(\lambda x. |f\ x| \text{ powr } p) \in O[F](\lambda x. |g\ x| \text{ powr } p)$ **by** (*intro bigo-powr*) *fact+*
 also from *assms*(4) **have** $(\lambda x. |g\ x| \text{ powr } p) \in \Theta[F](\lambda x. g\ x \text{ powr } p)$
 by (*intro bighetaI-cong*) (*auto elim!: eventually-mono*)
finally show *?thesis* .
qed

lemma *zero-in-smallo* [*simp*]: $(\lambda-. 0) \in o[F](f)$
 by (*intro landau-o.smallI*) *simp-all*

lemma *zero-in-bigo* [*simp*]: $(\lambda-. 0) \in O[F](f)$
 by (*intro landau-o.bigI* [*of 1*]) *simp-all*

lemma *in-bigomega-zero* [*simp*]: $f \in \Omega[F](\lambda x. 0)$
 by (*rule landau-omega.bigI* [*of 1*]) *simp-all*

lemma *in-smallomega-zero* [*simp*]: $f \in \omega[F](\lambda x. 0)$
 by (*simp add: smallomega-iff-smallo*)

lemma *in-smallo-zero-iff* [*simp*]: $f \in o[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) \ F$
proof
 assume $f \in o[F](\lambda x. 0)$
 from *landau-o.smallD* [*OF this, of 1*] **show** *eventually* $(\lambda x. f\ x = 0) \ F$ **by** *simp*
next
 assume *eventually* $(\lambda x. f\ x = 0) \ F$
 hence $\forall c > 0. \text{eventually } (\lambda x. (\text{norm } (f\ x)) \leq c * |0|) \ F$ **by** *simp*
 thus $f \in o[F](\lambda x. 0)$ **unfolding** *smallo-def* **by** *simp*
qed

lemma *in-bigo-zero-iff* [*simp*]: $f \in O[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f\ x = 0) \ F$

proof

assume $f \in O[F](\lambda-. 0)$
 thus *eventually* $(\lambda x. f x = 0)$ F **by** (*elim landau-o.bigE*) *simp*
next
 assume *eventually* $(\lambda x. f x = 0)$ F
 hence *eventually* $(\lambda x. (\text{norm } (f x)) \leq 1 * |0|)$ F **by** *simp*
 thus $f \in O[F](\lambda-. 0)$ **by** (*intro landau-o.bigI[of 1]*) *simp-all*
qed

lemma *zero-in-smallomega-iff* [simp]: $(\lambda-. 0) \in \omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$ F
 by (*simp add: smallomega-iff-smallo*)

lemma *zero-in-bigomega-iff* [simp]: $(\lambda-. 0) \in \Omega[F](f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$ F
 by (*simp add: bigomega-iff-bigo*)

lemma *zero-in-bigtheta-iff* [simp]: $(\lambda-. 0) \in \Theta[F](f) \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$ F
 unfolding *bigtheta-def* **by** *simp*

lemma *in-bigtheta-zero-iff* [simp]: $f \in \Theta[F](\lambda x. 0) \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$ F
 unfolding *bigtheta-def* **by** *simp*

lemma *cmult-in-bigo-iff* [simp]: $(\lambda x. c * f x) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$
 and *cmult-in-bigo-iff'* [simp]: $(\lambda x. f x * c) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$
 and *cmult-in-smallo-iff* [simp]: $(\lambda x. c * f x) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
 and *cmult-in-smallo-iff'* [simp]: $(\lambda x. f x * c) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
 by (*cases c = 0, simp, simp*)**+**

lemma *bigo-const* [simp]: $(\lambda-. c) \in O[F](\lambda-. 1)$ **by** (*rule bigoI[of - norm c]*) *simp*

lemma *bigo-const-iff* [simp]: $(\lambda-. c1) \in O[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 = 0 \vee c2 \neq 0$
 by (*cases c1 = 0; cases c2 = 0*)
 (*auto simp: bigo-def eventually-False intro: exI[of - 1] exI[of - norm c1 / norm c2]*)

lemma *bigomega-const-iff* [simp]: $(\lambda-. c1) \in \Omega[F](\lambda-. c2) \longleftrightarrow F = \text{bot} \vee c1 \neq 0 \vee c2 = 0$
 by (*cases c1 = 0; cases c2 = 0*)
 (*auto simp: bigomega-def eventually-False mult-le-0-iff intro: exI[of - 1] exI[of - norm c1 / norm c2]*)

lemma *smallo-real-nat-transfer*:
 $(f :: \text{real} \Rightarrow \text{real}) \in o(g) \Longrightarrow (\lambda x :: \text{nat}. f (\text{real } x)) \in o(\lambda x. g (\text{real } x))$

by (rule landau-o.small.compose[OF - filterlim-real-sequentially])

lemma bigo-real-nat-transfer:

$(f :: \text{real} \Rightarrow \text{real}) \in O(g) \Longrightarrow (\lambda x :: \text{nat}. f \text{ (real } x)) \in O(\lambda x. g \text{ (real } x))$

by (rule landau-o.big.compose[OF - filterlim-real-sequentially])

lemma smallomega-real-nat-transfer:

$(f :: \text{real} \Rightarrow \text{real}) \in \omega(g) \Longrightarrow (\lambda x :: \text{nat}. f \text{ (real } x)) \in \omega(\lambda x. g \text{ (real } x))$

by (rule landau-omega.small.compose[OF - filterlim-real-sequentially])

lemma bigomega-real-nat-transfer:

$(f :: \text{real} \Rightarrow \text{real}) \in \Omega(g) \Longrightarrow (\lambda x :: \text{nat}. f \text{ (real } x)) \in \Omega(\lambda x. g \text{ (real } x))$

by (rule landau-omega.big.compose[OF - filterlim-real-sequentially])

lemma bigtheta-real-nat-transfer:

$(f :: \text{real} \Rightarrow \text{real}) \in \Theta(g) \Longrightarrow (\lambda x :: \text{nat}. f \text{ (real } x)) \in \Theta(\lambda x. g \text{ (real } x))$

unfolding bigtheta-def **using** bigo-real-nat-transfer bigomega-real-nat-transfer
by blast

lemmas landau-real-nat-transfer [intro] =

bigo-real-nat-transfer smallo-real-nat-transfer bigomega-real-nat-transfer

smallomega-real-nat-transfer bigtheta-real-nat-transfer

lemma landau-symbol-if-at-top-eq [simp]:

assumes landau-symbol $L \ L' \ Lr$

shows $L \text{ at-top } (\lambda x :: 'a :: \text{linordered-semidom}. \text{ if } x = a \text{ then } f \ x \text{ else } g \ x) = L$
 $\text{at-top } (g)$

apply (rule landau-symbol.cong[OF assms])

by (auto simp add: frequently-def eventually-at-top-linorder dest!: spec [where
 $x = a + 1$])

lemmas landau-symbols-if-at-top-eq [simp] = landau-symbols[THEN landau-symbol-if-at-top-eq]

lemma sum-in-smallo:

assumes $f \in o[F](h) \ g \in o[F](h)$

shows $(\lambda x. f \ x + g \ x) \in o[F](h) \ (\lambda x. f \ x - g \ x) \in o[F](h)$

proof –

have $(\lambda x. f \ x + g \ x) \in o[F](h)$ **if** $f \in o[F](h) \ g \in o[F](h)$ **for** $f \ g$

proof (rule landau-o.smallI)

fix $c :: \text{real}$ **assume** $c > 0$

hence $c/2 > 0$ **by** simp

from that[THEN landau-o.smallD[OF - this]]

show eventually $(\lambda x. \text{norm } (f \ x + g \ x) \leq c * (\text{norm } (h \ x))) \ F$

by eventually-elim (auto intro: order.trans[OF norm-triangle-ineq])

qed

from this[of $f \ g$] this[of $f \ \lambda x. -g \ x$] assms

show $(\lambda x. f x + g x) \in o[F](h) \ (\lambda x. f x - g x) \in o[F](h)$ **by** *simp-all*
qed

lemma *big-sum-in-smallo*:

assumes $\bigwedge x. x \in A \implies f x \in o[F](g)$
shows $(\lambda x. \text{sum } (\lambda y. f y x) A) \in o[F](g)$
using *assms* **by** (*induction A rule: infinite-finite-induct*) (*auto intro: sum-in-smallo*)

lemma *sum-in-bigo*:

assumes $f \in O[F](h) \ g \in O[F](h)$
shows $(\lambda x. f x + g x) \in O[F](h) \ (\lambda x. f x - g x) \in O[F](h)$
proof –
have $(\lambda x. f x + g x) \in O[F](h)$ **if** $f \in O[F](h) \ g \in O[F](h)$ **for** $f \ g$
proof –
from *that* **obtain** $c1 \ c2$ **where** $*: c1 > 0 \ c2 > 0$
and $**: \forall_F x \text{ in } F. \text{norm } (f x) \leq c1 * \text{norm } (h x)$
 $\forall_F x \text{ in } F. \text{norm } (g x) \leq c2 * \text{norm } (h x)$
by (*elim landau-o.bigE*)
from $**$ **have** *eventually* $(\lambda x. \text{norm } (f x + g x) \leq (c1 + c2) * (\text{norm } (h x)))$
F
by *eventually-elim* (*auto simp: algebra-simps intro: order.trans[OF norm-triangle-ineq]*)
then show *?thesis* **by** (*rule bigoI*)
qed
from *assms* *this*[*of f g*] *this*[*of f* $\lambda x. - g x$]
show $(\lambda x. f x + g x) \in O[F](h) \ (\lambda x. f x - g x) \in O[F](h)$ **by** *simp-all*
qed

lemma *big-sum-in-bigo*:

assumes $\bigwedge x. x \in A \implies f x \in O[F](g)$
shows $(\lambda x. \text{sum } (\lambda y. f y x) A) \in O[F](g)$
using *assms* **by** (*induction A rule: infinite-finite-induct*) (*auto intro: sum-in-bigo*)

lemma *smallo-multiples*:

assumes $f: f \in o(\text{real})$ **and** $k > 0$
shows $(\lambda n. f (k * n)) \in o(\text{real})$
proof (*clarsimp simp: smallo-def*)
fix $c::\text{real}$
assume $c > 0$
then have $c/k > 0$
by (*simp add: assms*)
with *assms* **have** $\forall_F n \text{ in sequentially. } |f n| \leq c / \text{real } k * n$
by (*force simp: smallo-def del: divide-const-simps*)
then obtain N **where** $\bigwedge n. n \geq N \implies |f n| \leq c/k * n$
by (*meson eventually-at-top-linorder*)
then have $\bigwedge m. (k*m) \geq N \implies |f (k*m)| \leq c/k * (k*m)$
by *blast*
with $\langle k > 0 \rangle$ **have** $\forall_F m \text{ in sequentially. } |f (k*m)| \leq c/k * (k*m)$
by (*smt (verit, del-insts) One-nat-def Suc-leI eventually-at-top-linorderI mult-1-left mult-le-mono*)

then show $\forall_F n$ in sequentially. $|f(k * n)| \leq c * n$
 by eventually-elim (use $\langle k > 0 \rangle$ in auto)
 qed

lemma maxmin-in-smallo:

assumes $f \in o[F](h)$ $g \in o[F](h)$
 shows $(\lambda k. \max(f k) (g k)) \in o[F](h)$ $(\lambda k. \min(f k) (g k)) \in o[F](h)$
 proof –
 have $\forall_F x$ in F . $\text{norm}(\max(f x) (g x)) \leq c * \text{norm}(h x) \wedge \text{norm}(\min(f x) (g x)) \leq c * \text{norm}(h x)$
 if $c > 0$ for $c :: \text{real}$
 proof –
 from assms smallo-def that
 have $\forall_F x$ in F . $\text{norm}(f x) \leq c * \text{norm}(h x) \wedge \forall_F x$ in F . $\text{norm}(g x) \leq c * \text{norm}(h x)$
 by (auto simp: smallo-def)
 then show ?thesis
 by (smt (verit) eventually-elim2 max-def min-def)
 qed
 with assms show $(\lambda x. \max(f x) (g x)) \in o[F](h)$ $(\lambda x. \min(f x) (g x)) \in o[F](h)$
 by (smt (verit) eventually-elim2 landau-o.smallI)+
 qed

lemma le-imp-bigo-real:

assumes $c \geq 0$ eventually $(\lambda x. f x \leq c * (g x :: \text{real}))$ F eventually $(\lambda x. 0 \leq f x)$ F
 shows $f \in O[F](g)$
 proof –
 have eventually $(\lambda x. \text{norm}(f x) \leq c * \text{norm}(g x))$ F
 using assms(2,3)
 proof eventually-elim
 case (elim x)
 have $\text{norm}(f x) \leq c * g x$ using elim by simp
 also have $\dots \leq c * \text{norm}(g x)$ by (intro mult-left-mono assms) auto
 finally show ?case .
 qed
 thus ?thesis by (intro bigoI[of - c]) auto
 qed

context landau-symbol

begin

lemma mult-cancel-left:

assumes $f1 \in \Theta[F](g1)$ and eventually $(\lambda x. g1 x \neq 0)$ F
 notes $[trans] = \text{bigtheta-trans1 bigtheta-trans2}$
 shows $(\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x) \longleftrightarrow f2 \in L F (g2)$
 proof
 assume $A: (\lambda x. f1 x * f2 x) \in L F (\lambda x. g1 x * g2 x)$
 from assms have $\text{nz}: \text{eventually } (\lambda x. f1 x \neq 0)$ F by (simp add: eventu-

ally-nonzero-bigtheta)

hence $f2 \in \Theta[F](\lambda x. f1\ x * f2\ x / f1\ x)$

by (*intro bigthetaI-cong*) (*auto elim!:* *eventually-mono*)

also from *A assms nz* **have** $(\lambda x. f1\ x * f2\ x / f1\ x) \in L\ F\ (\lambda x. g1\ x * g2\ x / f1\ x)$

by (*intro divide-right*) *simp-all*

also from *assms nz* **have** $(\lambda x. g1\ x * g2\ x / f1\ x) \in \Theta[F](\lambda x. g1\ x * g2\ x / g1\ x)$

by (*intro landau-theta.mult landau-theta.divide*) (*simp-all add: bigtheta-sym*)

also from *assms* **have** $(\lambda x. g1\ x * g2\ x / g1\ x) \in \Theta[F](g2)$

by (*intro bigthetaI-cong*) (*auto elim!:* *eventually-mono*)

finally show $f2 \in L\ F\ (g2)$.

next

assume $f2 \in L\ F\ (g2)$

hence $(\lambda x. f1\ x * f2\ x) \in L\ F\ (\lambda x. f1\ x * g2\ x)$ **by** (*rule mult-left*)

also have $(\lambda x. f1\ x * g2\ x) \in \Theta[F](\lambda x. g1\ x * g2\ x)$

by (*intro landau-theta.mult-right assms*)

finally show $(\lambda x. f1\ x * f2\ x) \in L\ F\ (\lambda x. g1\ x * g2\ x)$.

qed

lemma *mult-cancel-right*:

assumes $f2 \in \Theta[F](g2)$ **and** *eventually* $(\lambda x. g2\ x \neq 0)\ F$

shows $(\lambda x. f1\ x * f2\ x) \in L\ F\ (\lambda x. g1\ x * g2\ x) \longleftrightarrow f1 \in L\ F\ (g1)$

by (*subst (1 2) mult.commute*) (*rule mult-cancel-left[OF assms]*)

lemma *divide-cancel-right*:

assumes $f2 \in \Theta[F](g2)$ **and** *eventually* $(\lambda x. g2\ x \neq 0)\ F$

shows $(\lambda x. f1\ x / f2\ x) \in L\ F\ (\lambda x. g1\ x / g2\ x) \longleftrightarrow f1 \in L\ F\ (g1)$

by (*subst (1 2) divide-inverse, intro mult-cancel-right bigtheta-inverse*) (*simp-all add: assms*)

lemma *divide-cancel-left*:

assumes $f1 \in \Theta[F](g1)$ **and** *eventually* $(\lambda x. g1\ x \neq 0)\ F$

shows $(\lambda x. f1\ x / f2\ x) \in L\ F\ (\lambda x. g1\ x / g2\ x) \longleftrightarrow$

$(\lambda x. \text{inverse}\ (f2\ x)) \in L\ F\ (\lambda x. \text{inverse}\ (g2\ x))$

by (*simp only: divide-inverse mult-cancel-left[OF assms]*)

end

lemma *powr-smallo-iff*:

assumes *filterlim g at-top* $F\ F \neq \text{bot}$

shows $(\lambda x. g\ x\ \text{powr}\ p :: \text{real}) \in o[F](\lambda x. g\ x\ \text{powr}\ q) \longleftrightarrow p < q$

proof –

from *assms* **have** *eventually* $(\lambda x. g\ x \geq 1)\ F$ **by** (*force simp: filterlim-at-top*)

hence *A: eventually* $(\lambda x. g\ x \neq 0)\ F$ **by** *eventually-elim simp*

have *B:* $(\lambda x. g\ x\ \text{powr}\ q) \in O[F](\lambda x. g\ x\ \text{powr}\ p) \implies (\lambda x. g\ x\ \text{powr}\ p) \notin o[F](\lambda x. g\ x\ \text{powr}\ q)$

proof

```

    assume  $(\lambda x. g \ x \ \text{powr} \ q) \in O[F](\lambda x. g \ x \ \text{powr} \ p) \ (\lambda x. g \ x \ \text{powr} \ p) \in o[F](\lambda x. g \ x \ \text{powr} \ q)$ 
    from landau-o.big-small-asymmetric[OF this] have eventually  $(\lambda x. g \ x = 0) \ F$ 
  by simp
    with A have eventually  $(\lambda :: 'a. \text{False}) \ F$  by eventually-elim simp
    thus False by (simp add: eventually-False assms)
  qed
  show ?thesis
  proof (cases  $p \ q$  rule: linorder-cases)
    assume  $p < q$ 
    hence  $(\lambda x. g \ x \ \text{powr} \ p) \in o[F](\lambda x. g \ x \ \text{powr} \ q)$  using assms A
    by (auto intro!: smalloI-tendsto tendsto-neg-powr simp flip: powr-diff)
    with  $\langle p < q \rangle$  show ?thesis by auto
  next
    assume  $p = q$ 
    hence  $(\lambda x. g \ x \ \text{powr} \ q) \in O[F](\lambda x. g \ x \ \text{powr} \ p)$  by (auto intro!: bignthetaD1)
    with  $B \ \langle p = q \rangle$  show ?thesis by auto
  next
    assume  $p > q$ 
    hence  $(\lambda x. g \ x \ \text{powr} \ q) \in O[F](\lambda x. g \ x \ \text{powr} \ p)$  using assms A
    by (auto intro!: smalloI-tendsto tendsto-neg-powr landau-o.small-imp-big simp flip: powr-diff)
    with  $B \ \langle p > q \rangle$  show ?thesis by auto
  qed
qed

lemma powr-bigo-iff:
  assumes filterlim g at-top  $F \ F \neq \text{bot}$ 
  shows  $(\lambda x. g \ x \ \text{powr} \ p :: \text{real}) \in O[F](\lambda x. g \ x \ \text{powr} \ q) \longleftrightarrow p \leq q$ 
  proof-
    from assms have eventually  $(\lambda x. g \ x \geq 1) \ F$  by (force simp: filterlim-at-top)
    hence A: eventually  $(\lambda x. g \ x \neq 0) \ F$  by eventually-elim simp
    have B:  $(\lambda x. g \ x \ \text{powr} \ q) \in o[F](\lambda x. g \ x \ \text{powr} \ p) \implies (\lambda x. g \ x \ \text{powr} \ p) \notin O[F](\lambda x. g \ x \ \text{powr} \ q)$ 
  proof
    assume  $(\lambda x. g \ x \ \text{powr} \ q) \in o[F](\lambda x. g \ x \ \text{powr} \ p) \ (\lambda x. g \ x \ \text{powr} \ p) \in O[F](\lambda x. g \ x \ \text{powr} \ q)$ 
    from landau-o.small-big-asymmetric[OF this] have eventually  $(\lambda x. g \ x = 0) \ F$ 
  by simp
    with A have eventually  $(\lambda :: 'a. \text{False}) \ F$  by eventually-elim simp
    thus False by (simp add: eventually-False assms)
  qed
  show ?thesis
  proof (cases  $p \ q$  rule: linorder-cases)
    assume  $p < q$ 
    hence  $(\lambda x. g \ x \ \text{powr} \ p) \in o[F](\lambda x. g \ x \ \text{powr} \ q)$  using assms A
    by (auto intro!: smalloI-tendsto tendsto-neg-powr simp flip: powr-diff)
    with  $\langle p < q \rangle$  show ?thesis by (auto intro: landau-o.small-imp-big)
  next

```

```

assume  $p = q$ 
hence  $(\lambda x. g \ x \ \text{powr} \ q) \in O[F](\lambda x. g \ x \ \text{powr} \ p)$  by  $(\text{auto intro!}: \text{bigthetaD1})$ 
with  $B \langle p = q \rangle$  show  $?thesis$  by  $\text{auto}$ 
next
assume  $p > q$ 
hence  $(\lambda x. g \ x \ \text{powr} \ q) \in o[F](\lambda x. g \ x \ \text{powr} \ p)$  using  $\text{assms } A$ 
by  $(\text{auto intro!}: \text{smalloI-tendsto tendsto-neg-powr simp flip: powr-diff})$ 
with  $B \langle p > q \rangle$  show  $?thesis$  by  $(\text{auto intro: landau-o.small-imp-big})$ 
qed
qed

lemma  $\text{powr-bigtheta-iff}$ :
assumes  $\text{filterlim } g \ \text{at-top } F \ F \neq \text{bot}$ 
shows  $(\lambda x. g \ x \ \text{powr} \ p :: \text{real}) \in \Theta[F](\lambda x. g \ x \ \text{powr} \ q) \longleftrightarrow p = q$ 
using  $\text{assms}$  unfolding  $\text{bigtheta-def}$  by  $(\text{auto simp: bigomega-iff-bigo powr-bigo-iff})$ 

```

54.3 Flatness of real functions

Given two real-valued functions f and g , we say that f is flatter than g if any power of $f(x)$ is asymptotically dominated by any positive power of $g(x)$. This is a useful notion since, given two products of powers of functions sorted by flatness, we can compare them asymptotically by simply comparing the exponent lists lexicographically.

A simple sufficient criterion for flatness is that $\ln f(x) \in o(\ln g(x))$, which we show now.

```

lemma  $\text{ln-smallo-imp-flat}$ :
fixes  $f \ g :: \text{real} \Rightarrow \text{real}$ 
assumes  $\text{lim-f: filterlim } f \ \text{at-top at-top}$ 
assumes  $\text{lim-g: filterlim } g \ \text{at-top at-top}$ 
assumes  $\text{ln-o-ln: } (\lambda x. \ln (f \ x)) \in o(\lambda x. \ln (g \ x))$ 
assumes  $q: q > 0$ 
shows  $(\lambda x. f \ x \ \text{powr} \ p) \in o(\lambda x. g \ x \ \text{powr} \ q)$ 
proof  $(\text{rule smalloI-tendsto})$ 
from  $\text{lim-f}$  have  $\text{eventually } (\lambda x. f \ x > 0) \ \text{at-top}$ 
by  $(\text{simp add: filterlim-at-top-dense})$ 
hence  $f\text{-nz: eventually } (\lambda x. f \ x \neq 0) \ \text{at-top}$  by  $\text{eventually-elim simp}$ 

from  $\text{lim-g}$  have  $g\text{-gt-1: eventually } (\lambda x. g \ x > 1) \ \text{at-top}$ 
by  $(\text{simp add: filterlim-at-top-dense})$ 
hence  $g\text{-nz: eventually } (\lambda x. g \ x \neq 0) \ \text{at-top}$  by  $\text{eventually-elim simp}$ 
thus  $\text{eventually } (\lambda x. g \ x \ \text{powr} \ q \neq 0) \ \text{at-top}$ 
by  $\text{eventually-elim simp}$ 

have  $\text{eq: eventually } (\lambda x. q * (p/q * (\ln (f \ x) / \ln (g \ x)) - 1) * \ln (g \ x) =$ 
 $p * \ln (f \ x) - q * \ln (g \ x)) \ \text{at-top}$ 
using  $g\text{-gt-1}$  by  $\text{eventually-elim (insert } q, \text{ simp-all add: field-simps)}$ 
have  $\text{filterlim } (\lambda x. q * (p/q * (\ln (f \ x) / \ln (g \ x)) - 1) * \ln (g \ x)) \ \text{at-bot at-top}$ 
by  $(\text{insert } q)$ 

```



```

(rule filterlim-tendsto-neg-mult-at-bot tendsto-mult
  tendsto-const tendsto-diff smalloD-tendsto[OF ln-o-ln] lim-g
  filterlim-compose[OF ln-at-top] | simp)+
hence filterlim ( $\lambda x. p * \ln (f x) - q * \ln (g x)$ ) at-bot at-top
  by (subst (asm) filterlim-cong[OF refl refl eq])
hence *: ( $(\lambda x. \exp (p * \ln (f x) - q * \ln (g x))) \longrightarrow 0$ ) at-top
  by (rule filterlim-compose[OF exp-at-bot])
have eq: eventually ( $\lambda x. \exp (p * \ln (f x) - q * \ln (g x)) = f x \text{ powr } p / g x \text{ powr } q$ ) at-top
  using f-nz g-nz by eventually-elim (simp add: powr-def exp-diff)
show (( $\lambda x. f x \text{ powr } p / g x \text{ powr } q$ )  $\longrightarrow 0$ ) at-top
  using * by (subst (asm) filterlim-cong[OF refl refl eq])
qed

```

```

lemma ln-smallo-imp-flat':
  fixes f g :: real  $\Rightarrow$  real
  assumes lim-f: filterlim f at-top at-top
  assumes lim-g: filterlim g at-top at-top
  assumes ln-o-ln: ( $\lambda x. \ln (f x) \in o(\lambda x. \ln (g x))$ )
  assumes q:  $q < 0$ 
  shows ( $\lambda x. g x \text{ powr } q \in o(\lambda x. f x \text{ powr } p)$ )
proof -
  from lim-f lim-g have eventually ( $\lambda x. f x > 0$ ) at-top eventually ( $\lambda x. g x > 0$ )
  at-top
  by (simp-all add: filterlim-at-top-dense)
hence eventually ( $\lambda x. f x \neq 0$ ) at-top eventually ( $\lambda x. g x \neq 0$ ) at-top
  by (auto elim: eventually-mono)
moreover from assms have ( $\lambda x. f x \text{ powr } -p \in o(\lambda x. g x \text{ powr } -q)$ )
  by (intro ln-smallo-imp-flat assms) simp-all
ultimately show ?thesis unfolding powr-minus
  by (simp add: landau-o.small.inverse-cancel)
qed

```

54.4 Asymptotic Equivalence

named-theorems *asympt-equiv-intros*

named-theorems *asympt-equiv-simps*

definition *asympt-equiv* :: ($'a \Rightarrow ('b :: \text{real-normed-field}) \Rightarrow 'a \text{ filter} \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$)

($\langle \langle \text{open-block notation} = \langle \text{mixfix asympt-equiv} \rangle - \sim[-] - \rangle [51, 10, 51] 50$)

where $f \sim[F] g \longleftrightarrow ((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1) F$

abbreviation (*input*) *asympt-equiv-at-top* **where**

asympt-equiv-at-top $f g \equiv f \sim[at-top] g$

bundle *asympt-equiv-syntax*

begin

notation *asympt-equiv-at-top* (**infix** $\langle \sim \rangle$ 50)

end

lemma *asympt-equivI*: $((\lambda x. \text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x) \longrightarrow 1) \implies f \sim[F] g$
by (*simp add: asympt-equiv-def*)

lemma *asympt-equivD*: $f \sim[F] g \implies ((\lambda x. \text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x) \longrightarrow 1) \ F$
by (*simp add: asympt-equiv-def*)

lemma *asympt-equiv-filtermap-iff*:
 $f \sim[\text{filtermap } h\ F] g \iff (\lambda x. f\ (h\ x)) \sim[F] (\lambda x. g\ (h\ x))$
by (*simp add: asympt-equiv-def filterlim-filtermap*)

lemma *asympt-equiv-refl* [*simp, asympt-equiv-intros*]: $f \sim[F] f$
proof (*intro asympt-equivI*)
have *eventually* $(\lambda x. 1 = (\text{if } f\ x = 0 \wedge f\ x = 0 \text{ then } 1 \text{ else } f\ x / f\ x))\ F$
by (*intro always-eventually simp*)
moreover **have** $((\lambda \cdot. 1) \longrightarrow 1)\ F$ **by** *simp*
ultimately show $((\lambda x. \text{if } f\ x = 0 \wedge f\ x = 0 \text{ then } 1 \text{ else } f\ x / f\ x) \longrightarrow 1)\ F$
by (*simp add: tendsto-eventually*)
qed

lemma *asympt-equiv-symI*:
assumes $f \sim[F] g$
shows $g \sim[F] f$
using *tendsto-inverse[OF asympt-equivD[OF assms]]*
by (*auto intro!: asympt-equivI simp: if-distrib conj-commute cong: if-cong*)

lemma *asympt-equiv-sym*: $f \sim[F] g \iff g \sim[F] f$
by (*blast intro: asympt-equiv-symI*)

lemma *asympt-equivI'*:
assumes $((\lambda x. f\ x / g\ x) \longrightarrow 1)\ F$
shows $f \sim[F] g$
proof (*cases F = bot*)
case False
have *eventually* $(\lambda x. f\ x \neq 0)\ F$
proof (*rule ccontr*)
assume $\neg \text{eventually } (\lambda x. f\ x \neq 0)\ F$
hence *frequently* $(\lambda x. f\ x = 0)\ F$ **by** (*simp add: frequently-def*)
hence *frequently* $(\lambda x. f\ x / g\ x = 0)\ F$ **by** (*auto elim!: frequently-elim1*)
from *limit-frequently-eq[OF False this assms]* **show** *False* **by** *simp-all*
qed
hence *eventually* $(\lambda x. f\ x / g\ x = (\text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x))\ F$
by *eventually-elim simp*
with *assms* **show** $f \sim[F] g$ **unfolding** *asympt-equiv-def*
by (*rule Lim-transform-eventually*)
qed (*simp-all add: asympt-equiv-def*)

lemma *tendsto-imp-asymp-equiv-const*:

assumes $(f \longrightarrow c) \ F \ c \neq 0$

shows $f \sim[F] (\lambda\cdot. c)$

by (rule *asymp-equivI'* *tendsto-eq-intros* *assms refl*) + (use *assms* **in** *auto*)

lemma *asymp-equiv-cong*:

assumes *eventually* $(\lambda x. f1\ x = f2\ x) \ F$ *eventually* $(\lambda x. g1\ x = g2\ x) \ F$

shows $f1 \sim[F] g1 \longleftrightarrow f2 \sim[F] g2$

unfolding *asymp-equiv-def*

proof (rule *tendsto-cong*, *goal-cases*)

case 1

from *assms* **show** ?*case* **by** *eventually-elim simp*

qed

lemma *asymp-equiv-eventually-zeros*:

fixes $f\ g :: 'a \Rightarrow 'b :: \text{real-normed-field}$

assumes $f \sim[F] g$

shows *eventually* $(\lambda x. f\ x = 0 \longleftrightarrow g\ x = 0) \ F$

proof –

let ?*h* = $\lambda x. \text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x$

have *eventually* $(\lambda x. x \neq 0) \ (nhds\ (1::'b))$

by (rule *t1-space-nhds*) *auto*

hence *eventually* $(\lambda x. x \neq 0) \ (\text{filtermap } ?h\ F)$

using *assms* **unfolding** *asymp-equiv-def filterlim-def*

by (rule *filter-leD [rotated]*)

hence *eventually* $(\lambda x. ?h\ x \neq 0) \ F$ **by** (*simp add: eventually-filtermap*)

thus ?*thesis* **by** *eventually-elim (auto split: if-splits)*

qed

lemma *asymp-equiv-transfer*:

assumes $f1 \sim[F] g1$ *eventually* $(\lambda x. f1\ x = f2\ x) \ F$ *eventually* $(\lambda x. g1\ x = g2\ x) \ F$

shows $f2 \sim[F] g2$

using *assms*(1) *asymp-equiv-cong[OF assms(2,3)]* **by** *simp*

lemma *asymp-equiv-transfer-trans [trans]*:

assumes $(\lambda x. f\ x (h1\ x)) \sim[F] (\lambda x. g\ x (h1\ x))$

assumes *eventually* $(\lambda x. h1\ x = h2\ x) \ F$

shows $(\lambda x. f\ x (h2\ x)) \sim[F] (\lambda x. g\ x (h2\ x))$

by (rule *asymp-equiv-transfer[OF assms(1)]*) (*insert assms(2), auto elim!: eventually-mono*)

lemma *asymp-equiv-trans [trans]*:

fixes $f\ g\ h$

assumes $f \sim[F] g$ $g \sim[F] h$

shows $f \sim[F] h$

proof –

let ?*T* = $\lambda f\ g\ x. \text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x$

from *tendsto-mult*[*OF* *assms*[*THEN* *asympt-equivD*]]
have $((\lambda x. ?T f g x * ?T g h x) \longrightarrow 1) F$ **by** *simp*
moreover from *assms*[*THEN* *asympt-equiv-eventually-zeros*]
have *eventually* $(\lambda x. ?T f g x * ?T g h x = ?T f h x) F$ **by** *eventually-elim simp*
ultimately show *?thesis* **unfolding** *asympt-equiv-def* **by** (rule *Lim-transform-eventually*)
qed

lemma *asympt-equiv-trans-lift1* [*trans*]:
assumes $a \sim[F] f b \ b \sim[F] c \ \wedge c \ d. \ c \sim[F] d \implies f c \sim[F] f d$
shows $a \sim[F] f c$
using *assms* **by** (blast intro: *asympt-equiv-trans*)

lemma *asympt-equiv-trans-lift2* [*trans*]:
assumes $f a \sim[F] b \ a \sim[F] c \ \wedge c \ d. \ c \sim[F] d \implies f c \sim[F] f d$
shows $f c \sim[F] b$
using *asympt-equiv-symI*[*OF* *assms*(3)[*OF* *assms*(2)]] *assms*(1)
by (blast intro: *asympt-equiv-trans*)

lemma *asympt-equivD-const*:
assumes $f \sim[F] (\lambda-. c)$
shows $(f \longrightarrow c) F$
proof (cases $c = 0$)
 case *False*
 with *tendsto-mult-right*[*OF* *asympt-equivD*[*OF* *assms*], of c] **show** *?thesis* **by** *simp*
next
 case *True*
 with *asympt-equiv-eventually-zeros*[*OF* *assms*] **show** *?thesis*
 by (*simp add: tendsto-eventually*)
qed

lemma *asympt-equiv-refl-ev*:
assumes *eventually* $(\lambda x. f x = g x) F$
shows $f \sim[F] g$
by (intro *asympt-equivI* *tendsto-eventually*)
 (insert *assms*, *auto elim!*: *eventually-mono*)

lemma *asympt-equiv-nhds-iff*: $f \sim[nhds (z :: 'a :: t1-space)] g \longleftrightarrow f \sim[at z] g \wedge f z = g z$
by (*auto simp: asympt-equiv-def tendsto-nhds-iff*)

lemma *asympt-equiv-sandwich*:
fixes $f g h :: 'a \Rightarrow 'b :: \{\text{real-normed-field, order-topology, linordered-field}\}$
assumes *eventually* $(\lambda x. f x \geq 0) F$
assumes *eventually* $(\lambda x. f x \leq g x) F$
assumes *eventually* $(\lambda x. g x \leq h x) F$
assumes $f \sim[F] h$
shows $g \sim[F] f \ g \sim[F] h$
proof –
 show $g \sim[F] f$

```

proof (rule asymp-equivI, rule tendsto-sandwich)
  from assms(1-3) asymp-equiv-eventually-zeros[OF assms(4)]
  show eventually ( $\lambda n. (if\ h\ n = 0 \wedge f\ n = 0\ then\ 1\ else\ h\ n / f\ n) \geq$ 
    ( $if\ g\ n = 0 \wedge f\ n = 0\ then\ 1\ else\ g\ n / f\ n$ )) F
  by eventually-elim (auto intro!: divide-right-mono)
  from assms(1-3) asymp-equiv-eventually-zeros[OF assms(4)]
  show eventually ( $\lambda n. 1 \leq$ 
    ( $if\ g\ n = 0 \wedge f\ n = 0\ then\ 1\ else\ g\ n / f\ n$ )) F
  by eventually-elim (auto intro!: divide-right-mono)
qed (insert asymp-equiv-symI[OF assms(4)], simp-all add: asymp-equiv-def)
also note  $\langle f \sim[F] h \rangle$ 
finally show  $g \sim[F] h$  .
qed

```

```

lemma asymp-equiv-imp-eventually-same-sign:
  fixes f g :: real  $\Rightarrow$  real
  assumes f  $\sim[F]$  g
  shows eventually ( $\lambda x. sgn\ (f\ x) = sgn\ (g\ x)$ ) F
proof -
  from assms have ( $(\lambda x. sgn\ (if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x / g\ x)) \longrightarrow$ 
    sgn 1) F
  unfolding asymp-equiv-def by (rule tendsto-sgn) simp-all
  from order-tendstoD(1)[OF this, of 1/2]
  have eventually ( $\lambda x. sgn\ (if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x / g\ x) > 1/2$ ) F
  by simp
  thus eventually ( $\lambda x. sgn\ (f\ x) = sgn\ (g\ x)$ ) F
proof eventually-elim
  case (elim x)
  thus ?case
  by (cases f x 0 :: real rule: linorder-cases;
    cases g x 0 :: real rule: linorder-cases) simp-all
qed
qed

```

```

lemma
  fixes f g :: -  $\Rightarrow$  real
  assumes f  $\sim[F]$  g
  shows asymp-equiv-eventually-same-sign: eventually ( $\lambda x. sgn\ (f\ x) = sgn\ (g\ x)$ ) F (is ?th1)
  and asymp-equiv-eventually-neg-iff: eventually ( $\lambda x. f\ x < 0 \longleftrightarrow g\ x < 0$ )
    F (is ?th2)
  and asymp-equiv-eventually-pos-iff: eventually ( $\lambda x. f\ x > 0 \longleftrightarrow g\ x > 0$ )
    F (is ?th3)
proof -
  from assms have filterlim ( $\lambda x. if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x / g\ x$ ) (nhds
    1) F
  by (rule asymp-equivD)
  from order-tendstoD(1)[OF this zero-less-one]
  show ?th1 ?th2 ?th3

```

by (eventually-elim; force simp: sgn-if field-split-simps split: if-splits)+
qed

lemma *asympt-equiv-tendsto-transfer*:

assumes $f \sim[F] g$ and $(f \longrightarrow c) F$

shows $(g \longrightarrow c) F$

proof –

let $?h = \lambda x. (if\ g\ x = 0 \wedge f\ x = 0\ then\ 1\ else\ g\ x / f\ x) * f\ x$

from *assms*(1) have $g \sim[F] f$ by (rule *asympt-equiv-symI*)

hence *filterlim* $(\lambda x. if\ g\ x = 0 \wedge f\ x = 0\ then\ 1\ else\ g\ x / f\ x) (nhds\ 1) F$

by (rule *asympt-equivD*)

from *tendsto-mult*[*OF this assms*(2)] have $(?h \longrightarrow c) F$ by *simp*

moreover

have *eventually* $(\lambda x. ?h\ x = g\ x) F$

using *asympt-equiv-eventually-zeros*[*OF assms*(1)] by *eventually-elim simp*

ultimately show *?thesis*

by (rule *Lim-transform-eventually*)

qed

lemma *tendsto-asympt-equiv-cong*:

assumes $f \sim[F] g$

shows $(f \longrightarrow c) F \longleftrightarrow (g \longrightarrow c) F$

proof –

have $(f \longrightarrow c * 1) F$ if $fg: f \sim[F] g$ and $(g \longrightarrow c) F$ for $f\ g :: 'a \Rightarrow 'b$

proof –

from *that* have $*$: $((\lambda x. g\ x * (if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x / g\ x)) \longrightarrow c * 1) F$

by (*intro tendsto-intros asympt-equivD*)

have *eventually* $(\lambda x. g\ x * (if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x / g\ x) = f\ x) F$

using *asympt-equiv-eventually-zeros*[*OF fg*] by *eventually-elim simp*

with $*$ show *?thesis* by (rule *Lim-transform-eventually*)

qed

from *this*[*of f g*] *this*[*of g f*] *assms* show *?thesis* by (auto *simp: asympt-equiv-sym*)

qed

lemma *smallo-imp-eventually-sgn*:

fixes $f\ g :: real \Rightarrow real$

assumes $g \in o(f)$

shows *eventually* $(\lambda x. sgn\ (f\ x + g\ x) = sgn\ (f\ x))\ at-top$

proof –

have $0 < (1/2 :: real)$ by *simp*

from *landau-o.smallD*[*OF assms*, *OF this*]

have *eventually* $(\lambda x. |g\ x| \leq 1/2 * |f\ x|)\ at-top$ by *simp*

thus *?thesis*

proof *eventually-elim*

case (*elim x*)

thus *?case*

by (cases $f\ x\ 0 :: real$ rule: *linorder-cases*;

```

      cases  $f\ x + g\ x\ 0$ ::real rule: linorder-cases) simp-all
qed
qed

context
begin

private lemma asymp-equiv-add-rightI:
  assumes  $f \sim[F] g$   $h \in o[F](g)$ 
  shows  $(\lambda x. f\ x + h\ x) \sim[F] g$ 
proof -
  let ?T =  $\lambda f\ g\ x.$  if  $f\ x = 0 \wedge g\ x = 0$  then 1 else  $f\ x / g\ x$ 
  from landau-o.smallD[OF assms(2) zero-less-one]
  have ev: eventually  $(\lambda x. g\ x = 0 \longrightarrow h\ x = 0)$  F by eventually-elim auto
  have  $(\lambda x. f\ x + h\ x) \sim[F] g \longleftrightarrow ((\lambda x. ?T\ f\ g\ x + h\ x / g\ x) \longrightarrow 1)$  F
  unfolding asymp-equiv-def using ev
  by (intro tendsto-cong) (auto elim!: eventually-mono simp: field-split-simps)
  also have  $\dots \longleftrightarrow ((\lambda x. ?T\ f\ g\ x + h\ x / g\ x) \longrightarrow 1 + 0)$  F by simp
  also have  $\dots$  by (intro tendsto-intros asymp-equivD assms smalloD-tendsto)
  finally show  $(\lambda x. f\ x + h\ x) \sim[F] g$  .
qed

lemma asymp-equiv-add-right [asymp-equiv-simps]:
  assumes  $h \in o[F](g)$ 
  shows  $(\lambda x. f\ x + h\ x) \sim[F] g \longleftrightarrow f \sim[F] g$ 
proof
  assume  $(\lambda x. f\ x + h\ x) \sim[F] g$ 
  from asymp-equiv-add-rightI[OF this, of  $\lambda x. -h\ x$ ] assms show  $f \sim[F] g$ 
  by simp
qed (simp-all add: asymp-equiv-add-rightI assms)

end

lemma asymp-equiv-add-left [asymp-equiv-simps]:
  assumes  $h \in o[F](g)$ 
  shows  $(\lambda x. h\ x + f\ x) \sim[F] g \longleftrightarrow f \sim[F] g$ 
  using asymp-equiv-add-right[OF assms] by (simp add: add.commute)

lemma asymp-equiv-add-right' [asymp-equiv-simps]:
  assumes  $h \in o[F](g)$ 
  shows  $g \sim[F] (\lambda x. f\ x + h\ x) \longleftrightarrow g \sim[F] f$ 
  using asymp-equiv-add-right[OF assms] by (simp add: asymp-equiv-sym)

lemma asymp-equiv-add-left' [asymp-equiv-simps]:
  assumes  $h \in o[F](g)$ 
  shows  $g \sim[F] (\lambda x. h\ x + f\ x) \longleftrightarrow g \sim[F] f$ 
  using asymp-equiv-add-left[OF assms] by (simp add: asymp-equiv-sym)

lemma smallo-imp-asymp-equiv:

```

assumes $(\lambda x. f\ x - g\ x) \in o[F](g)$
shows $f \sim[F] g$
proof –
from *assms* **have** $(\lambda x. f\ x - g\ x + g\ x) \sim[F] g$
by (*subst asymp-equiv-add-left*) *simp-all*
thus *?thesis* **by** *simp*
qed

lemma *asymp-equiv-uminus* [*asymp-equiv-intros*]:
 $f \sim[F] g \implies (\lambda x. -f\ x) \sim[F] (\lambda x. -g\ x)$
by (*simp add: asymp-equiv-def cong: if-cong*)

lemma *asymp-equiv-uminus-iff* [*asymp-equiv-simps*]:
 $(\lambda x. -f\ x) \sim[F] g \iff f \sim[F] (\lambda x. -g\ x)$
by (*simp add: asymp-equiv-def cong: if-cong*)

lemma *asymp-equiv-mult* [*asymp-equiv-intros*]:
fixes $f1\ f2\ g1\ g2 :: 'a \Rightarrow 'b :: \text{real-normed-field}$
assumes $f1 \sim[F] g1\ f2 \sim[F] g2$
shows $(\lambda x. f1\ x * f2\ x) \sim[F] (\lambda x. g1\ x * g2\ x)$
proof –
let $?T = \lambda f\ g\ x. \text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x$
let $?S = \lambda x. (\text{if } f1\ x = 0 \wedge g1\ x = 0 \text{ then } 1 - ?T\ f2\ g2\ x$
 $\quad \text{else if } f2\ x = 0 \wedge g2\ x = 0 \text{ then } 1 - ?T\ f1\ g1\ x \text{ else } 0)$
let $?S' = \lambda x. ?T\ (\lambda x. f1\ x * f2\ x) (\lambda x. g1\ x * g2\ x)\ x - ?T\ f1\ g1\ x * ?T\ f2\ g2\ x$
have $A: ((\lambda x. 1 - ?T\ f\ g\ x) \longrightarrow 0)\ F$ **if** $f \sim[F] g$ **for** $f\ g :: 'a \Rightarrow 'b$
by (*rule tendsto-eq-intros refl asymp-equivD[OF that])+ simp-all*

from *assms* **have** $*$: $((\lambda x. ?T\ f1\ g1\ x * ?T\ f2\ g2\ x) \longrightarrow 1 * 1)\ F$
by (*intro tendsto-mult asymp-equivD*)
{
have $(?S \longrightarrow 0)\ F$
by (*intro filterlim-If assms[THEN A, THEN tendsto-mono[rotated]]*)
 $(\text{auto intro: le-infI1 le-infI2})$
moreover **have** *eventually* $(\lambda x. ?S\ x = ?S'\ x)\ F$
using *assms[THEN asymp-equiv-eventually-zeros]* **by** *eventually-elim auto*
ultimately **have** $(?S' \longrightarrow 0)\ F$ **by** (*rule Lim-transform-eventually*)
}
with $*$ **have** $(?T\ (\lambda x. f1\ x * f2\ x) (\lambda x. g1\ x * g2\ x) \longrightarrow 1 * 1)\ F$
by (*rule Lim-transform*)
then **show** *?thesis* **by** (*simp add: asymp-equiv-def*)
qed

lemma *asymp-equiv-power* [*asymp-equiv-intros*]:
 $f \sim[F] g \implies (\lambda x. f\ x \wedge n) \sim[F] (\lambda x. g\ x \wedge n)$
by (*induction n*) (*simp-all add: asymp-equiv-mult*)

lemma *asymp-equiv-inverse* [*asymp-equiv-intros*]:
assumes $f \sim[F] g$

shows $(\lambda x. \text{inverse } (f x)) \sim[F] (\lambda x. \text{inverse } (g x))$
proof –
from *tendsto-inverse*[*OF asymp-equivD*[*OF assms*]]
have $((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } g x / f x) \longrightarrow 1) F$
by (*simp add: if-distrib cong: if-cong*)
also have $(\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } g x / f x) =$
 $(\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } \text{inverse } (f x) / \text{inverse } (g x))$
by (*intro ext*) (*simp add: field-simps*)
finally show *?thesis* **by** (*simp add: asymp-equiv-def*)
qed

lemma *asymp-equiv-inverse-iff* [*asymp-equiv-simps*]:
 $(\lambda x. \text{inverse } (f x)) \sim[F] (\lambda x. \text{inverse } (g x)) \longleftrightarrow f \sim[F] g$
proof
assume $(\lambda x. \text{inverse } (f x)) \sim[F] (\lambda x. \text{inverse } (g x))$
hence $(\lambda x. \text{inverse } (\text{inverse } (f x))) \sim[F] (\lambda x. \text{inverse } (\text{inverse } (g x)))$ (**is** *?P*)
by (*rule asymp-equiv-inverse*)
also have *?P* $\longleftrightarrow f \sim[F] g$ **by** (*intro asymp-equiv-cong*) *simp-all*
finally show $f \sim[F] g$.
qed (*simp-all add: asymp-equiv-inverse*)

lemma *asymp-equiv-divide* [*asymp-equiv-intros*]:
assumes $f1 \sim[F] g1$ $f2 \sim[F] g2$
shows $(\lambda x. f1 x / f2 x) \sim[F] (\lambda x. g1 x / g2 x)$
using *asymp-equiv-mult*[*OF assms(1)* *asymp-equiv-inverse*[*OF assms(2)*]] **by**
(*simp add: field-simps*)

lemma *asymp-equivD-strong*:
assumes $f \sim[F] g$ *eventually* $(\lambda x. f x \neq 0 \vee g x \neq 0) F$
shows $((\lambda x. f x / g x) \longrightarrow 1) F$
proof –
from *assms(1)* **have** $((\lambda x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x) \longrightarrow 1) F$
by (*rule asymp-equivD*)
also have *?this* \longleftrightarrow *?thesis*
by (*intro filterlim-cong eventually-mono*[*OF assms(2)*]) *auto*
finally show *?thesis* .
qed

lemma *asymp-equiv-compose* [*asymp-equiv-intros*]:
assumes $f \sim[G] g$ *filterlim* *h* *G* *F*
shows $f \circ h \sim[F] g \circ h$
proof –
let *?T* = $\lambda f g x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x$
have $f \circ h \sim[F] g \circ h \longleftrightarrow ((?T f g \circ h) \longrightarrow 1) F$
by (*simp add: asymp-equiv-def o-def*)
also have $\dots \longleftrightarrow (?T f g \longrightarrow 1) (\text{filtermap } h F)$
by (*rule tendsto-compose-filtermap*)
also have \dots
by (*rule tendsto-mono*[*of - G*]) (*insert assms, simp-all add: asymp-equiv-def*)

filterlim-def)
finally show *?thesis* .
qed

lemma *asympt-equiv-compose'*:
assumes $f \sim [G] g$ *filterlim* h G F
shows $(\lambda x. f (h x)) \sim [F] (\lambda x. g (h x))$
using *asympt-equiv-compose* [*OF assms*] **by** (*simp add: o-def*)

lemma *asympt-equiv-powr-real* [*asympt-equiv-intros*]:
fixes $f g :: 'a \Rightarrow \text{real}$
assumes $f \sim [F] g$ *eventually* $(\lambda x. f x \geq 0)$ F *eventually* $(\lambda x. g x \geq 0)$ F
shows $(\lambda x. f x \text{ powr } y) \sim [F] (\lambda x. g x \text{ powr } y)$
proof –
let $?T = \lambda f g x. \text{if } f x = 0 \wedge g x = 0 \text{ then } 1 \text{ else } f x / g x$
have $((\lambda x. ?T f g x \text{ powr } y) \longrightarrow 1 \text{ powr } y) F$
by (*intro tendsto-intros asympt-equivD* [*OF assms*(1)]) *simp-all*
hence $((\lambda x. ?T f g x \text{ powr } y) \longrightarrow 1) F$ **by** *simp*
moreover have *eventually* $(\lambda x. ?T f g x \text{ powr } y = ?T (\lambda x. f x \text{ powr } y) (\lambda x. g x \text{ powr } y) x) F$
using *asympt-equiv-eventually-zeros* [*OF assms*(1)] *assms*(2,3)
by *eventually-elim* (*auto simp: powr-divide*)
ultimately show *?thesis* **unfolding** *asympt-equiv-def* **by** (*rule Lim-transform-eventually*)
qed

lemma *asympt-equiv-norm* [*asympt-equiv-intros*]:
fixes $f g :: 'a \Rightarrow 'b :: \text{real-normed-field}$
assumes $f \sim [F] g$
shows $(\lambda x. \text{norm } (f x)) \sim [F] (\lambda x. \text{norm } (g x))$
using *tendsto-norm* [*OF asympt-equivD* [*OF assms*]]
by (*simp add: if-distrib asympt-equiv-def norm-divide cong: if-cong*)

lemma *asympt-equiv-abs-real* [*asympt-equiv-intros*]:
fixes $f g :: 'a \Rightarrow \text{real}$
assumes $f \sim [F] g$
shows $(\lambda x. |f x|) \sim [F] (\lambda x. |g x|)$
using *tendsto-rabs* [*OF asympt-equivD* [*OF assms*]]
by (*simp add: if-distrib asympt-equiv-def cong: if-cong*)

lemma *asympt-equiv-imp-eventually-le*:
assumes $f \sim [F] g$ $c > 1$
shows *eventually* $(\lambda x. \text{norm } (f x) \leq c * \text{norm } (g x)) F$
proof –
from *order-tendstoD*(2) [*OF asympt-equivD* [*OF asympt-equiv-norm* [*OF assms*(1)]]
assms(2)]
asympt-equiv-eventually-zeros [*OF assms*(1)]
show *?thesis* **by** *eventually-elim* (*auto split: if-splits simp: field-simps*)
qed

```

lemma asympt-equiv-imp-eventually-ge:
  assumes  $f \sim[F] g$   $c < 1$ 
  shows eventually  $(\lambda x. \text{norm } (f\ x) \geq c * \text{norm } (g\ x))\ F$ 
proof –
  from order-tendstoD(1)[OF asympt-equivD[OF asympt-equiv-norm[OF assms(1)]]]
assms(2)]
    asympt-equiv-eventually-zeros[OF assms(1)]
  show ?thesis by eventually-elim (auto split: if-splits simp: field-simps)
qed

lemma asympt-equiv-imp-bigo:
  assumes  $f \sim[F] g$ 
  shows  $f \in O[F](g)$ 
proof (rule bigoI)
  have  $(3/2::\text{real}) > 1$  by simp
  from asympt-equiv-imp-eventually-le[OF assms this]
  show eventually  $(\lambda x. \text{norm } (f\ x) \leq 3/2 * \text{norm } (g\ x))\ F$ 
  by eventually-elim simp
qed

lemma asympt-equiv-imp-bigomega:
   $f \sim[F] g \implies f \in \Omega[F](g)$ 
  using asympt-equiv-imp-bigo[of  $g\ F\ f$ ] by (simp add: asympt-equiv-sym bigomega-iff-bigo)

lemma asympt-equiv-imp-bigtheta:
   $f \sim[F] g \implies f \in \Theta[F](g)$ 
  by (intro bigthetaI asympt-equiv-imp-bigo asympt-equiv-imp-bigomega)

lemma asympt-equiv-at-infinity-transfer:
  assumes  $f \sim[F] g$  filterlim  $f$  at-infinity  $F$ 
  shows filterlim  $g$  at-infinity  $F$ 
proof –
  from assms(1) have  $g \in \Theta[F](f)$  by (rule asympt-equiv-imp-bigtheta[OF asympt-equiv-symI])
  also from assms have  $f \in \omega[F](\lambda\cdot. 1)$  by (simp add: smallomega-1-conv-filterlim)
  finally show ?thesis by (simp add: smallomega-1-conv-filterlim)
qed

lemma asympt-equiv-at-top-transfer:
  fixes  $f\ g :: \text{real}$ 
  assumes  $f \sim[F] g$  filterlim  $f$  at-top  $F$ 
  shows filterlim  $g$  at-top  $F$ 
proof (rule filterlim-at-infinity-imp-filterlim-at-top)
  show filterlim  $g$  at-infinity  $F$ 
  by (rule asympt-equiv-at-infinity-transfer[OF assms(1) filterlim-mono[OF assms(2)]]]
    (auto simp: at-top-le-at-infinity)
  from assms(2) have eventually  $(\lambda x. f\ x > 0)\ F$ 
  using filterlim-at-top-dense by blast
  with asympt-equiv-eventually-pos-iff[OF assms(1)] show eventually  $(\lambda x. g\ x > 0)\ F$ 

```

by eventually-elim blast
qed

lemma asymp-equiv-at-bot-transfer:

fixes $f\ g :: - \Rightarrow \text{real}$
assumes $f \sim[F] g$ filterlim f at-bot F
shows filterlim g at-bot F
unfolding filterlim-uminus-at-bot
by (rule asymp-equiv-at-top-transfer[of $\lambda x. -f\ x\ F\ \lambda x. -g\ x$])
(insert assms, auto simp: filterlim-uminus-at-bot asymp-equiv-uminus)

lemma asymp-equivI'-const:

assumes $((\lambda x. f\ x / g\ x) \longrightarrow c)\ F\ c \neq 0$
shows $f \sim[F] (\lambda x. c * g\ x)$
using tendsto-mult[OF assms(1) tendsto-const[of inverse c]] assms(2)
by (intro asymp-equivI') (simp add: field-simps)

lemma asymp-equivI'-inverse-const:

assumes $((\lambda x. f\ x / g\ x) \longrightarrow \text{inverse } c)\ F\ c \neq 0$
shows $(\lambda x. c * f\ x) \sim[F] g$
using tendsto-mult[OF assms(1) tendsto-const[of c]] assms(2)
by (intro asymp-equivI') (simp add: field-simps)

lemma filterlim-at-bot-imp-at-infinity: filterlim f at-bot $F \implies$ filterlim f at-infinity F

for $f :: - \Rightarrow \text{real}$ using at-bot-le-at-infinity filterlim-mono by blast

lemma asymp-equiv-imp-diff-smallo:

assumes $f \sim[F] g$
shows $(\lambda x. f\ x - g\ x) \in o[F](g)$
proof (rule landau-o.smallII)
fix $c :: \text{real}$ assume $c > 0$
hence $c: \min\ c\ 1 > 0$ by simp
let $?h = \lambda x. \text{if } f\ x = 0 \wedge g\ x = 0 \text{ then } 1 \text{ else } f\ x / g\ x$
from assms have $((\lambda x. ?h\ x - 1) \longrightarrow 1 - 1)\ F$
by (intro tendsto-diff asymp-equivD tendsto-const)
from tendstoD[OF this c] show eventually $(\lambda x. \text{norm } (f\ x - g\ x) \leq c * \text{norm } (g\ x))\ F$
proof eventually-elim
case (elim x)
from elim have $\text{norm } (f\ x - g\ x) \leq \text{norm } (f\ x / g\ x - 1) * \text{norm } (g\ x)$
by (subst norm-mult [symmetric]) (auto split: if-splits simp add: algebra-simps)
also have $\text{norm } (f\ x / g\ x - 1) * \text{norm } (g\ x) \leq c * \text{norm } (g\ x)$ using elim
by (auto split: if-splits simp: mult-right-mono)
finally show ?case .
qed
qed

lemma asymp-equiv-altdef:

$f \sim[F] g \longleftrightarrow (\lambda x. f x - g x) \in o[F](g)$
by (rule iffI[OF asymp-equiv-imp-diff-smallo smallo-imp-asymp-equiv])

lemma asymp-equiv-0-left-iff [simp]: $(\lambda-. 0) \sim[F] f \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$
 F
and asymp-equiv-0-right-iff [simp]: $f \sim[F] (\lambda-. 0) \longleftrightarrow \text{eventually } (\lambda x. f x = 0)$
 F
by (simp-all add: asymp-equiv-altdef landau-o.small-refl-iff)

lemma asymp-equiv-sandwich-real:
fixes $f g l u :: 'a \Rightarrow \text{real}$
assumes $l \sim[F] g \ u \sim[F] g \text{ eventually } (\lambda x. f x \in \{l x..u x\}) F$
shows $f \sim[F] g$
unfolding asymp-equiv-altdef
proof (rule landau-o.smallI)
fix $c :: \text{real}$ **assume** $c: c > 0$
have $\text{eventually } (\lambda x. \text{norm } (f x - g x) \leq \max (\text{norm } (l x - g x)) (\text{norm } (u x - g x))) F$
using asms(3) **by** eventually-elim auto
moreover **have** $\text{eventually } (\lambda x. \text{norm } (l x - g x) \leq c * \text{norm } (g x)) F$
 $\text{eventually } (\lambda x. \text{norm } (u x - g x) \leq c * \text{norm } (g x)) F$
using asms(1,2) **by** (auto simp: asymp-equiv-altdef dest: landau-o.smallD[OF - c])
hence $\text{eventually } (\lambda x. \max (\text{norm } (l x - g x)) (\text{norm } (u x - g x)) \leq c * \text{norm } (g x)) F$
by eventually-elim simp
ultimately show $\text{eventually } (\lambda x. \text{norm } (f x - g x) \leq c * \text{norm } (g x)) F$
by eventually-elim (rule order.trans)
qed

lemma asymp-equiv-sandwich-real':
fixes $f g l u :: 'a \Rightarrow \text{real}$
assumes $f \sim[F] l \ f \sim[F] u \text{ eventually } (\lambda x. g x \in \{l x..u x\}) F$
shows $f \sim[F] g$
using asymp-equiv-sandwich-real[of l F f u g] **assms** **by** (simp add: asymp-equiv-sym)

lemma asymp-equiv-sandwich-real'':
fixes $f g l u :: 'a \Rightarrow \text{real}$
assumes $l1 \sim[F] u1 \ u1 \sim[F] l2 \ l2 \sim[F] u2$
 $\text{eventually } (\lambda x. f x \in \{l1 x..u1 x\}) F \text{ eventually } (\lambda x. g x \in \{l2 x..u2 x\}) F$
shows $f \sim[F] g$
by (meson asms asymp-equiv-sandwich-real asymp-equiv-sandwich-real' asymp-equiv-trans)

end

55 Values extended by a bottom element

theory Lattice-Constructions

imports Main

begin

datatype *'a bot* = *Value 'a* | *Bot*

instantiation *bot* :: (*preorder*) *preorder*

begin

definition *less-eq-bot* **where**

$x \leq y \longleftrightarrow (\text{case } x \text{ of } Bot \Rightarrow True \mid \text{Value } x \Rightarrow (\text{case } y \text{ of } Bot \Rightarrow False \mid \text{Value } y \Rightarrow x \leq y))$

definition *less-bot* **where**

$x < y \longleftrightarrow (\text{case } y \text{ of } Bot \Rightarrow False \mid \text{Value } y \Rightarrow (\text{case } x \text{ of } Bot \Rightarrow True \mid \text{Value } x \Rightarrow x < y))$

lemma *less-eq-bot-Bot* [*simp*]: $Bot \leq x$

by (*simp add: less-eq-bot-def*)

lemma *less-eq-bot-Bot-code* [*code*]: $Bot \leq x \longleftrightarrow True$

by *simp*

lemma *less-eq-bot-Bot-is-Bot*: $x \leq Bot \implies x = Bot$

by (*cases x*) (*simp-all add: less-eq-bot-def*)

lemma *less-eq-bot-Value-Bot* [*simp, code*]: $\text{Value } x \leq Bot \longleftrightarrow False$

by (*simp add: less-eq-bot-def*)

lemma *less-eq-bot-Value* [*simp, code*]: $\text{Value } x \leq \text{Value } y \longleftrightarrow x \leq y$

by (*simp add: less-eq-bot-def*)

lemma *less-bot-Bot* [*simp, code*]: $x < Bot \longleftrightarrow False$

by (*simp add: less-bot-def*)

lemma *less-bot-Bot-is-Value*: $Bot < x \implies \exists z. x = \text{Value } z$

by (*cases x*) (*simp-all add: less-bot-def*)

lemma *less-bot-Bot-Value* [*simp*]: $Bot < \text{Value } x$

by (*simp add: less-bot-def*)

lemma *less-bot-Bot-Value-code* [*code*]: $Bot < \text{Value } x \longleftrightarrow True$

by *simp*

lemma *less-bot-Value* [*simp, code*]: $\text{Value } x < \text{Value } y \longleftrightarrow x < y$

by (*simp add: less-bot-def*)

instance

by *standard*

(*auto simp add: less-eq-bot-def less-bot-def less-le-not-le elim: order-trans split: bot.splits*)

end

instance *bot* :: (*order*) *order*
by *standard* (*auto simp add: less-eq-bot-def less-bot-def split: bot.splits*)

instance *bot* :: (*linorder*) *linorder*
by *standard* (*auto simp add: less-eq-bot-def less-bot-def split: bot.splits*)

instantiation *bot* :: (*order*) *bot*
begin
 definition *bot* = *Bot*
 instance ..
end

instantiation *bot* :: (*top*) *top*
begin
 definition *top* = *Value top*
 instance ..
end

instantiation *bot* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-bot*
where
 inf *x y* =
 (*case* *x* of
 Bot \Rightarrow *Bot*
 | *Value v* \Rightarrow
 (*case* *y* of
 Bot \Rightarrow *Bot*
 | *Value v'* \Rightarrow *Value (inf v v')*))

instance
by *standard* (*auto simp add: inf-bot-def less-eq-bot-def split: bot.splits*)

end

instantiation *bot* :: (*semilattice-sup*) *semilattice-sup*
begin

definition *sup-bot*
where
 sup *x y* =
 (*case* *x* of
 Bot \Rightarrow *y*
 | *Value v* \Rightarrow
 (*case* *y* of

$$\begin{array}{l} \text{Bot} \Rightarrow x \\ | \text{Value } v' \Rightarrow \text{Value } (\text{sup } v \ v') \end{array}$$
instance**by** *standard* (*auto simp add: sup-bot-def less-eq-bot-def split: bot.splits*)**end****instance** *bot* :: (*lattice*) *bounded-lattice-bot***by** *intro-classes* (*simp add: bot-bot-def*)

55.1 Values extended by a top element

datatype *'a top* = *Value 'a* | *Top***instantiation** *top* :: (*preorder*) *preorder***begin****definition** *less-eq-top* **where**

$$x \leq y \longleftrightarrow (\text{case } y \text{ of } \text{Top} \Rightarrow \text{True} \mid \text{Value } y \Rightarrow (\text{case } x \text{ of } \text{Top} \Rightarrow \text{False} \mid \text{Value } x \Rightarrow x \leq y))$$
definition *less-top* **where**

$$x < y \longleftrightarrow (\text{case } x \text{ of } \text{Top} \Rightarrow \text{False} \mid \text{Value } x \Rightarrow (\text{case } y \text{ of } \text{Top} \Rightarrow \text{True} \mid \text{Value } y \Rightarrow x < y))$$
lemma *less-eq-top-Top* [*simp*]: $x \leq \text{Top}$ **by** (*simp add: less-eq-top-def*)**lemma** *less-eq-top-Top-code* [*code*]: $x \leq \text{Top} \longleftrightarrow \text{True}$ **by** *simp***lemma** *less-eq-top-is-Top*: $\text{Top} \leq x \implies x = \text{Top}$ **by** (*cases x*) (*simp-all add: less-eq-top-def*)**lemma** *less-eq-top-Top-Value* [*simp*, *code*]: $\text{Top} \leq \text{Value } x \longleftrightarrow \text{False}$ **by** (*simp add: less-eq-top-def*)**lemma** *less-eq-top-Value-Value* [*simp*, *code*]: $\text{Value } x \leq \text{Value } y \longleftrightarrow x \leq y$ **by** (*simp add: less-eq-top-def*)**lemma** *less-top-Top* [*simp*, *code*]: $\text{Top} < x \longleftrightarrow \text{False}$ **by** (*simp add: less-top-def*)**lemma** *less-top-Top-is-Value*: $x < \text{Top} \implies \exists z. x = \text{Value } z$ **by** (*cases x*) (*simp-all add: less-top-def*)**lemma** *less-top-Value-Top* [*simp*]: $\text{Value } x < \text{Top}$ **by** (*simp add: less-top-def*)

lemma *less-top-Value-Top-code* [*code*]: *Value x < Top* \longleftrightarrow *True*
by *simp*

lemma *less-top-Value* [*simp, code*]: *Value x < Value y* \longleftrightarrow *x < y*
by (*simp add: less-top-def*)

instance
by *standard*
 (*auto simp add: less-eq-top-def less-top-def less-le-not-le elim: order-trans split: top.splits*)

end

instance *top* :: (*order*) *order*
by *standard* (*auto simp add: less-eq-top-def less-top-def split: top.splits*)

instance *top* :: (*linorder*) *linorder*
by *standard* (*auto simp add: less-eq-top-def less-top-def split: top.splits*)

instantiation *top* :: (*order*) *top*
begin
definition *top* = *Top*
instance ..
end

instantiation *top* :: (*bot*) *bot*
begin
definition *bot* = *Value bot*
instance ..
end

instantiation *top* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-top*
where
inf x y =
 (*case x of*
 Top \Rightarrow *y*
 | *Value v* \Rightarrow
 (*case y of*
 Top \Rightarrow *x*
 | *Value v'* \Rightarrow *Value (inf v v')*))

instance
by *standard* (*auto simp add: inf-top-def less-eq-top-def split: top.splits*)
end

instantiation *top* :: (*semilattice-sup*) *semilattice-sup*
begin

definition *sup-top*
where

$$\begin{aligned} \text{sup } x \ y = & \\ & (\text{case } x \text{ of} \\ & \quad \text{Top} \Rightarrow \text{Top} \\ & | \text{Value } v \Rightarrow \\ & \quad (\text{case } y \text{ of} \\ & \quad \quad \text{Top} \Rightarrow \text{Top} \\ & \quad | \text{Value } v' \Rightarrow \text{Value } (\text{sup } v \ v')) \end{aligned}$$

instance

by *standard* (*auto simp add: sup-top-def less-eq-top-def split: top.splits*)

end

instance *top* :: (*lattice*) *bounded-lattice-top*

by *standard* (*simp add: top-top-def*)

55.2 Values extended by a top and a bottom element

datatype *'a flat-complete-lattice* = *Value 'a | Bot | Top*

instantiation *flat-complete-lattice* :: (*type*) *order*
begin

definition *less-eq-flat-complete-lattice*
where

$$\begin{aligned} x \leq y \equiv & \\ & (\text{case } x \text{ of} \\ & \quad \text{Bot} \Rightarrow \text{True} \\ & | \text{Value } v1 \Rightarrow \\ & \quad (\text{case } y \text{ of} \\ & \quad \quad \text{Bot} \Rightarrow \text{False} \\ & \quad | \text{Value } v2 \Rightarrow v1 = v2 \\ & \quad | \text{Top} \Rightarrow \text{True}) \\ & | \text{Top} \Rightarrow y = \text{Top}) \end{aligned}$$

definition *less-flat-complete-lattice*
where

$$\begin{aligned} x < y = & \\ & (\text{case } x \text{ of} \\ & \quad \text{Bot} \Rightarrow y \neq \text{Bot} \\ & | \text{Value } v1 \Rightarrow y = \text{Top} \\ & | \text{Top} \Rightarrow \text{False}) \end{aligned}$$

```

lemma [simp]:  $Bot \leq y$ 
  unfolding less-eq-flat-complete-lattice-def by auto

lemma [simp]:  $y \leq Top$ 
  unfolding less-eq-flat-complete-lattice-def by (auto split: flat-complete-lattice.splits)

lemma greater-than-two-values:
  assumes  $a \neq b$   $Value\ a \leq z$   $Value\ b \leq z$ 
  shows  $z = Top$ 
  using assms
  by (cases z) (auto simp add: less-eq-flat-complete-lattice-def)

lemma lesser-than-two-values:
  assumes  $a \neq b$   $z \leq Value\ a$   $z \leq Value\ b$ 
  shows  $z = Bot$ 
  using assms
  by (cases z) (auto simp add: less-eq-flat-complete-lattice-def)

instance
  by standard
  (auto simp add: less-eq-flat-complete-lattice-def less-flat-complete-lattice-def
    split: flat-complete-lattice.splits)

end

instantiation flat-complete-lattice :: (type) bot
begin
  definition bot = Bot
  instance ..
end

instantiation flat-complete-lattice :: (type) top
begin
  definition top = Top
  instance ..
end

instantiation flat-complete-lattice :: (type) lattice
begin

definition inf-flat-complete-lattice
where
  inf x y =
    (case x of
      Bot  $\Rightarrow$  Bot
    | Value v1  $\Rightarrow$ 
      (case y of
        Bot  $\Rightarrow$  Bot
      | Value v2  $\Rightarrow$  if v1 = v2 then x else Bot
    )

```

$$\begin{array}{l} | \text{Top} \Rightarrow x) \\ | \text{Top} \Rightarrow y) \end{array}$$
definition *sup-flat-complete-lattice***where**

$$\begin{array}{l} \text{sup } x \ y = \\ \quad (\text{case } x \text{ of} \\ \quad \quad \text{Bot} \Rightarrow y \\ | \text{Value } v1 \Rightarrow \\ \quad \quad (\text{case } y \text{ of} \\ \quad \quad \quad \text{Bot} \Rightarrow x \\ \quad \quad | \text{Value } v2 \Rightarrow \text{if } v1 = v2 \text{ then } x \text{ else Top} \\ \quad \quad | \text{Top} \Rightarrow \text{Top}) \\ | \text{Top} \Rightarrow \text{Top}) \end{array}$$
instance**by** *standard*

$$\begin{array}{l} (\text{auto simp add: inf-flat-complete-lattice-def sup-flat-complete-lattice-def} \\ \quad \text{less-eq-flat-complete-lattice-def split: flat-complete-lattice.splits}) \end{array}$$
end**instantiation** *flat-complete-lattice* :: (type) *complete-lattice***begin****definition** *Sup-flat-complete-lattice***where**

$$\begin{array}{l} \text{Sup } A = \\ \quad (\text{if } A = \{\} \vee A = \{\text{Bot}\} \text{ then Bot} \\ \quad \text{else if } \exists v. A - \{\text{Bot}\} = \{\text{Value } v\} \text{ then Value (THE } v. A - \{\text{Bot}\} = \{\text{Value} \\ v\}) \\ \quad \text{else Top}) \end{array}$$
definition *Inf-flat-complete-lattice***where**

$$\begin{array}{l} \text{Inf } A = \\ \quad (\text{if } A = \{\} \vee A = \{\text{Top}\} \text{ then Top} \\ \quad \text{else if } \exists v. A - \{\text{Top}\} = \{\text{Value } v\} \text{ then Value (THE } v. A - \{\text{Top}\} = \{\text{Value} \\ v\}) \\ \quad \text{else Bot}) \end{array}$$
instance**proof****fix** $x :: 'a$ *flat-complete-lattice***fix** A **assume** $x \in A$

{

fix v **assume** $A - \{\text{Top}\} = \{\text{Value } v\}$

```

    then have (THE v. A - {Top} = {Value v}) = v
      by (auto intro!: the1-equality)
    moreover
    from ⟨x ∈ A⟩ ⟨A - {Top} = {Value v}⟩ have x = Top ∨ x = Value v
      by auto
    ultimately have Value (THE v. A - {Top} = {Value v}) ≤ x
      by auto
  }
  with ⟨x ∈ A⟩ show Inf A ≤ x
    unfolding Inf-flat-complete-lattice-def
    by fastforce
next
fix z :: 'a flat-complete-lattice
fix A
show z ≤ Inf A if z: ⋀x. x ∈ A ⟹ z ≤ x
proof -
  consider A = {} ∨ A = {Top}
  | A ≠ {} A ≠ {Top} ∃ v. A - {Top} = {Value v}
  | A ≠ {} A ≠ {Top} ¬ (∃ v. A - {Top} = {Value v})
  by blast
  then show ?thesis
proof cases
  case 1
  then have Inf A = Top
    unfolding Inf-flat-complete-lattice-def by auto
  then show ?thesis by simp
next
  case 2
  then obtain v where v1: A - {Top} = {Value v}
    by auto
  then have v2: (THE v. A - {Top} = {Value v}) = v
    by (auto intro!: the1-equality)
  from 2 v2 have Inf: Inf A = Value v
    unfolding Inf-flat-complete-lattice-def by simp
  from v1 have Value v ∈ A by blast
  then have z ≤ Value v by (rule z)
  with Inf show ?thesis by simp
next
  case 3
  then have Inf: Inf A = Bot
    unfolding Inf-flat-complete-lattice-def by auto
  have z ≤ Bot
  proof (cases A - {Top} = {Bot})
    case True
    then have Bot ∈ A by blast
    then show ?thesis by (rule z)
  next
    case False
    from 3 obtain a1 where a1: a1 ∈ A - {Top}

```

```

    by auto
  from  $\exists \text{ False } a1$  obtain  $a2$  where  $a2 \in A - \{\text{Top}\} \wedge a1 \neq a2$ 
    by (cases  $a1$ ) auto
  with  $a1$   $z[\text{of } a1]$   $z[\text{of } a2]$  show ?thesis
    apply (cases  $a1$ )
    apply auto
    apply (cases  $a2$ )
    apply auto
    apply (auto dest!: lesser-than-two-values)
    done
  qed
  with  $\text{Inf}$  show ?thesis by simp
  qed
  qed
next
  fix  $x :: 'a \text{ flat-complete-lattice}$ 
  fix  $A$ 
  assume  $x \in A$ 
  {
    fix  $v$ 
    assume  $A - \{\text{Bot}\} = \{\text{Value } v\}$ 
    then have  $(\text{THE } v. A - \{\text{Bot}\} = \{\text{Value } v\}) = v$ 
      by (auto intro!: the1-equality)
    moreover
    from  $\langle x \in A \rangle \langle A - \{\text{Bot}\} = \{\text{Value } v\} \rangle$  have  $x = \text{Bot} \vee x = \text{Value } v$ 
      by auto
    ultimately have  $x \leq \text{Value } (\text{THE } v. A - \{\text{Bot}\} = \{\text{Value } v\})$ 
      by auto
  }
  with  $\langle x \in A \rangle$  show  $x \leq \text{Sup } A$ 
    unfolding  $\text{Sup-flat-complete-lattice-def}$ 
    by fastforce
next
  fix  $z :: 'a \text{ flat-complete-lattice}$ 
  fix  $A$ 
  show  $\text{Sup } A \leq z$  if  $z: \bigwedge x. x \in A \implies x \leq z$ 
  proof -
    consider  $A = \{\}$   $\vee$   $A = \{\text{Bot}\}$ 
    |  $A \neq \{\}$   $A \neq \{\text{Bot}\} \exists v. A - \{\text{Bot}\} = \{\text{Value } v\}$ 
    |  $A \neq \{\}$   $A \neq \{\text{Bot}\} \neg (\exists v. A - \{\text{Bot}\} = \{\text{Value } v\})$ 
    by blast
    then show ?thesis
  proof cases
    case 1
    then have  $\text{Sup } A = \text{Bot}$ 
      unfolding  $\text{Sup-flat-complete-lattice-def}$  by auto
    then show ?thesis by simp
  next
    case 2

```

```

    then obtain v where v1: A - {Bot} = {Value v}
      by auto
    then have v2: (THE v. A - {Bot} = {Value v}) = v
      by (auto intro!: the1-equality)
    from 2 v2 have Sup: Sup A = Value v
      unfolding Sup-flat-complete-lattice-def by simp
    from v1 have Value v ∈ A by blast
    then have Value v ≤ z by (rule z)
    with Sup show ?thesis by simp
  next
  case 3
  then have Sup: Sup A = Top
    unfolding Sup-flat-complete-lattice-def by auto
  have Top ≤ z
  proof (cases A - {Bot} = {Top})
    case True
    then have Top ∈ A by blast
    then show ?thesis by (rule z)
  next
  case False
  from 3 obtain a1 where a1: a1 ∈ A - {Bot}
    by auto
  from 3 False a1 obtain a2 where a2 ∈ A - {Bot} ∧ a1 ≠ a2
    by (cases a1) auto
  with a1 z[of a1] z[of a2] show ?thesis
    apply (cases a1)
    apply auto
    apply (cases a2)
    apply (auto dest!: greater-than-two-values)
    done
  qed
  with Sup show ?thesis by simp
qed
qed
next
show Inf {} = (top :: 'a flat-complete-lattice)
  by (simp add: Inf-flat-complete-lattice-def top-flat-complete-lattice-def)
show Sup {} = (bot :: 'a flat-complete-lattice)
  by (simp add: Sup-flat-complete-lattice-def bot-flat-complete-lattice-def)
qed
end
end

```

56 Infinite Streams

```

theory Stream
  imports Nat-Bijection

```

begin

codatatype (*sset*: 'a) *stream* =
 SCons (*shd*: 'a) (*stl*: 'a *stream*) (**infixr** <##> 65)
for
 map: *smap*
 rel: *stream-all2*

context
begin

— for code generation only

qualified definition *smember* :: 'a \Rightarrow 'a *stream* \Rightarrow bool **where**
 [*code-abbrev*]: *smember* *x s* $\longleftrightarrow x \in \text{sset } s$

lemma *smember-code*[*code*, *simp*]: *smember* *x* (*y* ## *s*) = (if *x* = *y* then True else
 smember *x s*)
unfolding *smember-def* **by** *auto*

end

lemmas *smap-simps*[*simp*] = *stream.map-sel*
lemmas *shd-sset* = *stream.set-sel*(1)
lemmas *stl-sset* = *stream.set-sel*(2)

theorem *sset-induct*[*consumes* 1, *case-names* *shd stl*, *induct set*: *sset*]:
 assumes *y* $\in \text{sset } s$ **and** $\bigwedge s. P (\text{shd } s) s$ **and** $\bigwedge s y. \llbracket y \in \text{sset } (\text{stl } s); P y (\text{stl } s) \rrbracket$
 $\implies P y s$
 shows *P y s*
using *assms* **by** *induct* (*metis stream.sel*(1), *auto*)

lemma *smap-ctr*: *smap* *f s* = *x* ## *s'* $\longleftrightarrow f (\text{shd } s) = x \wedge \text{smap } f (\text{stl } s) = s'$
by (*cases s*) *simp*

56.1 prepend list to stream

primrec *shift* :: 'a list \Rightarrow 'a *stream* \Rightarrow 'a *stream* (**infixr** <@-> 65) **where**
 shift [] *s* = *s*
 | *shift* (*x* # *xs*) *s* = *x* ## *shift xs s*

lemma *smap-shift*[*simp*]: *smap* *f* (*xs* @- *s*) = *map* *f* *xs* @- *smap* *f s*
by (*induct xs*) *auto*

lemma *shift-append*[*simp*]: (*xs* @ *ys*) @- *s* = *xs* @- *ys* @- *s*
by (*induct xs*) *auto*

lemma *shift-simps*[*simp*]:
 shd (*xs* @- *s*) = (if *xs* = [] then *shd s* else *hd xs*)
 stl (*xs* @- *s*) = (if *xs* = [] then *stl s* else *tl xs* @- *s*)

by (induct xs) auto

lemma sset-shift[simp]: $sset\ (xs\ @- \ s) = set\ xs \cup sset\ s$
 by (induct xs) auto

lemma shift-left-inj[simp]: $xs\ @- \ s1 = xs\ @- \ s2 \longleftrightarrow s1 = s2$
 by (induct xs) auto

56.2 set of streams with elements in some fixed set

context

notes [[inductive-internals]]

begin

coinductive-set

streams :: 'a set \Rightarrow 'a stream set

for A :: 'a set

where

Stream[intro!, simp, no-atp]: $\llbracket a \in A; s \in streams\ A \rrbracket \Longrightarrow a\ \#\#\ s \in streams\ A$

end

lemma in-streams: $stl\ s \in streams\ S \Longrightarrow shd\ s \in S \Longrightarrow s \in streams\ S$
 by (cases s) auto

lemma streamsE: $s \in streams\ A \Longrightarrow (shd\ s \in A \Longrightarrow stl\ s \in streams\ A \Longrightarrow P) \Longrightarrow P$
 by (erule streams.cases) simp-all

lemma Stream-image: $x\ \#\#\ y \in ((\#\#\)\ x')\ 'Y \longleftrightarrow x = x' \wedge y \in Y$
 by auto

lemma shift-streams: $\llbracket w \in lists\ A; s \in streams\ A \rrbracket \Longrightarrow w\ @- \ s \in streams\ A$
 by (induct w) auto

lemma streams-Stream: $x\ \#\#\ s \in streams\ A \longleftrightarrow x \in A \wedge s \in streams\ A$
 by (auto elim: streams.cases)

lemma streams-stl: $s \in streams\ A \Longrightarrow stl\ s \in streams\ A$
 by (cases s) (auto simp: streams-Stream)

lemma streams-shd: $s \in streams\ A \Longrightarrow shd\ s \in A$
 by (cases s) (auto simp: streams-Stream)

lemma sset-streams:

assumes $sset\ s \subseteq A$

shows $s \in streams\ A$

using assms **proof** (coinduction arbitrary: s)

case streams then show ?case by (cases s) simp

qed

lemma *streams-sset*:

assumes $s \in \text{streams } A$

shows $\text{sset } s \subseteq A$

proof

fix x **assume** $x \in \text{sset } s$ **from** *this* $\langle s \in \text{streams } A \rangle$ **show** $x \in A$

by (*induct s*) (*auto intro: streams-shd streams-stl*)

qed

lemma *streams-iff-sset*: $s \in \text{streams } A \iff \text{sset } s \subseteq A$

by (*metis sset-streams streams-sset*)

lemma *streams-mono*: $s \in \text{streams } A \implies A \subseteq B \implies s \in \text{streams } B$

unfolding *streams-iff-sset* **by** *auto*

lemma *streams-mono2*: $S \subseteq T \implies \text{streams } S \subseteq \text{streams } T$

by (*auto intro: streams-mono*)

lemma *smap-streams*: $s \in \text{streams } A \implies (\bigwedge x. x \in A \implies f x \in B) \implies \text{smap } f s \in \text{streams } B$

unfolding *streams-iff-sset stream.set-map* **by** *auto*

lemma *streams-empty*: $\text{streams } \{\} = \{\}$

by (*auto elim: streams.cases*)

lemma *streams-UNIV[simp]*: $\text{streams } UNIV = UNIV$

by (*auto simp: streams-iff-sset*)

56.3 nth, take, drop for streams

primrec *snth* :: $'a \text{ stream} \Rightarrow \text{nat} \Rightarrow 'a$ (**infixl** $\langle !! \rangle$ 100) **where**

$s !! 0 = \text{shd } s$

$| s !! \text{Suc } n = \text{stl } s !! n$

lemma *snth-Stream*: $(x \## s) !! \text{Suc } i = s !! i$

by *simp*

lemma *snth-smap[simp]*: $\text{smap } f s !! n = f (s !! n)$

by (*induct n arbitrary: s*) *auto*

lemma *shift-snth-less[simp]*: $p < \text{length } xs \implies (xs @- s) !! p = xs ! p$

by (*induct p arbitrary: xs*) (*auto simp: hd-conv-nth nth-tl*)

lemma *shift-snth-ge[simp]*: $p \geq \text{length } xs \implies (xs @- s) !! p = s !! (p - \text{length } xs)$

by (*induct p arbitrary: xs*) (*auto simp: Suc-diff-eq-diff-pred*)

lemma *shift-snth*: $(xs @- s) !! n = (\text{if } n < \text{length } xs \text{ then } xs ! n \text{ else } s !! (n - \text{length } xs))$

by *auto*

lemma *snth-sset*[*simp*]: $s !! n \in \text{sset } s$
by (*induct n arbitrary: s*) (*auto intro: shd-sset stl-sset*)

lemma *sset-range*: $\text{sset } s = \text{range } (\text{snth } s)$
proof (*intro equalityI subsetI*)
fix x **assume** $x \in \text{sset } s$
thus $x \in \text{range } (\text{snth } s)$
proof (*induct s*)
case (*stl s x*)
then obtain n **where** $x = \text{stl } s !! n$ **by** *auto*
thus $?case$ **by** (*auto intro: range-eqI[of - - Suc n]*)
qed (*auto intro: range-eqI[of - - 0]*)
qed *auto*

lemma *streams-iff-snth*: $s \in \text{streams } X \longleftrightarrow (\forall n. s !! n \in X)$
by (*force simp: streams-iff-sset sset-range*)

lemma *snth-in*: $s \in \text{streams } X \implies s !! n \in X$
by (*simp add: streams-iff-snth*)

primrec *stake* :: $\text{nat} \Rightarrow 'a \text{ stream} \Rightarrow 'a \text{ list}$ **where**
stake 0 s = []
| *stake (Suc n) s* = *shd s* # *stake n (stl s)*

lemma *length-stake*[*simp*]: $\text{length } (\text{stake } n \ s) = n$
by (*induct n arbitrary: s*) *auto*

lemma *stake-smap*[*simp*]: $\text{stake } n \ (\text{smap } f \ s) = \text{map } f \ (\text{stake } n \ s)$
by (*induct n arbitrary: s*) *auto*

lemma *take-stake*: $\text{take } n \ (\text{stake } m \ s) = \text{stake } (\min n \ m) \ s$
proof (*induct m arbitrary: s n*)
case (*Suc m*) **thus** $?case$ **by** (*cases n*) *auto*
qed *simp*

primrec *sdrop* :: $\text{nat} \Rightarrow 'a \text{ stream} \Rightarrow 'a \text{ stream}$ **where**
sdrop 0 s = *s*
| *sdrop (Suc n) s* = *sdrop n (stl s)*

lemma *sdrop-simps*[*simp*]:
 $\text{shd } (\text{sdrop } n \ s) = s !! n$ $\text{stl } (\text{sdrop } n \ s) = \text{sdrop } (\text{Suc } n) \ s$
by (*induct n arbitrary: s*) *auto*

lemma *sdrop-smap*[*simp*]: $\text{sdrop } n \ (\text{smap } f \ s) = \text{smap } f \ (\text{sdrop } n \ s)$
by (*induct n arbitrary: s*) *auto*

lemma *sdrop-stl*: $\text{sdrop } n \ (\text{stl } s) = \text{stl } (\text{sdrop } n \ s)$

by (*induct* n) *auto*

lemma *drop-stake*: $\text{drop } n (\text{stake } m \ s) = \text{stake } (m - n) (\text{sdrop } n \ s)$

proof (*induct* m *arbitrary*: $s \ n$)

case (*Suc* m) **thus** ?*case* **by** (*cases* n) *auto*

qed *simp*

lemma *stake-sdrop*: $\text{stake } n \ s \ @- \ \text{sdrop } n \ s = s$

by (*induct* n *arbitrary*: s) *auto*

lemma *id-stake-snth-sdrop*:

$s = \text{stake } i \ s \ @- \ s \ !! \ i \ ## \ \text{sdrop } (\text{Suc } i) \ s$

by (*subst* *stake-sdrop*[*symmetric*, *of* - i]) (*metis* *sdrop-simps* *stream.collapse*)

lemma *smap-alt*: $\text{smap } f \ s = s' \longleftrightarrow (\forall n. f \ (s \ !! \ n) = s' \ !! \ n) \ (\text{is } ?L = ?R)$

proof

assume ? R

then have $\bigwedge n. \text{smap } f \ (\text{sdrop } n \ s) = \text{sdrop } n \ s'$

by *coinduction* (*auto* *intro*: *exI*[*of* - 0] *simp* *del*: *sdrop.simps*(2))

then show ? L **using** *sdrop.simps*(1) **by** *metis*

qed *auto*

lemma *stake-invert-Nil*[*iff*]: $\text{stake } n \ s = [] \longleftrightarrow n = 0$

by (*induct* n) *auto*

lemma *sdrop-shift*: $\text{sdrop } i \ (w \ @- \ s) = \text{drop } i \ w \ @- \ \text{sdrop } (i - \text{length } w) \ s$

by (*induct* i *arbitrary*: $w \ s$) (*auto* *simp*: *drop-tl* *drop-Suc* *neq-Nil-conv*)

lemma *stake-shift*: $\text{stake } i \ (w \ @- \ s) = \text{take } i \ w \ @ \ \text{stake } (i - \text{length } w) \ s$

by (*induct* i *arbitrary*: $w \ s$) (*auto* *simp*: *neq-Nil-conv*)

lemma *stake-add*[*simp*]: $\text{stake } m \ s \ @ \ \text{stake } n \ (\text{sdrop } m \ s) = \text{stake } (m + n) \ s$

by (*induct* m *arbitrary*: s) *auto*

lemma *sdrop-add*[*simp*]: $\text{sdrop } n \ (\text{sdrop } m \ s) = \text{sdrop } (m + n) \ s$

by (*induct* m *arbitrary*: s) *auto*

lemma *sdrop-snth*: $\text{sdrop } n \ s \ !! \ m = s \ !! \ (n + m)$

by (*induct* n *arbitrary*: $m \ s$) *auto*

partial-function (*tailrec*) *sdrop-while* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \ \text{stream} \Rightarrow 'a \ \text{stream}$

where

$\text{sdrop-while } P \ s = (\text{if } P \ (\text{shd } s) \text{ then } \text{sdrop-while } P \ (\text{stl } s) \text{ else } s)$

lemma *sdrop-while-SCons*[*code*]:

$\text{sdrop-while } P \ (a \ ## \ s) = (\text{if } P \ a \text{ then } \text{sdrop-while } P \ s \text{ else } a \ ## \ s)$

by (*subst* *sdrop-while.simps*) *simp*

lemma *sdrop-while-sdrop-LEAST*:

assumes $\exists n. P (s !! n)$
shows $sdrop\text{-}while (Not \circ P) s = sdrop (LEAST n. P (s !! n)) s$
proof –
from *assms* **obtain** m **where** $P (s !! m) \wedge n. P (s !! n) \implies m \leq n$
and $*$: $(LEAST n. P (s !! n)) = m$ **by** *atomize-elim (auto intro: LeastI Least-le)*
thus *?thesis* **unfolding** $*$
proof (*induct m arbitrary: s*)
case (*Suc m*)
hence $sdrop\text{-}while (Not \circ P) (stl s) = sdrop m (stl s)$
by (*metis (full-types) not-less-eq-eq snth.simps(2)*)
moreover from *Suc(3)* **have** $\neg (P (s !! 0))$ **by** *blast*
ultimately show *?case* **by** (*subst sdrop-while.simps*) *simp*
qed (*metis comp-apply sdrop.simps(1) sdrop-while.simps snth.simps(1)*)
qed

primcorec *sfilter* **where**

$shd (sfilter P s) = shd (sdrop\text{-}while (Not \circ P) s)$
 $| stl (sfilter P s) = sfilter P (stl (sdrop\text{-}while (Not \circ P) s))$

lemma *sfilter-Stream*: $sfilter P (x \#\# s) = (if P x then x \#\# sfilter P s else sfilter P s)$

proof (*cases P x*)

case *True* **thus** *?thesis* **by** (*subst sfilter.ctr*) (*simp add: sdrop-while-SCons*)

next

case *False* **thus** *?thesis* **by** (*subst (1 2) sfilter.ctr*) (*simp add: sdrop-while-SCons*)

qed

56.4 unary predicates lifted to streams

definition *stream-all* $P s = (\forall p. P (s !! p))$

lemma *stream-all-iff[iff]*: $stream\text{-}all P s \longleftrightarrow Ball (sset s) P$

unfolding *stream-all-def sset-range* **by** *auto*

lemma *stream-all-shift[simp]*: $stream\text{-}all P (xs @- s) = (list\text{-}all P xs \wedge stream\text{-}all P s)$

unfolding *stream-all-iff list-all-iff* **by** *auto*

lemma *stream-all-Stream*: $stream\text{-}all P (x \#\# X) \longleftrightarrow P x \wedge stream\text{-}all P X$

by *simp*

56.5 recurring stream out of a list

primcorec *cycle* :: 'a list \Rightarrow 'a stream **where**

$shd (cycle xs) = hd xs$

$| stl (cycle xs) = cycle (tl xs @ [hd xs])$

lemma *cycle-decomp*: $u \neq [] \implies cycle u = u @- cycle u$

proof (*coinduction arbitrary: u*)

case *Eq-stream* **then show** *?case* **using** *stream.collapse[of cycle u]*

by (*auto intro!*: *exI*[*of* - *tl* *u* @ [*hd* *u*]])
qed

lemma *cycle-Cons*[*code*]: *cycle* (*x* # *xs*) = *x* ## *cycle* (*xs* @ [*x*])
by (*subst* *cycle.ctr*) *simp*

lemma *cycle-rotated*: $\llbracket v \neq []; \text{cycle } u = v @- s \rrbracket \implies \text{cycle } (tl \ u @ [hd \ u]) = tl \ v @- s$
by (*auto dest*: *arg-cong*[*of* - - *stl*])

lemma *stake-append*: *stake* *n* (*u* @- *s*) = *take* (*min* (*length* *u*) *n*) *u* @ *stake* (*n* - *length* *u*) *s*
proof (*induct* *n* *arbitrary*: *u*)
case (*Suc* *n*) **thus** ?*case* **by** (*cases* *u*) *auto*
qed *auto*

lemma *stake-cycle-le*[*simp*]:
assumes $u \neq []$ *n* < *length* *u*
shows *stake* *n* (*cycle* *u*) = *take* *n* *u*
using *min-absorb2*[*OF less-imp-le-nat*[*OF assms*(2)]]
by (*subst* *cycle-decomp*[*OF assms*(1)], *subst* *stake-append*) *auto*

lemma *stake-cycle-eq*[*simp*]: $u \neq [] \implies \text{stake } (\text{length } u) (\text{cycle } u) = u$
by (*subst* *cycle-decomp*) (*auto simp*: *stake-shift*)

lemma *sdrop-cycle-eq*[*simp*]: $u \neq [] \implies \text{sdrop } (\text{length } u) (\text{cycle } u) = \text{cycle } u$
by (*subst* *cycle-decomp*) (*auto simp*: *sdrop-shift*)

lemma *stake-cycle-eq-mod-0*[*simp*]: $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies$
 $\text{stake } n (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u) \ u)$
by (*induct* *n* *div* *length* *u* *arbitrary*: *n* *u*)
(auto simp: *stake-add* [*symmetric*] *mod-eq-0-iff-dvd* *elim!*: *dvdE*)

lemma *sdrop-cycle-eq-mod-0*[*simp*]: $\llbracket u \neq []; n \bmod \text{length } u = 0 \rrbracket \implies$
 $\text{sdrop } n (\text{cycle } u) = \text{cycle } u$
by (*induct* *n* *div* *length* *u* *arbitrary*: *n* *u*)
(auto simp: *sdrop-add* [*symmetric*] *mod-eq-0-iff-dvd* *elim!*: *dvdE*)

lemma *stake-cycle*: $u \neq [] \implies$
 $\text{stake } n (\text{cycle } u) = \text{concat } (\text{replicate } (n \text{ div } \text{length } u) \ u) @ \text{take } (n \bmod \text{length } u) \ u$
by (*subst* *div-mult-mod-eq*[*of* *n* *length* *u*, *symmetric*], *unfold* *stake-add*[*symmetric*])
auto

lemma *sdrop-cycle*: $u \neq [] \implies \text{sdrop } n (\text{cycle } u) = \text{cycle } (\text{rotate } (n \bmod \text{length } u) \ u)$
by (*induct* *n* *arbitrary*: *u*) (*auto simp*: *rotate1-rotate-swap* *rotate1-hd-tl* *rotate-conv-mod*[*symmetric*])

lemma *sset-cycle*[*simp*]:

```

assumes  $xs \neq []$ 
shows  $sset\ (cycle\ xs) = set\ xs$ 
proof (intro set-eqI iffI)
  fix  $x$ 
  assume  $x \in sset\ (cycle\ xs)$ 
  then show  $x \in set\ xs$  using assms
  by (induction cycle xs arbitrary: xs rule: sset-induct) (fastforce simp: neq-Nil-conv) +
qed (metis assms UnI1 cycle-decomp sset-shift)

```

56.6 iterated application of a function

```

primcorec siterate where
   $shd\ (siterate\ f\ x) = x$ 
   $| stl\ (siterate\ f\ x) = siterate\ f\ (f\ x)$ 

```

```

lemma stake-Suc:  $stake\ (Suc\ n)\ s = stake\ n\ s\ @\ [s\ !!\ n]$ 
by (induct n arbitrary: s) auto

```

```

lemma snth-siterate[simp]:  $siterate\ f\ x\ !!\ n = (f^{\sim n})\ x$ 
by (induct n arbitrary: x) (auto simp: funpow-swap1)

```

```

lemma sdrop-siterate[simp]:  $sdrop\ n\ (siterate\ f\ x) = siterate\ f\ ((f^{\sim n})\ x)$ 
by (induct n arbitrary: x) (auto simp: funpow-swap1)

```

```

lemma stake-siterate[simp]:  $stake\ n\ (siterate\ f\ x) = map\ (\lambda n. (f^{\sim n})\ x)\ [0\ ..<\ n]$ 
by (induct n arbitrary: x) (auto simp del: stake.simps(2) simp: stake-Suc)

```

```

lemma sset-siterate:  $sset\ (siterate\ f\ x) = \{(f^{\sim n})\ x \mid n.\ True\}$ 
by (auto simp: sset-range)

```

```

lemma smap-siterate:  $smap\ f\ (siterate\ f\ x) = siterate\ f\ (f\ x)$ 
by (coinduction arbitrary: x) auto

```

56.7 stream repeating a single element

```

abbreviation sconst  $\equiv siterate\ id$ 

```

```

lemma shift-replicate-sconst[simp]:  $replicate\ n\ x\ @- sconst\ x = sconst\ x$ 
by (subst (3) stake-sdrop[symmetric]) (simp add: map-replicate-trivial)

```

```

lemma sset-sconst[simp]:  $sset\ (sconst\ x) = \{x\}$ 
by (simp add: sset-siterate)

```

```

lemma sconst-alt:  $s = sconst\ x \longleftrightarrow sset\ s = \{x\}$ 

```

```

proof
  assume  $sset\ s = \{x\}$ 
  then show  $s = sconst\ x$ 
  proof (coinduction arbitrary: s)
    case Eq-stream
    then have  $shd\ s = x\ sset\ (stl\ s) \subseteq \{x\}$  by (cases s; auto) +

```

```

    then have sset (stl s) = {x} by (cases stl s) auto
    with ⟨shd s = x⟩ show ?case by auto
qed
qed simp

```

```

lemma sconst-cycle: sconst x = cycle [x]
  by coinduction auto

```

```

lemma smap-sconst: smap f (sconst x) = sconst (f x)
  by coinduction auto

```

```

lemma sconst-streams: x ∈ A ⟹ sconst x ∈ streams A
  by (simp add: streams-iff-sset)

```

```

lemma streams-empty-iff: streams S = {} ⟷ S = {}
proof safe
  fix x assume x ∈ S streams S = {}
  then have sconst x ∈ streams S
    by (intro sconst-streams)
  then show x ∈ {}
    unfolding ⟨streams S = {}⟩ by simp
qed (auto simp: streams-empty)

```

56.8 stream of natural numbers

abbreviation $fromN \equiv siterate\ Suc$

abbreviation $nats \equiv fromN\ 0$

```

lemma sset-fromN[simp]: sset (fromN n) = {n ..}
  by (auto simp add: sset-siterate le-iff-add)

```

```

lemma stream-smap-fromN: s = smap (λj. let i = j - n in s !! i) (fromN n)
  by (coinduction arbitrary: s n)
    (force simp: neq-Nil-conv Let-def Suc-diff-Suc simp flip: snth.simps(2)
      intro: stream.map-cong split: if-splits)

```

```

lemma stream-smap-nats: s = smap (snth s) nats
  using stream-smap-fromN[where n = 0] by simp

```

56.9 flatten a stream of lists

```

primcorec flat where
  shd (flat ws) = hd (shd ws)
| stl (flat ws) = flat (if tl (shd ws) = [] then stl ws else tl (shd ws) ## stl ws)

```

```

lemma flat-Cons[simp, code]: flat ((x # xs) ## ws) = x ## flat (if xs = [] then
ws else xs ## ws)
  by (subst flat.ctr) simp

```


lemma *flat-Stream[simp]*: $xs \neq [] \implies \text{flat } (xs \## ws) = xs @- \text{flat } ws$
by (*induct xs*) *auto*

lemma *flat-unfold*: $\text{shd } ws \neq [] \implies \text{flat } ws = \text{shd } ws @- \text{flat } (\text{stl } ws)$
by (*cases ws*) *auto*

lemma *flat-snth*: $\forall xs \in \text{sset } s. xs \neq [] \implies \text{flat } s !! n = (\text{if } n < \text{length } (\text{shd } s) \text{ then } \text{shd } s ! n \text{ else } \text{flat } (\text{stl } s) !! (n - \text{length } (\text{shd } s)))$
by (*metis flat-unfold not-less shd-sset shift-snth-ge shift-snth-less*)

lemma *sset-flat[simp]*: $\forall xs \in \text{sset } s. xs \neq [] \implies \text{sset } (\text{flat } s) = (\bigcup xs \in \text{sset } s. \text{set } xs)$ (**is** $?P \implies ?L = ?R$)

proof *safe*

fix x **assume** $?P \ x \in ?L$

then obtain m **where** $x = \text{flat } s !! m$ **by** (*metis image-iff sset-range*)

with $\langle ?P \rangle$ **obtain** $n \ m'$ **where** $x = s !! n ! m' \ m' < \text{length } (s !! n)$

proof (*atomize-elim, induct m arbitrary: s rule: less-induct*)

case (*less y*)

thus $?case$

proof (*cases y < length (shd s)*)

case *True* **thus** $?thesis$ **by** (*metis flat-snth less(2,3) snth.simps(1)*)

next

case *False*

hence $x = \text{flat } (\text{stl } s) !! (y - \text{length } (\text{shd } s))$ **by** (*metis less(2,3) flat-snth*)

moreover have $y - \text{length } (\text{shd } s) < y$

proof $-$

from *less(2)* **have** $*$: $\text{length } (\text{shd } s) > 0$ **by** (*cases s*) *simp-all*

with *False* **have** $y > 0$ **by** (*cases y*) *simp-all*

with $*$ **show** $?thesis$ **by** *simp*

qed

moreover have $\forall xs \in \text{sset } (\text{stl } s). xs \neq []$ **using** *less(2)* **by** (*cases s*) *auto*

ultimately have $\exists n \ m'. x = \text{stl } s !! n ! m' \wedge m' < \text{length } (\text{stl } s !! n)$ **by**

(*intro less(1)*) *auto*

thus $?thesis$ **by** (*metis snth.simps(2)*)

qed

qed

thus $x \in ?R$ **by** (*auto simp: sset-range dest!: nth-mem*)

next

fix $x \ xs$

assume $xs \in \text{sset } s \ ?P \ x \in \text{set } xs$

thus $x \in ?L$

by (*induct rule: sset-induct*)

(*metis UnI1 flat-unfold shift.simps(1) sset-shift,*

metis UnI2 flat-unfold shd-sset stl-sset sset-shift)

qed

56.10 merge a stream of streams

definition *smerge* :: $'a \text{ stream } \text{stream} \Rightarrow 'a \text{ stream}$ **where**

$smerge\ ss = flat\ (smap\ (\lambda n. map\ (\lambda s. s !! n)\ (stake\ (Suc\ n)\ ss)\ @\ stake\ n\ (ss !! n))\ nats)$

lemma *stake-nth[simp]*: $m < n \implies stake\ n\ s ! m = s !! m$
by (*induct* n *arbitrary*: $s\ m$) (*auto simp: nth-Cons', metis Suc-pred snth.simps(2)*)

lemma *snth-sset-smerge*: $ss !! n !! m \in sset\ (smmerge\ ss)$

proof (*cases* $n \leq m$)

case *False* **thus** ?thesis **unfolding** *smmerge-def*

by (*subst sset-flat*)

(*auto simp: stream.set-map in-set-conv-nth simp del: stake.simps*

intro!: $exI[of - n, OF\ disjI2]\ exI[of - m, OF\ mp]$)

next

case *True* **thus** ?thesis **unfolding** *smmerge-def*

by (*subst sset-flat*)

(*auto simp: stream.set-map in-set-conv-nth image-iff simp del: stake.simps*

snth.simps

intro!: $exI[of - m, OF\ disjI1]\ bexI[of - ss !! n]\ exI[of - n, OF\ mp]$)

qed

lemma *sset-smerge*: $sset\ (smmerge\ ss) = \bigcup (sset\ ' (sset\ ss))$

proof *safe*

fix x **assume** $x \in sset\ (smmerge\ ss)$

thus $x \in \bigcup (sset\ ' (sset\ ss))$

unfolding *smmerge-def* **by** (*subst (asm) sset-flat*)

(*auto simp: stream.set-map in-set-conv-nth sset-range simp del: stake.simps,*

fast+)

next

fix $s\ x$ **assume** $s \in sset\ ss\ x \in sset\ s$

thus $x \in sset\ (smmerge\ ss)$ **using** *snth-sset-smerge* **by** (*auto simp: sset-range*)

qed

56.11 product of two streams

definition *sproduct* :: $'a\ stream \Rightarrow 'b\ stream \Rightarrow ('a \times 'b)\ stream$ **where**

$sproduct\ s1\ s2 = smmerge\ (smap\ (\lambda x. smap\ (Pair\ x)\ s2)\ s1)$

lemma *sset-sproduct*: $sset\ (sproduct\ s1\ s2) = sset\ s1 \times sset\ s2$

unfolding *sproduct-def sset-smerge* **by** (*auto simp: stream.set-map*)

56.12 interleave two streams

primcorec *sinterleave* **where**

$shd\ (sinterleave\ s1\ s2) = shd\ s1$

| $stl\ (sinterleave\ s1\ s2) = sinterleave\ s2\ (stl\ s1)$

lemma *sinterleave-code*[*code*]:

$sinterleave\ (x\ \#\#\ s1)\ s2 = x\ \#\#\ sinterleave\ s2\ s1$

by (*subst sinterleave.ctr*) *simp*

lemma *sinterleave-snth*[simp]:

even $n \implies \text{sinterleave } s1 \ s2 !! n = s1 !! (n \text{ div } 2)$

odd $n \implies \text{sinterleave } s1 \ s2 !! n = s2 !! (n \text{ div } 2)$

by (*induct* n *arbitrary*: $s1 \ s2$) *simp-all*

lemma *sset-sinterleave*: $\text{sset } (\text{sinterleave } s1 \ s2) = \text{sset } s1 \cup \text{sset } s2$

proof (*intro equalityI subsetI*)

fix x **assume** $x \in \text{sset } (\text{sinterleave } s1 \ s2)$

then obtain n **where** $x = \text{sinterleave } s1 \ s2 !! n$ **unfolding** *sset-range* **by** *blast*

thus $x \in \text{sset } s1 \cup \text{sset } s2$ **by** (*cases even* n) *auto*

next

fix x **assume** $x \in \text{sset } s1 \cup \text{sset } s2$

thus $x \in \text{sset } (\text{sinterleave } s1 \ s2)$

proof

assume $x \in \text{sset } s1$

then obtain n **where** $x = s1 !! n$ **unfolding** *sset-range* **by** *blast*

hence $\text{sinterleave } s1 \ s2 !! (2 * n) = x$ **by** *simp*

thus *?thesis* **unfolding** *sset-range* **by** *blast*

next

assume $x \in \text{sset } s2$

then obtain n **where** $x = s2 !! n$ **unfolding** *sset-range* **by** *blast*

hence $\text{sinterleave } s1 \ s2 !! (2 * n + 1) = x$ **by** *simp*

thus *?thesis* **unfolding** *sset-range* **by** *blast*

qed

qed

56.13 zip

primcorec *szip* **where**

$\text{shd } (\text{szip } s1 \ s2) = (\text{shd } s1, \text{shd } s2)$

| $\text{stl } (\text{szip } s1 \ s2) = \text{szip } (\text{stl } s1) (\text{stl } s2)$

lemma *szip-unfold*[code]: $\text{szip } (a \ \#\# \ s1) (b \ \#\# \ s2) = (a, b) \ \#\# \ (\text{szip } s1 \ s2)$

by (*subst szip.ctr*) *simp*

lemma *snth-szip*[simp]: $\text{szip } s1 \ s2 !! n = (s1 !! n, s2 !! n)$

by (*induct* n *arbitrary*: $s1 \ s2$) *auto*

lemma *stake-szip*[simp]:

$\text{stake } n (\text{szip } s1 \ s2) = \text{zip } (\text{stake } n \ s1) (\text{stake } n \ s2)$

by (*induct* n *arbitrary*: $s1 \ s2$) *auto*

lemma *sdrop-szip*[simp]: $\text{sdrop } n (\text{szip } s1 \ s2) = \text{szip } (\text{sdrop } n \ s1) (\text{sdrop } n \ s2)$

by (*induct* n *arbitrary*: $s1 \ s2$) *auto*

lemma *smap-szip-fst*:

$\text{smap } (\lambda x. f \ (\text{fst } x)) (\text{szip } s1 \ s2) = \text{smap } f \ s1$

by (*coinduction* *arbitrary*: $s1 \ s2$) *auto*

lemma *smap-szip-snd*:
 $\text{smap } (\lambda x. g \text{ (snd } x)) \text{ (szip } s1 \text{ } s2) = \text{smap } g \text{ } s2$
by (*coinduction arbitrary: s1 s2*) *auto*

56.14 zip via function

primcorec *smap2* **where**
 $\text{shd } (\text{smap2 } f \text{ } s1 \text{ } s2) = f \text{ (shd } s1) \text{ (shd } s2)$
 $\mid \text{stl } (\text{smap2 } f \text{ } s1 \text{ } s2) = \text{smap2 } f \text{ (stl } s1) \text{ (stl } s2)$

lemma *smap2-unfold*[*code*]:
 $\text{smap2 } f \text{ (a \#\# } s1) \text{ (b \#\# } s2) = f \text{ a b \#\# (smap2 } f \text{ } s1 \text{ } s2)$
by (*subst smap2.ctr*) *simp*

lemma *smap2-szip*:
 $\text{smap2 } f \text{ } s1 \text{ } s2 = \text{smap } (\text{case-prod } f) \text{ (szip } s1 \text{ } s2)$
by (*coinduction arbitrary: s1 s2*) *auto*

lemma *smap-smap2*[*simp*]:
 $\text{smap } f \text{ (smap2 } g \text{ } s1 \text{ } s2) = \text{smap2 } (\lambda x y. f \text{ (g } x \text{ } y)) \text{ } s1 \text{ } s2$
unfolding *smap2-szip stream.map-comp o-def split-def ..*

lemma *smap2-alt*:
 $(\text{smap2 } f \text{ } s1 \text{ } s2 = s) = (\forall n. f \text{ (s1 !! } n) \text{ (s2 !! } n) = s \text{ !! } n)$
unfolding *smap2-szip smap-alt* **by** *auto*

lemma *snth-smap2*[*simp*]:
 $\text{smap2 } f \text{ } s1 \text{ } s2 \text{ !! } n = f \text{ (s1 !! } n) \text{ (s2 !! } n)$
by (*induct n arbitrary: s1 s2*) *auto*

lemma *stake-smap2*[*simp*]:
 $\text{stake } n \text{ (smap2 } f \text{ } s1 \text{ } s2) = \text{map } (\text{case-prod } f) \text{ (zip (stake } n \text{ } s1) \text{ (stake } n \text{ } s2))}$
by (*induct n arbitrary: s1 s2*) *auto*

lemma *sdrop-smap2*[*simp*]:
 $\text{sdrop } n \text{ (smap2 } f \text{ } s1 \text{ } s2) = \text{smap2 } f \text{ (sdrop } n \text{ } s1) \text{ (sdrop } n \text{ } s2)$
by (*induct n arbitrary: s1 s2*) *auto*

end

57 List prefixes, suffixes, and homeomorphic embedding

theory *Sublist*
imports *Main*
begin

57.1 Prefix order on lists

definition $prefix :: 'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where $prefix\ xs\ ys \longleftrightarrow (\exists\ zs.\ ys = xs @ zs)$

definition $strict-prefix :: 'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where $strict-prefix\ xs\ ys \longleftrightarrow prefix\ xs\ ys \wedge xs \neq ys$

global-interpretation $prefix-order$: ordering $prefix\ strict-prefix$
by *standard* (*auto simp add: prefix-def strict-prefix-def*)

interpretation $prefix-order$: order $prefix\ strict-prefix$
by *standard* (*auto simp: prefix-def strict-prefix-def*)

global-interpretation $prefix-bot$: ordering-top $\langle \lambda xs\ ys.\ prefix\ ys\ xs \rangle \langle \lambda xs\ ys.\ strict-prefix\ ys\ xs \rangle \langle [] \rangle$
by *standard* (*simp add: prefix-def*)

interpretation $prefix-bot$: order-bot *Nil prefix strict-prefix*
by *standard* (*simp add: prefix-def*)

lemma $prefixI$ [*intro?*]: $ys = xs @ zs \Longrightarrow prefix\ xs\ ys$
unfolding $prefix-def$ **by** *blast*

lemma $prefixE$ [*elim?*]:
assumes $prefix\ xs\ ys$
obtains zs **where** $ys = xs @ zs$
using *assms* **unfolding** $prefix-def$ **by** *blast*

lemma $strict-prefixI'$ [*intro?*]: $ys = xs @ z \# zs \Longrightarrow strict-prefix\ xs\ ys$
unfolding $strict-prefix-def\ prefix-def$ **by** *blast*

lemma $strict-prefixE'$ [*elim?*]:
assumes $strict-prefix\ xs\ ys$
obtains $z\ zs$ **where** $ys = xs @ z \# zs$

proof –

from $\langle strict-prefix\ xs\ ys \rangle$ **obtain** us **where** $ys = xs @ us$ **and** $xs \neq ys$
unfolding $strict-prefix-def\ prefix-def$ **by** *blast*
with *that* **show** *?thesis* **by** (*auto simp add: neq-Nil-conv*)

qed

lemma $strict-prefixI$ [*intro?*]: $prefix\ xs\ ys \Longrightarrow xs \neq ys \Longrightarrow strict-prefix\ xs\ ys$
by (*fact prefix-order.le-neq-trans*)

lemma $strict-prefixE$ [*elim?*]:
fixes $xs\ ys :: 'a\ list$
assumes $strict-prefix\ xs\ ys$
obtains $prefix\ xs\ ys$ **and** $xs \neq ys$
using *assms* **unfolding** $strict-prefix-def$ **by** *blast*

57.2 Basic properties of prefixes

theorem *Nil-prefix* [simp]: $\text{prefix } [] \ xs$
by (fact *prefix-bot.bot-least*)

theorem *prefix-Nil* [simp]: $(\text{prefix } xs \ []) = (xs = [])$
by (fact *prefix-bot.bot-unique*)

lemma *prefix-snoc* [simp]: $\text{prefix } xs \ (ys @ [y]) \longleftrightarrow xs = ys @ [y] \vee \text{prefix } xs \ ys$
proof

assume $\text{prefix } xs \ (ys @ [y])$
then obtain zs **where** $zs: ys @ [y] = xs @ zs \ ..$
show $xs = ys @ [y] \vee \text{prefix } xs \ ys$
by (metis *append-Nil2 butlast-append butlast-snoc prefixI zs*)

next

assume $xs = ys @ [y] \vee \text{prefix } xs \ ys$
then show $\text{prefix } xs \ (ys @ [y])$
using *prefix-def prefix-order.order-trans* **by** *blast*

qed

lemma *Cons-prefix-Cons* [simp]: $\text{prefix } (x \# xs) \ (y \# ys) = (x = y \wedge \text{prefix } xs \ ys)$
by (auto simp add: *prefix-def*)

lemma *prefix-code* [code]:
 $\text{prefix } [] \ xs \longleftrightarrow \text{True}$
 $\text{prefix } (x \# xs) \ [] \longleftrightarrow \text{False}$
 $\text{prefix } (x \# xs) \ (y \# ys) \longleftrightarrow x = y \wedge \text{prefix } xs \ ys$
by *simp-all*

lemma *same-prefix-prefix* [simp]: $\text{prefix } (xs @ ys) \ (xs @ zs) = \text{prefix } ys \ zs$
by (induct xs) *simp-all*

lemma *same-prefix-nil* [simp]: $\text{prefix } (xs @ ys) \ xs = (ys = [])$
by (simp add: *prefix-def*)

lemma *prefix-prefix* [simp]: $\text{prefix } xs \ ys \implies \text{prefix } xs \ (ys @ zs)$
unfolding *prefix-def* **by** *fastforce*

lemma *append-prefixD*: $\text{prefix } (xs @ ys) \ zs \implies \text{prefix } xs \ zs$
by (auto simp add: *prefix-def*)

theorem *prefix-Cons*: $\text{prefix } xs \ (y \# ys) = (xs = [] \vee (\exists zs. xs = y \# zs \wedge \text{prefix } zs \ ys))$
by (cases xs) (auto simp add: *prefix-def*)

theorem *prefix-append*:
 $\text{prefix } xs \ (ys @ zs) = (\text{prefix } xs \ ys \vee (\exists us. xs = ys @ us \wedge \text{prefix } us \ zs))$
proof (induct zs rule: *rev-induct*)
case *Nil*
then show ?case **by** *force*

```

next
  case (snoc x xs)
  then show ?case
    by (metis append.assoc prefix-snoc)
qed

```

```

lemma append-one-prefix:
  prefix xs ys  $\implies$  length xs < length ys  $\implies$  prefix (xs @ [ys ! length xs]) ys
proof (unfold prefix-def)
  assume a1:  $\exists$  zs. ys = xs @ zs
  then obtain sk :: 'a list where sk: ys = xs @ sk by fastforce
  assume a2: length xs < length ys
  have f1:  $\bigwedge v. (\square :: 'a \text{ list}) @ v = v$  using append-Nil2 by simp
  have  $\square \neq sk$  using a1 a2 sk less-not-refl by force
  hence  $\exists v. xs @ hd sk \# v = ys$  using sk by (metis hd-Cons-tl)
  thus  $\exists zs. ys = (xs @ [ys ! length xs]) @ zs$  using f1 by fastforce
qed

```

```

theorem prefix-length-le: prefix xs ys  $\implies$  length xs  $\leq$  length ys
  by (auto simp add: prefix-def)

```

```

lemma prefix-same-cases:
  prefix (xs1 :: 'a list) ys  $\implies$  prefix xs2 ys  $\implies$  prefix xs1 xs2  $\vee$  prefix xs2 xs1
  unfolding prefix-def by (force simp: append-eq-append-conv2)

```

```

lemma prefix-length-prefix:
  prefix ps xs  $\implies$  prefix qs xs  $\implies$  length ps  $\leq$  length qs  $\implies$  prefix ps qs
by (auto simp: prefix-def) (metis append-Nil2 append-eq-append-conv-if)

```

```

lemma set-mono-prefix: prefix xs ys  $\implies$  set xs  $\subseteq$  set ys
  by (auto simp add: prefix-def)

```

```

lemma take-is-prefix: prefix (take n xs) xs
  unfolding prefix-def by (metis append-take-drop-id)

```

```

lemma takeWhile-is-prefix: prefix (takeWhile P xs) xs
  unfolding prefix-def by (metis takeWhile-dropWhile-id)

```

```

lemma prefixeq-butlast: prefix (butlast xs) xs
  by (simp add: butlast-conv-take take-is-prefix)

```

```

lemma prefix-map-rightE:
  assumes prefix xs (map f ys)
  shows  $\exists xs'. \text{prefix } xs' \text{ ys} \wedge xs = \text{map } f \text{ } xs'$ 
proof -
  define n where n = length xs
  have xs = take n (map f ys)
    using assms by (auto simp: prefix-def n-def)
  thus ?thesis

```

by (intro exI[of - take n ys]) (auto simp: take-map take-is-prefix)
qed

lemma map-mono-prefix: prefix xs ys \implies prefix (map f xs) (map f ys)
by (auto simp: prefix-def)

lemma filter-mono-prefix: prefix xs ys \implies prefix (filter P xs) (filter P ys)
by (auto simp: prefix-def)

lemma sorted-antimono-prefix: prefix xs ys \implies sorted ys \implies sorted xs
by (metis sorted-append prefix-def)

lemma prefix-length-less: strict-prefix xs ys \implies length xs < length ys
by (auto simp: strict-prefix-def prefix-def)

lemma prefix-snocD: prefix (xs@[x]) ys \implies strict-prefix xs ys
by (simp add: strict-prefixI' prefix-order.dual-order.strict-trans1)

lemma strict-prefix-simps [simp, code]:
strict-prefix xs [] \longleftrightarrow False
strict-prefix [] (x # xs) \longleftrightarrow True
strict-prefix (x # xs) (y # ys) \longleftrightarrow x = y \wedge strict-prefix xs ys
by (simp-all add: strict-prefix-def cong: conj-cong)

lemma take-strict-prefix: strict-prefix xs ys \implies strict-prefix (take n xs) ys
proof (induct n arbitrary: xs ys)
case 0
then show ?case by (cases ys) simp-all
next
case (Suc n)
then show ?case by (metis prefix-order.less-trans strict-prefixI take-is-prefix)
qed

lemma prefix-takeWhile:
assumes prefix xs ys
shows prefix (takeWhile P xs) (takeWhile P ys)
proof –
from assms obtain zs where ys: ys = xs @ zs
by (auto simp: prefix-def)
have prefix (takeWhile P xs) (takeWhile P (xs @ zs))
by (induction xs) auto
thus ?thesis by (simp add: ys)
qed

lemma prefix-dropWhile:
assumes prefix xs ys
shows prefix (dropWhile P xs) (dropWhile P ys)
proof –
from assms obtain zs where ys: ys = xs @ zs


```

  by (auto simp: prefix-def)
  have prefix (dropWhile P xs) (dropWhile P (xs @ zs))
    by (induction xs) auto
  thus ?thesis by (simp add: ys)
qed

```

```

lemma prefix-remdups-adj:
  assumes prefix xs ys
  shows prefix (remdups-adj xs) (remdups-adj ys)
  using assms
proof (induction length xs arbitrary: xs ys rule: less-induct)
  case (less xs)
  show ?case
  proof (cases xs)
    case [simp]: (Cons x xs')
    then obtain y ys' where [simp]: ys = y # ys'
      using ⟨prefix xs ys⟩ by (cases ys) auto
    from less show ?thesis
      by (auto simp: remdups-adj-Cons' less-Suc-eq-le length-dropWhile-le
        intro!: less prefix-dropWhile)
  qed auto
qed

```

```

lemma not-prefix-cases:
  assumes pfx:  $\neg$  prefix ps ls
  obtains
    (c1)  $ps \neq []$  and  $ls = []$ 
  | (c2)  $a \text{ as } x \text{ xs}$  where  $ps = a \# \text{as}$  and  $ls = x \# \text{xs}$  and  $x = a$  and  $\neg$  prefix as xs
  | (c3)  $a \text{ as } x \text{ xs}$  where  $ps = a \# \text{as}$  and  $ls = x \# \text{xs}$  and  $x \neq a$ 
proof (cases ps)
  case Nil
  then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ⟨ps = a # as⟩
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append-Nil2 pfx c1 same-prefix-nil)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      have  $\neg$  prefix as xs using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
qed

```

qed
qed

lemma *not-prefix-induct* [consumes 1, case-names Nil Neq Eq]:
 assumes *np*: $\neg \text{prefix } ps \text{ } ls$
 and *base*: $\bigwedge x \text{ } xs. P \text{ } (x \# xs)$ []
 and *r1*: $\bigwedge x \text{ } xs \text{ } y \text{ } ys. x \neq y \implies P \text{ } (x \# xs) \text{ } (y \# ys)$
 and *r2*: $\bigwedge x \text{ } xs \text{ } y \text{ } ys. \llbracket x = y; \neg \text{prefix } xs \text{ } ys; P \text{ } xs \text{ } ys \rrbracket \implies P \text{ } (x \# xs) \text{ } (y \# ys)$
 shows *P ps ls* **using** *np*
proof (*induct ls arbitrary: ps*)
 case Nil
 then show ?case
 by (*auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base*)
next
 case (*Cons y ys*)
 then have *npfx*: $\neg \text{prefix } ps \text{ } (y \# ys)$ **by** *simp*
 then obtain *x xs* **where** *pv*: $ps = x \# xs$
by (*rule not-prefix-cases*) *auto*
 show ?case **by** (*metis Cons.hyps Cons-prefix-Cons npfx pv r1 r2*)
 qed

57.3 Prefixes

primrec *prefixes* **where**
prefixes [] = [] |
prefixes ($x \# xs$) = [] # *map* ((#) *x*) (*prefixes xs*)

lemma *in-set-prefixes[simp]*: $xs \in \text{set } (\text{prefixes } ys) \longleftrightarrow \text{prefix } xs \text{ } ys$

proof (*induct xs arbitrary: ys*)
 case Nil
 then show ?case **by** (*cases ys*) *auto*
next
 case (*Cons a xs*)
 then show ?case **by** (*cases ys*) *auto*
 qed

lemma *length-prefixes[simp]*: $\text{length } (\text{prefixes } xs) = \text{length } xs + 1$
by (*induction xs*) *auto*

lemma *distinct-prefixes [intro]*: *distinct* (*prefixes xs*)
by (*induction xs*) (*auto simp: distinct-map*)

lemma *prefixes-snoc [simp]*: $\text{prefixes } (xs @ [x]) = \text{prefixes } xs @ [xs @ [x]]$
by (*induction xs*) *auto*

lemma *prefixes-not-Nil [simp]*: $\text{prefixes } xs \neq []$
by (*cases xs*) *auto*

lemma *hd-prefixes [simp]*: $\text{hd } (\text{prefixes } xs) = []$

by (cases *xs*) *simp-all*

lemma *last-prefixes* [*simp*]: *last* (*prefixes xs*) = *xs*
by (*induction xs*) (*simp-all add: last-map*)

lemma *prefixes-append*:
prefixes (*xs @ ys*) = *prefixes xs @ map* ($\lambda y s'. xs @ y s'$) (*tl* (*prefixes ys*))
proof (*induction xs*)
case *Nil*
thus ?*case* **by** (cases *ys*) *auto*
qed *simp-all*

lemma *prefixes-eq-snoc*:
prefixes ys = *xs @ [x]* \longleftrightarrow
(*ys* = [] \wedge *xs* = [] \vee ($\exists z zs. ys = zs @ [z] \wedge xs = prefixes zs$)) $\wedge x = ys$
by (cases *ys* *rule: rev-cases*) *auto*

lemma *prefixes-tailrec* [*code*]:
prefixes xs = *rev* (*snd* (*foldl* ($\lambda(acc1, acc2) x. (x \# acc1, rev (x \# acc1) \# acc2)$)
([], []) *xs*))
proof –
have *foldl* ($\lambda(acc1, acc2) x. (x \# acc1, rev (x \# acc1) \# acc2)$) (*ys*, *rev ys* # *zs*)
xs =
(*rev xs @ ys*, *rev* (*map* ($\lambda as. rev ys @ as$) (*prefixes xs*)) @ *zs*) **for** *ys zs*
proof (*induction xs arbitrary: ys zs*)
case (*Cons x xs ys zs*)
from *Cons.IH*[*of x # ys rev ys # zs*]
show ?*case* **by** (*simp add: o-def*)
qed *simp-all*
from this [*of [] []*] **show** ?*thesis* **by** *simp*
qed

lemma *set-prefixes-eq*: *set* (*prefixes xs*) = {*ys. prefix ys xs*}
by *auto*

lemma *card-set-prefixes* [*simp*]: *card* (*set* (*prefixes xs*)) = *Suc* (*length xs*)
by (*subst distinct-card*) *auto*

lemma *set-prefixes-append*:
set (*prefixes* (*xs @ ys*)) = *set* (*prefixes xs*) \cup {*xs @ ys'* | *ys'. ys' ∈ set* (*prefixes ys*)}
by (*subst prefixes-append, cases ys*) *auto*

57.4 Longest Common Prefix

definition *Longest-common-prefix* :: 'a list set \Rightarrow 'a list **where**
Longest-common-prefix L = (*ARG-MAX length ps. $\forall xs \in L. prefix ps xs$*)

lemma *Longest-common-prefix-ex*: *L* $\neq \{\}$ \implies

$\exists ps. (\forall xs \in L. \text{prefix } ps \ xs) \wedge (\forall qs. (\forall xs \in L. \text{prefix } qs \ xs) \longrightarrow \text{size } qs \leq \text{size } ps)$
 (is - $\implies \exists ps. ?P \ L \ ps$)
proof(induction *LEAST* $n. \exists xs \in L. n = \text{length } xs$ arbitrary: L)
 case 0
 have $\square \in L$ **using** 0.hyps *LeastI*[of $\lambda n. \exists xs \in L. n = \text{length } xs$] $\langle L \neq \{\} \rangle$
 by auto
 hence $?P \ L \ \square$ **by**(auto)
 thus ?case ..
next
 case (Suc n)
 let ?EX = $\lambda n. \exists xs \in L. n = \text{length } xs$
 obtain $x \ xs$ **where** $xs: x \# xs \in L$ $\text{size } xs = n$ **using** Suc.premss Suc.hyps(2)
 by(metis *LeastI-ex*[of ?EX] *Suc-length-conv ex-in-conv*)
 hence $\square \notin L$ **using** Suc.hyps(2) **by** auto
 show ?case
 proof (cases $\forall xs \in L. \exists ys. xs = x \# ys$)
 case True
 let ?L = $\{ys. x \# ys \in L\}$
 have 1: (*LEAST* $n. \exists xs \in ?L. n = \text{length } xs$) = n
 using xs Suc.premss Suc.hyps(2) *Least-le*[of ?EX]
 by - (rule *Least-equality*, *fastforce*+)
 have 2: $?L \neq \{\}$ **using** $\langle x \# xs \in L \rangle$ **by** auto
 from Suc.hyps(1)[OF 1[symmetric] 2] **obtain** ps **where** IH: $?P \ ?L \ ps$..
 have $\text{length } qs \leq \text{Suc } (\text{length } ps)$
 if $\forall qs. (\forall xa. x \# xa \in L \longrightarrow \text{prefix } qs \ xa) \longrightarrow \text{length } qs \leq \text{length } ps$
 and $\forall xs \in L. \text{prefix } qs \ xs$ **for** qs
 proof -
 from that **have** $\text{length } (\text{tl } qs) \leq \text{length } ps$
 by (metis *Cons-prefix-Cons hd-Cons-tl list.sel*(2) *Nil-prefix*)
 thus ?thesis **by** auto
 qed
 hence $?P \ L \ (x \# ps)$ **using** True IH **by** auto
 thus ?thesis ..
next
 case False
 then obtain $y \ ys$ **where** $yys: x \neq y \ y \# ys \in L$ **using** $\langle \square \notin L \rangle$
 by (auto) (metis *list.exhaust*)
 have $\forall qs. (\forall xs \in L. \text{prefix } qs \ xs) \longrightarrow qs = \square$ **using** $yys \ \langle x \# xs \in L \rangle$
 by auto (metis *Cons-prefix-Cons prefix-Cons*)
 hence $?P \ L \ \square$ **by** auto
 thus ?thesis ..
qed
qed

lemma *Longest-common-prefix-unique*:
 $\langle \exists! ps. (\forall xs \in L. \text{prefix } ps \ xs) \wedge (\forall qs. (\forall xs \in L. \text{prefix } qs \ xs) \longrightarrow \text{length } qs \leq \text{length } ps) \rangle$
if $\langle L \neq \{\} \rangle$

apply (intro ex-ex1I[OF Longest-common-prefix-ex [OF that]])
by (meson that all-not-in-conv prefix-length-prefix prefix-order.dual-order.eq-iff)

lemma Longest-common-prefix-eq:

$\llbracket L \neq \{\}; \forall xs \in L. \text{prefix } ps \ xs;$
 $\forall qs. (\forall xs \in L. \text{prefix } qs \ xs) \longrightarrow \text{size } qs \leq \text{size } ps \rrbracket$
 $\implies \text{Longest-common-prefix } L = ps$

unfolding Longest-common-prefix-def arg-max-def is-arg-max-linorder
by(rule some1-equality[OF Longest-common-prefix-unique]) auto

lemma Longest-common-prefix-prefix:

$xs \in L \implies \text{prefix } (\text{Longest-common-prefix } L) \ xs$

unfolding Longest-common-prefix-def arg-max-def is-arg-max-linorder
by(rule someI2-ex[OF Longest-common-prefix-ex]) auto

lemma Longest-common-prefix-longest:

$L \neq \{\} \implies \forall xs \in L. \text{prefix } ps \ xs \implies \text{length } ps \leq \text{length } (\text{Longest-common-prefix } L)$

unfolding Longest-common-prefix-def arg-max-def is-arg-max-linorder
by(rule someI2-ex[OF Longest-common-prefix-ex]) auto

lemma Longest-common-prefix-max-prefix:

$L \neq \{\} \implies \forall xs \in L. \text{prefix } ps \ xs \implies \text{prefix } ps \ (\text{Longest-common-prefix } L)$

by(metis Longest-common-prefix-prefix Longest-common-prefix-longest
prefix-length-prefix ex-in-conv)

lemma Longest-common-prefix-Nil: $\llbracket \in L \implies \text{Longest-common-prefix } L = \llbracket$

using Longest-common-prefix-prefix prefix-Nil **by** blast

lemma Longest-common-prefix-image-Cons:

assumes $L \neq \{\}$

shows $\text{Longest-common-prefix } ((\#) \ x \ ' \ L) = x \ \# \ \text{Longest-common-prefix } L$

proof (intro Longest-common-prefix-eq strip)

show $\bigwedge qs. \forall xs \in (\#) \ x \ ' \ L. \text{prefix } qs \ xs \implies$

$\text{length } qs \leq \text{length } (x \ \# \ \text{Longest-common-prefix } L)$

by (metis assms Longest-common-prefix-longest[of L] Cons-prefix-Cons Suc-le-mono
hd-Cons-tl

image-eqI length-Cons prefix-bot.bot-least prefix-length-le)

qed (auto simp add: assms Longest-common-prefix-prefix)

lemma Longest-common-prefix-eq-Cons: **assumes** $L \neq \{\} \llbracket \notin L \ \forall xs \in L. \text{hd } xs =$
 x

shows $\text{Longest-common-prefix } L = x \ \# \ \text{Longest-common-prefix } \{ys. x \ \# \ ys \in L\}$

proof –

have $L = (\#) \ x \ ' \ \{ys. x \ \# \ ys \in L\}$ **using** assms(2,3)

by (auto simp: image-def)(metis hd-Cons-tl)

thus ?thesis

by (metis Longest-common-prefix-image-Cons image-is-empty assms(1))

qed

lemma *Longest-common-prefix-eq-Nil:*

$\llbracket x\#ys \in L; y\#zs \in L; x \neq y \rrbracket \implies \text{Longest-common-prefix } L = []$
by (*metis Longest-common-prefix-prefix list.inject prefix-Cons*)

fun *longest-common-prefix* :: 'a list \Rightarrow 'a list \Rightarrow 'a list **where**
longest-common-prefix (x#xs) (y#ys) =
 (if x=y then x # *longest-common-prefix* xs ys else []) |
longest-common-prefix - - = []

lemma *longest-common-prefix-prefix1:*

prefix (*longest-common-prefix* xs ys) xs
by(*induction* xs ys *rule: longest-common-prefix.induct*) *auto*

lemma *longest-common-prefix-prefix2:*

prefix (*longest-common-prefix* xs ys) ys
by(*induction* xs ys *rule: longest-common-prefix.induct*) *auto*

lemma *longest-common-prefix-max-prefix:*

$\llbracket \text{prefix } ps \ xs; \text{prefix } ps \ ys \rrbracket$
 $\implies \text{prefix } ps \ (\text{longest-common-prefix } xs \ ys)$
by(*induction* xs ys *arbitrary: ps rule: longest-common-prefix.induct*)
 (*auto simp: prefix-Cons*)

57.5 Parallel lists

definition *parallel* :: 'a list \Rightarrow 'a list \Rightarrow bool (**infixl** <||> 50)

where (xs || ys) = ($\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs$)

lemma *parallelI* [*intro*]: $\neg \text{prefix } xs \ ys \implies \neg \text{prefix } ys \ xs \implies xs \parallel ys$
unfolding *parallel-def* **by** *blast*

lemma *parallelE* [*elim*]:

assumes xs || ys
obtains $\neg \text{prefix } xs \ ys \wedge \neg \text{prefix } ys \ xs$
using *assms* **unfolding** *parallel-def* **by** *blast*

theorem *prefix-cases:*

obtains *prefix* xs ys | *strict-prefix* ys xs | xs || ys
unfolding *parallel-def* *strict-prefix-def* **by** *blast*

lemma *parallel-cancel:* a#xs || a#ys \implies xs || ys

by (*simp add: parallel-def*)

theorem *parallel-decomp:*

xs || ys $\implies \exists as \ b \ bs \ c \ cs. \ b \neq c \wedge xs = as @ b \# bs \wedge ys = as @ c \# cs$

proof (*induct rule: list-induct2', blast, force, force*)

case (4 x xs y ys)

then show ?case

proof (*cases* $x \neq y$, *blast*)
assume $\neg x \neq y$ **hence** $x = y$ **by** *blast*
then show *?thesis*
using $\lambda.hyps[OF\ parallel-cancel[OF\ \lambda.prem[s][folded\ \langle x = y \rangle]]]$
by (*meson Cons-eq-appendI*)
qed
qed

lemma *parallel-append*: $a \parallel b \implies a @ c \parallel b @ d$
by (*meson parallelE parallelI prefixI prefix-order.trans prefix-same-cases*)

lemma *parallel-appendI*: $xs \parallel ys \implies x = xs @ xs' \implies y = ys @ ys' \implies x \parallel y$
by (*simp add: parallel-append*)

lemma *parallel-commute*: $a \parallel b \longleftrightarrow b \parallel a$
unfolding *parallel-def* **by** *auto*

57.6 Suffix order on lists

definition *suffix* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where *suffix* $xs\ ys = (\exists\ zs.\ ys = zs @ xs)$

definition *strict-suffix* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where *strict-suffix* $xs\ ys \longleftrightarrow suffix\ xs\ ys \wedge xs \neq ys$

global-interpretation *suffix-order*: *ordering suffix strict-suffix*
by *standard (auto simp: suffix-def strict-suffix-def)*

interpretation *suffix-order*: *order suffix strict-suffix*
by *standard (auto simp: suffix-def strict-suffix-def)*

global-interpretation *suffix-bot*: *ordering-top* $\langle \lambda xs\ ys.\ suffix\ ys\ xs \rangle \langle \lambda xs\ ys.\ strict-suffix\ ys\ xs \rangle \langle [] \rangle$
by *standard (simp add: suffix-def)*

interpretation *suffix-bot*: *order-bot Nil suffix strict-suffix*
by *standard (simp add: suffix-def)*

lemma *suffixI* [*intro?*]: $ys = zs @ xs \implies suffix\ xs\ ys$
unfolding *suffix-def* **by** *blast*

lemma *suffixE* [*elim?*]:
assumes *suffix* $xs\ ys$
obtains zs **where** $ys = zs @ xs$
using *assms* **unfolding** *suffix-def* **by** *blast*

lemma *suffix-tl* [*simp*]: *suffix* (*tl* xs) xs
by (*induct xs*) (*auto simp: suffix-def*)

lemma *strict-suffix-tl* [*simp*]: $xs \neq [] \implies \text{strict-suffix } (tl\ xs)\ xs$
by (*induct xs*) (*auto simp: strict-suffix-def suffix-def*)

lemma *Nil-suffix* [*simp*]: $\text{suffix } []\ xs$
by (*simp add: suffix-def*)

lemma *suffix-Nil* [*simp*]: $(\text{suffix } xs\ []) = (xs = [])$
by (*auto simp add: suffix-def*)

lemma *suffix-ConsI*: $\text{suffix } xs\ ys \implies \text{suffix } xs\ (y \# ys)$
by (*auto simp add: suffix-def*)

lemma *suffix-ConsD*: $\text{suffix } (x \# xs)\ ys \implies \text{suffix } xs\ ys$
by (*auto simp add: suffix-def*)

lemma *suffix-appendI*: $\text{suffix } xs\ ys \implies \text{suffix } xs\ (zs @ ys)$
by (*auto simp add: suffix-def*)

lemma *suffix-appendD*: $\text{suffix } (zs @ xs)\ ys \implies \text{suffix } xs\ ys$
by (*auto simp add: suffix-def*)

lemma *strict-suffix-set-subset*: $\text{strict-suffix } xs\ ys \implies \text{set } xs \subseteq \text{set } ys$
by (*auto simp: strict-suffix-def suffix-def*)

lemma *set-mono-suffix*: $\text{suffix } xs\ ys \implies \text{set } xs \subseteq \text{set } ys$
by (*auto simp: suffix-def*)

lemma *sorted-antimono-suffix*: $\text{suffix } xs\ ys \implies \text{sorted } ys \implies \text{sorted } xs$
by (*metis sorted-append suffix-def*)

lemma *suffix-ConsD2*: $\text{suffix } (x \# xs)\ (y \# ys) \implies \text{suffix } xs\ ys$
proof –

assume $\text{suffix } (x \# xs)\ (y \# ys)$
then obtain zs **where** $y \# ys = zs @ x \# xs$..
then show *?thesis*
by (*induct zs*) (*auto intro!: suffix-appendI suffix-ConsI*)

qed

lemma *suffix-to-prefix* [*code*]: $\text{suffix } xs\ ys \longleftrightarrow \text{prefix } (rev\ xs)\ (rev\ ys)$
proof

assume $\text{suffix } xs\ ys$
then obtain zs **where** $ys = zs @ xs$..
then have $rev\ ys = rev\ xs @ rev\ zs$ **by** *simp*
then show $\text{prefix } (rev\ xs)\ (rev\ ys)$..

next

assume $\text{prefix } (rev\ xs)\ (rev\ ys)$
then obtain zs **where** $rev\ ys = rev\ xs @ zs$..
then have $rev\ (rev\ ys) = rev\ zs @ rev\ (rev\ xs)$ **by** *simp*
then have $ys = rev\ zs @ xs$ **by** *simp*

then show *suffix xs ys ..*
qed

lemma *strict-suffix-to-prefix* [code]: *strict-suffix xs ys \longleftrightarrow strict-prefix (rev xs) (rev ys)*
by (*auto simp: suffix-to-prefix strict-suffix-def strict-prefix-def*)

lemma *distinct-suffix*: *distinct ys \implies suffix xs ys \implies distinct xs*
by (*clarsimp elim!: suffixE*)

lemma *map-mono-suffix*: *suffix xs ys \implies suffix (map f xs) (map f ys)*
by (*auto elim!: suffixE intro: suffixI*)

lemma *map-mono-strict-suffix*: *strict-suffix xs ys \implies strict-suffix (map f xs) (map f ys)*
by (*auto simp: strict-suffix-def suffix-def*)

lemma *filter-mono-suffix*: *suffix xs ys \implies suffix (filter P xs) (filter P ys)*
by (*auto simp: suffix-def*)

lemma *suffix-drop*: *suffix (drop n as) as*
unfolding *suffix-def* **by** (*metis append-take-drop-id*)

lemma *suffix-dropWhile*: *suffix (dropWhile P xs) xs*
unfolding *suffix-def* **by** (*metis takeWhile-dropWhile-id*)

lemma *suffix-take*: *suffix xs ys \implies ys = take (length ys - length xs) ys @ xs*
by (*auto elim!: suffixE*)

lemma *strict-suffix-reflcp-conv*: *strict-suffix⁼⁼ = suffix*
by (*intro ext*) (*auto simp: suffix-def strict-suffix-def*)

lemma *suffix-lists*: *suffix xs ys \implies ys \in lists A \implies xs \in lists A*
unfolding *suffix-def* **by** *auto*

lemma *suffix-snoc* [simp]: *suffix xs (ys @ [y]) \longleftrightarrow xs = [] \vee (\exists zs. xs = zs @ [y] \wedge suffix zs ys)*
by (*cases xs rule: rev-cases*) (*auto simp: suffix-def*)

lemma *snoc-suffix-snoc* [simp]: *suffix (xs @ [x]) (ys @ [y]) = (x = y \wedge suffix xs ys)*
by (*auto simp add: suffix-def*)

lemma *same-suffix-suffix* [simp]: *suffix (ys @ xs) (zs @ xs) = suffix ys zs*
by (*simp add: suffix-to-prefix*)

lemma *same-suffix-nil* [simp]: *suffix (ys @ xs) xs = (ys = [])*
by (*simp add: suffix-to-prefix*)

theorem *suffix-Cons*: $\text{suffix } xs \ (y \# ys) \longleftrightarrow xs = y \# ys \vee \text{suffix } xs \ ys$
unfolding *suffix-def* **by** (*auto simp: Cons-eq-append-conv*)

theorem *suffix-append*:
 $\text{suffix } xs \ (ys \ @ \ zs) \longleftrightarrow \text{suffix } xs \ zs \vee (\exists xs'. xs = xs' \ @ \ zs \wedge \text{suffix } xs' \ ys)$
by (*auto simp: suffix-def append-eq-append-conv2*)

theorem *suffix-length-le*: $\text{suffix } xs \ ys \implies \text{length } xs \leq \text{length } ys$
by (*auto simp add: suffix-def*)

lemma *suffix-same-cases*:
 $\text{suffix } (xs_1 :: 'a \ \text{list}) \ ys \implies \text{suffix } xs_2 \ ys \implies \text{suffix } xs_1 \ xs_2 \vee \text{suffix } xs_2 \ xs_1$
unfolding *suffix-def* **by** (*force simp: append-eq-append-conv2*)

lemma *suffix-length-suffix*:
 $\text{suffix } ps \ xs \implies \text{suffix } qs \ xs \implies \text{length } ps \leq \text{length } qs \implies \text{suffix } ps \ qs$
by (*auto simp: suffix-to-prefix intro: prefix-length-prefix*)

lemma *suffix-length-less*: $\text{strict-suffix } xs \ ys \implies \text{length } xs < \text{length } ys$
by (*auto simp: strict-suffix-def suffix-def*)

lemma *suffix-ConsD'*: $\text{suffix } (x \# xs) \ ys \implies \text{strict-suffix } xs \ ys$
by (*auto simp: strict-suffix-def suffix-def*)

lemma *drop-strict-suffix*: $\text{strict-suffix } xs \ ys \implies \text{strict-suffix } (\text{drop } n \ xs) \ ys$
proof (*induct n arbitrary: xs ys*)
case 0
then show ?case **by** (*cases ys*) *simp-all*
next
case (*Suc n*)
then show ?case
by (*cases xs*) (*auto intro: Suc dest: suffix-ConsD' suffix-order.less-imp-le*)
qed

lemma *suffix-map-rightE*:
assumes $\text{suffix } xs \ (\text{map } f \ ys)$
shows $\exists xs'. \text{suffix } xs' \ ys \wedge xs = \text{map } f \ xs'$
proof –
from *assms* **obtain** xs' **where** $xs': \text{map } f \ ys = xs' \ @ \ xs$
by (*auto simp: suffix-def*)
define n **where** $n = \text{length } xs'$
have $xs = \text{drop } n \ (\text{map } f \ ys)$
by (*simp add: xs' n-def*)
thus ?thesis
by (*intro exI[of - drop n ys]*) (*auto simp: drop-map suffix-drop*)
qed

lemma *suffix-remdups-adj*: $\text{suffix } xs \ ys \implies \text{suffix } (\text{remdups-adj } xs) \ (\text{remdups-adj } ys)$

```

using prefix-remdups-adj[of rev xs rev ys]
by (simp add: suffix-to-prefix)

lemma not-suffix-cases:
  assumes pfx:  $\neg$  suffix ps ls
  obtains
    (c1) ps  $\neq$  [] and ls = []
  | (c2) a as x xs where ps = as@[a] and ls = xs@[x] and x = a and  $\neg$  suffix as
  xs
  | (c3) a as x xs where ps = as@[a] and ls = xs@[x] and x  $\neq$  a
proof (cases ps rule: rev-cases)
  case Nil
  then show ?thesis using pfx by simp
next
  case (snoc as a)
  note c =  $\langle ps = as@[a] \rangle$ 
  show ?thesis
  proof (cases ls rule: rev-cases)
  case Nil then show ?thesis by (metis append-Nil2 pfx c1 same-suffix-nil)
  next
  case (snoc xs x)
  show ?thesis
  proof (cases x = a)
  case True
  have  $\neg$  suffix as xs using pfx c snoc True by simp
  with c snoc True show ?thesis by (rule c2)
  next
  case False
  with c snoc show ?thesis by (rule c3)
  qed
qed
qed

lemma not-suffix-induct [consumes 1, case-names Nil Neq Eq]:
  assumes np:  $\neg$  suffix ps ls
  and base:  $\bigwedge x xs. P (xs@[x])$  []
  and r1:  $\bigwedge x xs y ys. x \neq y \implies P (xs@[x]) (ys@[y])$ 
  and r2:  $\bigwedge x xs y ys. \llbracket x = y; \neg \text{suffix } xs \text{ } ys; P \text{ } xs \text{ } ys \rrbracket \implies P (xs@[x]) (ys@[y])$ 
  shows P ps ls using np
proof (induct ls arbitrary: ps rule: rev-induct)
  case Nil
  then show ?case by (cases ps rule: rev-cases) (auto intro: base)
next
  case (snoc y ys ps)
  then have npfx:  $\neg$  suffix ps (ys @ [y]) by simp
  then obtain x xs where pv: ps = xs @ [x]
  by (rule not-suffix-cases) auto
  show ?case by (metis snoc.hyps snoc-suffix-snoc npfx pv r1 r2)
qed

```

lemma *parallelD1*: $x \parallel y \implies \neg \text{prefix } x \ y$
by *blast*

lemma *parallelD2*: $x \parallel y \implies \neg \text{prefix } y \ x$
by *blast*

lemma *parallel-Nil1* [*simp*]: $\neg x \parallel []$
unfolding *parallel-def* **by** *simp*

lemma *parallel-Nil2* [*simp*]: $\neg [] \parallel x$
unfolding *parallel-def* **by** *simp*

lemma *Cons-parallelI1*: $a \neq b \implies a \# as \parallel b \# bs$
by *auto*

lemma *Cons-parallelI2*: $\llbracket a = b; as \parallel bs \rrbracket \implies a \# as \parallel b \# bs$
by (*metis* *Cons-prefix-Cons parallelE parallelI*)

lemma *not-equal-is-parallel*:
assumes *neq*: $xs \neq ys$
and *len*: $\text{length } xs = \text{length } ys$
shows $xs \parallel ys$
using *len neq*
proof (*induct rule: list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a as b bs*)
have *ih*: $as \neq bs \implies as \parallel bs$ **by** *fact*
show *?case*
proof (*cases a = b*)
case *True*
then have $as \neq bs$ **using** *Cons* **by** *simp*
then show *?thesis* **by** (*rule Cons-parallelI2 [OF True ih]*)
next
case *False*
then show *?thesis* **by** (*rule Cons-parallelI1*)
qed
qed

57.7 Suffixes

primrec *suffixes* **where**

suffixes $[] = [[]]$
 $| \text{suffixes } (x \# xs) = \text{suffixes } xs @ [x \# xs]$

lemma *in-set-suffixes* [*simp*]: $xs \in \text{set } (\text{suffixes } ys) \longleftrightarrow \text{suffix } xs \ ys$

```

    by (induction ys) (auto simp: suffix-def Cons-eq-append-conv)

lemma distinct-suffixes [intro]: distinct (suffixes xs)
  by (induction xs) (auto simp: suffix-def)

lemma length-suffixes [simp]: length (suffixes xs) = Suc (length xs)
  by (induction xs) auto

lemma suffixes-snoc [simp]: suffixes (xs @ [x]) = [] # map (λys. ys @ [x]) (suffixes xs)
  by (induction xs) auto

lemma suffixes-not-Nil [simp]: suffixes xs ≠ []
  by (cases xs) auto

lemma hd-suffixes [simp]: hd (suffixes xs) = []
  by (induction xs) simp-all

lemma last-suffixes [simp]: last (suffixes xs) = xs
  by (cases xs) simp-all

lemma suffixes-append:
  suffixes (xs @ ys) = suffixes ys @ map (λxs'. xs' @ ys) (tl (suffixes xs))
proof (induction ys rule: rev-induct)
  case Nil
  thus ?case by (cases xs rule: rev-cases) auto
next
  case (snoc y ys)
  show ?case
    by (simp only: append.assoc [symmetric] suffixes-snoc snoc.IH) simp
qed

lemma suffixes-eq-snoc:
  suffixes ys = xs @ [x] ⟷
    (ys = [] ∧ xs = [] ∨ (∃ z zs. ys = z # zs ∧ xs = suffixes zs)) ∧ x = ys
  by (cases ys) auto

lemma suffixes-tailrec [code]:
  suffixes xs = rev (snd (foldl (λ(acc1, acc2) x. (x # acc1, (x # acc1) # acc2)) ([], []))
    (rev xs)))
proof -
  have foldl (λ(acc1, acc2) x. (x # acc1, (x # acc1) # acc2)) (ys, ys # zs) (rev xs)
  =
    (xs @ ys, rev (map (λas. as @ ys) (suffixes xs)) @ zs) for ys zs
proof (induction xs arbitrary: ys zs)
  case (Cons x xs ys zs)
  from Cons.IH[of ys zs]
  show ?case by (simp add: o-def case-prod-unfold)
qed simp-all

```

from *this* [of [] []] **show** ?thesis **by** simp
qed

lemma *set-suffixes-eq*: $\text{set } (\text{suffixes } xs) = \{ys. \text{suffix } ys \ xs\}$
by auto

lemma *card-set-suffixes* [simp]: $\text{card } (\text{set } (\text{suffixes } xs)) = \text{Suc } (\text{length } xs)$
by (subst distinct-card) auto

lemma *set-suffixes-append*:
 $\text{set } (\text{suffixes } (xs \ @ \ ys)) = \text{set } (\text{suffixes } ys) \cup \{xs' \ @ \ ys \mid xs'. xs' \in \text{set } (\text{suffixes } xs)\}$
by (subst suffixes-append, cases xs rule: rev-cases) auto

lemma *suffixes-conv-prefixes*: $\text{suffixes } xs = \text{map rev } (\text{prefixes } (\text{rev } xs))$
by (induction xs) auto

lemma *prefixes-conv-suffixes*: $\text{prefixes } xs = \text{map rev } (\text{suffixes } (\text{rev } xs))$
by (induction xs) auto

lemma *prefixes-rev*: $\text{prefixes } (\text{rev } xs) = \text{map rev } (\text{suffixes } xs)$
by (induction xs) auto

lemma *suffixes-rev*: $\text{suffixes } (\text{rev } xs) = \text{map rev } (\text{prefixes } xs)$
by (induction xs) auto

57.8 Homeomorphic embedding on lists

inductive *list-emb* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$
for $P :: ('a \Rightarrow 'a \Rightarrow \text{bool})$

where

list-emb-Nil [intro, simp]: $\text{list-emb } P \ [] \ ys$
| *list-emb-Cons* [intro]: $\text{list-emb } P \ xs \ ys \Longrightarrow \text{list-emb } P \ xs \ (y\#ys)$
| *list-emb-Cons2* [intro]: $P \ x \ y \Longrightarrow \text{list-emb } P \ xs \ ys \Longrightarrow \text{list-emb } P \ (x\#xs) \ (y\#ys)$

lemma *list-emb-mono*:
assumes $\bigwedge x \ y. P \ x \ y \longrightarrow Q \ x \ y$
shows $\text{list-emb } P \ xs \ ys \longrightarrow \text{list-emb } Q \ xs \ ys$

proof

assume $\text{list-emb } P \ xs \ ys$
then show $\text{list-emb } Q \ xs \ ys$ **by** (induct) (auto simp: assms)
qed

lemma *list-emb-Nil2* [simp]:
assumes $\text{list-emb } P \ xs \ []$ **shows** $xs = []$
using assms **by** (cases rule: list-emb.cases) auto

lemma *list-emb-refl*:

```

assumes  $\bigwedge x. x \in \text{set } xs \implies P \ x \ x$ 
shows  $\text{list-emb } P \ xs \ xs$ 
using assms by (induct xs) auto

lemma list-emb-Cons-Nil [simp]:  $\text{list-emb } P \ (x \# xs) \ [] = \text{False}$ 
proof
  show  $\text{False}$  if  $\text{list-emb } P \ (x \# xs) \ []$ 
    using list-emb-Nil2 [OF that] by simp
  show  $\text{list-emb } P \ (x \# xs) \ []$  if  $\text{False}$ 
    using that ..
qed

lemma list-emb-append2 [intro]:  $\text{list-emb } P \ xs \ ys \implies \text{list-emb } P \ xs \ (zs \ @ \ ys)$ 
by (induct zs) auto

lemma list-emb-prefix [intro]:
  assumes  $\text{list-emb } P \ xs \ ys$  shows  $\text{list-emb } P \ xs \ (ys \ @ \ zs)$ 
  using assms
  by (induct arbitrary: zs) auto

lemma list-emb-ConsD:
  assumes  $\text{list-emb } P \ (x \# xs) \ ys$ 
  shows  $\exists us \ v \ vs. ys = us \ @ \ v \ \# \ vs \wedge P \ x \ v \wedge \text{list-emb } P \ xs \ vs$ 
using assms
proof (induct x  $\equiv$  x # xs ys arbitrary: x xs)
  case list-emb-Cons
    then show ?case by (metis append-Cons)
next
  case (list-emb-Cons2 x y xs ys)
    then show ?case by blast
qed

lemma list-emb-appendD:
  assumes  $\text{list-emb } P \ (xs \ @ \ ys) \ zs$ 
  shows  $\exists us \ vs. zs = us \ @ \ vs \wedge \text{list-emb } P \ xs \ us \wedge \text{list-emb } P \ ys \ vs$ 
using assms
proof (induction xs arbitrary: ys zs)
  case Nil then show ?case by auto
next
  case (Cons x xs)
    then obtain us v vs where
      zs: zs = us @ v # vs and p: P x v and lh: list-emb P (xs @ ys) vs
    by (auto dest: list-emb-ConsD)
    obtain sk0 :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list and sk1 :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  'a list
    where
      sk:  $\forall x_0 \ x_1. \neg \text{list-emb } P \ (xs \ @ \ x_0) \ x_1 \vee sk_0 \ x_0 \ x_1 \ @ \ sk_1 \ x_0 \ x_1 = x_1 \wedge \text{list-emb } P \ xs \ (sk_0 \ x_0 \ x_1) \wedge \text{list-emb } P \ x_0 \ (sk_1 \ x_0 \ x_1)$ 
    using Cons(1) by (metis (no-types))
    hence  $\forall x_2. \text{list-emb } P \ (x \ # \ xs) \ (x_2 \ @ \ v \ \# \ sk_0 \ ys \ vs)$  using p lh by auto

```

thus ?case using lh zs sk by (metis (no-types) append-Cons append-assoc)
qed

lemma list-emb-strict-suffix:
assumes list-emb P xs ys and strict-suffix ys zs
shows list-emb P xs zs
using assms(2) and list-emb-append2 [OF assms(1)] by (auto simp: strict-suffix-def
suffix-def)

lemma list-emb-suffix:
assumes list-emb P xs ys and suffix ys zs
shows list-emb P xs zs
using assms and list-emb-strict-suffix
unfolding strict-suffix-reflclp-conv[symmetric] by auto

lemma list-emb-length: list-emb P xs ys \implies length xs \leq length ys
by (induct rule: list-emb.induct) auto

lemma list-emb-trans:
assumes $\bigwedge x y z. [x \in \text{set } xs; y \in \text{set } ys; z \in \text{set } zs; P x y; P y z] \implies P x z$
shows $[list-emb P xs ys; list-emb P ys zs] \implies list-emb P xs zs$
proof –
assume list-emb P xs ys and list-emb P ys zs
then show list-emb P xs zs using assms
proof (induction arbitrary: zs)
case list-emb-Nil show ?case by blast
next
case (list-emb-Cons xs ys y)
from list-emb-ConsD [OF $\langle list-emb P (y \# ys) zs \rangle$] obtain us v vs
where zs: zs = us @ v # vs and $P = y v$ and list-emb P ys vs by blast
then have list-emb P ys (v # vs) by blast
then have list-emb P ys zs unfolding zs by (rule list-emb-append2)
from list-emb-Cons.IH [OF this] and list-emb-Cons.premis show ?case by auto
next
case (list-emb-Cons2 x y xs ys)
from list-emb-ConsD [OF $\langle list-emb P (y \# ys) zs \rangle$] obtain us v vs
where zs: zs = us @ v # vs and $P y v$ and list-emb P ys vs by blast
with list-emb-Cons2 have list-emb P xs vs by auto
moreover have $P x v$
proof –
from zs have $v \in \text{set } zs$ by auto
moreover have $x \in \text{set } (x \# xs)$ and $y \in \text{set } (y \# ys)$ by simp-all
ultimately show ?thesis
using $\langle P x y \rangle$ and $\langle P y v \rangle$ and list-emb-Cons2
by blast
qed
ultimately have list-emb P (x # xs) (v # vs) by blast
then show ?case unfolding zs by (rule list-emb-append2)
qed

qed

lemma *list-emb-set*:

assumes *list-emb* *P* *xs* *ys* **and** $x \in \text{set } xs$
obtains *y* **where** $y \in \text{set } ys$ **and** $P \ x \ y$
using *assms* **by** (*induct*) *auto*

lemma *list-emb-Cons-iff1* [*simp*]:

assumes $P \ x \ y$
shows $\text{list-emb } P \ (x\#xs) \ (y\#ys) \longleftrightarrow \text{list-emb } P \ xs \ ys$
using *assms* **by** (*subst list-emb.simps*) (*auto dest: list-emb-ConsD*)

lemma *list-emb-Cons-iff2* [*simp*]:

assumes $\neg P \ x \ y$
shows $\text{list-emb } P \ (x\#xs) \ (y\#ys) \longleftrightarrow \text{list-emb } P \ (x\#xs) \ ys$
using *assms* **by** (*subst list-emb.simps*) *auto*

lemma *list-emb-code* [*code*]:

$\text{list-emb } P \ [] \ ys \longleftrightarrow \text{True}$
 $\text{list-emb } P \ (x\#xs) \ [] \longleftrightarrow \text{False}$
 $\text{list-emb } P \ (x\#xs) \ (y\#ys) \longleftrightarrow (\text{if } P \ x \ y \text{ then } \text{list-emb } P \ xs \ ys \text{ else } \text{list-emb } P \ (x\#xs) \ ys)$
by *simp-all*

57.9 Subsequences (special case of homeomorphic embedding)

abbreviation *subseq* :: '*a* list \Rightarrow '*a* list \Rightarrow bool

where $\text{subseq } xs \ ys \equiv \text{list-emb } (=) \ xs \ ys$

definition *strict-subseq* **where** $\text{strict-subseq } xs \ ys \longleftrightarrow xs \neq ys \wedge \text{subseq } xs \ ys$

lemma *subseq-Cons2*: $\text{subseq } xs \ ys \Longrightarrow \text{subseq } (x\#xs) \ (x\#ys)$ **by** *auto*

lemma *subseq-same-length*:

assumes $\text{subseq } xs \ ys$ **and** $\text{length } xs = \text{length } ys$ **shows** $xs = ys$
using *assms* **by** (*induct*) (*auto dest: list-emb-length*)

lemma *not-subseq-length* [*simp*]: $\text{length } ys < \text{length } xs \Longrightarrow \neg \text{subseq } xs \ ys$

by (*metis list-emb-length linorder-not-less*)

lemma *subseq-Cons'*: $\text{subseq } (x\#xs) \ ys \Longrightarrow \text{subseq } xs \ ys$

by (*induct xs, simp, blast dest: list-emb-ConsD*)

lemma *subseq-Cons2'*:

assumes $\text{subseq } (x\#xs) \ (x\#ys)$ **shows** $\text{subseq } xs \ ys$
using *assms* **by** (*cases*) (*rule subseq-Cons'*)

lemma *subseq-Cons2-neg*:

```

assumes subseq (x#xs) (y#ys)
shows  $x \neq y \implies \text{subseq } (x\#xs) \text{ } ys$ 
using assms by (cases) auto

```

```

lemma subseq-Cons2-iff [simp]:
   $\text{subseq } (x\#xs) \text{ } (y\#ys) = (\text{if } x = y \text{ then } \text{subseq } xs \text{ } ys \text{ else } \text{subseq } (x\#xs) \text{ } ys)$ 
by simp

```

```

lemma subseq-append':  $\text{subseq } (zs @ xs) \text{ } (zs @ ys) \longleftrightarrow \text{subseq } xs \text{ } ys$ 
by (induct zs) simp-all

```

global-interpretation *subseq-order*: *ordering subseq strict-subseq*
proof

```

  show  $\langle \text{subseq } xs \text{ } xs \rangle$  for xs ::  $\langle 'a \text{ list} \rangle$ 
    using refl by (rule list-emb-refl)
  show  $\langle \text{subseq } xs \text{ } zs \rangle$  if  $\langle \text{subseq } xs \text{ } ys \rangle$  and  $\langle \text{subseq } ys \text{ } zs \rangle$ 
    for xs ys zs ::  $\langle 'a \text{ list} \rangle$ 
    using trans [OF refl] that by (rule list-emb-trans) simp
  show  $\langle xs = ys \rangle$  if  $\langle \text{subseq } xs \text{ } ys \rangle$  and  $\langle \text{subseq } ys \text{ } xs \rangle$ 
    for xs ys ::  $\langle 'a \text{ list} \rangle$ 
  using that proof induction
    case list-emb-Nil
      from list-emb-Nil2 [OF this] show ?case by simp
    next
      case list-emb-Cons2
        then show ?case by simp
    next
      case list-emb-Cons
        hence False using subseq-Cons' by fastforce
        then show ?case ..
  qed
  show  $\langle \text{strict-subseq } xs \text{ } ys \longleftrightarrow \text{subseq } xs \text{ } ys \wedge xs \neq ys \rangle$ 
    for xs ys ::  $\langle 'a \text{ list} \rangle$ 
    by (auto simp: strict-subseq-def)
qed

```

interpretation *subseq-order*: *order subseq strict-subseq*
by (*rule* *ordering-orderI*) *standard*

```

lemma in-set-subseqs [simp]:  $xs \in \text{set } (\text{subseqs } ys) \longleftrightarrow \text{subseq } xs \text{ } ys$ 
proof

```

```

  assume  $xs \in \text{set } (\text{subseqs } ys)$ 
  thus  $\text{subseq } xs \text{ } ys$ 
    by (induction ys arbitrary: xs) (auto simp: Let-def)
  next
    have [simp]:  $\square \in \text{set } (\text{subseqs } ys)$  for ys ::  $\langle 'a \text{ list} \rangle$ 
      by (induction ys) (auto simp: Let-def)
    assume  $\text{subseq } xs \text{ } ys$ 
    thus  $xs \in \text{set } (\text{subseqs } ys)$ 

```

by (*induction xs ys rule: list-emb.induct*) (*auto simp: Let-def*)
qed

lemma *set-subseqs-eq*: $\text{set } (\text{subseqs } ys) = \{xs. \text{subseq } xs \text{ } ys\}$
by *auto*

lemma *subseq-append-le-same-iff*: $\text{subseq } (xs @ ys) \text{ } ys \longleftrightarrow xs = []$
by (*auto dest: list-emb-length*)

lemma *subseq-singleton-left*: $\text{subseq } [x] \text{ } ys \longleftrightarrow x \in \text{set } ys$
by (*fastforce dest: list-emb-ConsD split-list-last*)

lemma *list-emb-append-mono*:
[*list-emb P xs xs'; list-emb P ys ys'*] $\implies \text{list-emb } P \text{ } (xs@ys) \text{ } (xs'@ys')$
by (*induct rule: list-emb.induct*) *auto*

lemma *prefix-imp-subseq [intro]*: $\text{prefix } xs \text{ } ys \implies \text{subseq } xs \text{ } ys$
by (*auto simp: prefix-def*)

lemma *suffix-imp-subseq [intro]*: $\text{suffix } xs \text{ } ys \implies \text{subseq } xs \text{ } ys$
by (*auto simp: suffix-def*)

a subsequence of a sorted list

lemma *sorted-subset-imp-subseq*:
fixes *xs* :: '*a*::order list'
assumes $\text{set } xs \subseteq \text{set } ys$ *sorted-wrt* (*<*) *xs sorted-wrt* (*≤*) *ys*
shows *subseq xs ys*
using *assms*
proof (*induction xs arbitrary: ys*)
case *Nil*
then show ?*case*
by *auto*
next
case (*Cons x xs*)
then have $x \in \text{set } ys$
by *auto*
then obtain *us vs* where $\S: ys = us @ [x] @ vs$
by (*metis append.left-neutral append-eq-Cons-conv split-list*)
moreover
have $\text{set } xs \subseteq \text{set } vs$
using *Cons.prem*s by (*fastforce simp: § sorted-wrt-append*)
with *Cons* have *subseq xs vs*
by (*metis § sorted-wrt.simps(2) sorted-wrt-append*)
ultimately show ?*case*
by *auto*
qed

57.10 Appending elements

lemma *subseq-append [simp]*:

```

subseq (xs @ zs) (ys @ zs)  $\longleftrightarrow$  subseq xs ys (is ?l = ?r)
proof
  have xs' = xs @ zs  $\wedge$  ys' = ys @ zs  $\longrightarrow$  subseq xs ys
    if subseq xs' ys' for xs' ys' xs ys zs :: 'a list
    using that
  proof (induct arbitrary: xs ys zs)
    case list-emb-Nil
    show ?case by simp
  next
    case (list-emb-Cons xs' ys' x)
    have ?case if ys = []
      using list-emb-Cons(1) that by auto
    moreover
    have ?case if ys = x#us for us
      using list-emb-Cons(2) that by (simp add: list-emb.list-emb-Cons)
    ultimately show ?case
      by (auto simp: Cons-eq-append-conv)
  next
    case (list-emb-Cons2 x y xs' ys')
    have ?case if xs = []
      using list-emb-Cons2(1) that by auto
    moreover
    have ?case if xs = x#us ys = x#vs for us vs
      using list-emb-Cons2 that by auto
    moreover
    have ?case if xs = x#us ys = [] for us
      using list-emb-Cons2(2) that by bestsimp
    ultimately show ?case
      using  $\langle x = y \rangle$  by (auto simp: Cons-eq-append-conv)
  qed
  then show ?l  $\implies$  ?r by blast
  show ?r  $\implies$  ?l by (metis list-emb-append-mono subseq-order.order-refl)
qed

lemma subseq-append-iff:
  subseq xs (ys @ zs)  $\longleftrightarrow$  ( $\exists$  xs1 xs2. xs = xs1 @ xs2  $\wedge$  subseq xs1 ys  $\wedge$  subseq xs2
  zs)
  (is ?lhs = ?rhs)
proof
  assume ?lhs thus ?rhs
  proof (induction xs ys @ zs arbitrary: ys zs rule: list-emb.induct)
    case (list-emb-Cons xs ws y ys zs)
    from list-emb-Cons(2)[of tl ys zs] and list-emb-Cons(2)[of [] tl zs] and list-emb-Cons(1,3)
    show ?case by (cases ys) auto
  next
    case (list-emb-Cons2 x y xs ws ys zs)
    from list-emb-Cons2(3)[of tl ys zs] and list-emb-Cons2(3)[of [] tl zs]
    and list-emb-Cons2(1,2,4)
    show ?case by (cases ys) (auto simp: Cons-eq-append-conv)
  qed

```

qed *auto*
qed (*auto intro: list-emb-append-mono*)

lemma *subseq-appendE* [*case-names append*]:
assumes *subseq xs (ys @ zs)*
obtains *xs1 xs2* **where** *xs = xs1 @ xs2 subseq xs1 ys subseq xs2 zs*
using *assms* **by** (*subst (asm) subseq-append-iff*) *auto*

lemma *subseq-drop-many*: *subseq xs ys \implies subseq xs (zs @ ys)*
by (*induct zs*) *auto*

lemma *subseq-rev-drop-many*: *subseq xs ys \implies subseq xs (ys @ zs)*
by (*metis append-Nil2 list-emb-Nil list-emb-append-mono*)

57.11 Relation to standard list operations

lemma *subseq-map*:
assumes *subseq xs ys* **shows** *subseq (map f xs) (map f ys)*
using *assms* **by** (*induct*) *auto*

lemma *subseq-filter-left* [*simp*]: *subseq (filter P xs) xs*
by (*induct xs*) *auto*

lemma *subseq-filter* [*simp*]:
assumes *subseq xs ys* **shows** *subseq (filter P xs) (filter P ys)*
using *assms* **by** *induct auto*

lemma *subseq-conv-nths*: *subseq xs ys \longleftrightarrow ($\exists N. xs = nth\ ys\ N$)*
(is ?L = ?R)

proof
show *?R* **if** *?L* **using** *that*
proof (*induct*)
case *list-emb-Nil*
show *?case* **by** (*metis nth-empty*)
next
case (*list-emb-Cons xs ys x*)
then obtain *N* **where** *xs = nth\ ys\ N* **by** *blast*
then have *xs = nth\ (x#ys) (Suc ' N)*
by (*clarsimp simp add: nth-Cons inj-image-mem-iff*)
then show *?case* **by** *blast*
next
case (*list-emb-Cons2 x y xs ys*)
then obtain *N* **where** *xs = nth\ ys\ N* **by** *blast*
then have *x#xs = nth\ (x#ys) (insert 0 (Suc ' N))*
by (*clarsimp simp add: nth-Cons inj-image-mem-iff*)
moreover from *list-emb-Cons2* **have** *x = y* **by** *simp*
ultimately show *?case* **by** *blast*
qed
show *?L* **if** *?R*

```

proof –
  from that obtain  $N$  where  $xs = nth\ ys\ N$  ..
  moreover have  $subseq\ (nth\ ys\ N)\ ys$ 
  proof (induct  $ys$  arbitrary:  $N$ )
    case  $Nil$ 
    show  $?case$  by simp
  next
    case  $Cons$ 
    then show  $?case$  by (auto simp:  $nths-Cons$ )
  qed
  ultimately show  $?thesis$  by simp
qed
qed

```

57.12 Contiguous sublists

57.12.1 *sublist*

definition $sublist :: 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
 $sublist\ xs\ ys = (\exists\ ps\ ss. ys = ps @ xs @ ss)$

definition $strict-sublist :: 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
 $strict-sublist\ xs\ ys \longleftrightarrow sublist\ xs\ ys \wedge xs \neq ys$

interpretation $sublist-order$: *order* $sublist\ strict-sublist$

proof

```

  fix  $xs\ ys\ zs :: 'a\ list$ 
  assume  $sublist\ xs\ ys\ sublist\ ys\ zs$ 
  then obtain  $xs1\ xs2\ ys1\ ys2$  where  $ys = xs1 @ xs @ xs2$   $zs = ys1 @ ys @ ys2$ 
    by (auto simp:  $sublist-def$ )
  hence  $zs = (ys1 @ xs1) @ xs @ (xs2 @ ys2)$  by simp
  thus  $sublist\ xs\ zs$  unfolding  $sublist-def$  by blast
next
  fix  $xs\ ys :: 'a\ list$ 
  show  $xs = ys$  if  $sublist\ xs\ ys\ sublist\ ys\ xs$ 
  proof –
    from that obtain  $as\ bs\ cs\ ds$  where  $xs: xs = as @ ys @ bs$  and  $ys: ys = cs @$ 
 $xs @ ds$ 
    by (auto simp:  $sublist-def$ )
    have  $xs = as @ cs @ xs @ ds @ bs$  by (subst  $xs$ , subst  $ys$ ) auto
    also have  $length\ \dots = length\ as + length\ cs + length\ xs + length\ bs + length$ 
 $ds$ 
    by simp
    finally have  $as = []\ bs = []$  by simp-all
    with  $xs$  show  $?thesis$  by simp
  qed
  thus  $strict-sublist\ xs\ ys \longleftrightarrow (sublist\ xs\ ys \wedge \neg sublist\ ys\ xs)$ 
    by (auto simp:  $strict-sublist-def$ )
qed (auto simp:  $strict-sublist-def\ sublist-def$  intro:  $exI[of\ - []]$ )

```

lemma *sublist-Nil-left* [*simp*, *intro*]: *sublist* [] *ys*
by (*auto simp: sublist-def*)

lemma *sublist-Cons-Nil* [*simp*]: $\neg \text{sublist } (x \# xs) []$
by (*auto simp: sublist-def*)

lemma *sublist-Nil-right* [*simp*]: *sublist* *xs* [] $\longleftrightarrow xs = []$
by (*cases xs*) *auto*

lemma *sublist-appendI* [*simp*, *intro*]: *sublist* *xs* (*ps* @ *xs* @ *ss*)
by (*auto simp: sublist-def*)

lemma *sublist-append-leftI* [*simp*, *intro*]: *sublist* *xs* (*ps* @ *xs*)
by (*auto simp: sublist-def intro: exI[of - []]*)

lemma *sublist-append-rightI* [*simp*, *intro*]: *sublist* *xs* (*xs* @ *ss*)
by (*metis append-eq-append-conv2 sublist-appendI*)

lemma *sublist-altdef*: *sublist* *xs* *ys* $\longleftrightarrow (\exists ys'. \text{prefix } ys' \text{ } ys \wedge \text{suffix } xs \text{ } ys')$
by (*metis append-assoc prefix-def sublist-def suffix-def*)

lemma *sublist-altdef'*: *sublist* *xs* *ys* $\longleftrightarrow (\exists ys'. \text{suffix } ys' \text{ } ys \wedge \text{prefix } xs \text{ } ys')$
by (*metis prefixE prefixI sublist-appendI sublist-def suffixE suffixI*)

lemma *sublist-Cons-right*: *sublist* *xs* (*y* # *ys*) $\longleftrightarrow \text{prefix } xs \text{ } (y \# ys) \vee \text{sublist } xs \text{ } ys$
by (*auto simp: sublist-def prefix-def Cons-eq-append-conv*)

lemma *sublist-code* [*code*]:
sublist [] *ys* $\longleftrightarrow \text{True}$
sublist (*x* # *xs*) [] $\longleftrightarrow \text{False}$
sublist (*x* # *xs*) (*y* # *ys*) $\longleftrightarrow \text{prefix } (x \# xs) \text{ } (y \# ys) \vee \text{sublist } (x \# xs) \text{ } ys$
by (*simp-all add: sublist-Cons-right*)

lemma *sublist-append*:
sublist *xs* (*ys* @ *zs*) \longleftrightarrow
sublist *xs* *ys* \vee *sublist* *xs* *zs* $\vee (\exists xs1 \text{ } xs2. xs = xs1 @ xs2 \wedge \text{suffix } xs1 \text{ } ys \wedge \text{prefix } xs2 \text{ } zs)$
by (*auto simp: sublist-altdef prefix-append suffix-append*)

lemma *map-mono-sublist*:
assumes *sublist* *xs* *ys*
shows *sublist* (*map* *f* *xs*) (*map* *f* *ys*)
proof –
from *assms* **obtain** *xs1* *xs2* **where** *ys*: *ys* = *xs1* @ *xs* @ *xs2*
by (*auto simp: sublist-def*)
have *map* *f* *ys* = *map* *f* *xs1* @ *map* *f* *xs* @ *map* *f* *xs2*
by (*auto simp: ys*)
thus ?thesis

by (auto simp: sublist-def)
qed

lemma *sublist-length-le*: *sublist xs ys \implies length xs \leq length ys*
by (auto simp add: sublist-def)

lemma *set-mono-sublist*: *sublist xs ys \implies set xs \subseteq set ys*
by (auto simp add: sublist-def)

lemma *prefix-imp-sublist* [*simp, intro*]: *prefix xs ys \implies sublist xs ys*
by (auto simp: sublist-def prefix-def intro: exI[of - []])

lemma *suffix-imp-sublist* [*simp, intro*]: *suffix xs ys \implies sublist xs ys*
by (auto simp: sublist-def suffix-def intro: exI[of - []])

lemma *sublist-take* [*simp, intro*]: *sublist (take n xs) xs*
by (rule prefix-imp-sublist[OF take-is-prefix])

lemma *sublist-takeWhile* [*simp, intro*]: *sublist (takeWhile P xs) xs*
by (rule prefix-imp-sublist[OF takeWhile-is-prefix])

lemma *sublist-drop* [*simp, intro*]: *sublist (drop n xs) xs*
by (rule suffix-imp-sublist[OF suffix-drop])

lemma *sublist-dropWhile* [*simp, intro*]: *sublist (dropWhile P xs) xs*
by (rule suffix-imp-sublist[OF suffix-dropWhile])

lemma *sublist-tl* [*simp, intro*]: *sublist (tl xs) xs*
by (rule suffix-imp-sublist) (simp-all add: suffix-drop)

lemma *sublist-butlast* [*simp, intro*]: *sublist (butlast xs) xs*
by (rule prefix-imp-sublist) (simp-all add: prefixreq-butlast)

lemma *sublist-rev* [*simp*]: *sublist (rev xs) (rev ys) = sublist xs ys*
proof

assume *sublist (rev xs) (rev ys)*
then obtain *as bs* **where** *rev ys = as @ rev xs @ bs*
by (auto simp: sublist-def)
also have *rev ... = rev bs @ xs @ rev as* **by** *simp*
finally show *sublist xs ys* **by** *simp*

next

assume *sublist xs ys*
then obtain *as bs* **where** *ys = as @ xs @ bs*
by (auto simp: sublist-def)
also have *rev ... = rev bs @ rev xs @ rev as* **by** *simp*
finally show *sublist (rev xs) (rev ys)* **by** *simp*

qed

lemma *sublist-rev-left*: *sublist (rev xs) ys = sublist xs (rev ys)*

by (*subst sublist-rev [symmetric]*) (*simp only: rev-rev-ident*)

lemma *sublist-rev-right*: *sublist xs (rev ys) = sublist (rev xs) ys*
by (*subst sublist-rev [symmetric]*) (*simp only: rev-rev-ident*)

lemma *snoc-sublist-snoc*:
sublist (xs @ [x]) (ys @ [y]) \longleftrightarrow
(x = y \wedge suffix xs ys \vee sublist (xs @ [x]) ys)
by (*subst (1 2) sublist-rev [symmetric]*)
(simp del: sublist-rev add: sublist-Cons-right suffix-to-prefix)

lemma *sublist-snoc*:
sublist xs (ys @ [y]) \longleftrightarrow suffix xs (ys @ [y]) \vee sublist xs ys
by (*subst (1 2) sublist-rev [symmetric]*)
(simp del: sublist-rev add: sublist-Cons-right suffix-to-prefix)

lemma *sublist-imp-subseq [intro]*: *sublist xs ys \implies subseq xs ys*
by (*auto simp: sublist-def*)

lemma *sublist-map-rightE*:
assumes *sublist xs (map f ys)*
shows $\exists xs'. \text{sublist } xs' \text{ ys} \wedge xs = \text{map } f \text{ } xs'$
proof –
note *takedown = sublist-take sublist-drop*
define *n* **where** *n = (length ys – length xs)*
from *assms* **obtain** *xs1 xs2* **where** *xs12: map f ys = xs1 @ xs @ xs2*
by (*auto simp: sublist-def*)
define *n* **where** *n = length xs1*
have *xs = take (length xs) (drop n (map f ys))*
by (*simp add: xs12 n-def*)
thus *?thesis*
by (*intro exI[of - take (length xs) (drop n ys)]*)
(auto simp: take-map drop-map intro!: takedown[THEN sublist-order.order.trans])
qed

lemma *sublist-remdups-adj*:
assumes *sublist xs ys*
shows *sublist (remdups-adj xs) (remdups-adj ys)*
proof –
from *assms* **obtain** *xs1 xs2* **where** *ys: ys = xs1 @ xs @ xs2*
by (*auto simp: sublist-def*)
have *suffix (remdups-adj (xs @ xs2)) (remdups-adj (xs1 @ xs @ xs2))*
by (*rule suffix-remdups-adj, rule suffix-appendI*) *auto*
then obtain *zs1* **where** *zs1: remdups-adj (xs1 @ xs @ xs2) = zs1 @ remdups-adj (xs @ xs2)*
by (*auto simp: suffix-def*)
have *prefix (remdups-adj xs) (remdups-adj (xs @ xs2))*
by (*intro prefix-remdups-adj*) *auto*
then obtain *zs2* **where** *zs2: remdups-adj (xs @ xs2) = remdups-adj xs @ zs2*

```

    by (auto simp: prefix-def)
  show ?thesis
    by (simp add: ys zs1 zs2)
qed

```

57.12.2 *sublists*

```

primrec sublists :: 'a list  $\Rightarrow$  'a list list where
  sublists [] = [[]]
| sublists (x # xs) = sublists xs @ map ((#) x) (prefixes xs)

```

```

lemma in-set-sublists [simp]:  $xs \in \text{set } (\text{sublists } ys) \longleftrightarrow \text{sublist } xs \text{ } ys$ 
  by (induction ys arbitrary: xs) (auto simp: sublist-Cons-right prefix-Cons)

```

```

lemma set-sublists-eq:  $\text{set } (\text{sublists } xs) = \{ys. \text{sublist } ys \text{ } xs\}$ 
  by auto

```

```

lemma length-sublists [simp]:  $\text{length } (\text{sublists } xs) = \text{Suc } (\text{length } xs * \text{Suc } (\text{length } xs) \text{ div } 2)$ 
  by (induction xs) simp-all

```

57.13 Parametricity

```

context includes lifting-syntax
begin

```

```

private lemma prefix-primrec:
  prefix = rec-list ( $\lambda xs. \text{True}$ ) ( $\lambda x \text{ } xs \text{ } xsa \text{ } ys.$ 
    case ys of []  $\Rightarrow$  False |  $y \# ys \Rightarrow x = y \wedge xsa \text{ } ys$ )
proof (intro ext, goal-cases)
  case (1 xs ys)
  show ?case by (induction xs arbitrary: ys) (auto simp: prefix-Cons split: list.splits)
qed

```

```

private lemma sublist-primrec:
  sublist = ( $\lambda xs \text{ } ys. \text{rec-list } (\lambda xs. xs = []) (\lambda y \text{ } ys \text{ } ysa \text{ } xs. \text{prefix } xs (y \# ys) \vee ysa \text{ } xs) \text{ } ys \text{ } xs$ )
proof (intro ext, goal-cases)
  case (1 xs ys)
  show ?case by (induction ys) (auto simp: sublist-Cons-right)
qed

```

```

private lemma list-emb-primrec:
  list-emb = ( $\lambda uu \text{ } l' \text{ } l. \text{rec-list } (\lambda P \text{ } xs. \text{List.null } xs) (\lambda y \text{ } ys \text{ } ysa \text{ } P \text{ } xs. \text{case } xs \text{ of } [] \Rightarrow \text{True}$ 
    |  $x \# xs \Rightarrow \text{if } P \text{ } x \text{ } y \text{ then } ysa \text{ } P \text{ } xs \text{ else } ysa \text{ } P (x \# xs)) \text{ } l \text{ } uu \text{ } l'$ )
proof (intro ext, goal-cases)
  case (1 P xs ys)
  show ?case
    by (induction ys arbitrary: xs)

```

(*auto simp: list-emb-code split: list.splits*)
qed

lemma *prefix-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *prefix prefix*
 unfolding *prefix-primrec* **by** *transfer-prover*

lemma *suffix-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *suffix suffix*
 unfolding *suffix-to-prefix [abs-def]* **by** *transfer-prover*

lemma *sublist-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *sublist sublist*
 unfolding *sublist-primrec* **by** *transfer-prover*

lemma *parallel-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *parallel parallel*
 unfolding *parallel-def* **by** *transfer-prover*

lemma *list-emb-transfer* [*transfer-rule*]:
 ((*A ===> A ===> (=)*) ===> *list-all2 A ===> list-all2 A ===> (=)*)
list-emb list-emb
 unfolding *list-emb-primrec* **by** *transfer-prover*

lemma *strict-prefix-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *strict-prefix strict-prefix*
 unfolding *strict-prefix-def* **by** *transfer-prover*

lemma *strict-suffix-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *strict-suffix strict-suffix*
 unfolding *strict-suffix-def* **by** *transfer-prover*

lemma *strict-subseq-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *strict-subseq strict-subseq*
 unfolding *strict-subseq-def* **by** *transfer-prover*

lemma *strict-sublist-transfer* [*transfer-rule*]:
 assumes [*transfer-rule*]: *bi-unique A*
 shows (*list-all2 A ===> list-all2 A ===> (=)*) *strict-sublist strict-sublist*
 unfolding *strict-sublist-def* **by** *transfer-prover*

```

lemma prefixes-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A == => list-all2 (list-all2 A)) prefixes prefixes
  unfolding prefixes-def by transfer-prover

```

```

lemma suffixes-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A == => list-all2 (list-all2 A)) suffixes suffixes
  unfolding suffixes-def by transfer-prover

```

```

lemma sublists-transfer [transfer-rule]:
  assumes [transfer-rule]: bi-unique A
  shows (list-all2 A == => list-all2 (list-all2 A)) sublists sublists
  unfolding sublists-def by transfer-prover

```

```

end

```

```

end

```

58 Linear Temporal Logic on Streams

```

theory Linear-Temporal-Logic-on-Streams
  imports Stream Sublist Extended-Nat Infinite-Set
begin

```

59 Preliminaries

```

lemma shift-prefix:
  assumes xl @- xs = yl @- ys and length xl ≤ length yl
  shows prefix xl yl
  using assms proof(induct xl arbitrary: yl xs ys)
    case (Cons x xl yl xs ys)
    thus ?case by (cases yl) auto
  qed auto

lemma shift-prefix-cases:
  assumes xl @- xs = yl @- ys
  shows prefix xl yl ∨ prefix yl xl
  using shift-prefix[OF assms]
  by (cases length xl ≤ length yl) (metis, metis assms nat-le-linear shift-prefix)

```

60 Linear temporal logic

Propositional connectives:

```

abbreviation (input) IMPL (infix <impl> 60)
where  $\varphi \text{ impl } \psi \equiv \lambda xs. \varphi xs \longrightarrow \psi xs$ 

```

abbreviation (*input*) *OR* (**infix** $\langle or \rangle$ 60)

where φ *or* $\psi \equiv \lambda xs. \varphi xs \vee \psi xs$

abbreviation (*input*) *AND* (**infix** $\langle aand \rangle$ 60)

where φ *aand* $\psi \equiv \lambda xs. \varphi xs \wedge \psi xs$

abbreviation (*input*) *not* **where** $\text{not } \varphi \equiv \lambda xs. \neg \varphi xs$

abbreviation (*input*) *true* $\equiv \lambda xs. \text{True}$

abbreviation (*input*) *false* $\equiv \lambda xs. \text{False}$

lemma *impl-not-or*: $\varphi \text{ impl } \psi = (\text{not } \varphi) \text{ or } \psi$

by *blast*

lemma *not-or*: $\text{not } (\varphi \text{ or } \psi) = (\text{not } \varphi) \text{ aand } (\text{not } \psi)$

by *blast*

lemma *not-aand*: $\text{not } (\varphi \text{ aand } \psi) = (\text{not } \varphi) \text{ or } (\text{not } \psi)$

by *blast*

lemma *non-not[simp]*: $\text{not } (\text{not } \varphi) = \varphi$ **by** *simp*

Temporal (LTL) connectives:

fun *holds* **where** $\text{holds } P \text{ } xs \longleftrightarrow P (\text{shd } xs)$

fun *next* **where** $\text{next } \varphi \text{ } xs = \varphi (\text{stl } xs)$

definition *HLD* $s = \text{holds } (\lambda x. x \in s)$

abbreviation *HLD-next* (**infixr** $\langle \cdot \rangle$ 65) **where**

$s \cdot P \equiv \text{HLD } s \text{ aand next } P$

context

notes $[[\text{inductive-internals}]]$

begin

inductive *ev* **for** φ **where**

base: $\varphi \text{ } xs \Longrightarrow \text{ev } \varphi \text{ } xs$

|

step: $\text{ev } \varphi (\text{stl } xs) \Longrightarrow \text{ev } \varphi \text{ } xs$

coinductive *alw* **for** φ **where**

alw: $[[\varphi \text{ } xs; \text{alw } \varphi (\text{stl } xs)]] \Longrightarrow \text{alw } \varphi \text{ } xs$

— weak until:

coinductive *UNTIL* (**infix** $\langle \text{until} \rangle$ 60) **for** $\varphi \text{ } \psi$ **where**

base: $\psi \text{ } xs \Longrightarrow (\varphi \text{ until } \psi) \text{ } xs$

|

step: $\llbracket \varphi \text{ xs}; (\varphi \text{ until } \psi) \text{ (stl xs)} \rrbracket \Longrightarrow (\varphi \text{ until } \psi) \text{ xs}$

end

lemma *holds-mono*:

assumes *holds*: *holds* $P \text{ xs}$ **and** 0 : $\bigwedge x. P \ x \Longrightarrow Q \ x$

shows *holds* $Q \text{ xs}$

using *assms* **by** *auto*

lemma *holds-aand*:

$(\text{holds } P \text{ aand holds } Q) \text{ steps} \longleftrightarrow \text{holds } (\lambda \text{ step. } P \text{ step} \wedge Q \text{ step}) \text{ steps}$ **by** *auto*

lemma *HLD-iff*: $HLD \ s \ \omega \longleftrightarrow shd \ \omega \in s$

by (*simp add: HLD-def*)

lemma *HLD-Stream[simp]*: $HLD \ X \ (x \ \#\# \ \omega) \longleftrightarrow x \in X$

by (*simp add: HLD-iff*)

lemma *next-mono*:

assumes *next*: *next* $\varphi \text{ xs}$ **and** 0 : $\bigwedge xs. \varphi \ xs \Longrightarrow \psi \ xs$

shows *next* $\psi \ xs$

using *assms* **by** *auto*

declare *ev.intros*[*intro*]

declare *alw.cases*[*elim*]

lemma *ev-induct-strong*[*consumes 1, case-names base step*]:

$ev \ \varphi \ x \Longrightarrow (\bigwedge xs. \varphi \ xs \Longrightarrow P \ xs) \Longrightarrow (\bigwedge xs. ev \ \varphi \ (stl \ xs) \Longrightarrow \neg \varphi \ xs \Longrightarrow P \ (stl \ xs) \Longrightarrow P \ xs) \Longrightarrow P \ x$

by (*induct rule: ev.induct*) *auto*

lemma *alw-coinduct*[*consumes 1, case-names alw stl*]:

$X \ x \Longrightarrow (\bigwedge x. X \ x \Longrightarrow \varphi \ x) \Longrightarrow (\bigwedge x. X \ x \Longrightarrow \neg alw \ \varphi \ (stl \ x) \Longrightarrow X \ (stl \ x)) \Longrightarrow alw \ \varphi \ x$

using *alw.coinduct*[*of X x φ*] **by** *auto*

lemma *ev-mono*:

assumes *ev*: *ev* $\varphi \ xs$ **and** 0 : $\bigwedge xs. \varphi \ xs \Longrightarrow \psi \ xs$

shows *ev* $\psi \ xs$

using *ev* **by** *induct (auto simp: 0)*

lemma *alw-mono*:

assumes *alw*: *alw* $\varphi \ xs$ **and** 0 : $\bigwedge xs. \varphi \ xs \Longrightarrow \psi \ xs$

shows *alw* $\psi \ xs$

using *alw* **by** *coinduct (auto simp: 0)*

lemma *until-monoL*:

assumes *until*: $(\varphi 1 \text{ until } \psi) \text{ xs}$ **and** 0 : $\bigwedge xs. \varphi 1 \ xs \Longrightarrow \varphi 2 \ xs$

shows $(\varphi 2 \text{ until } \psi) \text{ xs}$

using *until* **by** *coinduct* (*auto elim: UNTIL.cases simp: 0*)

lemma *until-monoR*:

assumes *until*: $(\varphi \text{ until } \psi 1) \text{ } xs \text{ and } 0: \bigwedge xs. \psi 1 \text{ } xs \implies \psi 2 \text{ } xs$

shows $(\varphi \text{ until } \psi 2) \text{ } xs$

using *until* **by** *coinduct* (*auto elim: UNTIL.cases simp: 0*)

lemma *until-mono*:

assumes *until*: $(\varphi 1 \text{ until } \psi 1) \text{ } xs \text{ and}$

$0: \bigwedge xs. \varphi 1 \text{ } xs \implies \varphi 2 \text{ } xs \bigwedge xs. \psi 1 \text{ } xs \implies \psi 2 \text{ } xs$

shows $(\varphi 2 \text{ until } \psi 2) \text{ } xs$

using *until* **by** *coinduct* (*auto elim: UNTIL.cases simp: 0*)

lemma *until-false*: $\varphi \text{ until false} = \text{alw } \varphi$

proof–

{**fix** *xs* **assume** $(\varphi \text{ until false}) \text{ } xs$ **hence** $\text{alw } \varphi \text{ } xs$

by *coinduct* (*auto elim: UNTIL.cases*)

}

moreover

{**fix** *xs* **assume** $\text{alw } \varphi \text{ } xs$ **hence** $(\varphi \text{ until false}) \text{ } xs$

by *coinduct auto*

}

ultimately show *?thesis* **by** *blast*

qed

lemma *ev-nxt*: $\text{ev } \varphi = (\varphi \text{ or } \text{nxt } (\text{ev } \varphi))$

by (*rule ext*) (*metis ev.simps nxt.simps*)

lemma *alw-nxt*: $\text{alw } \varphi = (\varphi \text{ aand } \text{nxt } (\text{alw } \varphi))$

by (*rule ext*) (*metis alw.simps nxt.simps*)

lemma *ev-ev[simp]*: $\text{ev } (\text{ev } \varphi) = \text{ev } \varphi$

proof–

{**fix** *xs*

assume $\text{ev } (\text{ev } \varphi) \text{ } xs$ **hence** $\text{ev } \varphi \text{ } xs$

by *induct auto*

}

thus *?thesis* **by** *auto*

qed

lemma *alw-alw[simp]*: $\text{alw } (\text{alw } \varphi) = \text{alw } \varphi$

proof–

{**fix** *xs*

assume $\text{alw } \varphi \text{ } xs$ **hence** $\text{alw } (\text{alw } \varphi) \text{ } xs$

by *coinduct auto*

}

thus *?thesis* **by** *auto*

qed

```

lemma ev-shift:
assumes ev  $\varphi$  xs
shows ev  $\varphi$  (xl @- xs)
using assms by (induct xl) auto

lemma ev-imp-shift:
assumes ev  $\varphi$  xs shows  $\exists$  xl xs2. xs = xl @- xs2  $\wedge$   $\varphi$  xs2
using assms by induct (metis shift.simps(1), metis shift.simps(2) stream.collapse) +

lemma alw-ev-shift: alw  $\varphi$  xs1  $\implies$  ev (alw  $\varphi$ ) (xl @- xs1)
by (auto intro: ev-shift)

lemma alw-shift:
assumes alw  $\varphi$  (xl @- xs)
shows alw  $\varphi$  xs
using assms by (induct xl) auto

lemma ev-ex-nxt:
assumes ev  $\varphi$  xs
shows  $\exists$  n. (nxt  $\rightsquigarrow$  n)  $\varphi$  xs
using assms proof induct
  case (base xs) thus ?case by (intro exI[of - 0]) auto
next
  case (step xs)
  then obtain n where (nxt  $\rightsquigarrow$  n)  $\varphi$  (stl xs) by blast
  thus ?case by (intro exI[of - Suc n]) (metis funpow.simps(2) nxt.simps o-def)
qed

lemma alw-sdrop:
assumes alw  $\varphi$  xs shows alw  $\varphi$  (sdrop n xs)
by (metis alw-shift assms stake-sdrop)

lemma nxt-sdrop: (nxt  $\rightsquigarrow$  n)  $\varphi$  xs  $\longleftrightarrow$   $\varphi$  (sdrop n xs)
by (induct n arbitrary: xs) auto

definition wait  $\varphi$  xs  $\equiv$  LEAST n. (nxt  $\rightsquigarrow$  n)  $\varphi$  xs

lemma nxt-wait:
assumes ev  $\varphi$  xs shows (nxt  $\rightsquigarrow$  (wait  $\varphi$  xs))  $\varphi$  xs
unfolding wait-def using ev-ex-nxt[OF assms] by (rule LeastI-ex)

lemma nxt-wait-least:
assumes ev: ev  $\varphi$  xs and nxt: (nxt  $\rightsquigarrow$  n)  $\varphi$  xs shows wait  $\varphi$  xs  $\leq$  n
unfolding wait-def using ev-ex-nxt[OF ev] by (metis Least-le nxt)

lemma sdrop-wait:
assumes ev  $\varphi$  xs shows  $\varphi$  (sdrop (wait  $\varphi$  xs) xs)
using nxt-wait[OF assms] unfolding nxt-sdrop .

```


lemma *sdrop-wait-least*:

assumes *ev*: $ev\ \varphi\ xs$ **and** *nxt*: $\varphi\ (sdrop\ n\ xs)$ **shows** $wait\ \varphi\ xs \leq n$
using *assms* *nxt-wait-least* **unfolding** *nxt-sdrop* **by** *auto*

lemma *nxt-ev*: $(nxt \rightsquigarrow n)\ \varphi\ xs \implies ev\ \varphi\ xs$

by (*induct* *n* *arbitrary*: *xs*) *auto*

lemma *not-ev*: $not\ (ev\ \varphi) = alw\ (not\ \varphi)$

proof(*rule* *ext*, *safe*)

fix *xs* **assume** $not\ (ev\ \varphi)\ xs$ **thus** $alw\ (not\ \varphi)\ xs$

by (*coinduct*) *auto*

next

fix *xs* **assume** $ev\ \varphi\ xs$ **and** $alw\ (not\ \varphi)\ xs$ **thus** *False*

by (*induct*) *auto*

qed

lemma *not-alw*: $not\ (alw\ \varphi) = ev\ (not\ \varphi)$

proof–

have $not\ (alw\ \varphi) = not\ (alw\ (not\ (not\ \varphi)))$ **by** *simp*

also have $\dots = ev\ (not\ \varphi)$ **unfolding** *not-ev[symmetric]* **by** *simp*

finally show *?thesis* .

qed

lemma *not-ev-not[simp]*: $not\ (ev\ (not\ \varphi)) = alw\ \varphi$

unfolding *not-ev* **by** *simp*

lemma *not-alw-not[simp]*: $not\ (alw\ (not\ \varphi)) = ev\ \varphi$

unfolding *not-alw* **by** *simp*

lemma *alw-ev-sdrop*:

assumes $alw\ (ev\ \varphi)\ (sdrop\ m\ xs)$

shows $alw\ (ev\ \varphi)\ xs$

using *assms*

by *coinduct* (*metis* *alw-nxt* *ev-shift* *funpow-swap1* *nxt.simps* *nxt-sdrop* *stake-sdrop*)

lemma *ev-alw-imp-alw-ev*:

assumes $ev\ (alw\ \varphi)\ xs$ **shows** $alw\ (ev\ \varphi)\ xs$

using *assms* **by** *induct* (*metis* (*full-types*) *alw-mono* *ev.base*, *metis* *alw* *alw-nxt* *ev.step*)

lemma *alw-aand*: $alw\ (\varphi\ aand\ \psi) = alw\ \varphi\ aand\ alw\ \psi$

proof–

{fix *xs* **assume** $alw\ (\varphi\ aand\ \psi)\ xs$ **hence** $alw\ \varphi\ aand\ alw\ \psi)\ xs$

by (*auto* *elim*: *alw-mono*)

}

moreover

{fix *xs* **assume** $alw\ \varphi\ aand\ alw\ \psi)\ xs$ **hence** $alw\ (\varphi\ aand\ \psi)\ xs$

by *coinduct* *auto*

}

ultimately show *?thesis* by blast
qed

lemma *ev-or*: $ev (\varphi \text{ or } \psi) = ev \varphi \text{ or } ev \psi$
proof –
 {fix *xs* assume $(ev \varphi \text{ or } ev \psi) \text{ xs}$ hence $ev (\varphi \text{ or } \psi) \text{ xs}$
 by (auto elim: *ev-mono*)
 }
 moreover
 {fix *xs* assume $ev (\varphi \text{ or } \psi) \text{ xs}$ hence $(ev \varphi \text{ or } ev \psi) \text{ xs}$
 by induct auto
 }
 ultimately show *?thesis* by blast
qed

lemma *ev-alw-aand*:
assumes $\varphi: ev (alw \varphi) \text{ xs}$ and $\psi: ev (alw \psi) \text{ xs}$
shows $ev (alw (\varphi \text{ aand } \psi)) \text{ xs}$
proof –
 obtain *xl xs1* where $xs1: xs = xl @- xs1$ and $\varphi\varphi: alw \varphi \text{ xs1}$
 using φ by (metis *ev-imp-shift*)
 moreover obtain *yl ys1* where $xs2: xs = yl @- ys1$ and $\psi\psi: alw \psi \text{ ys1}$
 using ψ by (metis *ev-imp-shift*)
 ultimately have $0: xl @- xs1 = yl @- ys1$ by auto
 hence $prefix \text{ xl yl } \vee prefix \text{ yl xl}$ using *shift-prefix-cases* by auto
 thus *?thesis* proof
 assume $prefix \text{ xl yl}$
 then obtain *yl1* where $yl: yl = xl @ yl1$ by (elim *prefixE*)
 have $xs1': xs1 = yl1 @- ys1$ using 0 unfolding *yl* by simp
 have $alw \varphi \text{ ys1}$ using $\varphi\varphi$ unfolding $xs1'$ by (metis *alw-shift*)
 hence $alw (\varphi \text{ aand } \psi) \text{ ys1}$ using $\psi\psi$ unfolding *alw-aand* by auto
 thus *?thesis* unfolding $xs2$ by (auto intro: *alw-ev-shift*)
 next
 assume $prefix \text{ yl xl}$
 then obtain *xl1* where $xl: xl = yl @ xl1$ by (elim *prefixE*)
 have $ys1': ys1 = xl1 @- xs1$ using 0 unfolding *xl* by simp
 have $alw \psi \text{ xs1}$ using $\psi\psi$ unfolding $ys1'$ by (metis *alw-shift*)
 hence $alw (\varphi \text{ aand } \psi) \text{ xs1}$ using $\varphi\varphi$ unfolding *alw-aand* by auto
 thus *?thesis* unfolding $xs1$ by (auto intro: *alw-ev-shift*)
qed
qed

lemma *ev-alw-alw-impl*:
assumes $ev (alw \varphi) \text{ xs}$ and $alw (alw \varphi \text{ impl } ev \psi) \text{ xs}$
shows $ev \psi \text{ xs}$
using *assms* by induct auto

lemma *ev-alw-stl[simp]*: $ev (alw \varphi) (stl \text{ x}) \longleftrightarrow ev (alw \varphi) \text{ x}$
by (metis (full-types) *alw-nxt ev-nxt nxt.simps*)

lemma *alw-alw-impl-ev*:

alw (alw φ impl ev ψ) = (ev (alw φ) impl alw (ev ψ)) (is ?A = ?B)

proof–

{**fix** *xs* **assume** ?A *xs* \wedge ev (alw φ) *xs* **hence** alw (ev ψ) *xs*
by *coinduct (auto elim: ev-alw-alw-impl)*

}

moreover

{**fix** *xs* **assume** ?B *xs* **hence** ?A *xs*
by *coinduct auto*

}

ultimately show ?thesis **by** *blast*

qed

lemma *ev-alw-impl*:

assumes ev φ *xs* **and** alw (φ impl ψ) *xs* **shows** ev ψ *xs*

using *assms* **by** *induct auto*

lemma *ev-alw-impl-ev*:

assumes ev φ *xs* **and** alw (φ impl ev ψ) *xs* **shows** ev ψ *xs*

using *ev-alw-impl[OF assms]* **by** *simp*

lemma *alw-mp*:

assumes alw φ *xs* **and** alw (φ impl ψ) *xs*

shows alw ψ *xs*

proof–

{**assume** alw φ *xs* \wedge alw (φ impl ψ) *xs* **hence** ?thesis
by *coinduct auto*

}

thus ?thesis **using** *assms* **by** *auto*

qed

lemma *all-imp-alw*:

assumes \bigwedge *xs*. φ *xs* **shows** alw φ *xs*

proof–

{**assume** \forall *xs*. φ *xs*
hence ?thesis **by** *coinduct auto*

}

thus ?thesis **using** *assms* **by** *auto*

qed

lemma *alw-impl-ev-alw*:

assumes alw (φ impl ev ψ) *xs*

shows alw (ev φ impl ev ψ) *xs*

using *assms* **by** *coinduct (auto dest: ev-alw-impl)*

lemma *ev-holds-sset*:

ev (holds *P*) *xs* \longleftrightarrow (\exists *x* \in sset *xs*. *P* *x*) (is ?L \longleftrightarrow ?R)

proof *safe*

```

  assume ?L thus ?R by induct (metis holds.simps stream.set-sel(1), metis stl-sset)
next
  fix x assume x ∈ sset xs P x
  thus ?L by (induct rule: sset-induct) (simp-all add: ev.base ev.step)
qed

```

LTL as a program logic:

```

lemma alw-invar:
assumes  $\varphi$  xs and alw ( $\varphi$  impl nxt  $\varphi$ ) xs
shows alw  $\varphi$  xs
proof –
  {assume  $\varphi$  xs ∧ alw ( $\varphi$  impl nxt  $\varphi$ ) xs hence ?thesis
   by coinduct auto}
  thus ?thesis using assms by auto
qed

```

```

lemma variance:
assumes 1:  $\varphi$  xs and 2: alw ( $\varphi$  impl ( $\psi$  or nxt  $\varphi$ )) xs
shows (alw  $\varphi$  or ev  $\psi$ ) xs
proof –
  {assume  $\neg$  ev  $\psi$  xs hence alw (not  $\psi$ ) xs unfolding not-ev[symmetric] .
   moreover have alw (not  $\psi$  impl ( $\varphi$  impl nxt  $\varphi$ )) xs
   using 2 by coinduct auto
   ultimately have alw ( $\varphi$  impl nxt  $\varphi$ ) xs by (auto dest: alw-mp)
   with 1 have alw  $\varphi$  xs by (rule alw-invar)}
  thus ?thesis by blast
qed

```

```

lemma ev-alw-imp-nxt:
assumes e: ev  $\varphi$  xs and a: alw ( $\varphi$  impl (nxt  $\varphi$ )) xs
shows ev (alw  $\varphi$ ) xs
proof –
  obtain xl xs1 where xs: xs = xl @- xs1 and  $\varphi$ :  $\varphi$  xs1
  using e by (metis ev-imp-shift)
  have  $\varphi$  xs1 ∧ alw ( $\varphi$  impl (nxt  $\varphi$ )) xs1 using a  $\varphi$  unfolding xs by (metis
alw-shift)
  hence alw  $\varphi$  xs1 by (coinduct xs1 rule: alw.coinduct) auto
  thus ?thesis unfolding xs by (auto intro: alw-ev-shift)
qed

```

```

inductive ev-at :: ('a stream  $\Rightarrow$  bool)  $\Rightarrow$  nat  $\Rightarrow$  'a stream  $\Rightarrow$  bool for P :: 'a stream
 $\Rightarrow$  bool where
  base: P  $\omega \Rightarrow$  ev-at P 0  $\omega$ 
| step:  $\neg$  P  $\omega \Rightarrow$  ev-at P n (stl  $\omega \Rightarrow$  ev-at P (Suc n)  $\omega$ )

```

```

inductive-simps ev-at-0[simp]: ev-at P 0  $\omega$ 

```

inductive-simps *ev-at-Suc[simp]*: *ev-at P (Suc n) ω*

lemma *ev-at-imp-snth*: *ev-at P n ω \implies P (sdrop n ω)*
by (*induction n arbitrary: ω*) *auto*

lemma *ev-at-HLD-imp-snth*: *ev-at (HLD X) n ω \implies ω !! n \in X*
by (*auto dest!: ev-at-imp-snth simp: HLD-iff*)

lemma *ev-at-HLD-single-imp-snth*: *ev-at (HLD {x}) n ω \implies ω !! n = x*
by (*drule ev-at-HLD-imp-snth*) *simp*

lemma *ev-at-unique*: *ev-at P n ω \implies ev-at P m ω \implies n = m*

proof (*induction arbitrary: m rule: ev-at.induct*)

case (*base ω*) **then show** *?case*

by (*simp add: ev-at.simps[of - - ω]*)

next

case (*step ω n*) **from** *step.premis step.hyps step.IH[of m - 1]* **show** *?case*

by (*auto simp add: ev-at.simps[of - - ω]*)

qed

lemma *ev-iff-ev-at*: *ev P ω \longleftrightarrow ($\exists n. \text{ev-at } P \ n \ \omega$)*

proof

assume *ev P ω* **then show** $\exists n. \text{ev-at } P \ n \ \omega$

by (*induction rule: ev-induct-strong*) (*auto intro: ev-at.intros*)

next

assume $\exists n. \text{ev-at } P \ n \ \omega$

then obtain *n* **where** *ev-at P n ω*

by *auto*

then show *ev P ω*

by *induction auto*

qed

lemma *ev-at-shift*: *ev-at (HLD X) i (stake (Suc i) ω @- ω' :: 's stream) \longleftrightarrow ev-at (HLD X) i ω*

by (*induction i arbitrary: ω*) (*auto simp: HLD-iff*)

lemma *ev-iff-ev-at-unique*: *ev P ω \longleftrightarrow ($\exists! n. \text{ev-at } P \ n \ \omega$)*

by (*auto intro: ev-at-unique simp: ev-iff-ev-at*)

lemma *alw-HLD-iff-streams*: *alw (HLD X) ω \longleftrightarrow ω \in streams X*

proof

assume *alw (HLD X) ω* **then show** *ω \in streams X*

proof (*coinduction arbitrary: ω*)

case (*streams ω*) **then show** *?case* **by** (*cases ω*) *auto*

qed

next

assume *ω \in streams X* **then show** *alw (HLD X) ω*

proof (*coinduction arbitrary: ω*)

case (*alw ω*) **then show** *?case* **by** (*cases ω*) *auto*

qed
qed

lemma *not-HLD*: $\text{not } (\text{HLD } X) = \text{HLD } (\neg X)$
by (*auto simp: HLD-iff*)

lemma *not-alw-iff*: $\neg (\text{alw } P \ \omega) \longleftrightarrow \text{ev } (\text{not } P) \ \omega$
using *not-alw[of P]* by (*simp add: fun-eq-iff*)

lemma *not-ev-iff*: $\neg (\text{ev } P \ \omega) \longleftrightarrow \text{alw } (\text{not } P) \ \omega$
using *not-alw-iff[of not P ω, symmetric]* by *simp*

lemma *ev-Stream*: $\text{ev } P \ (x \ \#\# \ s) \longleftrightarrow P \ (x \ \#\# \ s) \vee \text{ev } P \ s$
by (*auto elim: ev.cases*)

lemma *alw-ev-imp-ev-alw*:
assumes *alw (ev P) ω* **shows** *ev (P a and alw (ev P)) ω*
proof –
have *ev P ω* **using** *assms* **by** *auto*
from *this assms* **show** *?thesis*
by *induct auto*
qed

lemma *ev-False*: $\text{ev } (\lambda x. \text{False}) \ \omega \longleftrightarrow \text{False}$
proof
assume *ev (λx. False) ω* **then show** *False*
by *induct auto*
qed *auto*

lemma *alw-False*: $\text{alw } (\lambda x. \text{False}) \ \omega \longleftrightarrow \text{False}$
by *auto*

lemma *ev-iff-sdrop*: $\text{ev } P \ \omega \longleftrightarrow (\exists m. P \ (\text{sdrop } m \ \omega))$
proof *safe*
assume *ev P ω* **then show** $\exists m. P \ (\text{sdrop } m \ \omega)$
by (*induct rule: ev-induct-strong*) (*auto intro: exI[of - 0] exI[of - Suc n for n]*)
next
fix *m* **assume** *P (sdrop m ω)* **then show** *ev P ω*
by (*induct m arbitrary: ω*) *auto*
qed

lemma *alw-iff-sdrop*: $\text{alw } P \ \omega \longleftrightarrow (\forall m. P \ (\text{sdrop } m \ \omega))$
proof *safe*
fix *m* **assume** *alw P ω* **then show** *P (sdrop m ω)*
by (*induct m arbitrary: ω*) *auto*
next
assume $\forall m. P \ (\text{sdrop } m \ \omega)$ **then show** *alw P ω*
by (*coinduction arbitrary: ω*) (*auto elim: allE[of - 0] allE[of - Suc n for n]*)
qed

lemma *infinite-iff-alw-ev*: $\text{infinite } \{m. P (\text{sdrop } m \ \omega)\} \longleftrightarrow \text{alw } (\text{ev } P) \ \omega$
unfolding *infinite-nat-iff-unbounded-le alw-iff-sdrop ev-iff-sdrop*
by *simp (metis le-Suc-ex le-add1)*

lemma *alw-inv*:
assumes *stl*: $\bigwedge s. f (\text{stl } s) = \text{stl } (f s)$
shows $\text{alw } P (f s) \longleftrightarrow \text{alw } (\lambda x. P (f x)) s$
proof
assume $\text{alw } P (f s)$ **then show** $\text{alw } (\lambda x. P (f x)) s$
by (*coinduction arbitrary: s rule: alw-coinduct*) (*auto simp: stl*)
next
assume $\text{alw } (\lambda x. P (f x)) s$ **then show** $\text{alw } P (f s)$
by (*coinduction arbitrary: s rule: alw-coinduct*) (*auto simp flip: stl*)
qed

lemma *ev-inv*:
assumes *stl*: $\bigwedge s. f (\text{stl } s) = \text{stl } (f s)$
shows $\text{ev } P (f s) \longleftrightarrow \text{ev } (\lambda x. P (f x)) s$
proof
assume $\text{ev } P (f s)$ **then show** $\text{ev } (\lambda x. P (f x)) s$
by (*induction f s arbitrary: s*) (*auto simp: stl*)
next
assume $\text{ev } (\lambda x. P (f x)) s$ **then show** $\text{ev } P (f s)$
by *induction* (*auto simp flip: stl*)
qed

lemma *alw-smap*: $\text{alw } P (\text{smap } f s) \longleftrightarrow \text{alw } (\lambda x. P (\text{smap } f x)) s$
by (*rule alw-inv*) *simp*

lemma *ev-smap*: $\text{ev } P (\text{smap } f s) \longleftrightarrow \text{ev } (\lambda x. P (\text{smap } f x)) s$
by (*rule ev-inv*) *simp*

lemma *alw-cong*:
assumes *P*: $\text{alw } P \ \omega$ **and** *eq*: $\bigwedge \omega. P \ \omega \implies Q1 \ \omega \longleftrightarrow Q2 \ \omega$
shows $\text{alw } Q1 \ \omega \longleftrightarrow \text{alw } Q2 \ \omega$
proof –
from *eq* **have** $(\text{alw } P \ \text{aand } Q1) = (\text{alw } P \ \text{aand } Q2)$ **by** *auto*
then have $\text{alw } (\text{alw } P \ \text{aand } Q1) \ \omega = \text{alw } (\text{alw } P \ \text{aand } Q2) \ \omega$ **by** *auto*
with *P* **show** $\text{alw } Q1 \ \omega \longleftrightarrow \text{alw } Q2 \ \omega$
by (*simp add: alw-aand*)
qed

lemma *ev-cong*:
assumes *P*: $\text{alw } P \ \omega$ **and** *eq*: $\bigwedge \omega. P \ \omega \implies Q1 \ \omega \longleftrightarrow Q2 \ \omega$
shows $\text{ev } Q1 \ \omega \longleftrightarrow \text{ev } Q2 \ \omega$
proof –
from *P* **have** $\text{alw } (\lambda xs. Q1 \ xs \longrightarrow Q2 \ xs) \ \omega$ **by** (*rule alw-mono*) (*simp add: eq*)

moreover from P **have** $alw (\lambda xs. Q2\ xs \longrightarrow Q1\ xs) \ \omega$ **by** (*rule alw-mono*) (*simp add: eq*)
moreover note $ev\text{-}alw\text{-}impl[of\ Q1\ \omega\ Q2]\ ev\text{-}alw\text{-}impl[of\ Q2\ \omega\ Q1]$
ultimately show $ev\ Q1\ \omega \longleftrightarrow ev\ Q2\ \omega$
by *auto*
qed

lemma *alwD*: $alw\ P\ x \Longrightarrow P\ x$
by *auto*

lemma *alw-alwD*: $alw\ P\ \omega \Longrightarrow alw\ (alw\ P)\ \omega$
by *simp*

lemma *alw-ev-stl*: $alw\ (ev\ P)\ (stl\ \omega) \longleftrightarrow alw\ (ev\ P)\ \omega$
by (*auto intro: alw.intros*)

lemma *holds-Stream*: $holds\ P\ (x\ \#\#\ s) \longleftrightarrow P\ x$
by *simp*

lemma *holds-eq1[simp]*: $holds\ ((=)\ x) = HLD\ \{x\}$
by *rule (auto simp: HLD-iff)*

lemma *holds-eq2[simp]*: $holds\ (\lambda y. y = x) = HLD\ \{x\}$
by *rule (auto simp: HLD-iff)*

lemma *not-holds-eq[simp]*: $holds\ (-\ (=)\ x) = not\ (HLD\ \{x\})$
by *rule (auto simp: HLD-iff)*

Strong until

context
notes *[[inductive-internals]]*
begin

inductive *suntil* (**infix** $\langle suntil \rangle\ 60$) **for** $\varphi\ \psi$ **where**
base: $\varphi\ \omega \Longrightarrow (\varphi\ suntil\ \psi)\ \omega$
step: $\varphi\ \omega \Longrightarrow (\varphi\ suntil\ \psi)\ (stl\ \omega) \Longrightarrow (\varphi\ suntil\ \psi)\ \omega$

inductive-simps *suntil-Stream*: $(\varphi\ suntil\ \psi)\ (x\ \#\#\ s)$

end

lemma *suntil-induct-strong[consumes 1, case-names base step]*:
 $(\varphi\ suntil\ \psi)\ x \Longrightarrow$
 $(\bigwedge \omega. \psi\ \omega \Longrightarrow P\ \omega) \Longrightarrow$
 $(\bigwedge \omega. \varphi\ \omega \Longrightarrow \neg \psi\ \omega \Longrightarrow (\varphi\ suntil\ \psi)\ (stl\ \omega) \Longrightarrow P\ (stl\ \omega) \Longrightarrow P\ \omega) \Longrightarrow P\ x$
using *suntil.induct[of $\varphi\ \psi\ x\ P$]* **by** *blast*

lemma *ev-suntil*: $(\varphi\ suntil\ \psi)\ \omega \Longrightarrow ev\ \psi\ \omega$
by (*induct rule: suntil.induct*) *auto*

lemma *suntil-inv*:

assumes *stl*: $\bigwedge s. f (stl\ s) = stl\ (f\ s)$

shows $(P\ suntil\ Q)\ (f\ s) \longleftrightarrow ((\lambda x. P\ (f\ x))\ suntil\ (\lambda x. Q\ (f\ x)))\ s$

proof

assume $(P\ suntil\ Q)\ (f\ s)$ **then show** $((\lambda x. P\ (f\ x))\ suntil\ (\lambda x. Q\ (f\ x)))\ s$

by (*induction* *f s arbitrary: s*) (*auto simp: stl intro: suntil.intros*)

next

assume $((\lambda x. P\ (f\ x))\ suntil\ (\lambda x. Q\ (f\ x)))\ s$ **then show** $(P\ suntil\ Q)\ (f\ s)$

by *induction* (*auto simp flip: stl intro: suntil.intros*)

qed

lemma *suntil-smap*: $(P\ suntil\ Q)\ (smap\ f\ s) \longleftrightarrow ((\lambda x. P\ (smap\ f\ x))\ suntil\ (\lambda x. Q\ (smap\ f\ x)))\ s$

by (*rule suntil-inv*) *simp*

lemma *hld-smap*: $HLD\ x\ (smap\ f\ s) = holds\ (\lambda y. f\ y \in x)\ s$

by (*simp add: HLD-def*)

lemma *suntil-mono*:

assumes *eq*: $\bigwedge \omega. P\ \omega \implies Q1\ \omega \implies Q2\ \omega \bigwedge \omega. P\ \omega \implies R1\ \omega \implies R2\ \omega$

assumes *: $(Q1\ suntil\ R1)\ \omega \text{ alw } P\ \omega$ **shows** $(Q2\ suntil\ R2)\ \omega$

using * **by** *induct* (*auto intro: eq suntil.intros*)

lemma *suntil-cong*:

$\text{alw } P\ \omega \implies (\bigwedge \omega. P\ \omega \implies Q1\ \omega \longleftrightarrow Q2\ \omega) \implies (\bigwedge \omega. P\ \omega \implies R1\ \omega \longleftrightarrow R2\ \omega) \implies$

$(Q1\ suntil\ R1)\ \omega \longleftrightarrow (Q2\ suntil\ R2)\ \omega$

using *suntil-mono*[*of P Q1 Q2 R1 R2 ω*] *suntil-mono*[*of P Q2 Q1 R2 R1 ω*] **by** *auto*

lemma *ev-suntil-iff*: $ev\ (P\ suntil\ Q)\ \omega \longleftrightarrow ev\ Q\ \omega$

proof

assume $ev\ (P\ suntil\ Q)\ \omega$ **then show** $ev\ Q\ \omega$

by *induct* (*auto dest: ev-suntil*)

next

assume $ev\ Q\ \omega$ **then show** $ev\ (P\ suntil\ Q)\ \omega$

by *induct* (*auto intro: suntil.intros*)

qed

lemma *true-suntil*: $((\lambda -. True)\ suntil\ P) = ev\ P$

by (*simp add: suntil-def ev-def*)

lemma *suntil-lfp*: $(\varphi\ suntil\ \psi) = lfp\ (\lambda P\ s. \psi\ s \vee (\varphi\ s \wedge P\ (stl\ s)))$

by (*simp add: suntil-def*)

lemma *sfilter-P*[*simp*]: $P\ (shd\ s) \implies sfilter\ P\ s = shd\ s\ \#\#\ sfilter\ P\ (stl\ s)$

using *sfilter-Stream*[*of P shd s stl s*] **by** *simp*

lemma *sfilter-not-P[simp]*: $\neg P \text{ (shd } s) \implies \text{sfilter } P \text{ } s = \text{sfilter } P \text{ (stl } s)$
using *sfilter-Stream[of P shd s stl s]* **by** *simp*

lemma *sfilter-eq*:
assumes *ev (holds P) s*
shows $\text{sfilter } P \text{ } s = x \text{ \#\# } s' \longleftrightarrow P \text{ } x \wedge (\text{not (holds } P) \text{ until (HLD \{x\} a and next (\lambda s. \text{sfilter } P \text{ } s = s'))}) s$
using *assms*
by (*induct rule: ev-induct-strong*)
(auto simp add: HLD-iff intro: until.intros elim: until.cases)

lemma *sfilter-streams*:
 $\text{alw (ev (holds } P)) \omega \implies \omega \in \text{streams } A \implies \text{sfilter } P \omega \in \text{streams } \{x \in A. P \text{ } x\}$
proof (*coinduction arbitrary: \omega*)
case (*streams \omega*)
then have *ev (holds P) \omega* **by** *blast*
from this streams show *?case*
by (*induct rule: ev-induct-strong*) (*auto elim: streamsE*)
qed

lemma *alw-sfilter*:
assumes **: alw (ev (holds P)) s*
shows $\text{alw } Q \text{ (sfilter } P \text{ } s) \longleftrightarrow \text{alw } (\lambda x. Q \text{ (sfilter } P \text{ } x)) s$
proof
assume *alw Q (sfilter P s)* **with *** **show** *alw (\lambda x. Q (sfilter P x)) s*
proof (*coinduction arbitrary: s rule: alw-coinduct*)
case (*stl s*)
then have *ev (holds P) s*
by *blast*
from this stl show *?case*
by (*induct rule: ev-induct-strong*) *auto*
qed *auto*
next
assume *alw (\lambda x. Q (sfilter P x)) s* **with *** **show** *alw Q (sfilter P s)*
proof (*coinduction arbitrary: s rule: alw-coinduct*)
case (*stl s*)
then have *ev (holds P) s*
by *blast*
from this stl show *?case*
by (*induct rule: ev-induct-strong*) *auto*
qed *auto*
qed

lemma *ev-sfilter*:
assumes **: alw (ev (holds P)) s*
shows $\text{ev } Q \text{ (sfilter } P \text{ } s) \longleftrightarrow \text{ev } (\lambda x. Q \text{ (sfilter } P \text{ } x)) s$
proof
assume *ev Q (sfilter P s)* **from this *** **show** *ev (\lambda x. Q (sfilter P x)) s*
proof (*induction sfilter P s arbitrary: s rule: ev-induct-strong*)

```

    case (step s)
    then have ev (holds P) s
      by blast
    from this step show ?case
      by (induct rule: ev-induct-strong) auto
  qed auto
next
  assume ev ( $\lambda x. Q$  (sfilter P x)) s then show ev Q (sfilter P s)
  proof (induction rule: ev-induct-strong)
    case (step s) then show ?case
      by (cases P (shd s)) auto
  qed auto
qed

lemma holds-sfilter:
  assumes ev (holds Q) s shows holds P (sfilter Q s)  $\longleftrightarrow$  (not (holds Q) until
(holds (Q aand P))) s
proof
  assume holds P (sfilter Q s) with assms show (not (holds Q) until (holds (Q
aand P))) s
    by (induct rule: ev-induct-strong) (auto intro: until.intros)
next
  assume (not (holds Q) until (holds (Q aand P))) s then show holds P (sfilter
Q s)
    by induct auto
qed

lemma until-aand-nxt:
  ( $\varphi$  until ( $\varphi$  aand nxt  $\psi$ ))  $\omega \longleftrightarrow$  ( $\varphi$  aand nxt ( $\varphi$  until  $\psi$ ))  $\omega$ 
proof
  assume ( $\varphi$  until ( $\varphi$  aand nxt  $\psi$ ))  $\omega$  then show ( $\varphi$  aand nxt ( $\varphi$  until  $\psi$ ))  $\omega$ 
    by induction (auto intro: until.intros)
next
  assume ( $\varphi$  aand nxt ( $\varphi$  until  $\psi$ ))  $\omega$ 
  then have ( $\varphi$  until  $\psi$ ) (stl  $\omega$ )  $\varphi$   $\omega$ 
    by auto
  then show ( $\varphi$  until ( $\varphi$  aand nxt  $\psi$ ))  $\omega$ 
    by (induction stl  $\omega$  arbitrary:  $\omega$ )
      (auto elim: until.cases intro: until.intros)
qed

lemma alw-sconst: alw P (sconst x)  $\longleftrightarrow$  P (sconst x)
proof
  assume P (sconst x) then show alw P (sconst x)
    by coinduction auto
qed auto

lemma ev-sconst: ev P (sconst x)  $\longleftrightarrow$  P (sconst x)
proof

```

```

assume  $ev\ P\ (sconst\ x)$  then show  $P\ (sconst\ x)$ 
  by (induction sconst x) auto
qed auto

```

```

lemma suntil-sconst:  $(\varphi\ suntil\ \psi)\ (sconst\ x) \longleftrightarrow \psi\ (sconst\ x)$ 
proof
  assume  $(\varphi\ suntil\ \psi)\ (sconst\ x)$  then show  $\psi\ (sconst\ x)$ 
    by (induction sconst x) auto
qed (auto intro: suntil.intros)

```

```

lemma hld-smap':  $HLD\ x\ (smap\ f\ s) = HLD\ (f\ -'x)\ s$ 
  by (simp add: HLD-def)

```

```

lemma pigeonhole-stream:
  assumes  $alw\ (HLD\ s)\ \omega$ 
  assumes finite s
  shows  $\exists x \in s. alw\ (ev\ (HLD\ \{x\}))\ \omega$ 
proof –
  have  $\forall i \in UNIV. \exists x \in s. \omega\ !!\ i = x$ 
    using  $\langle alw\ (HLD\ s)\ \omega \rangle$  by (simp add: alw-iff-sdrop HLD-iff)
  from pigeonhole-infinite-rel[OF infinite-UNIV-nat]  $\langle finite\ s \rangle$  this
  show ?thesis
    by (simp add: HLD-iff flip: infinite-iff-alw-ev)
qed

```

```

lemma ev-eq-suntil:  $ev\ P\ \omega \longleftrightarrow (not\ P\ suntil\ P)\ \omega$ 
proof
  assume  $ev\ P\ \omega$  then show  $((\lambda xs. \neg P\ xs)\ suntil\ P)\ \omega$ 
    by (induction rule: ev-induct-strong) (auto intro: suntil.intros)
qed (auto simp: ev-suntil)

```

61 Weak vs. strong until (contributed by Michael Foster, University of Sheffield)

```

lemma suntil-implies-until:  $(\varphi\ suntil\ \psi)\ \omega \implies (\varphi\ until\ \psi)\ \omega$ 
  by (induct rule: suntil-induct-strong) (auto intro: UNTIL.intros)

```

```

lemma alw-implies-until:  $alw\ \varphi\ \omega \implies (\varphi\ until\ \psi)\ \omega$ 
  unfolding until-false[symmetric] by (auto elim: until-mono)

```

```

lemma until-ev-suntil:  $(\varphi\ until\ \psi)\ \omega \implies ev\ \psi\ \omega \implies (\varphi\ suntil\ \psi)\ \omega$ 
proof (rotate-tac, induction rule: ev.induct)
  case (base xs)
  then show ?case
    by (simp add: suntil.base)
next
  case (step xs)
  then show ?case

```

by (metis UNTIL.cases suntil.base suntil.step)
qed

lemma suntil-as-until: $(\varphi \text{ suntil } \psi) \omega = ((\varphi \text{ until } \psi) \omega \wedge \text{ev } \psi \omega)$
using ev-suntil suntil-implies-until until-ev-suntil by blast

lemma until-not-released-now: $(\varphi \text{ until } \psi) \omega \implies \neg \psi \omega \implies \varphi \omega$
using UNTIL.cases by auto

lemma until-must-release-ev: $(\varphi \text{ until } \psi) \omega \implies \text{ev } (\text{not } \varphi) \omega \implies \text{ev } \psi \omega$

proof (rotate-tac, induction rule: ev.induct)

case (base xs)

then show ?case

using until-not-released-now by blast

next

case (step xs)

then show ?case

using UNTIL.cases by blast

qed

lemma until-as-suntil: $(\varphi \text{ until } \psi) \omega = ((\varphi \text{ suntil } \psi) \text{ or } (\text{alw } \varphi)) \omega$
using alw-implies-until not-alw-iff suntil-implies-until until-ev-suntil until-must-release-ev
by blast

lemma alw-holds: $\text{alw } (\text{holds } P) (h \# \# t) = (P \ h \wedge \text{alw } (\text{holds } P) \ t)$
by (metis alw.simps holds-Stream stream.sel(2))

lemma alw-holds2: $\text{alw } (\text{holds } P) \ ss = (P \ (\text{shd } ss) \wedge \text{alw } (\text{holds } P) \ (\text{stl } ss))$
by (meson alw.simps holds.elims(2) holds.elims(3))

lemma alw-eq-sconst: $(\text{alw } (\text{HLD } \{h\}) \ t) = (t = \text{sconst } h)$
unfolding sconst-alt alw-HLD-iff-streams streams-iff-sset
using stream.set-sel(1) by force

lemma sdrop-if-suntil: $(p \text{ suntil } q) \omega \implies \exists j. q \ (\text{sdrop } j \ \omega) \wedge (\forall k < j. p \ (\text{sdrop } k \ \omega))$

proof(induction rule: suntil.induct)

case (base ω)

then show ?case

by force

next

case (step ω)

then obtain j where $q \ (\text{sdrop } j \ (\text{stl } \omega)) \wedge \forall k < j. p \ (\text{sdrop } k \ (\text{stl } \omega))$ by blast

with step(1,2) show ?case

using ev-at-imp-snth less-Suc-eq-0-disj by (auto intro!: exI[where $x=j+1$])

qed

lemma not-suntil: $(\neg (p \text{ suntil } q) \omega) = (\neg (p \text{ until } q) \omega \vee \text{alw } (\text{not } q) \omega)$
by (simp add: suntil-as-until alw-iff-sdrop ev-iff-sdrop)

```

lemma sdrop-until:  $q \text{ (sdrop } j \text{ } \omega) \implies \forall k < j. p \text{ (sdrop } k \text{ } \omega) \implies (p \text{ until } q) \omega$ 
proof(induct j arbitrary:  $\omega$ )
  case 0
  then show ?case
    by (simp add: UNTIL.base)
next
  case (Suc j)
  then show ?case
    by (metis Suc-mono UNTIL.simps sdrop.simps(1) sdrop.simps(2) zero-less-Suc)
qed

lemma sdrop-suntil:  $q \text{ (sdrop } j \text{ } \omega) \implies (\forall k < j. p \text{ (sdrop } k \text{ } \omega)) \implies (p \text{ suntil } q) \omega$ 
  by (metis ev-iff-sdrop sdrop-until suntil-as-until)

lemma suntil-iff-sdrop:  $(p \text{ suntil } q) \omega = (\exists j. q \text{ (sdrop } j \text{ } \omega) \wedge (\forall k < j. p \text{ (sdrop } k \text{ } \omega)))$ 
  using sdrop-if-suntil sdrop-suntil by blast

end

```

62 Lists as vectors

```

theory ListVector
  imports Main
begin

```

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

```

abbreviation scale :: ('a::times)  $\Rightarrow$  'a list  $\Rightarrow$  'a list (infix  $\langle *_{\text{s}} \rangle$  70)
  where  $x *_{\text{s}} xs \equiv \text{map } ((*) \text{ } x) \text{ } xs$ 

```

```

lemma scaleI[simp]:  $(1::'a::monoid-mult) *_{\text{s}} xs = xs$ 
  by (induct xs simp-all)

```

62.1 + and −

```

fun zipwith0 :: ('a::zero  $\Rightarrow$  'b::zero  $\Rightarrow$  'c)  $\Rightarrow$  'a list  $\Rightarrow$  'b list  $\Rightarrow$  'c list
  where
    zipwith0  $f \text{ [] []} = []$  |
    zipwith0  $f (x \# xs) (y \# ys) = f \text{ } x \text{ } y \# \text{ } \text{zipwith0 } f \text{ } xs \text{ } ys$  |
    zipwith0  $f (x \# xs) [] = f \text{ } x \text{ } 0 \# \text{ } \text{zipwith0 } f \text{ } xs \text{ } []$  |
    zipwith0  $f [] (y \# ys) = f \text{ } 0 \text{ } y \# \text{ } \text{zipwith0 } f \text{ } [] \text{ } ys$ 

```

```

instantiation list :: ({zero, plus}) plus
begin

```

definition

list-add-def: $(+) = \text{zipwith0 } (+)$

instance ..

end

instantiation *list* :: $(\{zero, uminus\})$ *uminus*
begin

definition

list-uminus-def: $uminus = \text{map } uminus$

instance ..

end

instantiation *list* :: $(\{zero, minus\})$ *minus*
begin

definition

list-diff-def: $(-) = \text{zipwith0 } (-)$

instance ..

end

lemma *zipwith0-Nil[simp]*: $\text{zipwith0 } f \ [] \ ys = \text{map } (f \ 0) \ ys$
by (*induct ys*) *simp-all*

lemma *list-add-Nil[simp]*: $[] + xs = (xs::'a::\text{monoid-add list})$
by (*induct xs*) (*auto simp:list-add-def*)

lemma *list-add-Nil2[simp]*: $xs + [] = (xs::'a::\text{monoid-add list})$
by (*induct xs*) (*auto simp:list-add-def*)

lemma *list-add-Cons[simp]*: $(x\#xs) + (y\#ys) = (x+y)\#(xs+ys)$
by (*auto simp:list-add-def*)

lemma *list-diff-Nil[simp]*: $[] - xs = -(xs::'a::\text{group-add list})$
by (*induct xs*) (*auto simp:list-diff-def list-uminus-def*)

lemma *list-diff-Nil2[simp]*: $xs - [] = (xs::'a::\text{group-add list})$
by (*induct xs*) (*auto simp:list-diff-def*)

lemma *list-diff-Cons-Cons[simp]*: $(x\#xs) - (y\#ys) = (x-y)\#(xs-ys)$
by (*induct xs*) (*auto simp:list-diff-def*)

lemma *list-uminus-Cons[simp]*: $-(x\#xs) = (-x)\#(-xs)$

by (induct xs) (auto simp: list-uminus-def)

lemma self-list-diff:

$xs - xs = \text{replicate } (\text{length}(xs :: 'a::\text{group-add list})) \ 0$

by (induct xs) simp-all

lemma list-add-assoc:

fixes xs :: 'a::monoid-add list

shows $(xs + ys) + zs = xs + (ys + zs)$

proof (induct xs arbitrary: ys zs)

case Nil

then show ?case by simp

next

case (Cons a xs ys zs)

show ?case

by (cases ys; cases zs; simp add: add.assoc Cons)

qed

62.2 Inner product

definition iprod :: 'a::ring list \Rightarrow 'a list \Rightarrow 'a ($\langle \langle \text{open-block notation} = \langle \text{mixfix } \text{iprod} \rangle \rangle \langle -, - \rangle \rangle$)

where $\langle xs, ys \rangle = (\sum (x, y) \leftarrow \text{zip } xs \text{ } ys. x * y)$

lemma iprod-Nil[simp]: $\langle [], ys \rangle = 0$

by (simp add: iprod-def)

lemma iprod-Nil2[simp]: $\langle xs, [] \rangle = 0$

by (simp add: iprod-def)

lemma iprod-Cons[simp]: $\langle x \# xs, y \# ys \rangle = x * y + \langle xs, ys \rangle$

by (simp add: iprod-def)

lemma iprod0-if-coeffs0: $\forall c \in \text{set } cs. c = 0 \implies \langle cs, xs \rangle = 0$

proof (induct cs arbitrary: xs)

case Nil

then show ?case by simp

next

case (Cons a cs xs)

then show ?case

by (cases xs; fastforce)

qed

lemma iprod-uminus[simp]: $\langle -xs, ys \rangle = -\langle xs, ys \rangle$

by (simp add: iprod-def uminus-sum-list-map o-def split-def map-zip-map list-uminus-def)

lemma iprod-left-add-distrib: $\langle xs + ys, zs \rangle = \langle xs, zs \rangle + \langle ys, zs \rangle$

proof (induct xs arbitrary: ys zs)

case Nil


```

    then show ?case by simp
next
  case (Cons a xs ys zs)
  show ?case
    by (cases ys; cases zs; simp add: distrib-right Cons)
qed

lemma iprod-left-diff-distrib:  $\langle xs - ys, zs \rangle = \langle xs, zs \rangle - \langle ys, zs \rangle$ 
proof (induct xs arbitrary: ys zs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs ys zs)
  show ?case
    by (cases ys; cases zs; simp add: left-diff-distrib Cons)
qed

lemma iprod-assoc:  $\langle x *_s xs, ys \rangle = x * \langle xs, ys \rangle$ 
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case by simp
next
  case (Cons a xs ys)
  show ?case
    by (cases ys; simp add: distrib-left mult.assoc Cons)
qed

end

```

63 Definitions of Least Upper Bounds and Greatest Lower Bounds

```

theory Lub-Glb
imports Complex-Main
begin

```

Thanks to suggestions by James Margetson

```

definition settle :: 'a set  $\Rightarrow$  'a::ord  $\Rightarrow$  bool (infixl  $\langle * \leq \rangle$  70)
  where  $S * \leq x = (\forall y \in S. y \leq x)$ 

```

```

definition setge :: 'a::ord  $\Rightarrow$  'a set  $\Rightarrow$  bool (infixl  $\langle \leq * \rangle$  70)
  where  $x \leq * S = (\forall y \in S. x \leq y)$ 

```

63.1 Rules for the Relations $* \leq$ and $\leq *$

```

lemma settleI:  $\forall y \in S. y \leq x \implies S * \leq x$ 
  by (simp add: settle-def)

```

lemma *setleD*: $S * \leq x \implies y \in S \implies y \leq x$
by (*simp add: setle-def*)

lemma *setgeI*: $\forall y \in S. x \leq y \implies x \leq^* S$
by (*simp add: setge-def*)

lemma *setgeD*: $x \leq^* S \implies y \in S \implies x \leq y$
by (*simp add: setge-def*)

definition *leastP* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a::\text{ord} \Rightarrow \text{bool}$
where *leastP* *P* *x* = $(P\ x \wedge x \leq^* \text{Collect } P)$

definition *isUb* :: $'a\ \text{set} \Rightarrow 'a\ \text{set} \Rightarrow 'a::\text{ord} \Rightarrow \text{bool}$
where *isUb* *R* *S* *x* = $(S * \leq x \wedge x \in R)$

definition *isLub* :: $'a\ \text{set} \Rightarrow 'a\ \text{set} \Rightarrow 'a::\text{ord} \Rightarrow \text{bool}$
where *isLub* *R* *S* *x* = *leastP* (*isUb* *R* *S*) *x*

definition *ubs* :: $'a\ \text{set} \Rightarrow 'a::\text{ord}\ \text{set} \Rightarrow 'a\ \text{set}$
where *ubs* *R* *S* = *Collect* (*isUb* *R* *S*)

63.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: *leastP* *P* *x* $\implies P\ x$
by (*simp add: leastP-def*)

lemma *leastPD2*: *leastP* *P* *x* $\implies x \leq^* \text{Collect } P$
by (*simp add: leastP-def*)

lemma *leastPD3*: *leastP* *P* *x* $\implies y \in \text{Collect } P \implies x \leq y$
by (*blast dest!: leastPD2 setgeD*)

lemma *isLubD1*: *isLub* *R* *S* *x* $\implies S * \leq x$
by (*simp add: isLub-def isUb-def leastP-def*)

lemma *isLubD1a*: *isLub* *R* *S* *x* $\implies x \in R$
by (*simp add: isLub-def isUb-def leastP-def*)

lemma *isLub-isUb*: *isLub* *R* *S* *x* $\implies \text{isUb } R\ S\ x$
unfolding *isUb-def* **by** (*blast dest: isLubD1 isLubD1a*)

lemma *isLubD2*: *isLub* *R* *S* *x* $\implies y \in S \implies y \leq x$
by (*blast dest!: isLubD1 setleD*)

lemma *isLubD3*: *isLub* *R* *S* *x* $\implies \text{leastP } (\text{isUb } R\ S)\ x$
by (*simp add: isLub-def*)

lemma *isLubI1*: *leastP* (*isUb* *R* *S*) *x* $\implies \text{isLub } R\ S\ x$

by (simp add: isLub-def)

lemma isLubI2: $isUb\ R\ S\ x \implies x \leq^* Collect\ (isUb\ R\ S) \implies isLub\ R\ S\ x$
 by (simp add: isLub-def leastP-def)

lemma isUbD: $isUb\ R\ S\ x \implies y \in S \implies y \leq x$
 by (simp add: isUb-def settle-def)

lemma isUbD2: $isUb\ R\ S\ x \implies S \leq^* x$
 by (simp add: isUb-def)

lemma isUbD2a: $isUb\ R\ S\ x \implies x \in R$
 by (simp add: isUb-def)

lemma isUbI: $S \leq^* x \implies x \in R \implies isUb\ R\ S\ x$
 by (simp add: isUb-def)

lemma isLub-le-isUb: $isLub\ R\ S\ x \implies isUb\ R\ S\ y \implies x \leq y$
 unfolding isLub-def by (blast intro!: leastPD3)

lemma isLub-ubs: $isLub\ R\ S\ x \implies x \leq^* ubs\ R\ S$
 unfolding ubs-def isLub-def by (rule leastPD2)

lemma isLub-unique: $[| isLub\ R\ S\ x; isLub\ R\ S\ y |] \implies x = (y::'a::linorder)$
 apply (frule isLub-isUb)
 apply (frule-tac $x = y$ in isLub-isUb)
 apply (blast intro!: order-antisym dest!: isLub-le-isUb)
 done

lemma isUb-UNIV-I: $(\bigwedge y. y \in S \implies y \leq u) \implies isUb\ UNIV\ S\ u$
 by (simp add: isUbI settleI)

definition greatestP :: $('a \Rightarrow bool) \Rightarrow 'a::ord \Rightarrow bool$
 where greatestP $P\ x = (P\ x \wedge Collect\ P \leq^* x)$

definition isLb :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
 where isLb $R\ S\ x = (x \leq^* S \wedge x \in R)$

definition isGlb :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a::ord \Rightarrow bool$
 where isGlb $R\ S\ x = greatestP\ (isLb\ R\ S)\ x$

definition lbs :: $'a\ set \Rightarrow 'a::ord\ set \Rightarrow 'a\ set$
 where lbs $R\ S = Collect\ (isLb\ R\ S)$

63.3 Rules about the Operators *greatestP*, *isLb* and *isGlb*

lemma greatestPD1: $greatestP\ P\ x \implies P\ x$
 by (simp add: greatestP-def)

lemma *greatestPD2*: $\text{greatestP } P \ x \Longrightarrow \text{Collect } P \ * \leq x$
by (*simp add: greatestP-def*)

lemma *greatestPD3*: $\text{greatestP } P \ x \Longrightarrow y \in \text{Collect } P \Longrightarrow x \geq y$
by (*blast dest!: greatestPD2 settleD*)

lemma *isGlbD1*: $\text{isGlb } R \ S \ x \Longrightarrow x \leq^* S$
by (*simp add: isGlb-def isLb-def greatestP-def*)

lemma *isGlbD1a*: $\text{isGlb } R \ S \ x \Longrightarrow x \in R$
by (*simp add: isGlb-def isLb-def greatestP-def*)

lemma *isGlb-isLb*: $\text{isGlb } R \ S \ x \Longrightarrow \text{isLb } R \ S \ x$
unfolding *isLb-def* **by** (*blast dest: isGlbD1 isGlbD1a*)

lemma *isGlbD2*: $\text{isGlb } R \ S \ x \Longrightarrow y \in S \Longrightarrow y \geq x$
by (*blast dest!: isGlbD1 setgeD*)

lemma *isGlbD3*: $\text{isGlb } R \ S \ x \Longrightarrow \text{greatestP } (\text{isLb } R \ S) \ x$
by (*simp add: isGlb-def*)

lemma *isGlbI1*: $\text{greatestP } (\text{isLb } R \ S) \ x \Longrightarrow \text{isGlb } R \ S \ x$
by (*simp add: isGlb-def*)

lemma *isGlbI2*: $\text{isLb } R \ S \ x \Longrightarrow \text{Collect } (\text{isLb } R \ S) \ * \leq x \Longrightarrow \text{isGlb } R \ S \ x$
by (*simp add: isGlb-def greatestP-def*)

lemma *isLbD*: $\text{isLb } R \ S \ x \Longrightarrow y \in S \Longrightarrow y \geq x$
by (*simp add: isLb-def setge-def*)

lemma *isLbD2*: $\text{isLb } R \ S \ x \Longrightarrow x \leq^* S$
by (*simp add: isLb-def*)

lemma *isLbD2a*: $\text{isLb } R \ S \ x \Longrightarrow x \in R$
by (*simp add: isLb-def*)

lemma *isLbI*: $x \leq^* S \Longrightarrow x \in R \Longrightarrow \text{isLb } R \ S \ x$
by (*simp add: isLb-def*)

lemma *isGlb-le-isLb*: $\text{isGlb } R \ S \ x \Longrightarrow \text{isLb } R \ S \ y \Longrightarrow x \geq y$
unfolding *isGlb-def* **by** (*blast intro!: greatestPD3*)

lemma *isGlb-ubs*: $\text{isGlb } R \ S \ x \Longrightarrow \text{lbs } R \ S \ * \leq x$
unfolding *lbs-def isGlb-def* **by** (*rule greatestPD2*)

lemma *isGlb-unique*: $[\text{isGlb } R \ S \ x; \text{isGlb } R \ S \ y] \Longrightarrow x = (y::'a::\text{linorder})$
apply (*frule isGlb-isLb*)
apply (*frule-tac x = y in isGlb-isLb*)

apply (*blast intro!*: *order-antisym dest!*: *isGlb-le-isLb*)
done

lemma *bdd-above-setle*: *bdd-above* $A \longleftrightarrow (\exists a. A * \leq a)$
by (*auto simp*: *bdd-above-def setle-def*)

lemma *bdd-below-setge*: *bdd-below* $A \longleftrightarrow (\exists a. a \leq * A)$
by (*auto simp*: *bdd-below-def setge-def*)

lemma *isLub-cSup*:
 $(S :: 'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies (\exists b. S * \leq b) \implies \text{isLub } UNIV\ S\ (Sup\ S)$
by (*auto simp add*: *isLub-def setle-def leastP-def isUb-def intro!*: *setgeI cSup-upper cSup-least*)

lemma *isGlb-cInf*:
 $(S :: 'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies (\exists b. b \leq * S) \implies \text{isGlb } UNIV\ S\ (Inf\ S)$
by (*auto simp add*: *isGlb-def setge-def greatestP-def isLb-def intro!*: *setleI cInf-lower cInf-greatest*)

lemma *cSup-le*: $(S :: 'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies S * \leq b \implies Sup\ S \leq b$
by (*metis cSup-least setle-def*)

lemma *cInf-ge*: $(S :: 'a :: \text{conditionally-complete-lattice set}) \neq \{\} \implies b \leq * S \implies Inf\ S \geq b$
by (*metis cInf-greatest setge-def*)

lemma *cSup-bounds*:
fixes $S :: 'a :: \text{conditionally-complete-lattice set}$
shows $S \neq \{\} \implies a \leq * S \implies S * \leq b \implies a \leq Sup\ S \wedge Sup\ S \leq b$
using *cSup-least*[*of* $S\ b$] *cSup-upper2*[*of* $- S\ a$]
by (*auto simp*: *bdd-above-setle setge-def setle-def*)

lemma *cSup-unique*: $(S :: 'a :: \{\text{conditionally-complete-linorder, no-bot}\ set}) * \leq b \implies (\forall b' < b. \exists x \in S. b' < x) \implies Sup\ S = b$
by (*rule cSup-eq*) (*auto simp*: *not-le[symmetric] setle-def*)

lemma *cInf-unique*: $b \leq * (S :: 'a :: \{\text{conditionally-complete-linorder, no-top}\ set}) \implies (\forall b' > b. \exists x \in S. b' > x) \implies Inf\ S = b$
by (*rule cInf-eq*) (*auto simp*: *not-le[symmetric] setge-def*)

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: $\exists X. X \in S \implies \exists Y. \text{isUb } (UNIV :: \text{real set})\ S\ Y \implies \exists t. \text{isLub } (UNIV :: \text{real set})\ S\ t$
by (*intro exI*[*of* $- Sup\ S$] *isLub-cSup*) (*auto simp*: *setle-def isUb-def intro!*: *cSup-upper*)

lemma *Bseq-isUb*: $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \Longrightarrow \exists U. \text{isUb } (UNIV :: \text{real set}) \{x. \exists n. X \ n = x\} \ U$

by (*auto intro: isUbI settleI simp add: Bseq-def abs-le-iff*)

lemma *Bseq-isLub*: $\bigwedge X :: \text{nat} \Rightarrow \text{real}. \text{Bseq } X \Longrightarrow \exists U. \text{isLub } (UNIV :: \text{real set}) \{x. \exists n. X \ n = x\} \ U$

by (*blast intro: reals-complete Bseq-isUb*)

lemma *isLub-mono-imp-LIMSEQ*:

fixes $X :: \text{nat} \Rightarrow \text{real}$

assumes $u: \text{isLub } UNIV \{x. \exists n. X \ n = x\} \ u$

assumes $X: \forall m \ n. m \leq n \longrightarrow X \ m \leq X \ n$

shows $X \longrightarrow u$

proof –

have $X \longrightarrow (SUP \ i. X \ i)$

using $u[THEN \ \text{isLubD1}] \ X$

by (*intro LIMSEQ-incseq-SUP*) (*auto simp: incseq-def image-def eq-commute bdd-above-settle*)

also have $(SUP \ i. X \ i) = u$

using $\text{isLub-cSup}[of \ range \ X] \ u[THEN \ \text{isLubD1}]$

by (*intro isLub-unique[OF - u]*) (*auto simp add: image-def eq-commute*)

finally show *?thesis* .

qed

lemmas *real-isGlb-unique* = *isGlb-unique*[**where** 'a=real]

lemma *real-le-inf-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists b. b \leq^* s \Longrightarrow \text{Inf } s \leq \text{Inf } (t :: \text{real set})$

by (*rule cInf-superset-mono*) (*auto simp: bdd-below-setge*)

lemma *real-ge-sup-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists b. s \leq^* b \Longrightarrow \text{Sup } s \geq \text{Sup } (t :: \text{real set})$

by (*rule cSup-subset-mono*) (*auto simp: bdd-above-settle*)

end

64 An abstract view on maps for code generation.

theory *Mapping*

imports *Main AList*

begin

64.1 Parametricity transfer rules

lemma *map-of-foldr*: $\text{map-of } xs = \text{foldr } (\lambda(k, v) \ m. m(k \mapsto v)) \ xs \ \text{Map.empty}$
using *map-add-map-of-foldr* [*of Map.empty*] **by** *auto*

context **includes** *lifting-syntax*

begin

lemma *empty-parametric*: $(A \text{ ===> } \text{rel-option } B) \text{ Map.empty Map.empty}$
by *transfer-prover*

lemma *lookup-parametric*: $((A \text{ ===> } B) \text{ ===> } A \text{ ===> } B) (\lambda m \ k. \ m \ k) (\lambda m \ k. \ m \ k)$
by *transfer-prover*

lemma *update-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows $(A \text{ ===> } B \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } A \text{ ===> } \text{rel-option } B)$
 $(\lambda k \ v \ m. \ m(k \mapsto v)) (\lambda k \ v \ m. \ m(k \mapsto v))$
by *transfer-prover*

lemma *delete-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows $(A \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } A \text{ ===> } \text{rel-option } B)$
 $(\lambda k \ m. \ m(k := \text{None})) (\lambda k \ m. \ m(k := \text{None}))$
by *transfer-prover*

lemma *is-none-parametric* [*transfer-rule*]:
 $(\text{rel-option } A \text{ ===> } \text{HOL.eq}) \text{ Option.is-none Option.is-none}$
by (*auto simp add: Option.is-none-def rel-fun-def rel-option-iff split: option.split*)

lemma *dom-parametric*:
assumes [*transfer-rule*]: *bi-total A*
shows $((A \text{ ===> } \text{rel-option } B) \text{ ===> } \text{rel-set } A) \text{ dom dom}$
unfolding *dom-def [abs-def] Option.is-none-def [symmetric]* **by** *transfer-prover*

lemma *graph-parametric*:
assumes *bi-total A*
shows $((A \text{ ===> } \text{rel-option } B) \text{ ===> } \text{rel-set } (\text{rel-prod } A \ B)) \text{ Map.graph Map.graph}$
proof
fix *f g* **assume** $(A \text{ ===> } \text{rel-option } B) \ f \ g$
with *assms*[*unfolded bi-total-def*] **show** $\text{rel-set } (\text{rel-prod } A \ B) \ (\text{Map.graph } f)$
 $(\text{Map.graph } g)$
unfolding *graph-def rel-set-def rel-fun-def*
by *auto (metis option-rel-Some1 option-rel-Some2)+*
qed

lemma *map-of-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows $(\text{list-all2 } (\text{rel-prod } R1 \ R2) \text{ ===> } R1 \text{ ===> } \text{rel-option } R2) \text{ map-of}$
 map-of
unfolding *map-of-def* **by** *transfer-prover*

lemma *map-entry-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*

shows $(A \text{ ===> } (B \text{ ===> } B) \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } A \text{ ===> } \text{rel-option } B)$
 $(\lambda k f m. (\text{case } m \text{ of } \text{None} \Rightarrow m$
 $\quad | \text{Some } v \Rightarrow m (k \mapsto (f v)))) (\lambda k f m. (\text{case } m \text{ of } \text{None} \Rightarrow m$
 $\quad | \text{Some } v \Rightarrow m (k \mapsto (f v))))$
by *transfer-prover*

lemma *tabulate-parametric*:

assumes [*transfer-rule*]: *bi-unique A*
shows $(\text{list-all2 } A \text{ ===> } (A \text{ ===> } B) \text{ ===> } A \text{ ===> } \text{rel-option } B)$
 $(\lambda ks f. (\text{map-of } (\text{map } (\lambda k. (k, f k)) ks))) (\lambda ks f. (\text{map-of } (\text{map } (\lambda k. (k, f k))$
 $ks)))$
by *transfer-prover*

lemma *bulkload-parametric*:

$(\text{list-all2 } A \text{ ===> } \text{HOL.eq} \text{ ===> } \text{rel-option } A)$
 $(\lambda xs k. \text{if } k < \text{length } xs \text{ then } \text{Some } (xs ! k) \text{ else } \text{None})$
 $(\lambda xs k. \text{if } k < \text{length } xs \text{ then } \text{Some } (xs ! k) \text{ else } \text{None})$
proof
fix *xs ys*
assume *list-all2 A xs ys*
then show
 $(\text{HOL.eq} \text{ ===> } \text{rel-option } A)$
 $(\lambda k. \text{if } k < \text{length } xs \text{ then } \text{Some } (xs ! k) \text{ else } \text{None})$
 $(\lambda k. \text{if } k < \text{length } ys \text{ then } \text{Some } (ys ! k) \text{ else } \text{None})$
by *induct (auto simp add: list-all2-lengthD list-all2-nthD rel-funI)*
qed

lemma *map-parametric*:

$((A \text{ ===> } B) \text{ ===> } (C \text{ ===> } D) \text{ ===> } (B \text{ ===> } \text{rel-option } C) \text{ ===> } A \text{ ===> } \text{rel-option } D)$
 $(\lambda f g m. (\text{map-option } g \circ m \circ f)) (\lambda f g m. (\text{map-option } g \circ m \circ f))$
by *transfer-prover*

lemma *combine-with-key-parametric*:

$((A \text{ ===> } B \text{ ===> } B \text{ ===> } B) \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } (A \text{ ===> } \text{rel-option } B))$
 $(\lambda f m1 m2 x. \text{combine-options } (f x) (m1 x) (m2 x))$
 $(\lambda f m1 m2 x. \text{combine-options } (f x) (m1 x) (m2 x))$
unfolding *combine-options-def* **by** *transfer-prover*

lemma *combine-parametric*:

$((B \text{ ===> } B \text{ ===> } B) \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } (A \text{ ===> } \text{rel-option } B) \text{ ===> } (A \text{ ===> } \text{rel-option } B))$
 $(\lambda f m1 m2 x. \text{combine-options } f (m1 x) (m2 x))$
 $(\lambda f m1 m2 x. \text{combine-options } f (m1 x) (m2 x))$
unfolding *combine-options-def* **by** *transfer-prover*

end

64.2 Type definition and primitive operations

typedef (*'a*, *'b*) *mapping* = *UNIV* :: (*'a* \rightarrow *'b*) *set*
morphisms *rep Mapping* ..

setup-lifting *type-definition-mapping*

lift-definition *empty* :: (*'a*, *'b*) *mapping*
is *Map.empty* **parametric** *empty-parametric* .

lift-definition *lookup* :: (*'a*, *'b*) *mapping* \Rightarrow *'a* \Rightarrow *'b* *option*
is $\lambda m\ k. m\ k$ **parametric** *lookup-parametric* .

definition *lookup-default* *d m k* = (*case Mapping.lookup m k of* *None* \Rightarrow *d* | *Some v* \Rightarrow *v*)

lift-definition *update* :: *'a* \Rightarrow *'b* \Rightarrow (*'a*, *'b*) *mapping* \Rightarrow (*'a*, *'b*) *mapping*
is $\lambda k\ v\ m. m(k \mapsto v)$ **parametric** *update-parametric* .

lift-definition *delete* :: *'a* \Rightarrow (*'a*, *'b*) *mapping* \Rightarrow (*'a*, *'b*) *mapping*
is $\lambda k\ m. m(k := \text{None})$ **parametric** *delete-parametric* .

lift-definition *filter* :: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow (*'a*, *'b*) *mapping* \Rightarrow (*'a*, *'b*) *mapping*
is $\lambda P\ m\ k. \text{case } m\ k \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } v \Rightarrow \text{if } P\ k\ v \text{ then } \text{Some } v \text{ else } \text{None}$
 .

lift-definition *keys* :: (*'a*, *'b*) *mapping* \Rightarrow *'a* *set*
is *dom* **parametric** *dom-parametric* .

lift-definition *entries* :: (*'a*, *'b*) *mapping* \Rightarrow (*'a* \times *'b*) *set*
is *Map.graph* **parametric** *graph-parametric* .

lift-definition *tabulate* :: *'a* *list* \Rightarrow (*'a* \Rightarrow *'b*) \Rightarrow (*'a*, *'b*) *mapping*
is $\lambda ks\ f. (\text{map-of } (\text{List.map } (\lambda k. (k, f\ k))\ ks))$ **parametric** *tabulate-parametric* .

lift-definition *bulkload* :: *'a* *list* \Rightarrow (*nat*, *'a*) *mapping*
is $\lambda xs\ k. \text{if } k < \text{length } xs \text{ then } \text{Some } (xs\ !\ k) \text{ else } \text{None}$ **parametric** *bulkload-parametric* .

lift-definition *map* :: (*'c* \Rightarrow *'a*) \Rightarrow (*'b* \Rightarrow *'d*) \Rightarrow (*'a*, *'b*) *mapping* \Rightarrow (*'c*, *'d*) *mapping*
is $\lambda f\ g\ m. (\text{map-option } g \circ m \circ f)$ **parametric** *map-parametric* .

lift-definition *map-values* :: (*'c* \Rightarrow *'a* \Rightarrow *'b*) \Rightarrow (*'c*, *'a*) *mapping* \Rightarrow (*'c*, *'b*) *mapping*
is $\lambda f\ m\ x. \text{map-option } (f\ x)\ (m\ x)$.

lift-definition *combine-with-key* ::
 (*'a* \Rightarrow *'b* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow (*'a*, *'b*) *mapping* \Rightarrow (*'a*, *'b*) *mapping* \Rightarrow (*'a*, *'b*) *mapping*
is $\lambda f\ m1\ m2\ x. \text{combine-options } (f\ x)\ (m1\ x)\ (m2\ x)$ **parametric** *combine-with-key-parametric*
 .

lift-definition *combine* ::

$(\text{'b} \Rightarrow \text{'b} \Rightarrow \text{'b}) \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$
is $\lambda f m1 m2 x. \text{combine-options } f (m1 x) (m2 x)$ **parametric** *combine-parametric*
 .

definition *All-mapping* $m P \longleftrightarrow$

$(\forall x. \text{case Mapping.lookup } m x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } y \Rightarrow P x y)$

declare $[[\text{code drop: map}]]$

64.3 Functorial structure

functor *map*: *map*

by $(\text{transfer}, \text{auto simp add: fun-eq-iff option.map-comp option.map-id})+$

64.4 Derived operations

definition *ordered-keys* :: $(\text{'a}::\text{linorder}, \text{'b}) \text{ mapping} \Rightarrow \text{'a list}$

where *ordered-keys* $m = (\text{if finite (keys } m) \text{ then sorted-list-of-set (keys } m) \text{ else []})$

definition *ordered-entries* :: $(\text{'a}::\text{linorder}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a} \times \text{'b}) \text{ list}$

where *ordered-entries* $m = (\text{if finite (entries } m) \text{ then sorted-key-list-of-set fst (entries } m) \text{ else []})$

definition *fold* :: $(\text{'a}::\text{linorder} \Rightarrow \text{'b} \Rightarrow \text{'c} \Rightarrow \text{'c}) \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow \text{'c} \Rightarrow \text{'c}$

where *fold* $f m a = \text{List.fold (case-prod } f) (\text{ordered-entries } m) a$

definition *is-empty* :: $(\text{'a}, \text{'b}) \text{ mapping} \Rightarrow \text{bool}$

where *is-empty* $m \longleftrightarrow \text{keys } m = \{\}$

definition *size* :: $(\text{'a}, \text{'b}) \text{ mapping} \Rightarrow \text{nat}$

where *size* $m = (\text{if finite (keys } m) \text{ then card (keys } m) \text{ else } 0)$

definition *replace* :: $\text{'a} \Rightarrow \text{'b} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$

where *replace* $k v m = (\text{if } k \in \text{keys } m \text{ then update } k v m \text{ else } m)$

definition *default* :: $\text{'a} \Rightarrow \text{'b} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$

where *default* $k v m = (\text{if } k \in \text{keys } m \text{ then } m \text{ else update } k v m)$

Manual derivation of transfer rule is non-trivial

lift-definition *map-entry* :: $\text{'a} \Rightarrow (\text{'b} \Rightarrow \text{'b}) \Rightarrow (\text{'a}, \text{'b}) \text{ mapping} \Rightarrow (\text{'a}, \text{'b}) \text{ mapping}$
is

$\lambda k f m.$

$(\text{case } m k \text{ of}$

$\text{None} \Rightarrow m$

$\mid \text{Some } v \Rightarrow m (k \mapsto (f v)))$ **parametric** *map-entry-parametric* .

lemma *map-entry-code* $[\text{code}]$:

```

map-entry k f m =
  (case lookup m k of
    None  $\Rightarrow$  m
  | Some v  $\Rightarrow$  update k (f v) m)
by transfer rule

```

definition *map-default* :: 'a \Rightarrow 'b \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) mapping \Rightarrow ('a, 'b) mapping
where *map-default* k v f m = *map-entry* k f (default k v m)

definition *of-alist* :: ('k \times 'v) list \Rightarrow ('k, 'v) mapping
where *of-alist* xs = *foldr* ($\lambda(k, v) m. \text{update } k \ v \ m$) xs empty

instantiation *mapping* :: (type, type) equal
begin

definition *HOL.equal* m1 m2 \longleftrightarrow ($\forall k. \text{lookup } m1 \ k = \text{lookup } m2 \ k$)

instance

proof

```

show  $\bigwedge x \ y :: ('a, 'b) \text{ mapping}. \text{equal-class.equal } x \ y = (x = y)$ 
  unfolding equal-mapping-def
  by transfer auto

```

qed

end

context includes *lifting-syntax*
begin

lemma [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-total* A

and [*transfer-rule*]: *bi-unique* B

shows (*pcr-mapping* A B \implies *pcr-mapping* A B \implies (=)) *HOL.eq* *HOL.equal*

unfolding *equal* **by** *transfer-prover*

lemma *of-alist-transfer* [*transfer-rule*]:

assumes [*transfer-rule*]: *bi-unique* R1

shows (*list-all2* (*rel-prod* R1 R2) \implies *pcr-mapping* R1 R2) *map-of of-alist*

unfolding *of-alist-def* [*abs-def*] *map-of-foldr* [*abs-def*] **by** *transfer-prover*

end

64.5 Properties

lemma *mapping-eqI*: ($\bigwedge x. \text{lookup } m \ x = \text{lookup } m' \ x$) $\implies m = m'$
by *transfer* (*simp add: fun-eq-iff*)

lemma *mapping-eqI'*:

```

assumes  $\bigwedge x. x \in \text{Mapping.keys } m \implies \text{Mapping.lookup-default } d \ m \ x = \text{Mapping.lookup-default } d \ m' \ x$ 
and  $\text{Mapping.keys } m = \text{Mapping.keys } m'$ 
shows  $m = m'$ 
proof (intro mapping-eqI)
show  $\text{Mapping.lookup } m \ x = \text{Mapping.lookup } m' \ x$  for  $x$ 
proof (cases Mapping.lookup m x)
  case None
  then have  $x \notin \text{Mapping.keys } m$ 
  by transfer (simp add: dom-def)
  then have  $x \notin \text{Mapping.keys } m'$ 
  by (simp add: assms)
  then have  $\text{Mapping.lookup } m' \ x = \text{None}$ 
  by transfer (simp add: dom-def)
  with None show ?thesis
  by simp
next
  case (Some y)
  then have  $A: x \in \text{Mapping.keys } m$ 
  by transfer (simp add: dom-def)
  then have  $x \in \text{Mapping.keys } m'$ 
  by (simp add: assms)
  then have  $\exists y'. \text{Mapping.lookup } m' \ x = \text{Some } y'$ 
  by transfer (simp add: dom-def)
  with Some assms(1)[OF A] show ?thesis
  by (auto simp add: lookup-default-def)
qed
qed

lemma lookup-update[simp]:  $\text{lookup } (\text{update } k \ v \ m) \ k = \text{Some } v$ 
by transfer simp

lemma lookup-update-neq[simp]:  $k \neq k' \implies \text{lookup } (\text{update } k \ v \ m) \ k' = \text{lookup } m \ k'$ 
by transfer simp

lemma lookup-update':  $\text{lookup } (\text{update } k \ v \ m) \ k' = (\text{if } k = k' \text{ then } \text{Some } v \text{ else } \text{lookup } m \ k')$ 
by transfer simp

lemma lookup-empty[simp]:  $\text{lookup empty } k = \text{None}$ 
by transfer simp

lemma lookup-delete[simp]:  $\text{lookup } (\text{delete } k \ m) \ k = \text{None}$ 
by transfer simp

lemma lookup-delete-neq[simp]:  $k \neq k' \implies \text{lookup } (\text{delete } k \ m) \ k' = \text{lookup } m \ k'$ 
by transfer simp

```

lemma *lookup-filter:*

lookup (filter P m) k =
(case lookup m k of
None \Rightarrow None
| Some v \Rightarrow if P k v then Some v else None)
by *transfer simp-all*

lemma *lookup-map-values:* *lookup (map-values f m) k = map-option (f k) (lookup m k)*

by *transfer simp-all*

lemma *lookup-default-empty:* *lookup-default d empty k = d*

by *(simp add: lookup-default-def lookup-empty)*

lemma *lookup-default-update:* *lookup-default d (update k v m) k = v*

by *(simp add: lookup-default-def)*

lemma *lookup-default-update-neq:*

k \neq k' \implies lookup-default d (update k v m) k' = lookup-default d m k'
by *(simp add: lookup-default-def)*

lemma *lookup-default-update':*

lookup-default d (update k v m) k' = (if k = k' then v else lookup-default d m k')
by *(auto simp: lookup-default-update lookup-default-update-neq)*

lemma *lookup-default-filter:*

lookup-default d (filter P m) k =
(if P k (lookup-default d m k) then lookup-default d m k else d)
by *(simp add: lookup-default-def lookup-filter split: option.splits)*

lemma *lookup-default-map-values:*

lookup-default (f k d) (map-values f m) k = f k (lookup-default d m k)
by *(simp add: lookup-default-def lookup-map-values split: option.splits)*

lemma *lookup-combine-with-key:*

Mapping.lookup (combine-with-key f m1 m2) x =
combine-options (f x) (Mapping.lookup m1 x) (Mapping.lookup m2 x)
by *transfer (auto split: option.splits)*

lemma *combine-altdef:* *combine f m1 m2 = combine-with-key (λ -. f) m1 m2*

by *transfer' (rule refl)*

lemma *lookup-combine:*

Mapping.lookup (combine f m1 m2) x =
combine-options f (Mapping.lookup m1 x) (Mapping.lookup m2 x)
by *transfer (auto split: option.splits)*

lemma *lookup-default-neutral-combine-with-key:*

assumes $\bigwedge x. f k d x = x \bigwedge x. f k x d = x$

shows *Mapping.lookup-default d (combine-with-key f m1 m2) k =*
f k (Mapping.lookup-default d m1 k) (Mapping.lookup-default d m2 k)
by (*auto simp: lookup-default-def lookup-combine-with-key assms split: option.splits*)

lemma *lookup-default-neutral-combine:*
assumes $\bigwedge x. f\ d\ x = x \ \bigwedge x. f\ x\ d = x$
shows *Mapping.lookup-default d (combine f m1 m2) x =*
f (Mapping.lookup-default d m1 x) (Mapping.lookup-default d m2 x)
by (*auto simp: lookup-default-def lookup-combine assms split: option.splits*)

lemma *lookup-map-entry:* *lookup (map-entry x f m) x = map-option f (lookup m x)*
by *transfer (auto split: option.splits)*

lemma *lookup-map-entry-neq:* $x \neq y \implies \text{lookup (map-entry x f m) y} = \text{lookup m y}$
by *transfer (auto split: option.splits)*

lemma *lookup-map-entry':*
lookup (map-entry x f m) y =
(if x = y then map-option f (lookup m y) else lookup m y)
by *transfer (auto split: option.splits)*

lemma *lookup-default:* *lookup (default x d m) x = Some (lookup-default d m x)*
unfolding *lookup-default-def default-def*
by *transfer (auto split: option.splits)*

lemma *lookup-default-neq:* $x \neq y \implies \text{lookup (default x d m) y} = \text{lookup m y}$
unfolding *lookup-default-def default-def*
by *transfer (auto split: option.splits)*

lemma *lookup-default':*
lookup (default x d m) y =
(if x = y then Some (lookup-default d m x) else lookup m y)
unfolding *lookup-default-def default-def*
by *transfer (auto split: option.splits)*

lemma *lookup-map-default:* *lookup (map-default x d f m) x = Some (f (lookup-default d m x))*
unfolding *lookup-default-def default-def*
by (*simp add: map-default-def lookup-map-entry lookup-default lookup-default-def*)

lemma *lookup-map-default-neq:* $x \neq y \implies \text{lookup (map-default x d f m) y} = \text{lookup m y}$
unfolding *lookup-default-def default-def*
by (*simp add: map-default-def lookup-map-entry-neq lookup-default-neq*)

lemma *lookup-map-default':*
lookup (map-default x d f m) y =

(if $x = y$ then $\text{Some } (f \text{ (lookup-default } d \text{ } m \text{ } x))$ else $\text{lookup } m \text{ } y$)
unfolding $\text{lookup-default-def default-def}$
by ($\text{simp add: map-default-def lookup-map-entry' lookup-default' lookup-default-def}$)

lemma *lookup-tabulate*:
assumes *distinct xs*
shows $\text{Mapping.lookup } (\text{Mapping.tabulate } xs \text{ } f) \text{ } x = (\text{if } x \in \text{set } xs \text{ then } \text{Some } (f \text{ } x) \text{ else } \text{None})$
using *assms* **by** $\text{transfer (auto simp: map-of-eq-None-iff o-def dest!: map-of-SomeD)}$

lemma *lookup-of-alist*: $\text{lookup } (\text{of-alist } xs) \text{ } k = \text{map-of } xs \text{ } k$
by transfer simp-all

lemma *keys-is-none-rep* [*code-unfold*]: $k \in \text{keys } m \longleftrightarrow \neg (\text{Option.is-none } (\text{lookup } m \text{ } k))$
by $\text{transfer (auto simp add: Option.is-none-def)}$

lemma *update-update*:
 $\text{update } k \text{ } v \text{ } (\text{update } k \text{ } w \text{ } m) = \text{update } k \text{ } v \text{ } m$
 $k \neq l \implies \text{update } k \text{ } v \text{ } (\text{update } l \text{ } w \text{ } m) = \text{update } l \text{ } w \text{ } (\text{update } k \text{ } v \text{ } m)$
by ($\text{transfer; simp add: fun-upd-twist}$) $+$

lemma *update-delete* [*simp*]: $\text{update } k \text{ } v \text{ } (\text{delete } k \text{ } m) = \text{update } k \text{ } v \text{ } m$
by transfer simp

lemma *delete-update*:
 $\text{delete } k \text{ } (\text{update } k \text{ } v \text{ } m) = \text{delete } k \text{ } m$
 $k \neq l \implies \text{delete } k \text{ } (\text{update } l \text{ } v \text{ } m) = \text{update } l \text{ } v \text{ } (\text{delete } k \text{ } m)$
by ($\text{transfer; simp add: fun-upd-twist}$) $+$

lemma *delete-empty* [*simp*]: $\text{delete } k \text{ } \text{empty} = \text{empty}$
by transfer simp

lemma *Mapping-delete-if-notin-keys* [*simp*]:
 $k \notin \text{keys } m \implies \text{delete } k \text{ } m = m$
by transfer simp

lemma *replace-update*:
 $k \notin \text{keys } m \implies \text{replace } k \text{ } v \text{ } m = m$
 $k \in \text{keys } m \implies \text{replace } k \text{ } v \text{ } m = \text{update } k \text{ } v \text{ } m$
by ($\text{transfer; auto simp add: replace-def fun-upd-twist}$) $+$

lemma *map-values-update*: $\text{map-values } f \text{ } (\text{update } k \text{ } v \text{ } m) = \text{update } k \text{ } (f \text{ } v) \text{ } (\text{map-values } f \text{ } m)$
by $\text{transfer (simp-all add: fun-eq-iff)}$

lemma *size-mono*: $\text{finite } (\text{keys } m') \implies \text{keys } m \subseteq \text{keys } m' \implies \text{size } m \leq \text{size } m'$
unfolding *size-def* **by** ($\text{auto intro: card-mono}$)

lemma *size-empty* [*simp*]: *size empty* = 0
unfolding *size-def* **by** *transfer simp*

lemma *size-update*:
finite (keys m) \implies size (update k v m) =
(if k \in keys m then size m else Suc (size m))
unfolding *size-def* **by** *transfer (auto simp add: insert-dom)*

lemma *size-delete*: *size (delete k m) = (if k \in keys m then size m - 1 else size m)*
unfolding *size-def* **by** *transfer simp*

lemma *size-tabulate* [*simp*]: *size (tabulate ks f) = length (remdups ks)*
unfolding *size-def* **by** *transfer (auto simp add: map-of-map-restrict card-set comp-def)*

lemma *keys-filter*: *keys (filter P m) \subseteq keys m*
by *transfer (auto split: option.splits)*

lemma *size-filter*: *finite (keys m) \implies size (filter P m) \leq size m*
by (*intro size-mono keys-filter*)

lemma *bulkload-tabulate*: *bulkload xs = tabulate [0..*length xs*] (*nth xs*)*
by *transfer (auto simp add: map-of-map-restrict)*

lemma *is-empty-empty* [*simp*]: *is-empty empty*
unfolding *is-empty-def* **by** *transfer simp*

lemma *is-empty-update* [*simp*]: \neg *is-empty (update k v m)*
unfolding *is-empty-def* **by** *transfer simp*

lemma *is-empty-delete*: *is-empty (delete k m) \longleftrightarrow is-empty m \vee keys m = {k}*
unfolding *is-empty-def* **by** *transfer (auto simp del: dom-eq-empty-conv)*

lemma *is-empty-replace* [*simp*]: *is-empty (replace k v m) \longleftrightarrow is-empty m*
unfolding *is-empty-def* *replace-def* **by** *transfer auto*

lemma *is-empty-default* [*simp*]: \neg *is-empty (default k v m)*
unfolding *is-empty-def* *default-def* **by** *transfer auto*

lemma *is-empty-map-entry* [*simp*]: *is-empty (map-entry k f m) \longleftrightarrow is-empty m*
unfolding *is-empty-def* **by** *transfer (auto split: option.split)*

lemma *is-empty-map-values* [*simp*]: *is-empty (map-values f m) \longleftrightarrow is-empty m*
unfolding *is-empty-def* **by** *transfer (auto simp: fun-eq-iff)*

lemma *is-empty-map-default* [*simp*]: \neg *is-empty (map-default k v f m)*
by (*simp add: map-default-def*)

lemma *keys-dom-lookup*: *keys m = dom (Mapping.lookup m)*

by *transfer rule*

lemma *keys-empty* [*simp*]: $\text{keys empty} = \{\}$
by *transfer (fact dom-empty)*

lemma *in-keysD*: $k \in \text{keys } m \implies \exists v. \text{lookup } m \ k = \text{Some } v$
by *transfer (fact domD)*

lemma *keys-update* [*simp*]: $\text{keys (update } k \ v \ m) = \text{insert } k \ (\text{keys } m)$
by *transfer simp*

lemma *keys-delete* [*simp*]: $\text{keys (delete } k \ m) = \text{keys } m - \{k\}$
by *transfer simp*

lemma *keys-replace* [*simp*]: $\text{keys (replace } k \ v \ m) = \text{keys } m$
unfolding *replace-def* **by** *transfer (simp add: insert-absorb)*

lemma *keys-default* [*simp*]: $\text{keys (default } k \ v \ m) = \text{insert } k \ (\text{keys } m)$
unfolding *default-def* **by** *transfer (simp add: insert-absorb)*

lemma *keys-map-entry* [*simp*]: $\text{keys (map-entry } k \ f \ m) = \text{keys } m$
by *transfer (auto split: option.split)*

lemma *keys-map-default* [*simp*]: $\text{keys (map-default } k \ v \ f \ m) = \text{insert } k \ (\text{keys } m)$
by *(simp add: map-default-def)*

lemma *keys-map-values* [*simp*]: $\text{keys (map-values } f \ m) = \text{keys } m$
by *transfer (simp-all add: dom-def)*

lemma *keys-combine-with-key* [*simp*]:
 $\text{Mapping.keys (combine-with-key } f \ m1 \ m2) = \text{Mapping.keys } m1 \cup \text{Mapping.keys } m2$
by *transfer (auto simp: dom-def combine-options-def split: option.splits)*

lemma *keys-combine* [*simp*]: $\text{Mapping.keys (combine } f \ m1 \ m2) = \text{Mapping.keys } m1 \cup \text{Mapping.keys } m2$
by *(simp add: combine-altdef)*

lemma *keys-tabulate* [*simp*]: $\text{keys (tabulate } ks \ f) = \text{set } ks$
by *transfer (simp add: map-of-map-restrict o-def)*

lemma *keys-of-alist* [*simp*]: $\text{keys (of-alist } xs) = \text{set (List.map fst } xs)$
by *transfer (simp-all add: dom-map-of-conv-image-fst)*

lemma *keys-bulkload* [*simp*]: $\text{keys (bulkload } xs) = \{0..<\text{length } xs\}$
by *(simp add: bulkload-tabulate)*

lemma *finite-keys-update* [*simp*]:
 $\text{finite (keys (update } k \ v \ m)) = \text{finite (keys } m)$

by *transfer simp*

lemma *set-ordered-keys* [*simp*]:

finite (*Mapping.keys m*) \implies *set* (*Mapping.ordered-keys m*) = *Mapping.keys m*

unfolding *ordered-keys-def* **by** *transfer auto*

lemma *distinct-ordered-keys* [*simp*]: *distinct* (*ordered-keys m*)

by (*simp add: ordered-keys-def*)

lemma *ordered-keys-infinite* [*simp*]: \neg *finite* (*keys m*) \implies *ordered-keys m* = []

by (*simp add: ordered-keys-def*)

lemma *ordered-keys-empty* [*simp*]: *ordered-keys empty* = []

by (*simp add: ordered-keys-def*)

lemma *sorted-ordered-keys* [*simp*]: *sorted* (*ordered-keys m*)

unfolding *ordered-keys-def* **by** *simp*

lemma *ordered-keys-update* [*simp*]:

$k \in \text{keys } m \implies \text{ordered-keys } (\text{update } k \ v \ m) = \text{ordered-keys } m$

finite (*keys m*) $\implies k \notin \text{keys } m \implies$

ordered-keys (*update k v m*) = *insert k* (*ordered-keys m*)

by (*simp-all add: ordered-keys-def*)

(*auto simp only: sorted-list-of-set-insert-remove[symmetric] insert-absorb*)

lemma *ordered-keys-delete* [*simp*]: *ordered-keys* (*delete k m*) = *remove1 k* (*ordered-keys m*)

proof (*cases finite (keys m)*)

case *False*

then show *?thesis* **by** *simp*

next

case *fin: True*

show *?thesis*

proof (*cases k ∈ keys m*)

case *False*

with *fin have* $k \notin \text{set } (\text{sorted-list-of-set } (\text{keys } m))$

by *simp*

with *False show* *?thesis*

by (*simp add: ordered-keys-def remove1-idem*)

next

case *True*

with *fin show* *?thesis*

by (*simp add: ordered-keys-def sorted-list-of-set-remove*)

qed

qed

lemma *ordered-keys-replace* [*simp*]: *ordered-keys* (*replace k v m*) = *ordered-keys m*

by (*simp add: replace-def*)

lemma *ordered-keys-default* [simp]:

$k \in \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{ordered-keys } m$
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{default } k \ v \ m) = \text{insert } k$
 $(\text{ordered-keys } m)$
by (simp-all add: default-def)

lemma *ordered-keys-map-entry* [simp]: $\text{ordered-keys } (\text{map-entry } k \ f \ m) = \text{ordered-keys } m$

by (simp add: ordered-keys-def)

lemma *ordered-keys-map-default* [simp]:

$k \in \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{ordered-keys } m$
 $\text{finite } (\text{keys } m) \implies k \notin \text{keys } m \implies \text{ordered-keys } (\text{map-default } k \ v \ f \ m) = \text{insert } k$
 $(\text{ordered-keys } m)$
by (simp-all add: map-default-def)

lemma *ordered-keys-tabulate* [simp]: $\text{ordered-keys } (\text{tabulate } ks \ f) = \text{sort } (\text{remdups } ks)$

by (simp add: ordered-keys-def sorted-list-of-set-sort-remdups)

lemma *ordered-keys-bulkload* [simp]: $\text{ordered-keys } (\text{bulkload } ks) = [0..<\text{length } ks]$

by (simp add: ordered-keys-def)

lemma *tabulate-fold*: $\text{tabulate } xs \ f = \text{List.fold } (\lambda k \ m. \text{update } k \ (f \ k) \ m) \ xs \ \text{empty}$

proof transfer

fix $f :: 'a \Rightarrow 'b$ **and** xs

have $\text{map-of } (\text{List.map } (\lambda k. (k, f \ k))) \ xs = \text{foldr } (\lambda k \ m. m(k \mapsto f \ k)) \ xs$
 Map.empty

by (simp add: foldr-map comp-def map-of-foldr)

also have $\text{foldr } (\lambda k \ m. m(k \mapsto f \ k)) \ xs = \text{List.fold } (\lambda k \ m. m(k \mapsto f \ k)) \ xs$

by (rule foldr-fold) (simp add: fun-eq-iff)

ultimately show $\text{map-of } (\text{List.map } (\lambda k. (k, f \ k))) \ xs = \text{List.fold } (\lambda k \ m. m(k \mapsto f \ k)) \ xs \ \text{Map.empty}$

by simp

qed

lemma *All-mapping-mono*:

$(\bigwedge k \ v. k \in \text{keys } m \implies P \ k \ v \implies Q \ k \ v) \implies \text{All-mapping } m \ P \implies \text{All-mapping } m \ Q$

unfolding *All-mapping-def* **by** transfer (auto simp: *All-mapping-def* *dom-def* *split: option.splits*)

lemma *All-mapping-empty* [simp]: $\text{All-mapping } \text{Mapping.empty} \ P$

by (auto simp: *All-mapping-def* *lookup-empty*)

lemma *All-mapping-update-iff*:

$\text{All-mapping } (\text{Mapping.update } k \ v \ m) \ P \longleftrightarrow P \ k \ v \wedge \text{All-mapping } m \ (\lambda k' \ v'. k = k' \vee P \ k' \ v')$

unfolding *All-mapping-def*

```

proof safe
  assume  $\forall x. \text{case Mapping.lookup (Mapping.update } k \ v \ m) \ x \text{ of None} \Rightarrow \text{True} \mid$ 
 $\text{Some } y \Rightarrow P \ x \ y$ 
  then have *:  $\text{case Mapping.lookup (Mapping.update } k \ v \ m) \ x \text{ of None} \Rightarrow \text{True} \mid$ 
 $\text{Some } y \Rightarrow P \ x \ y$  for  $x$ 
    by blast
  from *[of  $k$ ] show  $P \ k \ v$ 
    by (simp add: lookup-update)
  show  $\text{case Mapping.lookup } m \ x \text{ of None} \Rightarrow \text{True} \mid \text{Some } v' \Rightarrow k = x \vee P \ x \ v'$ 
for  $x$ 
    using *[of  $x$ ] by (auto simp add: lookup-update' split: if-splits option.splits)
next
  assume  $P \ k \ v$ 
  assume  $\forall x. \text{case Mapping.lookup } m \ x \text{ of None} \Rightarrow \text{True} \mid \text{Some } v' \Rightarrow k = x \vee P$ 
 $x \ v'$ 
  then have  $A: \text{case Mapping.lookup } m \ x \text{ of None} \Rightarrow \text{True} \mid \text{Some } v' \Rightarrow k = x \vee$ 
 $P \ x \ v'$  for  $x$ 
    by blast
  show  $\text{case Mapping.lookup (Mapping.update } k \ v \ m) \ x \text{ of None} \Rightarrow \text{True} \mid \text{Some}$ 
 $xa \Rightarrow P \ x \ xa$  for  $x$ 
    using  $\langle P \ k \ v \rangle \ A$ [of  $x$ ] by (auto simp: lookup-update' split: option.splits)
qed

```

lemma *All-mapping-update:*

$P \ k \ v \implies \text{All-mapping } m \ (\lambda k' \ v'. k = k' \vee P \ k' \ v') \implies \text{All-mapping (Mapping.update } k \ v \ m) \ P$
by (simp add: All-mapping-update-iff)

lemma *All-mapping-filter-iff:* $\text{All-mapping (filter } P \ m) \ Q \longleftrightarrow \text{All-mapping } m \ (\lambda k \ v. P \ k \ v \longrightarrow Q \ k \ v)$

by (auto simp: All-mapping-def lookup-filter split: option.splits)

lemma *All-mapping-filter:* $\text{All-mapping } m \ Q \implies \text{All-mapping (filter } P \ m) \ Q$

by (auto simp: All-mapping-filter-iff intro: All-mapping-mono)

lemma *All-mapping-map-values:* $\text{All-mapping (map-values } f \ m) \ P \longleftrightarrow \text{All-mapping } m \ (\lambda k \ v. P \ k \ (f \ k \ v))$

by (auto simp: All-mapping-def lookup-map-values split: option.splits)

lemma *All-mapping-tabulate:* $(\forall x \in \text{set } xs. P \ x \ (f \ x)) \implies \text{All-mapping (Mapping.tabulate } xs \ f) \ P$

unfolding All-mapping-def

by transfer (auto split: option.split dest!: map-of-SomeD)

lemma *All-mapping-alist:*

$(\bigwedge k \ v. (k, v) \in \text{set } xs \implies P \ k \ v) \implies \text{All-mapping (Mapping.of-alist } xs) \ P$

by (auto simp: All-mapping-def lookup-of-alist dest!: map-of-SomeD split: option.splits)

lemma *combine-empty* [simp]: *combine f Mapping.empty y = y combine f y Mapping.empty = y*

by (transfer; force)+

lemma (in *abel-semigroup*) *comm-monoid-set-combine*: *comm-monoid-set (combine f) Mapping.empty*

by standard (transfer fixing: f, simp add: combine-options-ac[of f] ac-simps)+

locale *combine-mapping-abel-semigroup* = *abel-semigroup*

begin

sublocale *combine*: *comm-monoid-set combine f Mapping.empty*

by (rule *comm-monoid-set-combine*)

lemma *fold-combine-code*:

combine.F g (set xs) = foldr ($\lambda x. \text{combine } f (g x)$) (remdups xs) Mapping.empty

proof –

have *combine.F g (set xs) = foldr ($\lambda x. \text{combine } f (g x)$) xs Mapping.empty*

if *distinct xs* **for** *xs*

using *that* **by** (*induction xs*) *simp-all*

from *this*[of *remdups xs*] **show** *?thesis* **by** *simp*

qed

lemma *keys-fold-combine*: *finite A \implies Mapping.keys (combine.F g A) = ($\bigcup_{x \in A} \text{Mapping.keys } (g x)$)*

by (*induct A* rule: *finite-induct*) *simp-all*

end

64.5.1 entries, ordered-entries, and fold

context *linorder*

begin

sublocale *folding-Map-graph*: *folding-insort-key (\leq) ($<$) Map.graph m fst for m*

by *unfold-locales (fact inj-on-fst-graph)*

end

lemma *sorted-fst-list-of-set-insort-Map-graph*[simp]:

assumes *finite (dom m) fst x \notin dom m*

shows *sorted-key-list-of-set fst (insert x (Map.graph m))*

= insort-key fst x (sorted-key-list-of-set fst (Map.graph m))

proof(*cases x*)

case (*Pair k v*)

with $\langle \text{fst } x \notin \text{dom } m \rangle$ **have** *Map.graph m \subseteq Map.graph (m(k \mapsto v))*

by(*auto simp: graph-def*)

moreover from *Pair $\langle \text{fst } x \notin \text{dom } m \rangle$* **have** *(k, v) \notin Map.graph m*

using *graph-domD* **by** *fastforce*

```

ultimately show ?thesis
  using Pair assms folding-Map-graph.sorted-key-list-of-set-insert[where ?m=m(k
  ↦ v)]
  by auto
qed

```

```

lemma sorted-fst-list-of-set-insort-insert-Map-graph[simp]:
  assumes finite (dom m) fst x ∉ dom m
  shows sorted-key-list-of-set fst (insert x (Map.graph m))
    = insort-insert-key fst x (sorted-key-list-of-set fst (Map.graph m))
proof(cases x)
  case (Pair k v)
  with ⟨fst x ∉ dom m⟩ have Map.graph m ⊆ Map.graph (m(k ↦ v))
    by(auto simp: graph-def)
  with assms Pair show ?thesis
  unfolding sorted-fst-list-of-set-insort-Map-graph[OF assms] insort-insert-key-def
  using folding-Map-graph.set-sorted-key-list-of-set in-graphD by (fastforce split:
  if-splits)
qed

```

```

lemma linorder-finite-Map-induct[consumes 1, case-names empty update]:
  fixes m :: 'a::linorder → 'b
  assumes finite (dom m)
  assumes P Map.empty
  assumes ∧k v m. [finite (dom m); k ∉ dom m; (∧k'. k' ∈ dom m ⇒ k' ≤ k);
  P m]
    ⇒ P (m(k ↦ v))
  shows P m
proof -
  let ?key-list = λm. sorted-list-of-set (dom m)
  from assms(1,2) show ?thesis
  proof(induction length (?key-list m) arbitrary: m)
    case 0
    then have sorted-list-of-set (dom m) = []
      by auto
    with ⟨finite (dom m)⟩ have m = Map.empty
      by auto
    with ⟨P Map.empty⟩ show ?case by simp
  next
    case (Suc n)
    then obtain x xs where x-xs: sorted-list-of-set (dom m) = xs @ [x]
      by (metis append-butlast-last-id length-greater-0-conv zero-less-Suc)
    have sorted-list-of-set (dom (m(x := None))) = xs
    proof -
      have distinct (xs @ [x])
        by (metis sorted-list-of-set.distinct-sorted-key-list-of-set x-xs)
      then have remove1 x (xs @ [x]) = xs
        by (simp add: remove1-append)
      with ⟨finite (dom m)⟩ x-xs show ?thesis

```

```

    by (simp add: sorted-list-of-set-remove)
  qed
  moreover have  $k \leq x$  if  $k \in \text{dom } (m(x := \text{None}))$  for  $k$ 
  proof -
    from  $x$ -xs have sorted ( $xs @ [x]$ )
    by (metis sorted-list-of-set.sorted-sorted-key-list-of-set)
    moreover from  $\langle k \in \text{dom } (m(x := \text{None})) \rangle$  have  $k \in \text{set } xs$ 
    using  $\langle \text{finite } (\text{dom } m) \rangle \langle \text{sorted-list-of-set } (\text{dom } (m(x := \text{None}))) = xs \rangle$ 
    by auto
    ultimately show  $k \leq x$ 
    by (simp add: sorted-append)
  qed
  moreover from  $\langle \text{finite } (\text{dom } m) \rangle$  have  $\text{finite } (\text{dom } (m(x := \text{None})))$   $x \notin \text{dom } (m(x := \text{None}))$ 
  by simp-all
  moreover have  $P (m(x := \text{None}))$ 
  using Suc  $\langle \text{sorted-list-of-set } (\text{dom } (m(x := \text{None}))) = xs \rangle$   $x$ -xs by auto
  ultimately show ?case
  using assms(3)[where ?m=m(x := None)] by (metis fun-upd-triv fun-upd-upd
not-Some-eq)
  qed
  qed

```

lemma *delete-insort-fst[simp]*: $\text{AList.delete } k \ (\text{insort-key fst } (k, v) \ xs) = \text{AList.delete } k \ xs$
 by (induction xs) simp-all

lemma *insort-fst-delete*: $\llbracket \text{fst } x \neq k2; \text{sorted } (\text{List.map fst } xs) \rrbracket$
 $\implies \text{insort-key fst } x \ (\text{AList.delete } k2 \ xs) = \text{AList.delete } k2 \ (\text{insort-key fst } x \ xs)$
 by (induction xs) (fastforce simp add: insort-is-Cons order-trans)+

lemma *sorted-fst-list-of-set-Map-graph-fun-upd-None[simp]*:
 $\text{sorted-key-list-of-set fst } (\text{Map.graph } (m(k := \text{None})))$
 $= \text{AList.delete } k \ (\text{sorted-key-list-of-set fst } (\text{Map.graph } m))$
proof(cases finite (Map.graph m))
 assume finite (Map.graph m)
 from this[unfolded finite-graph-iff-finite-dom] show ?thesis
proof(induction rule: finite-Map-induct)
 let ?list-of=sorted-key-list-of-set fst
 case (update k2 v2 m)
 note [simp] = $\langle k2 \notin \text{dom } m \rangle \langle \text{finite } (\text{dom } m) \rangle$

have right-eq: $\text{AList.delete } k \ (\text{?list-of } (\text{Map.graph } (m(k2 \mapsto v2))))$
 $= \text{AList.delete } k \ (\text{insort-key fst } (k2, v2) \ (\text{?list-of } (\text{Map.graph } m)))$
 by simp

show ?case
proof(cases $k = k2$)
 case True

```

then have ?list-of (Map.graph ((m(k2 ↦ v2))(k := None)))
  = AList.delete k (insort-key fst (k2, v2) (?list-of (Map.graph m)))
using fst-graph-eq-dom update.IH by auto
then show ?thesis
using right-eq by metis
next
case False
then have AList.delete k (insort-key fst (k2, v2) (?list-of (Map.graph m)))
  = insort-key fst (k2, v2) (?list-of (Map.graph (m(k := None))))
by (auto simp add: insort-fst-delete update.IH
    folding-Map-graph.sorted-sorted-key-list-of-set[OF subset-refl])
also have ... = ?list-of (insert (k2, v2) (Map.graph (m(k := None))))
by auto
also from False ⟨k2 ∉ dom m⟩ have ... = ?list-of (Map.graph ((m(k2 ↦
v2))(k := None)))
by (metis graph-map-upd domIff fun-upd-triv fun-upd-twist)
finally show ?thesis using right-eq by metis
qed
qed simp
qed simp

lemma entries-empty[simp]: entries empty = {}
by transfer (fact graph-empty)

lemma entries-lookup: entries m = Map.graph (lookup m)
by transfer rule

lemma in-entriesI: lookup m k = Some v ⟹ (k, v) ∈ entries m
by transfer (fact in-graphI)

lemma in-entriesD: (k, v) ∈ entries m ⟹ lookup m k = Some v
by transfer (fact in-graphD)

lemma fst-image-entries-eq-keys[simp]: fst ` Mapping.entries m = Mapping.keys
m
by transfer (fact fst-graph-eq-dom)

lemma finite-entries-iff-finite-keys[simp]:
  finite (entries m) = finite (keys m)
by transfer (fact finite-graph-iff-finite-dom)

lemma entries-update:
  entries (update k v m) = insert (k, v) (entries (delete k m))
by transfer (fact graph-map-upd)

lemma entries-delete:
  entries (delete k m) = {e ∈ entries m. fst e ≠ k}
by transfer (fact graph-fun-upd-None)

```


lemma *entries-of-alist*[simp]:
 $\text{distinct } (\text{List.map fst } xs) \implies \text{entries } (\text{of-alist } xs) = \text{set } xs$
by *transfer* (*fact graph-map-of-if-distinct-dom*)

lemma *entries-keysD*:
 $x \in \text{entries } m \implies \text{fst } x \in \text{keys } m$
by *transfer* (*fact graph-domD*)

lemma *set-ordered-entries*[simp]:
 $\text{finite } (\text{keys } m) \implies \text{set } (\text{ordered-entries } m) = \text{entries } m$
unfolding *ordered-entries-def*
by *transfer* (*auto simp: folding-Map-graph.set-sorted-key-list-of-set[OF subset-refl]*)

lemma *distinct-ordered-entries*[simp]: $\text{distinct } (\text{List.map fst } (\text{ordered-entries } m))$
unfolding *ordered-entries-def*
by *transfer* (*simp add: folding-Map-graph.distinct-sorted-key-list-of-set[OF subset-refl]*)

lemma *sorted-ordered-entries*[simp]: $\text{sorted } (\text{List.map fst } (\text{ordered-entries } m))$
unfolding *ordered-entries-def*
by *transfer* (*auto intro: folding-Map-graph.sorted-sorted-key-list-of-set*)

lemma *ordered-entries-infinite*[simp]:
 $\neg \text{finite } (\text{Mapping.keys } m) \implies \text{ordered-entries } m = []$
by (*simp add: ordered-entries-def*)

lemma *ordered-entries-empty*[simp]: $\text{ordered-entries empty} = []$
by (*simp add: ordered-entries-def*)

lemma *ordered-entries-update*[simp]:
assumes *finite (keys m)*
shows $\text{ordered-entries } (\text{update } k \ v \ m) = \text{insert-insert-key fst } (k, v) \ (\text{AList.delete } k \ (\text{ordered-entries } m))$
proof –
let *?list-of=sorted-key-list-of-set fst and ?insert=insert-insert-key fst*

have $*$: $\text{?list-of } (\text{insert } (k, v) \ (\text{Map.graph } (m(k := \text{None})))) = \text{?insert } (k, v) \ (\text{AList.delete } k \ (\text{?list-of } (\text{Map.graph } m)))$ **if** *finite (dom m)* **for** *m*
proof –
from $\langle \text{finite } (\text{dom } m) \rangle$ **have** $\text{?list-of } (\text{insert } (k, v) \ (\text{Map.graph } (m(k := \text{None})))) = \text{?insert } (k, v) \ (\text{?list-of } (\text{Map.graph } (m(k := \text{None}))))$
by (*intro sorted-fst-list-of-set-insert-insert-Map-graph (simp-all add: subset-insertI)*)
then show *?thesis* **by** *simp*
qed
from *assms* **show** *?thesis*
unfolding *ordered-entries-def*
by (*transfer fixing: k v (use * in auto)*)

qed

lemma *ordered-entries-delete*[simp]:

ordered-entries (delete k m) = *AList.delete* k (*ordered-entries* m)

unfolding *ordered-entries-def* **by** *transfer auto*

lemma *map-fst-ordered-entries*[simp]:

List.map fst (*ordered-entries* m) = *ordered-keys* m

proof(*cases finite* (*Mapping.keys* m))

case *True*

then have *set* (*List.map fst* (*Mapping.ordered-entries* m)) = *set* (*Mapping.ordered-keys* m)

unfolding *ordered-entries-def ordered-keys-def*

by (*transfer*) (*simp add: folding-Map-graph.set-sorted-key-list-of-set[OF subset-refl] fst-graph-eq-dom*)

with True show *List.map fst* (*Mapping.ordered-entries* m) = *Mapping.ordered-keys* m

m

by (*metis distinct-ordered-entries ordered-keys-def sorted-list-of-set.idem-if-sorted-distinct*

sorted-list-of-set.set-sorted-key-list-of-set sorted-ordered-entries)

next

case *False*

then show *?thesis*

unfolding *ordered-entries-def ordered-keys-def* **by** *simp*

qed

lemma *fold-empty*[simp]: *fold f empty a* = a

unfolding *fold-def* **by** *simp*

lemma *insort-key-is-snoc-if-sorted-and-distinct*:

assumes *sorted* (*List.map f xs*) $f\ y \notin f\ ' \text{ set } xs \ \forall x \in \text{set } xs. f\ x \leq f\ y$

shows *insort-key f y xs* = $xs @ [y]$

using *assms by (induction xs) (auto dest!: insort-is-Cons)*

lemma *fold-update*:

assumes *finite* (*keys m*)

assumes $k \notin \text{keys } m \wedge k'. k' \in \text{keys } m \implies k' \leq k$

shows *fold f* (*update k v m*) a = $f\ k\ v$ (*fold f m a*)

proof –

from *assms have* *k-notin-entries*: $k \notin \text{fst } ' \text{ set } (\text{ordered-entries } m)$

using *entries-keysD* **by** *fastforce*

with *assms have* *ordered-entries* (*update k v m*)

= *insort-insert-key fst* (k, v) (*ordered-entries m*)

by *simp*

also from *k-notin-entries have* $\dots = \text{ordered-entries } m @ [(k, v)]$

proof –

from *assms have* $\forall x \in \text{set } (\text{ordered-entries } m). \text{fst } x \leq \text{fst } (k, v)$

unfolding *ordered-entries-def*

by *transfer (fastforce simp: folding-Map-graph.set-sorted-key-list-of-set[OF*

```

order-refl]
      dest: graph-domD)
  from insert-key-is-snoc-if-sorted-and-distinct[OF - - this] k-notin-entries ⟨finite
(keys m)⟩
  show ?thesis
  using sorted-ordered-keys
  unfolding insert-insert-key-def by auto
qed
finally show ?thesis unfolding fold-def by simp
qed

lemma linorder-finite-Mapping-induct[consumes 1, case-names empty update]:
  fixes m :: ('a::linorder, 'b) mapping
  assumes finite (keys m)
  assumes P empty
  assumes  $\bigwedge k v m.$ 
     $\llbracket \text{finite } (\text{keys } m); k \notin \text{keys } m; (\bigwedge k'. k' \in \text{keys } m \implies k' \leq k); P m \rrbracket$ 
     $\implies P (\text{update } k v m)$ 
  shows P m
  using assms by transfer (simp add: linorder-finite-Map-induct)

```

64.6 Code generator setup

```

hide-const (open) empty is-empty rep lookup lookup-default filter update delete
ordered-keys
  keys size replace default map-entry map-default tabulate bulkload map map-values
  combine of-alist
  entries ordered-entries fold

end

```

65 Monad notation for arbitrary types

```

theory Monad-Syntax
  imports Main
begin

```

We provide a convenient *do*-notation for monadic expressions well-known from Haskell. *Let* is printed specially in *do*-expressions.

```

consts
  bind :: 'a  $\Rightarrow$  ('b  $\Rightarrow$  'c)  $\Rightarrow$  'd (infixl <math>\gg> 54)

```

```

notation (ASCII)
  bind (infixl <math>\gg> 54)

```

```

abbreviation (do-notation)
  bind-do :: 'a  $\Rightarrow$  ('b  $\Rightarrow$  'c)  $\Rightarrow$  'd
  where bind-do  $\equiv$  bind

```

notation (output)

bind-do (**infixl** $\langle \gg \rangle$ 54)

notation (ASCII output)

bind-do (**infixl** $\langle > > = \rangle$ 54)

nonterminal *do-binds* and *do-bind***syntax**

-do-block :: *do-binds* \Rightarrow 'a
 ($\langle \langle \text{open-block notation} = \langle \text{mixfix do block} \rangle \rangle \text{do } \{ // (2 \text{ } -) // \} \rangle$ [12] 62)
-do-bind :: [pttrn, 'a] \Rightarrow *do-bind*
 ($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix do bind} \rangle \rangle - \leftarrow / - \rangle$ 13)
-do-let :: [pttrn, 'a] \Rightarrow *do-bind*
 ($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix do let} \rangle \rangle \text{let } - = / - \rangle$ [1000, 13] 13)
-do-then :: 'a \Rightarrow *do-bind* ($\langle - \rangle$ [14] 13)
-do-final :: 'a \Rightarrow *do-binds* ($\langle - \rangle$)
-do-cons :: [*do-bind*, *do-binds*] \Rightarrow *do-binds*
 ($\langle \langle \text{open-block notation} = \langle \text{infix do next} \rangle \rangle - ; / - \rangle$ [13, 12] 12)
-thenM :: ['a, 'b] \Rightarrow 'c (**infixl** $\langle \gg \rangle$ 54)

syntax (ASCII)

-do-bind :: [pttrn, 'a] \Rightarrow *do-bind*
 ($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix do bind} \rangle \rangle - < - / - \rangle$ 13)
-thenM :: ['a, 'b] \Rightarrow 'c (**infixl** $\langle > > \rangle$ 54)

syntax-consts

-do-block -do-cons -do-bind -do-then \Rightarrow *bind and*
-do-let \Rightarrow *Let*

translations

-do-block (*-do-cons* (*-do-then* t) (*-do-final* e))
 \Rightarrow *CONST bind-do* t ($\lambda \cdot$. e)
-do-block (*-do-cons* (*-do-bind* p t) (*-do-final* e))
 \Rightarrow *CONST bind-do* t (λp . e)
-do-block (*-do-cons* (*-do-let* p t) bs)
 \Rightarrow *let* p = t *in* *-do-block* bs
-do-block (*-do-cons* b (*-do-cons* c cs))
 \Rightarrow *-do-block* (*-do-cons* b (*-do-final* (*-do-block* (*-do-cons* c cs))))
-do-cons (*-do-let* p t) (*-do-final* s)
 \Rightarrow *-do-final* (*let* p = t *in* s)
-do-block (*-do-final* e) \rightarrow e
 (m \gg n) \rightarrow (m \gg ($\lambda \cdot$. n))

adhoc-overloading

bind \Rightarrow *Set.bind Predicate.bind Option.bind List.bind*

end

66 Less common functions on lists

```
theory More-List
imports Main
begin
```

definition *strip-while* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$
where

strip-while $P = \text{rev} \circ \text{dropWhile } P \circ \text{rev}$

lemma *strip-while-rev* [simp]:
strip-while $P (\text{rev } xs) = \text{rev } (\text{dropWhile } P \text{ } xs)$
by (simp add: *strip-while-def*)

lemma *strip-while-Nil* [simp]:
strip-while $P [] = []$
by (simp add: *strip-while-def*)

lemma *strip-while-append* [simp]:
 $\neg P \ x \Longrightarrow \text{strip-while } P \ (xs @ [x]) = xs @ [x]$
by (simp add: *strip-while-def*)

lemma *strip-while-append-rec* [simp]:
 $P \ x \Longrightarrow \text{strip-while } P \ (xs @ [x]) = \text{strip-while } P \ xs$
by (simp add: *strip-while-def*)

lemma *strip-while-Cons* [simp]:
 $\neg P \ x \Longrightarrow \text{strip-while } P \ (x \# xs) = x \# \text{strip-while } P \ xs$
by (induct xs rule: *rev-induct*) (simp-all add: *strip-while-def*)

lemma *strip-while-eq-Nil* [simp]:
 $\text{strip-while } P \ xs = [] \longleftrightarrow (\forall x \in \text{set } xs. P \ x)$
by (simp add: *strip-while-def*)

lemma *strip-while-eq-Cons-rec*:
 $\text{strip-while } P \ (x \# xs) = x \# \text{strip-while } P \ xs \longleftrightarrow \neg (P \ x \wedge (\forall x \in \text{set } xs. P \ x))$
by (induct xs rule: *rev-induct*) (simp-all add: *strip-while-def*)

lemma *split-strip-while-append*:
fixes $xs :: 'a \text{ list}$
obtains $ys \ zs :: 'a \text{ list}$
where $\text{strip-while } P \ xs = ys$ **and** $\forall x \in \text{set } zs. P \ x$ **and** $xs = ys @ zs$
proof (rule *that*)
show $\text{strip-while } P \ xs = \text{strip-while } P \ xs \dots$
show $\forall x \in \text{set } (\text{rev } (\text{takeWhile } P \ (\text{rev } xs))). P \ x$ **by** (simp add: *takeWhile-eq-all-conv* [symmetric])
have $\text{rev } xs = \text{rev } (\text{strip-while } P \ xs @ \text{rev } (\text{takeWhile } P \ (\text{rev } xs)))$
by (simp add: *strip-while-def*)
then show $xs = \text{strip-while } P \ xs @ \text{rev } (\text{takeWhile } P \ (\text{rev } xs))$

by (*simp only: rev-is-rev-conv*)
qed

lemma *strip-while-snoc* [*simp*]:
 $\text{strip-while } P \ (xs \ @ \ [x]) = (\text{if } P \ x \ \text{then } \text{strip-while } P \ xs \ \text{else } xs \ @ \ [x])$
 by (*simp add: strip-while-def*)

lemma *strip-while-map*:
 $\text{strip-while } P \ (\text{map } f \ xs) = \text{map } f \ (\text{strip-while } (P \circ f) \ xs)$
 by (*simp add: strip-while-def rev-map dropWhile-map*)

lemma *strip-while-dropWhile-commute*:
 $\text{strip-while } P \ (\text{dropWhile } Q \ xs) = \text{dropWhile } Q \ (\text{strip-while } P \ xs)$
proof (*induct xs*)

case *Nil*
 then show ?case
 by *simp*
 next
 case (*Cons x xs*)
 show ?case
proof (*cases* $\forall y \in \text{set } xs. P \ y$)
 case *True*
 with *dropWhile-append2* [*of rev xs*] show ?thesis
 by (*auto simp add: strip-while-def dest: set-dropWhileD*)
 next
 case *False*
 then obtain *y* where $y \in \text{set } xs$ and $\neg P \ y$
 by *blast*
 with *Cons dropWhile-append3* [*of P y rev xs*] show ?thesis
 by (*simp add: strip-while-def*)
 qed
 qed

lemma *dropWhile-strip-while-commute*:
 $\text{dropWhile } P \ (\text{strip-while } Q \ xs) = \text{strip-while } Q \ (\text{dropWhile } P \ xs)$
 by (*simp add: strip-while-dropWhile-commute*)

definition *no-leading* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$
where
 $\text{no-leading } P \ xs \longleftrightarrow (xs \neq [] \longrightarrow \neg P \ (\text{hd } xs))$

lemma *no-leading-Nil* [*iff*]:
 $\text{no-leading } P \ []$
 by (*simp add: no-leading-def*)

lemma *no-leading-Cons* [*iff*]:
 $\text{no-leading } P \ (x \ # \ xs) \longleftrightarrow \neg P \ x$
 by (*simp add: no-leading-def*)

lemma *no-leading-append* [simp]:

no-leading P ($xs @ ys$) \longleftrightarrow *no-leading* P $xs \wedge (xs = [] \longrightarrow \text{no-leading } P \text{ } ys)$
by (induct xs) simp-all

lemma *no-leading-dropWhile* [simp]:

no-leading P ($\text{dropWhile } P \text{ } xs$)
by (induct xs) simp-all

lemma *dropWhile-eq-obtain-leading*:

assumes $\text{dropWhile } P \text{ } xs = ys$

obtains zs **where** $xs = zs @ ys$ **and** $\bigwedge z. z \in \text{set } zs \implies P \text{ } z$ **and** *no-leading* P ys

proof –

from *assms* **have** $\exists zs. xs = zs @ ys \wedge (\forall z \in \text{set } zs. P \text{ } z) \wedge \text{no-leading } P \text{ } ys$

proof (induct xs arbitrary: ys)

case *Nil* **then show** ?case **by** simp

next

case (*Cons* $x \text{ } xs \text{ } ys$)

show ?case **proof** (cases $P \text{ } x$)

case *True* **with** Cons.hyps [of ys] Cons.prem s

have $\exists zs. xs = zs @ ys \wedge (\forall a \in \text{set } zs. P \text{ } a) \wedge \text{no-leading } P \text{ } ys$

by simp

then obtain zs **where** $xs = zs @ ys$ **and** $\bigwedge z. z \in \text{set } zs \implies P \text{ } z$

and *: *no-leading* $P \text{ } ys$

by blast

with *True* **have** $x \# xs = (x \# zs) @ ys$ **and** $\bigwedge z. z \in \text{set } (x \# zs) \implies P \text{ } z$

by auto

with * **show** ?thesis

by blast **next**

case *False*

with Cons **show** ?thesis **by** (cases ys) simp-all

qed

qed

with *that* **show** *thesis*

by blast

qed

lemma *dropWhile-idem-iff*:

$\text{dropWhile } P \text{ } xs = xs \longleftrightarrow \text{no-leading } P \text{ } xs$

by (cases xs) (auto elim: *dropWhile-eq-obtain-leading*)

abbreviation *no-trailing* :: ($'a \Rightarrow \text{bool}$) $\Rightarrow 'a \text{ list} \Rightarrow \text{bool}$

where

no-trailing $P \text{ } xs \equiv \text{no-leading } P \text{ } (\text{rev } xs)$

lemma *no-trailing-unfold*:

$\text{no-trailing } P \text{ } xs \longleftrightarrow (xs \neq [] \longrightarrow \neg P \text{ } (\text{last } xs))$

by (*induct xs*) *simp-all*

lemma *no-trailing-Nil* [*iff*]:

no-trailing P []

by *simp*

lemma *no-trailing-Cons* [*simp*]:

no-trailing P (x # xs) \longleftrightarrow no-trailing P xs \wedge (xs = [] \longrightarrow \neg P x)

by *simp*

lemma *no-trailing-append*:

no-trailing P (xs @ ys) \longleftrightarrow no-trailing P ys \wedge (ys = [] \longrightarrow no-trailing P xs)

by (*induct xs*) *simp-all*

lemma *no-trailing-append-Cons* [*simp*]:

no-trailing P (xs @ y # ys) \longleftrightarrow no-trailing P (y # ys)

by *simp*

lemma *no-trailing-strip-while* [*simp*]:

no-trailing P (strip-while P xs)

by (*induct xs rule: rev-induct*) *simp-all*

lemma *strip-while-idem* [*simp*]:

no-trailing P xs \implies strip-while P xs = xs

by (*cases xs rule: rev-cases*) *simp-all*

lemma *strip-while-eq-obtain-trailing*:

assumes *strip-while P xs = ys*

obtains *zs where xs = ys @ zs and $\bigwedge z. z \in \text{set } zs \implies P z$ and no-trailing P*
ys

proof –

from *assms have* *rev (rev (dropWhile P (rev xs))) = rev ys*

by (*simp add: strip-while-def*)

then have *dropWhile P (rev xs) = rev ys*

by *simp*

then obtain *zs where A: rev xs = zs @ rev ys and B: $\bigwedge z. z \in \text{set } zs \implies P z$*

and *C: no-trailing P ys*

using *dropWhile-eq-obtain-leading* **by** *blast*

from *A have* *rev (rev xs) = rev (zs @ rev ys)*

by *simp*

then have *xs = ys @ rev zs*

by *simp*

moreover from *B have* *$\bigwedge z. z \in \text{set } (rev zs) \implies P z$*

by *simp*

ultimately show *thesis using that C* **by** *blast*

qed

lemma *strip-while-idem-iff*:

strip-while P xs = xs \longleftrightarrow no-trailing P xs

proof –

define *ys* **where** $ys = rev\ xs$
 moreover have $strip_while\ P\ (rev\ ys) = rev\ ys \longleftrightarrow no_trailing\ P\ (rev\ ys)$
 by (*simp add: dropWhile-idem-iff*)
 ultimately show *?thesis* **by** *simp*
qed

lemma *no-trailing-map*:

$no_trailing\ P\ (map\ f\ xs) \longleftrightarrow no_trailing\ (P \circ f)\ xs$
by (*simp add: last-map no-trailing-unfold*)

lemma *no-trailing-drop* [*simp*]:

$no_trailing\ P\ (drop\ n\ xs)$ **if** $no_trailing\ P\ xs$

proof –

from that have $no_trailing\ P\ (take\ n\ xs @ drop\ n\ xs)$
 by *simp*
 then show *?thesis*
 by (*simp only: no-trailing-append*)
qed

lemma *no-trailing-upt* [*simp*]:

$no_trailing\ P\ [n..<m] \longleftrightarrow (n < m \longrightarrow \neg P\ (m - 1))$
by (*auto simp add: no-trailing-unfold*)

definition *nth-default* :: $'a \Rightarrow 'a\ list \Rightarrow nat \Rightarrow 'a$

where

$nth_default\ dflt\ xs\ n = (if\ n < length\ xs\ then\ xs\ !\ n\ else\ dflt)$

lemma *nth-default-nth*:

$n < length\ xs \implies nth_default\ dflt\ xs\ n = xs\ !\ n$
by (*simp add: nth-default-def*)

lemma *nth-default-beyond*:

$length\ xs \leq n \implies nth_default\ dflt\ xs\ n = dflt$
by (*simp add: nth-default-def*)

lemma *nth-default-Nil* [*simp*]:

$nth_default\ dflt\ []\ n = dflt$
by (*simp add: nth-default-def*)

lemma *nth-default-Cons*:

$nth_default\ dflt\ (x \# xs)\ n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ n' \Rightarrow nth_default\ dflt\ xs\ n')$
by (*simp add: nth-default-def split: nat.split*)

lemma *nth-default-Cons-0* [*simp*]:

$nth_default\ dflt\ (x \# xs)\ 0 = x$
by (*simp add: nth-default-Cons*)

lemma *nth-default-Cons-Suc* [simp]:

nth-default dflt (x # xs) (Suc n) = nth-default dflt xs n

by (simp add: *nth-default-Cons*)

lemma *nth-default-replicate-dflt* [simp]:

nth-default dflt (replicate n dflt) m = dflt

by (simp add: *nth-default-def*)

lemma *nth-default-append*:

nth-default dflt (xs @ ys) n =

(if n < length xs then nth xs n else nth-default dflt ys (n - length xs))

by (auto simp add: *nth-default-def nth-append*)

lemma *nth-default-append-trailing* [simp]:

nth-default dflt (xs @ replicate n dflt) = nth-default dflt xs

by (simp add: *fun-eq-iff nth-default-append*) (simp add: *nth-default-def*)

lemma *nth-default-snoc-default* [simp]:

nth-default dflt (xs @ [dflt]) = nth-default dflt xs

by (auto simp add: *nth-default-def fun-eq-iff nth-append*)

lemma *nth-default-eq-dflt-iff*:

nth-default dflt xs k = dflt \longleftrightarrow (k < length xs \longrightarrow xs ! k = dflt)

by (simp add: *nth-default-def*)

lemma *nth-default-take-eq*:

nth-default dflt (take m xs) n =

(if n < m then nth-default dflt xs n else dflt)

by (simp add: *nth-default-def*)

lemma *in-enumerate-iff-nth-default-eq*:

x \neq dflt \implies (n, x) \in set (enumerate 0 xs) \longleftrightarrow nth-default dflt xs n = x

by (auto simp add: *nth-default-def in-set-conv-nth enumerate-eq-zip*)

lemma *last-conv-nth-default*:

assumes *xs \neq []*

shows *last xs = nth-default dflt xs (length xs - 1)*

using *assms* **by** (simp add: *nth-default-def last-conv-nth*)

lemma *nth-default-map-eq*:

f dflt' = dflt \implies nth-default dflt (map f xs) n = f (nth-default dflt' xs n)

by (simp add: *nth-default-def*)

lemma *finite-nth-default-neq-default* [simp]:

finite {k. nth-default dflt xs k \neq dflt}

by (simp add: *nth-default-def*)

lemma *sorted-list-of-set-nth-default*:

sorted-list-of-set {k. nth-default dflt xs k \neq dflt} = map fst (filter (λ (-, x). x \neq

dflt) (*enumerate 0 xs*)
by (*rule sorted-distinct-set-unique*) (*auto simp add: nth-default-def in-set-conv-nth*
sorted-filter distinct-map-filter enumerate-eq-zip intro: rev-image-eqI)

lemma *map-nth-default*:

map (*nth-default x xs*) [*0..<length xs*] = *xs*

proof –

have *: *map* (*nth-default x xs*) [*0..<length xs*] = *map* (*List.nth xs*) [*0..<length xs*]

by (*rule map-cong*) (*simp-all add: nth-default-nth*)

show ?thesis **by** (*simp add: * map-nth*)

qed

lemma *range-nth-default* [*simp*]:

range (*nth-default dflt xs*) = *insert dflt (set xs)*

by (*auto simp add: nth-default-def [abs-def] in-set-conv-nth*)

lemma *nth-strip-while*:

assumes *n < length (strip-while P xs)*

shows *strip-while P xs ! n = xs ! n*

proof –

have *length (dropWhile P (rev xs)) + length (takeWhile P (rev xs)) = length xs*

by (*subst add.commute*)

(*simp add: arg-cong [where f=length, OF takeWhile-dropWhile-id, unfolded length-append]*)

then show ?thesis **using** *assms*

by (*simp add: strip-while-def rev-nth dropWhile-nth*)

qed

lemma *length-strip-while-le*:

length (strip-while P xs) ≤ length xs

unfolding *strip-while-def o-def length-rev*

by (*subst (2) length-rev[symmetric]*)

(*simp add: strip-while-def length-dropWhile-le del: length-rev*)

lemma *nth-default-strip-while-dflt* [*simp*]:

nth-default dflt (strip-while ((=) dflt) xs) = nth-default dflt xs

by (*induct xs rule: rev-induct*) *auto*

lemma *nth-default-eq-iff*:

nth-default dflt xs = nth-default dflt ys

\longleftrightarrow *strip-while (HOL.eq dflt) xs = strip-while (HOL.eq dflt) ys* (**is** ?*P* \longleftrightarrow ?*Q*)

proof

let ?*strip-while* = *strip-while (HOL.eq dflt)*

let ?*xs* = ?*strip-while xs*

let ?*ys* = ?*strip-while ys*

assume ?*P*

then have *eq: nth-default dflt ?xs = nth-default dflt ?ys*

```

    by simp
  have len: length ?xs = length ?ys
  proof (rule ccontr)
    assume neq:  $\neg$  ?thesis
    { fix xs ys :: 'a list
      let ?xs = ?strip-while xs
      let ?ys = ?strip-while ys
      assume eq: nth-default dflt ?xs = nth-default dflt ?ys
      assume len: length ?xs < length ?ys
      then have length ?ys > 0 by arith
      then have ?ys  $\neq$  [] by simp
      with last-conv-nth-default [of ?ys dflt]
      have last ?ys = nth-default dflt ?ys (length ?ys - 1)
        by auto
      moreover from  $\langle ?ys \neq [] \rangle$  no-trailing-strip-while [of HOL.eq dflt ys]
        have last ?ys  $\neq$  dflt by (simp add: no-trailing-unfold)
      ultimately have nth-default dflt ?xs (length ?ys - 1)  $\neq$  dflt
        using eq by simp
      moreover from len have length ?ys - 1  $\geq$  length ?xs by simp
      ultimately have False by (simp only: nth-default-beyond) simp
    }
    from this [of xs ys] this [of ys xs] neq eq show False
    by (auto simp only: linorder-class.neq-iff)
  qed
  then show ?Q
  proof (rule nth-equalityI [rule-format])
    fix n
    assume n:  $n < \text{length } ?xs$ 
    with len have n < length ?ys
      by simp
    with n have xs: nth-default dflt ?xs n = ?xs ! n
      and ys: nth-default dflt ?ys n = ?ys ! n
      by (simp-all only: nth-default-nth)
    with eq show ?xs ! n = ?ys ! n
      by simp
  qed
  next
  assume ?Q
  then have nth-default dflt (strip-while (HOL.eq dflt) xs) = nth-default dflt
    (strip-while (HOL.eq dflt) ys)
    by simp
  then show ?P
    by simp
  qed

lemma nth-default-map2:
   $\langle \text{nth-default } d \text{ (map2 } f \text{ xs ys) } n = f \text{ (nth-default } d1 \text{ xs } n) \text{ (nth-default } d2 \text{ ys } n) \rangle$ 
  if  $\langle \text{length } xs = \text{length } ys \rangle$  and  $\langle f \text{ } d1 \text{ } d2 = d \rangle$  for bs cs
  using that proof (induction xs ys arbitrary: n rule: list-induct2)

```

```

    case Nil
    then show ?case
      by simp
next
  case (Cons x xs y ys)
  then show ?case
    by (cases n) simp-all
qed

end

```

```

theory Cancellation
imports Main
begin

```

```

named-theorems cancelation-simproc-pre ‹These theorems are here to normalise
the term. Special
  handling of constructors should be here. Remark that only the simproc @{term
NO-MATCH} is also
  included.›

```

```

named-theorems cancelation-simproc-post ‹These theorems are here to normalise
the term, after the
  cancelation simproc. Normalisation of ‹iterate-add› back to the normale repre-
sentation
  should be put here.›

```

```

named-theorems cancelation-simproc-eq-elim ‹These theorems are here to help
deriving contradiction
  (e.g., ‹Suc - = 0›).›

```

```

definition iterate-add :: ‹nat  $\Rightarrow$  'a::cancel-comm-monoid-add  $\Rightarrow$  'a› where
  ‹iterate-add n a = (((+) a)  $\frown$  n) 0›

```

```

lemma iterate-add-simps[simp]:
  ‹iterate-add 0 a = 0›
  ‹iterate-add (Suc n) a = a + iterate-add n a›
  unfolding iterate-add-def by auto

```

```

lemma iterate-add-empty[simp]: ‹iterate-add n 0 = 0›
  unfolding iterate-add-def by (induction n) auto

```

```

lemma iterate-add-distrib[simp]: ‹iterate-add (m+n) a = iterate-add m a + iter-
ate-add n a›
  by (induction n) (auto simp: ac-simps)

```

```

lemma iterate-add-Numeral1: ‹iterate-add n Numeral1 = of-nat n›
  by (induction n) auto

```

lemma *iterate-add-1*: $\langle \text{iterate-add } n \ 1 = \text{of-nat } n \rangle$
using *iterate-add-Numeral1* **by** *auto*

lemma *iterate-add-eq-add-iff1*:
 $\langle i \leq j \implies (\text{iterate-add } j \ u + m = \text{iterate-add } i \ u + n) = (\text{iterate-add } (j - i) \ u + m = n) \rangle$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

lemma *iterate-add-eq-add-iff2*:
 $\langle i \leq j \implies (\text{iterate-add } i \ u + m = \text{iterate-add } j \ u + n) = (m = \text{iterate-add } (j - i) \ u + n) \rangle$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

lemma *iterate-add-less-iff1*:
 $j \leq (i::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le}\}) + m < \text{iterate-add } j \ u + n) = (\text{iterate-add } (i - j) \ u + m < n)$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

lemma *iterate-add-less-iff2*:
 $i \leq (j::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le}\}) + m < \text{iterate-add } j \ u + n) = (m < \text{iterate-add } (j - i) \ u + n)$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

lemma *iterate-add-less-eq-iff1*:
 $j \leq (i::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le}\}) + m \leq \text{iterate-add } j \ u + n) = (\text{iterate-add } (i - j) \ u + m \leq n)$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

lemma *iterate-add-less-eq-iff2*:
 $i \leq (j::\text{nat}) \implies (\text{iterate-add } i \ (u::'a :: \{\text{cancel-comm-monoid-add, ordered-ab-semigroup-add-imp-le}\}) + m \leq \text{iterate-add } j \ u + n) = (m \leq \text{iterate-add } (j - i) \ u + n)$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

lemma *iterate-add-add-eq1*:
 $j \leq (i::\text{nat}) \implies ((\text{iterate-add } i \ u + m) - (\text{iterate-add } j \ u + n)) = ((\text{iterate-add } (i - j) \ u + m) - n)$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

lemma *iterate-add-diff-add-eq2*:
 $i \leq (j::\text{nat}) \implies ((\text{iterate-add } i \ u + m) - (\text{iterate-add } j \ u + n)) = (m - (\text{iterate-add } (j - i) \ u + n))$
by (*auto dest!*: *le-Suc-ex add-right-imp-eq simp: ab-semigroup-add-class.add-ac(1)*)

Simproc Set-Up

ML-file $\langle \text{Cancellation/cancel.ML} \rangle$

ML-file $\langle \text{Cancellation/cancel-data.ML} \rangle$

ML-file $\langle \text{Cancellation/cancel-simprocs.ML} \rangle$

end

67 (Finite) Multisets

```
theory Multiset
  imports Cancellation
begin
```

67.1 The type of multisets

```
typedef 'a multiset = ⟨{f :: 'a ⇒ nat. finite {x. f x > 0}}⟩
  morphisms count Abs-multiset
proof
  show ⟨(λx. 0::nat) ∈ {f. finite {x. f x > 0}}⟩
    by simp
qed
```

setup-lifting type-definition-multiset

```
lemma count-Abs-multiset:
  ⟨count (Abs-multiset f) = f⟩ if ⟨finite {x. f x > 0}⟩
  by (rule Abs-multiset-inverse) (simp add: that)
```

```
lemma multiset-eq-iff: M = N ⟷ (∀ a. count M a = count N a)
  by (simp only: count-inject [symmetric] fun-eq-iff)
```

```
lemma multiset-eqI: (⋀x. count A x = count B x) ⟹ A = B
  using multiset-eq-iff by auto
```

Preservation of the representing set *multiset*.

```
lemma diff-preserves-multiset:
  ⟨finite {x. 0 < M x - N x}⟩ if ⟨finite {x. 0 < M x}⟩ for M N :: 'a ⇒ nat
  using that by (rule rev-finite-subset) auto
```

```
lemma filter-preserves-multiset:
  ⟨finite {x. 0 < (if P x then M x else 0)}⟩ if ⟨finite {x. 0 < M x}⟩ for M N ::
  'a ⇒ nat
  using that by (rule rev-finite-subset) auto
```

```
lemmas in-multiset = diff-preserves-multiset filter-preserves-multiset
```

67.2 Representing multisets

Multiset enumeration

```
instantiation multiset :: (type) cancel-comm-monoid-add
begin
```

```
lift-definition zero-multiset :: 'a multiset
```

```

is  $\langle \lambda a. 0 \rangle$ 
by simp

abbreviation empty-mset ::  $\langle 'a \text{ multiset} \rangle (\langle \{\#\} \rangle)$ 
where  $\langle \text{empty-mset} \equiv 0 \rangle$ 

lift-definition plus-multiset ::  $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ 
is  $\langle \lambda M N a. M a + N a \rangle$ 
by simp

lift-definition minus-multiset ::  $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ 
is  $\langle \lambda M N a. M a - N a \rangle$ 
by (rule diff-preserves-multiset)

instance
by (standard; transfer) (simp-all add: fun-eq-iff)

end

context
begin

qualified definition is-empty ::  $'a \text{ multiset} \Rightarrow \text{bool}$  where
  [code-abbrev]: is-empty  $A \longleftrightarrow A = \{\#\}$ 

end

lemma add-mset-in-multiset:
   $\langle \text{finite } \{x. 0 < (\text{if } x = a \text{ then } \text{Suc } (M x) \text{ else } M x)\} \rangle$ 
if  $\langle \text{finite } \{x. 0 < M x\} \rangle$ 
using that by (simp add: flip: insert-Collect)

lift-definition add-mset ::  $'a \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$  is
   $\lambda a M b. \text{if } b = a \text{ then } \text{Suc } (M b) \text{ else } M b$ 
by (rule add-mset-in-multiset)

syntax
  -multiset ::  $\text{args} \Rightarrow 'a \text{ multiset}$  ( $\langle (\langle \text{indent}=2 \text{ notation}=\langle \text{mixfix multiset enumeration} \rangle \{\#\text{-}\#\} \rangle) \rangle$ )
syntax-consts
  -multiset  $\rightleftharpoons$  add-mset

translations
   $\{\#\text{x}, \text{x}\#\} == \text{CONST add-mset } x \ \{\#\text{x}\#\}$ 
   $\{\#\text{x}\#\} == \text{CONST add-mset } x \ \{\#\}$ 

lemma count-empty [simp]:  $\text{count } \{\#\} a = 0$ 
by (simp add: zero-multiset.rep-eq)

lemma count-add-mset [simp]:

```


count (*add-mset* *b* *A*) *a* = (if *b* = *a* then *Suc* (*count* *A* *a*) else *count* *A* *a*)
by (*simp add: add-mset.rep-eq*)

lemma *count-single*: *count* {*#b#*} *a* = (if *b* = *a* then 1 else 0)
by *simp*

lemma
add-mset-not-empty [*simp*]: $\langle \text{add-mset } a \ A \neq \{ \# \} \rangle$ **and**
empty-not-add-mset [*simp*]: $\{ \# \} \neq \text{add-mset } a \ A$
by (*auto simp: multiset-eq-iff*)

lemma *add-mset-add-mset-same-iff* [*simp*]:
add-mset *a* *A* = *add-mset* *a* *B* \longleftrightarrow *A* = *B*
by (*auto simp: multiset-eq-iff*)

lemma *add-mset-commute*:
add-mset *x* (*add-mset* *y* *M*) = *add-mset* *y* (*add-mset* *x* *M*)
by (*auto simp: multiset-eq-iff*)

67.3 Basic operations

67.3.1 Conversion to set and membership

definition *set-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ set} \rangle$
where $\langle \text{set-mset } M = \{x. \text{count } M \ x > 0\} \rangle$

abbreviation *member-mset* :: $\langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$
where $\langle \text{member-mset } a \ M \equiv a \in \text{set-mset } M \rangle$

notation
member-mset ($\langle '(\in \#)' \rangle$) **and**
member-mset ($\langle (\langle \text{notation} = \langle \text{infix } \in \# \rangle \rangle - / \in \# -) \rangle$ [50, 51] 50)

notation (*ASCII*)
member-mset ($\langle '(:\#)' \rangle$) **and**
member-mset ($\langle (\langle \text{notation} = \langle \text{infix } :\# \rangle \rangle - / :\# -) \rangle$ [50, 51] 50)

abbreviation *not-member-mset* :: $\langle 'a \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool} \rangle$
where $\langle \text{not-member-mset } a \ M \equiv a \notin \text{set-mset } M \rangle$

notation
not-member-mset ($\langle '(\notin \#)' \rangle$) **and**
not-member-mset ($\langle (\langle \text{notation} = \langle \text{infix } \notin \# \rangle \rangle - / \notin \# -) \rangle$ [50, 51] 50)

notation (*ASCII*)
not-member-mset ($\langle '(\sim \#)' \rangle$) **and**
not-member-mset ($\langle (\langle \text{notation} = \langle \text{infix } \sim \# \rangle \rangle - / \sim \# -) \rangle$ [50, 51] 50)

context
begin

qualified abbreviation $Ball :: 'a \text{ multiset} \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where $Ball \ M \equiv Set.Ball \ (set-mset \ M)$

qualified abbreviation $Bex :: 'a \text{ multiset} \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where $Bex \ M \equiv Set.Bex \ (set-mset \ M)$

end

syntax

- $MBall \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \forall \rangle \forall -\in\#-./ - \rangle \rangle [0, 0, 10] \ 10)$
 - $MBex \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \exists \rangle \exists -\in\#-./ - \rangle \rangle [0, 0, 10] \ 10)$

syntax (ASCII)

- $MBall \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \forall \rangle \forall -\in\#-./ - \rangle \rangle [0, 0, 10] \ 10)$
 - $MBex \quad :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow bool$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \exists \rangle \exists -\in\#-./ - \rangle \rangle [0, 0, 10] \ 10)$

syntax-consts

- $MBall \equiv Multiset.Ball$ **and**
 - $MBex \equiv Multiset.Bex$

translations

$\forall x \in \#A. P \equiv CONST \ Multiset.Ball \ A \ (\lambda x. P)$
 $\exists x \in \#A. P \equiv CONST \ Multiset.Bex \ A \ (\lambda x. P)$

typed-print-translation \langle

$[(\text{const-syntax} \langle Multiset.Ball \rangle, \text{Syntax-Trans.preserve-binder-abs2-tr}' \text{syntax-const} \langle -MBall \rangle),$
 $(\text{const-syntax} \langle Multiset.Bex \rangle, \text{Syntax-Trans.preserve-binder-abs2-tr}' \text{syntax-const} \langle -MBex \rangle)]$
 \rangle — to avoid eta-contraction of body

lemma *count-eq-zero-iff*:

$count \ M \ x = 0 \longleftrightarrow x \notin \# \ M$
by (*auto simp add: set-mset-def*)

lemma *not-in-iff*:

$x \notin \# \ M \longleftrightarrow count \ M \ x = 0$
by (*auto simp add: count-eq-zero-iff*)

lemma *count-greater-zero-iff* [*simp*]:

$count \ M \ x > 0 \longleftrightarrow x \in \# \ M$
by (*auto simp add: set-mset-def*)

lemma *count-inI*:

assumes $count \ M \ x = 0 \implies False$
shows $x \in \# \ M$

proof (*rule ccontr*)

assume $x \notin \# \ M$
with *assms* **show** $False$ **by** (*simp add: not-in-iff*)

qed

lemma *in-countE*:

assumes $x \in\# M$

obtains n **where** $\text{count } M \ x = \text{Suc } n$

proof –

from *assms* **have** $\text{count } M \ x > 0$ **by** *simp*

then obtain n **where** $\text{count } M \ x = \text{Suc } n$

using *gr0-conv-Suc* **by** *blast*

with that show *thesis* .

qed

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:

$\text{count } M \ x \geq \text{Suc } 0 \longleftrightarrow x \in\# M$

by (*simp add: Suc-le-eq*)

lemma *count-greater-eq-one-iff* [*simp*]:

$\text{count } M \ x \geq 1 \longleftrightarrow x \in\# M$

by *simp*

lemma *set-mset-empty* [*simp*]:

$\text{set-mset } \{\#\} = \{\}$

by (*simp add: set-mset-def*)

lemma *set-mset-single*:

$\text{set-mset } \{\#b\# \} = \{b\}$

by (*simp add: set-mset-def*)

lemma *set-mset-eq-empty-iff* [*simp*]:

$\text{set-mset } M = \{\} \longleftrightarrow M = \{\#\}$

by (*auto simp add: multiset-eq-iff count-eq-zero-iff*)

lemma *finite-set-mset* [*iff*]:

finite (*set-mset* M)

using *count [of M]* **by** *simp*

lemma *set-mset-add-mset-insert* [*simp*]: $\langle \text{set-mset } (\text{add-mset } a \ A) = \text{insert } a \ (\text{set-mset } A) \rangle$

by (*auto simp flip: count-greater-eq-Suc-zero-iff split: if-splits*)

lemma *multiset-nonemptyE* [*elim*]:

assumes $A \neq \{\#\}$

obtains x **where** $x \in\# A$

proof –

have $\exists x. x \in\# A$ **by** (*rule ccontr*) (*insert assms, auto*)

with that show *?thesis* **by** *blast*

qed

lemma *count-gt-imp-in-mset*: $\text{count } M \ x > n \implies x \in\# M$

using *count-greater-zero-iff* by *fastforce*

67.3.2 Union

lemma *count-union* [*simp*]:
 $\text{count } (M + N) \ a = \text{count } M \ a + \text{count } N \ a$
 by (*simp add: plus-multiset.rep-eq*)

lemma *set-mset-union* [*simp*]:
 $\text{set-mset } (M + N) = \text{set-mset } M \cup \text{set-mset } N$
 by (*simp only: set-eq-iff count-greater-zero-iff [symmetric] count-union*) *simp*

lemma *union-mset-add-mset-left* [*simp*]:
 $\text{add-mset } a \ A + B = \text{add-mset } a \ (A + B)$
 by (*auto simp: multiset-eq-iff*)

lemma *union-mset-add-mset-right* [*simp*]:
 $A + \text{add-mset } a \ B = \text{add-mset } a \ (A + B)$
 by (*auto simp: multiset-eq-iff*)

lemma *add-mset-add-single*: $\langle \text{add-mset } a \ A = A + \{\#a\# \} \rangle$
 by (*subst union-mset-add-mset-right, subst add.comm-neutral*) *standard*

67.3.3 Difference

instance *multiset* :: (type) *comm-monoid-diff*
 by *standard* (*transfer; simp add: fun-eq-iff*)

lemma *count-diff* [*simp*]:
 $\text{count } (M - N) \ a = \text{count } M \ a - \text{count } N \ a$
 by (*simp add: minus-multiset.rep-eq*)

lemma *add-mset-diff-bothsides*:
 $\langle \text{add-mset } a \ M - \text{add-mset } a \ A = M - A \rangle$
 by (*auto simp: multiset-eq-iff*)

lemma *in-diff-count*:
 $a \in\# M - N \longleftrightarrow \text{count } N \ a < \text{count } M \ a$
 by (*simp add: set-mset-def*)

lemma *count-in-diffI*:
 assumes $\bigwedge n. \text{count } N \ x = n + \text{count } M \ x \implies \text{False}$
 shows $x \in\# M - N$
proof (*rule ccontr*)
 assume $x \notin\# M - N$
 then have $\text{count } N \ x = (\text{count } N \ x - \text{count } M \ x) + \text{count } M \ x$
 by (*simp add: in-diff-count not-less*)
 with *assms* show *False* by *auto*
qed

lemma *in-diff-countE*:
 assumes $x \in \# M - N$
 obtains n where $\text{count } M x = \text{Suc } n + \text{count } N x$
proof –
 from *assms* have $\text{count } M x - \text{count } N x > 0$ **by** (*simp add: in-diff-count*)
 then have $\text{count } M x > \text{count } N x$ **by** *simp*
 then obtain n where $\text{count } M x = \text{Suc } n + \text{count } N x$
 using *less-iff-Suc-add* **by** *auto*
 with *that* **show** *thesis* .
qed

lemma *in-diffD*:
 assumes $a \in \# M - N$
 shows $a \in \# M$
proof –
 have $0 \leq \text{count } N a$ **by** *simp*
 also from *assms* have $\text{count } N a < \text{count } M a$
by (*simp add: in-diff-count*)
 finally **show** *?thesis* **by** *simp*
qed

lemma *set-mset-diff*:
 $\text{set-mset } (M - N) = \{a. \text{count } N a < \text{count } M a\}$
by (*simp add: set-mset-def*)

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
by *rule* (*fact Groups.diff-zero, fact Groups.zero-diff*)

lemma *diff-cancel*: $A - A = \{\#\}$
by (*fact Groups.diff-cancel*)

lemma *diff-union-cancelR*: $M + N - N = (M :: 'a \text{ multiset})$
by (*fact add-diff-cancel-right*)

lemma *diff-union-cancelL*: $N + M - N = (M :: 'a \text{ multiset})$
by (*fact add-diff-cancel-left*)

lemma *diff-right-commute*:
 fixes $M N Q :: 'a \text{ multiset}$
 shows $M - N - Q = M - Q - N$
by (*fact diff-right-commute*)

lemma *diff-add*:
 fixes $M N Q :: 'a \text{ multiset}$
 shows $M - (N + Q) = M - N - Q$
by (*rule sym*) (*fact diff-diff-add*)

lemma *insert-DiffM* [*simp*]: $x \in \# M \implies \text{add-mset } x (M - \{\#x\}) = M$

by (*clarsimp simp: multiset-eq-iff*)

lemma *insert-DiffM2*: $x \in\# M \implies (M - \{\#x\}) + \{\#x\} = M$
by *simp*

lemma *diff-union-swap*: $a \neq b \implies \text{add-mset } b (M - \{\#a\}) = \text{add-mset } b M - \{\#a\}$
by (*auto simp add: multiset-eq-iff*)

lemma *diff-add-mset-swap* [*simp*]: $b \notin\# A \implies \text{add-mset } b M - A = \text{add-mset } b (M - A)$
by (*auto simp add: multiset-eq-iff simp: not-in-iff*)

lemma *diff-union-swap2* [*simp*]: $y \in\# M \implies \text{add-mset } x M - \{\#y\} = \text{add-mset } x (M - \{\#y\})$
by (*metis add-mset-diff-both-sides diff-union-swap diff-zero insert-DiffM*)

lemma *diff-diff-add-mset* [*simp*]: $(M::'a \text{ multiset}) - N - P = M - (N + P)$
by (*rule diff-diff-add*)

lemma *diff-union-single-conv*:
 $a \in\# J \implies I + J - \{\#a\} = I + (J - \{\#a\})$
by (*simp add: multiset-eq-iff Suc-le-eq*)

lemma *mset-add* [*elim?*]:
assumes $a \in\# A$
obtains B **where** $A = \text{add-mset } a B$
proof –
from *assms* **have** $A = \text{add-mset } a (A - \{\#a\})$
by *simp*
with *that* **show** *thesis* .
qed

lemma *union-iff*:
 $a \in\# A + B \iff a \in\# A \vee a \in\# B$
by *auto*

lemma *count-minus-inter-lt-count-minus-inter-iff*:
 $\text{count } (M2 - M1) y < \text{count } (M1 - M2) y \iff y \in\# M1 - M2$
by (*meson count-greater-zero-iff gr-implies-not-zero in-diff-count leI order.strict-trans2 order-less-asym*)

lemma *minus-inter-eq-minus-inter-iff*:
 $(M1 - M2) = (M2 - M1) \iff \text{set-mset } (M1 - M2) = \text{set-mset } (M2 - M1)$
by (*metis add.commute count-diff count-eq-zero-iff diff-add-zero in-diff-countE multiset-eq-iff*)

67.3.4 Min and Max

abbreviation *Min-mset* :: 'a::linorder multiset \Rightarrow 'a **where**
Min-mset $m \equiv \text{Min } (\text{set-mset } m)$

abbreviation *Max-mset* :: 'a::linorder multiset \Rightarrow 'a **where**
Max-mset $m \equiv \text{Max } (\text{set-mset } m)$

lemma

Min-in-mset: $M \neq \{\#\} \implies \text{Min-mset } M \in\# M$ **and**

Max-in-mset: $M \neq \{\#\} \implies \text{Max-mset } M \in\# M$

by *simp+*

67.3.5 Equality of multisets

lemma *single-eq-single* [*simp*]: $\{\#a\# \} = \{\#b\# \} \longleftrightarrow a = b$

by (*auto simp add: multiset-eq-iff*)

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$

by (*auto simp add: multiset-eq-iff*)

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$

by (*auto simp add: multiset-eq-iff*)

lemma *multi-self-add-other-not-self* [*simp*]: $M = \text{add-mset } x M \longleftrightarrow \text{False}$

by (*auto simp add: multiset-eq-iff*)

lemma *add-mset-remove-trivial* [*simp*]: $\langle \text{add-mset } x M - \{\#x\# \} = M \rangle$

by (*auto simp: multiset-eq-iff*)

lemma *diff-single-trivial*: $\neg x \in\# M \implies M - \{\#x\# \} = M$

by (*auto simp add: multiset-eq-iff not-in-iff*)

lemma *diff-single-eq-union*: $x \in\# M \implies M - \{\#x\# \} = N \longleftrightarrow M = \text{add-mset } x N$

by *auto*

lemma *union-single-eq-diff*: $\text{add-mset } x M = N \implies M = N - \{\#x\# \}$

unfolding *add-mset-add-single*[*of - M*] **by** (*fact add-implies-diff*)

lemma *union-single-eq-member*: $\text{add-mset } x M = N \implies x \in\# N$

by *auto*

lemma *add-mset-remove-trivial-If*:

$\text{add-mset } a (N - \{\#a\# \}) = (\text{if } a \in\# N \text{ then } N \text{ else } \text{add-mset } a N)$

by (*simp add: diff-single-trivial*)

lemma *add-mset-remove-trivial-eq*: $\langle N = \text{add-mset } a (N - \{\#a\# \}) \longleftrightarrow a \in\# N \rangle$

by (*auto simp: add-mset-remove-trivial-If*)

lemma *union-is-single*:

$M + N = \{\#a\# \} \longleftrightarrow M = \{\#a\# \} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\# \}$
 (is ?lhs = ?rhs)

proof

show ?lhs **if** ?rhs **using** that **by** auto

show ?rhs **if** ?lhs

by (metis Multiset.diff-cancel add.commute add-diff-cancel-left' diff-add-zero
 diff-single-trivial insert-DiffM that)

qed

lemma *single-is-union*: $\{\#a\# \} = M + N \longleftrightarrow \{\#a\# \} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\# \} = N$

by (auto simp add: eq-commute [of $\{\#a\# \}$ $M + N$] union-is-single)

lemma *add-eq-conv-diff*:

$\text{add-mset } a \ M = \text{add-mset } b \ N \longleftrightarrow M = N \wedge a = b \vee M = \text{add-mset } b \ (N - \{\#a\# \}) \wedge N = \text{add-mset } a \ (M - \{\#b\# \})$
 (is ?lhs \longleftrightarrow ?rhs)

proof

show ?lhs **if** ?rhs

using that

by (auto simp add: add-mset-commute[*of* $a \ b$])

show ?rhs **if** ?lhs

proof (cases $a = b$)

case True **with** $\langle ?lhs \rangle$ **show** ?thesis **by** simp

next

case False

from $\langle ?lhs \rangle$ **have** $a \in \# \text{ add-mset } b \ N$ **by** (rule union-single-eq-member)

with False **have** $a \in \# \ N$ **by** auto

moreover from $\langle ?lhs \rangle$ **have** $M = \text{add-mset } b \ N - \{\#a\# \}$ **by** (rule union-single-eq-diff)

moreover note False

ultimately show ?thesis **by** (auto simp add: diff-right-commute [of $-\{\#a\# \}$])

qed

qed

lemma *add-mset-eq-single [iff]*: $\text{add-mset } b \ M = \{\#a\# \} \longleftrightarrow b = a \wedge M = \{\#\}$

by (auto simp: add-eq-conv-diff)

lemma *single-eq-add-mset [iff]*: $\{\#a\# \} = \text{add-mset } b \ M \longleftrightarrow b = a \wedge M = \{\#\}$

by (auto simp: add-eq-conv-diff)

lemma *insert-noteq-member*:

assumes BC: $\text{add-mset } b \ B = \text{add-mset } c \ C$

and bnotc: $b \neq c$

shows $c \in \# \ B$

proof –

have $c \in \# \ \text{add-mset } c \ C$ **by** simp

have $nc: \neg c \in\# \{ \#b\# \}$ **using** $bnotc$ **by** $simp$
then have $c \in\# \text{add-mset } b \ B$ **using** BC **by** $simp$
then show $c \in\# B$ **using** nc **by** $simp$
qed

lemma add-eq-conv-ex :
 $(\text{add-mset } a \ M = \text{add-mset } b \ N) =$
 $(M = N \wedge a = b \vee (\exists K. M = \text{add-mset } b \ K \wedge N = \text{add-mset } a \ K))$
by $(\text{auto } simp \text{ add: add-eq-conv-diff})$

lemma $\text{multi-member-split}$: $x \in\# M \implies \exists A. M = \text{add-mset } x \ A$
by $(\text{rule exI } [\text{where } x = M - \{ \#x\# \}]) \text{ simp}$

lemma $\text{multiset-add-sub-el-shuffle}$:
assumes $c \in\# B$
and $b \neq c$
shows $\text{add-mset } b \ (B - \{ \#c\# \}) = \text{add-mset } b \ B - \{ \#c\# \}$
proof –
from $\langle c \in\# B \rangle$ **obtain** A **where** $B: B = \text{add-mset } c \ A$
by $(\text{blast dest: multi-member-split})$
have $\text{add-mset } b \ A = \text{add-mset } c \ (\text{add-mset } b \ A) - \{ \#c\# \}$ **by** $simp$
then have $\text{add-mset } b \ A = \text{add-mset } b \ (\text{add-mset } c \ A) - \{ \#c\# \}$
by $(simp \text{ add: } \langle b \neq c \rangle)$
then show $?thesis$ **using** B **by** $simp$
qed

lemma $\text{add-mset-eq-singleton-iff}$ [iff]:
 $\text{add-mset } x \ M = \{ \#y\# \} \longleftrightarrow M = \{ \# \} \wedge x = y$
by $auto$

67.3.6 Pointwise ordering induced by count

definition $\text{subseteq-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\langle \subseteq\# \rangle$ 50)
where $A \subseteq\# B \longleftrightarrow (\forall a. \text{count } A \ a \leq \text{count } B \ a)$

definition $\text{subset-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\langle \subset\# \rangle$ 50)
where $A \subset\# B \longleftrightarrow A \subseteq\# B \wedge A \neq B$

abbreviation $(\text{input}) \text{supseteq-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\langle \supseteq\# \rangle$ 50)
where $\text{supseteq-mset } A \ B \equiv B \subseteq\# A$

abbreviation $(\text{input}) \text{supset-mset} :: 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$ (**infix** $\langle \supset\# \rangle$ 50)
where $\text{supset-mset } A \ B \equiv B \subset\# A$

notation (input)
 subseteq-mset (**infix** $\langle \leq\# \rangle$ 50) **and**
 supseteq-mset (**infix** $\langle \geq\# \rangle$ 50)

notation (*ASCII*)

subseq-mset (**infix** $\langle \leq \# \rangle$ 50) **and**
subset-mset (**infix** $\langle \subset \# \rangle$ 50) **and**
supseq-mset (**infix** $\langle \geq \# \rangle$ 50) **and**
supset-mset (**infix** $\langle \supset \# \rangle$ 50)

global-interpretation *subset-mset*: *ordering* $\langle (\subseteq \#) \rangle \langle (\subset \#) \rangle$

by *standard* (*auto simp add: subset-mset-def subseq-mset-def multiset-eq-iff intro: order.trans order.antisym*)

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le* $\langle (+) \rangle \langle (-) \rangle \langle (\subseteq \#) \rangle \langle (\subset \#) \rangle$

by *standard* (*auto simp add: subset-mset-def subseq-mset-def multiset-eq-iff intro: order.trans antisym*)

— FIXME: avoid junk stemming from type class interpretation

interpretation *subset-mset*: *ordered-ab-semigroup-monoid-add-imp-le* $(+) 0 (-) (\subseteq \#) (\subset \#)$

by *standard*

— FIXME: avoid junk stemming from type class interpretation

lemma *mset-subset-eqI*:

$(\bigwedge a. \text{count } A \ a \leq \text{count } B \ a) \implies A \subseteq \# B$

by (*simp add: subseq-mset-def*)

lemma *mset-subset-eq-count*:

$A \subseteq \# B \implies \text{count } A \ a \leq \text{count } B \ a$

by (*simp add: subseq-mset-def*)

lemma *mset-subset-eq-exists-conv*: $(A::'a \text{ multiset}) \subseteq \# B \longleftrightarrow (\exists C. B = A + C)$

unfolding *subseq-mset-def*

by (*metis add-diff-cancel-left' count-diff count-union le-Suc-ex le-add-same-cancel1 multiset-eq-iff zero-le*)

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff* $(+) 0 (\subseteq \#) (\subset \#) (-)$

by *standard* (*simp, fact mset-subset-eq-exists-conv*)

— FIXME: avoid junk stemming from type class interpretation

declare *subset-mset.add-diff-assoc*[*simp*] *subset-mset.add-diff-assoc2*[*simp*]

lemma *mset-subset-eq-mono-add-right-cancel*: $(A::'a \text{ multiset}) + C \subseteq \# B + C \longleftrightarrow A \subseteq \# B$

by (*fact subset-mset.add-le-cancel-right*)

lemma *mset-subset-eq-mono-add-left-cancel*: $C + (A::'a \text{ multiset}) \subseteq \# C + B \longleftrightarrow A \subseteq \# B$

by (*fact subset-mset.add-le-cancel-left*)

lemma *mset-subset-eq-mono-add*: $(A::'a \text{ multiset}) \subseteq\# B \implies C \subseteq\# D \implies A + C \subseteq\# B + D$
by (*fact subset-mset.add-mono*)

lemma *mset-subset-eq-add-left*: $(A::'a \text{ multiset}) \subseteq\# A + B$
by *simp*

lemma *mset-subset-eq-add-right*: $B \subseteq\# (A::'a \text{ multiset}) + B$
by *simp*

lemma *single-subset-iff* [*simp*]:
 $\{\#a\# \} \subseteq\# M \longleftrightarrow a \in\# M$
by (*auto simp add: subseteq-mset-def Suc-le-eq*)

lemma *mset-subset-eq-single*: $a \in\# B \implies \{\#a\# \} \subseteq\# B$
by *simp*

lemma *mset-subset-eq-add-mset-cancel*: $\langle \text{add-mset } a \ A \subseteq\# \text{add-mset } a \ B \longleftrightarrow A \subseteq\# B \rangle$
unfolding *add-mset-add-single*[*of* - *A*] *add-mset-add-single*[*of* - *B*]
by (*rule mset-subset-eq-mono-add-right-cancel*)

lemma *multiset-diff-union-assoc*:
fixes *A B C D* :: *'a multiset*
shows $C \subseteq\# B \implies A + B - C = A + (B - C)$
by (*fact subset-mset.diff-add-assoc*)

lemma *mset-subset-eq-multiset-union-diff-commute*:
fixes *A B C D* :: *'a multiset*
shows $B \subseteq\# A \implies A - B + C = A + C - B$
by (*fact subset-mset.add-diff-assoc2*)

lemma *diff-subset-eq-self*[*simp*]:
 $(M::'a \text{ multiset}) - N \subseteq\# M$
by (*simp add: subseteq-mset-def*)

lemma *mset-subset-eqD*:
assumes $A \subseteq\# B$ **and** $x \in\# A$
shows $x \in\# B$
proof –
from $\langle x \in\# A \rangle$ **have** *count A x > 0* **by** *simp*
also from $\langle A \subseteq\# B \rangle$ **have** *count A x ≤ count B x*
by (*simp add: subseteq-mset-def*)
finally show *?thesis* **by** *simp*
qed

lemma *mset-subsetD*:
 $A \subset\# B \implies x \in\# A \implies x \in\# B$

```

by (auto intro: mset-subset-eqD [of A])

lemma set-mset-mono:
   $A \subseteq\# B \implies \text{set-mset } A \subseteq \text{set-mset } B$ 
  by (metis mset-subset-eqD subsetI)

lemma mset-subset-eq-insertD:
  assumes add-mset x A  $\subseteq\# B$ 
  shows  $x \in\# B \wedge A \subset\# B$ 
proof
  show  $x \in\# B$ 
    using assms by (simp add: mset-subset-eqD)
  have  $A \subseteq\# \text{add-mset } x A$ 
    by (metis (no-types) add-mset-add-single mset-subset-eq-add-left)
  then have  $A \subset\# \text{add-mset } x A$ 
    by (meson multi-self-add-other-not-self subset-mset.le-imp-less-or-eq)
  then show  $A \subset\# B$ 
    using assms subset-mset.strict-trans2 by blast
qed

lemma mset-subset-insertD:
   $\text{add-mset } x A \subset\# B \implies x \in\# B \wedge A \subset\# B$ 
  by (rule mset-subset-eq-insertD) simp

lemma mset-subset-of-empty[simp]:  $A \subset\# \{\#\} \longleftrightarrow \text{False}$ 
  by (simp only: subset-mset.not-less-zero)

lemma empty-subset-add-mset[simp]:  $\{\#\} \subset\# \text{add-mset } x M$ 
  by (auto intro: subset-mset.gr-zeroI)

lemma empty-le:  $\{\#\} \subseteq\# A$ 
  by (fact subset-mset.zero-le)

lemma insert-subset-eq-iff:
   $\text{add-mset } a A \subseteq\# B \longleftrightarrow a \in\# B \wedge A \subseteq\# B - \{\#a\# \}$ 
  using mset-subset-eq-insertD subset-mset.le-diff-conv2 by fastforce

lemma insert-union-subset-iff:
   $\text{add-mset } a A \subset\# B \longleftrightarrow a \in\# B \wedge A \subset\# B - \{\#a\# \}$ 
  by (auto simp add: insert-subset-eq-iff subset-mset-def)

lemma subset-eq-diff-conv:
   $A - C \subseteq\# B \longleftrightarrow A \subseteq\# B + C$ 
  by (simp add: subseteq-mset-def le-diff-conv)

lemma multi-psub-of-add-self [simp]:  $A \subset\# \text{add-mset } x A$ 
  by (auto simp: subset-mset-def subseteq-mset-def)

lemma multi-psub-self:  $A \subset\# A = \text{False}$ 

```

by *simp*

lemma *mset-subset-add-mset* [*simp*]: $\text{add-mset } x \ N \subseteq\# \text{ add-mset } x \ M \longleftrightarrow N \subseteq\# M$
unfolding *add-mset-add-single*[*of - N*] *add-mset-add-single*[*of - M*]
by (*fact subset-mset.add-less-cancel-right*)

lemma *mset-subset-diff-self*: $c \in\# B \implies B - \{\#c\} \subseteq\# B$
by (*auto simp: subset-mset-def elim: mset-add*)

lemma *Diff-eq-empty-iff-mset*: $A - B = \{\#\} \longleftrightarrow A \subseteq\# B$
by (*auto simp: multiset-eq-iff subseteq-mset-def*)

lemma *add-mset-subseteq-single-iff*[*iff*]: $\text{add-mset } a \ M \subseteq\# \{\#b\} \longleftrightarrow M = \{\#\} \wedge a = b$

proof

assume *A*: $\text{add-mset } a \ M \subseteq\# \{\#b\}$
then have $\langle a = b \rangle$
by (*auto dest: mset-subset-eq-insertD*)
then show $M = \{\#\} \wedge a = b$
using *A* **by** (*simp add: mset-subset-eq-add-mset-cancel*)

qed *simp*

lemma *nonempty-subseteq-mset-eq-single*: $M \neq \{\#\} \implies M \subseteq\# \{\#x\} \implies M = \{\#x\}$
by (*cases M*) (*metis single-is-union subset-mset.less-eqE*)

lemma *nonempty-subseteq-mset-iff-single*: $(M \neq \{\#\} \wedge M \subseteq\# \{\#x\} \wedge P) \longleftrightarrow M = \{\#x\} \wedge P$
by (*cases M*) (*metis empty-not-add-mset nonempty-subseteq-mset-eq-single subset-mset.order-refl*)

67.3.7 Intersection and bounded union

definition *inter-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ (**infixl** $\langle \cap\# \rangle$ 70)
where $\langle A \cap\# B = A - (A - B) \rangle$

lemma *count-inter-mset* [*simp*]:
 $\langle \text{count } (A \cap\# B) \ x = \min (\text{count } A \ x) (\text{count } B \ x) \rangle$
by (*simp add: inter-mset-def*)

interpretation *subset-mset*: *semilattice-inf* $\langle (\cap\#) \rangle \langle (\subseteq\#) \rangle \langle (\subset\#) \rangle$
by *standard* (*simp-all add: multiset-eq-iff subseteq-mset-def*)
 — FIXME: avoid junk stemming from type class interpretation

definition *union-mset* :: $\langle 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$ (**infixl** $\langle \cup\# \rangle$)

70)

where $\langle A \cup\# B = A + (B - A) \rangle$ **lemma** *count-union-mset* [simp]: $\langle \text{count } (A \cup\# B) \ x = \max (\text{count } A \ x) (\text{count } B \ x) \rangle$ **by** (*simp add: union-mset-def*)**global-interpretation** *subset-mset: semilattice-neutr-order* $\langle (\cup\#) \rangle \langle \{\#\} \rangle \langle (\supset\#) \rangle \langle (\supset\#) \rangle$ **proof****show** $\bigwedge a \ b. (b \subseteq\# a) = (a = a \cup\# b)$ **by** (*simp add: Diff-eq-empty-iff-mset union-mset-def*)**show** $\bigwedge a \ b. (b \subset\# a) = (a = a \cup\# b \wedge a \neq b)$ **by** (*metis Diff-eq-empty-iff-mset add-cancel-left-right subset-mset-def union-mset-def*)**qed** (*auto simp: multiset-eqI union-mset-def*)**interpretation** *subset-mset: semilattice-sup* $\langle (\cup\#) \rangle \langle (\subseteq\#) \rangle \langle (\subset\#) \rangle$ **proof** —**have** [simp]: $m \leq n \implies q \leq n \implies m + (q - m) \leq n$ **for** $m \ n \ q :: \text{nat}$ **by** *arith***show** *class.semilattice-sup* $(\cup\#) (\subseteq\#) (\subset\#)$ **by** *standard (auto simp add: union-mset-def subseteq-mset-def)***qed** — *FIXME: avoid junk stemming from type class interpretation***interpretation** *subset-mset: bounded-lattice-bot* $(\cap\#) (\subseteq\#) (\subset\#)$
 $(\cup\#) \{\#\}$ **by** *standard auto*— *FIXME: avoid junk stemming from type class interpretation*

67.3.8 Additional intersection facts

lemma *set-mset-inter* [simp]: $\text{set-mset } (A \cap\# B) = \text{set-mset } A \cap \text{set-mset } B$ **by** (*simp only: set-mset-def*) *auto***lemma** *diff-intersect-left-idem* [simp]: $M - M \cap\# N = M - N$ **by** (*simp add: multiset-eq-iff min-def*)**lemma** *diff-intersect-right-idem* [simp]: $M - N \cap\# M = M - N$ **by** (*simp add: multiset-eq-iff min-def*)**lemma** *multiset-inter-single*[simp]: $a \neq b \implies \{\#a\# \} \cap\# \{\#b\# \} = \{\#\}$ **by** (*rule multiset-eqI*) *auto***lemma** *multiset-union-diff-commute*:**assumes** $B \cap\# C = \{\#\}$ **shows** $A + B - C = A - C + B$

proof (*rule multiset-eqI*)
fix x
from *assms* **have** $\min (\text{count } B \ x) (\text{count } C \ x) = 0$
by (*auto simp add: multiset-eq-iff*)
then have $\text{count } B \ x = 0 \vee \text{count } C \ x = 0$
unfolding *min-def* **by** (*auto split: if-splits*)
then show $\text{count } (A + B - C) \ x = \text{count } (A - C + B) \ x$
by *auto*
qed

lemma *disjunct-not-in*:
 $A \cap \# B = \{\#\} \longleftrightarrow (\forall a. a \notin \# A \vee a \notin \# B)$
by (*metis disjoint-iff set-mset-eq-empty-iff set-mset-inter*)

lemma *inter-mset-empty-distrib-right*: $A \cap \# (B + C) = \{\#\} \longleftrightarrow A \cap \# B = \{\#\} \wedge A \cap \# C = \{\#\}$
by (*meson disjunct-not-in union-iff*)

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap \# C = \{\#\} \longleftrightarrow A \cap \# C = \{\#\} \wedge B \cap \# C = \{\#\}$
by (*meson disjunct-not-in union-iff*)

lemma *add-mset-inter-add-mset* [*simp*]:
 $\text{add-mset } a \ A \cap \# \text{ add-mset } a \ B = \text{add-mset } a \ (A \cap \# B)$
by (*rule multiset-eqI*) *simp*

lemma *add-mset-disjoint* [*simp*]:
 $\text{add-mset } a \ A \cap \# B = \{\#\} \longleftrightarrow a \notin \# B \wedge A \cap \# B = \{\#\}$
 $\{\#\} = \text{add-mset } a \ A \cap \# B \longleftrightarrow a \notin \# B \wedge \{\#\} = A \cap \# B$
by (*auto simp: disjunct-not-in*)

lemma *disjoint-add-mset* [*simp*]:
 $B \cap \# \text{ add-mset } a \ A = \{\#\} \longleftrightarrow a \notin \# B \wedge B \cap \# A = \{\#\}$
 $\{\#\} = A \cap \# \text{ add-mset } b \ B \longleftrightarrow b \notin \# A \wedge \{\#\} = A \cap \# B$
by (*auto simp: disjunct-not-in*)

lemma *inter-add-left1*: $\neg x \in \# N \implies (\text{add-mset } x \ M) \cap \# N = M \cap \# N$
by (*simp add: multiset-eq-iff not-in-iff*)

lemma *inter-add-left2*: $x \in \# N \implies (\text{add-mset } x \ M) \cap \# N = \text{add-mset } x \ (M \cap \# (N - \{\#x\}))$
by (*auto simp add: multiset-eq-iff elim: mset-add*)

lemma *inter-add-right1*: $\neg x \in \# N \implies N \cap \# (\text{add-mset } x \ M) = N \cap \# M$
by (*simp add: multiset-eq-iff not-in-iff*)

lemma *inter-add-right2*: $x \in \# N \implies N \cap \# (\text{add-mset } x \ M) = \text{add-mset } x \ ((N - \{\#x\}) \cap \# M)$
by (*auto simp add: multiset-eq-iff elim: mset-add*)

lemma *disjunct-set-mset-diff*:
 assumes $M \cap\# N = \{\#\}$
 shows $\text{set-mset } (M - N) = \text{set-mset } M$
proof (rule *set-eqI*)
 fix a
 from *assms* have $a \notin\# M \vee a \notin\# N$
 by (simp add: *disjunct-not-in*)
 then show $a \in\# M - N \longleftrightarrow a \in\# M$
 by (auto dest: *in-diffD*) (simp add: *in-diff-count not-in-iff*)
qed

lemma *at-most-one-mset-mset-diff*:
 assumes $a \notin\# M - \{\#a\}$
 shows $\text{set-mset } (M - \{\#a\}) = \text{set-mset } M - \{a\}$
 using *assms* by (auto simp add: *not-in-iff in-diff-count set-eq-iff*)

lemma *more-than-one-mset-mset-diff*:
 assumes $a \in\# M - \{\#a\}$
 shows $\text{set-mset } (M - \{\#a\}) = \text{set-mset } M$
proof (rule *set-eqI*)
 fix b
 have $\text{Suc } 0 < \text{count } M b \implies \text{count } M b > 0$ by *arith*
 then show $b \in\# M - \{\#a\} \longleftrightarrow b \in\# M$
 using *assms* by (auto simp add: *in-diff-count*)
qed

lemma *inter-iff*:
 $a \in\# A \cap\# B \longleftrightarrow a \in\# A \wedge a \in\# B$
 by *simp*

lemma *inter-union-distrib-left*:
 $A \cap\# B + C = (A + C) \cap\# (B + C)$
 by (simp add: *multiset-eq-iff min-add-distrib-left*)

lemma *inter-union-distrib-right*:
 $C + A \cap\# B = (C + A) \cap\# (C + B)$
 using *inter-union-distrib-left* [of $A B C$] by (simp add: *ac-simps*)

lemma *inter-subset-eq-union*:
 $A \cap\# B \subseteq\# A + B$
 by (auto simp add: *subsemaq-mset-def*)

67.3.9 Additional bounded union facts

lemma *set-mset-sup* [simp]:
 $\langle \text{set-mset } (A \cup\# B) = \text{set-mset } A \cup \text{set-mset } B \rangle$
 by (simp only: *set-mset-def*) (auto simp add: *less-max-iff-disj*)

lemma *sup-union-left1* [simp]: $\neg x \in\# N \implies (\text{add-mset } x \ M) \cup\# N = \text{add-mset } x \ (M \cup\# N)$

by (*simp add: multiset-eq-iff not-in-iff*)

lemma *sup-union-left2*: $x \in\# N \implies (\text{add-mset } x \ M) \cup\# N = \text{add-mset } x \ (M \cup\# (N - \{\#x\}))$

by (*simp add: multiset-eq-iff*)

lemma *sup-union-right1* [simp]: $\neg x \in\# N \implies N \cup\# (\text{add-mset } x \ M) = \text{add-mset } x \ (N \cup\# M)$

by (*simp add: multiset-eq-iff not-in-iff*)

lemma *sup-union-right2*: $x \in\# N \implies N \cup\# (\text{add-mset } x \ M) = \text{add-mset } x \ ((N - \{\#x\}) \cup\# M)$

by (*simp add: multiset-eq-iff*)

lemma *sup-union-distrib-left*:

$A \cup\# B + C = (A + C) \cup\# (B + C)$

by (*simp add: multiset-eq-iff max-add-distrib-left*)

lemma *union-sup-distrib-right*:

$C + A \cup\# B = (C + A) \cup\# (C + B)$

using *sup-union-distrib-left* [of *A B C*] **by** (*simp add: ac-simps*)

lemma *union-diff-inter-eq-sup*:

$A + B - A \cap\# B = A \cup\# B$

by (*auto simp add: multiset-eq-iff*)

lemma *union-diff-sup-eq-inter*:

$A + B - A \cup\# B = A \cap\# B$

by (*auto simp add: multiset-eq-iff*)

lemma *add-mset-union*:

$\langle \text{add-mset } a \ A \cup\# \text{add-mset } a \ B = \text{add-mset } a \ (A \cup\# B) \rangle$

by (*auto simp: multiset-eq-iff max-def*)

67.4 Replicate and repeat operations

definition *replicate-mset* :: $\text{nat} \Rightarrow 'a \Rightarrow 'a \text{ multiset}$ **where**

replicate-mset *n x* = $(\text{add-mset } x \ \overset{\sim}{\sim} n) \ \{\#\}$

lemma *replicate-mset-0* [simp]: *replicate-mset* 0 *x* = $\{\#\}$

unfolding *replicate-mset-def* **by** *simp*

lemma *replicate-mset-Suc* [simp]: *replicate-mset* (*Suc n*) *x* = *add-mset* *x* (*replicate-mset* *n x*)

unfolding *replicate-mset-def* **by** (*induct n*) (*auto intro: add commute*)

lemma *count-replicate-mset* [simp]: *count* (*replicate-mset* *n x*) *y* = (if *y* = *x* then

$n \text{ else } 0)$

unfolding *replicate-mset-def* **by** (*induct* n) *auto*

lift-definition *repeat-mset* :: $\langle \text{nat} \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \rangle$
is $\langle \lambda n \ M \ a. \ n * M \ a \rangle$ **by** *simp*

lemma *count-repeat-mset* [*simp*]: $\text{count} (\text{repeat-mset } i \ A) \ a = i * \text{count } A \ a$
by *transfer rule*

lemma *repeat-mset-0* [*simp*]:
 $\langle \text{repeat-mset } 0 \ M = \{\#\} \rangle$
by *transfer simp*

lemma *repeat-mset-Suc* [*simp*]:
 $\langle \text{repeat-mset} (\text{Suc } n) \ M = M + \text{repeat-mset } n \ M \rangle$
by *transfer simp*

lemma *repeat-mset-right* [*simp*]: $\text{repeat-mset } a \ (\text{repeat-mset } b \ A) = \text{repeat-mset } (a * b) \ A$
by (*auto simp: multiset-eq-iff left-diff-distrib'*)

lemma *left-diff-repeat-mset-distrib'*: $\langle \text{repeat-mset } (i - j) \ u = \text{repeat-mset } i \ u - \text{repeat-mset } j \ u \rangle$
by (*auto simp: multiset-eq-iff left-diff-distrib'*)

lemma *left-add-mult-distrib-mset*:
 $\text{repeat-mset } i \ u + (\text{repeat-mset } j \ u + k) = \text{repeat-mset } (i+j) \ u + k$
by (*auto simp: multiset-eq-iff add-mult-distrib*)

lemma *repeat-mset-distrib*:
 $\text{repeat-mset } (m + n) \ A = \text{repeat-mset } m \ A + \text{repeat-mset } n \ A$
by (*auto simp: multiset-eq-iff Nat.add-mult-distrib*)

lemma *repeat-mset-distrib2* [*simp*]:
 $\text{repeat-mset } n \ (A + B) = \text{repeat-mset } n \ A + \text{repeat-mset } n \ B$
by (*auto simp: multiset-eq-iff add-mult-distrib2*)

lemma *repeat-mset-replicate-mset* [*simp*]:
 $\text{repeat-mset } n \ \{\#a\# \} = \text{replicate-mset } n \ a$
by (*auto simp: multiset-eq-iff*)

lemma *repeat-mset-distrib-add-mset* [*simp*]:
 $\text{repeat-mset } n \ (\text{add-mset } a \ A) = \text{replicate-mset } n \ a + \text{repeat-mset } n \ A$
by (*auto simp: multiset-eq-iff*)

lemma *repeat-mset-empty* [*simp*]: $\text{repeat-mset } n \ \{\#\} = \{\#\}$
by *transfer simp*

lemma *set-mset-sum*: $\text{finite } A \implies \text{set-mset} \ (\sum x \in A. f \ x) = (\bigcup x \in A. \text{set-mset} \ (f$

$x))$
by (*induction A rule: finite-induct*) *auto*

67.4.1 Simprocs

lemma *repeat-mset-iterate-add*: $\langle \text{repeat-mset } n \ M = \text{iterate-add } n \ M \rangle$
unfolding *iterate-add-def* **by** (*induction n*) *auto*

lemma *mset-subseteq-add-iff1*:
 $j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subseteq\# n)$
by (*auto simp add: subseteq-mset-def nat-le-add-iff1*)

lemma *mset-subseteq-add-iff2*:
 $i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subseteq\# \text{repeat-mset } j \ u + n) = (m \subseteq\# \text{repeat-mset } (j-i) \ u + n)$
by (*auto simp add: subseteq-mset-def nat-le-add-iff2*)

lemma *mset-subset-add-iff1*:
 $j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \subset\# n)$
unfolding *subset-mset-def repeat-mset-iterate-add*
by (*simp add: iterate-add-eq-add-iff1 mset-subseteq-add-iff1 [unfolded repeat-mset-iterate-add]*)

lemma *mset-subset-add-iff2*:
 $i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \subset\# \text{repeat-mset } j \ u + n) = (m \subset\# \text{repeat-mset } (j-i) \ u + n)$
unfolding *subset-mset-def repeat-mset-iterate-add*
by (*simp add: iterate-add-eq-add-iff2 mset-subseteq-add-iff2 [unfolded repeat-mset-iterate-add]*)

ML-file $\langle \text{multiset-simprocs.ML} \rangle$

lemma *add-mset-replicate-mset-safe[cancelation-simproc-pre]*: $\langle \text{NO-MATCH } \{\#\} \ M \implies \text{add-mset } a \ M = \{\#a\# \} + M \rangle$
by *simp*

declare *repeat-mset-iterate-add[cancelation-simproc-pre]*

declare *iterate-add-distrib[cancelation-simproc-pre]*

declare *repeat-mset-iterate-add[symmetric, cancelation-simproc-post]*

declare *add-mset-not-empty[cancelation-simproc-eq-elim]*
empty-not-add-mset[cancelation-simproc-eq-elim]
subset-mset.le-zero-eq[cancelation-simproc-eq-elim]
empty-not-add-mset[cancelation-simproc-eq-elim]
add-mset-not-empty[cancelation-simproc-eq-elim]
subset-mset.le-zero-eq[cancelation-simproc-eq-elim]
le-zero-eq[cancelation-simproc-eq-elim]

simproc-setup *mseteq-cancel*

$((l::'a \text{ multiset}) + m = n \mid (l::'a \text{ multiset}) = m + n \mid$
 $\text{add-mset } a \ m = n \mid m = \text{add-mset } a \ n \mid$
 $\text{replicate-mset } p \ a = n \mid m = \text{replicate-mset } p \ a \mid$
 $\text{repeat-mset } p \ m = n \mid m = \text{repeat-mset } p \ m) =$
 $\langle K \text{ Cancel-Simprocs.eq-cancel} \rangle$

simproc-setup *msetsubset-cancel*

$((l::'a \text{ multiset}) + m \subset\# n \mid (l::'a \text{ multiset}) \subset\# m + n \mid$
 $\text{add-mset } a \ m \subset\# n \mid m \subset\# \text{add-mset } a \ n \mid$
 $\text{replicate-mset } p \ r \subset\# n \mid m \subset\# \text{replicate-mset } p \ r \mid$
 $\text{repeat-mset } p \ m \subset\# n \mid m \subset\# \text{repeat-mset } p \ m) =$
 $\langle K \text{ Multiset-Simprocs.subset-cancel-msets} \rangle$

simproc-setup *msetsubset-eq-cancel*

$((l::'a \text{ multiset}) + m \subseteq\# n \mid (l::'a \text{ multiset}) \subseteq\# m + n \mid$
 $\text{add-mset } a \ m \subseteq\# n \mid m \subseteq\# \text{add-mset } a \ n \mid$
 $\text{replicate-mset } p \ r \subseteq\# n \mid m \subseteq\# \text{replicate-mset } p \ r \mid$
 $\text{repeat-mset } p \ m \subseteq\# n \mid m \subseteq\# \text{repeat-mset } p \ m) =$
 $\langle K \text{ Multiset-Simprocs.subseteq-cancel-msets} \rangle$

simproc-setup *msetdiff-cancel*

$((l::'a \text{ multiset}) + m) - n \mid (l::'a \text{ multiset}) - (m + n) \mid$
 $\text{add-mset } a \ m - n \mid m - \text{add-mset } a \ n \mid$
 $\text{replicate-mset } p \ r - n \mid m - \text{replicate-mset } p \ r \mid$
 $\text{repeat-mset } p \ m - n \mid m - \text{repeat-mset } p \ m) =$
 $\langle K \text{ Cancel-Simprocs.diff-cancel} \rangle$

67.4.2 Conditionally complete lattice

instantiation *multiset* :: (type) Inf

begin

lift-definition *Inf-multiset* :: 'a multiset set \Rightarrow 'a multiset **is**

$\lambda A \ i. \text{ if } A = \{\} \text{ then } 0 \text{ else } \text{Inf } ((\lambda f. f \ i) \ 'A)$

proof –

fix *A* :: ('a \Rightarrow nat) set

assume *: $\bigwedge f. f \in A \Longrightarrow \text{finite } \{x. 0 < f \ x\}$

show $\langle \text{finite } \{i. 0 < (\text{if } A = \{\} \text{ then } 0 \text{ else } \text{INF } f \in A. f \ i)\} \rangle$

proof (cases *A* = $\{\}$)

case *False*

then obtain *f* **where** *f* $\in A$ **by** *blast*

hence $\{i. \text{Inf } ((\lambda f. f \ i) \ 'A) > 0\} \subseteq \{i. f \ i > 0\}$

by (auto intro: *less-le-trans*[*OF* - *cInf-lower*])

moreover from $\langle f \in A \rangle$ * **have** *finite* ... **by** *simp*

ultimately have *finite* $\{i. \text{Inf } ((\lambda f. f \ i) \ 'A) > 0\}$ **by** (rule *finite-subset*)

with *False* **show** ?thesis **by** *simp*

qed *simp-all*

qed

instance ..

end

lemma *Inf-multiset-empty*: $\text{Inf } \{\} = \{\#\}$
 by *transfer simp-all*

lemma *count-Inf-multiset-nonempty*: $A \neq \{\} \implies \text{count } (\text{Inf } A) \ x = \text{Inf } ((\lambda X. \text{count } X \ x) \ ` A)$
 by *transfer simp-all*

instantiation *multiset* :: (type) *Sup*
 begin

definition *Sup-multiset* :: 'a multiset set \Rightarrow 'a multiset **where**
 $\text{Sup-multiset } A = (\text{if } A \neq \{\} \wedge \text{subset-mset.bdd-above } A \text{ then}$
 $\text{Abs-multiset } (\lambda i. \text{Sup } ((\lambda X. \text{count } X \ i) \ ` A)) \text{ else } \{\#\})$

lemma *Sup-multiset-empty*: $\text{Sup } \{\} = \{\#\}$
 by (*simp add: Sup-multiset-def*)

lemma *Sup-multiset-unbounded*: $\neg \text{subset-mset.bdd-above } A \implies \text{Sup } A = \{\#\}$
 by (*simp add: Sup-multiset-def*)

instance ..

end

lemma *bdd-above-multiset-imp-bdd-above-count*:
 assumes *subset-mset.bdd-above* ($A :: 'a \text{ multiset set}$)
 shows *bdd-above* $((\lambda X. \text{count } X \ x) \ ` A)$
proof –
 from *assms* **obtain** Y **where** $Y: \forall X \in A. X \subseteq\# Y$
 by (*meson subset-mset.bdd-above.E*)
 hence $\text{count } X \ x \leq \text{count } Y \ x$ **if** $X \in A$ **for** X
 using *that* **by** (*auto intro: mset-subset-eq-count*)
 thus *?thesis* **by** (*intro bdd-aboveI[of - count Y x]*) *auto*
qed

lemma *bdd-above-multiset-imp-finite-support*:
 assumes $A \neq \{\}$ *subset-mset.bdd-above* ($A :: 'a \text{ multiset set}$)
 shows *finite* $(\bigcup X \in A. \{x. \text{count } X \ x > 0\})$
proof –
 from *assms* **obtain** Y **where** $Y: \forall X \in A. X \subseteq\# Y$
 by (*meson subset-mset.bdd-above.E*)
 hence $\text{count } X \ x \leq \text{count } Y \ x$ **if** $X \in A$ **for** X
 using *that* **by** (*auto intro: mset-subset-eq-count*)

hence $(\bigcup X \in A. \{x. \text{count } X \ x > 0\}) \subseteq \{x. \text{count } Y \ x > 0\}$
 by *safe (erule less-le-trans)*
 moreover have *finite ... by simp*
 ultimately show *?thesis by (rule finite-subset)*
 qed

lemma *Sup-multiset-in-multiset:*

$\langle \text{finite } \{i. 0 < (\text{SUP } M \in A. \text{count } M \ i)\} \rangle$
 if $\langle A \neq \{\} \rangle \langle \text{subset-mset.bdd-above } A \rangle$
proof –
 have $\{i. \text{Sup } ((\lambda X. \text{count } X \ i) \text{ ` } A) > 0\} \subseteq (\bigcup X \in A. \{i. 0 < \text{count } X \ i\})$
proof *safe*
 fix *i* assume *pos*: $(\text{SUP } X \in A. \text{count } X \ i) > 0$
 show $i \in (\bigcup X \in A. \{i. 0 < \text{count } X \ i\})$
proof *(rule ccontr)*
 assume $i \notin (\bigcup X \in A. \{i. 0 < \text{count } X \ i\})$
 hence $\forall X \in A. \text{count } X \ i \leq 0$ **by** *(auto simp: count-eq-zero-iff)*
 with *that* have $(\text{SUP } X \in A. \text{count } X \ i) \leq 0$
 by *(intro cSup-least bdd-above-multiset-imp-bdd-above-count) auto*
 with *pos* show *False by simp*
 qed
 qed
 moreover from *that* have *finite ...*
 by *(rule bdd-above-multiset-imp-finite-support)*
 ultimately show *finite* $\{i. \text{Sup } ((\lambda X. \text{count } X \ i) \text{ ` } A) > 0\}$
 by *(rule finite-subset)*
 qed

lemma *count-Sup-multiset-nonempty:*

$\langle \text{count } (\text{Sup } A) \ x = (\text{SUP } X \in A. \text{count } X \ x) \rangle$
 if $\langle A \neq \{\} \rangle \langle \text{subset-mset.bdd-above } A \rangle$
 using *that* **by** *(simp add: Sup-multiset-def Sup-multiset-in-multiset count-Abs-multiset)*

interpretation *subset-mset: conditionally-complete-lattice* *Inf Sup* $(\cap \#)$ $(\subseteq \#)$ $(\subset \#)$ $(\cup \#)$

proof

fix *X* :: 'a multiset and *A*
 assume $X \in A$
 show $\text{Inf } A \subseteq \# X$
 by *(metis \langle X \in A \rangle count-Inf-multiset-nonempty empty-iff image-eqI mset-subset-eqI wellorder-Inf-le1)*

next

fix *X* :: 'a multiset and *A*
 assume *nonempty*: $A \neq \{\}$ and *le*: $\bigwedge Y. Y \in A \implies X \subseteq \# Y$
 show $X \subseteq \# \text{Inf } A$
proof *(rule mset-subset-eqI)*
 fix *x*
 from *nonempty* have $\text{count } X \ x \leq (\text{INF } X \in A. \text{count } X \ x)$
 by *(intro cInf-greatest) (auto intro: mset-subset-eq-count le)*

also from *nonempty* **have** $\dots = \text{count } (\text{Inf } A) \ x$ **by** (*simp add: count-Inf-multiset-nonempty*)
finally show $\text{count } X \ x \leq \text{count } (\text{Inf } A) \ x$.
qed
next
fix $X :: 'a \text{ multiset}$ **and** A
assume $X: X \in A$ **and** $\text{bdd}: \text{subset-mset.bdd-above } A$
show $X \subseteq\# \text{Sup } A$
proof (*rule mset-subset-eqI*)
fix x
from X **have** $A \neq \{\}$ **by** *auto*
have $\text{count } X \ x \leq (\text{SUP } X \in A. \text{count } X \ x)$
by (*intro cSUP-upper X bdd-above-multiset-imp-bdd-above-count bdd*)
also from *count-Sup-multiset-nonempty*[*OF* $\langle A \neq \{\} \rangle$ *bdd*]
have $(\text{SUP } X \in A. \text{count } X \ x) = \text{count } (\text{Sup } A) \ x$ **by** *simp*
finally show $\text{count } X \ x \leq \text{count } (\text{Sup } A) \ x$.
qed
next
fix $X :: 'a \text{ multiset}$ **and** A
assume *nonempty*: $A \neq \{\}$ **and** $\text{ge}: \bigwedge Y. Y \in A \implies Y \subseteq\# X$
from ge **have** $\text{bdd}: \text{subset-mset.bdd-above } A$
by *blast*
show $\text{Sup } A \subseteq\# X$
proof (*rule mset-subset-eqI*)
fix x
from *count-Sup-multiset-nonempty*[*OF* $\langle A \neq \{\} \rangle$ *bdd*]
have $\text{count } (\text{Sup } A) \ x = (\text{SUP } X \in A. \text{count } X \ x)$.
also from *nonempty* **have** $\dots \leq \text{count } X \ x$
by (*intro cSup-least*) (*auto intro: mset-subset-eq-count ge*)
finally show $\text{count } (\text{Sup } A) \ x \leq \text{count } X \ x$.
qed
qed — FIXME: avoid junk stemming from type class interpretation

lemma *set-mset-Inf*:
assumes $A \neq \{\}$
shows $\text{set-mset } (\text{Inf } A) = (\bigcap X \in A. \text{set-mset } X)$
proof *safe*
fix $x \ X$ **assume** $x \in\# \text{Inf } A$ $X \in A$
hence *nonempty*: $A \neq \{\}$ **by** (*auto simp: Inf-multiset-empty*)
from $\langle x \in\# \text{Inf } A \rangle$ **have** $\{\#x\} \subseteq\# \text{Inf } A$ **by** *auto*
also from $\langle X \in A \rangle$ **have** $\dots \subseteq\# X$ **by** (*rule subset-mset.cInf-lower*) *simp-all*
finally show $x \in\# X$ **by** *simp*
next
fix x **assume** $x: x \in (\bigcap X \in A. \text{set-mset } X)$
hence $\{\#x\} \subseteq\# X$ **if** $X \in A$ **for** X **using that** **by** *auto*
from *assms* **and this** **have** $\{\#x\} \subseteq\# \text{Inf } A$ **by** (*rule subset-mset.cInf-greatest*)
thus $x \in\# \text{Inf } A$ **by** *simp*
qed

lemma *in-Inf-multiset-iff*:

assumes $A \neq \{\}$
shows $x \in\# \text{Inf } A \longleftrightarrow (\forall X \in A. x \in\# X)$
proof –
from *assms* **have** $\text{set-mset } (\text{Inf } A) = (\bigcap X \in A. \text{set-mset } X)$ **by** (rule *set-mset-Inf*)
also have $x \in \dots \longleftrightarrow (\forall X \in A. x \in\# X)$ **by** *simp*
finally show *?thesis* .
qed

lemma *in-Inf-multisetD*: $x \in\# \text{Inf } A \implies X \in A \implies x \in\# X$
by (*subst (asm) in-Inf-multiset-iff*) *auto*

lemma *set-mset-Sup*:
assumes *subset-mset.bdd-above A*
shows $\text{set-mset } (\text{Sup } A) = (\bigcup X \in A. \text{set-mset } X)$
proof *safe*
fix x **assume** $x \in\# \text{Sup } A$
hence *nonempty*: $A \neq \{\}$ **by** (*auto simp: Sup-multiset-empty*)
show $x \in (\bigcup X \in A. \text{set-mset } X)$
proof (*rule ccontr*)
assume $x \notin (\bigcup X \in A. \text{set-mset } X)$
have $\text{count } X \ x \leq \text{count } (\text{Sup } A) \ x$ **if** $X \in A$ **for** X
using *that* **by** (*intro mset-subset-eq-count subset-mset.cSup-upper assms*)
with x **have** $X \subseteq\# \text{Sup } A - \{\#x\}$ **if** $X \in A$ **for** X
using *that* **by** (*auto simp: subseteq-mset-def algebra-simps not-in-iff*)
hence $\text{Sup } A \subseteq\# \text{Sup } A - \{\#x\}$ **by** (*intro subset-mset.cSup-least nonempty*)
with $\langle x \in\# \text{Sup } A \rangle$ **show** *False*
using *mset-subset-diff-self* **by** *fastforce*
qed
next
fix $x \ X$ **assume** $x \in \text{set-mset } X \ X \in A$
hence $\{\#x\} \subseteq\# X$ **by** *auto*
also have $X \subseteq\# \text{Sup } A$ **by** (*intro subset-mset.cSup-upper $\langle X \in A \rangle$ assms*)
finally show $x \in \text{set-mset } (\text{Sup } A)$ **by** *simp*
qed

lemma *in-Sup-multiset-iff*:
assumes *subset-mset.bdd-above A*
shows $x \in\# \text{Sup } A \longleftrightarrow (\exists X \in A. x \in\# X)$
by (*simp add: assms set-mset-Sup*)

lemma *in-Sup-multisetD*:
assumes $x \in\# \text{Sup } A$
shows $\exists X \in A. x \in\# X$
using *Sup-multiset-unbounded assms in-Sup-multiset-iff* **by** *fastforce*

interpretation *subset-mset*: *distrib-lattice* $(\cap\#)$ $(\subseteq\#)$ $(\subset\#)$ $(\cup\#)$

proof
fix $A \ B \ C :: 'a \text{ multiset}$
show $A \cup\# (B \cap\# C) = A \cup\# B \cap\# (A \cup\# C)$

by (intro multiset-eqI) simp-all
qed — FIXME: avoid junk stemming from type class interpretation

67.4.3 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: ('a \Rightarrow bool) \Rightarrow 'a multiset \Rightarrow 'a multiset
is $\lambda P M. \lambda x. \text{if } P x \text{ then } M x \text{ else } 0$
by (rule filter-preserves-multiset)

syntax (ASCII)

-MCollect :: ptttrn \Rightarrow 'a multiset \Rightarrow bool \Rightarrow 'a multiset
($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - : \# - / - \# \} \rangle$)

syntax

-MCollect :: ptttrn \Rightarrow 'a multiset \Rightarrow bool \Rightarrow 'a multiset
($\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix multiset comprehension} \rangle \rangle \{ \# - \in \# - / - \# \} \rangle$)

syntax-consts

-MCollect == *filter-mset*

translations

$\{ \# x \in \# M. P \# \} == \text{CONST } \text{filter-mset } (\lambda x. P) M$

lemma *count-filter-mset [simp]*:

count (filter-mset P M) a = (if P a then count M a else 0)

by (simp add: filter-mset.rep-eq)

lemma *set-mset-filter [simp]*:

set-mset (filter-mset P M) = $\{ a \in \text{set-mset } M. P a \}$

by (simp only: set-eq-iff count-greater-zero-iff [symmetric] count-filter-mset) simp

lemma *filter-empty-mset [simp]*: *filter-mset P $\{ \# \} = \{ \# \}$*

by (rule multiset-eqI) simp

lemma *filter-single-mset*: *filter-mset P $\{ \# x \# \} = (\text{if } P x \text{ then } \{ \# x \# \} \text{ else } \{ \# \})$*

by (rule multiset-eqI) simp

lemma *filter-union-mset [simp]*: *filter-mset P (M + N) = filter-mset P M + filter-mset P N*

by (rule multiset-eqI) simp

lemma *filter-diff-mset [simp]*: *filter-mset P (M - N) = filter-mset P M - filter-mset P N*

by (rule multiset-eqI) simp

lemma *filter-inter-mset [simp]*: *filter-mset P (M \cap # N) = filter-mset P M \cap # filter-mset P N*

by (rule multiset-eqI) simp

lemma *filter-sup-mset [simp]*: *filter-mset P (A \cup # B) = filter-mset P A \cup # filter-mset P B*

```

by (rule multiset-eqI) simp

lemma filter-mset-add-mset [simp]:
  filter-mset P (add-mset x A) =
    (if P x then add-mset x (filter-mset P A) else filter-mset P A)
by (auto simp: multiset-eq-iff)

lemma multiset-filter-subset[simp]: filter-mset f M  $\subseteq\#$  M
by (simp add: mset-subset-eqI)

lemma filter-mset-mono-strong:
  assumes A  $\subseteq\#$  B  $\wedge x. x \in\# A \implies P x \implies Q x$ 
  shows filter-mset P A  $\subseteq\#$  filter-mset Q B
by (rule mset-subset-eqI) (insert assms, auto simp: mset-subset-eq-count count-eq-zero-iff)

lemma multiset-filter-mono:
  assumes A  $\subseteq\#$  B
  shows filter-mset f A  $\subseteq\#$  filter-mset f B
  using filter-mset-mono-strong[OF  $\langle A \subseteq\# B \rangle$ ] .

lemma filter-mset-eq-conv:
  filter-mset P M = N  $\longleftrightarrow$  N  $\subseteq\#$  M  $\wedge (\forall b \in\# N. P b) \wedge (\forall a \in\# M - N. \neg P a)$ 
  (is ?P  $\longleftrightarrow$  ?Q)
proof
  assume ?P then show ?Q by auto (simp add: multiset-eq-iff in-diff-count)
next
  assume ?Q
  then obtain Q where M: M = N + Q
    by (auto simp add: mset-subset-eq-exists-conv)
  then have MN: M - N = Q by simp
  show ?P
  proof (rule multiset-eqI)
    fix a
    from  $\langle ?Q \rangle$  MN have *:  $\neg P a \implies a \notin\# N$   $P a \implies a \notin\# Q$ 
    by auto
    show count (filter-mset P M) a = count N a
  proof (cases a  $\in\#$  M)
    case True
    with * show ?thesis
    by (simp add: not-in-iff M)
  next
    case False then have count M a = 0
    by (simp add: not-in-iff)
    with M show ?thesis by simp
  qed
qed
qed

```

lemma *filter-mset-eq-mempty-iff*[simp]: $\text{filter-mset } P \ A = \{\#\} \longleftrightarrow (\forall x. x \in\# A \longrightarrow \neg P \ x)$

by (*auto simp: multiset-eq-iff count-eq-zero-iff*)

lemma *filter-filter-mset*: $\text{filter-mset } P \ (\text{filter-mset } Q \ M) = \{\#x \in\# M. Q \ x \wedge P \ x\# \}$

by (*auto simp: multiset-eq-iff*)

lemma

filter-mset-True[simp]: $\{\#y \in\# M. \text{True}\#\} = M$ **and**

filter-mset-False[simp]: $\{\#y \in\# M. \text{False}\#\} = \{\#\}$

by (*auto simp: multiset-eq-iff*)

lemma *filter-mset-cong0*:

assumes $\bigwedge x. x \in\# M \implies f \ x \longleftrightarrow g \ x$

shows $\text{filter-mset } f \ M = \text{filter-mset } g \ M$

proof (*rule subset-mset.antisym; unfold subseteq-mset-def; rule allI*)

fix x

show $\text{count } (\text{filter-mset } f \ M) \ x \leq \text{count } (\text{filter-mset } g \ M) \ x$

using *assms* **by** (*cases* $x \in\# M$) (*simp-all add: not-in-iff*)

next

fix x

show $\text{count } (\text{filter-mset } g \ M) \ x \leq \text{count } (\text{filter-mset } f \ M) \ x$

using *assms* **by** (*cases* $x \in\# M$) (*simp-all add: not-in-iff*)

qed

lemma *filter-mset-cong*:

assumes $M = M'$ **and** $\bigwedge x. x \in\# M' \implies f \ x \longleftrightarrow g \ x$

shows $\text{filter-mset } f \ M = \text{filter-mset } g \ M'$

unfolding $\langle M = M' \rangle$

using *assms* **by** (*auto intro: filter-mset-cong0*)

lemma *filter-eq-replicate-mset*: $\{\#y \in\# D. y = x\#\} = \text{replicate-mset } (\text{count } D \ x)$

x

by (*induct* D) (*simp add: multiset-eqI*)

67.4.4 Size

definition *wcount* **where** $wcount \ f \ M = (\lambda x. \text{count } M \ x * \text{Suc } (f \ x))$

lemma *wcount-union*: $wcount \ f \ (M + N) \ a = wcount \ f \ M \ a + wcount \ f \ N \ a$

by (*auto simp: wcount-def add-mult-distrib*)

lemma *wcount-add-mset*:

$wcount \ f \ (\text{add-mset } x \ M) \ a = (\text{if } x = a \text{ then } \text{Suc } (f \ a) \text{ else } 0) + wcount \ f \ M \ a$

unfolding *add-mset-add-single*[*of - M*] *wcount-union* **by** (*auto simp: wcount-def*)

definition *size-multiset* :: $('a \Rightarrow \text{nat}) \Rightarrow 'a \text{ multiset} \Rightarrow \text{nat}$ **where**

$\text{size-multiset } f \ M = \text{sum } (wcount \ f \ M) \ (\text{set-mset } M)$

```

lemmas size-multiset-eq = size-multiset-def[unfolded wcount-def]

instantiation multiset :: (type) size
begin

definition size-multiset where
  size-multiset-overloaded-def: size-multiset = Multiset.size-multiset (λ-. 0)
instance ..

end

lemmas size-multiset-overloaded-eq =
  size-multiset-overloaded-def[THEN fun-cong, unfolded size-multiset-eq, simplified]

lemma size-multiset-empty [simp]: size-multiset f {#} = 0
  by (simp add: size-multiset-def)

lemma size-empty [simp]: size {#} = 0
  by (simp add: size-multiset-overloaded-def)

lemma size-multiset-single : size-multiset f {#b#} = Suc (f b)
  by (simp add: size-multiset-eq)

lemma size-single: size {#b#} = 1
  by (simp add: size-multiset-overloaded-def size-multiset-single)

lemma sum-wcount-Int:
  finite A  $\implies$  sum (wcount f N) (A  $\cap$  set-mset N) = sum (wcount f N) A
  by (induct rule: finite-induct)
  (simp-all add: Int-insert-left wcount-def count-eq-zero-iff)

lemma size-multiset-union [simp]:
  size-multiset f (M + N::'a multiset) = size-multiset f M + size-multiset f N
apply (simp add: size-multiset-def sum-Un-nat sum.distrib sum-wcount-Int wcount-union)
by (metis add-implies-diff finite-set-mset inf.commute sum-wcount-Int)

lemma size-multiset-add-mset [simp]:
  size-multiset f (add-mset a M) = Suc (f a) + size-multiset f M
by (metis add.commute add-mset-add-single size-multiset-single size-multiset-union)

lemma size-add-mset [simp]: size (add-mset a A) = Suc (size A)
  by (simp add: size-multiset-overloaded-def wcount-add-mset)

lemma size-union [simp]: size (M + N::'a multiset) = size M + size N
  by (auto simp add: size-multiset-overloaded-def)

lemma size-multiset-eq-0-iff-empty [iff]:
  size-multiset f M = 0  $\longleftrightarrow$  M = {#}

```

by (auto simp add: size-multiset-eq count-eq-zero-iff)

lemma size-eq-0-iff-empty [iff]: $(\text{size } M = 0) = (M = \{\#\})$
 by (auto simp add: size-multiset-overloaded-def)

lemma nonempty-has-size: $(S \neq \{\#\}) = (0 < \text{size } S)$
 by (metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty)

lemma size-eq-Suc-imp-elem: $\text{size } M = \text{Suc } n \implies \exists a. a \in\# M$
 using all-not-in-conv by fastforce

lemma size-eq-Suc-imp-eq-union:
 assumes $\text{size } M = \text{Suc } n$
 shows $\exists a N. M = \text{add-mset } a N$
 by (metis assms insert-DiffM size-eq-Suc-imp-elem)

lemma size-mset-mono:
 fixes $A B :: 'a \text{ multiset}$
 assumes $A \subseteq\# B$
 shows $\text{size } A \leq \text{size } B$
proof –
 from assms[unfolded mset-subset-eq-exists-conv]
 obtain C where $B: B = A + C$ by auto
 show ?thesis unfolding B by (induct C) auto
qed

lemma size-filter-mset-lesseq[simp]: $\text{size } (\text{filter-mset } f M) \leq \text{size } M$
 by (rule size-mset-mono[OF multiset-filter-subset])

lemma size-Diff-submset:
 $M \subseteq\# M' \implies \text{size } (M' - M) = \text{size } M' - \text{size } (M :: 'a \text{ multiset})$
 by (metis add-diff-cancel-left' size-union mset-subset-eq-exists-conv)

lemma size-lt-imp-ex-count-lt: $\text{size } M < \text{size } N \implies \exists x \in\# N. \text{count } M x < \text{count } N x$
 by (metis count-eq-zero-iff leD not-le-imp-less not-less-zero size-mset-mono sub-
 seteq-mset-def)

67.5 Induction and case splits

theorem multiset-induct [case-names empty add, induct type: multiset]:
 assumes empty: $P \{\#\}$
 assumes add: $\bigwedge x M. P M \implies P (\text{add-mset } x M)$
 shows $P M$
proof (induct size M arbitrary: M)
 case 0 thus $P M$ by (simp add: empty)
next
 case (Suc k)
 obtain $N x$ where $M = \text{add-mset } x N$

```

    using  $\langle \text{Suc } k = \text{size } M \rangle$  [symmetric]
    using size-eq-Suc-imp-eq-union by fast
    with Suc add show  $P \ M$  by simp
qed

```

```

lemma multiset-induct-min[case-names empty add]:
  fixes  $M :: 'a::\text{linorder multiset}$ 
  assumes
    empty:  $P \ \{\#\}$  and
    add:  $\bigwedge x \ M. P \ M \implies (\forall y \in \# \ M. y \geq x) \implies P \ (\text{add-mset } x \ M)$ 
  shows  $P \ M$ 
proof (induct size  $M$  arbitrary:  $M$ )
  case (Suc  $k$ )
  note  $ih = \text{this}(1)$  and  $Sk\text{-eq-sz-}M = \text{this}(2)$ 

  let  $?y = \text{Min-mset } M$ 
  let  $?N = M - \{\# ?y \#\}$ 

  have  $M: M = \text{add-mset } ?y \ ?N$ 
  by (metis Min-in Sk-eq-sz- $M$  finite-set-mset insert-DiffM lessI not-less-zero
    set-mset-eq-empty-iff size-empty)
  show ?case
  by (subst  $M$ , rule add, rule  $ih$ , metis  $M$  Sk-eq-sz- $M$  nat.inject size-add-mset,
    meson Min-le finite-set-mset in-diffD)
qed (simp add: empty)

```

```

lemma multiset-induct-max[case-names empty add]:
  fixes  $M :: 'a::\text{linorder multiset}$ 
  assumes
    empty:  $P \ \{\#\}$  and
    add:  $\bigwedge x \ M. P \ M \implies (\forall y \in \# \ M. y \leq x) \implies P \ (\text{add-mset } x \ M)$ 
  shows  $P \ M$ 
proof (induct size  $M$  arbitrary:  $M$ )
  case (Suc  $k$ )
  note  $ih = \text{this}(1)$  and  $Sk\text{-eq-sz-}M = \text{this}(2)$ 

  let  $?y = \text{Max-mset } M$ 
  let  $?N = M - \{\# ?y \#\}$ 

  have  $M: M = \text{add-mset } ?y \ ?N$ 
  by (metis Max-in Sk-eq-sz- $M$  finite-set-mset insert-DiffM lessI not-less-zero
    set-mset-eq-empty-iff size-empty)
  show ?case
  by (subst  $M$ , rule add, rule  $ih$ , metis  $M$  Sk-eq-sz- $M$  nat.inject size-add-mset,
    meson Max-ge finite-set-mset in-diffD)
qed (simp add: empty)

```

```

lemma multi-nonempty-split:  $M \neq \{\#\} \implies \exists A \ a. M = \text{add-mset } a \ A$ 
  by (induct  $M$ ) auto

```

```

lemma multiset-cases [cases type]:
  obtains (empty)  $M = \{\#\}$  | (add)  $x \ N$  where  $M = \text{add-mset } x \ N$ 
  by (induct M) simp-all

lemma multi-drop-mem-not-eq:  $c \in\# B \implies B - \{\#c\} \neq B$ 
  by (cases B = \{\#\}) (auto dest: multi-member-split)

lemma union-filter-mset-complement[simp]:
   $\forall x. P \ x = (\neg Q \ x) \implies \text{filter-mset } P \ M + \text{filter-mset } Q \ M = M$ 
  by (subst multiset-eq-iff) auto

lemma multiset-partition:  $M = \{\#x \in\# M. P \ x\} + \{\#x \in\# M. \neg P \ x\}$ 
  by simp

lemma mset-subset-size:  $A \subset\# B \implies \text{size } A < \text{size } B$ 
proof (induct A arbitrary: B)
  case empty
  then show ?case
    using nonempty-has-size by auto
next
  case (add x A)
  have  $\text{add-mset } x \ A \subseteq\# B$ 
    by (meson add.prem subset-mset-def)
  then show ?case
    using add.prem subset-mset.less-eqE by fastforce
qed

lemma size-1-singleton-mset:  $\text{size } M = 1 \implies \exists a. M = \{\#a\}$ 
  by (cases M) auto

lemma set-mset-subset-singletonD:
  assumes  $\text{set-mset } A \subseteq \{x\}$ 
  shows  $A = \text{replicate-mset } (\text{size } A) \ x$ 
  using assms by (induction A) auto

lemma count-conv-size-mset:  $\text{count } A \ x = \text{size } (\text{filter-mset } (\lambda y. y = x) \ A)$ 
  by (induction A) auto

lemma size-conv-count-bool-mset:  $\text{size } A = \text{count } A \ \text{True} + \text{count } A \ \text{False}$ 
  by (induction A) auto

```

67.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

```

lemma wf-subset-mset-rel:  $\text{wf } \{(M, N :: 'a \text{ multiset}). M \subset\# N\}$ 
  using mset-subset-size wfp-def wfp-if-convertible-to-nat by blast

```

```

lemma wfp-subset-mset[simp]:  $\text{wfp } (\subset\#)$ 

```

```

by (rule wf-subset-mset-rel[to-pred])

lemma full-multiset-induct [case-names less]:
  assumes ih:  $\bigwedge B. \forall (A::'a \text{ multiset}). A \subset\# B \longrightarrow P A \Longrightarrow P B$ 
  shows  $P B$ 
  apply (rule wf-subset-mset-rel [THEN wf-induct])
  apply (rule ih, auto)
  done

lemma multi-subset-induct [consumes 2, case-names empty add]:
  assumes  $F \subseteq\# A$ 
  and empty:  $P \{\#\}$ 
  and insert:  $\bigwedge a F. a \in\# A \Longrightarrow P F \Longrightarrow P (\text{add-mset } a F)$ 
  shows  $P F$ 
proof –
  from  $\langle F \subseteq\# A \rangle$ 
  show ?thesis
  proof (induct F)
    show  $P \{\#\}$  by fact
  next
    fix  $x F$ 
    assume  $P: F \subseteq\# A \Longrightarrow P F$  and  $i: \text{add-mset } x F \subseteq\# A$ 
    show  $P (\text{add-mset } x F)$ 
    proof (rule insert)
      from  $i$  show  $x \in\# A$  by (auto dest: mset-subset-eq-insertD)
      from  $i$  have  $F \subseteq\# A$  by (auto dest: mset-subset-eq-insertD)
      with  $P$  show  $P F$  .
    qed
  qed
qed

```

67.6 Least and greatest elements

context begin

qualified lemma

```

assumes
   $M \neq \{\#\}$  and
  transp-on (set-mset M) R and
  totalp-on (set-mset M) R
shows
  bex-least-element:  $(\exists l \in\# M. \forall x \in\# M. x \neq l \longrightarrow R l x)$  and
  bex-greatest-element:  $(\exists g \in\# M. \forall x \in\# M. x \neq g \longrightarrow R x g)$ 
using assms
by (auto intro: Finite-Set.bex-least-element Finite-Set.bex-greatest-element)

end

```


67.7 The fold combinator

definition *fold-mset* :: ($'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'a$ multiset $\Rightarrow 'b$)

where

fold-mset f s $M = \text{Finite-Set.fold } (\lambda x. f\ x \ \frown \text{count } M\ x)\ s\ (\text{set-mset } M)$

lemma *fold-mset-empty* [*simp*]: *fold-mset* f s $\{\#\}$ = s

by (*simp add: fold-mset-def*)

lemma *fold-mset-single* [*simp*]: *fold-mset* f s $\{\#x\# \} = f\ x\ s$

by (*simp add: fold-mset-def*)

context *comp-fun-commute*

begin

lemma *fold-mset-add-mset* [*simp*]: *fold-mset* f s (*add-mset* x M) = $f\ x$ (*fold-mset* f s M)

proof –

interpret *mset*: *comp-fun-commute* $\lambda y. f\ y \ \frown \text{count } M\ y$

by (*fact comp-fun-commute-funpow*)

interpret *mset-union*: *comp-fun-commute* $\lambda y. f\ y \ \frown \text{count } (\text{add-mset } x\ M)\ y$

by (*fact comp-fun-commute-funpow*)

show ?thesis

proof (*cases* $x \in \text{set-mset } M$)

case *False*

then have *: *count* (*add-mset* x M) $x = 1$

by (*simp add: not-in-iff*)

from *False* **have** *Finite-Set.fold* ($\lambda y. f\ y \ \frown \text{count } (\text{add-mset } x\ M)\ y$) s (*set-mset* M) =

Finite-Set.fold ($\lambda y. f\ y \ \frown \text{count } M\ y$) s (*set-mset* M)

by (*auto intro!*: *Finite-Set.fold-cong comp-fun-commute-on-funpow*)

with *False* * **show** ?thesis

by (*simp add: fold-mset-def del: count-add-mset*)

next

case *True*

define N **where** $N = \text{set-mset } M - \{x\}$

from *N-def* *True* **have** *: *set-mset* $M = \text{insert } x\ N\ x \notin N$ *finite* N **by** *auto*

then have *Finite-Set.fold* ($\lambda y. f\ y \ \frown \text{count } (\text{add-mset } x\ M)\ y$) $s\ N =$

Finite-Set.fold ($\lambda y. f\ y \ \frown \text{count } M\ y$) $s\ N$

by (*auto intro!*: *Finite-Set.fold-cong comp-fun-commute-on-funpow*)

with * **show** ?thesis **by** (*simp add: fold-mset-def del: count-add-mset*) *simp*

qed

qed

lemma *fold-mset-fun-left-comm*: $f\ x$ (*fold-mset* f s M) = *fold-mset* f ($f\ x\ s$) M

by (*induct* M) (*simp-all add: fun-left-comm*)

lemma *fold-mset-union* [*simp*]: *fold-mset* f s ($M + N$) = *fold-mset* f (*fold-mset* f s M) N

by (*induct* M) (*simp-all add: fold-mset-fun-left-comm*)

```

lemma fold-mset-fusion:
  assumes comp-fun-commute g
    and *:  $\bigwedge x y. h (g x y) = f x (h y)$ 
  shows  $h (fold-mset g w A) = fold-mset f (h w) A$ 
proof –
  interpret comp-fun-commute g by (fact assms)
  from * show ?thesis by (induct A) auto
qed

end

```

```

lemma union-fold-mset-add-mset:  $A + B = fold-mset add-mset A B$ 
proof –
  interpret comp-fun-commute add-mset
    by standard auto
  show ?thesis
    by (induction B) auto
qed

```

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like $fold-mset F z \{\#\} = z$ where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

67.8 Image

definition *image-mset* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ multiset} \Rightarrow 'b \text{ multiset}$ **where**
image-mset f = $fold-mset (add-mset \circ f) \{\#\}$

```

lemma comp-fun-commute-mset-image: comp-fun-commute (add-mset \circ f)
  by unfold-locales (simp add: fun-eq-iff)

```

```

lemma image-mset-empty [simp]: image-mset f  $\{\#\} = \{\#\}$ 
  by (simp add: image-mset-def)

```

```

lemma image-mset-single: image-mset f  $\{\#x\# \} = \{\#f x\# \}$ 
  by (simp add: comp-fun-commute.fold-mset-add-mset comp-fun-commute-mset-image image-mset-def)

```

```

lemma image-mset-union [simp]: image-mset f  $(M + N) = image-mset f M + image-mset f N$ 

```

```

proof –
  interpret comp-fun-commute add-mset \circ f
    by (fact comp-fun-commute-mset-image)
  show ?thesis by (induct N) (simp-all add: image-mset-def)
qed

```

corollary *image-mset-add-mset* [simp]:

image-mset f (*add-mset* a M) = *add-mset* (f a) (*image-mset* f M)

unfolding *image-mset-union add-mset-add-single*[of a M] **by** (*simp add: image-mset-single*)

lemma *set-image-mset* [simp]: *set-mset* (*image-mset* f M) = *image* f (*set-mset* M)

by (*induct* M) *simp-all*

lemma *size-image-mset* [simp]: *size* (*image-mset* f M) = *size* M

by (*induct* M) *simp-all*

lemma *image-mset-is-empty-iff* [simp]: *image-mset* f M = {#} \longleftrightarrow M = {#}

by (*cases* M) *auto*

lemma *image-mset-If*:

image-mset ($\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else } g \ x$) A =

image-mset f (*filter-mset* P A) + *image-mset* g (*filter-mset* ($\lambda x. \neg P \ x$) A)

by (*induction* A) *auto*

lemma *filter-mset-image-mset*:

filter-mset P (*image-mset* f A) = *image-mset* f (*filter-mset* ($\lambda x. P \ (f \ x)$) A)

by (*induction* A) *auto*

lemma *image-mset-Diff*:

assumes $B \subseteq\# A$

shows *image-mset* f ($A - B$) = *image-mset* f A - *image-mset* f B

proof -

have *image-mset* f ($A - B + B$) = *image-mset* f ($A - B$) + *image-mset* f B

by *simp*

also from *assms* **have** $A - B + B = A$

by (*simp add: subset-mset.diff-add*)

finally show ?thesis **by** *simp*

qed

lemma *minus-add-mset-if-not-in-lhs*[simp]: $x \notin\# A \implies A - \text{add-mset } x \ B = A - B$

by (*metis diff-intersect-left-idem inter-add-right1*)

lemma *image-mset-diff-if-inj*:

fixes $f \ A \ B$

assumes *inj* f

shows *image-mset* f ($A - B$) = *image-mset* f A - *image-mset* f B

proof (*induction* B)

case *empty*

show ?case

by *simp*

next

case (*add* $x \ B$)

```

show ?case
proof (cases  $x \in \# A - B$ )
  case True

    have  $\text{image-mset } f (A - \text{add-mset } x B) =$ 
       $\text{add-mset } (f x) (\text{image-mset } f (A - \text{add-mset } x B)) - \{\#f x\}$ 
    unfolding  $\text{add-mset-remove-trivial}$  ..

    also have  $\dots = \text{image-mset } f (\text{add-mset } x (A - \text{add-mset } x B)) - \{\#f x\}$ 
    unfolding  $\text{image-mset-add-mset}$  ..

    also have  $\dots = \text{image-mset } f (\text{add-mset } x (A - B - \{\#x\})) - \{\#f x\}$ 
    unfolding  $\text{add-mset-add-single[symmetric]}$   $\text{diff-diff-add-mset}$  ..

    also have  $\dots = \text{image-mset } f (A - B) - \{\#f x\}$ 
    unfolding  $\text{insert-DiffM[OF } \langle x \in \# A - B \rangle]$  ..

    also have  $\dots = \text{image-mset } f A - \text{image-mset } f B - \{\#f x\}$ 
    unfolding  $\text{add.IH}$  ..

    also have  $\dots = \text{image-mset } f A - \text{image-mset } f (\text{add-mset } x B)$ 
    unfolding  $\text{diff-diff-add-mset}$   $\text{add-mset-add-single[symmetric]}$   $\text{image-mset-add-mset}$ 
  ..

  finally show ?thesis .
next
case False

  hence  $\text{image-mset } f (A - \text{add-mset } x B) = \text{image-mset } f (A - B)$ 
  using  $\text{diff-single-trivial}$  by fastforce

  also have  $\dots = \text{image-mset } f A - \text{image-mset } f B - \{\#f x\}$ 
  proof -
    have  $f x \notin f ` \text{set-mset } (A - B)$ 
    using  $\text{False[folded inj-image-mem-iff[OF } \langle \text{inj } f \rangle]]$  .

    hence  $f x \notin \# \text{image-mset } f (A - B)$ 
    unfolding  $\text{set-image-mset}$  .

  thus ?thesis
  unfolding  $\text{add.IH[symmetric]}$ 
  by (metis  $\text{diff-single-trivial}$ )
qed

  also have  $\dots = \text{image-mset } f A - \text{image-mset } f (\text{add-mset } x B)$ 
  by simp

  finally show ?thesis .
qed

```

qed

lemma *count-image-mset*:

$\langle \text{count } (\text{image-mset } f \ A) \ x = (\sum y \in f \ - \{x\} \cap \text{set-mset } A. \text{count } A \ y) \rangle$

proof (*induction A*)

case *empty*

then show *?case by simp*

next

case (*add x A*)

moreover have *: (*if x = y then Suc n else n*) = *n* + (*if x = y then 1 else 0*)

for *n y*

by *simp*

ultimately show *?case*

by (*auto simp: sum.distrib intro!: sum.mono-neutral-left*)

qed

lemma *count-image-mset'*:

$\langle \text{count } (\text{image-mset } f \ X) \ y = (\sum x \mid x \in \# \ X \wedge y = f \ x. \text{count } X \ x) \rangle$

by (*auto simp add: count-image-mset simp flip: singleton-conv2 simp add: Collect-conj-eq ac-simps*)

lemma *image-mset-subseteq-mono*: $A \subseteq \# \ B \implies \text{image-mset } f \ A \subseteq \# \ \text{image-mset } f \ B$

by (*metis image-mset-union subset-mset.le-iff-add*)

lemma *image-mset-subset-mono*: $M \subset \# \ N \implies \text{image-mset } f \ M \subset \# \ \text{image-mset } f \ N$

by (*metis (no-types) Diff-eq-empty-iff-mset image-mset-Diff image-mset-is-empty-iff image-mset-subseteq-mono subset-mset.less-le-not-le*)

syntax (*ASCII*)

-comprehension-mset :: '*a* \Rightarrow '*b* \Rightarrow '*b* multiset \Rightarrow '*a* multiset

($\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \{ \# \text{-} / \cdot \text{-} : \# \text{-} \# \} \rangle \rangle$)

syntax

-comprehension-mset :: '*a* \Rightarrow '*b* \Rightarrow '*b* multiset \Rightarrow '*a* multiset

($\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \{ \# \text{-} / \cdot \text{-} \in \# \text{-} \# \} \rangle \rangle$)

syntax-consts

-comprehension-mset \equiv *image-mset*

translations

$\{ \# e. x \in \# \ M \# \} \equiv \text{CONST image-mset } (\lambda x. e) \ M$

syntax (*ASCII*)

-comprehension-mset' :: '*a* \Rightarrow '*b* \Rightarrow '*b* multiset \Rightarrow *bool* \Rightarrow '*a* multiset

($\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \{ \# \text{-} / \mid \text{-} : \# \text{-} / \text{-} \# \} \rangle \rangle$)

syntax

-comprehension-mset' :: '*a* \Rightarrow '*b* \Rightarrow '*b* multiset \Rightarrow *bool* \Rightarrow '*a* multiset

($\langle \langle \text{notation} = \langle \text{mixfix multiset comprehension} \rangle \{ \# \text{-} / \mid \text{-} \in \# \text{-} / \text{-} \# \} \rangle \rangle$)

syntax-consts

-comprehension-mset' \equiv *image-mset*

translations

$$\{\#e \mid x \in \#M. P\# \} \mapsto \{\#e. x \in \# \{\# x \in \#M. P\#\#\}$$

This allows to write not just filters like $\{\#x \in \# M. x < c\# \}$ but also images like $\{\#x + x. x \in \# M\# \}$ and $\{\#x+x \mid x \in \#M. x < c\# \}$, where the latter is currently displayed as $\{\#x + x. x \in \# \{\#x \in \# M. x < c\#\#\}$.

lemma *in-image-mset*: $y \in \# \{\#f x. x \in \# M\# \} \longleftrightarrow y \in f \text{ ' set-mset } M$
by *simp*

functor *image-mset*: *image-mset*

proof –

fix $f\ g$ **show** *image-mset* $f \circ \text{image-mset } g = \text{image-mset } (f \circ g)$

proof

fix A

show $(\text{image-mset } f \circ \text{image-mset } g) A = \text{image-mset } (f \circ g) A$

by $(\text{induct } A) \text{ simp-all}$

qed

show *image-mset* $\text{id} = \text{id}$

proof

fix A

show *image-mset* $\text{id } A = \text{id } A$

by $(\text{induct } A) \text{ simp-all}$

qed

qed

declare

image-mset.id [*simp*]

image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id* $x = x$

unfolding *id-def* **by** *auto*

lemma *image-mset-cong*: $(\bigwedge x. x \in \# M \implies f x = g x) \implies \{\#f x. x \in \# M\# \} = \{\#g x. x \in \# M\# \}$

by $(\text{induct } M) \text{ auto}$

lemma *image-mset-cong-pair*:

$(\forall x\ y. (x, y) \in \# M \longrightarrow f x\ y = g x\ y) \implies \{\#f x\ y. (x, y) \in \# M\# \} = \{\#g x\ y. (x, y) \in \# M\# \}$

by $(\text{metis } \text{image-mset-cong } \text{split-cong})$

lemma *image-mset-const-eq*:

$\{\#c. a \in \# M\# \} = \text{replicate-mset } (\text{size } M) c$

by $(\text{induct } M) \text{ simp-all}$

lemma *image-mset-filter-mset-swap*:

image-mset f $(\text{filter-mset } (\lambda x. P (f x)) M) = \text{filter-mset } P (\text{image-mset } f M)$

by $(\text{induction } M \text{ rule: multiset-induct}) \text{ simp-all}$

lemma *image-mset-eq-plusD*:

image-mset f A = B + C $\implies \exists B' C'. A = B' + C' \wedge B = \text{image-mset f } B' \wedge C = \text{image-mset f } C'$

proof (*induction A arbitrary: B C*)

case *empty*

thus *?case by simp*

next

case (*add x A*)

show *?case*

proof (*cases f x $\in \# B$*)

case *True*

with *add.prem*s **have** *image-mset f A = (B - {#f x#}) + C*

by (*metis add-mset-remove-trivial image-mset-add-mset mset-subset-eq-single subset-mset.add-diff-assoc2*)

thus *?thesis*

using *add.IH add.prem*s **by** *force*

next

case *False*

with *add.prem*s **have** *image-mset f A = B + (C - {#f x#})*

by (*metis diff-single-eq-union diff-union-single-conv image-mset-add-mset union-iff*

union-single-eq-member)

then show *?thesis*

using *add.IH add.prem*s **by** *force*

qed

qed

lemma *image-mset-eq-image-mset-plusD*:

assumes *image-mset f A = image-mset f B + C* **and** *inj-f: inj-on f (set-mset A \cup set-mset B)*

shows $\exists C'. A = B + C' \wedge C = \text{image-mset f } C'$

using *assms*

proof (*induction A arbitrary: B C*)

case *empty*

thus *?case by simp*

next

case (*add x A*)

show *?case*

proof (*cases x $\in \# B$*)

case *True*

with *add.prem*s **have** *image-mset f A = image-mset f (B - {#x#}) + C*

by (*smt (verit) add-mset-add-mset-same-iff image-mset-add-mset insert-DiffM union-mset-add-mset-left*)

with *add.IH* **have** $\exists M3'. A = B - \{ \#x\# \} + M3' \wedge \text{image-mset f } M3' = C$

by (*smt (verit, del-insts) True Un-insert-left Un-insert-right add.prem*s(2) *inj-on-insert*

insert-DiffM set-mset-add-mset-insert)

with *True* **show** *?thesis*

by *auto*

```

next
  case False
  with add.premis(2) have  $f\ x \notin \#$  image-mset  $f\ B$ 
  by auto
  with add.premis(1) have  $\text{image-mset } f\ A = \text{image-mset } f\ B + (C - \{\#f\ x\# \})$ 
  by (metis (no-types, lifting) diff-union-single-conv image-eqI image-mset-Diff
    image-mset-single mset-subset-eq-single set-image-mset union-iff union-single-eq-diff
    union-single-eq-member)
  with add.premis(2) add.IH have  $\exists M3'. A = B + M3' \wedge C - \{\#f\ x\# \} =$ 
    image-mset  $f\ M3'$ 
  by auto
  then show ?thesis
  by (metis add.premis(1) add-diff-cancel-left' image-mset-Diff mset-subset-eq-add-left
    union-mset-add-mset-right)
qed
qed

```

lemma *image-mset-eq-plus-image-msetD*:
 $\text{image-mset } f\ A = B + \text{image-mset } f\ C \implies \text{inj-on } f\ (\text{set-mset } A \cup \text{set-mset } C)$
 \implies
 $\exists B'. A = B' + C \wedge B = \text{image-mset } f\ B'$
unfolding *add.commute[of B] add.commute[of - C]*
by (rule image-mset-eq-image-mset-plusD; assumption)

67.9 Further conversions

primrec *mset* :: 'a list \Rightarrow 'a multiset **where**
 $\text{mset } [] = \{\#\}$ |
 $\text{mset } (a \# x) = \text{add-mset } a\ (\text{mset } x)$

lemma *in-multiset-in-set*:
 $x \in \# \text{ mset } xs \longleftrightarrow x \in \text{set } xs$
by (induct xs) simp-all

lemma *count-mset*:
 $\text{count } (\text{mset } xs)\ x = \text{count-list } xs\ x$
by (induct xs) simp-all

lemma *mset-zero-iff[simp]*: $(\text{mset } x = \{\#\}) = (x = [])$
by (induct x) auto

lemma *mset-zero-iff-right[simp]*: $(\{\#\} = \text{mset } x) = (x = [])$
by (induct x) auto

lemma *mset-replicate [simp]*: $\text{mset } (\text{replicate } n\ x) = \text{replicate-mset } n\ x$
by (induction n) auto

lemma *count-mset-gt-0*: $x \in \text{set } xs \implies \text{count } (\text{mset } xs)\ x > 0$
by (induction xs) auto

lemma *count-mset-0-iff* [*simp*]: $\text{count } (\text{mset } xs) \ x = 0 \longleftrightarrow x \notin \text{set } xs$
by (*induction xs*) *auto*

lemma *mset-single-iff*[*iff*]: $\text{mset } xs = \{\#x\#\} \longleftrightarrow xs = [x]$
by (*cases xs*) *auto*

lemma *mset-single-iff-right*[*iff*]: $\{\#x\#\} = \text{mset } xs \longleftrightarrow xs = [x]$
by (*cases xs*) *auto*

lemma *set-mset-mset*[*simp*]: $\text{set-mset } (\text{mset } xs) = \text{set } xs$
by (*induct xs*) *auto*

lemma *set-mset-comp-mset* [*simp*]: $\text{set-mset} \circ \text{mset} = \text{set}$
by (*simp add: fun-eq-iff*)

lemma *size-mset* [*simp*]: $\text{size } (\text{mset } xs) = \text{length } xs$
by (*induct xs*) *simp-all*

lemma *mset-append* [*simp*]: $\text{mset } (xs @ ys) = \text{mset } xs + \text{mset } ys$
by (*induct xs arbitrary: ys*) *auto*

lemma *mset-filter*[*simp*]: $\text{mset } (\text{filter } P \ xs) = \{\#x \in \# \text{ mset } xs. \ P \ x \ \#\}$
by (*induct xs*) *simp-all*

lemma *mset-rev* [*simp*]:
 $\text{mset } (\text{rev } xs) = \text{mset } xs$
by (*induct xs*) *simp-all*

lemma *surj-mset*: *surj mset*
unfolding *surj-def*
proof (*rule allI*)
fix *M*
show $\exists xs. \ M = \text{mset } xs$
by (*induction M*) (*auto intro: exI[of - - \# -]*)
qed

lemma *distinct-count-atmost-1*:
 $\text{distinct } x = (\forall a. \ \text{count } (\text{mset } x) \ a = (\text{if } a \in \text{set } x \text{ then } 1 \text{ else } 0))$
proof (*induct x*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **show** *?case* (**is** *?lhs* \longleftrightarrow *?rhs*)
proof
assume *?lhs* **then show** *?rhs* **using** *Cons* **by** *simp*
next
assume *?rhs* **then have** $x \notin \text{set } xs$
by (*simp split: if-splits*)
moreover from $\langle ?rhs \rangle$ **have** $(\forall a. \ \text{count } (\text{mset } xs) \ a =$

```

      (if a ∈ set xs then 1 else 0))
    by (auto split: if-splits simp add: count-eq-zero-iff)
    ultimately show ?lhs using Cons by simp
  qed
qed

```

```

lemma mset-eq-setD:
  assumes mset xs = mset ys
  shows set xs = set ys
proof -
  from assms have set-mset (mset xs) = set-mset (mset ys)
    by simp
  then show ?thesis by simp
qed

```

```

lemma set-eq-iff-mset-eq-distinct:
  ⟨distinct x ⟹ distinct y ⟹ set x = set y ⟷ mset x = mset y⟩
  by (auto simp: multiset-eq-iff distinct-count-atmost-1)

```

```

lemma set-eq-iff-mset-remdups-eq:
  ⟨set x = set y ⟷ mset (remdups x) = mset (remdups y)⟩
  using set-eq-iff-mset-eq-distinct by fastforce

```

```

lemma mset-eq-imp-distinct-iff:
  ⟨distinct xs ⟷ distinct ys⟩ if ⟨mset xs = mset ys⟩
  using that by (auto simp add: distinct-count-atmost-1 dest: mset-eq-setD)

```

```

lemma nth-mem-mset: i < length ls ⟹ (ls ! i) ∈# mset ls
proof (induct ls arbitrary: i)
  case Nil
  then show ?case by simp
next
  case Cons
  then show ?case by (cases i) auto
qed

```

```

lemma mset-remove1[simp]: mset (remove1 a xs) = mset xs - {#a#}
  by (induct xs) (auto simp add: multiset-eq-iff)

```

```

lemma mset-eq-length:
  assumes mset xs = mset ys
  shows length xs = length ys
  using assms by (metis size-mset)

```

```

lemma mset-eq-length-filter:
  assumes mset xs = mset ys
  shows count-list xs z = count-list ys z
  using assms by (metis count-mset)

```

lemma *fold-multiset-equiv*:
 $\langle \text{List.fold } f \text{ } xs = \text{List.fold } f \text{ } ys \rangle$
 if f : $\langle \bigwedge x y. x \in \text{set } xs \implies y \in \text{set } xs \implies f x \circ f y = f y \circ f x \rangle$
 and $\langle \text{mset } xs = \text{mset } ys \rangle$
using f $\langle \text{mset } xs = \text{mset } ys \rangle$ [symmetric] **proof** (*induction xs arbitrary: ys*)
 case Nil
 then show ?case by simp
next
 case (Cons x xs)
 then have *: $\langle \text{set } ys = \text{set } (x \# xs) \rangle$
 by (blast dest: mset-eq-setD)
 have $\langle \bigwedge x y. x \in \text{set } ys \implies y \in \text{set } ys \implies f x \circ f y = f y \circ f x \rangle$
 by (rule Cons.prem1) (simp-all add: *)
 moreover from * have $\langle x \in \text{set } ys \rangle$
 by simp
 ultimately have $\langle \text{List.fold } f \text{ } ys = \text{List.fold } f \text{ } (\text{remove1 } x \text{ } ys) \circ f x \rangle$
 by (fact fold-remove1-split)
 moreover from Cons.prem1 have $\langle \text{List.fold } f \text{ } xs = \text{List.fold } f \text{ } (\text{remove1 } x \text{ } xs) \rangle$
 by (auto intro: Cons.IH)
 ultimately show ?case
 by simp
qed

lemma *fold-permuted-eq*:
 $\langle \text{List.fold } (\odot) \text{ } xs \text{ } z = \text{List.fold } (\odot) \text{ } ys \text{ } z \rangle$
 if $\langle \text{mset } xs = \text{mset } ys \rangle$
 and $\langle P \text{ } z \rangle$ and P : $\langle \bigwedge x z. x \in \text{set } xs \implies P \text{ } z \implies P \text{ } (x \odot z) \rangle$
 and f : $\langle \bigwedge x y z. x \in \text{set } xs \implies y \in \text{set } xs \implies P \text{ } z \implies x \odot (y \odot z) = y \odot (x \odot z) \rangle$
 for f (infixl \odot 70)
using $\langle P \text{ } z \rangle$ $P \text{ } f$ $\langle \text{mset } xs = \text{mset } ys \rangle$ [symmetric] **proof** (*induction xs arbitrary: ys z*)
 case Nil
 then show ?case by simp
next
 case (Cons x xs)
 then have *: $\langle \text{set } ys = \text{set } (x \# xs) \rangle$
 by (blast dest: mset-eq-setD)
 have $\langle P \text{ } z \rangle$
 by (fact Cons.prem1)
 moreover have $\langle \bigwedge x z. x \in \text{set } ys \implies P \text{ } z \implies P \text{ } (x \odot z) \rangle$
 by (rule Cons.prem2) (simp-all add: *)
 moreover have $\langle \bigwedge x y z. x \in \text{set } ys \implies y \in \text{set } ys \implies P \text{ } z \implies x \odot (y \odot z) = y \odot (x \odot z) \rangle$
 by (rule Cons.prem3) (simp-all add: *)
 moreover from * have $\langle x \in \text{set } ys \rangle$
 by simp
 ultimately have $\langle \text{fold } (\odot) \text{ } ys \text{ } z = \text{fold } (\odot) \text{ } (\text{remove1 } x \text{ } ys) \text{ } (x \odot z) \rangle$
 by (induction ys arbitrary: z) auto

moreover from *Cons.prem*s **have** $\langle \text{fold } (\odot) \text{ } xs \text{ } (x \odot z) = \text{fold } (\odot) \text{ } (\text{remove1 } x \text{ } ys) \text{ } (x \odot z) \rangle$
by (*auto intro: Cons.IH*)
ultimately show *?case*
by *simp*
qed

lemma *mset-shuffles*: $zs \in \text{shuffles } xs \text{ } ys \implies \text{mset } zs = \text{mset } xs + \text{mset } ys$
by (*induction xs ys arbitrary: zs rule: shuffles.induct*) *auto*

lemma *mset-insort* [*simp*]: $\text{mset } (\text{insort } x \text{ } xs) = \text{add-mset } x \text{ } (\text{mset } xs)$
by (*induct xs simp-all*)

lemma *mset-map* [*simp*]: $\text{mset } (\text{map } f \text{ } xs) = \text{image-mset } f \text{ } (\text{mset } xs)$
by (*induct xs simp-all*)

lemma *mset-removeAll-eq*:
 $\langle \text{mset } (\text{removeAll } x \text{ } xs) = \text{filter-mset } ((\neq) \text{ } x) \text{ } (\text{mset } xs) \rangle$
by (*induction xs*) *auto*

lemma *singleton-set-mset-subset*: **fixes** $X \ Y :: 'a \text{ list set}$
assumes $\forall xs \in X. \text{set } xs \subseteq \{a\}$ *mset ' X* \subseteq *mset ' Y*
shows $X \subseteq Y$

proof
fix xs **assume** $xs \in X$
obtain ys **where** $ys \in Y$ $\text{mset } xs = \text{mset } ys$
using $\langle xs \in X \rangle$ *assms(2)* **by** *auto*
then show $xs \in Y$ **using** $\langle xs \in X \rangle$ *assms(1)* ys
by (*metis singleton-iff mset-eq-setD replicate-eqI set-empty subset-singletonD size-mset*)
qed

lemma *singleton-set-mset-eq*: **fixes** $X \ Y :: 'a \text{ list set}$
assumes $\forall xs \in X. \text{set } xs \subseteq \{a\}$ *mset ' X* $=$ *mset ' Y*
shows $X = Y$

proof –
have $\forall ys \in Y. \text{set } ys \subseteq \{a\}$
by (*metis (mono-tags, lifting) assms image-iff mset-eq-setD*)
thus *?thesis*
by (*metis antisym assms(1,2) singleton-set-mset-subset subset-refl*)
qed

global-interpretation *mset-set*: *folding add-mset {#}*
defines $\text{mset-set} = \text{folding-on.F add-mset } \{ \# \}$
by *standard (simp add: fun-eq-iff)*

lemma *sum-multiset-singleton* [*simp*]: $\text{sum } (\lambda n. \{ \# n \# \}) \text{ } A = \text{mset-set } A$
by (*induction A rule: infinite-finite-induct*) *auto*

lemma *count-mset-set* [simp]:

$\text{finite } A \implies x \in A \implies \text{count } (\text{mset-set } A) \ x = 1$ (is PROP ?P)

$\neg \text{finite } A \implies \text{count } (\text{mset-set } A) \ x = 0$ (is PROP ?Q)

$x \notin A \implies \text{count } (\text{mset-set } A) \ x = 0$ (is PROP ?R)

proof –

have *: $\text{count } (\text{mset-set } A) \ x = 0$ if $x \notin A$ **for** A

proof (cases *finite A*)

case *False* **then show** ?thesis **by** *simp*

next

case *True* **from** *True* $\langle x \notin A \rangle$ **show** ?thesis **by** (induct *A*) *auto*

qed

then show PROP ?P PROP ?Q PROP ?R

by (*auto elim!*: *Set.set-insert*)

qed — TODO: maybe define *mset-set* also in terms of *Abs-multiset*

lemma *elem-mset-set*[simp, intro]: $\text{finite } A \implies x \in \# \text{ mset-set } A \longleftrightarrow x \in A$

by (induct *A* rule: *finite-induct*) *simp-all*

lemma *mset-set-Union*:

$\text{finite } A \implies \text{finite } B \implies A \cap B = \{\} \implies \text{mset-set } (A \cup B) = \text{mset-set } A + \text{mset-set } B$

by (induction *A* rule: *finite-induct*) *auto*

lemma *filter-mset-mset-set* [simp]:

$\text{finite } A \implies \text{filter-mset } P \ (\text{mset-set } A) = \text{mset-set } \{x \in A. P \ x\}$

proof (induction *A* rule: *finite-induct*)

case (*insert x A*)

from *insert.hyps* **have** $\text{filter-mset } P \ (\text{mset-set } (\text{insert } x \ A)) =$

$\text{filter-mset } P \ (\text{mset-set } A) + \text{mset-set } (\text{if } P \ x \ \text{then } \{x\} \ \text{else } \{\})$

by *simp*

also have $\text{filter-mset } P \ (\text{mset-set } A) = \text{mset-set } \{x \in A. P \ x\}$

by (rule *insert.IH*)

also from *insert.hyps*

have $\dots + \text{mset-set } (\text{if } P \ x \ \text{then } \{x\} \ \text{else } \{\}) =$

$\text{mset-set } (\{x \in A. P \ x\} \cup (\text{if } P \ x \ \text{then } \{x\} \ \text{else } \{\}))$ (is - = *mset-set ?A*)

by (*intro mset-set-Union* [symmetric]) *simp-all*

also from *insert.hyps* **have** $?A = \{y \in \text{insert } x \ A. P \ y\}$ **by** *auto*

finally show ?case .

qed *simp-all*

lemma *mset-set-Diff*:

assumes $\text{finite } A \ B \subseteq A$

shows $\text{mset-set } (A - B) = \text{mset-set } A - \text{mset-set } B$

proof –

from *assms* **have** $\text{mset-set } ((A - B) \cup B) = \text{mset-set } (A - B) + \text{mset-set } B$

by (*intro mset-set-Union*) (*auto dest: finite-subset*)

also from *assms* **have** $A - B \cup B = A$ **by** *blast*

finally show ?thesis **by** *simp*

qed

lemma *mset-minus-list-mset*[simp]: $mset(minus-list-mset\ xs\ ys) = mset\ xs - mset\ ys$

by (*simp add: count-mset multiset-eq-iff*)

lemma *mset-set-set*: $distinct\ xs \implies mset-set\ (set\ xs) = mset\ xs$

by (*induction xs simp-all*)

lemma *count-mset-set'*: $count\ (mset-set\ A)\ x = (if\ finite\ A \wedge x \in A\ then\ 1\ else\ 0)$

by *auto*

lemma *subset-imp-msubset-mset-set*:

assumes $A \subseteq B$ *finite B*

shows $mset-set\ A \subseteq\# mset-set\ B$

proof (*rule mset-subset-eqI*)

fix $x :: 'a$

from *assms* **have** *finite A* **by** (*rule finite-subset*)

with *assms* **show** $count\ (mset-set\ A)\ x \leq count\ (mset-set\ B)\ x$

by (*cases x ∈ A; cases x ∈ B auto*)

qed

lemma *mset-set-set-mset-msubset*: $mset-set\ (set-mset\ A) \subseteq\# A$

proof (*rule mset-subset-eqI*)

fix x **show** $count\ (mset-set\ (set-mset\ A))\ x \leq count\ A\ x$

by (*cases x ∈# A simp-all*)

qed

lemma *mset-set-upto-eq-mset-upto*:

$\langle mset-set\ \{..$

by (*induction n (auto simp: ac-simps lessThan-Suc)*)

context *linorder*

begin

definition *sorted-list-of-multiset* :: $'a\ multiset \Rightarrow 'a\ list$

where

$sorted-list-of-multiset\ M = fold-mset\ insert\ []\ M$

lemma *sorted-list-of-multiset-empty* [simp]:

$sorted-list-of-multiset\ \{\#\} = []$

by (*simp add: sorted-list-of-multiset-def*)

lemma *sorted-list-of-multiset-singleton* [simp]:

$sorted-list-of-multiset\ \{\#x\# \} = [x]$

proof –

interpret *comp-fun-commute insert* **by** (*fact comp-fun-commute-insert*)

show *?thesis* **by** (*simp add: sorted-list-of-multiset-def*)

qed

```

lemma sorted-list-of-multiset-insert [simp]:
  sorted-list-of-multiset (add-mset x M) = List.insort x (sorted-list-of-multiset M)
proof –
  interpret comp-fun-commute-insort by (fact comp-fun-commute-insort)
  show ?thesis by (simp add: sorted-list-of-multiset-def)
qed

end

lemma mset-sorted-list-of-multiset[simp]: mset (sorted-list-of-multiset M) = M
  by (induct M) simp-all

lemma sorted-list-of-multiset-mset[simp]: sorted-list-of-multiset (mset xs) = sort
xs
  by (induct xs) simp-all

lemma finite-set-mset-mset-set[simp]: finite A  $\implies$  set-mset (mset-set A) = A
  by auto

lemma mset-set-empty-iff: mset-set A = {#}  $\longleftrightarrow$  A = {}  $\vee$  infinite A
  using finite-set-mset-mset-set by fastforce

lemma infinite-set-mset-mset-set:  $\neg$  finite A  $\implies$  set-mset (mset-set A) = {}
  by simp

lemma set-sorted-list-of-multiset [simp]:
  set (sorted-list-of-multiset M) = set-mset M
  by (induct M) (simp-all add: set-insort-key)

lemma sorted-sorted-list-of-multiset [iff]:
   $\langle$ sorted (sorted-list-of-multiset M) $\rangle$ 
  by (induction M) (simp-all add: sorted-insort)

lemma sorted-list-of-mset-set [simp]:
  sorted-list-of-multiset (mset-set A) = sorted-list-of-set A
  by (cases finite A) (induct A rule: finite-induct, simp-all)

lemma mset-upt [simp]: mset [m..n] = mset-set {m..n}
  by (metis distinct-upt mset-set-set set-upt)

lemma image-mset-map-of:
  distinct (map fst xs)  $\implies$  {#the (map-of xs i). i  $\in$  # mset (map fst xs)#} = mset
(map snd xs)
proof (induction xs)
  case (Cons x xs)
  have {#the (map-of (x # xs) i). i  $\in$  # mset (map fst (x # xs))#} =
    add-mset (snd x) {#the (if i = fst x then Some (snd x) else map-of xs i).
      i  $\in$  # mset (map fst xs)#} (is - = add-mset - ?A) by simp

```

```

also from Cons.premis have ?A = {#the (map-of xs i). i :# mset (map fst
xs)#}
  by (cases x, intro image-mset-cong) (auto simp: in-multiset-in-set)
also from Cons.premis have ... = mset (map snd xs) by (intro Cons.IH)
simp-all
finally show ?case by simp
qed simp-all

```

```

lemma msubset-mset-set-iff[simp]:
  assumes finite A finite B
  shows mset-set A  $\subseteq$  # mset-set B  $\longleftrightarrow$  A  $\subseteq$  B
  using assms set-mset-mono subset-imp-msubset-mset-set by fastforce

```

```

lemma mset-set-eq-iff[simp]:
  assumes finite A finite B
  shows mset-set A = mset-set B  $\longleftrightarrow$  A = B
  using assms by (fastforce dest: finite-set-mset-mset-set)

```

```

lemma image-mset-mset-set:
  assumes inj-on f A
  shows image-mset f (mset-set A) = mset-set (f ` A)
proof cases
  assume finite A
  from this <inj-on f A> show ?thesis
    by (induct A) auto
next
  assume infinite A
  from this <inj-on f A> have infinite (f ` A)
    using finite-imageD by blast
  from <infinite A> <infinite (f ` A)> show ?thesis by simp
qed

```

67.10 More properties of the replicate, repeat, and image operations

```

lemma in-replicate-mset[simp]: x  $\in$  # replicate-mset n y  $\longleftrightarrow$  n > 0  $\wedge$  x = y
  unfolding replicate-mset-def by (induct n) auto

```

```

lemma set-mset-replicate-mset-subset[simp]: set-mset (replicate-mset n x) = (if n
= 0 then {} else {x})
  by auto

```

```

lemma size-replicate-mset[simp]: size (replicate-mset n M) = n
  by (induct n, simp-all)

```

```

lemma size-repeat-mset [simp]: size (repeat-mset n A) = n * size A
  by (induction n) auto

```

```

lemma size-multiset-sum [simp]: size ( $\sum$  x $\in$ A. f x :: 'a multiset) = ( $\sum$  x $\in$ A. size

```


(f x))

by (induction A rule: infinite-finite-induct) auto

lemma size-multiset-sum-list [simp]: size ($\sum X \leftarrow Xs. X :: 'a \text{ multiset}$) = ($\sum X \leftarrow Xs.$
size X)

by (induction Xs) auto

lemma count-le-replicate-mset-subset-eq: $n \leq \text{count } M \ x \longleftrightarrow \text{replicate-mset } n \ x \subseteq \# M$

by (auto simp add: mset-subset-eqI) (metis count-replicate-mset subseteq-mset-def)

lemma replicate-count-mset-eq-filter-eq: replicate (count (mset xs) k) k = filter (HOL.eq k) xs

by (induct xs) auto

lemma replicate-mset-eq-empty-iff [simp]: replicate-mset n a = {#} $\longleftrightarrow n = 0$

by (induct n) simp-all

lemma replicate-mset-eq-iff:

replicate-mset m a = replicate-mset n b $\longleftrightarrow m = 0 \wedge n = 0 \vee m = n \wedge a = b$

by (auto simp add: multiset-eq-iff)

lemma repeat-mset-cancel1: repeat-mset a A = repeat-mset a B $\longleftrightarrow A = B \vee a = 0$

by (auto simp: multiset-eq-iff)

lemma repeat-mset-cancel2: repeat-mset a A = repeat-mset b A $\longleftrightarrow a = b \vee A = \{ \# \}$

by (auto simp: multiset-eq-iff)

lemma repeat-mset-eq-empty-iff: repeat-mset n A = {#} $\longleftrightarrow n = 0 \vee A = \{ \# \}$

by (cases n) auto

lemma image-replicate-mset [simp]:

image-mset f (replicate-mset n a) = replicate-mset n (f a)

by (induct n) simp-all

lemma replicate-mset-msubseteq-iff:

replicate-mset m a $\subseteq \#$ replicate-mset n b $\longleftrightarrow m = 0 \vee a = b \wedge m \leq n$

by (cases m)

(auto simp: insert-subset-eq-iff simp flip: count-le-replicate-mset-subset-eq)

lemma msubseteq-replicate-msetE:

assumes A $\subseteq \#$ replicate-mset n a

obtains m where $m \leq n$ and A = replicate-mset m a

proof (cases n = 0)

case True

with assms that show thesis

by simp

```

next
  case False
  from assms have set-mset  $A \subseteq \text{set-mset } (\text{replicate-mset } n \ a)$ 
    by (rule set-mset-mono)
  with False have set-mset  $A \subseteq \{a\}$ 
    by simp
  then have  $\exists m. A = \text{replicate-mset } m \ a$ 
  proof (induction A)
    case empty
    then show ?case
      by simp
  next
  case (add b A)
  then obtain m where  $A = \text{replicate-mset } m \ a$ 
    by auto
  with add.prems show ?case
    by (auto intro: exI [of - Suc m])
qed
then obtain m where  $A: A = \text{replicate-mset } m \ a \ ..$ 
with assms have  $m \leq n$ 
  by (auto simp add: replicate-mset-msubseteq-iff)
then show thesis using A ..
qed

lemma count-image-mset-lt-imp-lt-raw:
  assumes
    finite A and
     $A = \text{set-mset } M \cup \text{set-mset } N$  and
     $\text{count } (\text{image-mset } f \ M) \ b < \text{count } (\text{image-mset } f \ N) \ b$ 
  shows  $\exists x. f \ x = b \wedge \text{count } M \ x < \text{count } N \ x$ 
  using assms
proof (induct A arbitrary: M N b rule: finite-induct)
  case (insert x F)
  note fin = this(1) and x-ni-f = this(2) and ih = this(3) and x-f-eq-m-n =
this(4) and
    cnt-b = this(5)

  let ?Ma =  $\{\#y \in \# M. y \neq x\# \}$ 
  let ?Mb =  $\{\#y \in \# M. y = x\# \}$ 
  let ?Na =  $\{\#y \in \# N. y \neq x\# \}$ 
  let ?Nb =  $\{\#y \in \# N. y = x\# \}$ 

  have m-part:  $M = ?Mb + ?Ma$  and n-part:  $N = ?Nb + ?Na$ 
    using multiset-partition by blast+

  have f-eq-ma-na:  $F = \text{set-mset } ?Ma \cup \text{set-mset } ?Na$ 
    using x-f-eq-m-n x-ni-f by auto

  show ?case

```

```

proof (cases count (image-mset f ?Ma) b < count (image-mset f ?Na) b)
  case cnt-ba: True
  obtain xa where f xa = b and count ?Ma xa < count ?Na xa
  using ih[OF f-eq-ma-na cnt-ba] by blast
  thus ?thesis
  by (metis count-filter-mset not-less0)
next
  case cnt-ba: False
  have fx-eq-b: f x = b
  using cnt-b cnt-ba
  by (subst (asm) m-part, subst (asm) n-part,
    auto simp: filter-eq-replicate-mset split: if-splits)
  moreover have count M x < count N x
  using cnt-b cnt-ba
  by (subst (asm) m-part, subst (asm) n-part,
    auto simp: filter-eq-replicate-mset split: if-splits)
  ultimately show ?thesis
  by blast
qed
qed auto

```

lemma count-image-mset-lt-imp-lt:

```

assumes cnt-b: count (image-mset f M) b < count (image-mset f N) b
shows  $\exists x. f x = b \wedge \text{count } M x < \text{count } N x$ 
by (rule count-image-mset-lt-imp-lt-raw[of set-mset M  $\cup$  set-mset N, OF - refl
cnt-b]) auto

```

lemma count-image-mset-le-imp-lt-raw:

```

assumes
  finite A and
  A = set-mset M  $\cup$  set-mset N and
  count (image-mset f M) (f a) + count N a < count (image-mset f N) (f a) +
count M a
shows  $\exists b. f b = f a \wedge \text{count } M b < \text{count } N b$ 
using assms
proof (induct A arbitrary: M N rule: finite-induct)
  case (insert x F)
  note fin = this(1) and x-ni-f = this(2) and ih = this(3) and x-f-eq-m-n =
this(4) and
  cnt-lt = this(5)

```

```

let ?Ma = {#y  $\in$  # M. y  $\neq$  x#}
let ?Mb = {#y  $\in$  # M. y = x#}
let ?Na = {#y  $\in$  # N. y  $\neq$  x#}
let ?Nb = {#y  $\in$  # N. y = x#}

```

```

have m-part: M = ?Mb + ?Ma and n-part: N = ?Nb + ?Na
using multiset-partition by blast+

```

```

have f-eq-ma-na:  $F = \text{set-mset } ?Ma \cup \text{set-mset } ?Na$ 
  using x-f-eq-m-n x-ni-f by auto

show ?case
proof (cases  $f x = f a$ )
  case fx-ne-fa: False

    have cnt-fma-fa:  $\text{count } (\text{image-mset } f ?Ma) (f a) = \text{count } (\text{image-mset } f M) (f a)$ 
    a)
      using fx-ne-fa by (subst (2) m-part) (auto simp: filter-eq-replicate-mset)
    have cnt-fna-fa:  $\text{count } (\text{image-mset } f ?Na) (f a) = \text{count } (\text{image-mset } f N) (f a)$ 
    a)
      using fx-ne-fa by (subst (2) n-part) (auto simp: filter-eq-replicate-mset)
    have cnt-ma-a:  $\text{count } ?Ma a = \text{count } M a$ 
      using fx-ne-fa by (subst (2) m-part) (auto simp: filter-eq-replicate-mset)
    have cnt-na-a:  $\text{count } ?Na a = \text{count } N a$ 
      using fx-ne-fa by (subst (2) n-part) (auto simp: filter-eq-replicate-mset)

    obtain b where fb-eq-fa:  $f b = f a$  and cnt-b:  $\text{count } ?Ma b < \text{count } ?Na b$ 
      using ih[OF f-eq-ma-na] cnt-lt unfolding cnt-fma-fa cnt-fna-fa cnt-ma-a
    cnt-na-a by blast
    have fx-ne-fb:  $f x \neq f b$ 
      using fb-eq-fa fx-ne-fa by simp

    have cnt-ma-b:  $\text{count } ?Ma b = \text{count } M b$ 
      using fx-ne-fb by (subst (2) m-part) auto
    have cnt-na-b:  $\text{count } ?Na b = \text{count } N b$ 
      using fx-ne-fb by (subst (2) n-part) auto

    show ?thesis
      using fb-eq-fa cnt-b unfolding cnt-ma-b cnt-na-b by blast
  next
    case fx-eq-fa: True
    show ?thesis
    proof (cases  $x = a$ )
      case x-eq-a: True
      have count (image-mset f ?Ma) (f a) + count ?Na a
        < count (image-mset f ?Na) (f a) + count ?Ma a
        using cnt-lt x-eq-a by (subst (asm) (1 2) m-part, subst (asm) (1 2) n-part,
          auto simp: filter-eq-replicate-mset)
      thus ?thesis
        using ih[OF f-eq-ma-na] by (metis count-filter-mset nat-neq-iff)
    next
      case x-ne-a: False
      show ?thesis
      proof (cases  $\text{count } M x < \text{count } N x$ )
        case True
        thus ?thesis
          using fx-eq-fa by blast
      end
    end
  end
end

```

```

next
  case False
  hence cnt-x: count M x ≥ count N x
    by fastforce
  have count M x + count (image-mset f ?Ma) (f a) + count ?Na a
    < count N x + count (image-mset f ?Na) (f a) + count ?Ma a
    using cnt-lt x-ne-a fx-eq-fa by (subst (asm) (1 2) m-part, subst (asm) (1
2) n-part,
      auto simp: filter-eq-replicate-mset)
  hence count (image-mset f ?Ma) (f a) + count ?Na a
    < count (image-mset f ?Na) (f a) + count ?Ma a
    using cnt-x by linarith
  thus ?thesis
    using ih[OF f-eq-ma-na] by (metis count-filter-mset nat-neq-iff)
qed
qed
qed
qed auto

```

lemma *count-image-mset-le-imp-lt*:

```

assumes
  count (image-mset f M) (f a) ≤ count (image-mset f N) (f a) and
  count M a > count N a
shows  $\exists b. f b = f a \wedge \text{count } M b < \text{count } N b$ 
  using assms by (auto intro: count-image-mset-le-imp-lt-raw[of set-mset M ∪
set-mset N])

```

lemma *size-filter-unsat-elem*:

```

assumes x ∈# M and  $\neg P x$ 
shows size {#x ∈# M. P x#} < size M
proof -
  have size (filter-mset P M) ≠ size M
    using assms
    by (metis dual-order.strict-iff-order filter-mset-eq-conv mset-subset-size sub-
set-mset.nless-le)
  then show ?thesis
    by (meson leD nat-neq-iff size-filter-mset-lesseq)
qed

```

lemma *size-filter-ne-elem*: $x \in\# M \implies \text{size } \{ \#y \in\# M. y \neq x \# \} < \text{size } M$
 by (*simp add: size-filter-unsat-elem*[of *x M λy. y ≠ x*])

lemma *size-eq-ex-count-lt*:

```

assumes size M = size N and M ≠ N
shows  $\exists x. \text{count } M x < \text{count } N x$ 
proof -
  from  $\langle M \neq N \rangle$  obtain x where count M x ≠ count N x
    using count-inject by blast
  then consider (lt) count M x < count N x | (gt) count M x > count N x

```

```

    by linarith
  then show ?thesis
proof cases
  case lt
    then show ?thesis ..
next
  case gt
    from ⟨size M = size N⟩ have size {#y ∈# M. y = x#} + size {#y ∈# M.
y ≠ x#} =
      size {#y ∈# N. y = x#} + size {#y ∈# N. y ≠ x#}
    using multiset-partition by (metis size-union)
    with gt have *: size {#y ∈# M. y ≠ x#} < size {#y ∈# N. y ≠ x#}
    by (simp add: filter-eq-replicate-mset)
    then obtain y where count {#y ∈# M. y ≠ x#} y < count {#y ∈# N. y
≠ x#} y
    using size-lt-imp-ex-count-lt[OF *] by blast
    then have count M y < count N y
    by (metis count-filter-mset less-asm)
    then show ?thesis ..
qed
qed

```

67.11 Big operators

```

locale comm-monoid-mset = comm-monoid
begin

```

```

interpretation comp-fun-commute f
  by standard (simp add: fun-eq-iff left-commute)

```

```

interpretation comp?: comp-fun-commute f ∘ g
  by (fact comp-comp-fun-commute)

```

```

context
begin

```

```

definition F :: 'a multiset ⇒ 'a
  where eq-fold: F M = fold-mset f 1 M

```

```

lemma empty [simp]: F {#} = 1
  by (simp add: eq-fold)

```

```

lemma singleton [simp]: F {#x#} = x

```

```

proof —
  interpret comp-fun-commute
    by standard (simp add: fun-eq-iff left-commute)
  show ?thesis by (simp add: eq-fold)
qed

```

lemma *union* [simp]: $F (M + N) = F M * F N$

proof –

interpret *comp-fun-commute* *f*

by *standard* (*simp add: fun-eq-iff left-commute*)

show *?thesis*

by (*induct N*) (*simp-all add: left-commute eq-fold*)

qed

lemma *add-mset* [simp]: $F (add-mset\ x\ N) = x * F N$

unfolding *add-mset-add-single*[*of x N*] *union* **by** (*simp add: ac-simps*)

lemma *insert* [simp]:

shows $F (image-mset\ g\ (add-mset\ x\ A)) = g\ x * F (image-mset\ g\ A)$

by (*simp add: eq-fold*)

lemma *remove*:

assumes $x \in\# A$

shows $F A = x * F (A - \{\#x\# \})$

using *multi-member-split*[*OF assms*] **by** *auto*

lemma *neutral*:

$\forall x \in\# A. x = \mathbf{1} \implies F A = \mathbf{1}$

by (*induct A*) *simp-all*

lemma *neutral-const* [simp]:

$F (image-mset\ (\lambda_. \mathbf{1})\ A) = \mathbf{1}$

by (*simp add: neutral*)

private lemma *F-image-mset-product*:

$F \{\#g\ x\ j * F \{\#g\ i\ j. i \in\# A\#\}. j \in\# B\#\} =$

$F (image-mset\ (g\ x)\ B) * F \{\#F \{\#g\ i\ j. i \in\# A\#\}. j \in\# B\#\}$

by (*induction B*) (*simp-all add: left-commute semigroup.assoc semigroup-axioms*)

lemma *swap*:

$F (image-mset\ (\lambda i. F (image-mset\ (g\ i)\ B))\ A) =$

$F (image-mset\ (\lambda j. F (image-mset\ (\lambda i. g\ i\ j)\ A))\ B)$

apply (*induction A, simp*)

apply (*induction B, auto simp add: F-image-mset-product ac-simps*)

done

lemma *distrib*: $F (image-mset\ (\lambda x. g\ x * h\ x)\ A) = F (image-mset\ g\ A) * F (image-mset\ h\ A)$

by (*induction A*) (*auto simp: ac-simps*)

lemma *union-disjoint*:

$A \cap\# B = \{\#\} \implies F (image-mset\ g\ (A \cup\# B)) = F (image-mset\ g\ A) * F (image-mset\ g\ B)$

by (*induction A*) (*auto simp: ac-simps*)

end
end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* $((+) :: 'a \text{ multiset} \Rightarrow - \Rightarrow -)$
by *standard* (*simp add: add-ac comp-def*)

declare *comp-fun-commute.fold-mset-add-mset*[*OF comp-fun-commute-plus-mset, simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: $x \in\# \text{ fold-mset } (+) \ M \ NN \longleftrightarrow x \in\# \ M \vee (\exists N. N \in\# \ NN \wedge x \in\# \ N)$
by (*induct NN*) *auto*

context *comm-monoid-add*
begin

sublocale *sum-mset*: *comm-monoid-mset plus 0*
defines *sum-mset* = *sum-mset.F* ..

lemma *sum-unfold-sum-mset*:
 $\text{sum } f \ A = \text{sum-mset } (\text{image-mset } f \ (\text{mset-set } A))$
by (*cases finite A*) (*induct A rule: finite-induct, simp-all*)

end

notation *sum-mset* ($\langle \sum \# \rangle$)

syntax (*ASCII*)
 $\text{-sum-mset-image} :: \text{pttrn} \Rightarrow 'b \text{ set} \Rightarrow 'a \Rightarrow 'a :: \text{comm-monoid-add}$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } SUM \rangle \rangle SUM \text{ :-}\#-. \text{ -} \rangle [0, 51, 10] 10)$

syntax
 $\text{-sum-mset-image} :: \text{pttrn} \Rightarrow 'b \text{ set} \Rightarrow 'a \Rightarrow 'a :: \text{comm-monoid-add}$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \sum \rangle \rangle \sum \text{ :-}\#-. \text{ -} \rangle [0, 51, 10] 10)$

syntax-consts
 $\text{-sum-mset-image} \Rightarrow \text{sum-mset}$

translations
 $\sum i \in\# \ A. \ b \Rightarrow \text{CONST } \text{sum-mset } (\text{CONST } \text{image-mset } (\lambda i. \ b) \ A)$

context *comm-monoid-add*
begin

lemma *sum-mset-sum-list*:
 $\text{sum-mset } (\text{mset } xs) = \text{sum-list } xs$
by (*induction xs*) *auto*

end

context *canonically-ordered-monoid-add*

begin

lemma *sum-mset-0-iff* [simp]:

$sum\text{-}mset\ M = 0 \iff (\forall x \in set\text{-}mset\ M. x = 0)$

by (induction M) auto

end

context *ordered-comm-monoid-add*

begin

lemma *sum-mset-mono*:

$sum\text{-}mset\ (image\text{-}mset\ f\ K) \leq sum\text{-}mset\ (image\text{-}mset\ g\ K)$

if $\bigwedge i. i \in\# K \implies f\ i \leq g\ i$

using that **by** (induction K) (simp-all add: add-mono)

end

context *cancel-comm-monoid-add*

begin

lemma *sum-mset-diff*:

$sum\text{-}mset\ (M - N) = sum\text{-}mset\ M - sum\text{-}mset\ N$ **if** $N \subseteq\# M$ **for** $M\ N :: 'a\ multiset$

using that **by** (auto simp add: subset-mset.le-iff-add)

end

context *semiring-0*

begin

lemma *sum-mset-distrib-left*:

$c * (\sum x \in\# M. f\ x) = (\sum x \in\# M. c * f\ x)$

by (induction M) (simp-all add: algebra-simps)

lemma *sum-mset-distrib-right*:

$(\sum x \in\# M. f\ x) * c = (\sum x \in\# M. f\ x * c)$

by (induction M) (simp-all add: algebra-simps)

end

lemma *sum-mset-product*:

fixes $f :: 'a :: \{comm\text{-}monoid\text{-}add, times\} \Rightarrow 'b :: semiring\text{-}0$

shows $(\sum i \in\# A. f\ i) * (\sum i \in\# B. g\ i) = (\sum i \in\# A. \sum j \in\# B. f\ i * g\ j)$

by (subst sum-mset.swap) (simp add: sum-mset-distrib-left sum-mset-distrib-right)

context *semiring-1*

begin

lemma *sum-mset-replicate-mset* [simp]:

$sum\text{-}mset\ (replicate\text{-}mset\ n\ a) = of\text{-}nat\ n * a$

by (induction n) (simp-all add: algebra-simps)

lemma *sum-mset-delta*:

$sum\text{-}mset\ (image\text{-}mset\ (\lambda x. \text{if } x = y \text{ then } c \text{ else } 0)\ A) = c * of\text{-}nat\ (count\ A\ y)$

by (induction A) (simp-all add: algebra-simps)

lemma *sum-mset-delta'*:

$sum\text{-}mset\ (image\text{-}mset\ (\lambda x. \text{if } y = x \text{ then } c \text{ else } 0)\ A) = c * of\text{-}nat\ (count\ A\ y)$

by (induction A) (simp-all add: algebra-simps)

end

lemma *of-nat-sum-mset* [simp]:

$of\text{-}nat\ (sum\text{-}mset\ A) = sum\text{-}mset\ (image\text{-}mset\ of\text{-}nat\ A)$

by (induction A) auto

lemma *size-eq-sum-mset*:

$size\ M = (\sum a \in \#M. 1)$

using *image-mset-const-eq* [of $1 :: nat\ M$] **by** *simp*

lemma *size-mset-set* [simp]:

$size\ (mset\text{-}set\ A) = card\ A$

by (simp only: *size-eq-sum-mset card-eq-sum sum-unfold-sum-mset*)

lemma *sum-mset-constant* [simp]:

fixes $y :: 'b::semiring-1$

shows $\langle (\sum x \in \#A. y) = of\text{-}nat\ (size\ A) * y \rangle$

by (induction A) (auto simp: algebra-simps)

lemma *set-mset-Union-mset*[simp]: $set\text{-}mset\ (\sum \# MM) = (\bigcup M \in set\text{-}mset\ MM. set\text{-}mset\ M)$

by (induct MM) auto

lemma *in-Union-mset-iff*[iff]: $x \in \# \sum \# MM \longleftrightarrow (\exists M. M \in \# MM \wedge x \in \# M)$

by (induct MM) auto

lemma *count-sum*:

$count\ (sum\ f\ A)\ x = sum\ (\lambda a. count\ (f\ a)\ x)\ A$

by (induct A rule: *infinite-finite-induct*) *simp-all*

lemma *sum-eq-empty-iff*:

assumes *finite* A

shows $sum\ f\ A = \{\#\} \longleftrightarrow (\forall a \in A. f\ a = \{\#\})$

using *assms* **by** *induct simp-all*

lemma *mset-concat*: $mset\ (concat\ xss) = (\sum xs \leftarrow xss. mset\ xs)$

by (induction xss) auto

```

lemma set-mset-sum-list [simp]: set-mset (sum-list xs) = ( $\bigcup x \in \text{set } xs. \text{set-mset } x$ )
  by (induction xs) auto

lemma filter-mset-sum-list: filter-mset P (sum-list xs) = sum-list (map (filter-mset
P) xs)
  by (induction xs) simp-all

lemma sum-mset-singleton-mset [simp]: ( $\sum x \in \#A. \{ \#f \ x \# \}$ ) = image-mset f A
  by (induction A) auto

lemma sum-list-singleton-mset [simp]: ( $\sum x \leftarrow xs. \{ \#f \ x \# \}$ ) = image-mset f (mset
xs)
  by (induction xs) auto

lemma Union-mset-empty-conv[simp]:  $\sum \# \ M = \{ \# \} \longleftrightarrow (\forall i \in \#M. i = \{ \# \})$ 
  by (induction M) auto

lemma Union-image-single-mset[simp]:  $\sum \# \ (\text{image-mset } (\lambda x. \{ \#x \# \}) \ m) = m$ 
  by(induction m) auto

lemma size-mset-sum-mset-conv [simp]: size ( $\sum \# \ A :: 'a \text{ multiset}$ ) = ( $\sum X \in \#A. \text{size } X$ )
  by (induction A) auto

lemma sum-mset-image-mset-mono-strong:
  assumes  $A \subseteq \# B$  and f-subseq-g:  $\bigwedge x. x \in \# A \implies f \ x \subseteq \# g \ x$ 
  shows  $(\sum x \in \#A. f \ x) \subseteq \# (\sum x \in \#B. g \ x)$ 
proof –
  define B' where
     $B' = B - A$ 

  have  $B = A + B'$ 
    using B'-def assms(1) by fastforce

  have  $\sum \# \ (\text{image-mset } f \ A) \subseteq \# \sum \# \ (\text{image-mset } g \ A)$ 
    using f-subseq-g by (induction A) (auto intro!: subset-mset.add-mono)
  also have  $\dots \subseteq \# \sum \# \ (\text{image-mset } g \ A) + \sum \# \ (\text{image-mset } g \ B')$ 
    by simp
  also have  $\dots = \sum \# \ (\text{image-mset } g \ A + \text{image-mset } g \ B')$ 
    by simp
  also have  $\dots = \sum \# \ (\text{image-mset } g \ (A + B'))$ 
    by simp
  also have  $\dots = \sum \# \ (\text{image-mset } g \ B)$ 
    unfolding  $\langle B = A + B' \rangle$  ..
  finally show ?thesis .
qed

context comm-monoid-mult

```

begin

sublocale *prod-mset: comm-monoid-mset times 1*
defines *prod-mset = prod-mset.F ..*

lemma *prod-mset-empty:*
prod-mset {#} = 1
by (*fact prod-mset.empty*)

lemma *prod-mset-singleton:*
prod-mset {#x#} = x
by (*fact prod-mset.singleton*)

lemma *prod-mset-Un:*
*prod-mset (A + B) = prod-mset A * prod-mset B*
by (*fact prod-mset.union*)

lemma *prod-mset-prod-list:*
prod-mset (mset xs) = prod-list xs
by (*induct xs*) *auto*

lemma *prod-mset-replicate-mset [simp]:*
prod-mset (replicate-mset n a) = a ^ n
by (*induct n*) *simp-all*

lemma *prod-unfold-prod-mset:*
prod f A = prod-mset (image-mset f (mset-set A))
by (*cases finite A*) (*induct A rule: finite-induct, simp-all*)

lemma *prod-mset-multiplicity:*
prod-mset M = prod ($\lambda x. x ^ \text{count } M x$) (set-mset M)
by (*simp add: fold-mset-def prod.eq-fold prod-mset.eq-fold funpow-times-power comp-def*)

lemma *prod-mset-delta:* *prod-mset (image-mset ($\lambda x. \text{if } x = y \text{ then } c \text{ else } 1$) A) =*
 $c ^ \text{count } A y$
by (*induction A*) *simp-all*

lemma *prod-mset-delta':* *prod-mset (image-mset ($\lambda x. \text{if } y = x \text{ then } c \text{ else } 1$) A) =*
 $c ^ \text{count } A y$
by (*induction A*) *simp-all*

lemma *prod-mset-subset-imp-dvd:*
assumes $A \subseteq\# B$
shows *prod-mset A dvd prod-mset B*

proof –

from *assms* **have** $B = (B - A) + A$ **by** (*simp add: subset-mset.diff-add*)
also have *prod-mset ... = prod-mset (B - A) * prod-mset A* **by** *simp*
also have *prod-mset A dvd ...* **by** *simp*

finally show *?thesis* .
qed

lemma *dvd-prod-mset*:
assumes $x \in\# A$
shows $x \text{ dvd } \text{prod-mset } A$
using *assms prod-mset-subset-imp-dvd* [of $\{\#x\}$ A] **by** *simp*

end

notation *prod-mset* ($\langle \prod \# \rangle$)

syntax (*ASCII*)

-prod-mset-image :: $\text{pttrn} \Rightarrow 'b \text{ set} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-mult}$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \text{PROD} \rangle \text{PROD } -: \# \cdot - \rangle [0, 51, 10] 10)$

syntax

-prod-mset-image :: $\text{pttrn} \Rightarrow 'b \text{ set} \Rightarrow 'a \Rightarrow 'a::\text{comm-monoid-mult}$
 $(\langle \langle \text{indent}=3 \text{ notation}=\langle \text{binder } \prod \rangle \prod - \in \# \cdot - \rangle [0, 51, 10] 10)$

syntax-consts

-prod-mset-image \Rightarrow *prod-mset*

translations

$\prod i \in\# A. b \Rightarrow \text{CONST } \text{prod-mset } (\text{CONST } \text{image-mset } (\lambda i. b) A)$

lemma *prod-mset-constant* [*simp*]: $(\prod - \in\# A. c) = c \wedge \text{size } A$
by (*simp add: image-mset-const-eq*)

lemma (*in semidom*) *prod-mset-zero-iff* [*iff*]:

$\text{prod-mset } A = 0 \longleftrightarrow 0 \in\# A$

by (*induct A*) *auto*

lemma (*in semidom-divide*) *prod-mset-diff*:

assumes $B \subseteq\# A$ **and** $0 \notin\# B$

shows $\text{prod-mset } (A - B) = \text{prod-mset } A \text{ div } \text{prod-mset } B$

proof –

from *assms* **obtain** C **where** $A = B + C$

by (*metis subset-mset.add-diff-inverse*)

with *assms* **show** *?thesis* **by** *simp*

qed

lemma (*in semidom-divide*) *prod-mset-minus*:

assumes $a \in\# A$ **and** $a \neq 0$

shows $\text{prod-mset } (A - \{\#a\}) = \text{prod-mset } A \text{ div } a$

using *assms prod-mset-diff* [of $\{\#a\}$ A] **by** *auto*

lemma (*in normalization-semidom*) *normalize-prod-mset-normalize*:

$\text{normalize } (\text{prod-mset } (\text{image-mset } \text{normalize } A)) = \text{normalize } (\text{prod-mset } A)$

proof (*induction A*)

case (*add x A*)

have $\text{normalize } (\text{prod-mset } (\text{image-mset } \text{normalize } (\text{add-mset } x A))) =$

```

      normalize (x * normalize (prod-mset (image-mset normalize A)))
    by simp
  also note add.IH
  finally show ?case by simp
qed auto

```

```

lemma (in algebraic-semidom) is-unit-prod-mset-iff:
  is-unit (prod-mset A)  $\longleftrightarrow$  ( $\forall x \in \# A. \text{is-unit } x$ )
  by (induct A) (auto simp: is-unit-mult-iff)

```

```

lemma (in normalization-semidom-multiplicative) normalize-prod-mset:
  normalize (prod-mset A) = prod-mset (image-mset normalize A)
  by (induct A) (simp-all add: normalize-mult)

```

```

lemma (in normalization-semidom-multiplicative) normalized-prod-msetI:
  assumes  $\bigwedge a. a \in \# A \implies \text{normalize } a = a$ 
  shows normalize (prod-mset A) = prod-mset A
  proof -
    from assms have image-mset normalize A = A
      by (induct A) simp-all
    then show ?thesis by (simp add: normalize-prod-mset)
  qed

```

```

lemma image-prod-mset-multiplicity:
  prod-mset (image-mset f M) = prod ( $\lambda x. f x \wedge \text{count } M x$ ) (set-mset M)
  proof (induction M)
    case (add x M)
    show ?case
    proof (cases x  $\in$  set-mset M)
      case True
      have  $(\prod_{y \in \text{set-mset } (add-mset x M)}. f y \wedge \text{count } (add-mset x M) y) =$ 
         $(\prod_{y \in \text{set-mset } M}. (\text{if } y = x \text{ then } f x \text{ else } 1) * f y \wedge \text{count } M y)$ 
        using True add by (intro prod.cong) auto
      also have  $\dots = f x * (\prod_{y \in \text{set-mset } M}. f y \wedge \text{count } M y)$ 
        using True by (subst prod.distrib) auto
      also note add.IH [symmetric]
      finally show ?thesis using True by simp
    next
      case False
      hence  $(\prod_{y \in \text{set-mset } (add-mset x M)}. f y \wedge \text{count } (add-mset x M) y) =$ 
         $f x * (\prod_{y \in \text{set-mset } M}. f y \wedge \text{count } (add-mset x M) y)$ 
        by (auto simp: not-in-iff)
      also have  $(\prod_{y \in \text{set-mset } M}. f y \wedge \text{count } (add-mset x M) y) =$ 
         $(\prod_{y \in \text{set-mset } M}. f y \wedge \text{count } M y)$ 
        using False by (intro prod.cong) auto
      also note add.IH [symmetric]
      finally show ?thesis by simp
    qed
  qed auto

```

67.12 Multiset as order-ignorant lists

context *linorder*
begin

lemma *mset-insort [simp]*:
 $mset\ (insort\ key\ k\ x\ xs) = add\ mset\ x\ (mset\ xs)$
by (*induct xs*) *simp-all*

lemma *mset-sort [simp]*:
 $mset\ (sort\ key\ k\ xs) = mset\ xs$
by (*induct xs*) *simp-all*

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma *properties-for-sort-key*:
assumes $mset\ ys = mset\ xs$
and $\bigwedge k. k \in set\ ys \implies filter\ (\lambda x. f\ k = f\ x)\ ys = filter\ (\lambda x. f\ k = f\ x)\ xs$
and *sorted (map f ys)*
shows $sort\ key\ f\ xs = ys$
using *assms*
proof (*induct xs arbitrary: ys*)
case *Nil* **then show** ?case **by** *simp*
next
case (*Cons x xs*)
from *Cons.prem1* **have**
 $\forall k \in set\ ys. filter\ (\lambda x. f\ k = f\ x)\ (remove1\ x\ ys) = filter\ (\lambda x. f\ k = f\ x)\ xs$
by (*simp add: filter-remove1*)
with *Cons.prem2* **have** $sort\ key\ f\ xs = remove1\ x\ ys$
by (*auto intro!: Cons.hyps simp add: sorted-map-remove1*)
moreover from *Cons.prem3* **have** $x \in \# mset\ ys$
by *auto*
then have $x \in set\ ys$
by *simp*
ultimately show ?case **using** *Cons.prem4* **by** (*simp add: insort-key-remove1*)
qed

lemma *properties-for-sort*:
assumes *multiset: mset ys = mset xs*
and *sorted ys*
shows $sort\ xs = ys$
proof (*rule properties-for-sort-key*)
from *multiset* **show** $mset\ ys = mset\ xs$.
from $\langle sorted\ ys \rangle$ **show** $sorted\ (map\ (\lambda x. x)\ ys)$ **by** *simp*
from *multiset* **have** $length\ (filter\ (\lambda y. k = y)\ ys) = length\ (filter\ (\lambda x. k = x)\ xs)$
for k
by (*metis mset-filter size-mset*)
then have $replicate\ (length\ (filter\ (\lambda y. k = y)\ ys))\ k =$
 $replicate\ (length\ (filter\ (\lambda x. k = x)\ xs))\ k$ **for** k
by *simp*

then show $k \in \text{set } ys \implies \text{filter } (\lambda y. k = y) \text{ } ys = \text{filter } (\lambda x. k = x) \text{ } xs$ for k
 by (simp add: replicate-length-filter)
 qed

lemma *sort-key-inj-key-eq*:
 assumes *mset-equal*: $\text{mset } xs = \text{mset } ys$
 and *inj-on* f (*set* xs)
 and *sorted* ($\text{map } f \text{ } ys$)
 shows $\text{sort-key } f \text{ } xs = ys$
proof (rule *properties-for-sort-key*)
 from *mset-equal*
 show $\text{mset } ys = \text{mset } xs$ by *simp*
 from $\langle \text{sorted } (\text{map } f \text{ } ys) \rangle$
 show $\text{sorted } (\text{map } f \text{ } ys)$.
 show $[x \leftarrow ys . f \text{ } k = f \text{ } x] = [x \leftarrow xs . f \text{ } k = f \text{ } x]$ if $k \in \text{set } ys$ for k
proof –
 from *mset-equal*
 have *set-equal*: $\text{set } xs = \text{set } ys$ by (rule *mset-eq-setD*)
 with that have $\text{insert } k \text{ } (\text{set } ys) = \text{set } ys$ by *auto*
 with $\langle \text{inj-on } f \text{ } (\text{set } xs) \rangle$ have *inj*: $\text{inj-on } f \text{ } (\text{insert } k \text{ } (\text{set } ys))$
 by (simp add: *set-equal*)
 from *inj* have $[x \leftarrow ys . f \text{ } k = f \text{ } x] = \text{filter } (\text{HOL.eq } k) \text{ } ys$
 by (auto intro!: *inj-on-filter-key-eq*)
 also have $\dots = \text{replicate } (\text{count } (\text{mset } ys) \text{ } k) \text{ } k$
 by (simp add: *replicate-count-mset-eq-filter-eq*)
 also have $\dots = \text{replicate } (\text{count } (\text{mset } xs) \text{ } k) \text{ } k$
 using *mset-equal* by *simp*
 also have $\dots = \text{filter } (\text{HOL.eq } k) \text{ } xs$
 by (simp add: *replicate-count-mset-eq-filter-eq*)
 also have $\dots = [x \leftarrow xs . f \text{ } k = f \text{ } x]$
 using *inj* by (auto intro!: *inj-on-filter-key-eq* [symmetric] simp add: *set-equal*)
 finally show ?thesis .
 qed
 qed

lemma *sort-key-eq-sort-key*:
 assumes $\text{mset } xs = \text{mset } ys$
 and *inj-on* f (*set* xs)
 shows $\text{sort-key } f \text{ } xs = \text{sort-key } f \text{ } ys$
 by (rule *sort-key-inj-key-eq*) (simp-all add: *assms*)

lemma *sort-key-by-quicksort*:
 $\text{sort-key } f \text{ } xs = \text{sort-key } f \text{ } [x \leftarrow xs. f \text{ } x < f \text{ } (xs ! (\text{length } xs \text{ div } 2))]$
 @ $[x \leftarrow xs. f \text{ } x = f \text{ } (xs ! (\text{length } xs \text{ div } 2))]$
 @ $\text{sort-key } f \text{ } [x \leftarrow xs. f \text{ } x > f \text{ } (xs ! (\text{length } xs \text{ div } 2))]$ (is $\text{sort-key } f \text{ } ?lhs = ?rhs$)
proof (rule *properties-for-sort-key*)
 show $\text{mset } ?rhs = \text{mset } ?lhs$
 by (rule *multiset-eqI*) *auto*
 show $\text{sorted } (\text{map } f \text{ } ?rhs)$


```

  by (auto simp add: sorted-append intro: sorted-map-same)
next
  fix l
  assume l ∈ set ?rhs
  let ?pivot = f (xs ! (length xs div 2))
  have *:  $\bigwedge x. f\ l = f\ x \longleftrightarrow f\ x = f\ l$  by auto
  have  $[x \leftarrow \text{sort-key } f\ xs . f\ x = f\ l] = [x \leftarrow xs. f\ x = f\ l]$ 
  unfolding filter-sort by (rule properties-for-sort-key) (auto intro: sorted-map-same)
  with * have **:  $[x \leftarrow \text{sort-key } f\ xs . f\ l = f\ x] = [x \leftarrow xs. f\ l = f\ x]$  by simp
  have  $\bigwedge x\ P. P\ (f\ x) \text{ ?pivot} \wedge f\ l = f\ x \longleftrightarrow P\ (f\ l) \text{ ?pivot} \wedge f\ l = f\ x$  by auto
  then have  $\bigwedge P. [x \leftarrow \text{sort-key } f\ xs . P\ (f\ x) \text{ ?pivot} \wedge f\ l = f\ x] =$ 
     $[x \leftarrow \text{sort-key } f\ xs. P\ (f\ l) \text{ ?pivot} \wedge f\ l = f\ x]$  by simp
  note *** = this [of (<)] this [of (>)] this [of (=)]
  show  $[x \leftarrow ?rhs. f\ l = f\ x] = [x \leftarrow ?lhs. f\ l = f\ x]$ 
  proof (cases f l ?pivot rule: linorder-cases)
    case less
    then have  $f\ l \neq \text{?pivot}$  and  $\neg f\ l > \text{?pivot}$  by auto
    with less show ?thesis
    by (simp add: filter-sort [symmetric] ** ***)
  next
    case equal then show ?thesis
    by (simp add: * less-le)
  next
    case greater
    then have  $f\ l \neq \text{?pivot}$  and  $\neg f\ l < \text{?pivot}$  by auto
    with greater show ?thesis
    by (simp add: filter-sort [symmetric] ** ***)
  qed
qed

```

```

lemma sort-by-quicksort:
  sort xs = sort  $[x \leftarrow xs. x < xs ! (\text{length } xs \text{ div } 2)]$ 
  @  $[x \leftarrow xs. x = xs ! (\text{length } xs \text{ div } 2)]$ 
  @ sort  $[x \leftarrow xs. x > xs ! (\text{length } xs \text{ div } 2)]$  (is sort ?lhs = ?rhs)
  using sort-key-by-quicksort [of  $\lambda x. x$ , symmetric] by simp

```

```

lemma sort-append:
  assumes  $\bigwedge x\ y. x \in \text{set } xs \implies y \in \text{set } ys \implies x \leq y$ 
  shows sort (xs @ ys) = sort xs @ sort ys
  using assms by (intro properties-for-sort) (auto simp: sorted-append)

```

```

lemma sort-append-replicate-left:
  ( $\bigwedge y. y \in \text{set } xs \implies x \leq y$ )  $\implies$  sort (replicate n x @ xs) = replicate n x @ sort
  xs
  by (subst sort-append) auto

```

```

lemma sort-append-replicate-right:
  ( $\bigwedge y. y \in \text{set } xs \implies x \geq y$ )  $\implies$  sort (xs @ replicate n x) = sort xs @ replicate n
  x

```

by (*subst sort-append*) *auto*

A stable parameterized quicksort

definition *part* :: ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b list \Rightarrow 'b list \times 'b list \times 'b list **where**
part *f* *pivot* *xs* = ([*x* \leftarrow *xs*. *f* *x* < *pivot*], [*x* \leftarrow *xs*. *f* *x* = *pivot*], [*x* \leftarrow *xs*. *pivot* < *f* *x*])

lemma *part-code* [*code*]:

part *f* *pivot* [] = ([], [], [])
part *f* *pivot* (*x* # *xs*) = (let (*lts*, *eqs*, *gts*) = *part* *f* *pivot* *xs*; *x'* = *f* *x* in
 if *x'* < *pivot* then (*x* # *lts*, *eqs*, *gts*)
 else if *x'* > *pivot* then (*lts*, *eqs*, *x* # *gts*)
 else (*lts*, *x* # *eqs*, *gts*))
by (*auto simp add: part-def Let-def split-def*)

lemma *sort-key-by-quicksort-code* [*code*]:

sort-key *f* *xs* =
 (case *xs* of
 [] \Rightarrow []
 | [*x*] \Rightarrow *xs*
 | [*x*, *y*] \Rightarrow (if *f* *x* \leq *f* *y* then *xs* else [*y*, *x*])
 | - \Rightarrow
 let (*lts*, *eqs*, *gts*) = *part* *f* (*f* (*xs* ! (*length* *xs* div 2))) *xs*
 in *sort-key* *f* *lts* @ *eqs* @ *sort-key* *f* *gts*)

proof (*cases xs*)

case *Nil* **then show** ?thesis **by** *simp*

next

case (*Cons* - *ys*) **note** *hyps* = *Cons* **show** ?thesis

proof (*cases ys*)

case *Nil* **with** *hyps* **show** ?thesis **by** *simp*

next

case (*Cons* - *zs*) **note** *hyps* = *hyps* *Cons* **show** ?thesis

proof (*cases zs*)

case *Nil* **with** *hyps* **show** ?thesis **by** *auto*

next

case *Cons*

from *sort-key-by-quicksort* [*of f xs*]

have *sort-key* *f* *xs* = (let (*lts*, *eqs*, *gts*) = *part* *f* (*f* (*xs* ! (*length* *xs* div 2))) *xs*
 in *sort-key* *f* *lts* @ *eqs* @ *sort-key* *f* *gts*)

by (*simp only: split-def Let-def part-def fst-conv snd-conv*)

with *hyps* *Cons* **show** ?thesis **by** (*simp only: list.cases*)

qed

qed

qed

end

hide-const (*open*) *part*

lemma *sort-sorted-list-of-multiset-eq* [simp]:
 $\langle \text{sort } (\text{sorted-list-of-multiset } M) = \text{sorted-list-of-multiset } M \rangle$ **for** $M :: \langle 'a :: \text{linorder multiset} \rangle$
by (rule *properties-for-sort*) *simp-all*

lemma *mset-remdups-subset-eq*: $\text{mset } (\text{remdups } xs) \subseteq \# \text{ mset } xs$
by (induct xs) (auto intro: *subset-mset.order-trans*)

lemma *mset-update*:
 $i < \text{length } ls \implies \text{mset } (ls[i := v]) = \text{add-mset } v \ (\text{mset } ls - \{\#ls ! i\# \})$
proof (induct ls arbitrary: i)
case *Nil* **then show** $?case$ **by** *simp*
next
case (*Cons* x xs)
show $?case$
proof (*cases* i)
case 0 **then show** $?thesis$ **by** *simp*
next
case (*Suc* i')
with *Cons* **show** $?thesis$
by (*cases* $\langle x = xs ! i' \rangle$) *auto*
qed
qed

lemma *mset-swap*:
 $i < \text{length } ls \implies j < \text{length } ls \implies$
 $\text{mset } (ls[j := ls ! i, i := ls ! j]) = \text{mset } ls$
by (*cases* $i = j$) (*simp-all add: mset-update nth-mem-mset*)

lemma *mset-eq-finite*:
 $\langle \text{finite } \{ys. \text{mset } ys = \text{mset } xs\} \rangle$
proof –
have $\langle \{ys. \text{mset } ys = \text{mset } xs\} \subseteq \{ys. \text{set } ys \subseteq \text{set } xs \wedge \text{length } ys \leq \text{length } xs\} \rangle$
by (auto *simp add: dest: mset-eq-setD mset-eq-length*)
moreover have $\langle \text{finite } \{ys. \text{set } ys \subseteq \text{set } xs \wedge \text{length } ys \leq \text{length } xs\} \rangle$
using *finite-lists-length-le* **by** *blast*
ultimately show $?thesis$
by (rule *finite-subset*)
qed

67.13 The multiset order

definition *mult1* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ multiset} \times 'a \text{ multiset}) \text{ set}$ **where**
 $\text{mult1 } r = \{(N, M). \exists a M0 K. M = \text{add-mset } a M0 \wedge N = M0 + K \wedge$
 $(\forall b. b \in \# K \longrightarrow (b, a) \in r)\}$

definition *mult* :: $('a \times 'a) \text{ set} \Rightarrow ('a \text{ multiset} \times 'a \text{ multiset}) \text{ set}$ **where**
 $\text{mult } r = (\text{mult1 } r)^+$

definition $multp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow bool$ **where**
 $multp\ r\ M\ N \longleftrightarrow (M, N) \in mult\ \{(x, y). r\ x\ y\}$

declare $multp\text{-}def[pred\text{-}set\text{-}conv]$

lemma $mult1I$:

assumes $M = add\text{-}mset\ a\ M0$ **and** $N = M0 + K$ **and** $\bigwedge b. b \in\# K \implies (b, a) \in r$
shows $(N, M) \in mult1\ r$
using $assms$ **unfolding** $mult1\text{-}def$ **by** $blast$

lemma $mult1E$:

assumes $(N, M) \in mult1\ r$
obtains $a\ M0\ K$ **where** $M = add\text{-}mset\ a\ M0$ $N = M0 + K$ $\bigwedge b. b \in\# K \implies (b, a) \in r$
using $assms$ **unfolding** $mult1\text{-}def$ **by** $blast$

lemma $mono\text{-}mult1$:

assumes $r \subseteq r'$ **shows** $mult1\ r \subseteq mult1\ r'$
unfolding $mult1\text{-}def$ **using** $assms$ **by** $blast$

lemma $mono\text{-}mult$:

assumes $r \subseteq r'$ **shows** $mult\ r \subseteq mult\ r'$
unfolding $mult\text{-}def$ **using** $mono\text{-}mult1[OF\ assms]$ $trancl\text{-}mono$ **by** $blast$

lemma $mono\text{-}multp[mono]$: $r \leq r' \implies multp\ r \leq multp\ r'$

unfolding $le\text{-}fun\text{-}def\ le\text{-}bool\text{-}def$

proof ($intro\ allI\ impI$)

fix $M\ N :: 'a\ multiset$

assume $\forall x\ xa. r\ x\ xa \longrightarrow r'\ x\ xa$

hence $\{(x, y). r\ x\ y\} \subseteq \{(x, y). r'\ x\ y\}$

by $blast$

thus $multp\ r\ M\ N \implies multp\ r'\ M\ N$

unfolding $multp\text{-}def$

by ($fact\ mono\text{-}mult[THEN\ subsetD, rotated]$)

qed

lemma $not\text{-}less\text{-}empty\ [iff]$: $(M, \{\#\}) \notin mult1\ r$

by ($simp\ add: mult1\text{-}def$)

67.13.1 Well-foundedness

lemma $less\text{-}add$:

assumes $mult1: (N, add\text{-}mset\ a\ M0) \in mult1\ r$

shows

$(\exists M. (M, M0) \in mult1\ r \wedge N = add\text{-}mset\ a\ M) \vee$

$(\exists K. (\forall b. b \in\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$

proof –

let $?r = \lambda K\ a. \forall b. b \in\# K \longrightarrow (b, a) \in r$

```

let ?R =  $\lambda N M. \exists a M0 K. M = \text{add-mset } a M0 \wedge N = M0 + K \wedge ?r K a$ 
obtain a' M0' K where M0:  $\text{add-mset } a M0 = \text{add-mset } a' M0'$ 
  and N:  $N = M0' + K$ 
  and r:  $?r K a'$ 
  using mult1 unfolding mult1-def by auto
show ?thesis (is ?case1  $\vee$  ?case2)
proof –
  from M0 consider  $M0 = M0' a = a'$ 
    | K' where  $M0 = \text{add-mset } a' K' M0' = \text{add-mset } a K'$ 
    by atomize-elim (simp only: add-eq-conv-ex)
  then show ?thesis
proof cases
  case 1
    with N r have  $?r K a \wedge N = M0 + K$  by simp
    then have ?case2 ..
    then show ?thesis ..
  next
    case 2
      from N 2(2) have n:  $N = \text{add-mset } a (K' + K)$  by simp
      with r 2(1) have ?R (K' + K) M0 by blast
      with n have ?case1 by (simp add: mult1-def)
      then show ?thesis ..
  qed
qed
qed

```

lemma all-accessible:

```

assumes wf r
shows  $\forall M. M \in \text{Wellfounded.acc } (\text{mult1 } r)$ 
proof
  let ?R = mult1 r
  let ?W = Wellfounded.acc ?R
  {
    fix M M0 a
    assume M0:  $M0 \in ?W$ 
    and wf-hyp:  $\bigwedge b. (b, a) \in r \implies (\forall M \in ?W. \text{add-mset } b M \in ?W)$ 
    and acc-hyp:  $\forall M. (M, M0) \in ?R \longrightarrow \text{add-mset } a M \in ?W$ 
    have  $\text{add-mset } a M0 \in ?W$ 
    proof (rule accI [of add-mset a M0])
      fix N
      assume  $(N, \text{add-mset } a M0) \in ?R$ 
      then consider M where  $(M, M0) \in ?R N = \text{add-mset } a M$ 
        | K where  $\forall b. b \in \# K \longrightarrow (b, a) \in r N = M0 + K$ 
        by atomize-elim (rule less-add)
      then show  $N \in ?W$ 
    proof cases
    case 1
      from acc-hyp have  $(M, M0) \in ?R \longrightarrow \text{add-mset } a M \in ?W$  ..
      from this and  $\langle (M, M0) \in ?R \rangle$  have  $\text{add-mset } a M \in ?W$  ..

```

```

    then show  $N \in ?W$  by (simp only:  $\langle N = \text{add-mset } a \ M \rangle$ )
  next
    case 2
    from this(1) have  $M0 + K \in ?W$ 
    proof (induct K)
      case empty
      from M0 show  $M0 + \{\#\} \in ?W$  by simp
    next
      case (add x K)
      from add.prem1 have  $(x, a) \in r$  by simp
      with wf-hyp have  $\forall M \in ?W. \text{add-mset } x \ M \in ?W$  by blast
      moreover from add have  $M0 + K \in ?W$  by simp
      ultimately have  $\text{add-mset } x \ (M0 + K) \in ?W$  ..
      then show  $M0 + (\text{add-mset } x \ K) \in ?W$  by simp
    qed
    then show  $N \in ?W$  by (simp only: 2(2))
  qed
} note tedious-reasoning = this

show  $M \in ?W$  for M
proof (induct M)
  show  $\{\#\} \in ?W$ 
  proof (rule accI)
    fix b assume  $(b, \{\#\}) \in ?R$ 
    with not-less-empty show  $b \in ?W$  by contradiction
  qed

  fix M a assume  $M \in ?W$ 
  from  $\langle \text{wf } r \rangle$  have  $\forall M \in ?W. \text{add-mset } a \ M \in ?W$ 
  proof induct
    fix a
    assume r:  $\bigwedge b. (b, a) \in r \implies (\forall M \in ?W. \text{add-mset } b \ M \in ?W)$ 
    show  $\forall M \in ?W. \text{add-mset } a \ M \in ?W$ 
    proof
      fix M assume  $M \in ?W$ 
      then show  $\text{add-mset } a \ M \in ?W$ 
      by (rule acc-induct) (rule tedious-reasoning [OF - r])
    qed
  qed
  from this and  $\langle M \in ?W \rangle$  show  $\text{add-mset } a \ M \in ?W$  ..
qed

lemma wf-mult1:  $\text{wf } r \implies \text{wf } (\text{mult1 } r)$ 
  by (rule acc-wfI) (rule all-accessible)

lemma wf-mult:  $\text{wf } r \implies \text{wf } (\text{mult } r)$ 
  unfolding mult-def by (rule wf-trancl) (rule wf-mult1)

```

lemma *wfp-multp*: $wfp\ r \implies wfp\ (multp\ r)$
unfolding *multp-def wfp-def*
by (*simp add: wf-mult*)

67.13.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:

assumes
trans: *trans* *r* **and**
MN: $(M, N) \in mult\ r$
shows $\exists I\ J\ K. N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in set-mset\ K. \exists j \in set-mset\ J. (k, j) \in r)$
using *MN* **unfolding** *mult-def mult1-def*
proof (*induction rule: converse-trancl-induct*)
case (*base y*)
then show *?case* **by force**
next
case (*step y z*) **note** *yz = this(1)* **and** *zN = this(2)* **and** *N-decomp = this(3)*
obtain *I J K* **where**
N: $N = I + J\ z = I + K\ J \neq \{\#\} \forall k \in \#K. \exists j \in \#J. (k, j) \in r$
using *N-decomp* **by blast**
obtain *a M0 K'* **where**
z: $z = add-mset\ a\ M0$ **and** *y*: $y = M0 + K'$ **and** *K*: $\forall b. b \in \#K' \longrightarrow (b, a) \in r$
using *yz* **by blast**
show *?case*
proof (*cases a ∈ # K*)
case *True*
moreover have $\exists j \in \#J. (k, j) \in r$ **if** $k \in \#K'$ **for** *k*
using *K N trans True* **by** (*meson that transE*)
ultimately show *?thesis*
by (*rule-tac x = I in exI, rule-tac x = J in exI, rule-tac x = (K - {#a#}) + K' in exI*)
(use z y N in <auto simp del: subset-mset.add-diff-assoc2 dest: in-diffD>)
next
case *False*
then have $a \in \#I$ **by** (*metis N(2) union-iff union-single-eq-member z*)
moreover have $M0 = I + K - \{\#a\}$
using *N(2) z* **by force**
ultimately show *?thesis*
by (*rule-tac x = I - {#a#} in exI, rule-tac x = add-mset a J in exI, rule-tac x = K + K' in exI*)
(use z y N False K in <auto simp: add.assoc>)
qed
qed

lemma *multp-implies-one-step*:

$transp\ R \Longrightarrow multp\ R\ M\ N \Longrightarrow \exists I\ J\ K. N = I + J \wedge M = I + K \wedge J \neq \{\#\}$
 $\wedge (\forall k \in \#K. \exists x \in \#J. R\ k\ x)$

by (*rule mult-implies-one-step[to-pred]*)

lemma *one-step-implies-mult*:

assumes

$J \neq \{\#\}$ **and**

$\forall k \in set-mset\ K. \exists j \in set-mset\ J. (k, j) \in r$

shows $(I + K, I + J) \in mult\ r$

using *assms*

proof (*induction size J arbitrary: I J K*)

case 0

then show *?case* **by** *auto*

next

case (*Suc n*) **note** $IH = this(1)$ **and** $size-J = this(2)[THEN\ sym]$

obtain $J'\ a$ **where** $J: J = add-mset\ a\ J'$

using $size-J$ **by** (*blast dest: size-eq-Suc-imp-eq-union*)

show *?case*

proof (*cases J' = {\#}*)

case *True*

then show *?thesis*

using $J\ Suc$ **by** (*fastforce simp add: mult-def mult1-def*)

next

case [*simp*]: *False*

have $K: K = \{\#x \in \#K. (x, a) \in r\# \} + \{\#x \in \#K. (x, a) \notin r\# \}$

by *simp*

have $(I + K, (I + \{\#x \in \#K. (x, a) \in r\# \}) + J') \in mult\ r$

using $IH[of\ J'\ \{\#x \in \#K. (x, a) \notin r\# \}\ I + \{\#x \in \#K. (x, a) \in r\# \}]$

$J\ Suc.prems\ K\ size-J$ **by** (*auto simp: ac-simps*)

moreover have $(I + \{\#x \in \#K. (x, a) \in r\# \} + J', I + J) \in mult\ r$

by (*fastforce simp: J mult1-def mult-def*)

ultimately show *?thesis*

unfolding *mult-def* **by** *simp*

qed

qed

lemma *one-step-implies-multp*:

$J \neq \{\#\} \Longrightarrow \forall k \in \#K. \exists j \in \#J. R\ k\ j \Longrightarrow multp\ R\ (I + K)\ (I + J)$

by (*rule one-step-implies-mult[of - - {(x, y). r x y} for r, folded multp-def, simplified]*)

lemma *subset-implies-mult*:

assumes $sub: A \subset \#B$

shows $(A, B) \in mult\ r$

proof –

have $ApBmA: A + (B - A) = B$

using *sub* **by** *simp*

have $BmA: B - A \neq \{\#\}$

using *sub* **by** (*simp add: Diff-eq-empty-iff-mset subset-mset.less-le-not-le*)

thus *?thesis*
by (rule one-step-implies-mult[of $B - A \{\#\} - A$, unfolded $ApBmA$, simplified])
qed

lemma *subset-implies-mult*: $A \subset \# B \implies \text{multp } r \ A \ B$
by (rule subset-implies-mult[of - - $\{(x, y). r \ x \ y\}$ for r , folded *multp-def*])

lemma *multp-repeat-mset-repeat-msetI*:
assumes *transp R* **and** *multp R A B* **and** $n \neq 0$
shows *multp R (repeat-mset n A) (repeat-mset n B)*
proof –
from $\langle \text{transp } R \rangle \langle \text{multp } R \ A \ B \rangle$ **obtain** $I \ J \ K$ **where**
 $B = I + J$ **and** $A = I + K$ **and** $J \neq \{\#\}$ **and** $\forall k \in \# K. \exists x \in \# J. R \ k \ x$
by (auto dest: *multp-implies-one-step*)

have *repeat-n-A-eq*: $\text{repeat-mset } n \ A = \text{repeat-mset } n \ I + \text{repeat-mset } n \ K$
using $\langle A = I + K \rangle$ **by** *simp*

have *repeat-n-B-eq*: $\text{repeat-mset } n \ B = \text{repeat-mset } n \ I + \text{repeat-mset } n \ J$
using $\langle B = I + J \rangle$ **by** *simp*

show *?thesis*
unfolding *repeat-n-A-eq repeat-n-B-eq*
proof (rule one-step-implies-multp)
from $\langle n \neq 0 \rangle$ **show** $\text{repeat-mset } n \ J \neq \{\#\}$
using $\langle J \neq \{\#\} \rangle$
by (simp add: *repeat-mset-eq-empty-iff*)
next
show $\forall k \in \# \text{repeat-mset } n \ K. \exists j \in \# \text{repeat-mset } n \ J. R \ k \ j$
using $\langle \forall k \in \# K. \exists x \in \# J. R \ k \ x \rangle$
by (metis count-greater-zero-iff nat-0-less-mult-iff *repeat-mset.rep-eq*)
qed
qed

67.13.3 Monotonicity

lemma *multp-mono-strong*:
assumes *multp R M1 M2* **and** *transp R* **and**
 $S\text{-if-}R: \bigwedge x \ y. x \in \text{set-mset } M1 \implies y \in \text{set-mset } M2 \implies R \ x \ y \implies S \ x \ y$
shows *multp S M1 M2*
proof –
obtain $I \ J \ K$ **where** $M2 = I + J$ **and** $M1 = I + K$ **and** $J \neq \{\#\}$ **and** $\forall k \in \# K. \exists x \in \# J. R \ k \ x$
using *multp-implies-one-step*[OF $\langle \text{transp } R \rangle \langle \text{multp } R \ M1 \ M2 \rangle$] **by** *auto*
show *?thesis*
unfolding $\langle M2 = I + J \rangle \langle M1 = I + K \rangle$
proof (rule one-step-implies-multp[OF $\langle J \neq \{\#\} \rangle$])
show $\forall k \in \# K. \exists j \in \# J. S \ k \ j$
using *S-if-R*

by (metis $\langle M1 = I + K \rangle \langle M2 = I + J \rangle \langle \forall k \in \#K. \exists x \in \#J. R\ k\ x \rangle$ union-iff)
 qed
 qed

lemma *mult-mono-strong*:

assumes $(M1, M2) \in \text{mult } r$ and $\text{trans } r$ and
 $S\text{-if-}R: \bigwedge x\ y. x \in \text{set-mset } M1 \implies y \in \text{set-mset } M2 \implies (x, y) \in r \implies (x, y) \in s$
 shows $(M1, M2) \in \text{mult } s$
 using *assms mult-mono-strong*[of $\lambda x\ y. (x, y) \in r\ M1\ M2\ \lambda x\ y. (x, y) \in s$,
unfolded mult-def trans-trans-eq, simplified]
 by *blast*

lemma *monotone-on-multip-multip-image-mset*:

assumes *monotone-on* A *orda ordb* f and *transp orda*
 shows *monotone-on* $\{M. \text{set-mset } M \subseteq A\}$ (*multip orda*) (*multip ordb*) (*image-mset* f)
proof (*rule monotone-onI*)
 fix $M1\ M2$
 assume
 $M1\text{-in}: M1 \in \{M. \text{set-mset } M \subseteq A\}$ and
 $M2\text{-in}: M2 \in \{M. \text{set-mset } M \subseteq A\}$ and
 $M1\text{-lt-}M2: \text{multip orda } M1\ M2$

from *multip-implies-one-step*[*OF* $\langle \text{transp orda} \rangle\ M1\text{-lt-}M2]$ **obtain** $I\ J\ K$ **where**

$M2\text{-eq}: M2 = I + J$ and
 $M1\text{-eq}: M1 = I + K$ and
 $J\text{-neq-mempty}: J \neq \{\#\}$ and
 $\text{ball-}K\text{-less}: \forall k \in \#K. \exists x \in \#J. \text{orda } k\ x$
 by *metis*

have *multip ordb* (*image-mset* $f\ I + \text{image-mset } f\ K$) (*image-mset* $f\ I + \text{image-mset } f\ J$)

proof (*intro one-step-implies-multip ballI*)

show *image-mset* $f\ J \neq \{\#\}$
 using $J\text{-neq-mempty}$ by *simp*

next

fix k' **assume** $k' \in \#\text{image-mset } f\ K$
then obtain k **where** $k' = f\ k$ and $k\text{-in}: k \in \#K$
 by *auto*
then obtain j **where** $j\text{-in}: j \in \#J$ and *orda* $k\ j$
 using $\text{ball-}K\text{-less}$ by *auto*

have *ordb* ($f\ k$) ($f\ j$)

proof (*rule* $\langle \text{monotone-on } A\ \text{orda ordb } f \rangle$ [*THEN monotone-onD, OF* - - $\langle \text{orda } k\ j \rangle$])

show $k \in A$
 using $M1\text{-eq } M1\text{-in } k\text{-in}$ by *auto*
next

```

    show  $j \in A$ 
    using  $M2\text{-eq } M2\text{-in } j\text{-in}$  by auto
  qed
  thus  $\exists j \in \# \text{image-mset } f J. \text{ordb } k' j$ 
    using  $\langle j \in \# J \rangle \langle k' = f k \rangle$  by auto
  qed
  thus  $\text{multp ordb } (\text{image-mset } f M1) (\text{image-mset } f M2)$ 
    by (simp add:  $M1\text{-eq } M2\text{-eq}$ )
  qed

```

lemma *monotone-multp-multp-image-mset:*
 assumes *monotone orda ordb f and transp orda*
 shows *monotone (multp orda) (multp ordb) (image-mset f)*
 by (rule *monotone-on-multp-multp-image-mset[OF assms, simplified]*)

lemma *multp-image-mset-image-msetI:*
 assumes *multp $(\lambda x y. R (f x) (f y)) M1 M2$ and transp R*
 shows *multp R (image-mset f M1) (image-mset f M2)*
proof –
 from $\langle \text{transp } R \rangle$ have *transp $(\lambda x y. R (f x) (f y))$*
 by (auto intro: *transpI dest: transpD*)
 with $\langle \text{multp } (\lambda x y. R (f x) (f y)) M1 M2 \rangle$ obtain *I J K where*
 $M2 = I + J$ and $M1 = I + K$ and $J \neq \{\#\}$ and $\forall k \in \# K. \exists x \in \# J. R (f k)$
 $(f x)$
 using *multp-implies-one-step* by blast

have *multp R (image-mset f I + image-mset f K) (image-mset f I + image-mset f J)*
proof (rule *one-step-implies-multp*)
 show *image-mset f J $\neq \{\#\}$*
 by (simp add: $\langle J \neq \{\#\} \rangle$)
 next
 show $\forall k \in \# \text{image-mset } f K. \exists j \in \# \text{image-mset } f J. R k j$
 by (simp add: $\langle \forall k \in \# K. \exists x \in \# J. R (f k) (f x) \rangle$)
 qed
 thus ?thesis
 by (simp add: $\langle M1 = I + K \rangle \langle M2 = I + J \rangle$)
 qed

lemma *multp-image-mset-image-msetD:*
 assumes
 $\text{multp } R (\text{image-mset } f A) (\text{image-mset } f B)$ and
 $\text{transp } R$ and
 $\text{inj-on-}f: \text{inj-on } f (\text{set-mset } A \cup \text{set-mset } B)$
 shows *multp $(\lambda x y. R (f x) (f y)) A B$*
proof –
 from *assms(1,2)* obtain *I J K where*
 $f\text{-}B\text{-eq: image-mset } f B = I + J$ and
 $f\text{-}A\text{-eq: image-mset } f A = I + K$ and

J-neq-mempty: $J \neq \{\#\}$ **and**
ball-K-less: $\forall k \in \#K. \exists x \in \#J. R \ k \ x$
by (*auto dest: multp-implies-one-step*)

from *f-B-eq* **obtain** $I' \ J'$ **where**
B-def: $B = I' + J'$ **and** *I-def*: $I = \text{image-mset } f \ I'$ **and** *J-def*: $J = \text{image-mset } f \ J'$
using *image-mset-eq-plusD* **by** *blast*

from *inj-on-f* **have** *inj-on-f'*: *inj-on* $f \ (\text{set-mset } A \cup \text{set-mset } I')$
by (*rule inj-on-subset*) (*auto simp add: B-def*)

from *f-A-eq* **obtain** K' **where**
A-def: $A = I' + K'$ **and** *K-def*: $K = \text{image-mset } f \ K'$
by (*auto simp: I-def dest: image-mset-eq-image-mset-plusD[OF - inj-on-f']*)

show *?thesis*
unfolding *A-def B-def*
proof (*intro one-step-implies-mult ballI*)
from *J-neq-mempty* **show** $J' \neq \{\#\}$
by (*simp add: J-def*)

next
fix k **assume** $k \in \#K'$
with *ball-K-less* **obtain** j' **where** $j' \in \#J$ **and** $R \ (f \ k) \ j'$
using *K-def* **by** *auto*
moreover then obtain j **where** $j \in \#J'$ **and** $f \ j = j'$
using *J-def* **by** *auto*
ultimately show $\exists j \in \#J'. R \ (f \ k) \ (f \ j)$
by *blast*

qed
qed

67.13.4 The multiset extension is cancellative for multiset union

lemma *mult-cancel*:
assumes *trans s* **and** *irrefl-on (set-mset Z) s*
shows $(X + Z, Y + Z) \in \text{mult } s \longleftrightarrow (X, Y) \in \text{mult } s$ (**is** $?L \longleftrightarrow ?R$)
proof
assume $?L$ **thus** $?R$
using $\langle \text{irrefl-on } (\text{set-mset } Z) \ s \rangle$
proof (*induct Z*)
case (*add z Z*)
obtain $X' \ Y' \ Z'$ **where** $*$: $\text{add-mset } z \ X + Z = Z' + X' \ \text{add-mset } z \ Y + Z = Z' + Y' \ Y' \neq \{\#\}$
 $\forall x \in \text{set-mset } X'. \exists y \in \text{set-mset } Y'. (x, y) \in s$
using *mult-implies-one-step[OF trans s add(2)]* **by** *auto*
consider $Z2$ **where** $Z' = \text{add-mset } z \ Z2 \mid X2 \ Y2$ **where** $X' = \text{add-mset } z \ X2$
 $Y' = \text{add-mset } z \ Y2$
using $*(1,2)$ **by** (*metis add-mset-remove-trivial-If insert-iff set-mset-add-mset-insert*)

```

union-iff)
  thus ?case
  proof (cases)
    case 1 thus ?thesis
      using * one-step-implies-mult[of Y' X' s Z2] add(3)
      by (auto simp: add commute[of - {#-#}]) add.assoc intro: add(1) elim:
irrefl-on-subset)
  next
    case 2 then obtain y where y ∈ set-mset Y2 (z, y) ∈ s
      using *(4) ⟨irrefl-on (set-mset (add-mset z Z)) s⟩
      by (auto simp: irrefl-on-def)
    moreover from this transD[OF ⟨trans s⟩ - this(2)]
    have x' ∈ set-mset X2 ⟹ ∃ y ∈ set-mset Y2. (x', y) ∈ s for x'
      using 2 *(4)[rule-format, of x'] by auto
    ultimately show ?thesis
      using * one-step-implies-mult[of Y2 X2 s Z'] 2 add(3)
      by (force simp: add commute[of {#-#}] add.assoc[symmetric] intro: add(1)
elim: irrefl-on-subset)
  qed
qed auto
next
  assume ?R then obtain I J K
    where Y = I + J X = I + K J ≠ {#} ∀ k ∈ set-mset K. ∃ j ∈ set-mset J.
(k, j) ∈ s
    using mult-implies-one-step[OF ⟨trans s⟩] by blast
  thus ?L using one-step-implies-mult[of J K s I + Z] by (auto simp: ac-simps)
qed

lemma multp-cancel:
  transp R ⟹ irreflp-on (set-mset Z) R ⟹ multp R (X + Z) (Y + Z) ⟷
multp R X Y
  by (rule mult-cancel[to-pred])

lemma mult-cancel-add-mset:
  trans r ⟹ irreflp-on {z} r ⟹
((add-mset z X, add-mset z Y) ∈ mult r) = ((X, Y) ∈ mult r)
  by (rule mult-cancel[of - {#-#}, simplified])

lemma multp-cancel-add-mset:
  transp R ⟹ irreflp-on {z} R ⟹ multp R (add-mset z X) (add-mset z Y) =
multp R X Y
  by (rule mult-cancel-add-mset[to-pred, folded bot-set-def])

lemma mult-cancel-max0:
  assumes trans s and irreflp-on (set-mset X ∩ set-mset Y) s
  shows (X, Y) ∈ mult s ⟷ (X - X ∩# Y, Y - X ∩# Y) ∈ mult s (is ?L
⟷ ?R)
proof -
  have (X - X ∩# Y + X ∩# Y, Y - X ∩# Y + X ∩# Y) ∈ mult s ⟷ (X

```

$- X \cap\# Y, Y - X \cap\# Y) \in \text{mult } s$
proof (rule *mult-cancel*)
 from *assms* **show** *trans s*
 by *simp*
next
 from *assms* **show** *irrefl-on (set-mset (X $\cap\#$ Y)) s*
 by *simp*
qed
moreover **have** $X - X \cap\# Y + X \cap\# Y = X Y - X \cap\# Y + X \cap\# Y = Y$
 by (auto *simp flip: count-inject*)
ultimately show *?thesis*
 by *simp*
qed

lemma *mult-cancel-max*:

$\text{trans } r \implies \text{irrefl-on (set-mset } X \cap \text{set-mset } Y) r \implies$
 $(X, Y) \in \text{mult } r \longleftrightarrow (X - Y, Y - X) \in \text{mult } r$
by (rule *mult-cancel-max0[simplified]*)

lemma *multp-cancel-max*:

$\text{transp } R \implies \text{irreflp-on (set-mset } X \cap \text{set-mset } Y) R \implies \text{multp } R X Y \longleftrightarrow$
 $\text{multp } R (X - Y) (Y - X)$
by (rule *mult-cancel-max[to-pred]*)

67.13.5 Strict partial-order properties

lemma *mult1-lessE*:

assumes $(N, M) \in \text{mult1 } \{(a, b). r \ a \ b\}$ **and** *asympt r*
obtains $a \ M0 \ K$ **where** $M = \text{add-mset } a \ M0 \ N = M0 + K$
 $a \notin\# K \ \wedge \ b \in\# K \implies r \ b \ a$
proof –
 from *assms* **obtain** $a \ M0 \ K$ **where** $M = \text{add-mset } a \ M0 \ N = M0 + K$ **and**
 $\ast: b \in\# K \implies r \ b \ a$ **for** b **by** (blast *elim: mult1E*)
moreover **from** \ast [of a] **have** $a \notin\# K$
 using $\langle \text{asympt } r \rangle$ **by** (meson *asymptD*)
ultimately show *thesis* **by** (auto *intro: that*)
qed

lemma *trans-on-mult*:

assumes *trans-on A r* **and** $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$
shows *trans-on B (mult r)*
using *assms* **by** (metis *mult-def subset-UNIV trans-on-subset trans-trancl*)

lemma *trans-mult*: $\text{trans } r \implies \text{trans (mult } r)$

using *trans-on-mult[of UNIV r UNIV, simplified]* .

lemma *transp-on-multp*:

assumes *transp-on A r* **and** $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$
shows *transp-on B (multp r)*

by (metis mult-def multp-def transD trans-transcl trans-onI)

lemma *transp-multp*: $\text{transp } r \implies \text{transp } (\text{multp } r)$
 using *transp-on-multp*[of UNIV r UNIV, simplified] .

lemma *irrefl-mult*:

assumes *trans* r *irrefl* r

shows *irrefl* (*mult* r)

proof (intro *irreflI* *notI*)

fix M

assume $(M, M) \in \text{mult } r$

then obtain $I J K$ where $M = I + J$ and $M = I + K$

and $J \neq \{\#\}$ and $(\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J. (k, j) \in r)$

using *mult-implies-one-step*[OF $\langle \text{trans } r \rangle$] by *blast*

then have $*$: $K \neq \{\#\}$ and $**$: $\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } K. (k, j) \in r$ by

auto

have *finite* (*set-mset* K) by *simp*

hence *set-mset* $K = \{\}$

using $**$

proof (*induction* rule: *finite-induct*)

case *empty*

thus ?case by *simp*

next

case (*insert* x F)

have *False*

using $\langle \text{irrefl } r \rangle$ [*unfolded irrefl-def*, *rule-format*]

using $\langle \text{trans } r \rangle$ [*THEN transD*]

by (metis *equals0D* *insert.IH* *insert.premis* *insertE* *insertI1* *insertI2*)

thus ?case ..

qed

with $*$ show *False* by *simp*

qed

lemma *irreflp-multp*: $\text{transp } R \implies \text{irreflp } R \implies \text{irreflp } (\text{multp } R)$

by (rule *irrefl-mult*[of $\{(x, y). r \ x \ y\}$ for r ,

folded transp-trans-eq irreflp-irrefl-eq, simplified, folded multp-def])

instantiation *multiset* :: (*preorder*) *order* **begin**

definition *less-multiset* :: ' a *multiset* \Rightarrow ' a *multiset* \Rightarrow *bool*

where $M < N \longleftrightarrow \text{multp } (<) \ M \ N$

definition *less-eq-multiset* :: ' a *multiset* \Rightarrow ' a *multiset* \Rightarrow *bool*

where *less-eq-multiset* $M \ N \longleftrightarrow M < N \vee M = N$

instance

proof *intro-classes*

fix $M \ N :: 'a$ *multiset*

show $(M < N) = (M \leq N \wedge \neg N \leq M)$

```

    unfolding less-eq-multiset-def less-multiset-def
  by (metis irreftp-def irreftp-on-less irreftp-multip transpE transp-on-less transp-multip)
next
  fix M :: 'a multiset
  show M ≤ M
    unfolding less-eq-multiset-def
    by simp
next
  fix M1 M2 M3 :: 'a multiset
  show M1 ≤ M2 ⟹ M2 ≤ M3 ⟹ M1 ≤ M3
    unfolding less-eq-multiset-def less-multiset-def
    using transp-multip[OF transp-on-less, THEN transpD]
    by blast
next
  fix M N :: 'a multiset
  show M ≤ N ⟹ N ≤ M ⟹ M = N
    unfolding less-eq-multiset-def less-multiset-def
    using transp-multip[OF transp-on-less, THEN transpD]
    using irreftp-multip[OF transp-on-less irreftp-on-less, unfolded irreftp-def, rule-format]
    by blast
qed

end

```

```

lemma mset-le-irrefl [elim!]:
  fixes M :: 'a::preorder multiset
  shows M < M ⟹ R
  by simp

```

```

lemma wfp-less-multiset[simp]:
  assumes wf: wfp ((<) :: ('a :: preorder) ⇒ 'a ⇒ bool)
  shows wfp ((<) :: 'a multiset ⇒ 'a multiset ⇒ bool)
  unfolding less-multiset-def
  using wfp-multip[OF wf] .

```

67.13.6 Strict total-order properties

```

lemma total-on-mult:
  assumes total-on A r and trans r and  $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$ 
  shows total-on B (mult r)
proof (rule total-onI)
  fix M1 M2 assume M1 ∈ B and M2 ∈ B and M1 ≠ M2
  let ?I = M1 ∩# M2
  show (M1, M2) ∈ mult r ∨ (M2, M1) ∈ mult r
proof (cases M1 - ?I = {#} ∨ M2 - ?I = {#})
  case True
  with ⟨M1 ≠ M2⟩ show ?thesis
  by (metis Diff-eq-empty-iff-mset diff-intersect-left-idem diff-intersect-right-idem
    subset-implies-mult subset-mset.less-le)

```



```

next
  case False
  from assms(1) have total-on (set-mset (M1 - ?I)) r
  by (meson ⟨M1 ∈ B⟩ assms(3) diff-subset-eq-self set-mset-mono total-on-subset)
  with False obtain greatest1 where
    greatest1-in: greatest1 ∈# M1 - ?I and
    greatest1-greatest: ∀ x ∈# M1 - ?I. greatest1 ≠ x ⟶ (x, greatest1) ∈ r
    using Multiset.bex-greatest-element[to-set, of M1 - ?I r]
    by (metis assms(2) subset-UNIV trans-on-subset)

  from assms(1) have total-on (set-mset (M2 - ?I)) r
  by (meson ⟨M2 ∈ B⟩ assms(3) diff-subset-eq-self set-mset-mono total-on-subset)
  with False obtain greatest2 where
    greatest2-in: greatest2 ∈# M2 - ?I and
    greatest2-greatest: ∀ x ∈# M2 - ?I. greatest2 ≠ x ⟶ (x, greatest2) ∈ r
    using Multiset.bex-greatest-element[to-set, of M2 - ?I r]
    by (metis assms(2) subset-UNIV trans-on-subset)

  have greatest1 ≠ greatest2
  using greatest1-in ⟨greatest2 ∈# M2 - ?I⟩
  by (metis diff-intersect-left-idem diff-intersect-right-idem dual-order.eq-iff
in-diff-count
  in-diff-countE le-add-same-cancel2 less-irrefl zero-le)
  hence (greatest1, greatest2) ∈ r ∨ (greatest2, greatest1) ∈ r
  using ⟨total-on A r⟩[unfolded total-on-def, rule-format, of greatest1 greatest2]
  ⟨M1 ∈ B⟩ ⟨M2 ∈ B⟩ greatest1-in greatest2-in assms(3)
  by (meson in-diffD in-mono)
  thus ?thesis
  proof (elim disjE)
    assume (greatest1, greatest2) ∈ r
    have (?I + (M1 - ?I), ?I + (M2 - ?I)) ∈ mult r
    proof (rule one-step-implies-mult[of M2 - ?I M1 - ?I r ?I])
      show M2 - ?I ≠ {#}
      using False by force
    next
      show ∀ k ∈# M1 - ?I. ∃ j ∈# M2 - ?I. (k, j) ∈ r
      using ⟨(greatest1, greatest2) ∈ r⟩ greatest2-in greatest1-greatest
      by (metis assms(2) transD)
    qed
    hence (M1, M2) ∈ mult r
    by (metis subset-mset.add-diff-inverse subset-mset.inf.cobounded1
subset-mset.inf.cobounded2)
    thus (M1, M2) ∈ mult r ∨ (M2, M1) ∈ mult r ..
  next
    assume (greatest2, greatest1) ∈ r
    have (?I + (M2 - ?I), ?I + (M1 - ?I)) ∈ mult r
    proof (rule one-step-implies-mult[of M1 - ?I M2 - ?I r ?I])
      show M1 - M1 ∩# M2 ≠ {#}
      using False by force
    end
  end

```

```

next
  show  $\forall k \in \#M2 - ?I. \exists j \in \#M1 - ?I. (k, j) \in r$ 
    using  $\langle \text{greatest2}, \text{greatest1} \rangle \in r \rangle \text{greatest1-in greatest2-greatest}$ 
    by  $(\text{metis assms}(2) \text{transD})$ 
qed
hence  $(M2, M1) \in \text{mult } r$ 
  by  $(\text{metis subset-mset.add-diff-inverse subset-mset.inf.cobounded1}$ 
     $\text{subset-mset.inf.cobounded2})$ 
thus  $(M1, M2) \in \text{mult } r \vee (M2, M1) \in \text{mult } r ..$ 
qed
qed
qed

```

lemma *total-mult*: $\text{total } r \implies \text{trans } r \implies \text{total } (\text{mult } r)$
 by $(\text{rule total-on-mult[of UNIV } r \text{ UNIV, simplified]})$

lemma *totalp-on-multip*:
 $\text{totalp-on } A \ R \implies \text{transp } R \implies (\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A) \implies \text{totalp-on } B \ (\text{multp } R)$
 using *total-on-mult*[of $A \ \{(x,y). R \ x \ y\} \ B, \text{to-pred}$]
 by $(\text{simp add: multp-def total-on-def totalp-on-def})$

lemma *totalp-multip*: $\text{totalp } R \implies \text{transp } R \implies \text{totalp } (\text{multp } R)$
 by $(\text{rule totalp-on-multip[of UNIV } R \text{ UNIV, simplified]})$

67.14 Quasi-executable version of the multiset extension

Predicate variants of *mult* and the reflexive closure of *mult*, which are executable whenever the given predicate *P* is. Together with the standard code equations for $(\cap\#)$ and $(-)$ this should yield quadratic (with respect to calls to *P*) implementations of *multp-code* and *multeqp-code*.

definition *multp-code* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$
where

multp-code $P \ N \ M =$
 $(\text{let } Z = M \cap\# \ N; X = M - Z \text{ in}$
 $X \neq \{\#\} \wedge (\text{let } Y = N - Z \text{ in } (\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P \ y \ x)))$

definition *multeqp-code* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset} \Rightarrow \text{bool}$
where

multeqp-code $P \ N \ M =$
 $(\text{let } Z = M \cap\# \ N; X = M - Z; Y = N - Z \text{ in}$
 $(\forall y \in \text{set-mset } Y. \exists x \in \text{set-mset } X. P \ y \ x))$

lemma *multp-code-iff-mult*:

assumes *irrefl-on* $(\text{set-mset } N \cap \text{set-mset } M) \ R$ **and** *trans* R **and**
 $[\text{simp}]: \bigwedge x \ y. P \ x \ y \longleftrightarrow (x, y) \in R$
shows $\text{multp-code } P \ N \ M \longleftrightarrow (N, M) \in \text{mult } R \text{ (is } ?L \longleftrightarrow ?R)$

proof –

have *: $M \cap\# \ N + (N - M \cap\# \ N) = N \ M \cap\# \ N + (M - M \cap\# \ N) = M$

```

  (M - M ∩# N) ∩# (N - M ∩# N) = {#} by (auto simp flip: count-inject)
show ?thesis
proof
  assume ?L thus ?R
    using one-step-implies-mult[of M - M ∩# N N - M ∩# N R M ∩# N] *
    by (auto simp: multp-code-def Let-def)
next
  have [dest!]: I = {#} if (I + J) ∩# (I + K) = {#} for I J K
    using that by (metis inter-union-distrib-right union-eq-empty)
  assume ?R thus ?L
    using mult-cancel-max
    using mult-implies-one-step[OF assms(2), of N - M ∩# N M - M ∩# N]
    mult-cancel-max[OF assms(2,1)] * by (auto simp: multp-code-def)
qed
qed

```

lemma multp-code-iff-multp:

```

  irreflp-on (set-mset M ∩ set-mset N) R  $\implies$  transp R  $\implies$  multp-code R M N
 $\longleftrightarrow$  multp R M N
  using multp-code-iff-mult[simplified, to-pred, of M N R R] by simp

```

lemma multp-code-eq-multp:

```

  assumes irreflp R and transp R
  shows multp-code R = multp R
proof (intro ext)
  fix M N
  show multp-code R M N = multp R M N
  proof (rule multp-code-iff-multp)
    from assms show irreflp-on (set-mset M ∩ set-mset N) R
    by (auto intro: irreflp-on-subset)
  next
    from assms show transp R
    by simp
  qed
qed

```

lemma multeqp-code-iff-reflcl-mult:

```

  assumes irreflp-on (set-mset N ∩ set-mset M) R and trans R and  $\bigwedge x y. P x y$ 
 $\longleftrightarrow (x, y) \in R$ 
  shows multeqp-code P N M  $\longleftrightarrow (N, M) \in (\text{mult } R)^=$ 
proof -
  have  $\exists y. \text{count } M y < \text{count } N y$  if  $N \neq M$   $M - M \cap\# N = \{ \# \}$ 
  proof -
    from that obtain y where  $\text{count } N y \neq \text{count } M y$ 
    by (auto simp flip: count-inject)
  then show ?thesis
    using  $\langle M - M \cap\# N = \{ \# \} \rangle$ 
    by (auto simp flip: count-inject dest!: le-neq-implies-less fun-cong[of - y])
  qed
qed

```

```

then have multeqp-code  $P\ N\ M \longleftrightarrow \text{multp-code } P\ N\ M \vee N = M$ 
  by (auto simp: multeqp-code-def multp-code-def Let-def in-diff-count)
thus ?thesis
  using multp-code-iff-mult[OF assms] by simp
qed

```

```

lemma multeqp-code-iff-reflclp-multp:
   $\text{irreflp-on } (\text{set-mset } M \cap \text{set-mset } N) \ R \implies \text{transp } R \implies \text{multeqp-code } R\ M\ N$ 
 $\longleftrightarrow (\text{multp } R)^{==} M\ N$ 
  using multeqp-code-iff-reflcl-mult[simplified, to-pred, of  $M\ N\ R\ R$ ] by simp

```

```

lemma multeqp-code-eq-reflclp-multp:
  assumes  $\text{irreflp } R$  and  $\text{transp } R$ 
  shows  $\text{multeqp-code } R = (\text{multp } R)^{==}$ 
proof (intro ext)
  fix  $M\ N$ 
  show  $\text{multeqp-code } R\ M\ N \longleftrightarrow (\text{multp } R)^{==} M\ N$ 
  proof (rule multeqp-code-iff-reflclp-multp)
    from assms show  $\text{irreflp-on } (\text{set-mset } M \cap \text{set-mset } N) \ R$ 
    by (auto intro: irreflp-on-subset)
  next
    from assms show  $\text{transp } R$ 
    by simp
  qed
qed

```

67.14.1 Monotonicity of multiset union

```

lemma mult1-union:  $(B, D) \in \text{mult1 } r \implies (C + B, C + D) \in \text{mult1 } r$ 
  by (force simp: mult1-def)

```

```

lemma union-le-mono2:  $B < D \implies C + B < C + (D :: 'a :: \text{preorder multiset})$ 
  unfolding less-multiset-def multp-def mult-def
  by (induction rule: trancl-induct; blast intro: mult1-union trancl-trans)

```

```

lemma union-le-mono1:  $B < D \implies B + C < D + (C :: 'a :: \text{preorder multiset})$ 
  by (metis add.commute union-le-mono2)

```

```

lemma union-less-mono:
  fixes  $A\ B\ C\ D :: 'a :: \text{preorder multiset}$ 
  shows  $A < C \implies B < D \implies A + B < C + D$ 
  by (blast intro!: union-le-mono1 union-le-mono2 less-trans)

```

```

instantiation multiset :: (preorder) ordered-ab-semigroup-add
begin
instance
  by standard (auto simp add: less-eq-multiset-def intro: union-le-mono2)
end

```

67.14.2 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in \# \ XS \implies x \in \# \ \{\# \ y \ \# \} + XS$
and *multi-member-this*: $x \in \# \ \{\# \ x \ \# \} + XS$
and *multi-member-last*: $x \in \# \ \{\# \ x \ \# \}$
by *auto*

definition *ms-strict* = *mult pair-less*

definition *ms-weak* = *ms-strict* \cup *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
unfolding *reduction-pair-def* *ms-strict-def* *ms-weak-def* *pair-less-def*
by (*auto* *intro*: *wf-mult1* *wf-trancl* *simp*: *mult-def*)

lemma *smsI*:

(*set-mset* *A*, *set-mset* *B*) \in *max-strict* \implies (*Z* + *A*, *Z* + *B*) \in *ms-strict*

unfolding *ms-strict-def*

by (*rule one-step-implies-mult*) (*auto* *simp* *add*: *max-strict-def* *pair-less-def* *elim!*: *max-ext.cases*)

lemma *wmsI*:

(*set-mset* *A*, *set-mset* *B*) \in *max-strict* \vee *A* = $\{\#\}$ \wedge *B* = $\{\#\}$

\implies (*Z* + *A*, *Z* + *B*) \in *ms-weak*

unfolding *ms-weak-def* *ms-strict-def*

by (*auto* *simp* *add*: *pair-less-def* *max-strict-def* *elim!*: *max-ext.cases* *intro*: *one-step-implies-mult*)

inductive *pw-leq*

where

pw-leq-empty: *pw-leq* $\{\#\}$ $\{\#\}$

| *pw-leq-step*: $\llbracket (x, y) \in \text{pair-leq}; \text{pw-leq } X \ Y \rrbracket \implies \text{pw-leq } (\{\#x\# \} + X) (\{\#y\# \} + Y)$

lemma *pw-leq-lstep*:

(*x*, *y*) \in *pair-leq* \implies *pw-leq* $\{\#x\# \}$ $\{\#y\# \}$

by (*drule* *pw-leq-step*) (*rule* *pw-leq-empty*, *simp*)

lemma *pw-leq-split*:

assumes *pw-leq* *X* *Y*

shows $\exists A \ B \ Z. X = A + Z \wedge Y = B + Z \wedge ((\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$

using *assms*

proof *induct*

case *pw-leq-empty* **thus** ?*case* **by** *auto*

next

case (*pw-leq-step* *x* *y* *X* *Y*)

then obtain *A* *B* *Z* **where**

[*simp*]: *X* = *A* + *Z* *Y* = *B* + *Z*

and 1[*simp*]: (*set-mset* *A*, *set-mset* *B*) \in *max-strict* \vee (*B* = $\{\#\}$ \wedge *A* = $\{\#\}$)

by *auto*

from *pw-leq-step* **consider** *x* = *y* | (*x*, *y*) \in *pair-less*

unfolding *pair-leq-def* **by** *auto*

```

thus ?case
proof cases
  case [simp]: 1
  have  $\{\#x\# \} + X = A + (\{\#y\# \} + Z) \wedge \{\#y\# \} + Y = B + (\{\#y\# \} + Z) \wedge$ 
     $((\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$ 
    by auto
  thus ?thesis by blast
next
  case 2
  let ?A' =  $\{\#x\# \} + A$  and ?B' =  $\{\#y\# \} + B$ 
  have  $\{\#x\# \} + X = ?A' + Z$ 
     $\{\#y\# \} + Y = ?B' + Z$ 
    by auto
  moreover have
     $(\text{set-mset } ?A', \text{set-mset } ?B') \in \text{max-strict}$ 
    using 1 2 unfolding max-strict-def
    by (auto elim!: max-ext.cases)
  ultimately show ?thesis by blast
qed
qed

lemma
  assumes pwleq: pw-leq Z Z'
  shows ms-strictI:  $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B)$ 
     $\in \text{ms-strict}$ 
    and ms-weakI1:  $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict} \implies (Z + A, Z' + B)$ 
     $\in \text{ms-weak}$ 
    and ms-weakI2:  $(Z + \{\#\}, Z' + \{\#\}) \in \text{ms-weak}$ 
proof –
  from pwleq-split[OF pwleq]
  obtain A' B' Z''
    where [simp]:  $Z = A' + Z''$   $Z' = B' + Z''$ 
    and mx-or-empty:  $(\text{set-mset } A', \text{set-mset } B') \in \text{max-strict} \vee (A' = \{\#\} \wedge B' = \{\#\})$ 
    by blast
  {
    assume max:  $(\text{set-mset } A, \text{set-mset } B) \in \text{max-strict}$ 
    from mx-or-empty
    have  $(Z'' + (A + A'), Z'' + (B + B')) \in \text{ms-strict}$ 
    proof
      assume max':  $(\text{set-mset } A', \text{set-mset } B') \in \text{max-strict}$ 
      with max have  $(\text{set-mset } (A + A'), \text{set-mset } (B + B')) \in \text{max-strict}$ 
        by (auto simp: max-strict-def intro: max-ext-additive)
      thus ?thesis by (rule smsI)
    next
      assume [simp]:  $A' = \{\#\} \wedge B' = \{\#\}$ 
      show ?thesis by (rule smsI) (auto intro: max)
    qed
    thus  $(Z + A, Z' + B) \in \text{ms-strict}$  by (simp add: ac-simps)
  }

```

```

    thus (Z + A, Z' + B) ∈ ms-weak by (simp add: ms-weak-def)
  }
  from mx-or-empty
  have (Z'' + A', Z'' + B') ∈ ms-weak by (rule wmsI)
  thus (Z + {#}, Z' + {#}) ∈ ms-weak by (simp add: ac-simps)
qed

```

```

lemma empty-neutral: {#} + x = x x + {#} = x
and nonempty-plus: {# x #} + rs ≠ {#}
and nonempty-single: {# x #} ≠ {#}
by auto

```

```

setup <
  let
    fun msetT T = Type <multiset T>;

    fun mk-mset T [] = instantiate <'a = T in term <{#}>>
      | mk-mset T [x] = instantiate <'a = T and x in term <{#x#}>>
      | mk-mset T (x :: xs) = Const <plus <msetT T> for <mk-mset T [x]> <mk-mset
T xs>>

    fun mset-member-tac ctxt m i =
      if m <= 0 then
        resolve-tac ctxt @ {thms multi-member-this} i ORELSE
        resolve-tac ctxt @ {thms multi-member-last} i
      else
        resolve-tac ctxt @ {thms multi-member-skip} i THEN mset-member-tac ctxt
(m - 1) i

    fun mset-nonempty-tac ctxt =
      resolve-tac ctxt @ {thms nonempty-plus} ORELSE'
      resolve-tac ctxt @ {thms nonempty-single}

    fun regroup-munion-conv ctxt =
      Function-Lib.regroup-conv ctxt const-abbrev <empty-mset> const-name <plus>
      (map (fn t => t RS eq-reflection) (@ {thms ac-simps} @ @ {thms empty-neutral}))

    fun unfold-pwleq-tac ctxt i =
      (resolve-tac ctxt @ {thms pw-leq-step} i THEN (fn st => unfold-pwleq-tac ctxt
(i + 1) st))
      ORELSE (resolve-tac ctxt @ {thms pw-leq-lstep} i)
      ORELSE (resolve-tac ctxt @ {thms pw-leq-empty} i)

    val set-mset-simps = [@ {thm set-mset-empty}, @ {thm set-mset-single}, @ {thm
set-mset-union},
      @ {thm Un-insert-left}, @ {thm Un-empty-left}]

  in
    ScnpReconstruct.multiset-setup (ScnpReconstruct.Multiset
{

```

```

msetT=msetT, mk-mset=mk-mset, mset-regroup-conv=regroup-munion-conv,
mset-member-tac=mset-member-tac, mset-nonempty-tac=mset-nonempty-tac,
mset-pwleq-tac=unfold-pwleq-tac, set-of-simps=set-mset-simps,
smsI'= @{thm ms-strictI}, wmsI2''= @{thm ms-weakI2}, wmsI1= @{thm
ms-weakI1},
reduction-pair = @{thm ms-reduction-pair}
})
end
>

```

67.15 Legacy theorem bindings

lemmas *multi-count-eq = multiset-eq-iff [symmetric]*

lemma *union-commute*: $M + N = N + (M::'a \text{ multiset})$
by (*fact add.commute*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a \text{ multiset}))$
by (*fact add.assoc*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a \text{ multiset}))$
by (*fact add.left-commute*)

lemmas *union-ac = union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a \text{ multiset})$
by (*fact add-right-cancel*)

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a \text{ multiset})$
by (*fact add-left-cancel*)

lemma *multi-union-self-other-eq*: $(A::'a \text{ multiset}) + X = A + Y \Longrightarrow X = Y$
by (*fact add-left-imp-eq*)

lemma *mset-subset-trans*: $(M::'a \text{ multiset}) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
by (*fact subset-mset.less-trans*)

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
by (*fact subset-mset.inf.commute*)

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
by (*fact subset-mset.inf.assoc [symmetric]*)

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
by (*fact subset-mset.inf.left-commute*)

lemmas *multiset-inter-ac =*
multiset-inter-commute
multiset-inter-assoc

multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg M < (M::'a::preorder\ multiset)$
by (*fact less-irrefl*)

lemma *mset-le-trans*: $K < M \implies M < N \implies K < (N::'a::preorder\ multiset)$
by (*fact less-trans*)

lemma *mset-le-not-sym*: $M < N \implies \neg N < (M::'a::preorder\ multiset)$
by (*fact less-not-sym*)

lemma *mset-le-asy*: $M < N \implies (\neg P \implies N < (M::'a::preorder\ multiset)) \implies P$
by (*fact less-asy*)

declaration \langle

let
 fun *multiset-postproc* - *maybe-name* *all-values* (*T* *as* *Type* ($-$, [*elem-T*])) (*Const* - $\$ t'$) =
 let
 val (*maybe-opt*, *ps*) =
 Nitpick-Model.dest-plain-fun *t'*
 $||> (\sim\sim)$
 $||> \text{map } (\text{apsnd } (\text{snd } o \text{ HOLogic.dest-number}))$
 fun *elems-for* *t* =
 (*case* *AList.lookup* (=) *ps* *t* *of*
 SOME *n* \Rightarrow *replicate* *n* *t*
 | *NONE* \Rightarrow [*Const* (*maybe-name*, *elem-T* \dashrightarrow *elem-T*) $\$ t$])
 in
 (*case* *maps* *elems-for* (*all-values* *elem-T*) @
 (*if* *maybe-opt* *then* [*Const* (*Nitpick-Model.unrep-mixfix* (()), *elem-T*])

else []) *of*
 [] \Rightarrow **Const** \langle *Groups.zero* *T* \rangle
 | *ts* \Rightarrow *foldl1* (*fn* (*s*, *t*) \Rightarrow **Const** \langle *add-mset* *elem-T* *for* *s* *t* \rangle *ts*)
 end
 | *multiset-postproc* - - - *t* = *t*
 in *Nitpick-Model.register-term-postprocessor* **typ** \langle 'a *multiset* \rangle *multiset-postproc*
end
 \rangle

67.16 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: $\{\#\} = \text{mset } []$
by *simp*

lemma [*code*]: $\text{add-mset } x (\text{mset } xs) = \text{mset } (x \# xs)$
by *simp*

lemma [code]: *Multiset.is-empty* (*mset xs*) \longleftrightarrow *List.null xs*
by (*simp add: Multiset.is-empty-def*)

lemma *union-code* [code]: *mset xs* + *mset ys* = *mset (xs @ ys)*
by *simp*

lemma [code]: *image-mset f* (*mset xs*) = *mset (map f xs)*
by *simp*

lemma [code]: *filter-mset f* (*mset xs*) = *mset (filter f xs)*
by *simp*

lemma [code]: *mset xs* − *mset ys* = *mset (minus-list-mset xs ys)*
by *simp*

lemma [code]:
mset xs \cap *mset ys* =
*mset (snd (fold (λx (ys, zs).
if $x \in \text{set } ys$ then (remove1 x ys, x # zs) else (ys, zs)) xs (ys, [])))*
proof −
have $\bigwedge zs.$ *mset (snd (fold (λx (ys, zs).
if $x \in \text{set } ys$ then (remove1 x ys, x # zs) else (ys, zs)) xs (ys, zs)))* =
(*mset xs* \cap *mset ys*) + *mset zs*
by (*induct xs arbitrary: ys*)
(*auto simp add: inter-add-right1 inter-add-right2 ac-simps*)
then show ?thesis **by** *simp*
qed

lemma [code]:
mset xs \cup *mset ys* =
mset (case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs (ys, [])))
proof −
have $\bigwedge zs.$ *mset (case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs (ys, zs)))* =
(*mset xs* \cup *mset ys*) + *mset zs*
by (*induct xs arbitrary: ys*) (*simp-all add: multiset-eq-iff*)
then show ?thesis **by** *simp*
qed

declare *in-multiset-in-set* [code-unfold]

lemma [code]: *count (mset xs) x* = *fold (λy. if $x = y$ then Suc else id) xs 0*
proof −
have $\bigwedge n.$ *fold (λy. if $x = y$ then Suc else id) xs n* = *count (mset xs) x* + *n*
by (*induct xs*) *simp-all*
then show ?thesis **by** *simp*
qed

declare *set-mset-mset* [code]

declare *sorted-list-of-multiset-mset* [code]

lemma [code]: — not very efficient, but representation-ignorant!

mset-set *A* = *mset* (*sorted-list-of-set* *A*)

by (*metis mset-sorted-list-of-multiset sorted-list-of-mset-set*)

declare *size-mset* [code]

fun *subset-eq-mset-impl* :: 'a list \Rightarrow 'a list \Rightarrow bool option **where**

subset-eq-mset-impl [] *ys* = *Some* (*ys* \neq [])

| *subset-eq-mset-impl* (*Cons* *x xs*) *ys* = (case *List.extract* ((=) *x*) *ys* of
None \Rightarrow None

| *Some* (*ys1*, *ys2*) \Rightarrow *subset-eq-mset-impl* *xs* (*ys1* @ *ys2*))

lemma *subset-eq-mset-impl*: (*subset-eq-mset-impl* *xs* *ys* = None \longleftrightarrow \neg *mset* *xs* \subseteq # *mset* *ys*) \wedge

(*subset-eq-mset-impl* *xs* *ys* = *Some* *True* \longleftrightarrow *mset* *xs* \subset # *mset* *ys*) \wedge

(*subset-eq-mset-impl* *xs* *ys* = *Some* *False* \longrightarrow *mset* *xs* = *mset* *ys*)

proof (*induct xs arbitrary: ys*)

case (*Nil* *ys*)

show ?case **by** (*auto simp: subset-mset.zero-less-iff-neq-zero*)

next

case (*Cons* *x xs* *ys*)

show ?case

proof (*cases List.extract* ((=) *x*) *ys*)

case None

hence *x*: *x* \notin *set* *ys* **by** (*simp add: extract-None-iff*)

have *nle*: *False* **if** *mset* (*x* # *xs*) \subseteq # *mset* *ys*

using *set-mset-mono*[*OF that*] *x* **by** *simp*

moreover

have *False* **if** *mset* (*x* # *xs*) \subset # *mset* *ys*

proof —

from *that* **have** *mset* (*x* # *xs*) \subseteq # *mset* *ys* **by** *auto*

from *nle*[*OF this*] **show** ?thesis .

qed

ultimately show ?thesis **using** None **by** *auto*

next

case (*Some* *res*)

obtain *ys1* *y* *ys2* **where** *res*: *res* = (*ys1*, *y*, *ys2*) **by** (*cases res, auto*)

note *Some* = *Some*[*unfolded res*]

from *extract-SomeE*[*OF Some*] **have** *ys* = *ys1* @ *x* # *ys2* **by** *simp*

hence *id*: *mset* *ys* = *add-mset* *x* (*mset* (*ys1* @ *ys2*))

by *auto*

show ?thesis **unfolding** *subset-eq-mset-impl.simps*

by (*simp add: Some id Cons*)

qed

qed

lemma [code]: $mset\ xs \subseteq\# mset\ ys \longleftrightarrow subset\text{-}eq\text{-}mset\text{-}impl\ xs\ ys \neq None$
by (simp add: subset-eq-mset-impl)

lemma [code]: $mset\ xs \subset\# mset\ ys \longleftrightarrow subset\text{-}eq\text{-}mset\text{-}impl\ xs\ ys = Some\ True$
using subset-eq-mset-impl **by** blast

instantiation multiset :: (equal) equal
begin

definition

[code del]: $HOL.equal\ A\ (B :: 'a\ multiset) \longleftrightarrow A = B$

lemma [code]: $HOL.equal\ (mset\ xs)\ (mset\ ys) \longleftrightarrow subset\text{-}eq\text{-}mset\text{-}impl\ xs\ ys = Some\ False$

unfolding equal-multiset-def

using subset-eq-mset-impl[of xs ys] **by** (cases subset-eq-mset-impl xs ys, auto)

instance

by standard (simp add: equal-multiset-def)

end

declare sum-mset-sum-list [code]

lemma [code]: $prod\text{-}mset\ (mset\ xs) = fold\ times\ xs\ 1$

proof –

have $\bigwedge x. fold\ times\ xs\ x = prod\text{-}mset\ (mset\ xs) * x$

by (induct xs) (simp-all add: ac-simps)

then show ?thesis **by** simp

qed

Exercise for the casual reader: add implementations for (\leq) and $(<)$ (multiset order).

Quickcheck generators

context

includes term-syntax

begin

definition

$msetify :: 'a::typerep\ list \times (unit \Rightarrow Code\text{-}Evaluation.term)$

$\Rightarrow 'a\ multiset \times (unit \Rightarrow Code\text{-}Evaluation.term)$ **where**

[code-unfold]: $msetify\ xs = Code\text{-}Evaluation.valtermify\ mset\ \{\cdot\}\ xs$

end

instantiation multiset :: (random) random

begin

context

includes *state-combinator-syntax*
begin

definition

Quickcheck-Random.random i = Quickcheck-Random.random i $\circ\rightarrow$ ($\lambda xs.$ Pair (msetify xs))

instance ..

end

end

instantiation *multiset* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-multiset* :: (*'a multiset* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*

where

full-exhaustive-multiset f i = Quickcheck-Exhaustive.full-exhaustive ($\lambda xs.$ f (msetify xs)) i

instance ..

end

hide-const (**open**) *msetify*

67.17 BNF setup

definition *rel-mset* **where**

rel-mset R X Y \longleftrightarrow ($\exists xs\ ys.$ mset xs = X \wedge mset ys = Y \wedge list-all2 R xs ys)

lemma *mset-zip-take-Cons-drop-twice:*

assumes *length xs = length ys j \leq length xs*

shows *mset (zip (take j xs @ x # drop j xs) (take j ys @ y # drop j ys)) = add-mset (x,y) (mset (zip xs ys))*

using *assms*

proof (*induct xs ys arbitrary: x y j rule: list-induct2*)

case *Nil*

thus *?case*

by *simp*

next

case (*Cons x xs y ys*)

thus *?case*

proof (*cases j = 0*)

case *True*

thus *?thesis*

by *simp*

```

next
  case False
  then obtain k where k: j = Suc k
    by (cases j) simp
  hence k ≤ length xs
    using Cons.prems by auto
  hence mset (zip (take k xs @ x # drop k xs) (take k ys @ y # drop k ys)) =
    add-mset (x,y) (mset (zip xs ys))
    by (rule Cons.hyps(2))
  thus ?thesis
    unfolding k by auto
qed
qed

lemma ex-mset-zip-left:
  assumes length xs = length ys mset xs' = mset xs
  shows ∃ ys'. length ys' = length xs' ∧ mset (zip xs' ys') = mset (zip xs ys)
using assms
proof (induct xs ys arbitrary: xs' rule: list-induct2)
  case Nil
  thus ?case
    by auto
next
  case (Cons x xs y ys xs')
  obtain j where j-len: j < length xs' and nth-j: xs' ! j = x
    by (metis Cons.prems in-set-conv-nth list.set-intros(1) mset-eq-setD)

  define xsa where xsa = take j xs' @ drop (Suc j) xs'
  have mset xs' = {#x#} + mset xsa
    unfolding xsa-def using j-len nth-j
  by (metis Cons-nth-drop-Suc union-mset-add-mset-right add-mset-remove-trivial
add-diff-cancel-left'
    append-take-drop-id mset.simps(2) mset-append)
  hence ms-x: mset xsa = mset xs
    by (simp add: Cons.prems)
  then obtain ysa where
    len-a: length ysa = length xsa and ms-a: mset (zip xsa ysa) = mset (zip xs ys)
    using Cons.hyps(2) by blast

  define ys' where ys' = take j ysa @ y # drop j ysa
  have xs': xs' = take j xsa @ x # drop j xsa
    using ms-x j-len nth-j Cons.prems xsa-def
  by (metis append-eq-append-conv append-take-drop-id diff-Suc-Suc Cons-nth-drop-Suc
length-Cons
    length-drop size-mset)
  have j-len': j ≤ length xsa
    using j-len xs' xsa-def
  by (metis add-Suc-right append-take-drop-id length-Cons length-append less-eq-Suc-le
not-less)

```

```

have length ys' = length xs'
  unfolding ys'-def using Cons.premis len-a ms-x
  by (metis add-Suc-right append-take-drop-id length-Cons length-append mset-eq-length)
moreover have mset (zip xs' ys') = mset (zip (x # xs) (y # ys))
  unfolding xs' ys'-def
  by (rule trans[OF mset-zip-take-Cons-drop-twice])
    (auto simp: len-a ms-a j-len')
ultimately show ?case
  by blast
qed

```

lemma *list-all2-reorder-left-invariance*:

assumes *rel*: *list-all2* *R* *xs* *ys* and *ms-x*: *mset* *xs'* = *mset* *xs*
 shows \exists *ys'*. *list-all2* *R* *xs'* *ys'* \wedge *mset* *ys'* = *mset* *ys*

proof –

```

have len: length xs = length ys
  using rel list-all2-conv-all-nth by auto
obtain ys' where
  len': length xs' = length ys' and ms-xy: mset (zip xs' ys') = mset (zip xs ys)
  using len ms-x by (metis ex-mset-zip-left)
have list-all2 R xs' ys'
  using assms(1) len' ms-xy unfolding list-all2-iff by (blast dest: mset-eq-setD)
moreover have mset ys' = mset ys
  using len len' ms-xy map-snd-zip mset-map by metis
ultimately show ?thesis
  by blast
qed

```

lemma *ex-mset*: \exists *xs*. *mset* *xs* = *X*

by (induct *X*) (simp, metis mset.simps(2))

inductive *pred-mset* :: (*'a* \Rightarrow bool) \Rightarrow *'a* multiset \Rightarrow bool

where

```

  pred-mset P {#}
|  $\llbracket P\ a; \text{pred-mset } P\ M \rrbracket \Longrightarrow \text{pred-mset } P\ (\text{add-mset } a\ M)$ 

```

lemma *pred-mset-iff*: — TODO: alias for *Multiset.Ball*

$\langle \text{pred-mset } P\ M \longleftrightarrow \text{Multiset.Ball } M\ P \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)

proof

```

  assume ?P
  then show ?Q by induction simp-all
next
  assume ?Q
  then show ?P
    by (induction M) (auto intro: pred-mset.intros)
qed

```

bnf *'a* multiset

map: *image-mset*

```

sets: set-mset
bd: natLeq
wits: {#}
rel: rel-mset
pred: pred-mset
proof –
  show image-mset (g ∘ f) = image-mset g ∘ image-mset f for f g
  unfolding comp-def by (rule ext) (simp add: comp-def image-mset.compositionality)
  show (∧z. z ∈ set-mset X ⇒ f z = g z) ⇒ image-mset f X = image-mset g
X for f g X
  by (induct X) simp-all
  show card-order natLeq
  by (rule natLeq-card-order)
  show BNF-Cardinal-Arithmetic.cinfinite natLeq
  by (rule natLeq-cinfinite)
  show regularCard natLeq
  by (rule regularCard-natLeq)
  show ordLess2 (card-of (set-mset X)) natLeq for X
  by transfer
  (auto simp: finite-iff-ordLess-natLeq[symmetric])
  show rel-mset R OO rel-mset S ≤ rel-mset (R OO S) for R S
  unfolding rel-mset-def[abs-def] OO-def
  by (smt (verit, ccfv-SIG) list-all2-reorder-left-invariance list-all2-trans predi-
cate2I)
  show rel-mset R =
    (λx y. ∃z. set-mset z ⊆ {(x, y). R x y} ∧
    image-mset fst z = x ∧ image-mset snd z = y) for R
  unfolding rel-mset-def[abs-def]
  by (metis (no-types, lifting) ex-mset list.in-rel mem-Collect-eq mset-map set-mset-mset)
  show pred-mset P = (λx. Ball (set-mset x) P) for P
  by (simp add: fun-eq-iff pred-mset-iff)
qed auto

inductive rel-mset' :: ⟨('a ⇒ 'b ⇒ bool) ⇒ 'a multiset ⇒ 'b multiset ⇒ bool⟩
where
  Zero[intro]: rel-mset' R {#} {#}
| Plus[intro]: [R a b; rel-mset' R M N] ⇒ rel-mset' R (add-mset a M) (add-mset
b N)

lemma rel-mset-Zero: rel-mset R {#} {#}
unfolding rel-mset-def Grp-def by auto

declare multiset.count[simp]
declare count-Abs-multiset[simp]
declare multiset.count-inverse[simp]

lemma rel-mset-Plus:
  assumes ab: R a b
  and MN: rel-mset R M N

```



```

shows rel-mset R (add-mset a M) (add-mset b N)
proof –
  have  $\exists ya. \text{add-mset } a \text{ (image-mset fst } y) = \text{image-mset fst } ya \wedge$ 
     $\text{add-mset } b \text{ (image-mset snd } y) = \text{image-mset snd } ya \wedge$ 
     $\text{set-mset } ya \subseteq \{(x, y). R \ x \ y\}$ 
    if R a b and  $\text{set-mset } y \subseteq \{(x, y). R \ x \ y\}$  for y
    using that by (intro exI[of - add-mset (a,b) y]) auto
  thus ?thesis
  using assms
  unfolding multiset.rel-compp-Grp Grp-def by blast
qed

lemma rel-mset'-imp-rel-mset: rel-mset' R M N  $\implies$  rel-mset R M N
  by (induct rule: rel-mset'.induct) (auto simp: rel-mset-Zero rel-mset-Plus)

lemma rel-mset-size: rel-mset R M N  $\implies$  size M = size N
  unfolding multiset.rel-compp-Grp Grp-def by auto

lemma rel-mset-Zero-iff [simp]:
  shows rel-mset rel  $\{\#\}$  Y  $\longleftrightarrow$  Y =  $\{\#\}$  and rel-mset rel X  $\{\#\}$   $\longleftrightarrow$  X =  $\{\#\}$ 
  by (auto simp add: rel-mset-Zero dest: rel-mset-size)

lemma multiset-induct2[case-names empty addL addR]:
  assumes empty: P  $\{\#\}$   $\{\#\}$ 
    and addL:  $\bigwedge a \ M \ N. P \ M \ N \implies P \ (\text{add-mset } a \ M) \ N$ 
    and addR:  $\bigwedge a \ M \ N. P \ M \ N \implies P \ M \ (\text{add-mset } a \ N)$ 
  shows P M N
  by (induct N rule: multiset-induct; induct M rule: multiset-induct) (auto simp:
assms)

lemma multiset-induct2-size[consumes 1, case-names empty add]:
  assumes c: size M = size N
    and empty: P  $\{\#\}$   $\{\#\}$ 
    and add:  $\bigwedge a \ b \ M \ N \ a \ b. P \ M \ N \implies P \ (\text{add-mset } a \ M) \ (\text{add-mset } b \ N)$ 
  shows P M N
  using c
proof (induct M arbitrary: N rule: measure-induct-rule[of size])
  case (less M)
  show ?case
  proof(cases M =  $\{\#\}$ )
    case True hence N =  $\{\#\}$  using less.prems by auto
    thus ?thesis using True empty by auto
  next
    case False then obtain M1 a where M: M = add-mset a M1 by (metis
multi-nonempty-split)
    have N  $\neq$   $\{\#\}$  using False less.prems by auto
    then obtain N1 b where N: N = add-mset b N1 by (metis multi-nonempty-split)
    have size M1 = size N1 using less.prems unfolding M N by auto
    thus ?thesis using M N less.hyps add by auto

```

qed
qed

lemma *msed-map-invL*:

assumes *image-mset* *f* (*add-mset* *a* *M*) = *N*

shows $\exists N1. N = \text{add-mset } (f \ a) \ N1 \wedge \text{image-mset } f \ M = N1$

proof –

have $f \ a \in\# \ N$

using *assms* *multiset.set-map*[*of f add-mset a M*] **by** *auto*

then obtain *N1* **where** $N = \text{add-mset } (f \ a) \ N1$ **using** *multi-member-split*

by *metis*

have *image-mset* *f* *M* = *N1* **using** *assms* **unfolding** *N* **by** *simp*

thus *?thesis* **using** *N* **by** *blast*

qed

lemma *msed-map-invR*:

assumes *image-mset* *f* *M* = *add-mset* *b* *N*

shows $\exists M1 \ a. M = \text{add-mset } a \ M1 \wedge f \ a = b \wedge \text{image-mset } f \ M1 = N$

proof –

obtain *a* **where** $a \in\# \ M$ **and** $f \ a = b$

using *multiset.set-map*[*of f M*] **unfolding** *assms*

by (*metis image-iff union-single-eq-member*)

then obtain *M1* **where** $M = \text{add-mset } a \ M1$ **using** *multi-member-split* **by**

metis

have *image-mset* *f* *M1* = *N* **using** *assms* **unfolding** *M fa[symmetric]* **by** *simp*

thus *?thesis* **using** *M fa* **by** *blast*

qed

lemma *msed-rel-invL*:

assumes *rel-mset* *R* (*add-mset* *a* *M*) *N*

shows $\exists N1 \ b. N = \text{add-mset } b \ N1 \wedge R \ a \ b \wedge \text{rel-mset } R \ M \ N1$

proof –

obtain *K* **where** $KM: \text{image-mset } \text{fst } K = \text{add-mset } a \ M$

and $KN: \text{image-mset } \text{snd } K = N$ **and** $sK: \text{set-mset } K \subseteq \{(a, b). R \ a \ b\}$

using *assms*

unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*

obtain *K1 ab* **where** $K: K = \text{add-mset } ab \ K1$ **and** $a: \text{fst } ab = a$

and $K1M: \text{image-mset } \text{fst } K1 = M$ **using** *msed-map-invR*[*OF KM*] **by** *auto*

obtain *N1* **where** $N = \text{add-mset } (\text{snd } ab) \ N1$ **and** $K1N1: \text{image-mset } \text{snd } K1 = N1$

using *msed-map-invL*[*OF KN*[*unfolded K*]] **by** *auto*

have $Rab: R \ a \ (\text{snd } ab)$ **using** *sK a* **unfolding** *K* **by** *auto*

have *rel-mset* *R* *M* *N1* **using** *sK K1M K1N1*

unfolding *K* *multiset.rel-compp-Grp Grp-def* **by** *auto*

thus *?thesis* **using** *N Rab* **by** *auto*

qed

lemma *msed-rel-invR*:

assumes *rel-mset* *R* *M* (*add-mset* *b* *N*)

shows $\exists M1\ a.\ M = \text{add-mset}\ a\ M1 \wedge R\ a\ b \wedge \text{rel-mset}\ R\ M1\ N$
proof –
 obtain K **where** $KN: \text{image-mset}\ \text{snd}\ K = \text{add-mset}\ b\ N$
 and $KM: \text{image-mset}\ \text{fst}\ K = M$ **and** $sK: \text{set-mset}\ K \subseteq \{(a, b). R\ a\ b\}$
 using *assms*
 unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*
 obtain $K1\ ab$ **where** $K: K = \text{add-mset}\ ab\ K1$ **and** $b: \text{snd}\ ab = b$
 and $K1N: \text{image-mset}\ \text{snd}\ K1 = N$ **using** *msed-map-invR[OF KN]* **by** *auto*
 obtain $M1$ **where** $M: M = \text{add-mset}\ (\text{fst}\ ab)\ M1$ **and** $K1M1: \text{image-mset}\ \text{fst}\ K1 = M1$
 using *msed-map-invL[OF KM[unfolded K]]* **by** *auto*
 have $Rab: R\ (\text{fst}\ ab)\ b$ **using** $sK\ b$ **unfolding** K **by** *auto*
 have $\text{rel-mset}\ R\ M1\ N$ **using** $sK\ K1N\ K1M1$
 unfolding K *multiset.rel-compp-Grp Grp-def* **by** *auto*
 thus *?thesis* **using** $M\ Rab$ **by** *auto*
qed

lemma *rel-mset-imp-rel-mset'*:
 assumes $\text{rel-mset}\ R\ M\ N$
 shows $\text{rel-mset}'\ R\ M\ N$
using *assms* **proof**(*induct M arbitrary: N rule: measure-induct-rule[of size]*)
 case (*less M*)
 have $c: \text{size}\ M = \text{size}\ N$ **using** *rel-mset-size[OF less.prem]* .
 show *?case*
proof(*cases M = \{\#\}*)
 case *True* **hence** $N = \{\#\}$ **using** c **by** *simp*
 thus *?thesis* **using** *True rel-mset'.Zero* **by** *auto*
 next
 case *False* **then** obtain $M1\ a$ **where** $M: M = \text{add-mset}\ a\ M1$ **by** (*metis multi-nonempty-split*)
 obtain $N1\ b$ **where** $N: N = \text{add-mset}\ b\ N1$ **and** $R: R\ a\ b$ **and** $ms: \text{rel-mset}\ R\ M1\ N1$
 using *msed-rel-invL[OF less.prem[unfolded M]]* **by** *auto*
 have $\text{rel-mset}'\ R\ M1\ N1$ **using** *less.hyps[of M1 N1] ms* **unfolding** M **by** *simp*
 thus *?thesis* **using** *rel-mset'.Plus[of R a b, OF R]* **unfolding** $M\ N$ **by** *simp*
qed
qed

lemma *rel-mset-rel-mset'*: $\text{rel-mset}\ R\ M\ N = \text{rel-mset}'\ R\ M\ N$
using *rel-mset-imp-rel-mset' rel-mset'-imp-rel-mset* **by** *auto*

The main end product for *rel-mset*: inductive characterization:

lemmas *rel-mset-induct[case-names empty add, induct pred: rel-mset] = rel-mset'.induct[unfolded rel-mset-rel-mset'[symmetric]]*

67.18 Size setup

lemma *size-multiset-o-map*: $\text{size-multiset}\ g \circ \text{image-mset}\ f = \text{size-multiset}\ (g \circ f)$
apply (*rule ext*)
subgoal for x **by** (*induct x*) *auto*

```

done

setup <
  BNF-LFP-Size.register-size-global type-name <multiset> const-name <size-multiset>
  @{thm size-multiset-overloaded-def}
  @{thms size-multiset-empty size-multiset-single size-multiset-union size-empty
size-single
  size-union}
  @{thms size-multiset-o-map}
>

```

67.19 Lemmas about Size

lemma *size-mset-SucE*: $\text{size } A = \text{Suc } n \implies (\bigwedge a B. A = \{\#a\} + B \implies \text{size } B = n \implies P) \implies P$
by (*cases A*) (*auto simp add: ac-simps*)

lemma *size-Suc-Diff1*: $x \in \# M \implies \text{Suc } (\text{size } (M - \{\#x\})) = \text{size } M$
using *arg-cong[OF insert-DiffM, of - - size]* **by** *simp*

lemma *size-Diff-singleton*: $x \in \# M \implies \text{size } (M - \{\#x\}) = \text{size } M - 1$
by (*simp flip: size-Suc-Diff1*)

lemma *size-Diff-singleton-if*: $\text{size } (A - \{\#x\}) = (\text{if } x \in \# A \text{ then } \text{size } A - 1 \text{ else } \text{size } A)$
by (*simp add: diff-single-trivial size-Diff-singleton*)

lemma *size-Un-Int*: $\text{size } A + \text{size } B = \text{size } (A \cup \# B) + \text{size } (A \cap \# B)$
by (*metis inter-subset-eq-union size-union subset-mset.diff-add union-diff-inter-eq-sup*)

lemma *size-Un-disjoint*: $A \cap \# B = \{\#\} \implies \text{size } (A \cup \# B) = \text{size } A + \text{size } B$
using *size-Un-Int[of A B]* **by** *simp*

lemma *size-Diff-subset-Int*: $\text{size } (M - M') = \text{size } M - \text{size } (M \cap \# M')$
by (*metis diff-intersect-left-idem size-Diff-submset subset-mset.inf-le1*)

lemma *diff-size-le-size-Diff*: $\text{size } (M :: \text{multiset}) - \text{size } M' \leq \text{size } (M - M')$
by (*simp add: diff-le-mono2 size-Diff-subset-Int size-mset-mono*)

lemma *size-Diff1-less*: $x \in \# M \implies \text{size } (M - \{\#x\}) < \text{size } M$
by (*rule Suc-less-SucD*) (*simp add: size-Suc-Diff1*)

lemma *size-Diff2-less*: $x \in \# M \implies y \in \# M \implies \text{size } (M - \{\#x\} - \{\#y\}) < \text{size } M$
by (*metis less-imp-diff-less size-Diff1-less size-Diff-subset-Int*)

lemma *size-Diff1-le*: $\text{size } (M - \{\#x\}) \leq \text{size } M$
by (*cases x* $\in \# M$) (*simp-all add: size-Diff1-less less-imp-le diff-single-trivial*)

lemma *size-psubset*: $M \subseteq\# M' \implies \text{size } M < \text{size } M' \implies M \subset\# M'$
using *less-irrefl subset-mset-def* **by** *blast*

lifting-update *multiset.lifting*
lifting-forget *multiset.lifting*

hide-const (**open**) *wcount*

67.20 The set of multisets of a given size

The following operator gives the set of all multisets consisting of n elements drawn from the set A . In other words: all the different ways to put n unlabelled balls into the labelled bins A .

definition *multisets-of-size* :: 'a set \Rightarrow nat \Rightarrow 'a multiset set **where**
multisets-of-size $A\ n = \{X. \text{set-mset } X \subseteq A \wedge \text{size } X = n\}$

lemma
assumes $X \in \text{multisets-of-size } A\ n$
shows *multisets-of-size-subset*: $\text{set-mset } X \subseteq A$
and *multisets-of-size-size*: $\text{size } X = n$
using *assms* **by** (*auto simp: multisets-of-size-def*)

lemma *multisets-of-size-mono*:
assumes $A \subseteq B$
shows *multisets-of-size* $A\ n \subseteq \text{multisets-of-size } B\ n$
unfolding *multisets-of-size-def*
by (*intro Collect-mono*) (*use assms in auto*)

lemma *multisets-of-size-0* [*simp*]: $\text{multisets-of-size } A\ 0 = \{\{\#\}\}$

proof (*intro equalityI subsetI*)
fix $h :: 'a \text{ multiset}$ **assume** $h \in \{\{\#\}\}$
thus $h \in \text{multisets-of-size } A\ 0$
by (*auto simp: multisets-of-size-def*)
qed (*auto simp: multisets-of-size-def fun-eq-iff*)

lemma *multisets-of-size-empty* [*simp*]: $n > 0 \implies \text{multisets-of-size } \{\}\ n = \{\}$
by (*auto simp: multisets-of-size-def fun-eq-iff*)

lemma *count-le-size*: $\text{count } X\ x \leq \text{size } X$
by (*induction X*) *auto*

lemma *bij-betw-multisets-of-size-insert*:
assumes $a \notin A$
shows *bij-betw* $(\lambda(m,X). X + \text{replicate-mset } m\ a)$
 $(\text{SIGMA } m:\{0..n\}. \text{multisets-of-size } A\ (n - m))\ (\text{multisets-of-size } (\text{insert } a\ A)\ n)$
proof –
define B **where** $B = (\text{SIGMA } m:\{0..n\}. \text{multisets-of-size } A\ (n - m))$
define C **where** $C = (\text{multisets-of-size } (\text{insert } a\ A)\ n)$

```

define f where f = ( $\lambda(m, X). X + \text{replicate-mset } m \ a$ )
define g where g = ( $\lambda X. (\text{count } X \ a, \text{filter-mset } (\lambda x. x \neq a) \ X)$ )
note defs = B-def C-def f-def g-def

have *: size (filter-mset ( $\lambda x. x \neq a$ ) X) = size X - count X a for X
proof -
  have size X = size (filter-mset ( $\lambda x. x \neq a$ ) X) + count X a
    by (induction X) auto
  thus ?thesis
    by linarith
qed

have 1: f mX ∈ C if mX ∈ B for mX
  using that by (auto simp: multisets-of-size-def defs split: if-splits)

have 2: g X ∈ B if X ∈ C for X
  using that by (auto simp: multisets-of-size-def count-le-size * defs)

have 3: g (f mX) = mX if mX ∈ B for mX
  using that assms
  by (auto simp: multisets-of-size-def multiset-eq-iff defs simp flip: not-in-iff)

have 4: f (g X) = X if X ∈ C for X
  using that
  by (auto simp: multisets-of-size-def multiset-eq-iff defs simp flip: not-in-iff)

have f ‘ B = C
  using 1 2 4 unfolding set-eq-iff image-iff by metis
moreover have inj-on f B
  using 3 unfolding inj-on-def by metis
ultimately show ?thesis
  unfolding bij-betw-def defs by metis
qed

lemma multisets-of-size-insert:
  assumes a ∉ A
  shows multisets-of-size (insert a A) n =
    ( $\bigcup_{m \leq n. (\lambda X. X + \text{replicate-mset } m \ a) \text{ ‘ multisets-of-size } A \ (n - m)}$ )
proof -
  have multisets-of-size (insert a A) n =
    ( $\lambda(m, X). X + \text{replicate-mset } m \ a$ ) ‘ ( $\text{SIGMA } m:\{0..n\}. \text{multisets-of-size } A \ (n - m)$ )
  using bij-betw-multisets-of-size-insert[OF assms, of n] unfolding bij-betw-def
by simp
  also have ... = ( $\bigcup_{m \leq n. (\lambda X. X + \text{replicate-mset } m \ a) \text{ ‘ multisets-of-size } A \ (n - m)}$ )
  unfolding Sigma-def image-UN atLeast0AtMost image-insert image-empty
prod.case
    UNION-singleton-eq-range image-image by (rule refl)

```

finally show *?thesis* .
qed

primrec *multisets-of-size-list* :: 'a list \Rightarrow nat \Rightarrow 'a list list **where**
multisets-of-size-list [] n = (if $n = 0$ then [[]] else [])
| *multisets-of-size-list* ($x \# xs$) n =
[*replicate* m x @ ys . $m \leftarrow [0..n+1]$, $ys \leftarrow multisets-of-size-list\ xs\ (n - m)$]

lemma *multisets-of-size-list-correct*:

assumes *distinct xs*

shows $mset\ 'set\ (multisets-of-size-list\ xs\ n) = multisets-of-size\ (set\ xs)\ n$

using *assms*

proof (*induction xs arbitrary: n*)

case *Nil*

thus *?case*

by (*cases n = 0*) *auto*

next

case (*Cons x xs n*)

have *IH*: $multisets-of-size\ (set\ xs)\ n = mset\ 'set\ (multisets-of-size-list\ xs\ n)$ **for**

n

by (*rule Cons.IH [symmetric]*) (*use Cons.prem*s **in** *auto*)

from *Cons.prem*s **have** $x \notin set\ xs$

by *auto*

thus *?case*

by (*simp add: multisets-of-size-insert image-UN atLeastLessThanSuc-atLeastAtMost image-image*

add-ac atLeast0AtMost IH del: upt-Suc)

qed

lemma *multisets-of-size-code* [*code*]:

$multisets-of-size\ (set\ xs)\ n = set\ (map\ mset\ (multisets-of-size-list\ (remdups\ xs)\ n))$

using *multisets-of-size-list-correct*[*of remdups xs*] **by** *simp*

lemma *finite-multisets-of-size* [*intro*]:

assumes *finite A*

shows *finite* ($multisets-of-size\ A\ n$)

using *assms*

proof (*induction arbitrary: n rule: finite-induct*)

case *empty*

thus *?case*

by (*cases n = 0*) *auto*

next

case (*insert x A n*)

have *finite* ($SIGMA\ m:\{0..n\}.\ multisets-of-size\ A\ (n - m)$)

by (*auto intro: insert.IH*)

also have *?this* \longleftrightarrow *finite* ($multisets-of-size\ (insert\ x\ A)\ n$)

by (*rule bij-betw-finite, rule bij-betw-multisets-of-size-insert*) *fact*

finally show *?case* .

qed

lemma *card-multisets-of-size*:

assumes *finite A*

shows $\text{card} (\text{multisets-of-size } A \ n) = (\text{card } A + n - 1) \text{ choose } n$

using *assms*

proof (*induction A arbitrary: n rule: finite-induct*)

case *empty*

thus *?case*

by (*cases n = 0*) *auto*

next

case (*insert a A n*)

have $\text{card} (\text{multisets-of-size} (\text{insert } a \ A) \ n) = \text{card} (\text{SIGMA } m:\{0..n\}. \text{multisets-of-size } A \ (n - m))$

using *bij-betw-same-card[OF bij-betw-multisets-of-size-insert[of a A], of n] insert.hyps*

by *simp*

also have $\dots = (\sum m=0..n. \text{card} (\text{multisets-of-size } A \ (n - m)))$

by (*intro card-SigmaI*) (*use insert.hyps in auto*)

also have $\dots = (\sum m=0..n. (\text{card } A + (n - m) - 1) \text{ choose } (n - m))$

by (*intro sum.cong insert.IH refl*)

also have $\dots = (\sum m=0..n. (\text{card } A + m - 1) \text{ choose } m)$

by (*intro sum.reindex-bij-witness[of - $\lambda m. n - m$ $\lambda m. n - m$] auto*)

also have $\dots = (\text{card } A + n) \text{ choose } n$

proof (*cases card A = 0*)

case *True*

have $(\sum m=0..n. (\text{card } A + m - 1) \text{ choose } m) = (\sum m \in \{0\}. (m-1) \text{ choose } m)$

by (*intro sum.mono-neutral-cong-right*) (*use True in auto*)

also have $\dots = 1$

by *simp*

also have $\dots = (\text{card } A + n) \text{ choose } n$

using *True by simp*

finally show *?thesis* .

next

case *False*

have $(\sum m=0..n. (\text{card } A + m - 1) \text{ choose } m) = (\sum m \leq n. ((\text{card } A - 1) + m) \text{ choose } m)$

by (*intro sum.cong*) (*use False in <auto simp: algebra-simps>*)

also have $\dots = (\sum m \leq n. ((\text{card } A - 1) + m) \text{ choose } (\text{card } A - 1))$

by (*subst binomial-symmetric*) *auto*

also have $\dots = (\text{card } A + n) \text{ choose } n$

using *choose-rising-sum(2)[of card A - 1 n] False by simp*

finally show *?thesis* .

qed

finally show *?case*

using *insert.hyps by simp*

qed

end

68 More Theorems about the Multiset Order

theory *Multiset-Order*
imports *Multiset*
begin

68.1 Alternative Characterizations

68.1.1 The Dershowitz–Manna Ordering

definition *multp_{DM}* **where**

$multp_{DM} \ r \ M \ N \longleftrightarrow$
 $(\exists X \ Y. \ X \neq \{\#\} \wedge X \subseteq\# \ N \wedge M = (N - X) + Y \wedge (\forall k. \ k \in\# \ Y \longrightarrow (\exists a. \ a \in\# \ X \wedge r \ k \ a)))$

lemma *multp_{DM}-imp-multp*:

$multp_{DM} \ r \ M \ N \implies multp \ r \ M \ N$

proof –

assume $multp_{DM} \ r \ M \ N$

then obtain $X \ Y$ **where**

$X \neq \{\#\}$ **and** $X \subseteq\# \ N$ **and** $M = N - X + Y$ **and** $\forall k. \ k \in\# \ Y \longrightarrow (\exists a. \ a \in\# \ X \wedge r \ k \ a)$

unfolding *multp_{DM}-def* **by** *blast*

then have $multp \ r \ (N - X + Y) \ (N - X + X)$

by (*intro one-step-implies-multp*) (*auto simp: Bex-def trans-def*)

with $\langle M = N - X + Y \rangle \langle X \subseteq\# \ N \rangle$ **show** $multp \ r \ M \ N$

by (*metis subset-mset.diff-add*)

qed

68.1.2 The Huet–Oppen Ordering

definition *multp_{HO}* **where**

$multp_{HO} \ r \ M \ N \longleftrightarrow M \neq N \wedge (\forall y. \ count \ N \ y < count \ M \ y \longrightarrow (\exists x. \ r \ y \ x \wedge count \ M \ x < count \ N \ x))$

lemma *multp-imp-multp_{HO}*:

assumes *asympt* r **and** *transp* r

shows $multp \ r \ M \ N \implies multp_{HO} \ r \ M \ N$

unfolding *multp-def mult-def*

proof (*induction rule: trancl-induct*)

case (*base P*)

then show *?case*

using $\langle asympt \ r \rangle$

by (*auto elim!: mult1-lessE simp: count-eq-zero-iff multp_{HO}-def split: if-splits dest!: Suc-lessD*)

next

case (*step N P*)

```

from step(3) have  $M \neq N$  and
  **:  $\bigwedge y. \text{count } N y < \text{count } M y \implies (\exists x. r y x \wedge \text{count } M x < \text{count } N x)$ 
  by (simp-all add: multpHO-def)
from step(2) obtain  $M0 a K$  where
  *:  $P = \text{add-mset } a M0 N = M0 + K a \notin \# K \bigwedge b. b \in \# K \implies r b a$ 
  using  $\langle \text{asympt } r \rangle$  by (auto elim: mult1-lessE)
from  $\langle M \neq N \rangle$  **  $*(1,2,3)$  have  $M \neq P$ 
  using  $*(4)$   $\langle \text{asympt } r \rangle$ 
  by (metis asymptD add-cancel-right-right add-diff-cancel-left' add-mset-add-single
count-inI
count-union diff-diff-add-mset diff-single-trivial in-diff-count multi-member-last)
moreover
have count-a:  $\exists z. r a z \wedge \text{count } M z < \text{count } P z$  if  $\text{count } P a \leq \text{count } M a$ 
proof –
  from  $\langle a \notin \# K \rangle$  and that have  $\text{count } N a < \text{count } M a$ 
  unfolding  $*(1,2)$  by (auto simp add: not-in-iff)
  with ** obtain  $z$  where  $z: r a z \text{count } M z < \text{count } N z$ 
  by blast
  with * have  $\text{count } N z \leq \text{count } P z$ 
  using  $\langle \text{asympt } r \rangle$ 
  by (metis add-diff-cancel-left' add-mset-add-single asymptD diff-diff-add-mset
diff-single-trivial in-diff-count not-le-imp-less)
  with  $z$  show ?thesis by auto
qed
have  $\exists x. r y x \wedge \text{count } M x < \text{count } P x$  if count-y:  $\text{count } P y < \text{count } M y$  for
y
proof (cases y = a)
  case True
    with count-y count-a show ?thesis by auto
  next
    case False
    show ?thesis
    proof (cases y ∈ # K)
      case True
        with  $*(4)$  have  $r y a$  by simp
        then show ?thesis
        by (cases count P a ≤ count M a) (auto dest: count-a intro: ⟨transp r⟩[THEN
transpD])
      next
        case False
        with  $\langle y \neq a \rangle$  have  $\text{count } P y = \text{count } N y$  unfolding  $*(1,2)$ 
        by (simp add: not-in-iff)
        with count-y ** obtain  $z$  where  $z: r y z \text{count } M z < \text{count } N z$  by auto
        show ?thesis
        proof (cases z ∈ # K)
          case True
            with  $*(4)$  have  $r z a$  by simp
            with  $z(1)$  show ?thesis
            by (cases count P a ≤ count M a) (auto dest!: count-a intro: ⟨transp

```

```

r>[THEN transpD])
  next
  case False
  with ⟨a ∉# K⟩ have count N z ≤ count P z unfolding *
    by (auto simp add: not-in-iff)
  with z show ?thesis by auto
qed
qed
qed
ultimately show ?case unfolding multpHO-def by blast
qed

lemma multpHO-imp-multpDM: multpHO r M N ⇒ multpDM r M N
unfolding multpDM-def
proof (intro iffI exI conjI)
  assume multpHO r M N
  then obtain z where z: count M z < count N z
    unfolding multpHO-def by (auto simp: multiset-eq-iff nat-neq-iff)
  define X where X = N - M
  define Y where Y = M - N
  from z show X ≠ {} unfolding X-def by (auto simp: multiset-eq-iff not-less-eq-eq
    Suc-le-eq)
  from z show X ⊆# N unfolding X-def by auto
  show M = (N - X) + Y unfolding X-def Y-def multiset-eq-iff count-union
    count-diff by force
  show ∀k. k ∈# Y ⟶ (∃ a. a ∈# X ∧ r k a)
  proof (intro allI impI)
    fix k
    assume k ∈# Y
    then have count N k < count M k unfolding Y-def
      by (auto simp add: in-diff-count)
    with ⟨multpHO r M N⟩ obtain a where r k a and count M a < count N a
      unfolding multpHO-def by blast
    then show ∃ a. a ∈# X ∧ r k a unfolding X-def
      by (auto simp add: in-diff-count)
  qed
qed

lemma multp-eq-multpDM: asymp r ⇒ transp r ⇒ multp r = multpDM r
using multpDM-imp-multp multp-imp-multpHO[THEN multpHO-imp-multpDM]
by blast

lemma multp-eq-multpHO: asymp r ⇒ transp r ⇒ multp r = multpHO r
using multpHO-imp-multpDM[THEN multpDM-imp-multp] multp-imp-multpHO
by blast

lemma multpDM-plus-plusI[simp]:
  assumes multpDM R M1 M2
  shows multpDM R (M + M1) (M + M2)

```

proof –

from *assms* **obtain** $X \ Y$ **where**
 $X \neq \{\#\}$ **and** $X \subseteq\# M2$ **and** $M1 = M2 - X + Y$ **and** $\forall k. k \in\# Y \longrightarrow$
 $(\exists a. a \in\# X \wedge R \ k \ a)$
 unfolding *multp_{DM}-def* **by** *auto*

show *multp_{DM}* $R \ (M + M1) \ (M + M2)$
 unfolding *multp_{DM}-def*
 proof (*intro exI conjI*)
 show $X \neq \{\#\}$
 using $\langle X \neq \{\#\} \rangle$ **by** *simp*
 next
 show $X \subseteq\# M + M2$
 using $\langle X \subseteq\# M2 \rangle$
 by (*simp add: subset-mset.add-increasing*)
 next
 show $M + M1 = M + M2 - X + Y$
 using $\langle X \subseteq\# M2 \rangle \ \langle M1 = M2 - X + Y \rangle$
 by (*metis multiset-diff-union-assoc union-assoc*)
 next
 show $\forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge R \ k \ a)$
 using $\langle \forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge R \ k \ a) \rangle$ **by** *simp*
 qed
qed

lemma *multp_{HO}-plus-plus[simp]*: *multp_{HO}* $R \ (M + M1) \ (M + M2) \longleftrightarrow$ *multp_{HO}*
 $R \ M1 \ M2$
unfolding *multp_{HO}-def* **by** *simp*

lemma *strict-subset-implies-multp_{DM}*: $A \subset\# B \implies$ *multp_{DM}* $r \ A \ B$
unfolding *multp_{DM}-def*
by (*metis add.right-neutral add-diff-cancel-right' empty-iff mset-subset-eq-add-right*
set-mset-empty subset-mset.lessE)

lemma *strict-subset-implies-multp_{HO}*: $A \subset\# B \implies$ *multp_{HO}* $r \ A \ B$
unfolding *multp_{HO}-def*
by (*simp add: leD mset-subset-eq-count*)

lemma *multp_{HO}-implies-one-step-strong*:
assumes *multp_{HO}* $R \ A \ B$
defines $J \equiv B - A$ **and** $K \equiv A - B$
shows $J \neq \{\#\}$ **and** $\forall k \in\# K. \exists x \in\# J. R \ k \ x$
proof –
show $J \neq \{\#\}$
using $\langle \text{multp}_{HO} \ R \ A \ B \rangle$
by (*metis Diff-eq-empty-iff-mset J-def add.right-neutral multp_{DM}-def multp_{HO}-imp-multp_{DM}*
multp_{HO}-plus-plus subset-mset.add-diff-inverse subset-mset.le-zero-eq)

show $\forall k \in\# K. \exists x \in\# J. R \ k \ x$

using $\langle \text{multp}_{HO} R A B \rangle$
by (*metis J-def K-def in-diff-count multp_{HO}-def*)
qed

lemma *multp_{HO}-minus-inter-minus-inter-iff*:
fixes $M1\ M2 :: - \text{multiset}$
shows $\text{multp}_{HO} R (M1 - M2) (M2 - M1) \longleftrightarrow \text{multp}_{HO} R M1 M2$
by (*metis diff-intersect-left-idem multiset-inter-commute multp_{HO}-plus-plus subset-mset.add-diff-inverse subset-mset.inf.cobounded1*)

lemma *multp_{HO}-iff-set-mset-less_{HO}-set-mset*:
 $\text{multp}_{HO} R M1 M2 \longleftrightarrow (\text{set-mset } (M1 - M2) \neq \text{set-mset } (M2 - M1) \wedge$
 $(\forall y \in \# M1 - M2. (\exists x \in \# M2 - M1. R\ y\ x)))$
unfolding *multp_{HO}-minus-inter-minus-inter-iff* [of $R\ M1\ M2$, *symmetric*]
unfolding *multp_{HO}-def*
unfolding *count-minus-inter-lt-count-minus-inter-iff*
unfolding *minus-inter-eq-minus-inter-iff*
by *auto*

68.1.3 Monotonicity

lemma *multp_{DM}-mono-strong*:
 $\text{multp}_{DM} R M1 M2 \implies (\bigwedge x\ y. x \in \# M1 \implies y \in \# M2 \implies R\ x\ y \implies S\ x\ y)$
 $\implies \text{multp}_{DM} S M1 M2$
unfolding *multp_{DM}-def*
by (*metis add-diff-cancel-left' in-diffD subset-mset.diff-add*)

lemma *multp_{HO}-mono-strong*:
 $\text{multp}_{HO} R M1 M2 \implies (\bigwedge x\ y. x \in \# M1 \implies y \in \# M2 \implies R\ x\ y \implies S\ x\ y)$
 $\implies \text{multp}_{HO} S M1 M2$
unfolding *multp_{HO}-def*
by (*metis count-inI less-zeroE*)

68.1.4 Properties of Orders

Asymmetry The following lemma is a negative result stating that asymmetry of an arbitrary binary relation cannot be simply lifted to *multp_{HO}*. It suffices to have four distinct values to build a counterexample.

lemma *asympt-not-liftable-to-multp_{HO}*:
fixes $a\ b\ c\ d :: 'a$
assumes *distinct* $[a, b, c, d]$
shows $\neg (\forall (R :: 'a \Rightarrow 'a \Rightarrow \text{bool}). \text{asympt } R \longrightarrow \text{asympt } (\text{multp}_{HO} R))$
proof –
define $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**
 $R = (\lambda x\ y. x = a \wedge y = c \vee x = b \wedge y = d \vee x = c \wedge y = b \vee x = d \wedge y = a)$
from *assms*(1) **have** $\{\#a, \#b\} \neq \{\#c, \#d\}$
by (*metis add-mset-add-single distinct.simps(2) list.set(1) list.simps(15) multi-member-this set-mset-add-mset-insert set-mset-single*)

```

from assms(1) have asym R
  by (auto simp: R-def intro: asym-onI)
moreover have  $\neg$  asym (multpHO R)
  unfolding asym-on-def Set.ball-simps not-all not-imp not-not
proof (intro exI conjI)
  show multpHO R  $\{ \#a, b \} \{ \#c, d \}$ 
    unfolding multpHO-def
    using  $\langle \{ \#a, b \} \neq \{ \#c, d \} \rangle$  R-def assms by auto
next
  show multpHO R  $\{ \#c, d \} \{ \#a, b \}$ 
    unfolding multpHO-def
    using  $\langle \{ \#a, b \} \neq \{ \#c, d \} \rangle$  R-def assms by auto
qed
ultimately show ?thesis
  unfolding not-all not-imp by auto
qed

```

However, if the binary relation is both asymmetric and transitive, then *multp_{HO}* is also asymmetric.

lemma *asym-on-multp_{HO}*:

```

assumes asym-on A R and trans-on A R and
  B-sub-A:  $\bigwedge M. M \in B \implies \text{set-mset } M \subseteq A$ 
shows asym-on B (multpHO R)
proof (rule asym-onI)
  fix M1 M2 :: 'a multiset
  assume M1 ∈ B M2 ∈ B multpHO R M1 M2

```

```

from  $\langle \text{trans-on } A \text{ } R \rangle$  B-sub-A have tran: trans-on (set-mset (M1 - M2)) R
  using  $\langle M1 \in B \rangle$ 
  by (meson in-diffD subset-eq trans-on-subset)

```

```

from  $\langle \text{asym-on } A \text{ } R \rangle$  B-sub-A have asym: asym-on (set-mset (M1 - M2)) R
  using  $\langle M1 \in B \rangle$ 
  by (meson in-diffD subset-eq asym-on-subset)

```

```

show  $\neg \text{multp}_{HO} \text{ } R \text{ } M2 \text{ } M1$ 
proof (cases M1 - M2 = {#})
  case True
    then show ?thesis
      using multpHO-implies-one-step-strong(1) by metis
  next
    case False
    hence  $\exists m \in \#M1 - M2. \forall x \in \#M1 - M2. x \neq m \longrightarrow \neg R \text{ } m \text{ } x$ 
    using Finite-Set.bex-max-element[of set-mset (M1 - M2) R, OF finite-set-mset
asym tran]
    by simp
    with  $\langle \text{trans-on } A \text{ } R \rangle$  B-sub-A have  $\exists y \in \#M2 - M1. \forall x \in \#M1 - M2. \neg R$ 
y x

```

```

    using  $\langle \text{multp}_{HO} R M1 M2 \rangle [THEN \text{multp}_{HO}\text{-implies-one-step-strong}(2)]$ 
    using asym [THEN irreflp-on-if-asym-on, THEN irreflp-onD]
    by (metis  $\langle M1 \in B \rangle \langle M2 \in B \rangle \text{in-diffD subsetD transp-onD}$ )
  thus ?thesis
  unfolding multpHO-iff-set-mset-lessHO-set-mset by simp
qed
qed

```

```

lemma asym-p-multpHO:
  assumes asym R and transp R
  shows asym (multpHO R)
  using assms asym-on-multpHO [of UNIV, simplified] by metis

```

Irreflexivity lemma *irreflp-on-multp_{HO}* [*simp*]: *irreflp-on* *B* (*multp_{HO}* *R*)
 by (*simp add: irreflp-onI multp_{HO}-def*)

```

Transitivity lemma transp-on-multpHO:
  assumes asym-on A R and transp-on A R and B-sub-A:  $\bigwedge M. M \in B \implies$ 
set-mset M  $\subseteq$  A
  shows transp-on B (multpHO R)
proof (rule transp-onI)
  from assms have asym-on B (multpHO R)
  using asym-on-multpHO by metis

```

```

fix M1 M2 M3
assume hyps: M1  $\in$  B M2  $\in$  B M3  $\in$  B multpHO R M1 M2 multpHO R M2 M3

from assms have
  [intro]: asym-on (set-mset M1  $\cup$  set-mset M2) R transp-on (set-mset M1  $\cup$ 
set-mset M2) R
  using  $\langle M1 \in B \rangle \langle M2 \in B \rangle$ 
  by (simp-all add: asym-on-subset transp-on-subset)

```

```

from assms have transp-on (set-mset M1) R
  by (meson transp-on-subset hyps(1))

```

```

from  $\langle \text{multp}_{HO} R M1 M2 \rangle$  have
  M1  $\neq$  M2 and
   $\forall y. \text{count } M2 y < \text{count } M1 y \longrightarrow (\exists x. R y x \wedge \text{count } M1 x < \text{count } M2 x)$ 
  unfolding multpHO-def by simp-all

```

```

from  $\langle \text{multp}_{HO} R M2 M3 \rangle$  have
  M2  $\neq$  M3 and
   $\forall y. \text{count } M3 y < \text{count } M2 y \longrightarrow (\exists x. R y x \wedge \text{count } M2 x < \text{count } M3 x)$ 
  unfolding multpHO-def by simp-all

```

```

show multpHO R M1 M3
proof (rule ccontr)
  let ?P =  $\lambda x. \text{count } M3 x < \text{count } M1 x \wedge (\forall y. R x y \longrightarrow \text{count } M1 y \geq \text{count}$ 

```

$M3\ y)$

```

assume  $\neg \text{multp}_{HO}\ R\ M1\ M3$ 
hence  $M1 = M3 \vee (\exists x. ?P\ x)$ 
  unfolding  $\text{multp}_{HO}\text{-def}$  by force
thus False
proof (elim disjE)
  assume  $M1 = M3$ 
  thus False
    using  $\langle \text{asympt-on}\ B\ (\text{multp}_{HO}\ R) \rangle [THEN\ \text{asympt-onD}]$ 
    using  $\langle M2 \in B \rangle \langle M3 \in B \rangle \langle \text{multp}_{HO}\ R\ M1\ M2 \rangle \langle \text{multp}_{HO}\ R\ M2\ M3 \rangle$ 
    by metis
next
  assume  $\exists x. ?P\ x$ 
  hence  $\exists x \in \# M1 + M2. ?P\ x$ 
    by (auto simp: count-inI)
  have  $\exists y \in \# M1 + M2. ?P\ y \wedge (\forall z \in \# M1 + M2. R\ y\ z \longrightarrow \neg ?P\ z)$ 
  proof (rule Finite-Set.bex-max-element-with-property)
    show  $\exists x \in \# M1 + M2. ?P\ x$ 
      using  $\langle \exists x. ?P\ x \rangle$ 
      by (auto simp: count-inI)
  qed auto
then obtain  $x$  where
   $x \in \# M1 + M2$  and
   $\text{count}\ M3\ x < \text{count}\ M1\ x$  and
   $\forall y. R\ x\ y \longrightarrow \text{count}\ M1\ y \geq \text{count}\ M3\ y$  and
   $\forall y \in \# M1 + M2. R\ x\ y \longrightarrow \text{count}\ M3\ y < \text{count}\ M1\ y \longrightarrow (\exists z. R\ y\ z \wedge$ 
 $\text{count}\ M1\ z < \text{count}\ M3\ z)$ 
  by force

let  $?Q = \lambda x'. R^{==}\ x\ x' \wedge \text{count}\ M3\ x' < \text{count}\ M2\ x'$ 
show False
proof (cases  $\exists x'. ?Q\ x'$ )
  case True
    have  $\exists y \in \# M1 + M2. ?Q\ y \wedge (\forall z \in \# M1 + M2. R\ y\ z \longrightarrow \neg ?Q\ z)$ 
    proof (rule Finite-Set.bex-max-element-with-property)
      show  $\exists x \in \# M1 + M2. ?Q\ x$ 
        using  $\langle \exists x. ?Q\ x \rangle$ 
        by (auto simp: count-inI)
    qed auto
    then obtain  $x'$  where
     $x' \in \# M1 + M2$  and
     $R^{==}\ x\ x'$  and
     $\text{count}\ M3\ x' < \text{count}\ M2\ x'$  and
     $\text{maximality-}x': \forall z \in \# M1 + M2. R\ x'\ z \longrightarrow \neg (R^{==}\ x\ z) \vee \text{count}\ M3\ z$ 
 $\geq \text{count}\ M2\ z$ 
    by (auto simp: linorder-not-less)
    with  $\langle \text{multp}_{HO}\ R\ M2\ M3 \rangle$  obtain  $y'$  where
     $R\ x'\ y'$  and  $\text{count}\ M2\ y' < \text{count}\ M3\ y'$ 

```


unfolding *multp_{HO}-def* **by** *auto*
hence *count M2 y' < count M1 y'*
by (*smt (verit) <R⁼ x x'> <∀ y. R x y → count M3 y ≤ count M1 y*
<count M3 x < count M1 x> <count M3 x' < count M2 x'> assms(2))
count-inI
dual-order.strict-trans1 hyps(1) hyps(2) hyps(3) less-nat-zero-code
B-sub-A subsetD
sup2E transp-onD)
with *<multp_{HO} R M1 M2>* **obtain** *y''* **where**
R y' y'' and count M1 y'' < count M2 y''
unfolding *multp_{HO}-def* **by** *auto*
hence *count M3 y'' < count M2 y''*
by (*smt (verit, del-insts) <R x' y'> <R⁼ x x'> <∀ y. R x y → count M3 y*
≤ count M1 y>
<count M2 y' < count M3 y'> <count M3 x < count M1 x> <count M3
x' < count M2 x'>
assms(2) count-greater-zero-iff dual-order.strict-trans1 hyps(1) hyps(2)
hyps(3)
less-nat-zero-code linorder-not-less B-sub-A subset-iff sup2E transp-onD)
moreover have *count M2 y'' ≤ count M3 y''*
proof –
have *y'' ∈# M1 + M2*
by (*metis <count M1 y'' < count M2 y''> count-inI not-less-iff-gr-or-eq*
union-iff)
moreover have *R x' y''*
by (*metis <R x' y'> <R y' y''> <count M2 y' < count M1 y'>*
<transp-on (set-mset M1 ∪ set-mset M2) R> <x' ∈# M1 + M2>
calculation count-inI
nat-neq-iff set-mset-union transp-onD union-iff)
moreover have *R⁼ x y''*
using *<R⁼ x x'>*
by (*metis (mono-tags, opaque-lifting) <transp-on (set-mset M1 ∪ set-mset*
M2) R>
<x ∈# M1 + M2> <x' ∈# M1 + M2> calculation(1) calculation(2)
set-mset-union sup2I1
transp-onD transp-on-reflcp)
ultimately show *?thesis*
using *maximality-x'[rule-format, of y']* **by** *metis*
qed
ultimately show *?thesis*
by *linarith*
next
case *False*
hence $\bigwedge x'. R^= x x' \implies \text{count } M2 x' \leq \text{count } M3 x'$

```

      by auto
    hence count M2 x ≤ count M3 x
      by simp
    hence count M2 x < count M1 x
      using ⟨count M3 x < count M1 x⟩ by linarith
    with ⟨multpHO R M1 M2⟩ obtain y where
      R x y and count M1 y < count M2 y
      unfolding multpHO-def by auto
    hence count M3 y < count M2 y
      using ⟨∀ y. R x y ⟶ count M3 y ≤ count M1 y⟩ dual-order.strict-trans2
  by metis
    then show ?thesis
      using False ⟨R x y⟩ by auto
  qed
qed
qed
qed

```

```

lemma transp-multpHO:
  assumes asymp R and transp R
  shows transp (multpHO R)
  using assms transp-on-multpHO[of UNIV, simplified] by metis

```

Totality lemma totalp-on-multp_{DM}:

```

totalp-on A R ⟹ (⋀ M. M ∈ B ⟹ set-mset M ⊆ A) ⟹ totalp-on B (multpDM R)
  by (smt (verit, ccfv-SIG) count-inI in-mono multpHO-def multpHO-imp-multpDM
not-less-iff-gr-or-eq
totalp-onD totalp-onI)

```

```

lemma totalp-multpDM: totalp R ⟹ totalp (multpDM R)
  by (rule totalp-on-multpDM[of UNIV R UNIV, simplified])

```

```

lemma totalp-on-multpHO:
totalp-on A R ⟹ (⋀ M. M ∈ B ⟹ set-mset M ⊆ A) ⟹ totalp-on B (multpHO R)
  by (smt (verit, ccfv-SIG) count-inI in-mono multpHO-def not-less-iff-gr-or-eq
totalp-onD
totalp-onI)

```

```

lemma totalp-multpHO: totalp R ⟹ totalp (multpHO R)
  by (rule totalp-on-multpHO[of UNIV R UNIV, simplified])

```

Type Classes context preorder
begin

```

lemma order-mult: class.order
  (λM N. (M, N) ∈ mult {(x, y). x < y} ∨ M = N)
  (λM N. (M, N) ∈ mult {(x, y). x < y})

```

```

(is class.order ?le ?less)
proof -
  have irreft:  $\bigwedge M :: 'a \text{ multiset}. \neg ?less M M$ 
  proof
    fix M :: 'a multiset
    have trans  $\{(x'::'a, x). x' < x\}$ 
      by (rule transI) (blast intro: less-trans)
    moreover
    assume  $(M, M) \in \text{mult } \{(x, y). x < y\}$ 
    ultimately have  $\exists I J K. M = I + J \wedge M = I + K$ 
       $\wedge J \neq \{\#\} \wedge (\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J. (k, j) \in \{(x, y). x < y\})$ 
      by (rule mult-implies-one-step)
    then obtain I J K where  $M = I + J$  and  $M = I + K$ 
      and  $J \neq \{\#\}$  and  $(\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } J. (k, j) \in \{(x, y). x < y\})$ 
    by blast
    then have aux1:  $K \neq \{\#\}$  and aux2:  $\forall k \in \text{set-mset } K. \exists j \in \text{set-mset } K. k < j$ 
    by auto
    have finite (set-mset K) by simp
    moreover note aux2
    ultimately have set-mset K =  $\{\}$ 
      by (induct rule: finite-induct)
    (simp, metis (mono-tags) insert-absorb insert-iff insert-not-empty less-irreft
    less-trans)
    with aux1 show False by simp
  qed
  have trans:  $\bigwedge K M N :: 'a \text{ multiset}. ?less K M \implies ?less M N \implies ?less K N$ 
    unfolding mult-def by (blast intro: trancl-trans)
  show class.order ?le ?less
    by standard (auto simp add: less-eq-multiset-def irreft dest: trans)
qed

```

The Dershowitz–Manna ordering:

definition $less-multiset_{DM}$ **where**

```

less-multisetDM M N  $\longleftrightarrow$ 
   $(\exists X Y. X \neq \{\#\} \wedge X \subseteq\# N \wedge M = (N - X) + Y \wedge (\forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge k < a)))$ 

```

The Huet–Oppen ordering:

definition $less-multiset_{HO}$ **where**

```

less-multisetHO M N  $\longleftrightarrow M \neq N \wedge (\forall y. \text{count } N y < \text{count } M y \longrightarrow (\exists x. y < x \wedge \text{count } M x < \text{count } N x))$ 

```

lemma $mult-imp-less-multiset_{HO}$:

```

 $(M, N) \in \text{mult } \{(x, y). x < y\} \implies less-multiset_{HO} M N$ 
unfolding multp-def[of (<), symmetric]
using multp-imp-multpHO[of (<)]
by (simp add: less-multisetHO-def multpHO-def)

```

lemma $less-multiset_{DM}$ -*imp-mult*:

$less_multiset_{DM} M N \implies (M, N) \in mult \{(x, y). x < y\}$
unfolding $mult_def[of (<), symmetric]$
by (rule $mult_{DM}\text{-imp-mult}[of (<) M N]$) (simp add: $less_multiset_{DM}\text{-def } mult_{DM}\text{-def}$)

lemma $less_multiset_{HO}\text{-imp-less-multiset}_{DM}$: $less_multiset_{HO} M N \implies less_multiset_{DM} M N$
unfolding $less_multiset_{DM}\text{-def } less_multiset_{HO}\text{-def}$
unfolding $mult_{DM}\text{-def}[symmetric] mult_{HO}\text{-def}[symmetric]$
by (rule $mult_{HO}\text{-imp-mult}_{DM}$)

lemma $mult_less_multiset_{DM}$: $(M, N) \in mult \{(x, y). x < y\} \longleftrightarrow less_multiset_{DM} M N$
unfolding $mult_def[of (<), symmetric]$
using $mult_eq_mult_{DM}[of (<), simplified]$
by (simp add: $mult_{DM}\text{-def } less_multiset_{DM}\text{-def}$)

lemma $mult_less_multiset_{HO}$: $(M, N) \in mult \{(x, y). x < y\} \longleftrightarrow less_multiset_{HO} M N$
unfolding $mult_def[of (<), symmetric]$
using $mult_eq_mult_{HO}[of (<), simplified]$
by (simp add: $mult_{HO}\text{-def } less_multiset_{HO}\text{-def}$)

lemmas $mult_{DM} = mult_less_multiset_{DM}[unfolded less_multiset_{DM}\text{-def}]$
lemmas $mult_{HO} = mult_less_multiset_{HO}[unfolded less_multiset_{HO}\text{-def}]$

end

lemma $less_multiset_less_multiset_{HO}$: $M < N \longleftrightarrow less_multiset_{HO} M N$
unfolding $less_multiset_def mult_def mult_{HO} less_multiset_{HO}\text{-def} ..$

lemma $less_multiset_{DM}$:
 $M < N \longleftrightarrow (\exists X Y. X \neq \{\#\} \wedge X \subseteq\# N \wedge M = N - X + Y \wedge (\forall k. k \in\# Y \longrightarrow (\exists a. a \in\# X \wedge k < a)))$
by (rule $mult_{DM}[folded mult_def less_multiset_def]$)

lemma $less_multiset_{HO}$:
 $M < N \longleftrightarrow M \neq N \wedge (\forall y. count N y < count M y \longrightarrow (\exists x>y. count M x < count N x))$
by (rule $mult_{HO}[folded mult_def less_multiset_def]$)

lemma $subset_eq_imp_le_multiset$:
shows $M \subseteq\# N \implies M \leq N$
unfolding $less_eq_multiset_def less_multiset_{HO}$
by (simp add: $less_le_not_le subseteq_mset_def$)

lemma $le_multiset_right_total$: $M < add_mset x M$
unfolding $less_eq_multiset_def less_multiset_{HO}$ **by** simp

lemma *less-eq-multiset-empty-left[simp]*: $\{\#\} \leq M$
by (*simp add: subset-eq-imp-le-multiset*)

lemma *ex-gt-imp-less-multiset*: $(\exists y. y \in\# N \wedge (\forall x. x \in\# M \longrightarrow x < y)) \Longrightarrow M < N$
unfolding *less-multiset_{HO}*
by (*metis count-eq-zero-iff count-greater-zero-iff less-le-not-le*)

lemma *less-eq-multiset-empty-right[simp]*: $M \neq \{\#\} \Longrightarrow \neg M \leq \{\#\}$
by (*metis less-eq-multiset-empty-left antisym*)

lemma *le-multiset-empty-left[simp]*: $M \neq \{\#\} \Longrightarrow \{\#\} < M$
by (*simp add: less-multiset_{HO}*)

lemma *le-multiset-empty-right[simp]*: $\neg M < \{\#\}$
using *subset-mset.le-zero-eq less-multiset-def multp-def less-multiset_{DM}* **by** *blast*

lemma *union-le-diff-plus*: $P \subseteq\# M \Longrightarrow N < P \Longrightarrow M - P + N < M$
by (*drule subset-mset.diff-add[symmetric]*) (*metis union-le-mono2*)

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-monoid-add-imp-le*
begin

lemma *less-eq-multiset_{HO}*:
 $M \leq N \longleftrightarrow (\forall y. \text{count } N \ y < \text{count } M \ y \longrightarrow (\exists x. y < x \wedge \text{count } M \ x < \text{count } N \ x))$
by (*auto simp: less-eq-multiset-def less-multiset_{HO}*)

instance **by** *standard* (*auto simp: less-eq-multiset_{HO}*)

lemma
fixes $M \ N :: 'a \text{ multiset}$
shows *less-eq-multiset-plus-left*: $N \leq (M + N)$
and *less-eq-multiset-plus-right*: $M \leq (M + N)$
by *simp-all*

lemma
fixes $M \ N :: 'a \text{ multiset}$
shows *le-multiset-plus-left-nonempty*: $M \neq \{\#\} \Longrightarrow N < M + N$
and *le-multiset-plus-right-nonempty*: $N \neq \{\#\} \Longrightarrow M < M + N$
by *simp-all*

end

lemma *all-lt-Max-imp-lt-mset*: $N \neq \{\#\} \Longrightarrow (\forall a \in\# M. a < \text{Max } (\text{set-mset } N)) \Longrightarrow M < N$

by (*meson* *Max-in*[*OF* *finite-set-mset*] *ex-gt-imp-less-multiset* *set-mset-eq-empty-iff*)

lemma *lt-imp-ex-count-lt*: $M < N \implies \exists y. \text{count } M \ y < \text{count } N \ y$
by (*meson* *less-eq-multiset_{HO}* *less-le-not-le*)

lemma *subset-imp-less-mset*: $A \subseteq\# B \implies A < B$
by (*simp* *add: order.not-eq-order-implies-strict* *subset-eq-imp-le-multiset*)

lemma *image-mset-strict-mono*:

assumes *mono-f*: $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f \ x < f \ y$
and *less*: $M < N$

shows *image-mset* $f \ M < \text{image-mset } f \ N$

proof –

obtain $Y \ X$ **where**

y-nemp: $Y \neq \{\#\}$ **and** *y-sub-N*: $Y \subseteq\# N$ **and** *M-eq*: $M = N - Y + X$ **and**

ex-y: $\forall x. x \in\# X \longrightarrow (\exists y. y \in\# Y \wedge x < y)$

using *less[unfolded less-multiset_{DM}]* **by** *blast*

have *x-sub-M*: $X \subseteq\# M$

using *M-eq* **by** *simp*

let $?fY = \text{image-mset } f \ Y$

let $?fX = \text{image-mset } f \ X$

show *?thesis*

unfolding *less-multiset_{DM}*

proof (*intro exI conjI*)

show *image-mset* $f \ M = \text{image-mset } f \ N - ?fY + ?fX$

using *M-eq[THEN arg-cong, of image-mset f]* *y-sub-N*

by (*metis image-mset-Diff image-mset-union*)

next

obtain y **where** $y: \forall x. x \in\# X \longrightarrow y \ x \in\# Y \wedge x < y \ x$

using *ex-y* **by** *metis*

show $\forall fx. fx \in\# ?fX \longrightarrow (\exists fy. fy \in\# ?fY \wedge fx < fy)$

proof (*intro allI impI*)

fix fx

assume $fx \in\# ?fX$

then obtain x **where** $fx = f \ x$ **and** *x-in*: $x \in\# X$

by *auto*

hence *y-in*: $y \ x \in\# Y$ **and** *y-gt*: $x < y \ x$

using *y[rule-format, OF x-in]* **by** *blast+*

hence $f \ (y \ x) \in\# ?fY \wedge f \ x < f \ (y \ x)$

using *mono-f y-sub-N x-sub-M x-in*

by (*metis image-eqI in-image-mset mset-subset-eqD*)

thus $\exists fy. fy \in\# ?fY \wedge fx < fy$

unfolding fx **by** *auto*

qed

qed (*auto simp: y-nemp y-sub-N image-mset-subseteq-mono*)

qed

lemma *image-mset-mono*:

assumes *mono-f*: $\forall x \in \text{set-mset } M. \forall y \in \text{set-mset } N. x < y \longrightarrow f x < f y$

and *less*: $M \leq N$

shows $\text{image-mset } f M \leq \text{image-mset } f N$

by (*metis eq-iff image-mset-strict-mono less less-imp-le mono-f order.not-eq-order-implies-strict*)

lemma *mset-lt-single-right-iff[simp]*: $M < \{\#y\# \} \longleftrightarrow (\forall x \in \# M. x < y)$ **for** y

:: *'a::linorder*

proof (*rule iffI*)

assume $M\text{-lt-}y$: $M < \{\#y\# \}$

show $\forall x \in \# M. x < y$

proof

fix x

assume $x\text{-in}$: $x \in \# M$

hence M : $M - \{\#x\# \} + \{\#x\# \} = M$

by (*meson insert-DiffM2*)

hence $\neg \{\#x\# \} < \{\#y\# \} \implies x < y$

using $x\text{-in } M\text{-lt-}y$

by (*metis diff-single-eq-union le-multiset-empty-left less-add-same-cancel2*

mset-le-trans)

also have $\neg \{\#y\# \} < M$

using $M\text{-lt-}y$ *mset-le-not-sym* **by** *blast*

ultimately show $x < y$

by (*metis (no-types) Max-ge all-lt-Max-imp-lt-mset empty-iff finite-set-mset*

insertE

less-le-trans linorder-less-linear mset-le-not-sym set-mset-add-mset-insert

set-mset-eq-empty-iff x-in)

qed

next

assume $y\text{-max}$: $\forall x \in \# M. x < y$

show $M < \{\#y\# \}$

by (*rule all-lt-Max-imp-lt-mset*) (*auto intro!*: $y\text{-max}$)

qed

lemma *mset-le-single-right-iff[simp]*:

$M \leq \{\#y\# \} \longleftrightarrow M = \{\#y\# \} \vee (\forall x \in \# M. x < y)$ **for** $y :: 'a::linorder$

by (*meson less-eq-multiset-def mset-lt-single-right-iff*)

68.1.5 Simplifications

lemma *multp_{HO}-repeat-mset-repeat-mset[simp]*:

assumes $n \neq 0$

shows $\text{multp}_{HO} R (\text{repeat-mset } n A) (\text{repeat-mset } n B) \longleftrightarrow \text{multp}_{HO} R A B$

proof (*rule iffI*)

assume hyp : $\text{multp}_{HO} R (\text{repeat-mset } n A) (\text{repeat-mset } n B)$

hence

1: $\text{repeat-mset } n A \neq \text{repeat-mset } n B$ **and**

$2: \forall y. n * \text{count } B \ y < n * \text{count } A \ y \longrightarrow (\exists x. R \ y \ x \wedge n * \text{count } A \ x < n * \text{count } B \ x)$
 by (simp-all add: multp_{HO}-def)

from 1 $\langle n \neq 0 \rangle$ have $A \neq B$
 by auto

moreover from 2 $\langle n \neq 0 \rangle$ have $\forall y. \text{count } B \ y < \text{count } A \ y \longrightarrow (\exists x. R \ y \ x \wedge \text{count } A \ x < \text{count } B \ x)$
 by auto

ultimately show multp_{HO} $R \ A \ B$
 by (simp add: multp_{HO}-def)

next

assume multp_{HO} $R \ A \ B$
 hence 1: $A \neq B$ and 2: $\forall y. \text{count } B \ y < \text{count } A \ y \longrightarrow (\exists x. R \ y \ x \wedge \text{count } A \ x < \text{count } B \ x)$
 by (simp-all add: multp_{HO}-def)

from 1 have repeat-mset $n \ A \neq \text{repeat-mset } n \ B$
 by (simp add: assms repeat-mset-cancel1)

moreover from 2 have $\forall y. n * \text{count } B \ y < n * \text{count } A \ y \longrightarrow (\exists x. R \ y \ x \wedge n * \text{count } A \ x < n * \text{count } B \ x)$
 by auto

ultimately show multp_{HO} $R \ (\text{repeat-mset } n \ A) \ (\text{repeat-mset } n \ B)$
 by (simp add: multp_{HO}-def)

qed

lemma multp_{HO}-double-double[simp]: multp_{HO} $R \ (A + A) \ (B + B) \longleftrightarrow \text{multp}_{HO} \ R \ A \ B$
 using multp_{HO}-repeat-mset-repeat-mset[of 2]
 by (simp add: numeral-Bit0)

68.2 Simprocs

lemma mset-le-add-iff1:

$j \leq (i::\text{nat}) \implies (\text{repeat-mset } i \ u + m \leq \text{repeat-mset } j \ u + n) = (\text{repeat-mset } (i-j) \ u + m \leq n)$

proof –

assume $j \leq i$

then have $j + (i - j) = i$

using le-add-diff-inverse by blast

then show ?thesis

by (metis (no-types) add-le-cancel-left left-add-mult-distrib-mset)

qed

lemma mset-le-add-iff2:

$i \leq (j::\text{nat}) \implies (\text{repeat-mset } i \ u + m \leq \text{repeat-mset } j \ u + n) = (m \leq \text{repeat-mset } (j-i) \ u + n)$

proof –

assume $i \leq j$

then have $i + (j - i) = j$

using *le-add-diff-inverse* **by** *blast*

then show *?thesis*

by (*metis* (*no-types*) *add-le-cancel-left* *left-add-mult-distrib-mset*)

qed

simproc-setup *msetless-cancel*

$((l::'a::\text{preorder multiset}) + m < n \mid (l::'a \text{ multiset}) < m + n \mid$

$\text{add-mset } a \ m < n \mid m < \text{add-mset } a \ n \mid$

$\text{replicate-mset } p \ a < n \mid m < \text{replicate-mset } p \ a \mid$

$\text{repeat-mset } p \ m < n \mid m < \text{repeat-mset } p \ n) =$

$\langle K \text{ Cancel-Simprocs.less-cancel} \rangle$

simproc-setup *msetle-cancel*

$((l::'a::\text{preorder multiset}) + m \leq n \mid (l::'a \text{ multiset}) \leq m + n \mid$

$\text{add-mset } a \ m \leq n \mid m \leq \text{add-mset } a \ n \mid$

$\text{replicate-mset } p \ a \leq n \mid m \leq \text{replicate-mset } p \ a \mid$

$\text{repeat-mset } p \ m \leq n \mid m \leq \text{repeat-mset } p \ n) =$

$\langle K \text{ Cancel-Simprocs.less-eq-cancel} \rangle$

68.3 Additional facts and instantiations

lemma *ex-gt-count-imp-le-multiset*:

$(\forall y :: 'a :: \text{order}. y \in \# M + N \longrightarrow y \leq x) \implies \text{count } M \ x < \text{count } N \ x \implies M < N$

unfolding *less-multiset_{HO}*

by (*metis* *count-greater-zero-iff* *le-imp-less-or-eq* *less-imp-not-less* *not-gr-zero* *union-iff*)

lemma *mset-lt-single-iff*[*iff*]: $\{\#x\# \} < \{\#y\# \} \longleftrightarrow x < y$

unfolding *less-multiset_{HO}* **by** *simp*

lemma *mset-le-single-iff*[*iff*]: $\{\#x\# \} \leq \{\#y\# \} \longleftrightarrow x \leq y$ **for** $x \ y :: 'a :: \text{order}$

unfolding *less-eq-multiset_{HO}* **by** *force*

instance *multiset* :: (*linorder*) *linordered-cancel-ab-semigroup-add*

by *standard* (*metis* *less-eq-multiset_{HO}* *not-less-iff-gr-or-eq*)

lemma *less-eq-multiset-total*: $\neg M \leq N \implies N \leq M$ **for** $M \ N :: 'a :: \text{linorder multiset}$

by *simp*

instantiation *multiset* :: (*wellorder*) *wellorder*

begin

lemma *wf-less-multiset*: *wf* $\{(M :: 'a \text{ multiset}, N). M < N\}$

```

unfolding less-multiset-def multp-def by (auto intro: wf-mult wf)

instance
proof intro-classes
  fix P :: 'a multiset  $\Rightarrow$  bool and a :: 'a multiset
  have wfp ((<) :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool)
    using wfp-on-less .
  hence wfp ((<) :: 'a multiset  $\Rightarrow$  'a multiset  $\Rightarrow$  bool)
    unfolding less-multiset-def by (rule wfp-multp)
  thus ( $\bigwedge x. (\bigwedge y. y < x \Rightarrow P y) \Rightarrow P x \Rightarrow P a$ )
    unfolding wfp-on-def[of UNIV, simplified] by metis
qed

end

instantiation multiset :: (preorder) order-bot
begin

definition bot-multiset :: 'a multiset where bot-multiset = {#}

instance by standard (simp add: bot-multiset-def)

end

instance multiset :: (preorder) no-top
proof standard
  fix x :: 'a multiset
  obtain a :: 'a where True by simp
  have  $x < x + (x + \{ \#a\# \})$ 
    by simp
  then show  $\exists y. x < y$ 
    by blast
qed

instance multiset :: (preorder) ordered-cancel-comm-monoid-add
  by standard

instantiation multiset :: (linorder) distrib-lattice
begin

definition inf-multiset :: 'a multiset  $\Rightarrow$  'a multiset  $\Rightarrow$  'a multiset where
  inf-multiset A B = (if A < B then A else B)

definition sup-multiset :: 'a multiset  $\Rightarrow$  'a multiset  $\Rightarrow$  'a multiset where
  sup-multiset A B = (if B > A then B else A)

instance
  by intro-classes (auto simp: inf-multiset-def sup-multiset-def)

```

end

lemma *add-mset-lt-left-lt*: $a < b \implies \text{add-mset } a \ A < \text{add-mset } b \ A$
by *fastforce*

lemma *add-mset-le-left-le*: $a \leq b \implies \text{add-mset } a \ A \leq \text{add-mset } b \ A$ **for** $a :: 'a :: \text{linorder}$
by *fastforce*

lemma *add-mset-lt-right-lt*: $A < B \implies \text{add-mset } a \ A < \text{add-mset } a \ B$
by *fastforce*

lemma *add-mset-le-right-le*: $A \leq B \implies \text{add-mset } a \ A \leq \text{add-mset } a \ B$
by *fastforce*

lemma *add-mset-lt-lt-lt*:
assumes $a\text{-lt-}b$: $a < b$ **and** $A\text{-le-}B$: $A < B$
shows $\text{add-mset } a \ A < \text{add-mset } b \ B$
by (*rule less-trans* [*OF* *add-mset-lt-left-lt* [*OF* $a\text{-lt-}b$] *add-mset-lt-right-lt* [*OF* $A\text{-le-}B$]])

lemma *add-mset-lt-lt-le*: $a < b \implies A \leq B \implies \text{add-mset } a \ A < \text{add-mset } b \ B$
using *add-mset-lt-lt-lt le-neq-trans* **by** *fastforce*

lemma *add-mset-lt-le-lt*: $a \leq b \implies A < B \implies \text{add-mset } a \ A < \text{add-mset } b \ B$ **for** $a :: 'a :: \text{linorder}$
using *add-mset-lt-lt-lt* **by** (*metis add-mset-lt-right-lt le-less*)

lemma *add-mset-le-le-le*:
fixes $a :: 'a :: \text{linorder}$
assumes $a\text{-le-}b$: $a \leq b$ **and** $A\text{-le-}B$: $A \leq B$
shows $\text{add-mset } a \ A \leq \text{add-mset } b \ B$
by (*rule order.trans* [*OF* *add-mset-le-left-le* [*OF* $a\text{-le-}b$] *add-mset-le-right-le* [*OF* $A\text{-le-}B$]])

lemma *Max-lt-imp-lt-mset*:
assumes $n\text{-nemp}$: $N \neq \{\#\}$ **and** max : $\text{Max-mset } M < \text{Max-mset } N$ (**is** $?max\text{-}M < ?max\text{-}N$)
shows $M < N$
proof (*cases* $M = \{\#\}$)
case $m\text{-nemp}$: *False*

have $\text{max-}n\text{-in-}n$: $?max\text{-}N \in \# \ N$
using $n\text{-nemp}$ **by** *simp*
have $\text{max-}n\text{-nin-}m$: $?max\text{-}N \notin \# \ M$
using $\text{max } \text{Max-ge } leD$ **by** *auto*

have $M \neq N$
using max **by** *auto*
moreover

```

have  $\exists x > y. \text{count } M \ x < \text{count } N \ x$  if  $\text{count } N \ y < \text{count } M \ y$  for  $y$ 
proof -
  from that have  $y \in \# \ M$ 
  by (simp add: count-inI)
  then have  $?max\text{-}M \geq y$ 
  by simp
  then have  $?max\text{-}N > y$ 
  using max by auto
  then show ?thesis
  using max-n-nin-m max-n-in-n count-inI by force
qed
ultimately show ?thesis
  unfolding less-multisetHO by blast
qed (auto simp: n-nemp)

end

```

69 Fixed Length Lists

```

theory NList
imports Main
begin

```

```

definition nlists :: nat  $\Rightarrow$  'a set  $\Rightarrow$  'a list set
  where nlists  $n \ A = \{xs. \text{size } xs = n \wedge \text{set } xs \subseteq A\}$ 

```

```

lemma nlistsI:  $\llbracket \text{size } xs = n; \text{set } xs \subseteq A \rrbracket \Longrightarrow xs \in \text{nlists } n \ A$ 
  by (simp add: nlists-def)

```

These [simp] attributes are double-edged. Many proofs in Jinja rely on it but they can degrade performance.

```

lemma nlistsE-length [simp]:  $xs \in \text{nlists } n \ A \Longrightarrow \text{size } xs = n$ 
  by (simp add: nlists-def)

```

```

lemma in-nlists-UNIV:  $xs \in \text{nlists } k \ \text{UNIV} \longleftrightarrow \text{length } xs = k$ 
  unfolding nlists-def by(auto)

```

```

lemma less-lengthI:  $\llbracket xs \in \text{nlists } n \ A; p < n \rrbracket \Longrightarrow p < \text{size } xs$ 
  by (simp)

```

```

lemma nlistsE-set[simp]:  $xs \in \text{nlists } n \ A \Longrightarrow \text{set } xs \subseteq A$ 
  unfolding nlists-def by (simp)

```

```

lemma nlists-mono:
  assumes  $A \subseteq B$  shows  $\text{nlists } n \ A \subseteq \text{nlists } n \ B$ 
proof
  fix xs assume  $xs \in \text{nlists } n \ A$ 
  then obtain size:  $\text{size } xs = n$  and inA:  $\text{set } xs \subseteq A$  by (simp)
  with assms have  $\text{set } xs \subseteq B$  by simp

```

with *size* **show** $xs \in nlists\ n\ B$ **by**(*clarsimp intro!:* *nlistsI*)
qed

lemma *nlists-singleton*: $nlists\ n\ \{a\} = \{replicate\ n\ a\}$
unfolding *nlists-def* **by**(*auto simp: replicate-length-same dest!: subset-singletonD*)

lemma *nlists-n-0* [*simp*]: $nlists\ 0\ A = \{\}\}$
unfolding *nlists-def* **by** (*auto*)

lemma *in-nlists-Suc-iff*: $(xs \in nlists\ (Suc\ n)\ A) = (\exists y \in A. \exists ys \in nlists\ n\ A. xs = y \# ys)$
unfolding *nlists-def* **by** (*cases xs*) *auto*

lemma *Cons-in-nlists-Suc* [*iff*]: $(x \# xs \in nlists\ (Suc\ n)\ A) \longleftrightarrow (x \in A \wedge xs \in nlists\ n\ A)$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-Suc*: $nlists\ (Suc\ n)\ A = (\bigcup a \in A. (\#)\ a\ ' nlists\ n\ A)$
by(*auto simp: set-eq-iff image-iff in-nlists-Suc-iff*)

lemma *nlists-not-empty*: $A \neq \{\} \implies \exists xs. xs \in nlists\ n\ A$
by (*induct n*) (*auto simp: in-nlists-Suc-iff*)

lemma *nlistsE-nth-in*: $\llbracket xs \in nlists\ n\ A; i < n \rrbracket \implies xs!i \in A$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-Cons-Suc* [*elim!*]:
 $l \# xs \in nlists\ n\ A \implies (\bigwedge n'. n = Suc\ n' \implies l \in A \implies xs \in nlists\ n'\ A \implies P) \implies P$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-appendE* [*elim!*]:
 $a @ b \in nlists\ n\ A \implies (\bigwedge n1\ n2. n = n1 + n2 \implies a \in nlists\ n1\ A \implies b \in nlists\ n2\ A \implies P) \implies P$

proof –

have $\bigwedge n. a @ b \in nlists\ n\ A \implies \exists n1\ n2. n = n1 + n2 \wedge a \in nlists\ n1\ A \wedge b \in nlists\ n2\ A$

(**is** $\bigwedge n. ?list\ a\ n \implies \exists n1\ n2. ?P\ a\ n\ n1\ n2$)

proof (*induct a*)

fix *n* **assume** *?list* $\llbracket n$

hence *?P* $\llbracket n\ 0\ n$ **by** *simp*

thus $\exists n1\ n2. ?P\ \llbracket n\ n1\ n2$ **by** *fast*

next

fix *n l ls*

assume *?list* $(l \# ls)\ n$

then obtain *n'* **where** $n = Suc\ n'\ l \in A$ **and** $n': ls @ b \in nlists\ n'\ A$ **by** *fastforce*

assume $\bigwedge n. ls @ b \in nlists\ n\ A \implies \exists n1\ n2. n = n1 + n2 \wedge ls \in nlists\ n1\ A \wedge b \in nlists\ n2\ A$

from this and n' have $\exists n1\ n2. n' = n1 + n2 \wedge ls \in nlists\ n1\ A \wedge b \in nlists\ n2\ A$.
then obtain $n1\ n2$ where $n' = n1 + n2$ $ls \in nlists\ n1\ A\ b \in nlists\ n2\ A$ **by**
fast
with n have $?P\ (l\#ls)\ n\ (n1+1)\ n2$ **by** *simp*
thus $\exists n1\ n2. ?P\ (l\#ls)\ n\ n1\ n2$ **by** *fastforce*
qed
moreover assume $a@b \in nlists\ n\ A \wedge n1\ n2. n=n1+n2 \implies a \in nlists\ n1\ A$
 $\implies b \in nlists\ n2\ A \implies P$
ultimately show *?thesis* **by** *blast*
qed

lemma *nlists-update-in-list [simp, intro!]*:
 $\llbracket xs \in nlists\ n\ A; x \in A \rrbracket \implies xs[i := x] \in nlists\ n\ A$
by (*metis length-list-update nlistsE-length nlistsE-set nlistsI set-update-subsetI*)

lemma *nlists-appendI [intro?]*:
 $\llbracket a \in nlists\ n\ A; b \in nlists\ m\ A \rrbracket \implies a @ b \in nlists\ (n+m)\ A$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-append*:
 $xs @ ys \in nlists\ k\ A \longleftrightarrow$
 $k = length(xs @ ys) \wedge xs \in nlists\ (length\ xs)\ A \wedge ys \in nlists\ (length\ ys)\ A$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-map [simp]*: $(map\ f\ xs \in nlists\ (size\ xs)\ A) = (f\ ` set\ xs \subseteq A)$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-replicateI [intro]*: $x \in A \implies replicate\ n\ x \in nlists\ n\ A$
by (*induct n*) *auto*

Link to an executable version on lists in List.

lemma *nlists-set[code]*: $nlists\ n\ (set\ xs) = set(List.n-lists\ n\ xs)$
by (*metis nlists-def set-n-lists*)

end

70 Non-negative, non-positive integers and reals

theory *Nonpos-Ints*
imports *Complex-Main*
begin

70.1 Non-positive integers

The set of non-positive integers on a ring. (in analogy to the set of non-negative integers \mathbb{N}) This is useful e.g. for the Gamma function.

definition *nonpos-Ints* ($\langle \mathbb{Z}_{\leq 0} \rangle$) **where** $\mathbb{Z}_{\leq 0} = \{ \text{of-int } n \mid n. n \leq 0 \}$

lemma *zero-in-nonpos-Ints* [*simp,intro*]: $0 \in \mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def* **by** (*auto intro!*: *exI*[*of - 0::int*])

lemma *neg-one-in-nonpos-Ints* [*simp,intro*]: $-1 \in \mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def* **by** (*auto intro!*: *exI*[*of - -1::int*])

lemma *neg-numeral-in-nonpos-Ints* [*simp,intro*]: $-\text{numeral } n \in \mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def* **by** (*auto intro!*: *exI*[*of - -numeral n::int*])

lemma *one-notin-nonpos-Ints* [*simp*]: $(1 :: 'a :: \text{ring-char-0}) \notin \mathbb{Z}_{\leq 0}$
by (*auto simp*: *nonpos-Ints-def*)

lemma *numeral-notin-nonpos-Ints* [*simp*]: $(\text{numeral } n :: 'a :: \text{ring-char-0}) \notin \mathbb{Z}_{\leq 0}$
by (*auto simp*: *nonpos-Ints-def*)

lemma *minus-of-nat-in-nonpos-Ints* [*simp, intro*]: $-\text{of-nat } n \in \mathbb{Z}_{\leq 0}$

proof –

have $-\text{of-nat } n = \text{of-int } (-\text{int } n)$ **by** *simp*

also have $-\text{int } n \leq 0$ **by** *simp*

hence $\text{of-int } (-\text{int } n) \in \mathbb{Z}_{\leq 0}$ **unfolding** *nonpos-Ints-def* **by** *blast*

finally show *?thesis* .

qed

lemma *of-nat-in-nonpos-Ints-iff*: $(\text{of-nat } n :: 'a :: \{ \text{ring-1, ring-char-0} \}) \in \mathbb{Z}_{\leq 0}$
 $\longleftrightarrow n = 0$

proof

assume $(\text{of-nat } n :: 'a) \in \mathbb{Z}_{\leq 0}$

then obtain *m* **where** $\text{of-nat } n = (\text{of-int } m :: 'a)$ $m \leq 0$ **by** (*auto simp*:
nonpos-Ints-def)

hence $(\text{of-int } m :: 'a) = \text{of-nat } n$ **by** *simp*

also have $\dots = \text{of-int } (\text{int } n)$ **by** *simp*

finally have $m = \text{int } n$ **by** (*subst (asm)* *of-int-eq-iff*)

with $\langle m \leq 0 \rangle$ **show** $n = 0$ **by** *auto*

qed *simp*

lemma *nonpos-Ints-of-int*: $n \leq 0 \implies \text{of-int } n \in \mathbb{Z}_{\leq 0}$

unfolding *nonpos-Ints-def* **by** *blast*

lemma *nonpos-IntsI*:

$x \in \mathbb{Z} \implies x \leq 0 \implies (x :: 'a :: \text{linordered-idom}) \in \mathbb{Z}_{\leq 0}$

unfolding *nonpos-Ints-def Ints-def* **by** *auto*

lemma *nonpos-Ints-subset-Ints*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{Z}$

unfolding *nonpos-Ints-def Ints-def* **by** *blast*

lemma *nonpos-Ints-nonpos* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \implies x \leq (0 :: 'a :: \text{linordered-idom})$

unfolding *nonpos-Ints-def* **by** *auto*

lemma *nonpos-Ints-Int* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \implies x \in \mathbb{Z}$
unfolding *nonpos-Ints-def Ints-def* **by** *blast*

lemma *nonpos-Ints-cases*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = \text{of-int } n$ $n \leq 0$
using *assms* **unfolding** *nonpos-Ints-def* **by** (*auto elim!*: *Ints-cases*)

lemma *nonpos-Ints-cases'*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = -\text{of-nat } n$
proof –
from *assms* **obtain** m **where** $x = \text{of-int } m$ **and** $m: m \leq 0$ **by** (*auto elim!*:
nonpos-Ints-cases)
hence $x = -\text{of-int } (-m)$ **by** *auto*
also from m **have** $(\text{of-int } (-m) :: 'a) = \text{of-nat } (\text{nat } (-m))$ **by** *simp-all*
finally show *?thesis* **by** (*rule that*)
qed

lemma *of-real-in-nonpos-Ints-iff*: $(\text{of-real } x :: 'a :: \text{real-algebra-1}) \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
proof
assume $\text{of-real } x \in (\mathbb{Z}_{\leq 0} :: 'a \text{ set})$
then obtain n **where** $(\text{of-real } x :: 'a) = \text{of-int } n$ $n \leq 0$ **by** (*erule nonpos-Ints-cases*)
note $\langle \text{of-real } x = \text{of-int } n \rangle$
also have $\text{of-int } n = \text{of-real } (\text{of-int } n)$ **by** (*rule of-real-of-int-eq [symmetric]*)
finally have $x = \text{of-int } n$ **by** (*subst (asm) of-real-eq-iff*)
with $\langle n \leq 0 \rangle$ **show** $x \in \mathbb{Z}_{\leq 0}$ **by** (*simp add: nonpos-Ints-of-int*)
qed (*auto elim!*: *nonpos-Ints-cases intro!*: *nonpos-Ints-of-int*)

lemma *nonpos-Ints-altdef*: $\mathbb{Z}_{\leq 0} = \{n \in \mathbb{Z}. (n :: 'a :: \text{linordered-idom}) \leq 0\}$
by (*auto intro!*: *nonpos-IntsI elim!*: *nonpos-Ints-cases*)

lemma *uminus-in-Nats-iff*: $-x \in \mathbb{N} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
proof
assume $-x \in \mathbb{N}$
then obtain n **where** $n \geq 0$ $-x = \text{of-int } n$ **by** (*auto simp: Nats-altdef1*)
hence $-n \leq 0$ $x = \text{of-int } (-n)$ **by** (*simp-all add: eq-commute minus-equation-iff[of*
 $x]$)
thus $x \in \mathbb{Z}_{\leq 0}$ **unfolding** *nonpos-Ints-def* **by** *blast*
next
assume $x \in \mathbb{Z}_{\leq 0}$
then obtain n **where** $n \leq 0$ $x = \text{of-int } n$ **by** (*auto simp: nonpos-Ints-def*)
hence $-n \geq 0$ $-x = \text{of-int } (-n)$ **by** (*simp-all add: eq-commute minus-equation-iff[of*
 $x]$)
thus $-x \in \mathbb{N}$ **unfolding** *Nats-altdef1* **by** *blast*
qed

lemma *uminus-in-nonpos-Ints-iff*: $-x \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{N}$
using *uminus-in-Nats-iff*[*of* $-x$] **by** *simp*

lemma *nonpos-Ints-mult*: $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{Z}_{\leq 0} \implies x * y \in \mathbb{N}$
using *Nats-mult*[*of* $-x -y$] **by** (*simp add: uminus-in-Nats-iff*)

lemma *Nats-mult-nonpos-Ints*: $x \in \mathbb{N} \implies y \in \mathbb{Z}_{\leq 0} \implies x * y \in \mathbb{Z}_{\leq 0}$
using *Nats-mult*[*of* $x -y$] **by** (*simp add: uminus-in-Nats-iff*)

lemma *nonpos-Ints-mult-Nats*:
 $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{N} \implies x * y \in \mathbb{Z}_{\leq 0}$
using *Nats-mult*[*of* $-x y$] **by** (*simp add: uminus-in-Nats-iff*)

lemma *nonpos-Ints-add*:
 $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{Z}_{\leq 0} \implies x + y \in \mathbb{Z}_{\leq 0}$
using *Nats-add*[*of* $-x -y$] *uminus-in-Nats-iff*[*of* $y+x$, *simplified minus-add*]
by (*simp add: uminus-in-Nats-iff add.commute*)

lemma *nonpos-Ints-diff-Nats*:
 $x \in \mathbb{Z}_{\leq 0} \implies y \in \mathbb{N} \implies x - y \in \mathbb{Z}_{\leq 0}$
using *Nats-add*[*of* $-x y$] *uminus-in-Nats-iff*[*of* $x-y$, *simplified minus-add*]
by (*simp add: uminus-in-Nats-iff add.commute*)

lemma *Nats-diff-nonpos-Ints*:
 $x \in \mathbb{N} \implies y \in \mathbb{Z}_{\leq 0} \implies x - y \in \mathbb{N}$
using *Nats-add*[*of* $x -y$] **by** (*simp add: uminus-in-Nats-iff add.commute*)

lemma *plus-of-nat-eq-0-imp*: $z + \text{of-nat } n = 0 \implies z \in \mathbb{Z}_{\leq 0}$

proof –

assume $z + \text{of-nat } n = 0$

hence $A: z = - \text{of-nat } n$ **by** (*simp add: eq-neg-iff-add-eq-0*)

show $z \in \mathbb{Z}_{\leq 0}$ **by** (*subst A*) *simp*

qed

70.2 Non-negative reals

definition *nonneg-Reals* :: ‘*a* :: real-algebra-1 set ($\langle \mathbb{R}_{\geq 0} \rangle$)
where $\mathbb{R}_{\geq 0} = \{\text{of-real } r \mid r. r \geq 0\}$

lemma *nonneg-Reals-of-real-iff* [*simp*]: $\text{of-real } r \in \mathbb{R}_{\geq 0} \longleftrightarrow r \geq 0$
by (*force simp add: nonneg-Reals-def*)

lemma *nonneg-Reals-subset-Reals*: $\mathbb{R}_{\geq 0} \subseteq \mathbb{R}$
unfolding *nonneg-Reals-def Reals-def* **by** *blast*

lemma *nonneg-Reals-Real* [*dest*]: $x \in \mathbb{R}_{\geq 0} \implies x \in \mathbb{R}$
unfolding *nonneg-Reals-def Reals-def* **by** *blast*

lemma *nonneg-Reals-of-nat-I* [*simp*]: $\text{of-nat } n \in \mathbb{R}_{\geq 0}$

by (metis nonneg-Reals-of-real-iff of-nat-0-le-iff of-real-of-nat-eq)

lemma nonneg-Reals-cases:

assumes $x \in \mathbb{R}_{\geq 0}$

obtains r where $x = \text{of-real } r$ $r \geq 0$

using assms unfolding nonneg-Reals-def by (auto elim!: Reals-cases)

lemma nonneg-Reals-zero-I [simp]: $0 \in \mathbb{R}_{\geq 0}$

unfolding nonneg-Reals-def by auto

lemma nonneg-Reals-one-I [simp]: $1 \in \mathbb{R}_{\geq 0}$

by (metis (mono-tags, lifting) nonneg-Reals-of-nat-I of-nat-1)

lemma nonneg-Reals-minus-one-I [simp]: $-1 \notin \mathbb{R}_{\geq 0}$

by (metis nonneg-Reals-of-real-iff le-minus-one-simps(3) of-real-1 of-real-def real-vector.scale-minus-left)

lemma nonneg-Reals-numeral-I [simp]: numeral $w \in \mathbb{R}_{\geq 0}$

by (metis (no-types) nonneg-Reals-of-nat-I of-nat-numeral)

lemma nonneg-Reals-minus-numeral-I [simp]: $- \text{numeral } w \notin \mathbb{R}_{\geq 0}$

using nonneg-Reals-of-real-iff not-zero-le-neg-numeral by fastforce

lemma nonneg-Reals-add-I [simp]: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a + b \in \mathbb{R}_{\geq 0}$

apply (simp add: nonneg-Reals-def)

apply clarify

apply (rename-tac r s)

apply (rule-tac $x=r+s$ in exI , auto)

done

lemma nonneg-Reals-mult-I [simp]: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a * b \in \mathbb{R}_{\geq 0}$

unfolding nonneg-Reals-def by (auto simp: of-real-def)

lemma nonneg-Reals-inverse-I [simp]:

fixes $a :: 'a::\text{real-div-algebra}$

shows $a \in \mathbb{R}_{\geq 0} \implies \text{inverse } a \in \mathbb{R}_{\geq 0}$

by (simp add: nonneg-Reals-def image-iff) (metis inverse-nonnegative-iff-nonnegative of-real-inverse)

lemma nonneg-Reals-divide-I [simp]:

fixes $a :: 'a::\text{real-div-algebra}$

shows $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a / b \in \mathbb{R}_{\geq 0}$

by (simp add: divide-inverse)

lemma nonneg-Reals-pow-I [simp]: $a \in \mathbb{R}_{\geq 0} \implies a^{\wedge n} \in \mathbb{R}_{\geq 0}$

by (induction n) auto

lemma complex-nonneg-Reals-iff: $z \in \mathbb{R}_{\geq 0} \iff \text{Re } z \geq 0 \wedge \text{Im } z = 0$

by (auto simp: nonneg-Reals-def) (metis complex-of-real-def complex-surj)

lemma *ii-not-nonneg-Reals* [iff]: $i \notin \mathbb{R}_{\geq 0}$
by (*simp add: complex-nonneg-Reals-iff*)

70.3 Non-positive reals

definition *nonpos-Reals* :: ‘a::real-algebra-1 set ($\langle \mathbb{R}_{\leq 0} \rangle$)
where $\mathbb{R}_{\leq 0} = \{\text{of-real } r \mid r. r \leq 0\}$

lemma *nonpos-Reals-of-real-iff* [simp]: $\text{of-real } r \in \mathbb{R}_{\leq 0} \longleftrightarrow r \leq 0$
by (*force simp add: nonpos-Reals-def*)

lemma *nonpos-Reals-subset-Reals*: $\mathbb{R}_{\leq 0} \subseteq \mathbb{R}$
unfolding *nonpos-Reals-def Reals-def* **by** *blast*

lemma *nonpos-Ints-subset-nonpos-Reals*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{R}_{\leq 0}$
by (*metis nonpos-Ints-cases nonpos-Ints-nonpos nonpos-Ints-of-int nonpos-Reals-of-real-iff of-real-of-int-eq subsetI*)

lemma *nonpos-Reals-of-nat-iff* [simp]: $\text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow n = 0$
by (*metis nonpos-Reals-of-real-iff of-nat-le-0-iff of-real-of-nat-eq*)

lemma *nonpos-Reals-Real* [dest]: $x \in \mathbb{R}_{\leq 0} \implies x \in \mathbb{R}$
unfolding *nonpos-Reals-def Reals-def* **by** *blast*

lemma *nonpos-Reals-cases*:
assumes $x \in \mathbb{R}_{\leq 0}$
obtains r **where** $x = \text{of-real } r$ $r \leq 0$
using *assms* **unfolding** *nonpos-Reals-def* **by** (*auto elim!: Reals-cases*)

lemma *uminus-nonneg-Reals-iff* [simp]: $-x \in \mathbb{R}_{\geq 0} \longleftrightarrow x \in \mathbb{R}_{\leq 0}$
apply (*auto simp: nonpos-Reals-def nonneg-Reals-def*)
apply (*metis nonpos-Reals-of-real-iff minus-minus neg-le-0-iff-le of-real-minus*)
done

lemma *uminus-nonpos-Reals-iff* [simp]: $-x \in \mathbb{R}_{\leq 0} \longleftrightarrow x \in \mathbb{R}_{\geq 0}$
by (*metis (no-types) minus-minus uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-zero-I* [simp]: $0 \in \mathbb{R}_{\leq 0}$
unfolding *nonpos-Reals-def* **by** *force*

lemma *nonpos-Reals-one-I* [simp]: $1 \notin \mathbb{R}_{\leq 0}$
using *nonneg-Reals-minus-one-I uminus-nonneg-Reals-iff* **by** *blast*

lemma *nonpos-Reals-numeral-I* [simp]: $\text{numeral } w \notin \mathbb{R}_{\leq 0}$
using *nonneg-Reals-minus-numeral-I uminus-nonneg-Reals-iff* **by** *blast*

lemma *nonpos-Reals-add-I* [simp]: $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a + b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-add-I add-uminus-conv-diff minus-diff-eq minus-minus uminus-nonpos-Reals-iff*)

lemma *nonpos-Reals-mult-I1*: $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-mult-I mult-minus-right uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-mult-I2*: $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a * b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-mult-I mult-minus-left uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-mult-of-nat-iff*:
fixes *a* :: 'a :: real-div-algebra **shows** $a * \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n = 0$
apply (*auto intro: nonpos-Reals-mult-I2*)
apply (*auto simp: nonpos-Reals-def*)
apply (*rule-tac x=r/n in exI*)
apply (*auto simp: field-split-simps*)
done

lemma *nonpos-Reals-inverse-I*:
fixes *a* :: 'a :: real-div-algebra
shows $a \in \mathbb{R}_{\leq 0} \implies \text{inverse } a \in \mathbb{R}_{\leq 0}$
using *nonneg-Reals-inverse-I uminus-nonneg-Reals-iff* **by** *fastforce*

lemma *nonpos-Reals-divide-I1*:
fixes *a* :: 'a :: real-div-algebra
shows $\llbracket a \in \mathbb{R}_{\geq 0}; b \in \mathbb{R}_{\leq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$
by (*simp add: nonpos-Reals-inverse-I nonpos-Reals-mult-I1 divide-inverse*)

lemma *nonpos-Reals-divide-I2*:
fixes *a* :: 'a :: real-div-algebra
shows $\llbracket a \in \mathbb{R}_{\leq 0}; b \in \mathbb{R}_{\geq 0} \rrbracket \implies a / b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-divide-I minus-divide-left uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-divide-of-nat-iff*:
fixes *a* :: 'a :: real-div-algebra **shows** $a / \text{of-nat } n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n = 0$
apply (*auto intro: nonpos-Reals-divide-I2*)
apply (*auto simp: nonpos-Reals-def*)
apply (*rule-tac x=r*n in exI*)
apply (*auto simp: field-split-simps mult-le-0-iff*)
done

lemma *nonpos-Reals-inverse-iff* [*simp*]:
fixes *a* :: 'a :: real-div-algebra
shows $\text{inverse } a \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0}$
using *nonpos-Reals-inverse-I* **by** *fastforce*

lemma *nonpos-Reals-pow-I*: $\llbracket a \in \mathbb{R}_{\leq 0}; \text{odd } n \rrbracket \implies a^{\wedge n} \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-pow-I power-minus-odd uminus-nonneg-Reals-iff*)

lemma *complex-nonpos-Reals-iff*: $z \in \mathbb{R}_{\leq 0} \longleftrightarrow \text{Re } z \leq 0 \wedge \text{Im } z = 0$
using *complex-is-Real-iff* **by** (*force simp add: nonpos-Reals-def*)

lemma *ii-not-nonpos-Reals [iff]*: $i \notin \mathbb{R}_{\leq 0}$
by (*simp add: complex-nonpos-Reals-iff*)

lemma *plus-one-in-nonpos-Ints-imp*: $z + 1 \in \mathbb{Z}_{\leq 0} \implies z \in \mathbb{Z}_{\leq 0}$
using *nonpos-Ints-diff-Nats[$of\ z+1\ 1$]* **by** *simp-all*

lemma *of-int-in-nonpos-Ints-iff*:
 $(of-int\ n :: 'a :: ring-char-0) \in \mathbb{Z}_{\leq 0} \longleftrightarrow n \leq 0$
by (*auto simp: nonpos-Ints-def*)

lemma *one-plus-of-int-in-nonpos-Ints-iff*:
 $(1 + of-int\ n :: 'a :: ring-char-0) \in \mathbb{Z}_{\leq 0} \longleftrightarrow n \leq -1$
proof –
have $1 + of-int\ n = (of-int\ (n + 1) :: 'a)$ **by** *simp*
also have $\dots \in \mathbb{Z}_{\leq 0} \longleftrightarrow n + 1 \leq 0$ **by** (*subst of-int-in-nonpos-Ints-iff*) *simp-all*
also have $\dots \longleftrightarrow n \leq -1$ **by** *presburger*
finally show *?thesis* .
qed

lemma *one-minus-of-nat-in-nonpos-Ints-iff*:
 $(1 - of-nat\ n :: 'a :: ring-char-0) \in \mathbb{Z}_{\leq 0} \longleftrightarrow n > 0$
proof –
have $(1 - of-nat\ n :: 'a) = of-int\ (1 - int\ n)$ **by** *simp*
also have $\dots \in \mathbb{Z}_{\leq 0} \longleftrightarrow n > 0$ **by** (*subst of-int-in-nonpos-Ints-iff*) *presburger*
finally show *?thesis* .
qed

lemma *fraction-not-in-Nats*:
assumes $\neg n\ dvd\ m\ n \neq 0$
shows $of-int\ m / of-int\ n \notin (\mathbb{N} :: 'a :: \{division-ring, ring-char-0\}\ set)$
proof
assume $of-int\ m / of-int\ n \in (\mathbb{N} :: 'a\ set)$
also note *Nats-subset-Ints*
finally have $of-int\ m / of-int\ n \in (\mathbb{Z} :: 'a\ set)$.
moreover have $of-int\ m / of-int\ n \notin (\mathbb{Z} :: 'a\ set)$
using *assms* **by** (*intro fraction-not-in-Ints*)
ultimately show *False* **by** *contradiction*
qed

lemma *not-in-Ints-imp-not-in-nonpos-Ints*: $z \notin \mathbb{Z} \implies z \notin \mathbb{Z}_{\leq 0}$
by (*auto simp: Ints-def nonpos-Ints-def*)

lemma *double-in-nonpos-Ints-imp*:
assumes $2 * (z :: 'a :: field-char-0) \in \mathbb{Z}_{\leq 0}$
shows $z \in \mathbb{Z}_{\leq 0} \vee z + 1/2 \in \mathbb{Z}_{\leq 0}$
proof –
from *assms* **obtain** *k* **where** $2 * z = - of-nat\ k$ **by** (*elim nonpos-Ints-cases'*)
thus *?thesis* **by** (*cases even k*) (*auto elim!: evenE oddE simp: field-simps*)
qed

```

lemma fraction-numeral-Ints-iff [simp]:
  numeral a / numeral b ∈ (ℤ :: 'a :: {division-ring, ring-char-0} set)
  ⟷ (numeral b :: int) dvd numeral a (is ?L=?R)
proof
  show ?L ⟹ ?R
    by (metis fraction-not-in-Ints of-int-numeral zero-neq-numeral)
  assume ?R
  then obtain k::int where numeral a = numeral b * (of-int k :: 'a)
    unfolding dvd-def by (metis of-int-mult of-int-numeral)
  then show ?L
    by (metis Ints-of-int divide-eq-eq mult.commute of-int-mult of-int-numeral)
qed

```

```

lemma fraction-numeral-Ints-iff1 [simp]:
  1 / numeral b ∈ (ℤ :: 'a :: {division-ring, ring-char-0} set)
  ⟷ b = Num.One (is ?L=?R)
  using fraction-numeral-Ints-iff [of Num.One, where 'a='a] by simp

```

```

lemma fraction-numeral-Nats-iff [simp]:
  numeral a / numeral b ∈ (ℕ :: 'a :: {division-ring, ring-char-0} set)
  ⟷ (numeral b :: int) dvd numeral a (is ?L=?R)
proof
  show ?L ⟹ ?R
    using Nats-subset-Ints fraction-numeral-Ints-iff by blast
  assume ?R
  then obtain k::nat where numeral a = numeral b * (of-nat k :: 'a)
    unfolding dvd-def
    by (metis dvd-def int-dvd-int-iff of-nat-mult of-nat-numeral)
  then show ?L
    by (metis mult-of-nat-commute nonzero-divide-eq-eq of-nat-in-Nats
      zero-neq-numeral)
qed

```

```

lemma fraction-numeral-Nats-iff1 [simp]:
  1 / numeral b ∈ (ℕ :: 'a :: {division-ring, ring-char-0} set)
  ⟷ b = Num.One (is ?L=?R)
  using fraction-numeral-Nats-iff [of Num.One, where 'a='a] by simp
end

```

71 Numeral Syntax for Types

```

theory Numeral-Type
imports Cardinality
begin

```

71.1 Numeral Types

```
typedef num0 = UNIV :: nat set ..
typedef num1 = UNIV :: unit set ..
```

```
typedef 'a bit0 = {0 ..< 2 * int CARD('a::finite)}
proof
  show 0 ∈ {0 ..< 2 * int CARD('a)}
  by simp
qed
```

```
typedef 'a bit1 = {0 ..< 1 + 2 * int CARD('a::finite)}
proof
  show 0 ∈ {0 ..< 1 + 2 * int CARD('a)}
  by simp
qed
```

```
lemma card-num0 [simp]: CARD (num0) = 0
  unfolding type-definition.card [OF type-definition-num0]
  by simp
```

```
lemma infinite-num0: ¬ finite (UNIV :: num0 set)
  using card-num0[unfolded card-eq-0-iff]
  by simp
```

```
lemma card-num1 [simp]: CARD (num1) = 1
  unfolding type-definition.card [OF type-definition-num1]
  by (simp only: card-UNIV-unit)
```

```
lemma card-bit0 [simp]: CARD('a bit0) = 2 * CARD('a::finite)
  unfolding type-definition.card [OF type-definition-bit0]
  by simp
```

```
lemma card-bit1 [simp]: CARD('a bit1) = Suc (2 * CARD('a::finite))
  unfolding type-definition.card [OF type-definition-bit1]
  by simp
```

71.2 num1

```
instance num1 :: finite
proof
  show finite (UNIV::num1 set)
    unfolding type-definition.univ [OF type-definition-num1]
    using finite by (rule finite-imageI)
qed
```

```
instantiation num1 :: CARD-1
begin
```

```
instance
```

```

proof
  show  $CARD(num1) = 1$  by auto
qed

end

lemma num1-eq-iff:  $(x::num1) = (y::num1) \longleftrightarrow True$ 
  by (induct x, induct y simp)

instantiation num1 :: {comm-ring, comm-monoid-mult, numeral}
begin

instance
  by standard (simp-all add: num1-eq-iff)

end

lemma num1-eqI:
  fixes  $a::num1$  shows  $a = b$ 
by(simp add: num1-eq-iff)

lemma num1-eq1 [simp]:
  fixes  $a::num1$  shows  $a = 1$ 
  by (rule num1-eqI)

lemma forall-1 [simp]:  $(\forall i::num1. P\ i) \longleftrightarrow P\ 1$ 
  by (metis (full-types) num1-eq-iff)

lemma ex-1 [simp]:  $(\exists x::num1. P\ x) \longleftrightarrow P\ 1$ 
  by auto (metis (full-types) num1-eq-iff)

instantiation num1 :: linorder begin
definition  $a < b \longleftrightarrow Rep\_num1\ a < Rep\_num1\ b$ 
definition  $a \leq b \longleftrightarrow Rep\_num1\ a \leq Rep\_num1\ b$ 
instance
  by intro-classes (auto simp: less-eq-num1-def less-num1-def intro: num1-eqI)
end

instance num1 :: wellorder
  by intro-classes (auto simp: less-eq-num1-def less-num1-def)

instance bit0 :: (finite) card2
proof
  show finite (UNIV::'a bit0 set)
    unfolding type-definition.univ [OF type-definition-bit0]
    by simp
  show  $2 \leq CARD('a\ bit0)$ 
    by simp

```


qed

```
instance bit1 :: (finite) card2
proof
  show finite (UNIV::'a bit1 set)
    unfolding type-definition.univ [OF type-definition-bit1]
    by simp
  show 2 ≤ CARD('a bit1)
    by simp
qed
```

71.3 Locales for modular arithmetic subtypes

```
locale mod-type =
  fixes n :: int
  and Rep :: 'a::{zero,one,plus,times,uminus,minus} ⇒ int
  and Abs :: int ⇒ 'a::{zero,one,plus,times,uminus,minus}
  assumes type: type-definition Rep Abs {0.. $n$ }
  and size1: 1 < n
  and zero-def: 0 = Abs 0
  and one-def: 1 = Abs 1
  and add-def:  $x + y = \text{Abs } ((\text{Rep } x + \text{Rep } y) \bmod n)$ 
  and mult-def:  $x * y = \text{Abs } ((\text{Rep } x * \text{Rep } y) \bmod n)$ 
  and diff-def:  $x - y = \text{Abs } ((\text{Rep } x - \text{Rep } y) \bmod n)$ 
  and minus-def:  $-x = \text{Abs } ((-\text{Rep } x) \bmod n)$ 
begin

lemma size0: 0 < n
  using size1 by simp

lemmas definitions =
  zero-def one-def add-def mult-def minus-def diff-def

lemma Rep-less-n: Rep x < n
  by (rule type-definition.Rep [OF type, simplified, THEN conjunct2])

lemma Rep-le-n: Rep x ≤ n
  by (rule Rep-less-n [THEN order-less-imp-le])

lemma Rep-inject-sym:  $x = y \longleftrightarrow \text{Rep } x = \text{Rep } y$ 
  by (rule type-definition.Rep-inject [OF type, symmetric])

lemma Rep-inverse: Abs (Rep x) = x
  by (rule type-definition.Rep-inverse [OF type])

lemma Abs-inverse:  $m \in \{0.. $n$ \} \implies \text{Rep } (\text{Abs } m) = m$ 
  by (rule type-definition.Abs-inverse [OF type])

lemma Rep-Abs-mod: Rep (Abs (m mod n)) = m mod n
```

```

using size0 by (simp add: Abs-inverse)

lemma Rep-Abs-0: Rep (Abs 0) = 0
by (simp add: Abs-inverse size0)

lemma Rep-0: Rep 0 = 0
by (simp add: zero-def Rep-Abs-0)

lemma Rep-Abs-1: Rep (Abs 1) = 1
by (simp add: Abs-inverse size1)

lemma Rep-1: Rep 1 = 1
by (simp add: one-def Rep-Abs-1)

lemma Rep-mod: Rep x mod n = Rep x
using type-definition.Abs-cases [OF type]
by (metis Abs-inverse atLeastLessThan-iff mod-pos-pos-trivial)

lemmas Rep-simps =
  Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

lemma comm-ring-1: OFCLASS('a, comm-ring-1-class)
by intro-classes (auto simp: Rep-simps mod-simps field-simps definitions)

end

locale mod-ring = mod-type n Rep Abs
  for n :: int
  and Rep :: 'a::{comm-ring-1}  $\Rightarrow$  int
  and Abs :: int  $\Rightarrow$  'a::{comm-ring-1}
begin

lemma of-nat-eq: of-nat k = Abs (int k mod n)
by (induct k) (simp-all add: zero-def Rep-simps add-def one-def mod-simps
ac-simps)

lemma of-int-eq: of-int z = Abs (z mod n)
by (cases z rule: int-diff-cases) (simp add: Rep-simps of-nat-eq diff-def mod-simps)

lemma Rep-numeral: Rep (numeral w) = numeral w mod n
by (metis Rep-Abs-mod of-int-eq of-int-numeral)

lemma iszero-numeral:
  iszero (numeral w::'a)  $\longleftrightarrow$  numeral w mod n = 0
by (simp add: Rep-inject-sym Rep-numeral Rep-0 iszero-def)

lemma cases:
  assumes 1:  $\bigwedge z. \llbracket (x::'a) = \text{of-int } z; 0 \leq z; z < n \rrbracket \Longrightarrow P$ 
  shows P

```

by (*metis Rep-inverse Rep-less-n Rep-mod assms of-int-eq pos-mod-sign size0*)

lemma *induct*:

($\bigwedge z. \llbracket 0 \leq z; z < n \rrbracket \implies P \text{ (of-int } z) \implies P \text{ (} x :: 'a \text{)}$)
by (*cases x rule: cases simp*)

lemma *UNIV-eq*: (*UNIV* :: 'a set) = Abs ‘ {0..*n*}
using *type type-definition.univ* **by** *blast*

lemma *CARD-eq*: *CARD*('a) = nat *n*

proof –

have *CARD*('a) = card (Abs ‘ {0..*n*})
by (*simp add: UNIV-eq*)
also have *inj-on* Abs {0..*n*}
by (*metis Abs-inverse inj-onI*)
hence card (Abs ‘ {0..*n*}) = nat *n*
using *size1* **by** (*subst card-image*) *auto*
finally show ?*thesis* .

qed

lemma *CHAR-eq* [*simp*]: *CHAR*('a) = *CARD*('a)

proof (*rule CHAR-eqI*)

show *of-nat* (*CARD*('a)) = (0 :: 'a)
by (*simp add: CARD-eq of-nat-eq zero-def*)

next

fix *n* **assume** *of-nat* *n* = (0 :: 'a)

thus *CARD*('a) *dvd* *n*

by (*metis CARD-eq Rep-0 Rep-Abs-mod Rep-le-n mod-0-imp-dvd nat-dvd-iff of-nat-eq*)

qed

end

71.4 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation

bit0 and *bit1* :: (*finite*) {*zero,one,plus,times,uminus,minus*}

begin

definition *Abs-bit0'* :: *int* \Rightarrow 'a *bit0* **where**

Abs-bit0' *x* = *Abs-bit0* (*x mod int CARD*('a *bit0*))

definition *Abs-bit1'* :: *int* \Rightarrow 'a *bit1* **where**

Abs-bit1' *x* = *Abs-bit1* (*x mod int CARD*('a *bit1*))

definition 0 = *Abs-bit0* 0

definition 1 = *Abs-bit0* 1

definition $x + y = \text{Abs-bit0}' (\text{Rep-bit0 } x + \text{Rep-bit0 } y)$

definition $x * y = \text{Abs-bit0}' (\text{Rep-bit0 } x * \text{Rep-bit0 } y)$

definition $x - y = \text{Abs-bit0}' (\text{Rep-bit0 } x - \text{Rep-bit0 } y)$

definition $- x = \text{Abs-bit0}' (- \text{Rep-bit0 } x)$

definition $0 = \text{Abs-bit1 } 0$

definition $1 = \text{Abs-bit1 } 1$

definition $x + y = \text{Abs-bit1}' (\text{Rep-bit1 } x + \text{Rep-bit1 } y)$

definition $x * y = \text{Abs-bit1}' (\text{Rep-bit1 } x * \text{Rep-bit1 } y)$

definition $x - y = \text{Abs-bit1}' (\text{Rep-bit1 } x - \text{Rep-bit1 } y)$

definition $- x = \text{Abs-bit1}' (- \text{Rep-bit1 } x)$

instance ..

end

interpretation *bit0*:

mod-type int CARD('a::finite bit0)

Rep-bit0 :: 'a::finite bit0 \Rightarrow int

Abs-bit0 :: int \Rightarrow 'a::finite bit0

apply (*rule mod-type.intro*)

apply (*simp add: type-definition-bit0*)

apply (*rule one-less-int-card*)

apply (*rule zero-bit0-def*)

apply (*rule one-bit0-def*)

apply (*rule plus-bit0-def [unfolded Abs-bit0'-def]*)

apply (*rule times-bit0-def [unfolded Abs-bit0'-def]*)

apply (*rule minus-bit0-def [unfolded Abs-bit0'-def]*)

apply (*rule uminus-bit0-def [unfolded Abs-bit0'-def]*)

done

interpretation *bit1*:

mod-type int CARD('a::finite bit1)

Rep-bit1 :: 'a::finite bit1 \Rightarrow int

Abs-bit1 :: int \Rightarrow 'a::finite bit1

apply (*rule mod-type.intro*)

apply (*simp add: type-definition-bit1*)

apply (*rule one-less-int-card*)

apply (*rule zero-bit1-def*)

apply (*rule one-bit1-def*)

apply (*rule plus-bit1-def [unfolded Abs-bit1'-def]*)

apply (*rule times-bit1-def [unfolded Abs-bit1'-def]*)

apply (*rule minus-bit1-def [unfolded Abs-bit1'-def]*)

apply (*rule uminus-bit1-def [unfolded Abs-bit1'-def]*)

done

instance *bit0 :: (finite) comm-ring-1*

by (*rule bit0.comm-ring-1*)

```
instance bit1 :: (finite) comm-ring-1
  by (rule bit1.comm-ring-1)
```

```
interpretation bit0:
  mod-ring int CARD('a::finite bit0)
  Rep-bit0 :: 'a::finite bit0  $\Rightarrow$  int
  Abs-bit0 :: int  $\Rightarrow$  'a::finite bit0
  ..
```

```
interpretation bit1:
  mod-ring int CARD('a::finite bit1)
  Rep-bit1 :: 'a::finite bit1  $\Rightarrow$  int
  Abs-bit1 :: int  $\Rightarrow$  'a::finite bit1
  ..
```

Set up cases, induction, and arithmetic

```
lemmas bit0-cases [case-names of-int, cases type: bit0] = bit0.cases
lemmas bit1-cases [case-names of-int, cases type: bit1] = bit1.cases
```

```
lemmas bit0-induct [case-names of-int, induct type: bit0] = bit0.induct
lemmas bit1-induct [case-names of-int, induct type: bit1] = bit1.induct
```

```
lemmas bit0-iszero-numeral [simp] = bit0.iszero-numeral
lemmas bit1-iszero-numeral [simp] = bit1.iszero-numeral
```

```
lemmas [simp] = eq-numeral-iff-iszero [where 'a='a bit0] for dummy :: 'a::finite
lemmas [simp] = eq-numeral-iff-iszero [where 'a='a bit1] for dummy :: 'a::finite
```

71.5 Order instances

```
instantiation bit0 and bit1 :: (finite) linorder begin
```

```
definition a < b  $\longleftrightarrow$  Rep-bit0 a < Rep-bit0 b
```

```
definition a  $\leq$  b  $\longleftrightarrow$  Rep-bit0 a  $\leq$  Rep-bit0 b
```

```
definition a < b  $\longleftrightarrow$  Rep-bit1 a < Rep-bit1 b
```

```
definition a  $\leq$  b  $\longleftrightarrow$  Rep-bit1 a  $\leq$  Rep-bit1 b
```

```
instance
```

```
  by(intro-classes)
```

```
  (auto simp add: less-eq-bit0-def less-bit0-def less-eq-bit1-def less-bit1-def Rep-bit0-inject
  Rep-bit1-inject)
```

```
end
```

```
instance bit0 and bit1 :: (finite) wellorder
```

```
proof –
```

```
  have wf {(x :: 'a bit0, y). x < y}
```

```
    by(auto simp add: trancl-def tranclp-less intro!: finite-acyclic-wf acyclicI)
```

```
  thus OFCLASS('a bit0, wellorder-class)
```

```
    by(rule wf-wellorderI) intro-classes
```

```
next
```

```
  have wf {(x :: 'a bit1, y). x < y}
```

```

    by(auto simp add: trancl-def tranclp-less intro!: finite-acyclic-wf acyclicI)
  thus OFCLASS('a bit1, wellorder-class)
    by(rule wf-wellorderI) intro-classes
qed

```

71.6 Code setup and type classes for code generation

Code setup for *num0* and *num1*

```

definition Num0 :: num0 where Num0 = Abs-num0 0
code-datatype Num0

```

```

instantiation num0 :: equal begin
definition equal-num0 :: num0  $\Rightarrow$  num0  $\Rightarrow$  bool
  where equal-num0 = (=)
instance by intro-classes (simp add: equal-num0-def)
end

```

```

lemma equal-num0-code [code]:
  equal-class.equal Num0 Num0 = True
by(rule equal-refl)

```

```

code-datatype 1 :: num1

```

```

instantiation num1 :: equal begin
definition equal-num1 :: num1  $\Rightarrow$  num1  $\Rightarrow$  bool
  where equal-num1 = (=)
instance by intro-classes (simp add: equal-num1-def)
end

```

```

lemma equal-num1-code [code]:
  equal-class.equal (1 :: num1) 1 = True
by(rule equal-refl)

```

```

instantiation num1 :: enum begin
definition enum-class.enum = [1 :: num1]
definition enum-class.enum-all P = P (1 :: num1)
definition enum-class.enum-ex P = P (1 :: num1)
instance
  by intro-classes
  (auto simp add: enum-num1-def enum-all-num1-def enum-ex-num1-def num1-eq-iff
    Ball-def)
end

```

```

instantiation num0 and num1 :: card-UNIV begin
definition finite-UNIV = Phantom(num0) False
definition card-UNIV = Phantom(num0) 0
definition finite-UNIV = Phantom(num1) True
definition card-UNIV = Phantom(num1) 1
instance

```

```

by intro-classes
  (simp-all add: finite-UNIV-num0-def card-UNIV-num0-def infinite-num0 fi-
nite-UNIV-num1-def card-UNIV-num1-def)
end

```

Code setup for '*a bit0*' and '*a bit1*'

```

declare
  bit0.Rep-inverse[code abstype]
  bit0.Rep-0[code abstract]
  bit0.Rep-1[code abstract]

```

```

lemma Abs-bit0'-code [code abstract]:
  Rep-bit0 (Abs-bit0' x :: 'a :: finite bit0) = x mod int (CARD('a bit0))
by(auto simp add: Abs-bit0'-def intro!: Abs-bit0-inverse)

```

```

lemma inj-on-Abs-bit0:
  inj-on (Abs-bit0 :: int  $\Rightarrow$  'a bit0) {0.. $2 * \text{int } \text{CARD}('a :: \text{finite})$ }}
by(auto intro: inj-onI simp add: Abs-bit0-inject)

```

```

declare
  bit1.Rep-inverse[code abstype]
  bit1.Rep-0[code abstract]
  bit1.Rep-1[code abstract]

```

```

lemma Abs-bit1'-code [code abstract]:
  Rep-bit1 (Abs-bit1' x :: 'a :: finite bit1) = x mod int (CARD('a bit1))
by(auto simp add: Abs-bit1'-def intro!: Abs-bit1-inverse)

```

```

lemma inj-on-Abs-bit1:
  inj-on (Abs-bit1 :: int  $\Rightarrow$  'a bit1) {0.. $1 + 2 * \text{int } \text{CARD}('a :: \text{finite})$ }}
by(auto intro: inj-onI simp add: Abs-bit1-inject)

```

instantiation *bit0* and *bit1* :: (*finite*) *equal* **begin**

definition *equal-class.equal* *x y* \longleftrightarrow *Rep-bit0 x = Rep-bit0 y*

definition *equal-class.equal* *x y* \longleftrightarrow *Rep-bit1 x = Rep-bit1 y*

instance

by *intro-classes* (*simp-all add: equal-bit0-def equal-bit1-def Rep-bit0-inject Rep-bit1-inject*)

end

instantiation *bit0* :: (*finite*) *enum* **begin**

definition (*enum-class.enum* :: '*a bit0* list) = *map (Abs-bit0' \circ int) (upt 0 (CARD('a bit0)))*

definition *enum-class.enum-all* *P* = ($\forall b :: 'a \text{ bit0} \in \text{set enum-class.enum. } P \ b$)

definition *enum-class.enum-ex* *P* = ($\exists b :: 'a \text{ bit0} \in \text{set enum-class.enum. } P \ b$)

instance **proof**

```

show distinct (enum-class.enum :: 'a bit0 list)
by (simp add: enum-bit0-def distinct-map inj-on-def Abs-bit0'-def Abs-bit0-inject)

let ?Abs = Abs-bit0 :: -  $\Rightarrow$  'a bit0
interpret type-definition Rep-bit0 ?Abs {0.. $2 * \text{int CARD('a)}$ }
by (fact type-definition-bit0)
have UNIV = ?Abs ' {0.. $2 * \text{int CARD('a)}$ }
by (simp add: Abs-image)
also have ... = ?Abs ' (int ' {0.. $2 * \text{CARD('a)}$ })
by (simp add: image-int-atLeastLessThan)
also have ... = (?Abs  $\circ$  int) ' {0.. $2 * \text{CARD('a)}$ }
by (simp add: image-image cong: image-cong)
also have ... = set enum-class.enum
by (simp add: enum-bit0-def Abs-bit0'-def cong: image-cong-simp)
finally show univ-eq: (UNIV :: 'a bit0 set) = set enum-class.enum .

fix P :: 'a bit0  $\Rightarrow$  bool
show enum-class.enum-all P = Ball UNIV P
and enum-class.enum-ex P = Bex UNIV P
by (simp-all add: enum-all-bit0-def enum-ex-bit0-def univ-eq)
qed

end

instantiation bit1 :: (finite) enum begin
definition (enum-class.enum :: 'a bit1 list) = map (Abs-bit1'  $\circ$  int) (upt 0 (CARD('a bit1)))
definition enum-class.enum-all P = ( $\forall b :: 'a \text{ bit1} \in \text{set enum-class.enum. } P \ b$ )
definition enum-class.enum-ex P = ( $\exists b :: 'a \text{ bit1} \in \text{set enum-class.enum. } P \ b$ )

instance
proof (intro-classes)
show distinct (enum-class.enum :: 'a bit1 list)
by (simp only: Abs-bit1'-def zmod-int[symmetric] enum-bit1-def distinct-map
  Suc-eq-plus1 card-bit1 o-apply inj-on-def)
  (clarsimp simp add: Abs-bit1-inject)

let ?Abs = Abs-bit1 :: -  $\Rightarrow$  'a bit1
interpret type-definition Rep-bit1 ?Abs {0.. $1 + 2 * \text{int CARD('a)}$ }
by (fact type-definition-bit1)
have UNIV = ?Abs ' {0.. $1 + 2 * \text{int CARD('a)}$ }
by (simp add: Abs-image)
also have ... = ?Abs ' (int ' {0.. $1 + 2 * \text{CARD('a)}$ })
by (simp add: image-int-atLeastLessThan)
also have ... = (?Abs  $\circ$  int) ' {0.. $1 + 2 * \text{CARD('a)}$ }
by (simp add: image-image cong: image-cong)
finally show univ-eq: (UNIV :: 'a bit1 set) = set enum-class.enum
by (simp only: enum-bit1-def set-map set-upt) (simp add: Abs-bit1'-def cong:
  image-cong-simp)

```



```

fix  $P :: 'a \text{ bit1} \Rightarrow \text{bool}$ 
show  $\text{enum-class.enum-all } P = \text{Ball UNIV } P$ 
and  $\text{enum-class.enum-ex } P = \text{Bex UNIV } P$ 
by( $\text{simp-all add: enum-all-bit1-def enum-ex-bit1-def univ-eq}$ )
qed

end

instantiation  $\text{bit0 and bit1} :: (\text{finite}) \text{ finite-UNIV begin}$ 
definition  $\text{finite-UNIV} = \text{Phantom}('a \text{ bit0}) \text{ True}$ 
definition  $\text{finite-UNIV} = \text{Phantom}('a \text{ bit1}) \text{ True}$ 
instance by  $\text{intro-classes (simp-all add: finite-UNIV-bit0-def finite-UNIV-bit1-def)}$ 
end

instantiation  $\text{bit0 and bit1} :: (\{\text{finite}, \text{card-UNIV}\}) \text{ card-UNIV begin}$ 
definition  $\text{card-UNIV} = \text{Phantom}('a \text{ bit0}) (2 * \text{of-phantom } (\text{card-UNIV} :: 'a \text{ card-UNIV}))$ 
definition  $\text{card-UNIV} = \text{Phantom}('a \text{ bit1}) (1 + 2 * \text{of-phantom } (\text{card-UNIV} :: 'a \text{ card-UNIV}))$ 
instance by  $\text{intro-classes (simp-all add: card-UNIV-bit0-def card-UNIV-bit1-def card-UNIV)}$ 
end

```

71.7 Syntax

syntax

```

-NumeralType :: num-token => type (⟨⟨open-block notation=⟨type-literal number⟩⟩-⟩)
-NumeralType0 :: type (⟨⟨open-block notation=⟨type-literal number⟩⟩0⟩)
-NumeralType1 :: type (⟨⟨open-block notation=⟨type-literal number⟩⟩1⟩)

```

translations

```

(type) 1 == (type) num1
(type) 0 == (type) num0

```

parse-translation <

```

let
  fun mk-bintype n =
    let
      fun mk-bit 0 = Syntax.const type-syntax ⟨bit0⟩
        | mk-bit 1 = Syntax.const type-syntax ⟨bit1⟩;
      fun bin-of n =
        if n = 1 then Syntax.const type-syntax ⟨num1⟩
        else if n = 0 then Syntax.const type-syntax ⟨num0⟩
        else if n = ~1 then raise TERM (negative type numeral, [])
        else
          let val (q, r) = Integer.div-mod n 2;
              in mk-bit r $ bin-of q end;
    end

```

```

    in bin-of n end;

    fun numeral-tr [Free (str, -)] = mk-bintype (the (Int.fromString str))
      | numeral-tr ts = raise TERM (numeral-tr, ts);

    in [(syntax-const ⟨-NumeralType⟩, K numeral-tr)] end
  >

print-translation <
  let
    fun int-of [] = 0
      | int-of (b :: bs) = b + 2 * int-of bs;

    fun bin-of (Const (type-syntax ⟨num0⟩, -)) = []
      | bin-of (Const (type-syntax ⟨num1⟩, -)) = [1]
      | bin-of (Const (type-syntax ⟨bit0⟩, -) $ bs) = 0 :: bin-of bs
      | bin-of (Const (type-syntax ⟨bit1⟩, -) $ bs) = 1 :: bin-of bs
      | bin-of t = raise TERM (bin-of, [t]);

    fun bit-tr' b [t] =
      let
        val rev-digs = b :: bin-of t handle TERM - => raise Match
        val i = int-of rev-digs;
        val num = string-of-int (abs i);
      in
        Syntax.const syntax-const ⟨-NumeralType⟩ $ Syntax.free num
      end
      | bit-tr' b - = raise Match;
  in
    [(type-syntax ⟨bit0⟩, K (bit-tr' 0)),
     (type-syntax ⟨bit1⟩, K (bit-tr' 1))]
  end
  >

```

71.8 Examples

```

lemma CARD(0) = 0 by simp
lemma CARD(17) = 17 by simp
lemma CHAR(23) = 23 by simp
lemma 8 * 11 ^ 3 - 6 = (2::5) by simp

end

```

72 ω -words

```

theory Omega-Words-Fun

```

```

imports Infinite-Set
begin

```

Note: This theory is based on Stefan Merz’s work.

Automata recognize languages, which are sets of words. For the theory of ω -automata, we are mostly interested in ω -words, but it is sometimes useful to reason about finite words, too. We are modeling finite words as lists; this lets us benefit from the existing library. Other formalizations could be investigated, such as representing words as functions whose domains are initial intervals of the natural numbers.

72.1 Type declaration and elementary operations

We represent ω -words as functions from the natural numbers to the alphabet type. Other possible formalizations include a coinductive definition or a uniform encoding of finite and infinite words, as studied by Müller et al.

type-synonym

$'a \text{ word} = \text{nat} \Rightarrow 'a$

We can prefix a finite word to an ω -word, and a way to obtain an ω -word from a finite, non-empty word is by ω -iteration.

definition

$\text{conc} :: ['a \text{ list}, 'a \text{ word}] \Rightarrow 'a \text{ word}$ (**infixr** $\langle \frown \rangle$ 65)
where $w \frown x == \lambda n. \text{if } n < \text{length } w \text{ then } w!n \text{ else } x (n - \text{length } w)$

definition

$\text{iter} :: 'a \text{ list} \Rightarrow 'a \text{ word}$ ($\langle \langle \text{notation} = \langle \text{postfix } \omega \rangle \rangle^\omega \rangle$ [1000])
where $\text{iter } w == \text{if } w = [] \text{ then undefined else } (\lambda n. w!(n \bmod (\text{length } w)))$

lemma $\text{conc-empty}[\text{simp}]$: $[] \frown w = w$

unfolding conc-def **by** *auto*

lemma $\text{conc-fst}[\text{simp}]$: $n < \text{length } w \implies (w \frown x) n = w!n$

by ($\text{simp add: conc-def}$)

lemma $\text{conc-snd}[\text{simp}]$: $\neg(n < \text{length } w) \implies (w \frown x) n = x (n - \text{length } w)$

by ($\text{simp add: conc-def}$)

lemma $\text{iter-nth}[\text{simp}]$: $0 < \text{length } w \implies w^\omega n = w!(n \bmod (\text{length } w))$

by ($\text{simp add: iter-def}$)

lemma $\text{conc-conc}[\text{simp}]$: $u \frown v \frown w = (u @ v) \frown w$ (**is** $?lhs = ?rhs$)

proof

fix n

have u : $n < \text{length } u \implies ?lhs n = ?rhs n$

by ($\text{simp add: conc-def nth-append}$)

have v : $\neg(n < \text{length } u); n < \text{length } u + \text{length } v \implies ?lhs n = ?rhs n$

by ($\text{simp add: conc-def nth-append, arith}$)

have w : $\neg(n < \text{length } u + \text{length } v) \implies ?lhs n = ?rhs n$

by ($\text{simp add: conc-def nth-append, arith}$)

```

from  $u\ v\ w$  show  $?lhs\ n = ?rhs\ n$  by blast
qed

lemma range-conc[simp]:  $range\ (w_1 \frown w_2) = set\ w_1 \cup range\ w_2$ 
proof (intro equalityI subsetI)
  fix  $a$ 
  assume  $a \in range\ (w_1 \frown w_2)$ 
  then obtain  $i$  where  $1: a = (w_1 \frown w_2)\ i$  by auto
  then show  $a \in set\ w_1 \cup range\ w_2$ 
    unfolding  $1$  by (cases  $i < length\ w_1$ ) simp-all
next
  fix  $a$ 
  assume  $a: a \in set\ w_1 \cup range\ w_2$ 
  then show  $a \in range\ (w_1 \frown w_2)$ 
  proof
    assume  $a \in set\ w_1$ 
    then obtain  $i$  where  $1: i < length\ w_1\ a = w_1\ !\ i$ 
      using in-set-conv-nth by metis
    show ?thesis
  proof
    show  $a = (w_1 \frown w_2)\ i$  using  $1$  by auto
    show  $i \in UNIV$  by rule
  qed
next
  assume  $a \in range\ w_2$ 
  then obtain  $i$  where  $1: a = w_2\ i$  by auto
  show ?thesis
  proof
    show  $a = (w_1 \frown w_2)\ (length\ w_1 + i)$  using  $1$  by simp
    show  $length\ w_1 + i \in UNIV$  by rule
  qed
qed
qed

```

```

lemma iter-unroll:  $0 < length\ w \implies w^\omega = w \frown w^\omega$ 
by (simp add: fun-eq-iff mod-iff)

```

72.2 Subsequence, Prefix, and Suffix

```

definition suffix ::  $[nat, 'a\ word] \Rightarrow 'a\ word$ 
where suffix  $k\ x \equiv \lambda n. x\ (k+n)$ 

```

```

definition subsequence ::  $'a\ word \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list$ 
  ( $\langle \langle open\text{-}block\ notation = \langle mixfix\ subsequence \rangle \rangle \cdot [- \rightarrow -] \rangle \cdot 900$ )
where subsequence  $w\ i\ j \equiv map\ w\ [i..<j]$ 

```

```

abbreviation prefix ::  $nat \Rightarrow 'a\ word \Rightarrow 'a\ list$ 
where prefix  $n\ w \equiv subsequence\ w\ 0\ n$ 

```

lemma *suffix-nth* [*simp*]: (*suffix* *k* *x*) *n* = *x* (*k*+*n*)
by (*simp add: suffix-def*)

lemma *suffix-0* [*simp*]: *suffix* 0 *x* = *x*
by (*simp add: suffix-def*)

lemma *suffix-suffix* [*simp*]: *suffix* *m* (*suffix* *k* *x*) = *suffix* (*k*+*m*) *x*
by (*rule ext*) (*simp add: suffix-def add.assoc*)

lemma *subsequence-append*: *prefix* (*i* + *j*) *w* = *prefix* *i* *w* @ (*w* [*i* → *i* + *j*])
unfolding *map-append* [*symmetric*] *upt-add-eq-append* [*OF le0*] *subsequence-def* ..

lemma *subsequence-drop* [*simp*]: *drop* *i* (*w* [*j* → *k*]) = *w* [*j* + *i* → *k*]
by (*simp add: subsequence-def drop-map*)

lemma *subsequence-empty* [*simp*]: *w* [*i* → *j*] = [] \longleftrightarrow *j* ≤ *i*
by (*auto simp add: subsequence-def*)

lemma *subsequence-length* [*simp*]: *length* (*subsequence* *w* *i* *j*) = *j* - *i*
by (*simp add: subsequence-def*)

lemma *subsequence-nth* [*simp*]: *k* < *j* - *i* \implies (*w* [*i* → *j*]) ! *k* = *w* (*i* + *k*)
unfolding *subsequence-def*
by *auto*

lemma *subseq-to-zero* [*simp*]: *w* [*i* → 0] = []
by *simp*

lemma *subseq-to-smaller* [*simp*]: *i* ≥ *j* \implies *w* [*i* → *j*] = []
by *simp*

lemma *subseq-to-Suc* [*simp*]: *i* ≤ *j* \implies *w* [*i* → *Suc* *j*] = *w* [*i* → *j*] @ [*w* *j*]
by (*auto simp: subsequence-def*)

lemma *subsequence-singleton* [*simp*]: *w* [*i* → *Suc* *i*] = [*w* *i*]
by (*auto simp: subsequence-def*)

lemma *subsequence-prefix-suffix*: *prefix* (*j* - *i*) (*suffix* *i* *w*) = *w* [*i* → *j*]

proof (*cases* *i* ≤ *j*)

case *True*

have *w* [*i* → *j*] = *map* *w* (*map* ($\lambda n. n + i$) [*0*..*j* - *i*])

unfolding *map-add-upt* *subsequence-def*

using *le-add-diff-inverse2* [*OF True*] **by** *force*

also

have ... = *map* ($\lambda n. w$ (*n* + *i*)) [*0*..*j* - *i*]

unfolding *map-map comp-def* **by** *blast*

finally

```

  show ?thesis
    unfolding subsequence-def suffix-def add.commute[of i] by simp
next
  case False
  then show ?thesis
    by (simp add: subsequence-def)
qed

lemma prefix-suffix:  $x = \text{prefix } n \ x \smallfrown (\text{suffix } n \ x)$ 
  by (rule ext) (simp add: subsequence-def conc-def)

declare prefix-suffix[symmetric, simp]

lemma word-split: obtains  $v_1 \ v_2$  where  $v = v_1 \smallfrown v_2$  length  $v_1 = k$ 
proof
  show  $v = \text{prefix } k \ v \smallfrown \text{suffix } k \ v$ 
    by (rule prefix-suffix)
  show length (prefix  $k \ v$ ) =  $k$ 
    by simp
qed

lemma set-subsequence[simp]:  $\text{set } (w[i \rightarrow j]) = w\{i..<j\}$ 
  unfolding subsequence-def by auto

lemma subsequence-take[simp]:  $\text{take } i \ (w [j \rightarrow k]) = w [j \rightarrow \min (j + i) \ k]$ 
  by (simp add: subsequence-def take-map min-def)

lemma subsequence-shift[simp]:  $(\text{suffix } i \ w) [j \rightarrow k] = w [i + j \rightarrow i + k]$ 
  by (metis add-diff-cancel-left subsequence-prefix-suffix suffix-suffix)

lemma suffix-subseq-join[simp]:  $i \leq j \implies v [i \rightarrow j] \smallfrown \text{suffix } j \ v = \text{suffix } i \ v$ 
  by (metis (no-types, lifting) Nat.add-0-right le-add-diff-inverse prefix-suffix
    subsequence-shift suffix-suffix)

lemma prefix-conc-fst[simp]:
  assumes  $j \leq \text{length } w$ 
  shows  $\text{prefix } j \ (w \smallfrown w') = \text{take } j \ w$ 
proof -
  have  $\forall i < j. (\text{prefix } j \ (w \smallfrown w')) ! i = (\text{take } j \ w) ! i$ 
    using assms by (simp add: conc-fst subsequence-def)
  thus ?thesis
    by (simp add: assms list-eq-iff-nth-eq min.absorb2)
qed

lemma prefix-conc-snd[simp]:
  assumes  $n \geq \text{length } u$ 
  shows  $\text{prefix } n \ (u \smallfrown v) = u @ \text{prefix } (n - \text{length } u) \ v$ 

```

```

proof (intro nth-equalityI)
  show length (prefix n (u  $\frown$  v)) = length (u @ prefix (n - length u) v)
    using assms by simp
  fix i
  assume i < length (prefix n (u  $\frown$  v))
  then show prefix n (u  $\frown$  v) ! i = (u @ prefix (n - length u) v) ! i
    by (cases i < length u) (auto simp: nth-append)
qed

```

```

lemma prefix-conc-length[simp]: prefix (length w) (w  $\frown$  w') = w
  by simp

```

```

lemma suffix-conc-fst[simp]:
  assumes n  $\leq$  length u
  shows suffix n (u  $\frown$  v) = drop n u  $\frown$  v
proof
  show suffix n (u  $\frown$  v) i = (drop n u  $\frown$  v) i for i
    using assms by (cases n + i < length u) (auto simp: algebra-simps)
qed

```

```

lemma suffix-conc-snd[simp]:
  assumes n  $\geq$  length u
  shows suffix n (u  $\frown$  v) = suffix (n - length u) v
proof
  show suffix n (u  $\frown$  v) i = suffix (n - length u) v i for i
    using assms by simp
qed

```

```

lemma suffix-conc-length[simp]: suffix (length w) (w  $\frown$  w') = w'
  unfolding conc-def by force

```

```

lemma concat-eq[iff]:
  assumes length v1 = length v2
  shows v1  $\frown$  u1 = v2  $\frown$  u2  $\longleftrightarrow$  v1 = v2  $\wedge$  u1 = u2
    (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  then have 1: (v1  $\frown$  u1) i = (v2  $\frown$  u2) i for i by auto
  show ?rhs
  proof (intro conjI ext nth-equalityI)
    show length v1 = length v2 by (rule assms(1))
  next
    fix i
    assume 2: i < length v1
    have 3: i < length v2 using assms(1) 2 by simp
    show v1 ! i = v2 ! i using 1[of i] 2 3 by simp
  next
    show u1 i = u2 i for i
      using 1[of length v1 + i] assms(1) by simp

```

```

qed
next
  assume ?rhs
  then show ?lhs by simp
qed

lemma same-concat-eq[iff]:  $u \frown v = u \frown w \longleftrightarrow v = w$ 
  by simp

lemma comp-concat[simp]:  $f \circ u \frown v = \text{map } f \, u \frown (f \circ v)$ 
proof
  fix i
  show  $(f \circ u \frown v) \, i = (\text{map } f \, u \frown (f \circ v)) \, i$ 
    by (cases  $i < \text{length } u$ ) simp-all
qed

```

72.3 Prepending

```

primrec build :: 'a  $\Rightarrow$  'a word  $\Rightarrow$  'a word (infixr <##> 65)
  where  $(a \, \#\# \, w) \, 0 = a \mid (a \, \#\# \, w) \, (\text{Suc } i) = w \, i$ 

lemma build-eq[iff]:  $a_1 \, \#\# \, w_1 = a_2 \, \#\# \, w_2 \longleftrightarrow a_1 = a_2 \wedge w_1 = w_2$ 
proof
  assume 1:  $a_1 \, \#\# \, w_1 = a_2 \, \#\# \, w_2$ 
  have 2:  $(a_1 \, \#\# \, w_1) \, i = (a_2 \, \#\# \, w_2) \, i$  for i
    using 1 by auto
  show  $a_1 = a_2 \wedge w_1 = w_2$ 
  proof (intro conjI ext)
    show  $a_1 = a_2$ 
      using 2[of 0] by simp
    show  $w_1 \, i = w_2 \, i$  for i
      using 2[of Suc i] by simp
  qed
next
  assume 1:  $a_1 = a_2 \wedge w_1 = w_2$ 
  show  $a_1 \, \#\# \, w_1 = a_2 \, \#\# \, w_2$  using 1 by simp
qed

lemma build-cons[simp]:  $(a \, \# \, u) \frown v = a \, \#\# \, u \frown v$ 
proof
  fix i
  show  $((a \, \# \, u) \frown v) \, i = (a \, \#\# \, u \frown v) \, i$ 
  proof (cases i)
    case 0
    show ?thesis unfolding 0 by simp
  next
    case (Suc j)
    show ?thesis unfolding Suc by (cases  $j < \text{length } u$ , simp+)
  qed
qed

```


qed

lemma *build-append[simp]*: $(w @ a \# u) \frown v = w \frown a \#\# u \frown v$
unfolding *conc-conc[symmetric]* **by** *simp*

lemma *build-first[simp]*: $w \ 0 \#\# \text{suffix} (\text{Suc } 0) \ w = w$

proof

show $(w \ 0 \#\# \text{suffix} (\text{Suc } 0) \ w) \ i = w \ i$ **for** i
by $(\text{cases } i) \ \text{simp-all}$

qed

lemma *build-split[intro]*: $w = w \ 0 \#\# \text{suffix } 1 \ w$
by *simp*

lemma *build-range[simp]*: $\text{range} (a \#\# w) = \text{insert } a (\text{range } w)$

proof *safe*

show $(a \#\# w) \ i \notin \text{range } w \implies (a \#\# w) \ i = a$ **for** i
by $(\text{cases } i) \ \text{auto}$

show $a \in \text{range} (a \#\# w)$

proof (rule range-eqI)

show $a = (a \#\# w) \ 0$ **by** *simp*

qed

show $w \ i \in \text{range} (a \#\# w)$ **for** i

proof (rule range-eqI)

show $w \ i = (a \#\# w) (\text{Suc } i)$ **by** *simp*

qed

qed

lemma *suffix-singleton-suffix[simp]*: $w \ i \#\# \text{suffix} (\text{Suc } i) \ w = \text{suffix } i \ w$
using *suffix-subseq-join[of i Suc i w]*
by *simp*

Find the first occurrence of a letter from a given set

lemma *word-first-split-set*:

assumes $A \cap \text{range } w \neq \{\}$

obtains $u \ a \ v$ **where** $w = u \frown [a] \frown v$ $A \cap \text{set } u = \{\}$ $a \in A$

proof –

define i **where** $i = (\text{LEAST } i. \ w \ i \in A)$

show *?thesis*

proof

show $w = \text{prefix } i \ w \frown [w \ i] \frown \text{suffix} (\text{Suc } i) \ w$

by *simp*

show $A \cap \text{set} (\text{prefix } i \ w) = \{\}$

apply *safe*

subgoal premises *prems* **for** a

proof –

from *prems* **obtain** k **where** $\exists: k < i \ w \ k = a$

by *auto*

have $\neg: w \ k \notin A$

```

      using not-less-Least 3(1) unfolding i-def .
    show ?thesis
      using prems(1) 3(2) 4 by auto
  qed
done
show w i ∈ A
  using LeastI assms(1) unfolding i-def by fast
qed
qed

```

72.4 The limit set of an ω -word

The limit set (also called infinity set) of an ω -word is the set of letters that appear infinitely often in the word. This set plays an important role in defining acceptance conditions of ω -automata.

definition *limit* :: 'a word \Rightarrow 'a set
where *limit* $x \equiv \{a . \exists_{\infty} n . x\ n = a\}$

lemma *limit-iff-frequent*: $a \in \text{limit } x \longleftrightarrow (\exists_{\infty} n . x\ n = a)$
by (*simp add: limit-def*)

The following is a different way to define the limit, using the reverse image, making the laws about reverse image applicable to the limit set. (Might want to change the definition above?)

lemma *limit-vimage*: $(a \in \text{limit } x) = \text{infinite } (x - \{a\})$
by (*simp add: limit-def Inf-many-def vimage-def*)

lemma *two-in-limit-iff*:

$(\{a, b\} \subseteq \text{limit } x) =$
 $((\exists n. x\ n = a) \wedge (\forall n. x\ n = a \longrightarrow (\exists m > n. x\ m = b)) \wedge (\forall m. x\ m = b \longrightarrow (\exists n > m. x\ n = a)))$
(is ?lhs = (?r1 \wedge ?r2 \wedge ?r3))

proof

assume *lhs*: ?lhs

hence 1: ?r1 **by** (*auto simp: limit-def elim: INFM-EX*)

from *lhs* **have** $\forall n. \exists m > n. x\ m = b$ **by** (*auto simp: limit-def INFM-nat*)

hence 2: ?r2 **by** *simp*

from *lhs* **have** $\forall m. \exists n > m. x\ n = a$ **by** (*auto simp: limit-def INFM-nat*)

hence 3: ?r3 **by** *simp*

from 1 2 3 **show** ?r1 \wedge ?r2 \wedge ?r3 **by** *simp*

next

assume ?r1 \wedge ?r2 \wedge ?r3

hence 1: ?r1 **and** 2: ?r2 **and** 3: ?r3 **by** *simp+*

have *infa*: $\forall m. \exists n \geq m. x\ n = a$

proof

fix *m*

show $\exists n \geq m. x\ n = a$ **(is ?A m)**

proof (*induct m*)

```

    from 1 show ?A 0 by simp
  next
    fix m
    assume ih: ?A m
    then obtain n where n:  $n \geq m$   $x\ n = a$  by auto
    with 2 obtain k where k:  $k > n$   $x\ k = b$  by auto
    with 3 obtain l where l:  $l > k$   $x\ l = a$  by auto
    from n k l have  $l \geq \text{Suc } m$  by auto
    with l show ?A (Suc m) by auto
  qed
qed
hence infa':  $\exists_{\infty} n. x\ n = a$  by (simp add: INFM-nat-le)
have  $\forall n. \exists m > n. x\ m = b$ 
proof
  fix n
  from infa obtain k where k1:  $k \geq n$  and k2:  $x\ k = a$  by auto
  from 2 k2 obtain l where l1:  $l > k$  and l2:  $x\ l = b$  by auto
  from k1 l1 have  $l > n$  by auto
  with l2 show  $\exists m > n. x\ m = b$  by auto
qed
hence  $\exists_{\infty} m. x\ m = b$  by (simp add: INFM-nat)
with infa' show ?lhs by (auto simp: limit-def)
qed

```

For ω -words over a finite alphabet, the limit set is non-empty. Moreover, from some position onward, any such word contains only letters from its limit set.

```

lemma limit-nonempty:
  assumes fin: finite (range x)
  shows  $\exists a. a \in \text{limit } x$ 
proof -
  from fin obtain a where  $a \in \text{range } x \wedge \text{infinite } (x - \{a\})$ 
  by (rule inf-img-fin-domE) auto
  hence  $a \in \text{limit } x$ 
  by (auto simp add: limit-vimage)
  thus ?thesis ..
qed

```

lemmas $\text{limit-nonempty}E = \text{limit-nonempty}[THEN\ exE]$

```

lemma limit-inter-INF:
  assumes hyp:  $\text{limit } w \cap S \neq \{\}$ 
  shows  $\exists_{\infty} n. w\ n \in S$ 
proof -
  from hyp obtain x where  $\exists_{\infty} n. w\ n = x$  and  $x \in S$ 
  by (auto simp add: limit-def)
  thus ?thesis
  by (auto elim: INFM-mono)
qed

```

The reverse implication is true only if S is finite.

lemma *INF-limit-inter*:

assumes *hyp*: $\exists_{\infty} n. w\ n \in S$

and *fin*: *finite* ($S \cap \text{range } w$)

shows $\exists a. a \in \text{limit } w \cap S$

proof (*rule ccontr*)

assume *contra*: $\neg(\exists a. a \in \text{limit } w \cap S)$

hence $\forall a \in S. \text{finite } \{n. w\ n = a\}$

by (*auto simp add: limit-def Inf-many-def*)

with *fin* **have** *finite* ($\text{UN } a:S \cap \text{range } w. \{n. w\ n = a\}$)

by *auto*

moreover

have ($\text{UN } a:S \cap \text{range } w. \{n. w\ n = a\} = \{n. w\ n \in S\}$)

by *auto*

moreover

note *hyp*

ultimately show *False*

by (*simp add: Inf-many-def*)

qed

lemma *fin-ex-inf-eq-limit*: *finite* $A \implies (\exists_{\infty} i. w\ i \in A) \longleftrightarrow \text{limit } w \cap A \neq \{\}$

by (*metis INF-limit-inter equals0D finite-Int limit-inter-INF*)

lemma *limit-in-range-suffix*: *limit* $x \subseteq \text{range } (\text{suffix } k\ x)$

proof

fix a

assume $a \in \text{limit } x$

then obtain l **where**

$kl: k < l$ **and** $xl: x\ l = a$

by (*auto simp add: limit-def INFM-nat*)

from kl **obtain** m **where** $l = k+m$

by (*auto simp add: less-iff-Suc-add*)

with xl **show** $a \in \text{range } (\text{suffix } k\ x)$

by *auto*

qed

lemma *limit-in-range*: *limit* $r \subseteq \text{range } r$

using *limit-in-range-suffix*[*of* $r\ 0$] **by** *simp*

lemmas *limit-in-range-suffixD* = *limit-in-range-suffix*[*THEN* *subsetD*]

lemma *limit-subset*: *limit* $f \subseteq f\ \text{'}\{n..\}$

using *limit-in-range-suffix*[*of* $f\ n$] **unfolding** *suffix-def* **by** *auto*

theorem *limit-is-suffix*:

assumes *fin*: *finite* (*range* x)

shows $\exists k. \text{limit } x = \text{range } (\text{suffix } k\ x)$

proof –

have $\exists k. \text{range } (\text{suffix } k\ x) \subseteq \text{limit } x$

proof –

— The set of letters that are not in the limit is certainly finite.

from *fin* **have** *finite* (*range* *x* – *limit* *x*)

by *simp*

— Moreover, any such letter occurs only finitely often

moreover

have $\forall a \in \text{range } x - \text{limit } x. \text{finite } (x - \{a\})$

by (*auto simp add: limit-vimage*)

— Thus, there are only finitely many occurrences of such letters.

ultimately have *finite* ($\text{UN } a : \text{range } x - \text{limit } x. x - \{a\}$)

by (*blast intro: finite-UN-I*)

— Therefore these occurrences are within some initial interval.

then obtain *k* **where** ($\text{UN } a : \text{range } x - \text{limit } x. x - \{a\} \subseteq \{..<k\}$)

by (*blast dest: finite-nat-bounded*)

— This is just the bound we are looking for.

hence $\forall m. k \leq m \longrightarrow x m \in \text{limit } x$

by (*auto simp add: limit-vimage*)

hence *range* (*suffix* *k* *x*) $\subseteq \text{limit } x$

by *auto*

thus *?thesis* ..

qed

then obtain *k* **where** *range* (*suffix* *k* *x*) $\subseteq \text{limit } x$..

with *limit-in-range-suffix*

have *limit* *x* = *range* (*suffix* *k* *x*)

by (*rule subset-antisym*)

thus *?thesis* ..

qed

lemmas *limit-is-suffixE* = *limit-is-suffix*[*THEN exE*]

The limit set enjoys some simple algebraic laws with respect to concatenation, suffixes, iteration, and renaming.

theorem *limit-conc* [*simp*]: *limit* (*w* \frown *x*) = *limit* *x*

proof (*auto*)

fix *a* **assume** *a*: *a* $\in \text{limit } (w \frown x)$

have $\forall m. \exists n. m < n \wedge x n = a$

proof

fix *m*

from *a* **obtain** *n* **where** $m + \text{length } w < n \wedge (w \frown x) n = a$

by (*auto simp add: limit-def Inf-many-def infinite-nat-iff-unbounded*)

hence $m < n - \text{length } w \wedge x (n - \text{length } w) = a$

by (*auto simp add: conc-def*)

thus $\exists n. m < n \wedge x n = a$..

qed

hence *infinite* $\{n. x n = a\}$

by (*simp add: infinite-nat-iff-unbounded*)

thus *a* $\in \text{limit } x$

by (*simp add: limit-def Inf-many-def*)

next

fix a **assume** $a: a \in \text{limit } x$
have $\forall m. \text{length } w < m \longrightarrow (\exists n. m < n \wedge (w \smallfrown x) \restriction n = a)$
proof (*clarify*)
 fix m
 assume $m: \text{length } w < m$
 with a **obtain** n **where** $m - \text{length } w < n \wedge x \restriction n = a$
 by (*auto simp add: limit-def Inf-many-def infinite-nat-iff-unbounded*)
 with m **have** $m < n + \text{length } w \wedge (w \smallfrown x) \restriction (n + \text{length } w) = a$
 by (*simp add: conc-def, arith*)
 thus $\exists n. m < n \wedge (w \smallfrown x) \restriction n = a \dots$
qed
hence $\text{infinite } \{n. (w \smallfrown x) \restriction n = a\}$
 by (*simp add: unbounded-k-infinite*)
thus $a \in \text{limit } (w \smallfrown x)$
 by (*simp add: limit-def Inf-many-def*)
qed

theorem *limit-suffix* [*simp*]: $\text{limit } (\text{suffix } n \ x) = \text{limit } x$
proof –
 have $x = (\text{prefix } n \ x) \smallfrown (\text{suffix } n \ x)$
 by (*simp add: prefix-suffix*)
hence $\text{limit } x = \text{limit } (\text{prefix } n \ x \smallfrown \text{suffix } n \ x)$
 by *simp*
also have $\dots = \text{limit } (\text{suffix } n \ x)$
 by (*rule limit-conc*)
finally show *?thesis*
 by (*rule sym*)
qed

theorem *limit-iter* [*simp*]:
 assumes *nempty*: $0 < \text{length } w$
 shows $\text{limit } w^\omega = \text{set } w$
proof
 have $\text{limit } w^\omega \subseteq \text{range } w^\omega$
 by (*auto simp add: limit-def dest: INFM-EX*)
also from *nempty* **have** $\dots \subseteq \text{set } w$
 by *auto*
finally show $\text{limit } w^\omega \subseteq \text{set } w$.
next
 {
 fix a **assume** $a: a \in \text{set } w$
 then obtain k **where** $k: k < \text{length } w \wedge w \restriction k = a$
 by (*auto simp add: set-conv-nth*)
 — the following bound is terrible, but it simplifies the proof
 from *nempty* k **have** $\forall m. w^\omega \restriction ((\text{Suc } m) * (\text{length } w) + k) = a$
 by (*simp add: mod-add-left-eq [symmetric]*)
 moreover
 — why is the following so hard to prove??
 have $\forall m. m < (\text{Suc } m) * (\text{length } w) + k$

```

proof
  fix  $m$ 
  from nempty have  $1 \leq \text{length } w$  by arith
  hence  $m*1 \leq m*\text{length } w$  by simp
  hence  $m \leq m*\text{length } w$  by simp
  with nempty have  $m < \text{length } w + (m*\text{length } w) + k$  by arith
  thus  $m < (\text{Suc } m)*(\text{length } w) + k$  by simp
qed
moreover note nempty
ultimately have  $a \in \text{limit } w^\omega$ 
  by (auto simp add: limit-iff-frequent INFM-nat)
}
then show  $\text{set } w \subseteq \text{limit } w^\omega$  by auto
qed

```

```

lemma limit-o [simp]:
  assumes  $a: a \in \text{limit } w$ 
  shows  $f a \in \text{limit } (f \circ w)$ 
proof —
  from  $a$ 
  have  $\exists_\infty n. w\ n = a$ 
    by (simp add: limit-iff-frequent)
  hence  $\exists_\infty n. f\ (w\ n) = f\ a$ 
    by (rule INFM-mono, simp)
  thus  $f a \in \text{limit } (f \circ w)$ 
    by (simp add: limit-iff-frequent)
qed

```

The converse relation is not true in general: $f(a)$ can be in the limit of $f \circ w$ even though a is not in the limit of w . However, *limit* commutes with renaming if the function is injective. More generally, if $f(a)$ is the image of only finitely many elements, some of these must be in the limit of w .

```

lemma limit-o-inv:
  assumes  $\text{fin}: \text{finite } (f - \{x\})$ 
  and  $x: x \in \text{limit } (f \circ w)$ 
  shows  $\exists a \in (f - \{x\}). a \in \text{limit } w$ 
proof (rule ccontr)
  assume contra:  $\neg ?thesis$ 
  — hence, every element in the pre-image occurs only finitely often
  then have  $\forall a \in (f - \{x\}). \text{finite } \{n. w\ n = a\}$ 
    by (simp add: limit-def Inf-many-def)
  — so there are only finitely many occurrences of any such element
  with fin have  $\text{finite } (\bigcup a \in (f - \{x\}). \{n. w\ n = a\})$ 
    by auto
  — these are precisely those positions where  $x$  occurs in  $f \circ w$ 
  moreover
  have  $(\bigcup a \in (f - \{x\}). \{n. w\ n = a\}) = \{n. f(w\ n) = x\}$ 
    by auto
  ultimately

```

— so x can occur only finitely often in the translated word
have $\text{finite } \{n. f(w\ n) = x\}$
 by *simp*
 — ... which yields a contradiction
with x **show** *False*
 by (*simp add: limit-def Inf-many-def*)
qed

theorem *limit-inj [simp]*:
 assumes *inj*: $\text{inj } f$
 shows $\text{limit } (f \circ w) = f \text{ ` } (\text{limit } w)$
proof
 show $f \text{ ` } \text{limit } w \subseteq \text{limit } (f \circ w)$
 by *auto*
 show $\text{limit } (f \circ w) \subseteq f \text{ ` } \text{limit } w$
proof
 fix x
 assume $x: x \in \text{limit } (f \circ w)$
 from *inj* **have** $\text{finite } (f \text{ - ` } \{x\})$
 by (*blast intro: finite-vimageI*)
 with x **obtain** a **where** $a: a \in (f \text{ - ` } \{x\}) \wedge a \in \text{limit } w$
 by (*blast dest: limit-o-inv*)
 thus $x \in f \text{ ` } (\text{limit } w)$
 by *auto*
qed
qed

lemma *limit-inter-empty*:
 assumes *fin*: $\text{finite } (\text{range } w)$
 assumes *hyp*: $\text{limit } w \cap S = \{\}$
 shows $\forall_{\infty} n. w\ n \notin S$
proof —
 from *fin* **obtain** k **where** $k\text{-def}: \text{limit } w = \text{range } (\text{suffix } k\ w)$
 using *limit-is-suffix* **by** *blast*
 have $w\ (k + k') \notin S$ **for** k'
 using *hyp* **unfolding** $k\text{-def}$ suffix-def image-def **by** *blast*
 thus *?thesis*
unfolding *MOST-nat-le* **using** *le-Suc-ex* **by** *blast*
qed

If the limit is the suffix of the sequence’s range, we may increase the suffix index arbitrarily

lemma *limit-range-suffix-incr*:
 assumes $\text{limit } r = \text{range } (\text{suffix } i\ r)$
 assumes $j \geq i$
 shows $\text{limit } r = \text{range } (\text{suffix } j\ r)$
 (*is ?lhs = ?rhs*)
proof —
 have *?lhs* = $\text{range } (\text{suffix } i\ r)$


```

    using assms by simp
  moreover
  have ...  $\supseteq$  ?rhs using  $\langle j \geq i \rangle$ 
    by (metis (mono-tags, lifting) assms(2)
        image-subsetI le-Suc-ex range-eqI suffix-def suffix-suffix)
  moreover
  have ...  $\supseteq$  ?lhs by (rule limit-in-range-suffix)
  ultimately
  show ?lhs = ?rhs
    by (metis antisym-conv limit-in-range-suffix)
qed

```

For two finite sequences, we can find a common suffix index such that the limits can be represented as these suffixes’ ranges.

lemma *common-range-limit*:

```

  assumes finite (range x)
    and finite (range y)
  obtains i where limit x = range (suffix i x)
    and limit y = range (suffix i y)
proof -
  obtain i j where 1: limit x = range (suffix i x)
    and 2: limit y = range (suffix j y)
  using assms limit-is-suffix by metis
  have limit x = range (suffix (max i j) x)
    and limit y = range (suffix (max i j) y)
  using limit-range-suffix-incr[OF 1] limit-range-suffix-incr[OF 2]
  by auto
  thus ?thesis
    using that by metis
qed

```

72.5 Index sequences and piecewise definitions

A word can be defined piecewise: given a sequence of words w_0, w_1, \dots and a strictly increasing sequence of integers i_0, i_1, \dots where $i_0 = 0$, a single word is obtained by concatenating subwords of the w_n as given by the integers: the resulting word is

$$(w_0)_{i_0} \dots (w_0)_{i_1-1} (w_1)_{i_1} \dots (w_1)_{i_2-1} \dots$$

We prepare the field by proving some trivial facts about such sequences of indexes.

definition *idx-sequence* :: *nat word* \Rightarrow *bool*

where *idx-sequence* *idx* \equiv (*idx* 0 = 0) \wedge ($\forall n. \text{idx } n < \text{idx } (\text{Suc } n)$)

lemma *idx-sequence-less*:

```

  assumes iseq: idx-sequence idx
  shows idx n < idx (Suc(n+k))

```

```

proof (induct k)
  from iseq show  $\text{idx } n < \text{idx } (\text{Suc } (n + 0))$ 
    by (simp add: idx-sequence-def)
next
  fix k
  assume ih:  $\text{idx } n < \text{idx } (\text{Suc}(n+k))$ 
  from iseq have  $\text{idx } (\text{Suc}(n+k)) < \text{idx } (\text{Suc}(n + \text{Suc } k))$ 
    by (simp add: idx-sequence-def)
  with ih show  $\text{idx } n < \text{idx } (\text{Suc}(n + \text{Suc } k))$ 
    by (rule less-trans)
qed

```

```

lemma idx-sequence-inj:
  assumes iseq: idx-sequence idx
    and eq:  $\text{idx } m = \text{idx } n$ 
  shows  $m = n$ 
proof (cases m n rule: linorder-cases)
  case greater
  then obtain k where  $m = \text{Suc}(n+k)$ 
    by (auto simp add: less-iff-Suc-add)
  with iseq have  $\text{idx } n < \text{idx } m$ 
    by (simp add: idx-sequence-less)
  with eq show ?thesis
    by simp
next
  case less
  then obtain k where  $n = \text{Suc}(m+k)$ 
    by (auto simp add: less-iff-Suc-add)
  with iseq have  $\text{idx } m < \text{idx } n$ 
    by (simp add: idx-sequence-less)
  with eq show ?thesis
    by simp
qed

```

```

lemma idx-sequence-mono:
  assumes iseq: idx-sequence idx
    and m:  $m \leq n$ 
  shows  $\text{idx } m \leq \text{idx } n$ 
proof (cases m=n)
  case True
  thus ?thesis by simp
next
  case False
  with m have  $m < n$  by simp
  then obtain k where  $n = \text{Suc}(m+k)$ 
    by (auto simp add: less-iff-Suc-add)
  with iseq have  $\text{idx } m < \text{idx } n$ 
    by (simp add: idx-sequence-less)
  thus ?thesis by simp

```

qed

Given an index sequence, every natural number is contained in the interval defined by two adjacent indexes, and in fact this interval is determined uniquely.

lemma *idx-sequence-idx*:
assumes *idx-sequence idx*
shows $idx\ k \in \{idx\ k \ ..< \ idx\ (Suc\ k)\}$
using *assms* **by** (*auto simp add: idx-sequence-def*)

lemma *idx-sequence-interval*:
assumes *iseq: idx-sequence idx*
shows $\exists k. n \in \{idx\ k \ ..< \ idx\ (Suc\ k)\}$
 (**is** *?P n is* $\exists k. \ ?in\ n\ k$)
proof (*induct n*)
from *iseq* **have** $0 = idx\ 0$
by (*simp add: idx-sequence-def*)
moreover
from *iseq* **have** $idx\ 0 \in \{idx\ 0 \ ..< \ idx\ (Suc\ 0)\}$
by (*rule idx-sequence-idx*)
ultimately
show *?P 0* **by** *auto*
next
fix *n*
assume *?P n*
then obtain *k* **where** *k: ?in n k ..*
show *?P (Suc n)*
proof (*cases Suc n < idx (Suc k)*)
case *True*
with *k* **have** *?in (Suc n) k*
by *simp*
thus *?thesis ..*
next
case *False*
with *k* **have** $Suc\ n = idx\ (Suc\ k)$
by *auto*
with *iseq* **have** *?in (Suc n) (Suc k)*
by (*simp add: idx-sequence-def*)
thus *?thesis ..*
qed
qed

lemma *idx-sequence-interval-unique*:
assumes *iseq: idx-sequence idx*
and $k: n \in \{idx\ k \ ..< \ idx\ (Suc\ k)\}$
and $m: n \in \{idx\ m \ ..< \ idx\ (Suc\ m)\}$
shows $k = m$
proof (*cases k m rule: linorder-cases*)
case *less*

```

hence  $Suc\ k \leq m$  by simp
with isec have  $idx\ (Suc\ k) \leq idx\ m$ 
  by (rule idx-sequence-mono)
with m have  $idx\ (Suc\ k) \leq n$ 
  by auto
with k have False
  by simp
thus ?thesis ..

```

next

```

case greater
hence  $Suc\ m \leq k$  by simp
with isec have  $idx\ (Suc\ m) \leq idx\ k$ 
  by (rule idx-sequence-mono)
with k have  $idx\ (Suc\ m) \leq n$ 
  by auto
with m have False
  by simp
thus ?thesis ..

```

qed

lemma *idx-sequence-unique-interval*:

```

assumes isec: idx-sequence idx
shows  $\exists! k. n \in \{idx\ k \dots < idx\ (Suc\ k)\}$ 
proof (rule ex-ex1I)
  from isec show  $\exists k. n \in \{idx\ k \dots < idx\ (Suc\ k)\}$ 
    by (rule idx-sequence-interval)

```

next

```

fix k y
assume  $n \in \{idx\ k \dots < idx\ (Suc\ k)\}$  and  $n \in \{idx\ y \dots < idx\ (Suc\ y)\}$ 
with isec show  $k = y$  by (auto elim: idx-sequence-interval-unique)

```

qed

Now we can define the piecewise construction of a word using an index sequence.

definition *merge* :: 'a word word \Rightarrow nat word \Rightarrow 'a word

where *merge* *ws* *idx* $\equiv \lambda n. \text{let } i = \text{THE } i. n \in \{idx\ i \dots < idx\ (Suc\ i)\} \text{ in } ws\ i\ n$

lemma *merge*:

```

assumes idx: idx-sequence idx
and n:  $n \in \{idx\ i \dots < idx\ (Suc\ i)\}$ 
shows merge ws idx n = ws i n
proof -
  from n have  $(\text{THE } k. n \in \{idx\ k \dots < idx\ (Suc\ k)\}) = i$ 
    by (rule the-equality[OF - sym[OF idx-sequence-interval-unique[OF idx n]]])
  simp
  thus ?thesis
    by (simp add: merge-def Let-def)

```

qed

```

lemma merge0:
  assumes idx: idx-sequence idx
  shows merge ws idx 0 = ws 0 0
proof (rule merge[OF idx])
  from idx have idx 0 < idx (Suc 0)
    unfolding idx-sequence-def by blast
  with idx show 0 ∈ {idx 0 ..< idx (Suc 0)}
    by (simp add: idx-sequence-def)
qed

lemma merge-Suc:
  assumes idx: idx-sequence idx
    and n: n ∈ {idx i ..< idx (Suc i)}
  shows merge ws idx (Suc n) = (if Suc n = idx (Suc i) then ws (Suc i) else ws
i) (Suc n)
proof auto
  assume eq: Suc n = idx (Suc i)
  from idx have idx (Suc i) < idx (Suc (Suc i))
    unfolding idx-sequence-def by blast
  with eq idx show merge ws idx (idx (Suc i)) = ws (Suc i) (idx (Suc i))
    by (simp add: merge)
next
  assume neq: Suc n ≠ idx (Suc i)
  with n have Suc n ∈ {idx i ..< idx (Suc i) }
    by auto
  with idx show merge ws idx (Suc n) = ws i (Suc n)
    by (rule merge)
qed

end

```

73 Combinator syntax for generic, open state monads (single-threaded monads)

```

theory Open-State-Syntax
imports Main
begin

context
  includes state-combinator-syntax
begin

```

73.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is

transformed single-threadedly).

To enter from the Haskell world, https://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

73.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

“yielding” transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, “yielding” a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and $\alpha, \beta, \gamma, \dots$ for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ , the former value should not be used again. To achieve this, we use a set of monad combinators:

Given two transformations f and g , they may be directly composed using the $(\circ>)$ combinator, forming a forward composition: $(f \circ> g) s = f (g s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the $(\circ\rightarrow)$ combinator: $(f \circ\rightarrow (\lambda x. g)) s = (let (x, s') = f s in g s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *return* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.
- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units (“open monad”).
- The type of states may change due to a transformation.

73.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemmas *monad-simp* = *Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse* = *monad-simp fcomp-apply scomp-apply split-beta*

end

73.4 Do-syntax

nonterminal *sdo-binds* **and** *sdo-bind*

syntax

-*sdo-block* :: *sdo-binds* \Rightarrow '*a*
 ($\langle \langle \text{open-block notation} = \langle \text{mixfix exec block} \rangle \rangle \text{exec } \{ // (2 \text{ -}) // \} \rangle$ [12] 62)
 -*sdo-bind* :: [*pttrn*, '*a*] \Rightarrow *sdo-bind*
 ($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix exec bind} \rangle \rangle - \leftarrow / - \rangle$ 13)
 -*sdo-let* :: [*pttrn*, '*a*] \Rightarrow *sdo-bind*
 ($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix exec let} \rangle \rangle \text{let } - = / - \rangle$ [1000, 13] 13)
 -*sdo-then* :: '*a* \Rightarrow *sdo-bind* ($\langle - \rangle$ [14] 13)
 -*sdo-final* :: '*a* \Rightarrow *sdo-binds* ($\langle - \rangle$)
 -*sdo-cons* :: [*sdo-bind*, *sdo-binds*] \Rightarrow *sdo-binds*
 ($\langle \langle \text{open-block notation} = \langle \text{infix exec next} \rangle \rangle - ; / - \rangle$ [13, 12] 12)

syntax (*ASCII*)

-*sdo-bind* :: [*pttrn*, '*a*] \Rightarrow *sdo-bind*
 ($\langle \langle \text{indent} = 2 \text{ notation} = \langle \text{infix exec bind} \rangle \rangle - < - / - \rangle$ 13)

syntax-consts

-*sdo-let* == *Let*

translations

-*sdo-block* (-*sdo-cons* (-*sdo-bind* *p t*) (-*sdo-final* *e*))
 == *CONST scomp t* ($\lambda p. e$)
 -*sdo-block* (-*sdo-cons* (-*sdo-then* *t*) (-*sdo-final* *e*))
 ==> *CONST fcomp t e*
 -*sdo-final* (-*sdo-block* (-*sdo-cons* (-*sdo-then* *t*) (-*sdo-final* *e*)))
 <= -*sdo-final* (*CONST fcomp t e*)
 -*sdo-block* (-*sdo-cons* (-*sdo-then* *t*) *e*)
 <= *CONST fcomp t* (-*sdo-block* *e*)
 -*sdo-block* (-*sdo-cons* (-*sdo-let* *p t*) *bs*)
 == *let p = t in -sdo-block bs*
 -*sdo-block* (-*sdo-cons* *b* (-*sdo-cons* *c cs*))
 == -*sdo-block* (-*sdo-cons* *b* (-*sdo-final* (-*sdo-block* (-*sdo-cons* *c cs*))))
 -*sdo-cons* (-*sdo-let* *p t*) (-*sdo-final* *s*)

```

== -sdo-final (let p = t in s)
-sdo-block (-sdo-final e) => e

```

For an example, see `~/src/HOL/Proofs/Extraction/Higman_Extraction.thy`.

end

74 Canonical order on option type

```

theory Option-ord
imports Main
begin

```

```

unbundle lattice-syntax

```

```

instantiation option :: (preorder) preorder
begin

```

definition *less-eq-option* **where**

$$x \leq y \longleftrightarrow (\text{case } x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } y \Rightarrow x \leq y))$$

definition *less-option* **where**

$$x < y \longleftrightarrow (\text{case } y \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some } y \Rightarrow (\text{case } x \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } x \Rightarrow x < y))$$

lemma *less-eq-option-None* [*simp*]: $\text{None} \leq x$
by (*simp add: less-eq-option-def*)

lemma *less-eq-option-None-code* [*code*]: $\text{None} \leq x \longleftrightarrow \text{True}$
by *simp*

lemma *less-eq-option-None-is-None*: $x \leq \text{None} \implies x = \text{None}$
by (*cases x (simp-all add: less-eq-option-def)*)

lemma *less-eq-option-Some-None* [*simp, code*]: $\text{Some } x \leq \text{None} \longleftrightarrow \text{False}$
by (*simp add: less-eq-option-def*)

lemma *less-eq-option-Some* [*simp, code*]: $\text{Some } x \leq \text{Some } y \longleftrightarrow x \leq y$
by (*simp add: less-eq-option-def*)

lemma *less-option-None* [*simp, code*]: $x < \text{None} \longleftrightarrow \text{False}$
by (*simp add: less-option-def*)

lemma *less-option-None-is-Some*: $\text{None} < x \implies \exists z. x = \text{Some } z$
by (*cases x (simp-all add: less-option-def)*)

lemma *less-option-None-Some* [*simp*]: $\text{None} < \text{Some } x$
by (*simp add: less-option-def*)

lemma *less-option-None-Some-code* [*code*]: $\text{None} < \text{Some } x \longleftrightarrow \text{True}$
by *simp*

lemma *less-option-Some* [*simp*, *code*]: $\text{Some } x < \text{Some } y \longleftrightarrow x < y$
by (*simp add: less-option-def*)

instance
by *standard*
 (*auto simp add: less-eq-option-def less-option-def less-le-not-le*
elim: order-trans split: option.splits)

end

instance *option* :: (*order*) *order*
by *standard* (*auto simp add: less-eq-option-def less-option-def split: option.splits*)

instance *option* :: (*linorder*) *linorder*
by *standard* (*auto simp add: less-eq-option-def less-option-def split: option.splits*)

instantiation *option* :: (*order*) *order-bot*
begin

definition *bot-option* **where** $\perp = \text{None}$

instance
by *standard* (*simp add: bot-option-def*)

end

instantiation *option* :: (*order-top*) *order-top*
begin

definition *top-option* **where** $\top = \text{Some } \top$

instance
by *standard* (*simp add: top-option-def less-eq-option-def split: option.split*)

end

instance *option* :: (*wellorder*) *wellorder*

proof

fix $P :: 'a \text{ option} \Rightarrow \text{bool}$
fix $z :: 'a \text{ option}$
assume $H: \bigwedge x. (\bigwedge y. y < x \Longrightarrow P y) \Longrightarrow P x$
have $P \text{ None}$ **by** (*rule H*) *simp*
then have $P \text{ Some } [case\text{-names } \text{Some}]: P z$ **if** $\bigwedge x. z = \text{Some } x \Longrightarrow (P \circ \text{Some})$
x for z
using $\langle P \text{ None} \rangle$ **that** **by** (*cases z*) *simp-all*
show $P z$

```

proof (cases z rule: P-Some)
  case (Some w)
  show (P ∘ Some) w
  proof (induct rule: less-induct)
    case (less x)
    have P (Some x)
    proof (rule H)
      fix y :: 'a option
      assume y < Some x
      show P y
      proof (cases y rule: P-Some)
        case (Some v)
        with ⟨y < Some x⟩ have v < x by simp
        with less show (P ∘ Some) v .
      qed
    qed
  then show ?case by simp
qed
qed
qed

```

```

instantiation option :: (inf) inf
begin

```

```

definition inf-option where

```

```

  x ⊓ y = (case x of None ⇒ None | Some x ⇒ (case y of None ⇒ None | Some
y ⇒ Some (x ⊓ y)))

```

```

lemma inf-None-1 [simp, code]: None ⊓ y = None
by (simp add: inf-option-def)

```

```

lemma inf-None-2 [simp, code]: x ⊓ None = None
by (cases x) (simp-all add: inf-option-def)

```

```

lemma inf-Some [simp, code]: Some x ⊓ Some y = Some (x ⊓ y)
by (simp add: inf-option-def)

```

```

instance ..

```

```

end

```

```

instantiation option :: (sup) sup
begin

```

```

definition sup-option where

```

```

  x ⊔ y = (case x of None ⇒ y | Some x' ⇒ (case y of None ⇒ x | Some y ⇒
Some (x' ⊔ y)))

```

```

lemma sup-None-1 [simp, code]: None ⊔ y = y

```

```

    by (simp add: sup-option-def)

lemma sup-None-2 [simp, code]:  $x \sqcup \text{None} = x$ 
  by (cases x) (simp-all add: sup-option-def)

lemma sup-Some [simp, code]:  $\text{Some } x \sqcup \text{Some } y = \text{Some } (x \sqcup y)$ 
  by (simp add: sup-option-def)

instance ..

end

instance option :: (semilattice-inf) semilattice-inf
proof
  fix x y z :: 'a option
  show  $x \sqcap y \leq x$ 
    by (cases x, simp-all, cases y, simp-all)
  show  $x \sqcap y \leq y$ 
    by (cases x, simp-all, cases y, simp-all)
  show  $x \leq y \implies x \leq z \implies x \leq y \sqcap z$ 
    by (cases x, simp-all, cases y, simp-all, cases z, simp-all)
qed

instance option :: (semilattice-sup) semilattice-sup
proof
  fix x y z :: 'a option
  show  $x \leq x \sqcup y$ 
    by (cases x, simp-all, cases y, simp-all)
  show  $y \leq x \sqcup y$ 
    by (cases x, simp-all, cases y, simp-all)
  fix x y z :: 'a option
  show  $y \leq x \implies z \leq x \implies y \sqcup z \leq x$ 
    by (cases y, simp-all, cases z, simp-all, cases x, simp-all)
qed

instance option :: (lattice) lattice ..

instance option :: (lattice) bounded-lattice-bot ..

instance option :: (bounded-lattice-top) bounded-lattice-top ..

instance option :: (bounded-lattice-top) bounded-lattice ..

instance option :: (distrib-lattice) distrib-lattice
proof
  fix x y z :: 'a option
  show  $x \sqcup y \sqcap z = (x \sqcup y) \sqcap (x \sqcup z)$ 
    by (cases x, simp-all, cases y, simp-all, cases z, simp-all add: sup-inf-distrib1
inf-commute)

```

qed

instantiation *option* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-option* :: 'a *option set* \Rightarrow 'a *option* **where**
 $\bigcap A = (\text{if } \text{None} \in A \text{ then } \text{None} \text{ else } \text{Some } (\bigcap \text{Option.these } A))$

lemma *None-in-Inf* [*simp*]: $\text{None} \in A \implies \bigcap A = \text{None}$
by (*simp add: Inf-option-def*)

definition *Sup-option* :: 'a *option set* \Rightarrow 'a *option* **where**
 $\bigcup A = (\text{if } A = \{\} \vee A = \{\text{None}\} \text{ then } \text{None} \text{ else } \text{Some } (\bigcup \text{Option.these } A))$

lemma *empty-Sup* [*simp*]: $\bigcup \{\} = \text{None}$
by (*simp add: Sup-option-def*)

lemma *singleton-None-Sup* [*simp*]: $\bigcup \{\text{None}\} = \text{None}$
by (*simp add: Sup-option-def*)

instance

proof

fix *x* :: 'a *option* **and** *A*
assume $x \in A$
then show $\bigcap A \leq x$
by (*cases x*) (*auto simp add: Inf-option-def in-these-eq intro: Inf-lower*)

next

fix *z* :: 'a *option* **and** *A*
assume *: $\bigwedge x. x \in A \implies z \leq x$
show $z \leq \bigcap A$
proof (*cases z*)
case *None* **then show** ?thesis **by** *simp*
next
case (*Some y*)
show ?thesis
by (*auto simp add: Inf-option-def in-these-eq Some intro!: Inf-greatest dest!:*

*)

qed

next

fix *x* :: 'a *option* **and** *A*
assume $x \in A$
then show $x \leq \bigcup A$
by (*cases x*) (*auto simp add: Sup-option-def in-these-eq intro: Sup-upper*)

next

fix *z* :: 'a *option* **and** *A*
assume *: $\bigwedge x. x \in A \implies x \leq z$
show $\bigcup A \leq z$
proof (*cases z*)
case *None*

```

with * have  $\bigwedge x. x \in A \implies x = \text{None}$  by (auto dest: less-eq-option-None-is-None)
then have  $A = \{\}$   $\vee A = \{\text{None}\}$  by blast
then show ?thesis by (simp add: Sup-option-def)
next
  case (Some y)
  from * have  $\bigwedge w. \text{Some } w \in A \implies \text{Some } w \leq z$  .
  with Some have  $\bigwedge w. w \in \text{Option.these } A \implies w \leq y$ 
    by (simp add: in-these-eq)
  then have  $\bigsqcup \text{Option.these } A \leq y$  by (rule Sup-least)
  with Some show ?thesis by (simp add: Sup-option-def)
qed
next
  show  $\bigsqcup \{\} = (\perp :: 'a \text{ option})$ 
    by (auto simp: bot-option-def)
  show  $\bigsqcap \{\} = (\top :: 'a \text{ option})$ 
    by (auto simp: top-option-def Inf-option-def)
qed
end

lemma Some-Inf:
   $\text{Some } (\bigsqcap A) = \bigsqcap (\text{Some } ` A)$ 
  by (auto simp add: Inf-option-def)

lemma Some-Sup:
   $A \neq \{\} \implies \text{Some } (\bigsqcup A) = \bigsqcup (\text{Some } ` A)$ 
  by (auto simp add: Sup-option-def)

lemma Some-INF:
   $\text{Some } (\bigsqcap x \in A. f x) = (\bigsqcap x \in A. \text{Some } (f x))$ 
  by (simp add: Some-Inf image-comp)

lemma Some-SUP:
   $A \neq \{\} \implies \text{Some } (\bigsqcup x \in A. f x) = (\bigsqcup x \in A. \text{Some } (f x))$ 
  by (simp add: Some-Sup image-comp)

lemma option-Inf-Sup:  $\bigsqcap (\text{Sup } ` A) \leq \bigsqcup (\text{Inf } ` \{f ` A \mid f. \forall Y \in A. f Y \in Y\})$ 
  for A :: ('a::complete-distrib-lattice option) set set
proof (cases  $\{\} \in A$ )
  case True
    then show ?thesis
      by (rule INF-lower2, simp-all)
  next
    case False
    from this have X:  $\{\} \notin A$ 
      by simp
    then show ?thesis
      proof (cases  $\{\text{None}\} \in A$ )
      case True

```

```

then show ?thesis
  by (rule INF-lower2, simp-all)
next
  case False

  {fix y
    assume A: y ∈ A
    have Sup (y - {None}) = Sup y
    by (metis (no-types, lifting) Sup-option-def insert-Diff-single these-insert-None
these-not-empty-eq)
    from A and this have (∃ z. y - {None} = z - {None} ∧ z ∈ A) ∧ ⋒ y =
⋒ (y - {None})
    by auto
  }
  from this have A: Sup ‘ A = (Sup ‘ {y - {None} | y. y ∈ A})
  by (auto simp add: image-def)

  have [simp]: ⋒ y. y ∈ A ⇒ ∃ ya. {ya. ∃ x. x ∈ y ∧ (∃ y. x = Some y) ∧ ya =
the x}
    = {y. ∃ x ∈ ya - {None}. y = the x} ∧ ya ∈ A
  by (rule exI, auto)

  have [simp]: ⋒ y. y ∈ A ⇒
    (∃ ya. y - {None} = ya - {None} ∧ ya ∈ A) ∧ ⋒ {ya. ∃ x ∈ y - {None}.
ya = the x}
    = ⋒ {ya. ∃ x. x ∈ y ∧ (∃ y. x = Some y) ∧ ya = the x}
  apply (safe, blast)
  by (rule arg-cong [of - - Sup], auto)

  {fix y
    assume [simp]: y ∈ A
    have ∃ x. (∃ y. x = {ya. ∃ x ∈ y - {None}. ya = the x} ∧ y ∈ A) ∧ ⋒ {ya.
∃ x. x ∈ y ∧ (∃ y. x = Some y) ∧ ya = the x} = ⋒ x
    and ∃ x. (∃ y. x = y - {None} ∧ y ∈ A) ∧ ⋒ {ya. ∃ x ∈ y - {None}. ya = the
x} = ⋒ {y. ∃ xa. xa ∈ x ∧ (∃ y. xa = Some y) ∧ y = the xa}
    apply (rule exI [of - {ya. ∃ x. x ∈ y ∧ (∃ y. x = Some y) ∧ ya = the x}],
simp)
    by (rule exI [of - y - {None}], simp)
  }
  from this have C: (λx. (⋒ Option.these x)) ‘ {y - {None} | y. y ∈ A} = (Sup
‘ {the ‘ (y - {None}) | y. y ∈ A})
  by (simp add: image-def Option.these-def, safe, simp-all)

  have D: ∀ f . ∃ Y ∈ A. f Y ∉ Y ⇒ False
  by (drule spec [of - λ Y . SOME x . x ∈ Y], simp add: X some-in-eq)

  define F where F = (λ Y . SOME x::'a option . x ∈ (Y - {None}))

  have G: ⋒ Y . Y ∈ A ⇒ ∃ x . x ∈ Y - {None}
  by (metis False X all-not-in-conv insert-Diff-single these-insert-None these-not-empty-eq)

```

```

have  $F: \bigwedge Y. Y \in A \implies F Y \in (Y - \{None\})$ 
  by (metis F-def G empty-iff some-in-eq)

have  $Some \perp \leq Inf (F \text{ ` } A)$ 
  by (metis (no-types, lifting) Diff-iff F Inf-option-def bot.extremum image-iff less-eq-option-Some singletonI)

from this have  $Inf (F \text{ ` } A) \neq None$ 
  by (cases  $\bigcap x \in A. F x$ , simp-all)

from this have  $Inf (F \text{ ` } A) \neq None \wedge Inf (F \text{ ` } A) \in Inf \text{ ` } \{f \text{ ` } A \mid f. \forall Y \in A. f Y \in Y\}$ 
  using F by auto

from this have  $\exists x. x \neq None \wedge x \in Inf \text{ ` } \{f \text{ ` } A \mid f. \forall Y \in A. f Y \in Y\}$ 
  by blast

from this have  $E: Inf \text{ ` } \{f \text{ ` } A \mid f. \forall Y \in A. f Y \in Y\} = \{None\} \implies False$ 
  by blast

have [simp]:  $((\bigcap x \in \{f \text{ ` } A \mid f. \forall Y \in A. f Y \in Y\}. \bigcap x) = None) = False$ 
  by (metis (no-types, lifting) E Sup-option-def  $\langle \exists x. x \neq None \wedge x \in Inf \text{ ` } \{f \text{ ` } A \mid f. \forall Y \in A. f Y \in Y\} \rangle$  ex-in-conv option.simps(3))

have  $B: Option.these ((\lambda x. Some (\bigcap Option.these x)) \text{ ` } \{y - \{None\} \mid y. y \in A\})$ 
   $= ((\lambda x. (\bigcap Option.these x)) \text{ ` } \{y - \{None\} \mid y. y \in A\})$ 
  by (metis image-image these-image-Some-eq)
{
  fix f
  assume  $A: \bigwedge Y. (\exists y. Y = the \text{ ` } (y - \{None\}) \wedge y \in A) \implies f Y \in Y$ 

  have  $\bigwedge xa. xa \in A \implies f \{y. \exists a \in xa - \{None\}. y = the a\} = f (the \text{ ` } (xa - \{None\}))$ 
    by (simp add: image-def)
  from this have [simp]:  $\bigwedge xa. xa \in A \implies \exists x \in A. f \{y. \exists a \in xa - \{None\}. y = the a\} = f (the \text{ ` } (x - \{None\}))$ 
    by blast
  have  $\bigwedge xa. xa \in A \implies f (the \text{ ` } (xa - \{None\})) = f \{y. \exists a \in xa - \{None\}. y = the a\} \wedge xa \in A$ 
    by (simp add: image-def)
  from this have [simp]:  $\bigwedge xa. xa \in A \implies \exists x. f (the \text{ ` } (xa - \{None\})) = f \{y. \exists a \in x - \{None\}. y = the a\} \wedge x \in A$ 
    by blast

  {
    fix Y

```

```

have  $Y \in A \implies \text{Some } (f \text{ (the ' (} Y - \{None\}))) \in Y$ 
using  $A \text{ [of the ' (} Y - \{None\})]$  apply (simp add: image-def)
using option.collapse by fastforce
}
from this have [simp]:  $\bigwedge Y . Y \in A \implies \text{Some } (f \text{ (the ' (} Y - \{None\}))) \in Y$ 
by blast
have [simp]:  $(\bigcap_{x \in A} . \text{Some } (f \{y. \exists x \in x - \{None\}. y = \text{the } x\})) = \bigcap \{ \text{Some } (f \{y. \exists a \in x - \{None\}. y = \text{the } a\}) \mid x. x \in A \}$ 
by (simp add: Setcompr-eq-image)

have [simp]:  $\exists x. (\exists f. x = \{y. \exists x \in A. y = f x\} \wedge (\forall Y \in A. f Y \in Y)) \wedge \bigcap \{ \text{Some } (f \{y. \exists a \in x - \{None\}. y = \text{the } a\}) \mid x. x \in A \} = \bigcap x$ 
apply (rule exI [of - {Some } (f {y. \exists a \in x - \{None\}. y = \text{the } a\}) \mid x . x \in A]], safe)
by (rule exI [of - (\lambda Y . Some (f (the ' (Y - \{None\}))))], safe, simp-all)

{
fix  $xb$ 
have  $xb \in A \implies (\bigcap_{x \in \{ \text{the ' (} y - \{None\} \mid y. y \in A \}. f x \}} \{y. \exists x \in xb - \{None\}. y = \text{the } x\}) \leq f \{y. \exists x \in xb - \{None\}. y = \text{the } x\}$ 
apply (rule INF-lower2 [of {y. \exists x \in xb - \{None\}. y = \text{the } x}])
by blast+
}
from this have [simp]:  $(\bigcap_{x \in \{ \text{the ' (} y - \{None\} \mid y. y \in A \}. f x \}} \{y. y \in A\}. f x) \leq \text{the } (\bigcap_{Y \in A} . \text{Some } (f \text{ (the ' (} Y - \{None\}))))$ 
apply (simp add: Inf-option-def image-def Option.these-def)
by (rule Inf-greatest, clarsimp)
have [simp]:  $\text{the } (\bigcap_{Y \in A} . \text{Some } (f \text{ (the ' (} Y - \{None\})))) \in \text{Option.these } (\text{Inf ' } \{f \text{ ' } A \mid f. \forall Y \in A. f Y \in Y\})$ 
apply (auto simp add: Option.these-def)
apply (rule imageI)
apply auto
using  $\langle \bigwedge Y. Y \in A \implies \text{Some } (f \text{ (the ' (} Y - \{None\}))) \in Y \rangle$  apply blast
apply (auto simp add: Some-INF [symmetric])
done
have  $(\bigcap_{x \in \{ \text{the ' (} y - \{None\} \mid y. y \in A \}. f x \}} \{y. y \in A\}. f x) \leq \bigsqcup \text{Option.these } (\text{Inf ' } \{f \text{ ' } A \mid f. \forall Y \in A. f Y \in Y\})$ 
by (rule Sup-upper2 [of the (Inf ((\lambda Y . Some (f (the ' (Y - \{None\}))) )) ' A)], simp-all)
}
from this have  $X: \bigwedge f . \forall Y. (\exists y. Y = \text{the ' (} y - \{None\}) \wedge y \in A) \longrightarrow f Y \in Y \implies$ 
 $(\bigcap_{x \in \{ \text{the ' (} y - \{None\} \mid y. y \in A \}. f x \}} \{y. y \in A\}. f x) \leq \bigsqcup \text{Option.these } (\text{Inf ' } \{f \text{ ' } A \mid f. \forall Y \in A. f Y \in Y\})$ 
by blast

```

have [*simp*]: $\bigwedge x . x \in \{y - \{None\} \mid y. y \in A\} \implies x \neq \{\} \wedge x \neq \{None\}$


```

using  $F$  by fastforce

have  $(\text{Inf } (\text{Sup } 'A)) = (\text{Inf } (\text{Sup } ' \{y - \{None\} \mid y. y \in A\}))$ 
by (subst A, simp)

also have  $\dots = (\bigcap x \in \{y - \{None\} \mid y. y \in A\}. \text{ if } x = \{\} \vee x = \{None\} \text{ then } None \text{ else } Some (\bigcup Option.these x))$ 
by (simp add: Sup-option-def)

also have  $\dots = (\bigcap x \in \{y - \{None\} \mid y. y \in A\}. Some (\bigcup Option.these x))$ 
using  $G$  by fastforce

also have  $\dots = Some (\bigcap Option.these ((\lambda x. Some (\bigcup Option.these x)) ' \{y - \{None\} \mid y. y \in A\}))$ 
by (simp add: Inf-option-def, safe)

also have  $\dots = Some (\bigcap ((\lambda x. (\bigcup Option.these x)) ' \{y - \{None\} \mid y. y \in A\}))$ 
by (simp add: B)

also have  $\dots = Some (\text{Inf } (\text{Sup } ' \{the ' (y - \{None\}) \mid y. y \in A\}))$ 
by (unfold C, simp)
thm Inf-Sup
also have  $\dots = Some (\bigcup x \in \{f ' \{the ' (y - \{None\}) \mid y. y \in A\} \mid f. \forall Y. (\exists y. Y = the ' (y - \{None\}) \wedge y \in A) \longrightarrow f Y \in Y\}. \bigcap x)$ 
by (simp add: Inf-Sup)

also have  $\dots \leq \bigcup (\text{Inf } ' \{f ' A \mid f. \forall Y \in A. f Y \in Y\})$ 
proof (cases  $\bigcup (\text{Inf } ' \{f ' A \mid f. \forall Y \in A. f Y \in Y\})$ )
  case None
    then show ?thesis by (simp add: less-eq-option-def)
  next
    case (Some a)
    then show ?thesis
      apply simp
      apply (rule Sup-least, safe)
      apply (simp add: Sup-option-def)
      apply (cases  $(\forall f. \exists Y \in A. f Y \notin Y) \vee \text{Inf } ' \{f ' A \mid f. \forall Y \in A. f Y \in Y\} = \{None\}$ , simp-all)
      by (drule X, simp)
    qed
  finally show ?thesis by simp
qed
qed

instance option :: (complete-distrib-lattice) complete-distrib-lattice
by (standard, simp add: option-Inf-Sup)

instance option :: (complete-linorder) complete-linorder ..

```

unbundle *no lattice-syntax*

end

75 Futures and parallel lists for code generated towards Isabelle/ML

theory *Parallel*
imports *Main*
begin

75.1 Futures

datatype *'a future = fork unit \Rightarrow 'a*

primrec *join :: 'a future \Rightarrow 'a* **where**
join (fork f) = f ()

lemma *future-eqI [intro!]:*
assumes join f = join g
shows f = g
using *assms* **by** (*cases f, cases g (simp add: ext)*)

code-printing

type-constructor *future \rightarrow (Eval) - future*
| **constant** *fork \rightarrow (Eval) Future.fork*
| **constant** *join \rightarrow (Eval) Future.join*

code-reserved (*Eval*) *Future future*

75.2 Parallel lists

definition *map :: ('a \Rightarrow 'b) \Rightarrow 'a list \Rightarrow 'b list* **where**
[simp]: map = List.map

definition *forall :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow bool* **where**
forall = list-all

lemma *forall-all [simp]:*
forall P xs \longleftrightarrow ($\forall x \in \text{set } xs. P x$)
by (*simp add: forall-def list-all-iff*)

definition *exists :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow bool* **where**
exists = list-ex

lemma *exists-ex [simp]:*
exists P xs \longleftrightarrow ($\exists x \in \text{set } xs. P x$)
by (*simp add: exists-def list-ex-iff*)

```

code-printing
  constant map  $\rightarrow$  (Eval) Par'-List.map
| constant forall  $\rightarrow$  (Eval) Par'-List.forall
| constant exists  $\rightarrow$  (Eval) Par'-List.exists

code-reserved (Eval) Par-List

```

```
hide-const (open) fork join map exists forall
```

```
end
```

76 Input syntax for pattern aliases (or “as-patterns” in Haskell)

```

theory Pattern-Aliases
imports Main
begin

```

Most functional languages (Haskell, ML, Scala) support aliases in patterns. This allows to refer to a subpattern with a variable name. This theory implements this using a check phase. It works well for function definitions (see usage below). All features are packed into a **bundle**.

The following caveats should be kept in mind:

- The translation expects a term of the form $f\ x\ y = rhs$, where x and y are patterns that may contain aliases. The result of the translation is a nested *Let*-expression on the right hand side. The code generator *does not* print Isabelle pattern aliases to target language pattern aliases.
- The translation does not process nested equalities; only the top-level equality is translated.
- Terms that do not adhere to the above shape may either stay untranslated or produce an error message. The **fun** command will complain if pattern aliases are left untranslated. In particular, there are no checks whether the patterns are wellformed or linear.
- The corresponding uncheck phase attempts to reverse the translation (no guarantee). The additionally introduced variables are bound using a “fake quantifier” that does not appear in the output.
- To obtain reasonable induction principles in function definitions, the bundle also declares a custom congruence rule for *Let* that only affects **fun**. This congruence rule might lead to an explosion in term size (although that is rare)! In some circumstances (using *let* to destructure

tuples), the internal construction of functions stumbles over this rule and prints an error. To mitigate this, either

- activate the bundle locally (**context includes ... begin**) or
 - rewrite the *let*-expression to use *case*: *let* $(a, b) = x$ *in* (b, a) becomes *case* x *of* $(a, b) \Rightarrow (b, a)$.
- The bundle also adds the *Let* $?s \ ?f \equiv ?f \ ?s$ rule to the simpset.

76.1 Definition

consts

as :: $'a \Rightarrow 'a \Rightarrow 'a$
fake-quant :: $('a \Rightarrow \text{prop}) \Rightarrow \text{prop}$

lemma *let-cong-unfolding*: $M = N \Longrightarrow f \ N = g \ N \Longrightarrow \text{Let } M \ f = \text{Let } N \ g$
by *simp*

translations $P \leq \text{CONST } \text{fake-quant } (\lambda x. P)$

ML_⊂

local

fun let-typ $a \ b = a \ --> (a \ --> b) \ --> b$
fun as-typ $a = a \ --> a \ --> a$

fun strip-all $t =$
case try *Logic.dest-all-global* t *of*
 $NONE \Rightarrow ([], t)$
 $| SOME \ (var, t) \Rightarrow \text{apfst } (\text{cons } var) \ (\text{strip-all } t)$

fun all-Frees $t =$
fold-aterms $(\text{fn } Free \ (x, t) \Rightarrow \text{insert } (op =) \ (x, t) \ | \ - \Rightarrow I) \ t \ []$

fun subst-once $(old, new) \ t =$
let
fun go $t =$
if $t = old$ *then*
 $(new, true)$
else
case t *of*
 $u \ \$ \ v \Rightarrow$
let
 $val \ (u', \text{substituted}) = go \ u$
in
if *substituted* *then*
 $(u' \ \$ \ v, true)$

```

      else
        case go v of (v', substituted) => (u $ v', substituted)
      end
    | Abs (name, typ, t) =>
      (case go t of (t', substituted) => (Abs (name, typ, t'), substituted))
    | - => (t, false)
  in fst (go t) end

(* adapted from logic.ML *)
fun fake-const T = Const (const-name <fake-quant>, (T --> propT) --> propT);

fun dependent-fake-name v t =
  let
    val x = Term.term-name v
    val T = Term.fastype-of v
    val t' = Term.abstract-over (v, t)
  in if Term.is-dependent t' then fake-const T $ Abs (x, T, t') else t end

in

fun check-pattern-syntax t =
  case strip-all t of
    (vars, Const <Trueprop> $ (Const (const-name <HOL.eq>, -) $ lhs $ rhs)) =>
      let
        fun go (Const (const-name <as>, -) $ pat $ var, rhs) =
          let
            val (pat', rhs') = go (pat, rhs)
            val - = if is-Free var then () else error "Right-hand side of =: must
be a free variable"
            val rhs'' =
              Const (const-name <Let>, let-typ (fastype-of var) (fastype-of rhs)) $
                pat' $ lambda var rhs'
          in
            (pat', rhs'')
          end
        | go (t $ u, rhs) =
          let
            val (t', rhs') = go (t, rhs)
            val (u', rhs'') = go (u, rhs')
            in (t' $ u', rhs'') end
        | go (t, rhs) = (t, rhs)

      val (lhs', rhs') = go (lhs, rhs)

      val res = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs', rhs'))

      val frees = filter (member (op =) vars) (all-Frees res)
      in fold (fn v => Logic.dependent-all-name (, v)) (map Free frees) res end
    | - => t

```

```

fun uncheck-pattern-syntax ctxt t =
  case strip-all t of
    (vars, Const ⟨Trueprop⟩ $ (Const (const-name ⟨HOL.eq⟩, -) $ lhs $ rhs)) =>
      let
        (* restricted to going down abstractions; ignores eta-contracted rhs *)
        fun go lhs (rhs as Const (const-name ⟨Let⟩, -) $ pat $ Abs (name, typ, t))
      in
        ctxt frees =
          if exists-subterm (fn t' => t' = pat) lhs then
            let
              val ([name'], ctxt') = Variable.variant-fixes [name] ctxt
              val free = Free (name', typ)
              val subst = (pat, Const (const-name ⟨as⟩, as-typ typ) $ pat $ free)
              val lhs' = subst-once subst lhs
              val rhs' = subst-bound (free, t)
            in
              go lhs' rhs' ctxt' (Free (name', typ) :: frees)
            end
          else
            (lhs, rhs, ctxt, frees)
          | go lhs rhs ctxt frees = (lhs, rhs, ctxt, frees)

        val (lhs', rhs', -, frees) = go lhs rhs ctxt []

        val res =
          HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs', rhs'))
          |> fold (fn v => Logic.dependent-all-name (, v)) (map Free vars)
          |> fold dependent-fake-name frees
        in
          if null frees then t else res
        end
      | - => t

end
>

bundle pattern-aliases begin

  notation as (infixr ⟨=:⟩ 1)

  declaration ⟨K (Syntax-Phases.term-check 98 pattern-syntax (K (map check-pattern-syntax)))⟩
  declaration ⟨K (Syntax-Phases.term-uncheck 98 pattern-syntax (map o uncheck-pattern-syntax))⟩

  declare let-cong-unfolding [fundef-cong]
  declare Let-def [simp]

end

hide-const as

```

hide-const *fake-quant*

76.2 Usage

context includes *pattern-aliases* **begin**

Not very useful for plain definitions, but works anyway.

private definition *test-1* $x (y =: z) = y + z$

lemma *test-1* $x y = y + y$
by (*rule test-1-def*[*unfolded Let-def*])

Very useful for function definitions.

private fun *test-2* **where**
test-2 $(y \# (y' \# ys =: x') =: x) = x @ x' @ x' |$
test-2 - = []

lemma *test-2* $(y \# y' \# ys) = (y \# y' \# ys) @ (y' \# ys) @ (y' \# ys)$
by (*rule test-2.simps*[*unfolded Let-def*])

ML_⊂

let

val actual =
 @{*thm test-2.simps*(1)}
 |> *Thm.prop-of*
 |> *Syntax.pretty-term context*
 |> *Pretty.pure-string-of*
val expected = *test-2* (?*y* # (?*y'* # ?*ys* =: *x'*) =: *x*) = *x* @ *x'* @ *x'*
in assert (*actual* = *expected*) *end*
 >

end

end

77 Periodic Functions

theory *Periodic-Fun*
imports *Complex-Main*
begin

A locale for periodic functions. The idea is that one proves $f(x + p) = f(x)$ for some period p and gets derived results like $f(x - p) = f(x)$ and $f(x + 2p) = f(x)$ for free.

g and gm are “plus/minus k periods” functions. $g1$ and $gn1$ are “plus/minus one period” functions. This is useful e.g. if the period is one; the lemmas one gets are then $f(x + 1) = f x$ instead of $f(x + 1 * 1) = f x$ etc.

locale *periodic-fun* =

fixes $f :: ('a :: \{\text{ring-1}\}) \Rightarrow 'b$ **and** $g \text{ gm} :: 'a \Rightarrow 'a \Rightarrow 'a$ **and** $g1 \text{ gn1} :: 'a \Rightarrow 'a$
assumes $\text{plus-1}: f (g1 \ x) = f \ x$
assumes $\text{periodic-arg-plus-0}: g \ x \ 0 = x$
assumes $\text{periodic-arg-plus-distrib}: g \ x \ (\text{of-int } (m + n)) = g \ (g \ x \ (\text{of-int } n)) \ (\text{of-int } m)$
assumes $\text{plus-1-eq}: g \ x \ 1 = g1 \ x$ **and** $\text{minus-1-eq}: g \ x \ (-1) = gn1 \ x$
and $\text{minus-eq}: g \ x \ (-y) = gm \ x \ y$
begin

lemma $\text{plus-of-nat}: f (g \ x \ (\text{of-nat } n)) = f \ x$
by ($\text{induction } n$) ($\text{insert periodic-arg-plus-distrib}[\text{of - 1 int } n \text{ for } n]$,
 $\text{simp-all add: plus-1 periodic-arg-plus-0 plus-1-eq}$)

lemma $\text{minus-of-nat}: f (gm \ x \ (\text{of-nat } n)) = f \ x$
proof –
have $f (g \ x \ (- \text{of-nat } n)) = f (g (g \ x \ (- \text{of-nat } n)) (\text{of-nat } n))$
by ($\text{rule plus-of-nat}[\text{symmetric}]$)
also have $\dots = f (g (g \ x \ (\text{of-int } (- \text{of-nat } n))) (\text{of-int } (\text{of-nat } n)))$ **by** simp
also have $\dots = f \ x$
by ($\text{subst periodic-arg-plus-distrib} [\text{symmetric}]$) ($\text{simp add: periodic-arg-plus-0}$)
finally show $?thesis$ **by** ($\text{simp add: minus-eq}$)
qed

lemma $\text{plus-of-int}: f (g \ x \ (\text{of-int } n)) = f \ x$
by ($\text{induction } n$) ($\text{simp-all add: plus-of-nat minus-of-nat minus-eq del: of-nat-Suc}$)

lemma $\text{minus-of-int}: f (gm \ x \ (\text{of-int } n)) = f \ x$
using $\text{plus-of-int}[\text{of } x \ \text{of-int } (-n)]$ **by** ($\text{simp add: minus-eq}$)

lemma $\text{plus-numeral}: f (g \ x \ (\text{numeral } n)) = f \ x$
by ($\text{subst of-nat-numeral}[\text{symmetric}]$, subst plus-of-nat) (rule refl)

lemma $\text{minus-numeral}: f (gm \ x \ (\text{numeral } n)) = f \ x$
by ($\text{subst of-nat-numeral}[\text{symmetric}]$, $\text{subst minus-of-nat}$) (rule refl)

lemma $\text{minus-1}: f (gn1 \ x) = f \ x$
using $\text{minus-of-nat}[\text{of } x \ 1]$ **by** ($\text{simp flip: minus-1-eq minus-eq}$)

lemmas $\text{periodic-simps} = \text{plus-of-nat minus-of-nat plus-of-int minus-of-int}$
 $\text{plus-numeral minus-numeral plus-1 minus-1}$

end

Specialised case of the *periodic-fun* locale for periods that are not 1.
 Gives lemmas $f (x - \text{period}) = f \ x$ etc.

locale $\text{periodic-fun-simple} =$
fixes $f :: ('a :: \{\text{ring-1}\}) \Rightarrow 'b$ **and** $\text{period} :: 'a$
assumes $\text{plus-period}: f (x + \text{period}) = f \ x$
begin


```

sublocale periodic-fun  $f \lambda z x. z + x * \text{period} \lambda z x. z - x * \text{period}$ 
   $\lambda z. z + \text{period} \lambda z. z - \text{period}$ 
  by standard (simp-all add: ring-distrib plus-period)
end

```

Specialised case of the *periodic-fun* locale for period 1. Gives lemmas $f(x - 1) = f x$ etc.

```

locale periodic-fun-simple' =
  fixes  $f :: ('a :: \{\text{ring-1}\}) \Rightarrow 'b$ 
  assumes plus-period:  $f(x + 1) = f x$ 
begin
sublocale periodic-fun  $f \lambda z x. z + x \lambda z x. z - x \lambda z. z + 1 \lambda z. z - 1$ 
  by standard (simp-all add: ring-distrib plus-period)

lemma of-nat:  $f(\text{of-nat } n) = f 0$  using plus-of-nat[of 0 n] by simp
lemma uminus-of-nat:  $f(-\text{of-nat } n) = f 0$  using minus-of-nat[of 0 n] by simp
lemma of-int:  $f(\text{of-int } n) = f 0$  using plus-of-int[of 0 n] by simp
lemma uminus-of-int:  $f(-\text{of-int } n) = f 0$  using minus-of-int[of 0 n] by simp
lemma of-numeral:  $f(\text{numeral } n) = f 0$  using plus-numeral[of 0 n] by simp
lemma of-neg-numeral:  $f(-\text{numeral } n) = f 0$  using minus-numeral[of 0 n] by
  simp
lemma of-1:  $f 1 = f 0$  using plus-of-nat[of 0 1] by simp
lemma of-neg-1:  $f(-1) = f 0$  using minus-of-nat[of 0 1] by simp

lemmas periodic-simps' =
  of-nat uminus-of-nat of-int uminus-of-int of-numeral of-neg-numeral of-1 of-neg-1
end

```

```

lemma sin-plus-pi:  $\sin((z :: 'a :: \{\text{real-normed-field}, \text{banach}\}) + \text{of-real } \pi) = -\sin z$ 
by (simp add: sin-add)

```

```

lemma cos-plus-pi:  $\cos((z :: 'a :: \{\text{real-normed-field}, \text{banach}\}) + \text{of-real } \pi) = -\cos z$ 
by (simp add: cos-add)

```

```

interpretation sin: periodic-fun-simple  $\sin 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$ 
proof
  fix  $z :: 'a$ 
  have  $\sin(z + 2 * \text{of-real } \pi) = \sin(z + \text{of-real } \pi + \text{of-real } \pi)$  by (simp add: ac-simps)
  also have  $\dots = \sin z$  by (simp only: sin-plus-pi) simp
  finally show  $\sin(z + 2 * \text{of-real } \pi) = \sin z$  .
qed

```

```

interpretation cos: periodic-fun-simple  $\cos 2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$ 
proof
  fix  $z :: 'a$ 

```

have $\cos (z + 2 * \text{of-real } \pi) = \cos (z + \text{of-real } \pi + \text{of-real } \pi)$ **by** (*simp add: ac-simps*)
also have $\dots = \cos z$ **by** (*simp only: cos-plus-pi simp*)
finally show $\cos (z + 2 * \text{of-real } \pi) = \cos z$.
qed

interpretation *tan: periodic-fun-simple tan* $2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
by *standard (simp only: tan-def [abs-def] sin.plus-1 cos.plus-1)*

interpretation *cot: periodic-fun-simple cot* $2 * \text{of-real } \pi :: 'a :: \{\text{real-normed-field}, \text{banach}\}$
by *standard (simp only: cot-def [abs-def] sin.plus-1 cos.plus-1)*

lemma *cos-eq-neg-periodic-intro:*

assumes $x - y = 2 * (\text{of-int } k) * \pi + \pi \vee x + y = 2 * (\text{of-int } k) * \pi + \pi$
shows $\cos x = - \cos y$ **using** *assms*

proof

assume $x - y = 2 * (\text{of-int } k) * \pi + \pi$
then show *?thesis*
using *cos.periodic-simps[of y+pi]*
by (*auto simp add: algebra-simps*)

next

assume $x + y = 2 * \text{real-of-int } k * \pi + \pi$
then show *?thesis*
using *cos.periodic-simps[of -y+pi]*
by (*clarsimp simp add: algebra-simps (smt (verit))*)

qed

lemma *cos-eq-periodic-intro:*

assumes $x - y = 2 * (\text{of-int } k) * \pi \vee x + y = 2 * (\text{of-int } k) * \pi$
shows $\cos x = \cos y$
by (*smt (verit, best) assms cos-eq-neg-periodic-intro cos-minus-pi cos-periodic-pi*)

lemma *cos-eq-arccos-Ex:*

$\cos x = y \longleftrightarrow -1 \leq y \wedge y \leq 1 \wedge (\exists k :: \text{int}. x = \arccos y + 2 * k * \pi \vee x = - \arccos y + 2 * k * \pi)$ (**is** *?L=?R*)

proof

assume *?R then show* $\cos x = y$

by (*metis cos.plus-of-int cos-arccos cos-minus id-apply mult.assoc mult.left-commute of-real-eq-id*)

next

assume *L: ?L*

let *?goal* = $(\exists k :: \text{int}. x = \arccos y + 2 * k * \pi \vee x = - \arccos y + 2 * k * \pi)$

obtain $k :: \text{int}$ **where** $k: -\pi < x - k * (2 * \pi) \wedge x - k * (2 * \pi) \leq \pi$

using *ceiling-divide-lower [of 2*pi x-pi] ceiling-divide-upper [of 2*pi x-pi]*

by (*simp add: divide-simps algebra-simps (metis mult.commute)*)

have $*$: $\cos (x - k * 2 * \pi) = y$

using *cos.periodic-simps(3)[of x-k] L* **by** (*auto simp add: field-simps*)

then have $**$: *?goal* **when** $x - k * 2 * \pi \geq 0$

using *arccos-cos k that by force*

```

    then show  $-1 \leq y \wedge y \leq 1 \wedge ?goal$ 
    using * arccos-cos2 k(1) by force
qed

```

```

end

```

78 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view

```

theory Poly-Mapping
imports Groups-Big-Fun Fun-Lexorder More-List
begin

```

78.1 Preliminary: auxiliary operations for *almost everywhere zero*

A central notion for polynomials are functions being *almost everywhere zero*. For these we provide some auxiliary definitions and lemmas.

lemma *finite-mult-not-eq-zero-leftI*:

```

  fixes f :: 'b  $\Rightarrow$  'a :: mult-zero
  assumes finite {a. f a  $\neq$  0}
  shows finite {a. g a * f a  $\neq$  0}
  by (metis (mono-tags, lifting) Collect-mono assms mult-zero-right finite-subset)

```

lemma *finite-mult-not-eq-zero-rightI*:

```

  fixes f :: 'b  $\Rightarrow$  'a :: mult-zero
  assumes finite {a. f a  $\neq$  0}
  shows finite {a. f a * g a  $\neq$  0}
  by (metis (mono-tags, lifting) Collect-mono assms lambda-zero finite-subset)

```

lemma *finite-mult-not-eq-zero-prodI*:

```

  fixes f g :: 'a  $\Rightarrow$  'b::semiring-0
  assumes finite {a. f a  $\neq$  0} (is finite ?F)
  assumes finite {b. g b  $\neq$  0} (is finite ?G)
  shows finite {(a, b). f a * g b  $\neq$  0}

```

proof –

```

  from assms have finite (?F  $\times$  ?G)
  by blast
  then have finite {(a, b). f a  $\neq$  0  $\wedge$  g b  $\neq$  0}
  by simp
  then show ?thesis
  by (rule rev-finite-subset) auto

```

qed

lemma *finite-not-eq-zero-sumI*:

```

  fixes f g :: 'a::monoid-add  $\Rightarrow$  'b::semiring-0
  assumes finite {a. f a  $\neq$  0} (is finite ?F)

```

```

assumes finite {b. g b ≠ 0} (is finite ?G)
shows finite {a + b | a b. f a ≠ 0 ∧ g b ≠ 0} (is finite ?FG)
proof –
  from assms have finite (?F × ?G)
    by (simp add: finite-cartesian-product-iff)
  then have finite (case-prod plus ‘(?F × ?G))
    by (rule finite-imageI)
  also have case-prod plus ‘(?F × ?G) = ?FG
    by auto
  finally show ?thesis
    by simp
qed

```

```

lemma finite-mult-not-eq-zero-sumI:
  fixes f g :: 'a::monoid-add ⇒ 'b::semiring-0
  assumes finite {a. f a ≠ 0}
  assumes finite {b. g b ≠ 0}
  shows finite {a + b | a b. f a * g b ≠ 0}
proof –
  from assms
  have finite {a + b | a b. f a ≠ 0 ∧ g b ≠ 0}
    by (rule finite-not-eq-zero-sumI)
  then show ?thesis
    by (rule rev-finite-subset) (auto dest: mult-not-zero)
qed

```

```

lemma finite-Sum-any-not-eq-zero-weakenI:
  assumes finite {a. ∃ b. f a b ≠ 0}
  shows finite {a. Sum-any (f a) ≠ 0}
proof –
  have {a. Sum-any (f a) ≠ 0} ⊆ {a. ∃ b. f a b ≠ 0}
    by (auto elim: Sum-any.not-neutral-obtains-not-neutral)
  then show ?thesis using assms by (rule finite-subset)
qed

```

```

context zero
begin

```

```

definition when :: 'a ⇒ bool ⇒ 'a (infixl ⟨when⟩ 20)
where
  (a when P) = (if P then a else 0)

```

Case distinctions always complicate matters, particularly when nested. The (*when*) operation allows to minimise these if 0 is the false-case value and makes proof obligations much more readable.

```

lemma when [simp]:
  P ⇒ (a when P) = a
  ¬ P ⇒ (a when P) = 0
  by (simp-all add: when-def)

```

lemma *when-simps* [*simp*]:

(*a when True*) = *a*
 (*a when False*) = 0
by *simp-all*

lemma *when-cong*:

assumes $P \longleftrightarrow Q$
and $Q \implies a = b$
shows (*a when P*) = (*b when Q*)
using *assms* **by** (*simp add: when-def*)

lemma *zero-when* [*simp*]:

(0 *when P*) = 0
by (*simp add: when-def*)

lemma *when-when*:

(*a when P when Q*) = (*a when P* \wedge *Q*)
by (*cases Q*) *simp-all*

lemma *when-commute*:

(*a when Q when P*) = (*a when P when Q*)
by (*simp add: when-when conj-commute*)

lemma *when-neq-zero* [*simp*]:

(*a when P*) \neq 0 $\longleftrightarrow P \wedge a \neq 0$
by (*cases P*) *simp-all*

end

context *monoid-add*

begin

lemma *when-add-distrib*:

(*a + b when P*) = (*a when P*) + (*b when P*)
by (*simp add: when-def*)

end

context *semiring-1*

begin

lemma *zero-power-eq*:

$0 \wedge n = (1 \text{ when } n = 0)$
by (*simp add: power-0-left*)

end

context *comm-monoid-add*

begin

lemma *Sum-any-when-equal* [simp]:
 $(\sum a. (f a \text{ when } a = b)) = f b$
by (simp add: when-def)

lemma *Sum-any-when-equal'* [simp]:
 $(\sum a. (f a \text{ when } b = a)) = f b$
by (simp add: when-def)

lemma *Sum-any-when-independent*:
 $(\sum a. g a \text{ when } P) = ((\sum a. g a) \text{ when } P)$
by (cases P) simp-all

lemma *Sum-any-when-dependent-prod-right*:
 $(\sum (a, b). g a \text{ when } b = h a) = (\sum a. g a)$
proof –
have inj-on $(\lambda a. (a, h a)) \{a. g a \neq 0\}$
by (rule inj-onI) auto
then show ?thesis **unfolding** *Sum-any.expand-set*
by (rule sum.reindex-cong) auto
qed

lemma *Sum-any-when-dependent-prod-left*:
 $(\sum (a, b). g b \text{ when } a = h b) = (\sum b. g b)$
proof –
have $(\sum (a, b). g b \text{ when } a = h b) = (\sum (b, a). g b \text{ when } a = h b)$
by (rule *Sum-any.reindex-cong* [of prod.swap]) (simp-all add: fun-eq-iff)
then show ?thesis **by** (simp add: *Sum-any-when-dependent-prod-right*)
qed

end

context *cancel-comm-monoid-add*
begin

lemma *when-diff-distrib*:
 $(a - b \text{ when } P) = (a \text{ when } P) - (b \text{ when } P)$
by (simp add: when-def)

end

context *group-add*
begin

lemma *when-uminus-distrib*:
 $(- a \text{ when } P) = - (a \text{ when } P)$
by (simp add: when-def)

end

context *mult-zero*

begin

lemma *mult-when*:

$a * (b \text{ when } P) = (a * b \text{ when } P)$

by (*cases* *P*) *simp-all*

lemma *when-mult*:

$(a \text{ when } P) * b = (a * b \text{ when } P)$

by (*cases* *P*) *simp-all*

end

78.2 Type definition

The following type is of central importance:

typedef (**overloaded**) (*'a*, *'b*) *poly-mapping* ($\langle (- \Rightarrow_0 / -) \rangle [1, 0] 0$) =
 $\{f :: 'a \Rightarrow 'b :: \text{zero. finite } \{x. f\ x \neq 0\}\}$
morphisms *lookup Abs-poly-mapping*
using *not-finite-existsD* **by** *force*

declare *lookup-inverse* [*simp*]

declare *lookup-inject* [*simp*]

lemma *lookup-Abs-poly-mapping* [*simp*]:

$\text{finite } \{x. f\ x \neq 0\} \implies \text{lookup } (\text{Abs-poly-mapping } f) = f$

using *Abs-poly-mapping-inverse* [*of f*] **by** *simp*

lemma *finite-lookup* [*simp*]:

$\text{finite } \{k. \text{lookup } f\ k \neq 0\}$

using *poly-mapping.lookup* [*of f*] **by** *simp*

lemma *finite-lookup-nat* [*simp*]:

fixes $f :: 'a \Rightarrow_0 \text{nat}$

shows $\text{finite } \{k. 0 < \text{lookup } f\ k\}$

using *poly-mapping.lookup* [*of f*] **by** *simp*

lemma *poly-mapping-eqI*:

assumes $\bigwedge k. \text{lookup } f\ k = \text{lookup } g\ k$

shows $f = g$

using *assms* **unfolding** *poly-mapping.lookup-inject* [*symmetric*]

by *blast*

lemma *poly-mapping-eq-iff*: $a = b \iff \text{lookup } a = \text{lookup } b$

by *auto*

We model the universe of functions being *almost everywhere* zero by

means of a separate type $'a \Rightarrow_0 'b$. For convenience we provide a suggestive infix syntax which is a variant of the usual function space syntax. Conversion between both types happens through the morphisms

lookup

Abs-poly-mapping

satisfying

$$\text{Abs-poly-mapping } (\text{lookup } ?x) = ?x$$

$$\text{finite } \{x. ?f \ x \neq 0\} \implies \text{lookup } (\text{Abs-poly-mapping } ?f) = ?f$$

Luckily, we have rarely to deal with those low-level morphisms explicitly but rely on Isabelle’s *lifting* package with its method *transfer* and its specification tool *lift-definition*.

setup-lifting *type-definition-poly-mapping*

code-datatype *Abs-poly-mapping*—FIXME? workaround for preventing *code-abstract* setup

$'a \Rightarrow_0 'b$ serves distinctive purposes:

1. A clever nesting as $(\text{nat} \Rightarrow_0 \text{nat}) \Rightarrow_0 'a$ later in theory *MPoly* gives a suitable representation type for polynomials *almost for free*: Interpreting $\text{nat} \Rightarrow_0 \text{nat}$ as a mapping from variable identifiers to exponents yields monomials, and the whole type maps monomials to coefficients. Lets call this the *ultimate interpretation*.
2. A further more specialised type isomorphic to $\text{nat} \Rightarrow_0 'a$ is apt to direct implementation using code generation [1].

Note that despite the names *mapping* and *lookup* suggest something implementation-near, it is best to keep $'a \Rightarrow_0 'b$ as an abstract *algebraic* type providing operations like *addition*, *multiplication* without any notion of key-order, data structures etc. This implementations-specific notions are easily introduced later for particular implementations but do not provide any gain for specifying logical properties of polynomials.

78.3 Additive structure

The additive structure covers the usual operations 0 , $+$ and (unary and binary) $-$. Recalling the ultimate interpretation, it is obvious that these have just lift the corresponding operations on values to mappings.

Isabelle has a rich hierarchy of algebraic type classes, and in such situations of pointwise lifting a typical pattern is to have instantiations for a considerable number of type classes.

The operations themselves are specified using *lift-definition*, where the proofs of the *almost everywhere zero* property can be significantly involved.

The *lookup* operation is supposed to be usable explicitly (unless in other situations where the morphisms between types are somehow internal to the *lifting* package). Hence it is good style to provide explicit rewrite rules how *lookup* acts on operations immediately.

instantiation *poly-mapping* :: (*type*, *zero*) *zero*
begin

lift-definition *zero-poly-mapping* :: '*a* \Rightarrow_0 '*b*
 is $\lambda k. 0$
 by *simp*

instance ..

end

lemma *Abs-poly-mapping* [*simp*]: *Abs-poly-mapping* ($\lambda k. 0$) = 0
 by (*simp add: zero-poly-mapping.abs-eq*)

lemma *lookup-zero* [*simp*]: *lookup* 0 *k* = 0
 by *transfer rule*

instantiation *poly-mapping* :: (*type*, *monoid-add*) *monoid-add*
begin

lift-definition *plus-poly-mapping* ::

(*'a* \Rightarrow_0 '*b*) \Rightarrow (*'a* \Rightarrow_0 '*b*) \Rightarrow '*a* \Rightarrow_0 '*b*

is $\lambda f1 f2 k. f1 k + f2 k$

proof –

fix *f1 f2* :: '*a* \Rightarrow '*b*

assume *finite* {*k*. *f1* *k* \neq 0}

and *finite* {*k*. *f2* *k* \neq 0}

then have *finite* ({*k*. *f1* *k* \neq 0} \cup {*k*. *f2* *k* \neq 0}) **by** *auto*

moreover have {*x*. *f1* *x* + *f2* *x* \neq 0} \subseteq {*k*. *f1* *k* \neq 0} \cup {*k*. *f2* *k* \neq 0}

by *auto*

ultimately show *finite* {*x*. *f1* *x* + *f2* *x* \neq 0}

by (*blast intro: finite-subset*)

qed

instance

by *intro-classes* (*transfer*, *simp add: fun-eq-iff ac-simps*)+

end

lemma *lookup-add*: *lookup* (*f* + *g*) *k* = *lookup* *f* *k* + *lookup* *g* *k*
 by (*simp add: plus-poly-mapping.rep-eq*)

instance *poly-mapping* :: (type, comm-monoid-add) comm-monoid-add
 by *intro-classes* (transfer, simp add: fun-eq-iff ac-simps)+

lemma *lookup-sum*: lookup (sum pp X) i = sum (λx . lookup (pp x) i) X
 by (induction rule: infinite-finite-induct) (auto simp: lookup-add)

instantiation *poly-mapping* :: (type, cancel-comm-monoid-add) cancel-comm-monoid-add
 begin

lift-definition *minus-poly-mapping* :: ($'a \Rightarrow_0 'b$) \Rightarrow ($'a \Rightarrow_0 'b$) \Rightarrow $'a \Rightarrow_0 'b$
 is $\lambda f1 f2 k. f1 k - f2 k$

proof –

fix *f1 f2* :: $'a \Rightarrow 'b$

assume *finite* {*k*. *f1 k* $\neq 0$ }

and *finite* {*k*. *f2 k* $\neq 0$ }

then have *finite* ({*k*. *f1 k* $\neq 0$ } \cup {*k*. *f2 k* $\neq 0$ }) by auto

moreover have {*x*. *f1 x* – *f2 x* $\neq 0$ } \subseteq {*k*. *f1 k* $\neq 0$ } \cup {*k*. *f2 k* $\neq 0$ }

by auto

ultimately show *finite* {*x*. *f1 x* – *f2 x* $\neq 0$ } by (blast intro: finite-subset)

qed

instance

by *intro-classes* (transfer, simp add: fun-eq-iff diff-diff-add)+

end

instantiation *poly-mapping* :: (type, ab-group-add) ab-group-add
 begin

lift-definition *uminus-poly-mapping* :: ($'a \Rightarrow_0 'b$) \Rightarrow $'a \Rightarrow_0 'b$

is *uminus*

by *simp*

instance

by *intro-classes* (transfer, simp add: fun-eq-iff ac-simps)+

end

lemma *lookup-uminus* [*simp*]:

lookup (– *f*) *k* = – lookup *f k*

by transfer *simp*

lemma *lookup-minus*:

lookup (*f* – *g*) *k* = lookup *f k* – lookup *g k*

by transfer rule

78.4 Multiplicative structure

instantiation *poly-mapping* :: (zero, zero-neq-one) zero-neq-one
begin

lift-definition *one-poly-mapping* :: 'a \Rightarrow_0 'b
is $\lambda k. 1$ when $k = 0$
by *simp*

instance
by *intro-classes (transfer, simp add: fun-eq-iff)*

end

lemma *lookup-one*: *lookup 1 k = (1 when k = 0)*
by (*meson one-poly-mapping.rep-eq*)

lemma *lookup-one-zero* [*simp*]:
lookup 1 0 = 1
by (*simp add: one-poly-mapping.rep-eq*)

definition *prod-fun* :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a::monoid-add \Rightarrow 'b::semiring-0
where

$$\text{prod-fun } f1 \ f2 \ k = (\sum l. f1 \ l * (\sum q. (f2 \ q \text{ when } k = l + q)))$$

lemma *prod-fun-unfold-prod*:
fixes *f g* :: 'a :: monoid-add \Rightarrow 'b::semiring-0
assumes *fin-f*: *finite {a. f a \neq 0}*
assumes *fin-g*: *finite {b. g b \neq 0}*
shows *prod-fun f g k = ($\sum (a, b). f a * g b$ when $k = a + b$)*
proof –
let *?C* = *{a. f a \neq 0} \times {b. g b \neq 0}*
from *fin-f fin-g* **have** *finite ?C* **by** *blast*
moreover
have *{a. $\exists b. (f a * g b$ when $k = a + b$) \neq 0} \times*
*{b. $\exists a. (f a * g b$ when $k = a + b$) \neq 0} \subseteq {a. f a \neq 0} \times {b. g b \neq 0}*
by *auto*
ultimately show *?thesis* **using** *fin-g*
by (*auto simp: prod-fun-def*
Sum-any.cartesian-product [of {a. f a \neq 0} \times {b. g b \neq 0}] Sum-any-right-distrib
mult-when)
qed

lemma *finite-prod-fun*:
fixes *f1 f2* :: 'a :: monoid-add \Rightarrow 'b :: semiring-0
assumes *fin1*: *finite {l. f1 l \neq 0}*
and *fin2*: *finite {q. f2 q \neq 0}*
shows *finite {k. prod-fun f1 f2 k \neq 0}*
proof –
have *: *finite {k. ($\exists l. f1 \ l \neq 0 \wedge (\exists q. f2 \ q \neq 0 \wedge k = l + q)$)}*

```

    using assms by simp
  have aux: sum f2 {q. f2 q ≠ 0 ∧ k = l + q} = (∑ q. (f2 q when k = l + q))
for k l
proof -
  have {q. (f2 q when k = l + q) ≠ 0} ⊆ {q. f2 q ≠ 0 ∧ k = l + q} by auto
  with fin2 show ?thesis
    by (simp add: Sum-any.expand-superset [of {q. f2 q ≠ 0 ∧ k = l + q}])
qed
have {k. (∑ l. f1 l * sum f2 {q. f2 q ≠ 0 ∧ k = l + q}) ≠ 0}
  ⊆ {k. (∃ l. f1 l * sum f2 {q. f2 q ≠ 0 ∧ k = l + q} ≠ 0)}
  by (auto elim!: Sum-any.not-neutral-obtains-not-neutral)
also have ... ⊆ {k. (∃ l. f1 l ≠ 0 ∧ sum f2 {q. f2 q ≠ 0 ∧ k = l + q} ≠ 0)}
  by (auto dest: mult-not-zero)
also have ... ⊆ {k. (∃ l. f1 l ≠ 0 ∧ (∃ q. f2 q ≠ 0 ∧ k = l + q))}
  by (auto elim!: sum.not-neutral-contains-not-neutral)
finally have finite {k. (∑ l. f1 l * sum f2 {q. f2 q ≠ 0 ∧ k = l + q}) ≠ 0}
  using * by (rule finite-subset)
with aux have finite {k. (∑ l. f1 l * (∑ q. (f2 q when k = l + q))) ≠ 0}
  by simp
with fin2 show ?thesis
  by (simp add: prod-fun-def)
qed

```

instantiation *poly-mapping* :: (monoid-add, semiring-0) semiring-0
begin

lift-definition *times-poly-mapping* :: ('a ⇒₀ 'b) ⇒ ('a ⇒₀ 'b) ⇒ 'a ⇒₀ 'b
 is prod-fun
 by(rule finite-prod-fun)

instance

```

proof
  fix a b c :: 'a ⇒0 'b
  show a * b * c = a * (b * c)
proof transfer
    fix f g h :: 'a ⇒ 'b
    assume fin-f: finite {a. f a ≠ 0} (is finite ?F)
    assume fin-g: finite {b. g b ≠ 0} (is finite ?G)
    assume fin-h: finite {c. h c ≠ 0} (is finite ?H)
    from fin-f fin-g have fin-fg: finite {(a, b). f a * g b ≠ 0} (is finite ?FG)
      by (rule finite-mult-not-eq-zero-prodI)
    from fin-g fin-h have fin-gh: finite {(b, c). g b * h c ≠ 0} (is finite ?GH)
      by (rule finite-mult-not-eq-zero-prodI)
    from fin-f fin-g have fin-fg': finite {a + b | a b. f a * g b ≠ 0} (is finite ?FG')
      by (rule finite-mult-not-eq-zero-sumI)
    then have fin-fg'': finite {d. (∑ (a, b). f a * g b when d = a + b) ≠ 0}
      by (auto intro: finite-Sum-any-not-eq-zero-weakenI)
    from fin-g fin-h have fin-gh': finite {b + c | b c. g b * h c ≠ 0} (is finite ?GH')
      by (rule finite-mult-not-eq-zero-sumI)
  
```

```

then have fin-gh'': finite {d. ( $\sum (b, c).$  g b * h c when d = b + c)  $\neq 0$ }
  by (auto intro: finite-Sum-any-not-eq-zero-weakenI)
show prod-fun (prod-fun f g) h = prod-fun f (prod-fun g h) (is ?lhs = ?rhs)
proof
  fix k
  from fin-f fin-g fin-h fin-fg''
  have ?lhs k = ( $\sum (ab, c).$  ( $\sum (a, b).$  f a * g b when ab = a + b) * h c when
k = ab + c)
    by (simp add: prod-fun-unfold-prod)
  also have ... = ( $\sum (ab, c).$  ( $\sum (a, b).$  f a * g b * h c when k = ab + c when
ab = a + b))
    using fin-fg
  apply (simp add: Sum-any-left-distrib split-def flip: Sum-any-when-independent)
  apply (simp add: when-when when-mult mult-when conj-commute)
  done
  also have ... = ( $\sum (ab, c, a, b).$  f a * g b * h c when k = ab + c when ab
= a + b)
    apply (subst Sum-any.cartesian-product2 [of (?FG'  $\times$  ?H)  $\times$  ?FG])
  apply (auto simp: finite-cartesian-product-iff fin-fg fin-fg' fin-h dest: mult-not-zero)
  done
  also have ... = ( $\sum (ab, c, a, b).$  f a * g b * h c when k = a + b + c when
ab = a + b)
    by (rule Sum-any.cong) (simp add: split-def when-def)
  also have ... = ( $\sum (ab, cab).$  (case cab of (c, a, b)  $\Rightarrow$  f a * g b * h c when
k = a + b + c)
    when ab = (case cab of (c, a, b)  $\Rightarrow$  a + b))
    by (simp add: split-def)
  also have ... = ( $\sum (c, a, b).$  f a * g b * h c when k = a + b + c)
    by (simp add: Sum-any-when-dependent-prod-left)
  also have ... = ( $\sum (bc, cab).$  (case cab of (c, a, b)  $\Rightarrow$  f a * g b * h c when k
= a + b + c)
    when bc = (case cab of (c, a, b)  $\Rightarrow$  b + c))
    by (simp add: Sum-any-when-dependent-prod-left)
  also have ... = ( $\sum (bc, c, a, b).$  f a * g b * h c when k = a + b + c when
bc = b + c)
    by (simp add: split-def)
  also have ... = ( $\sum (bc, c, a, b).$  f a * g b * h c when bc = b + c when k =
a + bc)
    by (rule Sum-any.cong) (simp add: split-def when-def ac-simps)
  also have ... = ( $\sum (a, bc, b, c).$  f a * g b * h c when bc = b + c when k =
a + bc)
    proof –
      have bij ( $\lambda(a, d, b, c).$  (d, c, a, b))
        by (auto intro!: bijI injI surjI [of -  $\lambda(d, c, a, b).$  (a, d, b, c)] simp add:
split-def)
      then show ?thesis
        by (rule Sum-any.reindex-cong) auto
    qed
  also have ... = ( $\sum (a, bc).$  ( $\sum (b, c).$  f a * g b * h c when bc = b + c when

```

```

k = a + bc))
  apply (subst Sum-any.cartesian-product2 [of (?F × ?GH') × ?GH])
  apply (auto simp: finite-cartesian-product-iff fin-f fin-gh fin-gh' ac-simps
dest: mult-not-zero)
done
also have ... = (∑ (a, bc). f a * (∑ (b, c). g b * h c when bc = b + c) when
k = a + bc)
  apply (subst Sum-any-right-distrib)
  using fin-gh apply (simp add: split-def)
  apply (subst Sum-any-when-independent [symmetric])
  apply (simp add: when-when when-mult mult-when split-def ac-simps)
  done
also from fin-f fin-g fin-h fin-gh''
have ... = ?rhs k
  by (simp add: prod-fun-unfold-prod)
finally show ?lhs k = ?rhs k .
qed
qed
show (a + b) * c = a * c + b * c
proof transfer
  fix f g h :: 'a ⇒ 'b
  assume fin-f: finite {k. f k ≠ 0}
  assume fin-g: finite {k. g k ≠ 0}
  assume fin-h: finite {k. h k ≠ 0}
  show prod-fun (λk. f k + g k) h = (λk. prod-fun f h k + prod-fun g h k)
    apply (rule ext)
    apply (simp add: prod-fun-def algebra-simps)
    by (simp add: Sum-any.distrib fin-f fin-g finite-mult-not-eq-zero-rightI)
qed
show a * (b + c) = a * b + a * c
proof transfer
  fix f g h :: 'a ⇒ 'b
  assume fin-f: finite {k. f k ≠ 0}
  assume fin-g: finite {k. g k ≠ 0}
  assume fin-h: finite {k. h k ≠ 0}
  show prod-fun f (λk. g k + h k) = (λk. prod-fun f g k + prod-fun f h k)
    apply (rule ext)
    apply (auto simp: prod-fun-def Sum-any.distrib algebra-simps when-add-distrib
fin-g fin-h)
    by (simp add: Sum-any.distrib fin-f finite-mult-not-eq-zero-rightI)
qed
show 0 * a = 0
  by transfer (simp add: prod-fun-def [abs-def])
show a * 0 = 0
  by transfer (simp add: prod-fun-def [abs-def])
qed
end

```

lemma *lookup-mult*:

lookup ($f * g$) $k = (\sum l. \text{lookup } f \ l * (\sum q. \text{lookup } g \ q \text{ when } k = l + q))$

by *transfer* (*simp add: prod-fun-def*)

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0*) *comm-semiring-0*

proof

fix $a \ b \ c :: 'a \Rightarrow_0 'b$

show $a * b = b * a$

proof *transfer*

fix $f \ g :: 'a \Rightarrow 'b$

assume *fin-f*: *finite* $\{a. f \ a \neq 0\}$

assume *fin-g*: *finite* $\{b. g \ b \neq 0\}$

show *prod-fun* $f \ g = \text{prod-fun } g \ f$

proof

fix k

have *fin1*: $\bigwedge l. \text{finite } \{a. (f \ a \text{ when } k = l + a) \neq 0\}$

using *fin-f* **by** *auto*

have *fin2*: $\bigwedge l. \text{finite } \{b. (g \ b \text{ when } k = l + b) \neq 0\}$

using *fin-g* **by** *auto*

from *fin-f fin-g* **have** *finite* $\{(a, b). f \ a \neq 0 \wedge g \ b \neq 0\}$ (**is** *finite* ?*AB*)

by *simp*

have $(\sum a. \sum n. f \ a * (g \ n \text{ when } k = a + n)) = (\sum a. \sum n. g \ a * (f \ n \text{ when } k = a + n))$

by (*subst Sum-any.swap* [*OF* $\langle \text{finite } ?AB \rangle$]) (*auto simp: mult-when ac-simps*)

then show *prod-fun* $f \ g = \text{prod-fun } g \ f \ k$

by (*simp add: prod-fun-def Sum-any-right-distrib* [*OF fin2*] *Sum-any-right-distrib* [*OF fin1*])

qed

qed

show $(a + b) * c = a * c + b * c$

proof *transfer*

fix $f \ g \ h :: 'a \Rightarrow 'b$

assume *fin-f*: *finite* $\{k. f \ k \neq 0\}$

assume *fin-g*: *finite* $\{k. g \ k \neq 0\}$

assume *fin-h*: *finite* $\{k. h \ k \neq 0\}$

show *prod-fun* $(\lambda k. f \ k + g \ k) \ h = (\lambda k. \text{prod-fun } f \ h \ k + \text{prod-fun } g \ h \ k)$

by (*auto simp: prod-fun-def fun-eq-iff algebra-simps*

Sum-any.distrib fin-f fin-g finite-mult-not-eq-zero-rightI)

qed

qed

instance *poly-mapping* :: (*monoid-add*, *semiring-0-cancel*) *semiring-0-cancel*

..

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel*

..

instance *poly-mapping* :: (*monoid-add*, *semiring-1*) *semiring-1*

proof

```

fix a :: 'a  $\Rightarrow_0$  'b
show 1 * a = a
  by transfer (simp add: prod-fun-def [abs-def] when-mult)
show a * 1 = a
  apply transfer
  apply (simp add: prod-fun-def [abs-def] Sum-any-right-distrib Sum-any-left-distrib
mult-when)
  apply (subst when-commute)
  apply simp
  done
qed

```

```

instance poly-mapping :: (comm-monoid-add, comm-semiring-1) comm-semiring-1
proof
  fix a :: 'a  $\Rightarrow_0$  'b
  show 1 * a = a
    by transfer (simp add: prod-fun-def [abs-def])
qed

```

```

instance poly-mapping :: (monoid-add, semiring-1-cancel) semiring-1-cancel
..

```

```

instance poly-mapping :: (monoid-add, ring) ring
..

```

```

instance poly-mapping :: (comm-monoid-add, comm-ring) comm-ring
..

```

```

instance poly-mapping :: (monoid-add, ring-1) ring-1
..

```

```

instance poly-mapping :: (comm-monoid-add, comm-ring-1) comm-ring-1
..

```

78.5 Single-point mappings

```

lift-definition single :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'a  $\Rightarrow_0$  'b::zero
  is  $\lambda k v k'. (v \text{ when } k = k')$ 
  by simp

```

lemma inj-single [iff]:

```

  inj (single k)
proof (rule injI, transfer)
  fix k :: 'b and a b :: 'a::zero
  assume ( $\lambda k'. a \text{ when } k = k' = (\lambda k'. b \text{ when } k = k')$ )
  then have ( $\lambda k'. a \text{ when } k = k' = (\lambda k'. b \text{ when } k = k')$ ) k
    by (rule arg-cong)
  then show a = b by simp
qed

```


lemma *lookup-single*:

lookup (single k v) k' = (v when k = k')
by (*simp add: single.rep-eq*)

lemma *lookup-single-eq [simp]*:

lookup (single k v) k = v
by (*simp add: single.rep-eq*)

lemma *lookup-single-not-eq*:

$k \neq k' \implies \text{lookup (single k v) } k' = 0$
by (*simp add: single.rep-eq*)

lemma *single-zero [simp]*:

single k 0 = 0
by *transfer simp*

lemma *single-one [simp]*:

single 0 1 = 1
by *transfer simp*

lemma *single-add*:

single k (a + b) = single k a + single k b
by *transfer (simp add: fun-eq-iff when-add-distrib)*

lemma *single-uminus*:

single k (- a) = - single k a
by *transfer (simp add: fun-eq-iff when-uminus-distrib)*

lemma *single-diff*:

single k (a - b) = single k a - single k b
by *transfer (simp add: fun-eq-iff when-diff-distrib)*

lemma *single-numeral [simp]*:

single 0 (numeral n) = numeral n
by (*induct n*) (*simp-all only: numeral.simps numeral-add single-zero single-one single-add*)

lemma *lookup-numeral*:

lookup (numeral n) k = (numeral n when k = 0)

proof –

have *lookup (numeral n) k = lookup (single 0 (numeral n)) k*
by *simp*

then show *?thesis unfolding lookup-single by simp*

qed

lemma *single-of-nat [simp]*:

single 0 (of-nat n) = of-nat n
by (*induct n*) (*simp-all add: single-add*)

lemma *lookup-of-nat*:

lookup (of-nat n) k = (of-nat n when k = 0)

by (*metis lookup-single lookup-single-not-eq single-of-nat*)

lemma *of-nat-single*:

of-nat = single 0 \circ of-nat

by (*simp add: fun-eq-iff*)

lemma *mult-single*:

*single k a * single l b = single (k + l) (a * b)*

proof *transfer*

fix *k l :: 'a and a b :: 'b*

show *prod-fun ($\lambda k'. a$ when $k = k'$) ($\lambda k'. b$ when $l = k'$) = ($\lambda k'. a * b$ when $k + l = k'$)*

proof

fix *k'*

have *prod-fun ($\lambda k'. a$ when $k = k'$) ($\lambda k'. b$ when $l = k'$) $k' = (\sum n. a * b$ when $l = n$ when $k' = k + n)$*

by (*simp add: prod-fun-def Sum-any-right-distrib mult-when when-mult*)

also have *... = ($\sum n. a * b$ when $k' = k + n$ when $l = n)$*

by (*simp add: when-when conj-commute*)

also have *... = (a * b when $k' = k + l$)*

by *simp*

also have *... = (a * b when $k + l = k'$)*

by (*simp add: when-def*)

finally show *prod-fun ($\lambda k'. a$ when $k = k'$) ($\lambda k'. b$ when $l = k'$) $k' = (\lambda k'. a * b$ when $k + l = k')$ k' .*

qed

qed

instance *poly-mapping* :: (*monoid-add, semiring-char-0*) *semiring-char-0*

by *intro-classes (auto intro: inj-compose inj-of-nat simp add: of-nat-single)*

instance *poly-mapping* :: (*monoid-add, ring-char-0*) *ring-char-0*

..

lemma *single-of-int* [*simp*]:

single 0 (of-int k) = of-int k

by (*cases k (simp-all add: single-diff single-uminus)*)

lemma *lookup-of-int*:

lookup (of-int l) k = (of-int l when k = 0)

by (*metis lookup-single-not-eq single.rep-eq single-of-int*)

78.6 Integral domains

instance *poly-mapping* :: (*{ ordered-cancel-comm-monoid-add, linorder }*, *semiring-no-zero-divisors*) *semiring-no-zero-divisors*

The *linorder* constraint is a pragmatic device for the proof — maybe it can be dropped

proof

```

fix  $f\ g :: 'a \Rightarrow_0 'b$ 
assume  $f \neq 0$  and  $g \neq 0$ 
then show  $f * g \neq 0$ 
proof transfer
  fix  $f\ g :: 'a \Rightarrow 'b$ 
  define  $F$  where  $F = \{a. f\ a \neq 0\}$ 
  moreover define  $G$  where  $G = \{a. g\ a \neq 0\}$ 
  ultimately have  $[simp]$ :
     $\bigwedge a. f\ a \neq 0 \longleftrightarrow a \in F$ 
     $\bigwedge b. g\ b \neq 0 \longleftrightarrow b \in G$ 
    by simp-all
  assume finite  $\{a. f\ a \neq 0\}$ 
  then have  $[simp]$ : finite  $F$ 
    by simp
  assume finite  $\{a. g\ a \neq 0\}$ 
  then have  $[simp]$ : finite  $G$ 
    by simp
  assume  $f \neq (\lambda a. 0)$ 
  then obtain  $a$  where  $f\ a \neq 0$ 
    by (auto simp: fun-eq-iff)
  assume  $g \neq (\lambda b. 0)$ 
  then obtain  $b$  where  $g\ b \neq 0$ 
    by (auto simp: fun-eq-iff)
  from  $\langle f\ a \neq 0 \rangle$  and  $\langle g\ b \neq 0 \rangle$  have  $F \neq \{\}$  and  $G \neq \{\}$ 
    by auto
  note  $Max\ F = \langle finite\ F \rangle \langle F \neq \{\} \rangle$ 
  note  $Max\ G = \langle finite\ G \rangle \langle G \neq \{\} \rangle$ 
  from  $Max\ F$  and  $Max\ G$  have  $[simp]$ :
     $Max\ F \in F$ 
     $Max\ G \in G$ 
    by auto
  from  $Max\ F\ Max\ G$  have  $[dest!]$ :
     $\bigwedge a. a \in F \implies a \leq Max\ F$ 
     $\bigwedge b. b \in G \implies b \leq Max\ G$ 
    by auto
  define  $q$  where  $q = Max\ F + Max\ G$ 
  have  $(\sum (a, b). f\ a * g\ b\ when\ q = a + b) =$ 
     $(\sum (a, b). f\ a * g\ b\ when\ q = a + b\ when\ a \in F \wedge b \in G)$ 
    by (rule Sum-any.cong) (auto simp: split-def when-def q-def intro: ccontr)
  also have  $\dots =$ 
     $(\sum (a, b). f\ a * g\ b\ when\ (Max\ F, Max\ G) = (a, b))$ 
  proof (rule Sum-any.cong)
    fix  $ab :: 'a \times 'a$ 
    obtain  $a\ b$  where  $[simp]$ :  $ab = (a, b)$ 
      by (cases ab) simp-all
    have  $[dest!]$ :

```

```

    a ≤ Max F ⇒ Max F ≠ a ⇒ a < Max F
    b ≤ Max G ⇒ Max G ≠ b ⇒ b < Max G
  by auto
  show (case ab of (a, b) ⇒ f a * g b when q = a + b when a ∈ F ∧ b ∈ G) =
    (case ab of (a, b) ⇒ f a * g b when (Max F, Max G) = (a, b))
  by (auto simp: split-def when-def q-def dest: add-strict-mono [of a Max F b
Max G])
qed
also have ... = (∑ ab. (case ab of (a, b) ⇒ f a * g b) when
(Max F, Max G) = ab)
  unfolding split-def when-def by auto
also have ... ≠ 0
  by simp
finally have prod-fun f g q ≠ 0
  by (simp add: prod-fun-unfold-prod)
then show prod-fun f g ≠ (λk. 0)
  by (auto simp: fun-eq-iff)
qed
qed

```

```

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, ring-no-zero-divisors)
ring-no-zero-divisors
..

```

```

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, ring-1-no-zero-divisors)
ring-1-no-zero-divisors
..

```

```

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, idom) idom
..

```

78.7 Mapping order

```

instantiation poly-mapping :: (linorder, {zero, linorder}) linorder
begin

```

```

lift-definition less-poly-mapping :: ('a ⇒₀ 'b) ⇒ ('a ⇒₀ 'b) ⇒ bool
is less-fun
.

```

```

lift-definition less-eq-poly-mapping :: ('a ⇒₀ 'b) ⇒ ('a ⇒₀ 'b) ⇒ bool
is λf g. less-fun f g ∨ f = g
.

```

```

instance proof (rule linorder.intro-of-class)
show class.linorder (less-eq :: (- ⇒₀ -) ⇒ -) less
proof (rule linorder-strictI, rule order-strictI)
fix f g h :: 'a ⇒₀ 'b
show f ≤ g ⟷ f < g ∨ f = g

```

```

    by transfer (rule refl)
  show  $\neg f < f$ 
    by transfer (rule less-fun-irrefl)
  show  $f < g \vee f = g \vee g < f$ 
  proof transfer
    fix f g :: 'a  $\Rightarrow$  'b
    assume finite {k. f k  $\neq$  0} and finite {k. g k  $\neq$  0}
    then have finite ({k. f k  $\neq$  0}  $\cup$  {k. g k  $\neq$  0})
      by simp
    moreover have {k. f k  $\neq$  g k}  $\subseteq$  {k. f k  $\neq$  0}  $\cup$  {k. g k  $\neq$  0}
      by auto
    ultimately have finite {k. f k  $\neq$  g k}
      by (rule rev-finite-subset)
    then show less-fun f g  $\vee f = g \vee$  less-fun g f
      by (rule less-fun-trichotomy)
  qed
  assume f < g then show  $\neg g < f$ 
    by transfer (rule less-fun-asym)
  note <f < g> moreover assume g < h
    ultimately show f < h
      by transfer (rule less-fun-trans)
  qed
qed

end

instance poly-mapping :: (linorder, {ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le,
linorder}) ordered-ab-semigroup-add
proof (intro-classes, transfer)
  fix f g h :: 'a  $\Rightarrow$  'b
  assume *: less-fun f g  $\vee f = g$ 
  have less-fun ( $\lambda k. h k + f k$ ) ( $\lambda k. h k + g k$ ) if less-fun f g
    by (metis (no-types, lifting) less-fun-def add-strict-left-mono that)
  with * show less-fun ( $\lambda k. h k + f k$ ) ( $\lambda k. h k + g k$ )  $\vee$  ( $\lambda k. h k + f k$ ) = ( $\lambda k. h k + g k$ )
    by (auto simp: fun-eq-iff)
qed

instance poly-mapping :: (linorder, {ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le,
cancel-comm-monoid-add, linorder}) linordered-cancel-ab-semigroup-add
..

instance poly-mapping :: (linorder, {ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le,
cancel-comm-monoid-add, linorder}) ordered-comm-monoid-add
..

instance poly-mapping :: (linorder, {ordered-comm-monoid-add, ordered-ab-semigroup-add-imp-le,
cancel-comm-monoid-add, linorder}) ordered-cancel-comm-monoid-add
..

```

instance *poly-mapping* :: (*linorder*, *linordered-ab-group-add*) *linordered-ab-group-add*
 ..

For pragmatism we leave out the final elements in the hierarchy: *linordered-ring*, *linordered-ring-strict*, *linordered-idom*; remember that the order instance is a mere technical device, not a deeper algebraic property.

78.8 Fundamental mapping notions

lift-definition *keys* :: ($'a \Rightarrow_0 'b::\text{zero}$) $\Rightarrow 'a \text{ set}$
is $\lambda f. \{k. f\ k \neq 0\}$.

lift-definition *range* :: ($'a \Rightarrow_0 'b::\text{zero}$) $\Rightarrow 'b \text{ set}$
is $\lambda f :: 'a \Rightarrow 'b. \text{Set.range } f - \{0\}$.

lemma *finite-keys* [*simp*]:
finite (*keys* *f*)
by *transfer*

lemma *not-in-keys-iff-lookup-eq-zero*:
 $k \notin \text{keys } f \longleftrightarrow \text{lookup } f\ k = 0$
by *transfer simp*

lemma *lookup-not-eq-zero-eq-in-keys*:
 $\text{lookup } f\ k \neq 0 \longleftrightarrow k \in \text{keys } f$
by *transfer simp*

lemma *lookup-eq-zero-in-keys-contradict* [*dest*]:
 $\text{lookup } f\ k = 0 \implies \neg k \in \text{keys } f$
by (*simp add: not-in-keys-iff-lookup-eq-zero*)

lemma *finite-range* [*simp*]: *finite* (*Poly-Mapping.range* *p*)
proof *transfer*
fix *f* :: $'b \Rightarrow 'a$
assume *: *finite* $\{x. f\ x \neq 0\}$
have $\text{Set.range } f - \{0\} \subseteq f^{-1} \{x. f\ x \neq 0\}$
by *auto*
thus *finite* ($\text{Set.range } f - \{0\}$)
using * *finite-surj* **by** *blast*
qed

lemma *in-keys-lookup-in-range* [*simp*]:
 $k \in \text{keys } f \implies \text{lookup } f\ k \in \text{range } f$
by *transfer simp*

lemma *in-keys-iff*: $x \in (\text{keys } s) = (\text{lookup } s\ x \neq 0)$
by (*simp add: lookup-not-eq-zero-eq-in-keys*)

lemma *keys-zero* [*simp*]:

keys 0 = {}

by *transfer simp*

lemma *range-zero* [*simp*]:

range 0 = {}

by *transfer auto*

lemma *keys-add*:

keys (f + g) \subseteq *keys* f \cup *keys* g

by *transfer auto*

lemma *keys-one* [*simp*]:

keys 1 = {0}

by *transfer simp*

lemma *range-one* [*simp*]:

range 1 = {1}

by *transfer (auto simp: when-def)*

lemma *keys-single* [*simp*]:

keys (single k v) = (if v = 0 then {} else {k})

by *transfer simp*

lemma *range-single* [*simp*]:

range (single k v) = (if v = 0 then {} else {v})

by *transfer (auto simp: when-def)*

lemma *keys-mult*:

keys (f * g) \subseteq {a + b | a b. a \in *keys* f \wedge b \in *keys* g}

apply *transfer*

apply (force *simp: prod-fun-def dest!: mult-not-zero elim!: Sum-any.not-neutral-obtains-not-neutral*)

done

lemma *setsum-keys-plus-distrib*:

assumes *hom-0*: $\bigwedge k. f\ k\ 0 = 0$

and *hom-plus*: $\bigwedge k. k \in \text{Poly-Mapping.keys } p \cup \text{Poly-Mapping.keys } q \implies f\ k$
 (*Poly-Mapping.lookup* p k + *Poly-Mapping.lookup* q k) = f k (*Poly-Mapping.lookup* p k) + f k (*Poly-Mapping.lookup* q k)

shows

$(\sum_{k \in \text{Poly-Mapping.keys } (p + q)}. f\ k\ (\text{Poly-Mapping.lookup } (p + q)\ k)) =$

$(\sum_{k \in \text{Poly-Mapping.keys } p}. f\ k\ (\text{Poly-Mapping.lookup } p\ k)) +$

$(\sum_{k \in \text{Poly-Mapping.keys } q}. f\ k\ (\text{Poly-Mapping.lookup } q\ k))$

(**is** ?lhs = ?p + ?q)

proof –

let ?A = *Poly-Mapping.keys* p \cup *Poly-Mapping.keys* q

have ?lhs = $(\sum_{k \in ?A}. f\ k\ (\text{Poly-Mapping.lookup } p\ k + \text{Poly-Mapping.lookup } q\ k))$

by(intro *sum.mono-neutral-cong-left*) (*auto simp: sum.mono-neutral-cong-left*)

```

hom-0 in-keys-iff lookup-add)
  also have ... = ( $\sum k \in ?A. f k (Poly-Mapping.lookup p k) + f k (Poly-Mapping.lookup q k)$ )
  by (rule sum.cong) (simp-all add: hom-plus)
  also have ... = ( $\sum k \in ?A. f k (Poly-Mapping.lookup p k)$ ) + ( $\sum k \in ?A. f k (Poly-Mapping.lookup q k)$ )
  (is - = ?p' + ?q')
  by (simp add: sum.distrib)
  also have ?p' = ?p
  by (simp add: hom-0 in-keys-iff sum.mono-neutral-cong-right)
  also have ?q' = ?q
  by (simp add: hom-0 in-keys-iff sum.mono-neutral-cong-right)
  finally show ?thesis .
qed

```

78.9 Degree

definition *degree* :: ($nat \Rightarrow_0 'a :: zero$) $\Rightarrow nat$
where
degree *f* = *Max* (*insert* 0 (*Suc* ‘ *keys* *f*))

lemma *degree-zero* [*simp*]:
degree 0 = 0
unfolding *degree-def* **by** *transfer simp*

lemma *degree-one* [*simp*]:
degree 1 = 1
unfolding *degree-def* **by** *transfer simp*

lemma *degree-single-zero* [*simp*]:
degree (*single* *k* 0) = 0
unfolding *degree-def* **by** *transfer simp*

lemma *degree-single-not-zero* [*simp*]:
 $v \neq 0 \implies \text{degree } (\text{single } k \ v) = \text{Suc } k$
unfolding *degree-def* **by** *transfer simp*

lemma *degree-zero-iff* [*simp*]:
degree *f* = 0 $\longleftrightarrow f = 0$
unfolding *degree-def* **proof** *transfer*
fix *f* :: $nat \Rightarrow 'a$
assume *finite* {*n*. *f* *n* $\neq 0$ }
then have *fin*: *finite* (*insert* 0 (*Suc* ‘ {*n*. *f* *n* $\neq 0$ })) **by** *auto*
show *Max* (*insert* 0 (*Suc* ‘ {*n*. *f* *n* $\neq 0$ })) = 0 $\longleftrightarrow f = (\lambda n. 0)$ (**is** ?*P* \longleftrightarrow ?*Q*)
proof
assume ?*P*
have {*n*. *f* *n* $\neq 0$ } = {}
proof (rule *ccontr*)
assume {*n*. *f* *n* $\neq 0$ } $\neq \{\}$


```

    then obtain  $n$  where  $n \in \{n. f\ n \neq 0\}$  by blast
    then have  $\{n. f\ n \neq 0\} = \text{insert } n\ \{n. f\ n \neq 0\}$  by auto
    then have  $\text{Suc } ' \{n. f\ n \neq 0\} = \text{insert } (\text{Suc } n)\ (\text{Suc } ' \{n. f\ n \neq 0\})$  by auto
    with  $\langle ?P \rangle$  have  $\text{Max } (\text{insert } 0\ (\text{insert } (\text{Suc } n)\ (\text{Suc } ' \{n. f\ n \neq 0\}))) = 0$ 
  by simp
    then have  $\text{Max } (\text{insert } (\text{Suc } n)\ (\text{insert } 0\ (\text{Suc } ' \{n. f\ n \neq 0\}))) = 0$ 
    by (simp add: insert-commute)
    with fin have  $\text{max } (\text{Suc } n)\ (\text{Max } (\text{insert } 0\ (\text{Suc } ' \{n. f\ n \neq 0\}))) = 0$ 
    by simp
    then show False by simp
  qed
  then show  $?Q$  by (simp add: fun-eq-iff)
next
  assume  $?Q$  then show  $?P$  by simp
qed
qed

```

```

lemma degree-greater-zero-in-keys:
  assumes  $0 < \text{degree } f$ 
  shows  $\text{degree } f - 1 \in \text{keys } f$ 
proof -
  from assms have  $\text{keys } f \neq \{\}$ 
  by (auto simp: degree-def)
  then show  $?thesis$  unfolding degree-def
  by (simp add: mono-Max-commute [symmetric] mono-Suc)
qed

```

```

lemma in-keys-less-degree:
   $n \in \text{keys } f \implies n < \text{degree } f$ 
unfolding degree-def by transfer (auto simp: Max-gr-iff)

```

```

lemma beyond-degree-lookup-zero:
   $\text{degree } f \leq n \implies \text{lookup } f\ n = 0$ 
unfolding degree-def by transfer auto

```

```

lemma degree-add:
   $\text{degree } (f + g) \leq \max (\text{degree } f)\ (\text{Poly-Mapping.degree } g)$ 
unfolding degree-def proof transfer
  fix  $f\ g :: \text{nat} \Rightarrow 'a$ 
  assume  $f: \text{finite } \{x. f\ x \neq 0\}$ 
  assume  $g: \text{finite } \{x. g\ x \neq 0\}$ 
  let  $?f = \text{Max } (\text{insert } 0\ (\text{Suc } ' \{k. f\ k \neq 0\}))$ 
  let  $?g = \text{Max } (\text{insert } 0\ (\text{Suc } ' \{k. g\ k \neq 0\}))$ 
  have  $\text{Max } (\text{insert } 0\ (\text{Suc } ' \{k. f\ k + g\ k \neq 0\})) \leq \text{Max } (\text{insert } 0\ (\text{Suc } ' (\{k. f\ k \neq 0\} \cup \{k. g\ k \neq 0\})))$ 
  by (rule Max.subset-imp) (insert f g, auto)
  also have  $\dots = \max\ ?f\ ?g$ 
  using f g by (simp-all add: image-Un Max-Un [symmetric])
  finally show  $\text{Max } (\text{insert } 0\ (\text{Suc } ' \{k. f\ k + g\ k \neq 0\}))$ 

```

$$\leq \max (\text{Max} (\text{insert } 0 (\text{Suc } \{k. f\ k \neq 0\}))) (\text{Max} (\text{insert } 0 (\text{Suc } \{k. g\ k \neq 0\})))$$

qed

lemma *sorted-list-of-set-keys*:

sorted-list-of-set (keys f) = filter ($\lambda k. k \in \text{keys } f$) [0..*degree* f] (is - = ?r)

proof –

have keys f = set ?r

by (auto dest: in-keys-less-degree)

moreover have *sorted-list-of-set* (set ?r) = ?r

unfolding *sorted-list-of-set-sort-remdups*

by (simp add: remdups-filter filter-sort [symmetric])

ultimately show ?thesis by simp

qed

78.10 Inductive structure

lift-definition *update* :: 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow_0 'b::zero) \Rightarrow 'a \Rightarrow_0 'b

is $\lambda k\ v\ f. f(k := v)$

proof –

fix f :: 'a \Rightarrow 'b and k' v

assume finite {k. f k \neq 0}

then have finite (insert k' {k. f k \neq 0})

by simp

then show finite {k. (f(k' := v)) k \neq 0}

by (rule rev-finite-subset) auto

qed

lemma *update-induct* [case-names const *update*]:

assumes const': P 0

assumes *update*': $\bigwedge f\ a\ b. a \notin \text{keys } f \implies b \neq 0 \implies P\ f \implies P\ (\text{update } a\ b\ f)$

shows P f

proof –

obtain g where f = Abs-poly-mapping g and finite {a. g a \neq 0}

by (cases f) simp-all

define Q where Q g = P (Abs-poly-mapping g) for g

from $\langle \text{finite } \{a. g\ a \neq 0\} \rangle$ have Q g

proof (induct g rule: finite-update-induct)

case const with const' Q-def show ?case

by simp

next

case (update a b g)

from $\langle \text{finite } \{a. g\ a \neq 0\} \rangle \langle g\ a = 0 \rangle$ have $a \notin \text{keys } (\text{Abs-poly-mapping } g)$

by (simp add: Abs-poly-mapping-inverse keys.rep-eq)

moreover note $\langle b \neq 0 \rangle$

moreover from $\langle Q\ g \rangle$ have P (Abs-poly-mapping g)

by (simp add: Q-def)

ultimately have P (update a b (Abs-poly-mapping g))

```

    by (rule update')
  also from ⟨finite {a. g a ≠ 0}⟩
  have update a b (Abs-poly-mapping g) = Abs-poly-mapping (g(a := b))
    by (simp add: update.abs-eq eq-onp-same-args)
  finally show ?case
    by (simp add: Q-def fun-upd-def)
qed
then show ?thesis by (simp add: Q-def ⟨f = Abs-poly-mapping g⟩)
qed

```

lemma *lookup-update*:

```

lookup (update k v f) k' = (if k = k' then v else lookup f k')
by transfer simp

```

lemma *keys-update*:

```

keys (update k v f) = (if v = 0 then keys f - {k} else insert k (keys f))
by transfer auto

```

78.11 Quasi-functorial structure

```

lift-definition map :: ('b::zero ⇒ 'c::zero)
⇒ ('a ⇒0 'b) ⇒ ('a ⇒0 'c::zero)
is λg f k. g (f k) when f k ≠ 0
by simp

```

context

```

  fixes f :: 'b ⇒ 'a
  assumes inj-f: inj f
begin

```

```

lift-definition map-key :: ('a ⇒0 'c::zero) ⇒ 'b ⇒0 'c
is λp. p ∘ f

```

proof –

```

  fix g :: 'c ⇒ 'd and p :: 'a ⇒ 'c
  assume finite {x. p x ≠ 0}
  hence finite (f ` {y. p (f y) ≠ 0})
    by (simp add: rev-finite-subset subset-eq)
  thus finite {x. (p ∘ f) x ≠ 0} unfolding o-def
    by (metis finite-imageD injD inj-f inj-on-def)
qed

```

end

lemma *map-key-compose*:

```

  assumes [transfer-rule]: inj f inj g
  shows map-key f (map-key g p) = map-key (g ∘ f) p
proof –
  from assms have [transfer-rule]: inj (g ∘ f)
    by (simp add: inj-compose)

```

show *?thesis* **by** *transfer*(*simp add: o-assoc*)
qed

lemma *map-key-id*:
 $\text{map-key } (\lambda x. x) p = p$
proof –
have [*transfer-rule*]: *inj* ($\lambda x. x$) **by** *simp*
show *?thesis* **by** *transfer*(*simp add: o-def*)
qed

context
fixes $f :: 'a \Rightarrow 'b$
assumes *inj-f* [*transfer-rule*]: *inj f*
begin

lemma *map-key-map*:
 $\text{map-key } f (\text{map } g p) = \text{map } g (\text{map-key } f p)$
by *transfer* (*simp add: fun-eq-iff*)

lemma *map-key-plus*:
 $\text{map-key } f (p + q) = \text{map-key } f p + \text{map-key } f q$
by *transfer* (*simp add: fun-eq-iff*)

lemma *keys-map-key*:
 $\text{keys } (\text{map-key } f p) = f \text{ ` } \text{keys } p$
by *transfer auto*

lemma *map-key-zero* [*simp*]:
 $\text{map-key } f 0 = 0$
by *transfer* (*simp add: fun-eq-iff*)

lemma *map-key-single* [*simp*]:
 $\text{map-key } f (\text{single } (f k) v) = \text{single } k v$
by *transfer* (*simp add: fun-eq-iff inj-onD [OF inj-f] when-def*)

end

lemma *mult-map-scale-conv-mult*: $\text{map } ((*) s) p = \text{single } 0 s * p$
proof(*transfer fixing: s*)
fix $p :: 'a \Rightarrow 'b$
assume *: *finite* $\{x. p x \neq 0\}$
have *prod-fun* ($\lambda k'. s \text{ when } 0 = k'$) $p x = (\lambda k. s * p k \text{ when } p k \neq 0) x$ (**is** *?lhs*
 $= ?rhs$) **for** x
proof –
have *?lhs* $= (\sum l :: 'a. \text{if } l = 0 \text{ then } s * (\sum q. p q \text{ when } x = q) \text{ else } 0)$
by (*auto simp: prod-fun-def when-def intro: Sum-any.cong simp del: Sum-any.delta*)
also have $\dots = ?rhs$
by (*simp add: when-def*)
finally show *?thesis* .

qed
then show $(\lambda k. s * p \ k \text{ when } p \ k \neq 0) = \text{prod-fun } (\lambda k'. s \text{ when } 0 = k') \ p$
by $(\text{simp add: fun-eq-iff})$
qed

lemma *map-single* [simp]:
 $(c = 0 \implies f \ 0 = 0) \implies \text{map } f \ (\text{single } x \ c) = \text{single } x \ (f \ c)$
by $\text{transfer}(\text{auto simp: fun-eq-iff when-def})$

lemma *map-eq-zero-iff*: $\text{map } f \ p = 0 \longleftrightarrow (\forall k \in \text{keys } p. f \ (\text{lookup } p \ k) = 0)$
by $\text{transfer}(\text{auto simp: fun-eq-iff when-def})$

78.12 Canonical dense representation of $\text{nat} \Rightarrow_0 'a$

abbreviation *no-trailing-zeros* :: $'a :: \text{zero list} \Rightarrow \text{bool}$

where

$\text{no-trailing-zeros} \equiv \text{no-trailing } ((=) \ 0)$

lift-definition *nth* :: $'a \text{ list} \Rightarrow (\text{nat} \Rightarrow_0 'a :: \text{zero})$
is *nth-default* 0
by $(\text{fact finite-nth-default-neq-default})$

The opposite direction is directly specified on (later) type *nat-mapping*.

lemma *nth-Nil* [simp]:
 $\text{nth } [] = 0$
by $\text{transfer } (\text{simp add: fun-eq-iff})$

lemma *nth-singleton* [simp]:
 $\text{nth } [v] = \text{single } 0 \ v$
proof $(\text{transfer, rule ext})$
fix $n :: \text{nat}$ **and** $v :: 'a$
show $\text{nth-default } 0 \ [v] \ n = (v \text{ when } 0 = n)$
by $(\text{auto simp: nth-default-def nth-append})$
qed

lemma *nth-replicate* [simp]:
 $\text{nth } (\text{replicate } n \ 0 \ @ \ [v]) = \text{single } n \ v$
proof $(\text{transfer, rule ext})$
fix $m \ n :: \text{nat}$ **and** $v :: 'a$
show $\text{nth-default } 0 \ (\text{replicate } n \ 0 \ @ \ [v]) \ m = (v \text{ when } n = m)$
by $(\text{auto simp: nth-default-def nth-append})$
qed

lemma *nth-strip-while* [simp]:
 $\text{nth } (\text{strip-while } ((=) \ 0) \ xs) = \text{nth } xs$
by $\text{transfer } (\text{fact nth-default-strip-while-dflt})$

lemma *nth-strip-while'* [simp]:
 $\text{nth } (\text{strip-while } (\lambda k. k = 0) \ xs) = \text{nth } xs$
by $(\text{subst eq-commute}) \ (\text{fact nth-strip-while})$

lemma *nth-eq-iff*:

$nth\ xs = nth\ ys \longleftrightarrow strip_while\ (HOL.eq\ 0)\ xs = strip_while\ (HOL.eq\ 0)\ ys$
by *transfer (simp add: nth-default-eq-iff)*

lemma *lookup-nth [simp]*:

$lookup\ (nth\ xs) = nth_default\ 0\ xs$
by *(fact nth.rep-eq)*

lemma *keys-nth [simp]*:

$keys\ (nth\ xs) = fst\ ' \{(n, v) \in set\ (enumerate\ 0\ xs). v \neq 0\}$

proof *transfer*

fix $xs :: 'a\ list$

have $n \in fst\ ' \{(n, v). (n, v) \in set\ (enumerate\ 0\ xs) \wedge v \neq 0\}$

if $nth_default\ 0\ xs\ n \neq 0$ **for** n

proof $-$

from *that* **have** $n < length\ xs$ **and** $xs\ !\ n \neq 0$

by *(auto simp: nth-default-def split: if-splits)*

then have $(n, xs\ !\ n) \in \{(n, v). (n, v) \in set\ (enumerate\ 0\ xs) \wedge v \neq 0\}$ **(is**

$?x \in ?A)$

by *(auto simp: in-set-conv-nth enumerate-eq-zip)*

then have $fst\ ?x \in fst\ ' \ ?A$

by *blast*

then show $?thesis$

by *simp*

qed

then show $\{k. nth_default\ 0\ xs\ k \neq 0\} = fst\ ' \{(n, v). (n, v) \in set\ (enumerate\ 0\ xs) \wedge v \neq 0\}$

by *(auto simp: in-enumerate-iff-nth-default-eq)*

qed

lemma *range-nth [simp]*:

$range\ (nth\ xs) = set\ xs - \{0\}$
by *transfer simp*

lemma *degree-nth*:

$no_trailing_zeros\ xs \implies degree\ (nth\ xs) = length\ xs$

unfolding *degree-def* **proof** *transfer*

fix $xs :: 'a\ list$

assume $*$: *no-trailing-zeros xs*

let $?A = \{n. nth_default\ 0\ xs\ n \neq 0\}$

let $?f = nth_default\ 0\ xs$

let $?bound = Max\ (insert\ 0\ (Suc\ ' \{n. ?f\ n \neq 0\}))$

show $?bound = length\ xs$

proof *(cases xs = [])*

case *False*

with $*$ **obtain** n **where** $n: n < length\ xs$ $xs\ !\ n \neq 0$

by *(fastforce simp add: no-trailing-unfold last-conv-nth neq-Nil-conv)*

then have $?bound = Max\ (Suc\ ' \{k. (k < length\ xs \longrightarrow xs\ !\ k \neq (0::'a)) \wedge k$

```

< length xs})
  by (subst Max-insert) (auto simp: nth-default-def)
  also let ?A = {k. k < length xs ∧ xs ! k ≠ 0}
  have {k. (k < length xs ⟶ xs ! k ≠ (0::'a)) ∧ k < length xs} = ?A by auto
  also have Max (Suc ' ?A) = Suc (Max ?A) using n
    by (subst mono-Max-commute [where f = Suc, symmetric]) (auto simp:
mono-Suc)
  also {
    have Max ?A ∈ ?A using n Max-in [of ?A] by fastforce
    hence Suc (Max ?A) ≤ length xs by simp
    moreover from * False have length xs - 1 ∈ ?A
      by (auto simp: no-trailing-unfold last-conv-nth)
    hence length xs - 1 ≤ Max ?A using Max-ge[of ?A length xs - 1] by auto
    hence length xs ≤ Suc (Max ?A) by simp
    ultimately have Suc (Max ?A) = length xs by simp }
  finally show ?thesis .
qed simp
qed

```

lemma *nth-trailing-zeros* [simp]:
 $\text{nth } (xs \text{ @ replicate } n \ 0) = \text{nth } xs$
 by (simp add: nth.abs-eq)

lemma *nth-idem*:
 $\text{nth } (\text{List.map } (\text{lookup } f) \ [0..<\text{degree } f]) = f$
 unfolding degree-def by transfer
 (auto simp: nth-default-def fun-eq-iff not-less)

lemma *nth-idem-bound*:
 assumes $\text{degree } f \leq n$
 shows $\text{nth } (\text{List.map } (\text{lookup } f) \ [0..<n]) = f$
proof –
 from assms obtain m where $n = \text{degree } f + m$
 by (blast dest: le-Suc-ex)
 then have $[0..<n] = [0..<\text{degree } f] \text{ @ } [\text{degree } f..<\text{degree } f + m]$
 by (simp add: upt-add-eq-append [of 0])
 moreover have $\text{List.map } (\text{lookup } f) \ [\text{degree } f..<\text{degree } f + m] = \text{replicate } m \ 0$
 by (rule replicate-eqI) (auto simp: beyond-degree-lookup-zero)
 ultimately show ?thesis by (simp add: nth-idem)
qed

78.13 Canonical sparse representation of $'a \Rightarrow_0 'b$

lift-definition *the-value* :: $('a \times 'b) \text{ list} \Rightarrow 'a \Rightarrow_0 'b::\text{zero}$
 is $\lambda xs \ k. \text{ case map-of } xs \ k \text{ of None} \Rightarrow 0 \mid \text{Some } v \Rightarrow v$

proof –
 fix $xs :: ('a \times 'b) \text{ list}$
 have $\text{fin: finite } \{k. \exists v. \text{map-of } xs \ k = \text{Some } v\}$
 using finite-dom-map-of [of xs] unfolding dom-def by auto

then show *finite* $\{k. (\text{case map-of } xs \text{ } k \text{ of } None \Rightarrow 0 \mid Some \ v \Rightarrow v) \neq 0\}$
using *fin* **by** (*simp* *split*: *option.split*)
qed

definition *items* :: (*'a*::*linorder* \Rightarrow_0 *'b*::*zero*) \Rightarrow (*'a* \times *'b*) *list*
where
items *f* = *List.map* ($\lambda k. (k, \text{lookup } f \ k)$) (*sorted-list-of-set* (*keys* *f*))

For the canonical sparse representation we provide both directions of morphisms since the specification of ordered association lists in theory *OAL-ist* will support arbitrary linear orders *linorder* as keys, not just natural numbers *nat*.

lemma *the-value-items* [*simp*]:
the-value (*items* *f*) = *f*
unfolding *items-def*
by *transfer* (*simp* *add*: *fun-eq-iff* *map-of-map-restrict* *restrict-map-def*)

lemma *lookup-the-value*:
 $\text{lookup } (\text{the-value } xs) \ k = (\text{case map-of } xs \ k \text{ of } None \Rightarrow 0 \mid Some \ v \Rightarrow v)$
by (*simp* *add*: *the-value.rep-eq*)

lemma *items-the-value*:
assumes *sorted* (*List.map* *fst* *xs*) **and** *distinct* (*List.map* *fst* *xs*) **and** $0 \notin \text{snd } ' \text{ set } xs$
shows *items* (*the-value* *xs*) = *xs*
proof –
from *assms* **have** *sorted-list-of-set* (*set* (*List.map* *fst* *xs*)) = *List.map* *fst* *xs*
unfolding *sorted-list-of-set-sort-remdups* **by** (*simp* *add*: *distinct-remdups-id* *sort-key-id-if-sorted*)
moreover from *assms* **have** *keys* (*the-value* *xs*) = *fst* ' *set* *xs*
by *transfer* (*auto* *simp*: *image-def* *split*: *option.split* *dest*: *set-map-of-compr*)
ultimately show *?thesis*
unfolding *items-def* **using** *assms*
by (*auto* *simp*: *lookup-the-value* *intro*: *map-idI*)
qed

lemma *the-value-Nil* [*simp*]:
the-value [] = 0
by *transfer* (*simp* *add*: *fun-eq-iff*)

lemma *the-value-Cons* [*simp*]:
the-value (*x* # *xs*) = *update* (*fst* *x*) (*snd* *x*) (*the-value* *xs*)
by *transfer* (*simp* *add*: *fun-eq-iff*)

lemma *items-zero* [*simp*]:
items 0 = []
unfolding *items-def* **by** *simp*

lemma *items-one* [*simp*]:

items 1 = [(0, 1)]
unfolding *items-def* **by** *transfer simp*

lemma *items-single* [*simp*]:
items (single *k v*) = (if *v* = 0 then [] else [(*k*, *v*)])
unfolding *items-def* **by** *simp*

lemma *in-set-items-iff* [*simp*]:
(*k*, *v*) ∈ *set* (*items f*) \longleftrightarrow *k* ∈ *keys f* ∧ *lookup f k* = *v*
unfolding *items-def* **by** *transfer auto*

78.14 Size estimation

context
fixes *f* :: 'a ⇒ nat
and *g* :: 'b :: zero ⇒ nat
begin

definition *poly-mapping-size* :: ('a ⇒₀ 'b) ⇒ nat
where
poly-mapping-size m = *g* 0 + (∑ *k* ∈ *keys m*. *Suc* (*f k* + *g* (*lookup m k*)))

lemma *poly-mapping-size-0* [*simp*]:
poly-mapping-size 0 = *g* 0
by (*simp add: poly-mapping-size-def*)

lemma *poly-mapping-size-single* [*simp*]:
poly-mapping-size (single *k v*) = (if *v* = 0 then *g* 0 else *g* 0 + *f k* + *g v* + 1)
unfolding *poly-mapping-size-def* **by** *transfer simp*

lemma *keys-less-poly-mapping-size*:
k ∈ *keys m* ⇒ *f k* + *g* (*lookup m k*) < *poly-mapping-size m*
unfolding *poly-mapping-size-def*
proof *transfer*

fix *k* :: 'a **and** *m* :: 'a ⇒ 'b **and** *f* :: 'a ⇒ nat **and** *g*
let ?*keys* = {*k*. *m k* ≠ 0}
assume *: *finite* ?*keys* *k* ∈ ?*keys*
then have *f k* + *g* (*m k*) = (∑ *k'* ∈ ?*keys*. *f k'* + *g* (*m k'*) when *k'* = *k*)
by (*simp add: sum.delta when-def*)
also have ... < (∑ *k'* ∈ ?*keys*. *Suc* (*f k'* + *g* (*m k'*))) **using** *
by (*intro sum-strict-mono*) (*auto simp: when-def*)
also have ... ≤ *g* 0 + ... **by** *simp*
finally have *f k* + *g* (*m k*) < ...
then show *f k* + *g* (*m k*) < *g* 0 + (∑ *k* | *m k* ≠ 0. *Suc* (*f k* + *g* (*m k*)))
by *simp*
qed

lemma *lookup-le-poly-mapping-size*:
g (*lookup m k*) ≤ *poly-mapping-size m*

```

proof (cases  $k \in \text{keys } m$ )
  case True
    with keys-less-poly-mapping-size [of  $k \ m$ ]
    show ?thesis by simp
  next
    case False
    then show ?thesis
      by (simp add: Poly-Mapping.poly-mapping-size-def in-keys-iff)
qed

lemma poly-mapping-size-estimation:
   $k \in \text{keys } m \implies y \leq f \ k + g \ (\text{lookup } m \ k) \implies y < \text{poly-mapping-size } m$ 
  using keys-less-poly-mapping-size by (auto intro: le-less-trans)

lemma poly-mapping-size-estimation2:
  assumes  $v \in \text{range } m$  and  $y \leq g \ v$ 
  shows  $y < \text{poly-mapping-size } m$ 
proof –
  from assms obtain  $k$  where  $*: \text{lookup } m \ k = v \ v \neq 0$ 
    by transfer blast
  then have  $k \in \text{keys } m$ 
    by (simp add: in-keys-iff)
  with  $*$  show ?thesis
    by (simp add: Poly-Mapping.poly-mapping-size-estimation assms(2) trans-le-add2)
qed

end

lemma poly-mapping-size-one [simp]:
   $\text{poly-mapping-size } f \ g \ 1 = g \ 0 + f \ 0 + g \ 1 + 1$ 
  unfolding poly-mapping-size-def by transfer simp

lemma poly-mapping-size-cong [fundef-cong]:
   $m = m' \implies g \ 0 = g' \ 0 \implies (\bigwedge k. k \in \text{keys } m' \implies f \ k = f' \ k)$ 
   $\implies (\bigwedge v. v \in \text{range } m' \implies g \ v = g' \ v)$ 
   $\implies \text{poly-mapping-size } f \ g \ m = \text{poly-mapping-size } f' \ g' \ m'$ 
  by (auto simp: poly-mapping-size-def intro!: sum.cong)

instantiation poly-mapping :: (type, zero) size
begin

definition size = poly-mapping-size ( $\lambda \cdot. 0$ ) ( $\lambda \cdot. 0$ )

instance ..

end

```

78.15 Further mapping operations and properties

It is like in algebra: there are many definitions, some are also used

lift-definition *mapp* ::

$(\text{'a} \Rightarrow \text{'b} :: \text{zero} \Rightarrow \text{'c} :: \text{zero}) \Rightarrow (\text{'a} \Rightarrow_0 \text{'b}) \Rightarrow (\text{'a} \Rightarrow_0 \text{'c})$
is $\lambda f p k. (\text{if } k \in \text{keys } p \text{ then } f k (\text{lookup } p k) \text{ else } 0)$
by *simp*

lemma *mapp-cong* [*fundef-cong*]:

$\llbracket m = m'; \bigwedge k. k \in \text{keys } m' \implies f k (\text{lookup } m' k) = f' k (\text{lookup } m' k) \rrbracket$
 $\implies \text{mapp } f m = \text{mapp } f' m'$
by *transfer (auto simp: fun-eq-iff)*

lemma *lookup-mapp*:

$\text{lookup } (\text{mapp } f p) k = (f k (\text{lookup } p k) \text{ when } k \in \text{keys } p)$
by (*simp add: mapp.rep-eq*)

lemma *keys-mapp-subset*: $\text{keys } (\text{mapp } f p) \subseteq \text{keys } p$

by (*meson in-keys-iff mapp.rep-eq subsetI*)

78.16 Free Abelian Groups Over a Type

abbreviation *frag-of* :: $\text{'a} \Rightarrow \text{'a} \Rightarrow_0 \text{int}$

where *frag-of* $c \equiv \text{Poly-Mapping.single } c (1::\text{int})$

lemma *lookup-frag-of* [*simp*]:

$\text{Poly-Mapping.lookup}(\text{frag-of } c) = (\lambda x. \text{if } x = c \text{ then } 1 \text{ else } 0)$
by (*force simp add: lookup-single-not-eq*)

lemma *frag-of-nonzero* [*simp*]: $\text{frag-of } a \neq 0$

by (*metis lookup-single-eq lookup-zero zero-neq-one*)

definition *frag-cmul* :: $\text{int} \Rightarrow (\text{'a} \Rightarrow_0 \text{int}) \Rightarrow (\text{'a} \Rightarrow_0 \text{int})$

where *frag-cmul* $c a = \text{Abs-poly-mapping } (\lambda x. c * \text{Poly-Mapping.lookup } a x)$

lemma *frag-cmul-zero* [*simp*]: $\text{frag-cmul } 0 x = 0$

by (*simp add: frag-cmul-def*)

lemma *frag-cmul-zero2* [*simp*]: $\text{frag-cmul } c 0 = 0$

by (*simp add: frag-cmul-def*)

lemma *frag-cmul-one* [*simp*]: $\text{frag-cmul } 1 x = x$

by (*simp add: frag-cmul-def*)

lemma *frag-cmul-minus-one* [*simp*]: $\text{frag-cmul } (-1) x = -x$

by (*simp add: frag-cmul-def uminus-poly-mapping-def poly-mapping-eqI*)

lemma *frag-cmul-cmul* [*simp*]: $\text{frag-cmul } c (\text{frag-cmul } d x) = \text{frag-cmul } (c*d) x$

by (*simp add: frag-cmul-def mult-ac*)

lemma *lookup-frag-cmul* [simp]: $\text{poly-mapping.lookup}(\text{frag-cmul } c \ x) \ i = c * \text{poly-mapping.lookup } x \ i$
by (*simp add: frag-cmul-def*)

lemma *minus-frag-cmul* [simp]: $-\text{frag-cmul } k \ x = \text{frag-cmul } (-k) \ x$
by (*simp add: poly-mapping-eqI*)

lemma *keys-frag-of*: $\text{Poly-Mapping.keys}(\text{frag-of } a) = \{a\}$
by *simp*

lemma *finite-cmul-nonzero*: $\text{finite } \{x. \ c * \text{Poly-Mapping.lookup } a \ x \neq (0::\text{int})\}$
by *simp*

lemma *keys-cmul*: $\text{Poly-Mapping.keys}(\text{frag-cmul } c \ a) \subseteq \text{Poly-Mapping.keys } a$
using *finite-cmul-nonzero* [of *c a*]
by (*metis lookup-frag-cmul mult-zero-right not-in-keys-iff-lookup-eq-zero subsetI*)

lemma *keys-cmul-iff* [iff]: $i \in \text{Poly-Mapping.keys}(\text{frag-cmul } c \ x) \longleftrightarrow i \in \text{Poly-Mapping.keys } x \wedge c \neq 0$
by (*metis in-keys-iff lookup-frag-cmul mult-eq-0-iff*)

lemma *keys-minus* [simp]: $\text{Poly-Mapping.keys}(-a) = \text{Poly-Mapping.keys } a$
by (*metis (no-types, opaque-lifting) in-keys-iff lookup-uminus neg-equal-0-iff-equal subsetI subset-antisym*)

lemma *keys-diff*:
 $\text{Poly-Mapping.keys}(a - b) \subseteq \text{Poly-Mapping.keys } a \cup \text{Poly-Mapping.keys } b$
by (*auto simp: in-keys-iff lookup-minus*)

lemma *keys-eq-empty* [simp]: $\text{Poly-Mapping.keys } c = \{\} \longleftrightarrow c = 0$
by (*metis in-keys-iff keys-zero lookup-zero poly-mapping-eqI*)

lemma *frag-cmul-eq-0-iff* [simp]: $\text{frag-cmul } k \ c = 0 \longleftrightarrow k=0 \vee c=0$
by *auto* (*metis subsetI subset-antisym keys-cmul-iff keys-eq-empty*)

lemma *frag-of-eq*: $\text{frag-of } x = \text{frag-of } y \longleftrightarrow x = y$
by (*metis lookup-single-eq lookup-single-not-eq zero-neq-one*)

lemma *frag-cmul-distrib*: $\text{frag-cmul } (c+d) \ a = \text{frag-cmul } c \ a + \text{frag-cmul } d \ a$
by (*simp add: frag-cmul-def plus-poly-mapping-def int-distrib*)

lemma *frag-cmul-distrib2*: $\text{frag-cmul } c \ (a+b) = \text{frag-cmul } c \ a + \text{frag-cmul } c \ b$
by (*simp add: int-distrib(2) lookup-add poly-mapping-eqI*)

lemma *frag-cmul-diff-distrib*: $\text{frag-cmul } (a - b) \ c = \text{frag-cmul } a \ c - \text{frag-cmul } b \ c$
by (*auto simp: left-diff-distrib lookup-minus poly-mapping-eqI*)

lemma *frag-cmul-sum*:

$$\text{frag-cmul } a \text{ (sum } b \text{ } I) = (\sum_{i \in I}. \text{frag-cmul } a \text{ (} b \text{ } i))$$

proof (*induction rule: infinite-finite-induct*)

case (*insert i I*)

then show *?case*

by (*auto simp: algebra-simps frag-cmul-distrib2*)

qed *auto*

lemma *keys-sum*: $\text{Poly-Mapping.keys}(\text{sum } b \text{ } I) \subseteq (\bigcup i \in I. \text{Poly-Mapping.keys}(b \text{ } i))$

proof (*induction I rule: infinite-finite-induct*)

case (*insert i I*)

then show *?case*

using *keys-add [of b i sum b I]* **by** *auto*

qed *auto*

definition *frag-extend* :: $('b \Rightarrow 'a \Rightarrow_0 \text{int}) \Rightarrow ('b \Rightarrow_0 \text{int}) \Rightarrow 'a \Rightarrow_0 \text{int}$

where *frag-extend* $b \text{ } x \equiv (\sum_{i \in \text{Poly-Mapping.keys } x}. \text{frag-cmul } (\text{Poly-Mapping.lookup } x \text{ } i) \text{ (} b \text{ } i))$

lemma *frag-extend-0 [simp]*: *frag-extend* $b \text{ } 0 = 0$

by (*simp add: frag-extend-def*)

lemma *frag-extend-of [simp]*: *frag-extend* $f \text{ (frag-of } a) = f \text{ } a$

by (*simp add: frag-extend-def*)

lemma *frag-extend-cmul*:

$$\text{frag-extend } f \text{ (frag-cmul } c \text{ } x) = \text{frag-cmul } c \text{ (frag-extend } f \text{ } x)$$

by (*auto simp: frag-extend-def frag-cmul-sum intro: sum.mono-neutral-cong-left*)

lemma *frag-extend-minus*:

$$\text{frag-extend } f \text{ (} - \text{ } x) = - \text{ (frag-extend } f \text{ } x)$$

using *frag-extend-cmul [of f -1]* **by** *simp*

lemma *frag-extend-add*:

$$\text{frag-extend } f \text{ (} a + b) = (\text{frag-extend } f \text{ } a) + (\text{frag-extend } f \text{ } b)$$

proof –

have $*$: $(\sum_{i \in \text{Poly-Mapping.keys } a}. \text{frag-cmul } (\text{poly-mapping.lookup } a \text{ } i) \text{ (} f \text{ } i))$

$$= (\sum_{i \in \text{Poly-Mapping.keys } a \cup \text{Poly-Mapping.keys } b}. \text{frag-cmul } (\text{poly-mapping.lookup } a \text{ } i) \text{ (} f \text{ } i))$$

$$= (\sum_{i \in \text{Poly-Mapping.keys } b}. \text{frag-cmul } (\text{poly-mapping.lookup } b \text{ } i) \text{ (} f \text{ } i))$$

$$= (\sum_{i \in \text{Poly-Mapping.keys } a \cup \text{Poly-Mapping.keys } b}. \text{frag-cmul } (\text{poly-mapping.lookup } b \text{ } i) \text{ (} f \text{ } i))$$

by (*auto simp: in-keys-iff intro: sum.mono-neutral-cong-left*)

have *frag-extend* $f \text{ (} a + b) = (\sum_{i \in \text{Poly-Mapping.keys } (a + b)}. \text{frag-cmul } (\text{poly-mapping.lookup } a \text{ } i) \text{ (} f \text{ } i))$

$$+ \text{frag-cmul } (\text{poly-mapping.lookup } b \text{ } i) \text{ (} f \text{ } i))$$

by (*auto simp: frag-extend-def Poly-Mapping.lookup-add frag-cmul-distrib*)

also have ... = $(\sum i \in \text{Poly-Mapping.keys } a \cup \text{Poly-Mapping.keys } b. \text{frag-cmul} (\text{poly-mapping.lookup } a \ i) (f \ i))$
 $+ \text{frag-cmul} (\text{poly-mapping.lookup } b \ i) (f \ i))$
proof (rule *sum.mono-neutral-cong-left*)
show $\forall i \in \text{keys } a \cup \text{keys } b. \text{keys } (a + b).$
 $\text{frag-cmul} (\text{lookup } a \ i) (f \ i) + \text{frag-cmul} (\text{lookup } b \ i) (f \ i) = 0$
by (*metis DiffD2 frag-cmul-distrib frag-cmul-zero in-keys-iff lookup-add*)
qed (*auto simp: keys-add*)
also have ... = $(\text{frag-extend } f \ a) + (\text{frag-extend } f \ b)$
by (*auto simp: * sum.distrib frag-extend-def*)
finally show ?thesis .
qed

lemma *frag-extend-diff*:
 $\text{frag-extend } f \ (a - b) = (\text{frag-extend } f \ a) - (\text{frag-extend } f \ b)$
by (*metis (no-types, opaque-lifting) add-uminus-conv-diff frag-extend-add frag-extend-minus*)

lemma *frag-extend-sum*:
 $\text{finite } I \implies \text{frag-extend } f \ (\sum i \in I. g \ i) = \text{sum } (\text{frag-extend } f \ o \ g) \ I$
by (*induction I rule: finite-induct*) (*simp-all add: frag-extend-add*)

lemma *frag-extend-eq*:
 $(\bigwedge f. f \in \text{Poly-Mapping.keys } c \implies g \ f = h \ f) \implies \text{frag-extend } g \ c = \text{frag-extend } h \ c$
by (*simp add: frag-extend-def*)

lemma *frag-extend-eq-0*:
 $(\bigwedge x. x \in \text{Poly-Mapping.keys } c \implies f \ x = 0) \implies \text{frag-extend } f \ c = 0$
by (*simp add: frag-extend-def*)

lemma *keys-frag-extend*: $\text{Poly-Mapping.keys}(\text{frag-extend } f \ c) \subseteq (\bigcup x \in \text{Poly-Mapping.keys } c. \text{Poly-Mapping.keys}(f \ x))$
unfolding *frag-extend-def*
using *keys-sum* **by** *fastforce*

lemma *frag-expansion*: $a = \text{frag-extend } \text{frag-of } a$
proof –
have *: *finite I*
 $\implies \text{Poly-Mapping.lookup } (\sum i \in I. \text{frag-cmul} (\text{Poly-Mapping.lookup } a \ i) (\text{frag-of } i)) \ j =$
 $(\text{if } j \in I \text{ then } \text{Poly-Mapping.lookup } a \ j \text{ else } 0) \text{ for } I \ j$
by (*induction I rule: finite-induct*) (*auto simp: lookup-single lookup-add*)
show ?thesis
unfolding *frag-extend-def*
by (*rule poly-mapping-eqI*) (*fastforce simp add: in-keys-iff **)
qed

lemma *frag-closure-minus-cmul*:
assumes $P \ 0$ **and** $P: \bigwedge x \ y. \llbracket P \ x; P \ y \rrbracket \implies P(x - y) \ P \ c$

```

  shows  $P(\text{frag-cmul } k \ c)$ 
proof -
  have  $P(\text{frag-cmul } (\text{int } n) \ c)$  for  $n$ 
  proof (induction  $n$ )
    case 0
    then show ?case
    by (simp add: assms)
  next
    case (Suc  $n$ )
    then show ?case
    by (metis assms diff-0 diff-minus-eq-add frag-cmul-distrib frag-cmul-one
of-nat-Suc)
  qed
  then show ?thesis
  by (metis (no-types, opaque-lifting) add-diff-eq assms(2) diff-add-cancel frag-cmul-distrib
int-diff-cases)
qed

```

lemma *frag-induction* [consumes 1, case-names zero one diff]:

```

  assumes supp:  $\text{Poly-Mapping.keys } c \subseteq S$ 
  and 0:  $P \ 0$  and sing:  $\bigwedge x. x \in S \implies P(\text{frag-of } x)$ 
  and diff:  $\bigwedge a \ b. \llbracket P \ a; \ P \ b \rrbracket \implies P(a - b)$ 
  shows  $P \ c$ 
proof -
  have  $P(\sum_{i \in I}. \text{frag-cmul } (\text{poly-mapping.lookup } c \ i) \ (\text{frag-of } i))$ 
  if  $I \subseteq \text{Poly-Mapping.keys } c$  for  $I$ 
  using finite-subset [OF that finite-keys [of  $c$ ]] that supp
  proof (induction  $I$  arbitrary:  $c$  rule: finite-induct)
    case empty
    then show ?case
    by (auto simp: 0)
  next
    case (insert  $i \ I \ c$ )
    have  $ab: a + b = a - (0 - b)$  for  $a \ b :: 'a \Rightarrow_0 \ \text{int}$ 
    by simp
    have  $P_{\text{frag}}: P(\text{frag-cmul } (\text{poly-mapping.lookup } c \ i) \ (\text{frag-of } i))$ 
    by (metis 0 diff frag-closure-minus-cmul insert.premis insert-subset sing subset-iff)
    with insert show ?case
    by (metis (mono-tags, lifting) 0 ab diff insert-subset sum.insert)
  qed
  then show ?thesis
  by (subst frag-expansion) (auto simp: frag-extend-def)
qed

```

lemma *frag-extend-compose*:

```

  frag-extend  $f$  (frag-extend (frag-of  $o \ g$ )  $c$ ) = frag-extend ( $f \ o \ g$ )  $c$ 
  using subset-UNIV
  by (induction  $c$  rule: frag-induction) (auto simp: frag-extend-diff)

```

```

lemma frag-split:
  fixes  $c :: 'a \Rightarrow_0 \text{int}$ 
  assumes  $\text{Poly-Mapping.keys } c \subseteq S \cup T$ 
  obtains  $d \ e$  where  $\text{Poly-Mapping.keys } d \subseteq S \ \text{Poly-Mapping.keys } e \subseteq T \ d + e = c$ 
proof
  let  $?d = \text{frag-extend } (\lambda f. \text{if } f \in S \text{ then frag-of } f \text{ else } 0) \ c$ 
  let  $?e = \text{frag-extend } (\lambda f. \text{if } f \in S \text{ then } 0 \text{ else frag-of } f) \ c$ 
  show  $\text{Poly-Mapping.keys } ?d \subseteq S \ \text{Poly-Mapping.keys } ?e \subseteq T$ 
  using assms by (auto intro!; order-trans [OF keys-frag-extend] split; if-split-asm)
  show  $?d + ?e = c$ 
  using assms
  proof (induction c rule: frag-induction)
  case (diff a b)
  then show  $?case$ 
  by (metis (no-types, lifting) frag-extend-diff add-diff-eq diff-add-eq diff-add-eq-diff-diff-swap)
  qed auto
qed

hide-const (open) lookup single update keys range map map-key degree nth the-value items foldr mapp

end

```

79 Exponentiation by Squaring

```

theory Power-By-Squaring
  imports Main
begin

context
  fixes  $f :: 'a \Rightarrow 'a \Rightarrow 'a$ 
begin

function efficient-funpow  $:: 'a \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a$  where
  efficient-funpow  $y \ x \ 0 = y$ 
  | efficient-funpow  $y \ x \ (\text{Suc } 0) = f \ x \ y$ 
  |  $n \neq 0 \implies \text{even } n \implies \text{efficient-funpow } y \ x \ n = \text{efficient-funpow } y \ (f \ x \ x) \ (n \text{ div } 2)$ 
  |  $n \neq 1 \implies \text{odd } n \implies \text{efficient-funpow } y \ x \ n = \text{efficient-funpow } (f \ x \ y) \ (f \ x \ x) \ (n \text{ div } 2)$ 
  by force+
termination by (relation measure (snd  $\circ$  snd)) (auto elim: oddE)

lemma efficient-funpow-code [code]:
  efficient-funpow  $y \ x \ n =$ 
    (if  $n = 0$  then  $y$ 
     else if  $n = 1$  then  $f \ x \ y$ 

```



```

      else if even n then efficient-funpow y (f x x) (n div 2)
      else efficient-funpow (f x y) (f x x) (n div 2))
    by (induction y x n rule: efficient-funpow.induct) auto

end

lemma efficient-funpow-correct:
  assumes f-assoc:  $\bigwedge x z. f\ x\ (f\ x\ z) = f\ (f\ x\ x)\ z$ 
  shows efficient-funpow f y x n = (f x  $\frown$  n) y
proof -
  have [simp]:  $f\ \frown\ 2 = (\lambda x. f\ (f\ x))$  for f :: 'a  $\Rightarrow$  'a
  by (simp add: eval-nat-numeral o-def)
  show ?thesis
  by (induction y x n rule: efficient-funpow.induct[of f])
    (auto elim!: evenE oddE simp: funpow-mult [symmetric] funpow-Suc-right
      f-assoc
      simp del: funpow.simps(2))
qed

context monoid-mult
begin

lemma power-by-squaring: efficient-funpow (*) (1 :: 'a) = ( $\frown$ )
proof (intro ext)
  fix x :: 'a and n
  have efficient-funpow (*) 1 x n = ((*) x  $\frown$  n) 1
  by (subst efficient-funpow-correct) (simp-all add: mult.assoc)
  also have ... = x  $\frown$  n
  by (induction n) simp-all
  finally show efficient-funpow (*) 1 x n = x  $\frown$  n .
qed

end

end

```

80 Preorders with explicit equivalence relation

```

theory Preorder
imports Main
begin

class preorder-equiv = preorder
begin

definition equiv :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  where equiv x y  $\longleftrightarrow$  x  $\leq$  y  $\wedge$  y  $\leq$  x

```

notation

equiv $\langle \langle (\approx') \rangle \rangle$ **and**

equiv $\langle \langle \langle \text{notation} = \langle \text{infix } \approx \rangle \rangle - / \approx - \rangle \rangle$ [51, 51] 50)

lemma *equivD1*: $x \leq y$ **if** $x \approx y$
using *that* **by** (*simp add: equiv-def*)

lemma *equivD2*: $y \leq x$ **if** $x \approx y$
using *that* **by** (*simp add: equiv-def*)

lemma *equiv-refl* [*iff*]: $x \approx x$
by (*simp add: equiv-def*)

lemma *equiv-sym*: $x \approx y \longleftrightarrow y \approx x$
by (*auto simp add: equiv-def*)

lemma *equiv-trans*: $x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$
by (*auto simp: equiv-def intro: order-trans*)

lemma *equiv-antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
by (*simp only: equiv-def*)

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge \neg x \approx y$
by (*auto simp add: equiv-def less-le-not-le*)

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x \approx y$
by (*auto simp add: equiv-def less-le*)

lemma *le-imp-less-or-equiv*: $x \leq y \Longrightarrow x < y \vee x \approx y$
by (*simp add: less-le*)

lemma *less-imp-not-equiv*: $x < y \Longrightarrow \neg x \approx y$
by (*simp add: less-le*)

lemma *not-equiv-le-trans*: $\neg a \approx b \Longrightarrow a \leq b \Longrightarrow a < b$
by (*simp add: less-le*)

lemma *le-not-equiv-trans*: $a \leq b \Longrightarrow \neg a \approx b \Longrightarrow a < b$
by (*rule not-equiv-le-trans*)

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x \approx y$
by (*simp add: equiv-def*)

end

ML-file $\langle \sim \sim / \text{src} / \text{Provers} / \text{preorder.ML} \rangle$

ML \langle

structure *Quasi* = *Quasi-Tac*(

struct

```

val le-trans = @{thm order-trans};
val le-refl = @{thm order-refl};
val eqD1 = @{thm equivD1};
val eqD2 = @{thm equivD2};
val less-reflE = @{thm less-irrefl};
val less-imp-le = @{thm less-imp-le};
val le-neq-trans = @{thm le-not-equiv-trans};
val neq-le-trans = @{thm not-equiv-le-trans};
val less-imp-neq = @{thm less-imp-not-equiv};

fun decomp-quasi thy (Const (@{const-name less-eq}, -) $ t1 $ t2) = SOME (t1,
  <=, t2)
  | decomp-quasi thy (Const (@{const-name less}, -) $ t1 $ t2) = SOME (t1, <,
  t2)
  | decomp-quasi thy (Const (@{const-name equiv}, -) $ t1 $ t2) = SOME (t1, =,
  t2)
  | decomp-quasi thy (Const (@{const-name Not}, -) $ (Const (@{const-name
  equiv}, -) $ t1 $ t2)) = SOME (t1, ~=, t2)
  | decomp-quasi thy - = NONE;

fun decomp-trans thy t = case decomp-quasi thy t of
  x as SOME (t1, <=, t2) => x
  | - => NONE;

end
);
›

end

```

81 Additive group operations on product types

```

theory Product-Plus
imports Main
begin

```

81.1 Operations

```

instantiation prod :: (zero, zero) zero
begin

```

```

definition zero-prod-def: 0 = (0, 0)

```

```

instance ..
end

```

```

instantiation prod :: (plus, plus) plus

```

begin

definition *plus-prod-def*:

$$x + y = (fst\ x + fst\ y, snd\ x + snd\ y)$$

instance ..

end

instantiation *prod* :: (*minus*, *minus*) *minus*

begin

definition *minus-prod-def*:

$$x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$$

instance ..

end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*

begin

definition *uminus-prod-def*:

$$- x = (-\ fst\ x, -\ snd\ x)$$

instance ..

end

lemma *fst-zero* [*simp*]: *fst* 0 = 0

unfolding *zero-prod-def* **by** *simp*

lemma *snd-zero* [*simp*]: *snd* 0 = 0

unfolding *zero-prod-def* **by** *simp*

lemma *fst-add* [*simp*]: *fst* (*x* + *y*) = *fst* *x* + *fst* *y*

unfolding *plus-prod-def* **by** *simp*

lemma *snd-add* [*simp*]: *snd* (*x* + *y*) = *snd* *x* + *snd* *y*

unfolding *plus-prod-def* **by** *simp*

lemma *fst-diff* [*simp*]: *fst* (*x* - *y*) = *fst* *x* - *fst* *y*

unfolding *minus-prod-def* **by** *simp*

lemma *snd-diff* [*simp*]: *snd* (*x* - *y*) = *snd* *x* - *snd* *y*

unfolding *minus-prod-def* **by** *simp*

lemma *fst-uminus* [*simp*]: *fst* (- *x*) = - *fst* *x*

unfolding *uminus-prod-def* **by** *simp*

lemma *snd-uminus* [*simp*]: *snd* (- *x*) = - *snd* *x*

unfolding *uminus-prod-def* **by** *simp*

lemma *add-Pair* [*simp*]: $(a, b) + (c, d) = (a + c, b + d)$
unfolding *plus-prod-def* **by** *simp*

lemma *diff-Pair* [*simp*]: $(a, b) - (c, d) = (a - c, b - d)$
unfolding *minus-prod-def* **by** *simp*

lemma *uminus-Pair* [*simp*, *code*]: $-(a, b) = (-a, -b)$
unfolding *uminus-prod-def* **by** *simp*

81.2 Class instances

instance *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
by *standard* (*simp* *add*: *prod-eq-iff* *add.assoc*)

instance *prod* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
by *standard* (*simp* *add*: *prod-eq-iff* *add.commute*)

instance *prod* :: (*monoid-add*, *monoid-add*) *monoid-add*
by *standard* (*simp-all* *add*: *prod-eq-iff*)

instance *prod* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
by *standard* (*simp* *add*: *prod-eq-iff*)

instance *prod* :: (*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*
by *standard* (*simp-all* *add*: *prod-eq-iff*)

instance *prod* :: (*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
by *standard* (*simp-all* *add*: *prod-eq-iff* *diff-diff-eq*)

instance *prod* :: (*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
..

instance *prod* :: (*group-add*, *group-add*) *group-add*
by *standard* (*simp-all* *add*: *prod-eq-iff*)

instance *prod* :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
by *standard* (*simp-all* *add*: *prod-eq-iff*)

lemma *fst-sum*: $\text{fst} (\sum_{x \in A}. f\ x) = (\sum_{x \in A}. \text{fst} (f\ x))$

proof (*cases* *finite* *A*)

case *True*

then show *?thesis* **by** *induct* *simp-all*

next

case *False*

then show *?thesis* **by** *simp*

qed

lemma *snd-sum*: $\text{snd} (\sum_{x \in A}. f\ x) = (\sum_{x \in A}. \text{snd} (f\ x))$

```

proof (cases finite A)
  case True
    then show ?thesis by induct simp-all
next
  case False
    then show ?thesis by simp
qed

lemma sum-prod:  $(\sum x \in A. (f\ x, g\ x)) = (\sum x \in A. f\ x, \sum x \in A. g\ x)$ 
proof (cases finite A)
  case True
    then show ?thesis by induct (simp-all add: zero-prod-def)
next
  case False
    then show ?thesis by (simp add: zero-prod-def)
qed

end

```

82 Roots of real quadratics

```

theory Quadratic-Discriminant
imports Complex-Main
begin

```

```

definition discrim :: real  $\Rightarrow$  real  $\Rightarrow$  real  $\Rightarrow$  real
  where discrim a b c  $\equiv b^2 - 4 * a * c$ 

```

```

lemma complete-square:
   $a \neq 0 \implies a * x^2 + b * x + c = 0 \longleftrightarrow (2 * a * x + b)^2 = \text{discrim } a\ b\ c$ 
by (simp add: discrim-def) algebra

```

```

lemma discriminant-negative:
  fixes a b c x :: real
  assumes a  $\neq 0$ 
    and discrim a b c < 0
  shows a * x2 + b * x + c  $\neq 0$ 
proof –
  have (2 * a * x + b)2  $\geq 0$ 
    by simp
  with <discrim a b c < 0> have (2 * a * x + b)2  $\neq$  discrim a b c
    by arith
  with complete-square and <a  $\neq 0$ > show a * x2 + b * x + c  $\neq 0$ 
    by simp
qed

```

```

lemma plus-or-minus-sqrt:
  fixes x y :: real
  assumes y  $\geq 0$ 

```

shows $x^2 = y \longleftrightarrow x = \text{sqrt } y \vee x = - \text{sqrt } y$

proof

assume $x^2 = y$

then have $\text{sqrt } (x^2) = \text{sqrt } y$

by *simp*

then have $\text{sqrt } y = |x|$

by *simp*

then show $x = \text{sqrt } y \vee x = - \text{sqrt } y$

by *auto*

next

assume $x = \text{sqrt } y \vee x = - \text{sqrt } y$

then have $x^2 = (\text{sqrt } y)^2 \vee x^2 = (- \text{sqrt } y)^2$

by *auto*

with $\langle y \geq 0 \rangle$ show $x^2 = y$

by *simp*

qed

lemma *divide-non-zero*:

fixes $x \ y \ z :: \text{real}$

assumes $x \neq 0$

shows $x * y = z \longleftrightarrow y = z / x$

proof

show $y = z / x$ if $x * y = z$

using $\langle x \neq 0 \rangle$ that by (*simp add: field-simps*)

show $x * y = z$ if $y = z / x$

using $\langle x \neq 0 \rangle$ that by *simp*

qed

lemma *discriminant-nonneg*:

fixes $a \ b \ c \ x :: \text{real}$

assumes $a \neq 0$

and $\text{discrim } a \ b \ c \geq 0$

shows $a * x^2 + b * x + c = 0 \longleftrightarrow$

$x = (-b + \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a) \vee$

$x = (-b - \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a)$

proof –

from *complete-square* and *plus-or-minus-sqrt* and *assms*

have $a * x^2 + b * x + c = 0 \longleftrightarrow$

$(2 * a) * x + b = \text{sqrt } (\text{discrim } a \ b \ c) \vee$

$(2 * a) * x + b = - \text{sqrt } (\text{discrim } a \ b \ c)$

by *simp*

also have $\dots \longleftrightarrow (2 * a) * x = (-b + \text{sqrt } (\text{discrim } a \ b \ c)) \vee$

$(2 * a) * x = (-b - \text{sqrt } (\text{discrim } a \ b \ c))$

by *auto*

also from $\langle a \neq 0 \rangle$ and *divide-non-zero* [of $2 * a \ x$]

have $\dots \longleftrightarrow x = (-b + \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a) \vee$

$x = (-b - \text{sqrt } (\text{discrim } a \ b \ c)) / (2 * a)$

by *simp*

finally show $a * x^2 + b * x + c = 0 \longleftrightarrow$

$$x = (-b + \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a) \vee$$

$$x = (-b - \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a) .$$

qed

lemma *discriminant-zero*:fixes $a \ b \ c \ x :: \text{real}$ assumes $a \neq 0$ and $\text{discrim } a \ b \ c = 0$ shows $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b / (2 * a)$ by (*simp add: discriminant-nonneg assms*)theorem *discriminant-iff*:fixes $a \ b \ c \ x :: \text{real}$ assumes $a \neq 0$ shows $a * x^2 + b * x + c = 0 \longleftrightarrow$ $\text{discrim } a \ b \ c \geq 0 \wedge$ $(x = (-b + \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a) \vee$ $x = (-b - \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a))$

proof

assume $a * x^2 + b * x + c = 0$ with *discriminant-negative* and $\langle a \neq 0 \rangle$ have $\neg(\text{discrim } a \ b \ c < 0)$ by *auto*then have $\text{discrim } a \ b \ c \geq 0$ by *simp*with *discriminant-nonneg* and $\langle a * x^2 + b * x + c = 0 \rangle$ and $\langle a \neq 0 \rangle$ have $x = (-b + \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a) \vee$ $x = (-b - \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a)$ by *simp*with $\langle \text{discrim } a \ b \ c \geq 0 \rangle$ show $\text{discrim } a \ b \ c \geq 0 \wedge$ $(x = (-b + \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a) \vee$ $x = (-b - \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a)) ..$

next

assume $\text{discrim } a \ b \ c \geq 0 \wedge$ $(x = (-b + \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a) \vee$ $x = (-b - \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a))$ then have $\text{discrim } a \ b \ c \geq 0$ and $x = (-b + \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a) \vee$ $x = (-b - \text{sqrt}(\text{discrim } a \ b \ c)) / (2 * a)$ by *simp-all*with *discriminant-nonneg* and $\langle a \neq 0 \rangle$ show $a * x^2 + b * x + c = 0$ by *simp*

qed

lemma *discriminant-nonneg-ex*:fixes $a \ b \ c :: \text{real}$ assumes $a \neq 0$ and $\text{discrim } a \ b \ c \geq 0$ shows $\exists x. a * x^2 + b * x + c = 0$

by (auto simp: discriminant-nonneg assms)

lemma discriminant-pos-ex:

fixes $a\ b\ c :: \text{real}$
 assumes $a \neq 0$
 and $\text{discrim } a\ b\ c > 0$
 shows $\exists x\ y. x \neq y \wedge a * x^2 + b * x + c = 0 \wedge a * y^2 + b * y + c = 0$
 proof -
 let $?x = (-b + \text{sqrt } (\text{discrim } a\ b\ c)) / (2 * a)$
 let $?y = (-b - \text{sqrt } (\text{discrim } a\ b\ c)) / (2 * a)$
 from $\langle \text{discrim } a\ b\ c > 0 \rangle$ have $\text{sqrt } (\text{discrim } a\ b\ c) \neq 0$
 by simp
 then have $\text{sqrt } (\text{discrim } a\ b\ c) \neq -\text{sqrt } (\text{discrim } a\ b\ c)$
 by arith
 with $\langle a \neq 0 \rangle$ have $?x \neq ?y$
 by simp
 moreover from assms have $a * ?x^2 + b * ?x + c = 0$ and $a * ?y^2 + b * ?y$
 $+ c = 0$
 using discriminant-nonneg [of $a\ b\ c\ ?x$]
 and discriminant-nonneg [of $a\ b\ c\ ?y$]
 by simp-all
 ultimately show ?thesis
 by blast
 qed

lemma discriminant-pos-distinct:

fixes $a\ b\ c\ x :: \text{real}$
 assumes $a \neq 0$
 and $\text{discrim } a\ b\ c > 0$
 shows $\exists y. x \neq y \wedge a * y^2 + b * y + c = 0$
 proof -
 from discriminant-pos-ex and $\langle a \neq 0 \rangle$ and $\langle \text{discrim } a\ b\ c > 0 \rangle$
 obtain w and z where $w \neq z$
 and $a * w^2 + b * w + c = 0$ and $a * z^2 + b * z + c = 0$
 by blast
 show $\exists y. x \neq y \wedge a * y^2 + b * y + c = 0$
 proof (cases $x = w$)
 case True
 with $\langle w \neq z \rangle$ have $x \neq z$
 by simp
 with $\langle a * z^2 + b * z + c = 0 \rangle$ show ?thesis
 by auto
 next
 case False
 with $\langle a * w^2 + b * w + c = 0 \rangle$ show ?thesis
 by auto
 qed
 qed

```

lemma Rats-solution-QE:
  assumes  $a \in \mathbb{Q} \ b \in \mathbb{Q} \ a \neq 0$ 
  and  $a*x^2 + b*x + c = 0$ 
  and  $\text{sqrt}(\text{discrim } a \ b \ c) \in \mathbb{Q}$ 
  shows  $x \in \mathbb{Q}$ 
using assms(1,2,5) discriminant-iff[THEN iffD1, OF assms(3,4)] by auto

lemma Rats-solution-QE-converse:
  assumes  $a \in \mathbb{Q} \ b \in \mathbb{Q}$ 
  and  $a*x^2 + b*x + c = 0$ 
  and  $x \in \mathbb{Q}$ 
  shows  $\text{sqrt}(\text{discrim } a \ b \ c) \in \mathbb{Q}$ 
proof –
  from assms(3) have  $\text{discrim } a \ b \ c = (2*a*x+b)^2$  unfolding discrim-def by
algebra
  hence  $\text{sqrt}(\text{discrim } a \ b \ c) = |2*a*x+b|$  by (simp)
  thus ?thesis using  $\langle a \in \mathbb{Q} \rangle \langle b \in \mathbb{Q} \rangle \langle x \in \mathbb{Q} \rangle$  by (simp)
qed

end

```

83 Pretty syntax for Quotient operations

```

theory Quotient-Syntax
imports Main
begin

notation
  rel-conj (infixr  $\langle \text{OOO} \rangle$  75) and
  map-fun (infixr  $\langle \text{---} \rangle$  55) and
  rel-fun (infixr  $\langle \text{===} \rangle$  55)

end

```

84 Quotient infrastructure for the set type

```

theory Quotient-Set
imports Quotient-Syntax
begin

```

84.1 Contravariant set map (vimage) and set relator, rules for the Quotient package

definition $\text{rel-vset } R \ xs \ ys \equiv \forall x \ y. \ R \ x \ y \longrightarrow x \in xs \longleftrightarrow y \in ys$

```

lemma rel-vset-eq [id-simps]:
   $\text{rel-vset } (=) = (=)$ 
  by (subst fun-eq-iff, subst fun-eq-iff) (simp add: set-eq-iff rel-vset-def)

```

```

lemma rel-vset-equivp:
  assumes e: equivp R
  shows rel-vset R xs ys  $\longleftrightarrow$  xs = ys  $\wedge$  ( $\forall x\ y. x \in xs \longrightarrow R\ x\ y \longrightarrow y \in ys$ )
  unfolding rel-vset-def
  using equivp-reflp[OF e]
  by auto (metis, metis equivp-symp[OF e])

lemma set-quotient [quot-thm]:
  assumes Quotient3 R Abs Rep
  shows Quotient3 (rel-vset R) (vimage Rep) (vimage Abs)
proof (rule Quotient3I)
  from assms have  $\bigwedge x. Abs\ (Rep\ x) = x$  by (rule Quotient3-abs-rep)
  then show  $\bigwedge xs. Rep\ -' (Abs\ -' xs) = xs$ 
    unfolding vimage-def by auto
next
  show  $\bigwedge xs. rel-vset\ R\ (Abs\ -' xs)\ (Abs\ -' xs)$ 
    unfolding rel-vset-def vimage-def
    by auto (metis Quotient3-rel-abs[OF assms])+
next
  fix r s
  show rel-vset R r s = (rel-vset R r r  $\wedge$  rel-vset R s s  $\wedge$  Rep  $-' r = Rep\ -' s$ )
    unfolding rel-vset-def vimage-def set-eq-iff
    by auto (metis rep-abs-rsp[OF assms] assms[simplified Quotient3-def])+
qed

declare [mapQ3 set = (rel-vset, set-quotient)]

lemma empty-set-rsp[quot-respect]:
  rel-vset R {} {}
  unfolding rel-vset-def by simp

lemma collect-rsp[quot-respect]:
  assumes Quotient3 R Abs Rep
  shows ((R  $\implies$  (=))  $\implies$  rel-vset R) Collect Collect
  by (intro rel-funI) (simp add: rel-fun-def rel-vset-def)

lemma collect-prs[quot-preserve]:
  assumes Quotient3 R Abs Rep
  shows ((Abs  $\dashv\dashv\dashv id$ )  $\dashv\dashv\dashv$  ( $-'$ ) Rep) Collect = Collect
  unfolding fun-eq-iff
  by (simp add: Quotient3-abs-rep[OF assms])

lemma union-rsp[quot-respect]:
  assumes Quotient3 R Abs Rep
  shows (rel-vset R  $\implies$  rel-vset R  $\implies$  rel-vset R) ( $\cup$ ) ( $\cup$ )
  by (intro rel-funI) (simp add: rel-vset-def)

lemma union-prs[quot-preserve]:

```

```

assumes Quotient3 R Abs Rep
shows  $((-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Rep)\ (\cup) = (\cup)$ 
unfolding fun-eq-iff
by (simp add: Quotient3-abs-rep[OF set-quotient[OF assms]])

lemma diff-rsp[quot-respect]:
assumes Quotient3 R Abs Rep
shows  $(rel-vset\ R \implies rel-vset\ R \implies rel-vset\ R)\ (-)\ (-)$ 
by (intro rel-funI) (simp add: rel-vset-def)

lemma diff-prs[quot-preserve]:
assumes Quotient3 R Abs Rep
shows  $((-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Rep)\ (-) = (-)$ 
unfolding fun-eq-iff
by (simp add: Quotient3-abs-rep[OF set-quotient[OF assms]] vimage-Diff)

lemma inter-rsp[quot-respect]:
assumes Quotient3 R Abs Rep
shows  $(rel-vset\ R \implies rel-vset\ R \implies rel-vset\ R)\ (\cap)\ (\cap)$ 
by (intro rel-funI) (auto simp add: rel-vset-def)

lemma inter-prs[quot-preserve]:
assumes Quotient3 R Abs Rep
shows  $((-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow (-\dot{\cdot})\ Rep)\ (\cap) = (\cap)$ 
unfolding fun-eq-iff
by (simp add: Quotient3-abs-rep[OF set-quotient[OF assms]])

lemma mem-prs[quot-preserve]:
assumes Quotient3 R Abs Rep
shows  $(Rep \dashrightarrow (-\dot{\cdot})\ Abs \dashrightarrow id)\ (\in) = (\in)$ 
by (simp add: fun-eq-iff Quotient3-abs-rep[OF assms])

lemma mem-rsp[quot-respect]:
shows  $(R \implies rel-vset\ R \implies (=))\ (\in)\ (\in)$ 
by (intro rel-funI) (simp add: rel-vset-def)

end

```

85 Quotient infrastructure for the product type

```

theory Quotient-Product
imports Quotient-Syntax
begin

```

85.1 Rules for the Quotient package

```

lemma map-prod-id [id-simps]:
shows map-prod id id = id
by (simp add: fun-eq-iff)

```

```

lemma rel-prod-eq [id-simps]:
  shows rel-prod (=) (=) = (=)
  by (simp add: fun-eq-iff)

```

```

lemma prod-equivp [quot-equiv]:
  assumes equivp R1
  assumes equivp R2
  shows equivp (rel-prod R1 R2)
  using assms by (auto intro!: equivpI reflpI sympI transpI elim!: equivpE elim:
reflpE sympE transpE)

```

```

lemma prod-quotient [quot-thm]:
  assumes Quotient3 R1 Abs1 Rep1
  assumes Quotient3 R2 Abs2 Rep2
  shows Quotient3 (rel-prod R1 R2) (map-prod Abs1 Abs2) (map-prod Rep1 Rep2)
  apply (rule Quotient3I)
  apply (simp add: map-prod.compositionality comp-def map-prod.identity
Quotient3-abs-rep [OF assms(1)] Quotient3-abs-rep [OF assms(2)])
  apply (simp add: split-paired-all Quotient3-rel-rep [OF assms(1)] Quotient3-rel-rep
[OF assms(2)])
  using Quotient3-rel [OF assms(1)] Quotient3-rel [OF assms(2)]
  apply (auto simp add: split-paired-all)
  done

```

```

declare [[mapQ3 prod = (rel-prod, prod-quotient)]]

```

```

lemma Pair-rsp [quot-respect]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (R1  $\implies$  R2  $\implies$  rel-prod R1 R2) Pair Pair
  by (rule Pair-transfer)

```

```

lemma Pair-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (Rep1  $\dashv\dashv$  Rep2  $\dashv\dashv$  (map-prod Abs1 Abs2)) Pair = Pair
  apply(simp add: fun-eq-iff)
  apply(simp add: Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2])
  done

```

```

lemma fst-rsp [quot-respect]:
  assumes Quotient3 R1 Abs1 Rep1
  assumes Quotient3 R2 Abs2 Rep2
  shows (rel-prod R1 R2  $\implies$  R1) fst fst
  by auto

```

```

lemma fst-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1

```

assumes $q2$: *Quotient3* $R2$ $Abs2$ $Rep2$
shows ($map\text{-}prod\ Rep1\ Rep2\ \text{---}\>\ Abs1$) $fst = fst$
by (*simp add: fun-eq-iff Quotient3-abs-rep[OF q1]*)

lemma *snd-rsp [quot-respect]*:
assumes *Quotient3* $R1$ $Abs1$ $Rep1$
assumes *Quotient3* $R2$ $Abs2$ $Rep2$
shows ($rel\text{-}prod\ R1\ R2\ \text{==}\>\ R2$) $snd\ snd$
by *auto*

lemma *snd-prs [quot-preserve]*:
assumes $q1$: *Quotient3* $R1$ $Abs1$ $Rep1$
assumes $q2$: *Quotient3* $R2$ $Abs2$ $Rep2$
shows ($map\text{-}prod\ Rep1\ Rep2\ \text{---}\>\ Abs2$) $snd = snd$
by (*simp add: fun-eq-iff Quotient3-abs-rep[OF q2]*)

lemma *case-prod-rsp [quot-respect]*:
shows (($R1\ \text{==}\>\ R2\ \text{==}\>\ (=)$) $\text{==}\>\ (rel\text{-}prod\ R1\ R2)\ \text{==}\>\ (=)$)
case-prod case-prod
by (*rule case-prod-transfer*)

lemma *split-prs [quot-preserve]*:
assumes $q1$: *Quotient3* $R1$ $Abs1$ $Rep1$
and $q2$: *Quotient3* $R2$ $Abs2$ $Rep2$
shows ((($Abs1\ \text{---}\>\ Abs2\ \text{---}\>\ id$) $\text{---}\>\ map\text{-}prod\ Rep1\ Rep2\ \text{---}\>\ id$)
case-prod) = *case-prod*
by (*simp add: fun-eq-iff Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]*)

lemma [*quot-respect*]:
shows (($R2\ \text{==}\>\ R2\ \text{==}\>\ (=)$) $\text{==}\>\ (R1\ \text{==}\>\ R1\ \text{==}\>\ (=))\ \text{==}\>\ rel\text{-}prod\ R2\ R1\ \text{==}\>\ rel\text{-}prod\ R2\ R1\ \text{==}\>\ (=)$) $rel\text{-}prod\ rel\text{-}prod$
by (*rule prod.rel-transfer*)

lemma [*quot-preserve*]:
assumes $q1$: *Quotient3* $R1$ $abs1$ $rep1$
and $q2$: *Quotient3* $R2$ $abs2$ $rep2$
shows (($abs1\ \text{---}\>\ abs1\ \text{---}\>\ id$) $\text{---}\>\ (abs2\ \text{---}\>\ abs2\ \text{---}\>\ id)$
 $\text{---}\>$
 $map\text{-}prod\ rep1\ rep2\ \text{---}\>\ map\text{-}prod\ rep1\ rep2\ \text{---}\>\ id$) $rel\text{-}prod = rel\text{-}prod$
by (*simp add: fun-eq-iff Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2]*)

lemma [*quot-preserve*]:
shows($rel\text{-}prod\ ((rep1\ \text{---}\>\ rep1\ \text{---}\>\ id)\ R1)\ ((rep2\ \text{---}\>\ rep2\ \text{---}\>\ id)\ R2)$
 $(l1, l2)\ (r1, r2)) = (R1\ (rep1\ l1)\ (rep1\ r1) \wedge R2\ (rep2\ l2)\ (rep2\ r2))$
by *simp*

declare *prod.inject[quot-preserve]*

end

86 Quotient infrastructure for the option type

```
theory Quotient-Option
imports Quotient-Syntax
begin
```

86.1 Rules for the Quotient package

lemma *rel-option-map1*:

```
rel-option R (map-option f x) y  $\longleftrightarrow$  rel-option ( $\lambda x. R (f x)$ ) x y
by (simp add: rel-option-iff split: option.split)
```

lemma *rel-option-map2*:

```
rel-option R x (map-option f y)  $\longleftrightarrow$  rel-option ( $\lambda x y. R x (f y)$ ) x y
by (simp add: rel-option-iff split: option.split)
```

declare

```
map-option.id [id-simps]
option.rel-eq [id-simps]
```

lemma *reflp-rel-option*:

```
reflp R  $\implies$  reflp (rel-option R)
unfolding reflp-def split-option-all by simp
```

lemma *option-symp*:

```
symp R  $\implies$  symp (rel-option R)
unfolding symp-def split-option-all
by (simp only: option.rel-inject option.rel-distinct) fast
```

lemma *option-transp*:

```
transp R  $\implies$  transp (rel-option R)
unfolding transp-def split-option-all
by (simp only: option.rel-inject option.rel-distinct) fast
```

lemma *option-equivp* [quot-equiv]:

```
equivp R  $\implies$  equivp (rel-option R)
by (blast intro: equivpI reflp-rel-option option-symp option-transp elim: equivpE)
```

lemma *option-quotient* [quot-thm]:

```
assumes Quotient3 R Abs Rep
shows Quotient3 (rel-option R) (map-option Abs) (map-option Rep)
apply (rule Quotient3I)
apply (simp-all add: option.map-comp comp-def option.map-id[unfolded id-def]
option.rel-eq rel-option-map1 rel-option-map2 Quotient3-abs-rep [OF assms] Quo-
tient3-rel-rep [OF assms])
using Quotient3-rel [OF assms]
apply (simp add: rel-option-iff split: option.split)
```

```

done

declare [[mapQ3 option = (rel-option, option-quotient)]]

lemma option-None-rsp [quot-respect]:
  assumes q: Quotient3 R Abs Rep
  shows rel-option R None None
  by (rule option.ctr-transfer(1))

lemma option-Some-rsp [quot-respect]:
  assumes q: Quotient3 R Abs Rep
  shows (R ==> rel-option R) Some Some
  by (rule option.ctr-transfer(2))

lemma option-None-prs [quot-preserve]:
  assumes q: Quotient3 R Abs Rep
  shows map-option Abs None = None
  by (rule Option.option.map(1))

lemma option-Some-prs [quot-preserve]:
  assumes q: Quotient3 R Abs Rep
  shows (Rep ---> map-option Abs) Some = Some
  apply (simp add: fun-eq-iff)
  apply (simp add: Quotient3-abs-rep[OF q])
  done

end

```

87 Quotient infrastructure for the list type

```

theory Quotient-List
imports Quotient-Set Quotient-Product Quotient-Option
begin

```

87.1 Rules for the Quotient package

```

lemma map-id [id-simps]:
  map id = id
  by (fact List.map.id)

lemma list-all2-eq [id-simps]:
  list-all2 (=) = (=)
proof (rule ext)+
  fix xs ys
  show list-all2 (=) xs ys <=> xs = ys
  by (induct xs ys rule: list-induct2') simp-all
qed

lemma reflp-list-all2:

```



```

assumes reflp R
shows reflp (list-all2 R)
proof (rule reflpI)
  from assms have *:  $\bigwedge xs. R\ xs\ xs$  by (rule reflpE)
  fix xs
  show list-all2 R xs xs
    by (induct xs) (simp-all add: *)
qed

```

```

lemma list-symp:
  assumes symp R
  shows symp (list-all2 R)
proof (rule sympI)
  from assms have *:  $\bigwedge xs\ ys. R\ xs\ ys \implies R\ ys\ xs$  by (rule sympE)
  fix xs ys
  assume list-all2 R xs ys
  then show list-all2 R ys xs
    by (induct xs ys rule: list-induct2') (simp-all add: *)
qed

```

```

lemma list-transp:
  assumes transp R
  shows transp (list-all2 R)
proof (rule transpI)
  from assms have *:  $\bigwedge xs\ ys\ zs. R\ xs\ ys \implies R\ ys\ zs \implies R\ xs\ zs$  by (rule transpE)
  fix xs ys zs
  assume list-all2 R xs ys and list-all2 R ys zs
  then show list-all2 R xs zs
    by (induct arbitrary: zs) (auto simp: list-all2-Cons1 intro: *)
qed

```

```

lemma list-equivp [quot-equiv]:
  equivp R  $\implies$  equivp (list-all2 R)
  by (blast intro: equivpI reflp-list-all2 list-symp list-transp elim: equivpE)

```

```

lemma list-quotient3 [quot-thm]:
  assumes Quotient3 R Abs Rep
  shows Quotient3 (list-all2 R) (map Abs) (map Rep)
proof (rule Quotient3I)
  from assms have  $\bigwedge x. Abs\ (Rep\ x) = x$  by (rule Quotient3-abs-rep)
  then show  $\bigwedge xs. map\ Abs\ (map\ Rep\ xs) = xs$  by (simp add: comp-def)
next
  from assms have  $\bigwedge x\ y. R\ (Rep\ x)\ (Rep\ y) \longleftrightarrow x = y$  by (rule Quotient3-rel-rep)
  then show  $\bigwedge xs. list-all2\ R\ (map\ Rep\ xs)\ (map\ Rep\ xs)$ 
    by (simp add: list-all2-map1 list-all2-map2 list-all2-eq)
next
  fix xs ys
  from assms have  $\bigwedge x\ y. R\ x\ x \wedge R\ y\ y \wedge Abs\ x = Abs\ y \longleftrightarrow R\ x\ y$  by (rule
    Quotient3-rel)

```

then show $\text{list-all2 } R \text{ } xs \text{ } ys \longleftrightarrow \text{list-all2 } R \text{ } xs \text{ } xs \wedge \text{list-all2 } R \text{ } ys \text{ } ys \wedge \text{map } Abs$
 $xs = \text{map } Abs \text{ } ys$

by (*induct xs ys rule: list-induct2'*) *auto*
qed

declare $[[\text{mapQ3 } list = (\text{list-all2}, \text{list-quotient3})]]$

lemma *cons-prs* [*quot-preserve*]:

assumes $q: \text{Quotient3 } R \text{ } Abs \text{ } Rep$

shows $(Rep \dashrightarrow (\text{map } Rep) \dashrightarrow (\text{map } Abs)) (\#) = (\#)$

by (*auto simp add: fun-eq-iff comp-def Quotient3-abs-rep [OF q]*)

lemma *cons-rsp* [*quot-respect*]:

assumes $q: \text{Quotient3 } R \text{ } Abs \text{ } Rep$

shows $(R \implies \text{list-all2 } R \implies \text{list-all2 } R) (\#) (\#)$

by *auto*

lemma *nil-prs* [*quot-preserve*]:

assumes $q: \text{Quotient3 } R \text{ } Abs \text{ } Rep$

shows $\text{map } Abs \ [] = []$

by *simp*

lemma *nil-rsp* [*quot-respect*]:

assumes $q: \text{Quotient3 } R \text{ } Abs \text{ } Rep$

shows $\text{list-all2 } R \ [] \ []$

by *simp*

lemma *map-prs-aux*:

assumes $a: \text{Quotient3 } R1 \text{ } abs1 \text{ } rep1$

and $b: \text{Quotient3 } R2 \text{ } abs2 \text{ } rep2$

shows $(\text{map } abs2) (\text{map } ((abs1 \dashrightarrow rep2) f) (\text{map } rep1 \text{ } l)) = \text{map } f \text{ } l$

by (*induct l*)

(*simp-all add: Quotient3-abs-rep[OF a] Quotient3-abs-rep[OF b]*)

lemma *map-prs* [*quot-preserve*]:

assumes $a: \text{Quotient3 } R1 \text{ } abs1 \text{ } rep1$

and $b: \text{Quotient3 } R2 \text{ } abs2 \text{ } rep2$

shows $((abs1 \dashrightarrow rep2) \dashrightarrow (\text{map } rep1) \dashrightarrow (\text{map } abs2)) \text{map} = \text{map}$

and $((abs1 \dashrightarrow id) \dashrightarrow \text{map } rep1 \dashrightarrow id) \text{map} = \text{map}$

by (*simp-all only: fun-eq-iff map-prs-aux[OF a b] comp-def*)

(*simp-all add: Quotient3-abs-rep[OF a] Quotient3-abs-rep[OF b]*)

lemma *map-rsp* [*quot-respect*]:

assumes $q1: \text{Quotient3 } R1 \text{ } Abs1 \text{ } Rep1$

and $q2: \text{Quotient3 } R2 \text{ } Abs2 \text{ } Rep2$

shows $((R1 \implies R2) \implies (\text{list-all2 } R1) \implies \text{list-all2 } R2) \text{map } \text{map}$

and $((R1 \implies (=)) \implies (\text{list-all2 } R1) \implies (=)) \text{map } \text{map}$

unfolding *list-all2-eq [symmetric]* **by** (*rule list.map-transfer*)**+**

lemma *foldr-prs-aux*:

assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows *abs2 (foldr ((abs1 ----> abs2 ----> rep2) f) (map rep1 l) (rep2 e))*
 $= \text{foldr } f \ l \ e$
by (*induct l*) (*simp-all add: Quotient3-abs-rep[OF a] Quotient3-abs-rep[OF b]*)

lemma *foldr-prs [quot-preserve]*:

assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows $((abs1 \text{ ----> } abs2 \text{ ----> } rep2) \text{ ----> } (map \ rep1) \text{ ----> } rep2 \text{ ----> } abs2) \text{ foldr} = \text{foldr}$
apply (*simp add: fun-eq-iff*)
by (*simp only: fun-eq-iff foldr-prs-aux[OF a b]*)
(simp)

lemma *foldl-prs-aux*:

assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows *abs1 (foldl ((abs1 ----> abs2 ----> rep1) f) (rep1 e) (map rep2 l)) =*
 $\text{foldl } f \ e \ l$
by (*induct l arbitrary: e*) (*simp-all add: Quotient3-abs-rep[OF a] Quotient3-abs-rep[OF b]*)

lemma *foldl-prs [quot-preserve]*:

assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows $((abs1 \text{ ----> } abs2 \text{ ----> } rep1) \text{ ----> } rep1 \text{ ----> } (map \ rep2) \text{ ----> } abs1) \text{ foldl} = \text{foldl}$
by (*simp add: fun-eq-iff foldl-prs-aux [OF a b]*)

lemma *foldl-rsp[quot-respect]*:

assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows $((R1 \text{ ===> } R2 \text{ ===> } R1) \text{ ===> } R1 \text{ ===> } list\text{-all2 } R2 \text{ ===> } R1)$
 foldl foldl
by (*rule foldl-transfer*)

lemma *foldr-rsp[quot-respect]*:

assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows $((R1 \text{ ===> } R2 \text{ ===> } R2) \text{ ===> } list\text{-all2 } R1 \text{ ===> } R2 \text{ ===> } R2)$
 foldr foldr
by (*rule foldr-transfer*)

lemma *list-all2-rsp*:

assumes *r*: $\forall x \ y. \ R \ x \ y \longrightarrow (\forall a \ b. \ R \ a \ b \longrightarrow S \ x \ a = T \ y \ b)$
and *l1*: *list-all2 R x y*
and *l2*: *list-all2 R a b*

```

shows list-all2 S x a = list-all2 T y b
using l1 l2
by (induct arbitrary: a b rule: list-all2-induct,
    auto simp: list-all2-Cons1 list-all2-Cons2 r)

lemma [quot-respect]:
  ((R ==> R ==> (=)) ==> list-all2 R ==> list-all2 R ==> (=))
list-all2 list-all2
  by (rule list.rel-transfer)

lemma [quot-preserve]:
  assumes a: Quotient3 R abs1 rep1
  shows ((abs1 ----> abs1 ----> id) ----> map rep1 ----> map rep1 ---->
id) list-all2 = list-all2
  apply (simp add: fun-eq-iff)
  apply clarify
  apply (induct-tac xa xb rule: list-induct2')
  apply (simp-all add: Quotient3-abs-rep[OF a])
  done

lemma [quot-preserve]:
  assumes a: Quotient3 R abs1 rep1
  shows (list-all2 ((rep1 ----> rep1 ----> id) R) l m) = (l = m)
  by (induct l m rule: list-induct2') (simp-all add: Quotient3-rel-rep[OF a])

lemma list-all2-find-element:
  assumes a: x ∈ set a
  and b: list-all2 R a b
  shows ∃ y. (y ∈ set b ∧ R x y)
  using b a by induct auto

lemma list-all2-refl:
  assumes a: ∧ x y. R x y = (R x = R y)
  shows list-all2 R x x
  by (induct x) (auto simp add: a)

end

```

88 Quotient infrastructure for the sum type

```

theory Quotient-Sum
imports Quotient-Syntax
begin

```

88.1 Rules for the Quotient package

```

lemma rel-sum-map1:
  rel-sum R1 R2 (map-sum f1 f2 x) y ⟷ rel-sum (λx. R1 (f1 x)) (λx. R2 (f2 x))
x y

```

by (*rule* *sum.rel-map*(1))

lemma *rel-sum-map2*:

rel-sum *R1* *R2* *x* (*map-sum* *f1* *f2* *y*) \longleftrightarrow *rel-sum* ($\lambda x\ y. R1\ x\ (f1\ y)$) ($\lambda x\ y. R2\ x\ (f2\ y)$) *x* *y*
by (*rule* *sum.rel-map*(2))

lemma *map-sum-id* [*id-simps*]:

map-sum *id* *id* = *id*
by (*simp* *add*: *id-def* *map-sum.identity* *fun-eq-iff*)

lemma *rel-sum-eq* [*id-simps*]:

rel-sum (=) (=) = (=)
by (*rule* *sum.rel-eq*)

lemma *reflp-rel-sum*:

reflp *R1* \implies *reflp* *R2* \implies *reflp* (*rel-sum* *R1* *R2*)
unfolding *reflp-def* *split-sum-all* *rel-sum-simps* **by** *fast*

lemma *sum-symp*:

symp *R1* \implies *symp* *R2* \implies *symp* (*rel-sum* *R1* *R2*)
unfolding *symp-def* *split-sum-all* *rel-sum-simps* **by** *fast*

lemma *sum-transp*:

transp *R1* \implies *transp* *R2* \implies *transp* (*rel-sum* *R1* *R2*)
unfolding *transp-def* *split-sum-all* *rel-sum-simps* **by** *fast*

lemma *sum-equivp* [*quot-equiv*]:

equivp *R1* \implies *equivp* *R2* \implies *equivp* (*rel-sum* *R1* *R2*)
by (*blast* *intro*: *equivpI* *reflp-rel-sum* *sum-symp* *sum-transp* *elim*: *equivpE*)

lemma *sum-quotient* [*quot-thm*]:

assumes *q1*: *Quotient3* *R1* *Abs1* *Rep1*
assumes *q2*: *Quotient3* *R2* *Abs2* *Rep2*
shows *Quotient3* (*rel-sum* *R1* *R2*) (*map-sum* *Abs1* *Abs2*) (*map-sum* *Rep1* *Rep2*)
apply (*rule* *Quotient3I*)
apply (*simp-all* *add*: *map-sum.compositionality* *comp-def* *map-sum.identity* *rel-sum-eq* *rel-sum-map1* *rel-sum-map2*
Quotient3-abs-rep [*OF* *q1*] *Quotient3-rel-rep* [*OF* *q1*] *Quotient3-abs-rep* [*OF* *q2*] *Quotient3-rel-rep* [*OF* *q2*])
using *Quotient3-rel* [*OF* *q1*] *Quotient3-rel* [*OF* *q2*]
apply (*fastforce* *elim*!: *rel-sum.cases* *simp* *add*: *comp-def* *split*: *sum.split*)
done

declare [[*mapQ3* *sum* = (*rel-sum*, *sum-quotient*)]]

lemma *sum-Inl-rsp* [*quot-respect*]:

assumes *q1*: *Quotient3* *R1* *Abs1* *Rep1*
assumes *q2*: *Quotient3* *R2* *Abs2* *Rep2*

```

shows (R1 ==> rel-sum R1 R2) Inl Inl
by auto

lemma sum-Inr-rsp [quot-respect]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (R2 ==> rel-sum R1 R2) Inr Inr
  by auto

lemma sum-Inl-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (Rep1 ---> map-sum Abs1 Abs2) Inl = Inl
  apply(simp add: fun-eq-iff)
  apply(simp add: Quotient3-abs-rep[OF q1])
  done

lemma sum-Inr-prs [quot-preserve]:
  assumes q1: Quotient3 R1 Abs1 Rep1
  assumes q2: Quotient3 R2 Abs2 Rep2
  shows (Rep2 ---> map-sum Abs1 Abs2) Inr = Inr
  apply(simp add: fun-eq-iff)
  apply(simp add: Quotient3-abs-rep[OF q2])
  done

end

```

89 Quotient types

```

theory Quotient-Type
imports Main
begin

```

We introduce the notion of quotient types over equivalence relations via type classes.

89.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a \Rightarrow 'a \Rightarrow \text{bool}$.

```

class eqv =
  fixes eqv :: 'a => 'a => bool (infixl <~> 50)

class equiv = eqv +
  assumes equiv-refl [intro]: x ~ x
  and equiv-trans [trans]: x ~ y ==> y ~ z ==> x ~ z
  and equiv-sym [sym]: x ~ y ==> y ~ x
begin

```

```

lemma equiv-not-sym [sym]:  $\neg x \sim y \implies \neg y \sim x$ 
proof –
  assume  $\neg x \sim y$ 
  then show  $\neg y \sim x$  by (rule contrapos-nn) (rule equiv-sym)
qed

lemma not-equiv-trans1 [trans]:  $\neg x \sim y \implies y \sim z \implies \neg x \sim z$ 
proof –
  assume  $\neg x \sim y$  and  $y \sim z$ 
  show  $\neg x \sim z$ 
  proof
    assume  $x \sim z$ 
    also from  $\langle y \sim z \rangle$  have  $z \sim y$  ..
    finally have  $x \sim y$  .
    with  $\langle \neg x \sim y \rangle$  show False by contradiction
  qed
qed

lemma not-equiv-trans2 [trans]:  $x \sim y \implies \neg y \sim z \implies \neg x \sim z$ 
proof –
  assume  $\neg y \sim z$ 
  then have  $\neg z \sim y$  ..
  also
  assume  $x \sim y$ 
  then have  $y \sim x$  ..
  finally have  $\neg z \sim x$  .
  then show  $\neg x \sim z$  ..
qed

end

```

The quotient type '*a quot*' consists of all *equivalence classes* over elements of the base type '*a*'.

definition (in *eqv*) *quot* = $\{\{x. a \sim x\} \mid a. \text{True}\}$

typedef (overloaded) '*a quot*' = *quot* :: '*a*::*eqv set set*'
unfolding *quot-def* **by** *blast*

lemma *quotI* [*intro*]: $\{x. a \sim x\} \in \text{quot}$
unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]:
assumes $R \in \text{quot}$
obtains *a* **where** $R = \{x. a \sim x\}$
using *assms* **unfolding** *quot-def* **by** *blast*

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition *class* :: '*a*::*equiv*' \Rightarrow '*a quot*' ($\langle \langle \text{open-block notation} = \text{mixfix class} \rangle \rangle [-] \rangle$)

where $\lfloor a \rfloor = \text{Abs-quot } \{x. a \sim x\}$

theorem *quot-exhaust*: $\exists a. A = \lfloor a \rfloor$

proof (*cases A*)

fix R

assume $R: A = \text{Abs-quot } R$

assume $R \in \text{quot}$

then have $\exists a. R = \{x. a \sim x\}$ **by** *blast*

with R have $\exists a. A = \text{Abs-quot } \{x. a \sim x\}$ **by** *blast*

then show *?thesis unfolding class-def* .

qed

lemma *quot-cases* [*cases type: quot*]:

obtains a where $A = \lfloor a \rfloor$

using *quot-exhaust* **by** *blast*

89.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $\lfloor a \rfloor = \lfloor b \rfloor \longleftrightarrow a \sim b$

proof

assume *eq*: $\lfloor a \rfloor = \lfloor b \rfloor$

show $a \sim b$

proof –

from *eq* have $\{x. a \sim x\} = \{x. b \sim x\}$

by (*simp only: class-def Abs-quot-inject quotI*)

moreover have $a \sim a$..

ultimately have $a \in \{x. b \sim x\}$ **by** *blast*

then have $b \sim a$ **by** *blast*

then show *?thesis* ..

qed

next

assume *ab*: $a \sim b$

show $\lfloor a \rfloor = \lfloor b \rfloor$

proof –

have $\{x. a \sim x\} = \{x. b \sim x\}$

proof (*rule Collect-cong*)

fix x show $(a \sim x) = (b \sim x)$

proof

from *ab* have $b \sim a$..

also assume $a \sim x$

finally show $b \sim x$.

next

note *ab*

also assume $b \sim x$

finally show $a \sim x$.

qed

qed

then show *?thesis* **by** (*simp only: class-def*)

qed
qed

89.3 Picking representing elements

definition *pick* :: 'a::equiv quot \Rightarrow 'a
 where *pick* $A = (\text{SOME } a. A = \lfloor a \rfloor)$

theorem *pick-equiv* [intro]: *pick* $\lfloor a \rfloor \sim a$

proof (*unfold pick-def*)

show $(\text{SOME } x. \lfloor a \rfloor = \lfloor x \rfloor) \sim a$

proof (*rule someI2*)

show $\lfloor a \rfloor = \lfloor a \rfloor$..

fix x assume $\lfloor a \rfloor = \lfloor x \rfloor$

then have $a \sim x$..

then show $x \sim a$..

qed

qed

theorem *pick-inverse* [intro]: $\lfloor \text{pick } A \rfloor = A$

proof (*cases A*)

fix a assume $a: A = \lfloor a \rfloor$

then have $\text{pick } A \sim a$ by (*simp only: pick-equiv*)

then have $\lfloor \text{pick } A \rfloor = \lfloor a \rfloor$..

with a show ?thesis by *simp*

qed

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

theorem *quot-cond-function*:

assumes *eq*: $\bigwedge X Y. P X Y \Longrightarrow f X Y \equiv g (\text{pick } X) (\text{pick } Y)$

and *cong*: $\bigwedge x x' y y'. \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$

$\Longrightarrow P \lfloor x \rfloor \lfloor y \rfloor \Longrightarrow P \lfloor x' \rfloor \lfloor y' \rfloor \Longrightarrow g x y = g x' y'$

and *P*: $P \lfloor a \rfloor \lfloor b \rfloor$

shows $f \lfloor a \rfloor \lfloor b \rfloor = g a b$

proof –

from *eq* and *P* have $f \lfloor a \rfloor \lfloor b \rfloor = g (\text{pick } \lfloor a \rfloor) (\text{pick } \lfloor b \rfloor)$ by (*simp only:*)

also have $\dots = g a b$

proof (*rule cong*)

show $\lfloor \text{pick } \lfloor a \rfloor \rfloor = \lfloor a \rfloor$..

moreover

show $\lfloor \text{pick } \lfloor b \rfloor \rfloor = \lfloor b \rfloor$..

moreover

show $P \lfloor a \rfloor \lfloor b \rfloor$ by (*rule P*)

ultimately show $P \lfloor \text{pick } \lfloor a \rfloor \rfloor \lfloor \text{pick } \lfloor b \rfloor \rfloor$ by (*simp only:*)

qed

finally show ?thesis .

qed

theorem *quot-function*:

assumes $\bigwedge X Y. f X Y \equiv g (pick X) (pick Y)$
and $\bigwedge x x' y y'. [x] = [x'] \implies [y] = [y'] \implies g x y = g x' y'$
shows $f [a] [b] = g a b$
using *assms* **and** *TrueI*
by (*rule quot-cond-function*)

theorem *quot-function'*:

$(\bigwedge X Y. f X Y \equiv g (pick X) (pick Y)) \implies$
 $(\bigwedge x x' y y'. x \sim x' \implies y \sim y' \implies g x y = g x' y') \implies$
 $f [a] [b] = g a b$
by (*rule quot-function*) (*simp-all only: quot-equality*)

end

90 Ramsey’s Theorem

theory *Ramsey*

imports *Infinite-Set Equipollence FuncSet*
begin

90.1 Preliminary definitions

abbreviation *strict-sorted* :: ‘*a::linorder list* \Rightarrow *bool*’ **where**
strict-sorted \equiv *sorted-wrt* ($<$)

90.1.1 The n -element subsets of a set A

definition *nsets* :: ‘*a set, nat*’ \Rightarrow ‘*a set set*’ ($\langle \langle notation = \langle mixfix nsets \rangle \rangle [-]^{\sim} \rangle$
 $[0,999] 999$)
where $nsets A n \equiv \{N. N \subseteq A \wedge finite N \wedge card N = n\}$

lemma *finite-imp-finite-nsets*: $finite A \implies finite ([A]^k)$
by (*simp add: nsets-def*)

lemma *nsets-mono*: $A \subseteq B \implies nsets A n \subseteq nsets B n$
by (*auto simp: nsets-def*)

lemma *nsets-Pi-contra*: $A' \subseteq A \implies Pi ([A]^n) B \subseteq Pi ([A']^n) B$
by (*auto simp: nsets-def*)

lemma *nsets-2-eq*: $[A]^2 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \{\{x, y\}\})$
by (*auto simp: nsets-def card-2-iff*)

lemma *nsets2-E*:

assumes $e \in [A]^2$
obtains $x y$ **where** $e = \{x, y\}$ $x \in A$ $y \in A$ $x \neq y$
using *assms* **by** (*auto simp: nsets-def card-2-iff*)

lemma *nsets-doubleton-2-eq* [simp]: $[\{x, y\}]^2 = (\text{if } x=y \text{ then } \{\} \text{ else } \{\{x, y\}\})$
by (auto simp: *nsets-2-eq*)

lemma *doubleton-in-nsets-2* [simp]: $\{x, y\} \in [A]^2 \longleftrightarrow x \in A \wedge y \in A \wedge x \neq y$
by (auto simp: *nsets-2-eq Set.doubleton-eq-iff*)

lemma *nsets-3-eq*: $[A]^3 = (\bigcup x \in A. \bigcup y \in A - \{x\}. \bigcup z \in A - \{x, y\}. \{\{x, y, z\}\})$
by (simp add: *eval-nat-numeral nsets-def card-Suc-eq*) blast

lemma *nsets-4-eq*: $[A]^4 = (\bigcup u \in A. \bigcup x \in A - \{u\}. \bigcup y \in A - \{u, x\}. \bigcup z \in A - \{u, x, y\}. \{\{u, x, y, z\}\})$
(is - = ?rhs)

proof

show $[A]^4 \subseteq ?rhs$

by (clarsimp simp add: *nsets-def eval-nat-numeral card-Suc-eq*) blast

have $\bigwedge X. X \in ?rhs \implies \text{card } X = 4$

by (force simp: *card-2-iff*)

then show $?rhs \subseteq [A]^4$

by (auto simp: *nsets-def*)

qed

lemma *nsets-disjoint-2*:

$X \cap Y = \{\} \implies [X \cup Y]^2 = [X]^2 \cup [Y]^2 \cup (\bigcup x \in X. \bigcup y \in Y. \{\{x, y\}\})$

by (fastforce simp: *nsets-2-eq Set.doubleton-eq-iff*)

lemma *ordered-nsets-2-eq*:

fixes $A :: 'a::\text{linorder set}$

shows $[A]^2 = \{\{x, y\} \mid x \neq y. x \in A \wedge y \in A \wedge x < y\}$

(is - = ?rhs)

proof

show $[A]^2 \subseteq ?rhs$

by (auto simp: *nsets-def card-2-iff doubleton-eq-iff neq-iff*)

show $?rhs \subseteq [A]^2$

unfolding numeral-nat by (auto simp: *nsets-def card-Suc-eq*)

qed

lemma *ordered-nsets-3-eq*:

fixes $A :: 'a::\text{linorder set}$

shows $[A]^3 = \{\{x, y, z\} \mid x \neq y \neq z. x \in A \wedge y \in A \wedge z \in A \wedge x < y \wedge y < z\}$

(is - = ?rhs)

proof

show $[A]^3 \subseteq ?rhs$

unfolding *nsets-def card-3-iff*

by (smt (verit, del-insts) *Collect-mono-iff insert-commute insert-subset linorder-less-linear*)

have $\bigwedge X. X \in ?rhs \implies \text{card } X = 3$

by (force simp: *card-3-iff*)

then show $?rhs \subseteq [A]^3$

by (auto simp: nsets-def)
qed

lemma ordered-nsets-4-eq:
fixes $A :: 'a::linorder\ set$
defines $rhs \equiv \lambda U. \exists u\ x\ y\ z. U = \{u, x, y, z\} \wedge u \in A \wedge x \in A \wedge y \in A \wedge z \in A$
 $\wedge u < x \wedge x < y \wedge y < z$
shows $[A]^4 = Collect\ rhs$
proof –
have $rhs\ U$ if $U \in [A]^4$ for U
proof –
from that obtain l where $strict_sorted\ l$ $List.set\ l = U$ $length\ l = 4$ $U \subseteq A$
by (simp add: nsets-def) (metis finite-set-strict-sorted)
then show ?thesis
unfolding numeral-nat length-Suc-conv rhs-def by auto blast
qed
moreover
have $\bigwedge X. X \in Collect\ rhs \implies card\ X = 4 \wedge finite\ X \wedge X \subseteq A$
by (auto simp: rhs-def card-insert-if)
ultimately show ?thesis
unfolding nsets-def by blast
qed

lemma ordered-nsets-5-eq:
fixes $A :: 'a::linorder\ set$
defines $rhs \equiv \lambda U. \exists u\ v\ x\ y\ z. U = \{u, v, x, y, z\} \wedge u \in A \wedge v \in A \wedge x \in A \wedge y \in A \wedge z \in A$
 $\wedge u < v \wedge v < x \wedge x < y \wedge y < z$
shows $[A]^5 = Collect\ rhs$
proof –
have $rhs\ U$ if $U \in [A]^5$ for U
proof –
from that obtain l where $strict_sorted\ l$ $List.set\ l = U$ $length\ l = 5$ $U \subseteq A$
by (simp add: nsets-def) (metis finite-set-strict-sorted)
then show ?thesis
unfolding numeral-nat length-Suc-conv rhs-def by auto blast
qed
moreover
have $\bigwedge X. X \in Collect\ rhs \implies card\ X = 5 \wedge finite\ X \wedge X \subseteq A$
by (auto simp: rhs-def card-insert-if)
ultimately show ?thesis
unfolding nsets-def by blast
qed

lemma binomial-eq-nsets: $n\ choose\ k = card\ (nsets\ \{0..<n\}\ k)$
proof –
have $\{K. K \subseteq \{0..<n\} \wedge card\ K = k\} = \{N. N \subseteq \{0..<n\} \wedge finite\ N \wedge card\ N = k\}$
using infinite-super by blast
then show ?thesis

by (*simp add: binomial-def nsets-def*)
qed

lemma *nsets-eq-empty-iff*: $nsets\ A\ r = \{\} \longleftrightarrow finite\ A \wedge card\ A < r$
unfolding *nsets-def*
proof (*intro iffI conjI*)
 assume that: $\{N. N \subseteq A \wedge finite\ N \wedge card\ N = r\} = \{\}$
 show *finite A*
 using *infinite-arbitrarily-large that* by *auto*
 then have $\neg r \leq card\ A$
 using that by (*simp add: set-eq-iff*) (*metis obtain-subset-with-card-n*)
 then show $card\ A < r$
 using *not-less* by *blast*
next
 show $\{N. N \subseteq A \wedge finite\ N \wedge card\ N = r\} = \{\}$
 if $finite\ A \wedge card\ A < r$
 using that *card-mono leD* by *auto*
 qed

lemma *nsets-eq-empty*: $\llbracket finite\ A; card\ A < r \rrbracket \Longrightarrow nsets\ A\ r = \{\}$
 by (*simp add: nsets-eq-empty-iff*)

lemma *nsets-empty-iff*: $nsets\ \{\}\ r = (if\ r=0\ then\ \{\{\}\}\ else\ \{\})$
 by (*auto simp: nsets-def*)

lemma *nsets-singleton-iff*: $nsets\ \{a\}\ r = (if\ r=0\ then\ \{\{\}\}\ else\ if\ r=1\ then\ \{\{a\}\}\ else\ \{\})$
 by (*auto simp: nsets-def card-gt-0-iff subset-singleton-iff*)

lemma *nsets-self* [*simp*]: $nsets\ \{..<m\}\ m = \{\{..<m\}\}$
proof
 show $\{\{..<m\}\}^m \subseteq \{\{..<m\}\}$
 by (*force simp add: card-subset-eq nsets-def*)
 qed (*simp add: nsets-def*)

lemma *nsets-zero* [*simp*]: $nsets\ A\ 0 = \{\{\}\}$
 by (*auto simp: nsets-def*)

lemma *nsets-one*: $nsets\ A\ (Suc\ 0) = (\lambda x. \{x\})\ 'A$
 using *card-eq-SucD* by (*force simp: nsets-def*)

lemma *inj-on-nsets*:
 assumes *inj-on f A*
 shows *inj-on* $(\lambda X. f\ 'X)\ ([A]^n)$
 using *assms* by (*simp add: nsets-def inj-on-def inj-on-image-eq-iff*)

lemma *bij-betw-nsets*:
 assumes *bij-betw f A B*
 shows *bij-betw* $(\lambda X. f\ 'X)\ ([A]^n)\ ([B]^n)$

proof –
 have $\bigwedge X. [X \subseteq f \circ A; \text{finite } X] \implies \exists Y \subseteq A. \text{finite } Y \wedge \text{card } Y = \text{card } X \wedge X = f \circ Y$
 by (metis card-image inj-on-finite order-refl subset-image-inj)
 then have $(\cdot) f \circ [A]^n = [f \circ A]^n$
 using assms by (auto simp: nsets-def bij-betw-def image-iff card-image inj-on-subset)
 with assms show ?thesis
 by (auto simp: bij-betw-def inj-on-nsets)
qed

lemma *nset-image-obtains*:
 assumes $X \in [f \circ A]^k \text{ inj-on } f A$
 obtains Y where $Y \in [A]^k X = f \circ Y$
proof
 show $X = f \circ (A \cap f \circ X)$
 using assms by (auto simp: nsets-def)
 then show $A \text{ Int } (f \circ X) \in [A]^k$
 using assms
 unfolding nsets-def mem-Collect-eq
 by (metis card-image finite-image-iff inf-le1 inj-on-subset)
qed

lemma *nsets-image-funcset*:
 assumes $g \in S \rightarrow T$ and $\text{inj-on } g S$
 shows $(\lambda X. g \circ X) \in [S]^k \rightarrow [T]^k$
 using assms
 by (fastforce simp: nsets-def card-image inj-on-subset subset-iff simp flip: image-subset-iff-funcset)

lemma *nsets-compose-image-funcset*:
 assumes $f: f \in [T]^k \rightarrow D$ and $g \in S \rightarrow T$ and $\text{inj-on } g S$
 shows $f \circ (\lambda X. g \circ X) \in [S]^k \rightarrow D$
proof –
 have $(\lambda X. g \circ X) \in [S]^k \rightarrow [T]^k$
 using assms by (simp add: nsets-image-funcset)
 then show ?thesis
 using f by fastforce
qed

90.1.2 Further properties, involving equipollence

lemma *nsets-lepoll-cong*:
 assumes $A \lesssim B$
 shows $[A]^k \lesssim [B]^k$
proof –
 obtain f where $f: \text{inj-on } f A \text{ } f \circ A \subseteq B$
 by (meson assms lepoll-def)
 define F where $F \equiv \lambda N. f \circ N$
 have $\text{inj-on } F ([A]^k)$

```

    using  $F$ -def  $f$  inj-on-nsets by blast
  moreover
  have  $F \text{ ‘ } ([A]^k) \subseteq [B]^k$ 
    by (metis  $F$ -def bij-betw-def bij-betw-nsets  $f$  nsets-mono)
  ultimately show ?thesis
    by (meson lepoll-def)
qed

```

```

lemma nsets-epoll-cong:
  assumes  $A \approx B$ 
  shows  $[A]^k \approx [B]^k$ 
  by (meson assms eqpoll-imp-lepoll eqpoll-sym lepoll-antisym nsets-lepoll-cong)

```

```

lemma infinite-imp-infinite-nsets:
  assumes inf: infinite  $A$  and  $k > 0$ 
  shows infinite  $([A]^k)$ 
proof -
  obtain  $B$  where  $B \subset A$   $A \approx B$ 
    by (meson inf infinite-iff-psubset)
  then obtain  $a$  where  $a: a \in A$   $a \notin B$ 
    by blast
  then obtain  $N$  where  $N \subseteq B$  finite  $N$  card  $N = k-1$   $a \notin N$ 
    by (metis  $\langle A \approx B \rangle$  inf eqpoll-finite-iff infinite-arbitrarily-large subset-eq)
  with  $a \langle k > 0 \rangle \langle B \subset A \rangle$  have insert  $a$   $N \in [A]^k$ 
    by (simp add: nsets-def)
  with  $a$  have nsets  $B$   $k \neq$  nsets  $A$   $k$ 
    by (metis (no-types, lifting) in-mono insertI1 mem-Collect-eq nsets-def)
  moreover have nsets  $B$   $k \subseteq$  nsets  $A$   $k$ 
    using  $\langle B \subset A \rangle$  nsets-mono by auto
  ultimately show ?thesis
    unfolding infinite-iff-psubset-le
    by (meson  $\langle A \approx B \rangle$  eqpoll-imp-lepoll nsets-epoll-cong psubsetI)
qed

```

```

lemma finite-nsets-iff:
  assumes  $k > 0$ 
  shows finite  $([A]^k) \longleftrightarrow$  finite  $A$ 
  using assms finite-imp-finite-nsets infinite-imp-infinite-nsets by blast

```

```

lemma card-nsets [simp]: card (nsets  $A$   $k$ ) = card  $A$  choose  $k$ 
proof (cases finite  $A$ )
  case True
    then show ?thesis
      by (metis bij-betw-nsets bij-betw-same-card binomial-eq-nsets ex-bij-betw-nat-finite)
  next
    case False
      then show ?thesis
        by (cases  $k=0$ ; simp add: finite-nsets-iff)
qed

```

90.1.3 Partition predicates

definition *monochromatic* $\equiv \lambda \beta \alpha \gamma f i. \exists H \in \text{nsets } \beta \alpha. f \text{ ‘ } (\text{nsets } H \gamma) \subseteq \{i\}$
 uniform partition sizes

definition *partn* $:: 'a \text{ set} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'b \text{ set} \Rightarrow \text{bool}$
where *partn* $\beta \alpha \gamma \delta \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \delta. \exists \xi \in \delta. \text{monochromatic } \beta \alpha \gamma f \xi$
 partition sizes enumerated in a list

definition *partn-lst* $:: 'a \text{ set} \Rightarrow \text{nat list} \Rightarrow \text{nat} \Rightarrow \text{bool}$
where *partn-lst* $\beta \alpha \gamma \equiv \forall f \in \text{nsets } \beta \gamma \rightarrow \{..<\text{length } \alpha\}. \exists i < \text{length } \alpha. \text{monochromatic } \beta (\alpha!i) \gamma f i$

There’s always a 0-clique

lemma *partn-lst-0*: $\gamma > 0 \implies \text{partn-lst } \beta (0\#\alpha) \gamma$
by (*force simp: partn-lst-def monochromatic-def nsets-empty-iff*)

lemma *partn-lst-0'*: $\gamma > 0 \implies \text{partn-lst } \beta (a\#0\#\alpha) \gamma$
by (*force simp: partn-lst-def monochromatic-def nsets-empty-iff*)

lemma *partn-lst-greater-resource*:

fixes $M::\text{nat}$

assumes $M: \text{partn-lst } \{..<M\} \alpha \gamma$ **and** $M \leq N$

shows $\text{partn-lst } \{..<N\} \alpha \gamma$

proof (*clarsimp simp: partn-lst-def*)

fix f

assume $f \in \text{nsets } \{..<N\} \gamma \rightarrow \{..<\text{length } \alpha\}$

then have $f \in \text{nsets } \{..<M\} \gamma \rightarrow \{..<\text{length } \alpha\}$

by (*meson Pi-anti-mono $\langle M \leq N \rangle$ lessThan-subset-iff nsets-mono subsetD*)

then obtain $i H$ **where** $i: i < \text{length } \alpha$ **and** $H: H \in \text{nsets } \{..<M\} (\alpha!i)$ **and**

subi: $f \text{ ‘ } \text{nsets } H \gamma \subseteq \{i\}$

using M **unfolding** *partn-lst-def monochromatic-def* **by** *blast*

have $H \in \text{nsets } \{..<N\} (\alpha!i)$

using $\langle M \leq N \rangle H$ **by** (*auto simp: nsets-def subset-iff*)

then show $\exists i < \text{length } \alpha. \text{monochromatic } \{..<N\} (\alpha!i) \gamma f i$

using i *subi* **unfolding** *monochromatic-def* **by** *blast*

qed

lemma *partn-lst-fewer-colours*:

assumes *major*: $\text{partn-lst } \beta (n\#\alpha) \gamma$ **and** $n \geq \gamma$

shows $\text{partn-lst } \beta \alpha \gamma$

proof (*clarsimp simp: partn-lst-def*)

fix $f :: 'a \text{ set} \Rightarrow \text{nat}$

assume $f: f \in [\beta]^\gamma \rightarrow \{..<\text{length } \alpha\}$

then obtain $i H$ **where** $i: i < \text{Suc } (\text{length } \alpha)$

and $H: H \in [\beta]^{((n\#\alpha)!i)}$

and *hom*: $\forall x \in [H]^\gamma. \text{Suc } (f x) = i$

using $\langle n \geq \gamma \rangle$ *major* [*unfolded partn-lst-def, rule-format, of Suc o f*]

by (*fastforce simp: image-subset-iff nsets-eq-empty-iff monochromatic-def*)

show $\exists i < \text{length } \alpha. \text{monochromatic } \beta (\alpha!i) \gamma f i$


```

proof (cases i)
  case 0
  then have  $[H]^\gamma = \{\}$ 
    using hom by blast
  then show ?thesis
    using 0 H  $\langle n \geq \gamma \rangle$ 
    by (simp add: nsets-eq-empty-iff) (simp add: nsets-def)
next
  case (Suc i')
  then show ?thesis
    unfolding monochromatic-def using i H hom by auto
qed
qed

```

```

lemma partn-lst-eq-partn: partn-lst  $\{.. $n\}$   $[m, m]$  2 = partn  $\{.. $n\}$  m 2  $\{.. $2::nat\}$ 
proof -
  have  $\bigwedge i. i < 2 \implies [m, m] ! i = m$ 
    using less-2-cases-iff by force
  then show ?thesis
    by (auto simp: partn-lst-def partn-def numeral-2-eq-2 cong: conj-cong)
qed$$$ 
```

```

lemma partn-lstE:
  assumes partn-lst  $\beta$   $\alpha$   $\gamma$   $f \in \text{nsets } \beta$   $\gamma \rightarrow \{.. $l\}$  length  $\alpha = l$ 
  obtains i H where  $i < \text{length } \alpha$   $H \in \text{nsets } \beta$   $(\alpha ! i) f ' (\text{nsets } H \ \gamma) \subseteq \{i\}$ 
  using partn-lst-def monochromatic-def assms by metis$ 
```

```

lemma partn-lst-less:
  assumes M: partn-lst  $\beta$   $\alpha$  n and eq: length  $\alpha' = \text{length } \alpha$ 
  and le:  $\bigwedge i. i < \text{length } \alpha \implies \alpha ! i \leq \alpha' ! i$ 
  shows partn-lst  $\beta$   $\alpha'$  n
proof (clarsimp simp: partn-lst-def)
  fix f
  assume  $f \in [\beta]^n \rightarrow \{.. $\text{length } \alpha'\}$ 
  then obtain i H where  $i < \text{length } \alpha$ 
    and  $H \subseteq \beta$  and  $H: \text{card } H = (\alpha ! i)$  and finite H
    and  $f i ' \text{nsets } H \ n \subseteq \{i\}$ 
  using assms by (auto simp: partn-lst-def monochromatic-def nsets-def)
  then obtain bij where bij: bij-betw bij H  $\{0.. $\alpha' ! i\}$ 
  by (metis ex-bij-betw-finite-nat)
  then have inj: inj-on (inv-into H bij)  $\{0.. $\alpha' ! i\}$ 
  by (metis bij-betw-def dual-order.refl i inj-on-inv-into invl-subset le)
  define H' where  $H' = \text{inv-into } H \text{ bij } ' \{0.. $\alpha' ! i\}$ 
  show  $\exists i < \text{length } \alpha'. \text{monochromatic } \beta \ (\alpha' ! i) \ n \ f \ i$ 
    unfolding monochromatic-def
  proof (intro exI bexI conjI)
    show  $i < \text{length } \alpha'$ 
    by (simp add:  $\text{assms}(2)$  i)
    have  $H' \subseteq H$$$$$ 
```

```

    using bij  $\langle i < \text{length } \alpha \rangle$  bij-betw-imp-surj-on le
    by (force simp:  $H'$ -def image-subset-iff intro: inv-into-into)
  then have finite  $H'$ 
    by (simp add:  $\langle \text{finite } H \rangle$  finite-subset)
  with  $\langle H' \subseteq H \rangle$  have card $H'$ :  $\text{card } H' = (\alpha^! i)$ 
    unfolding  $H'$ -def by (simp add: inj card-image)
  show  $f^{\cdot} [H^!]^n \subseteq \{i\}$ 
    by (meson  $\langle H' \subseteq H \rangle$  dual-order.trans fi image-mono nsets-mono)
  show  $H' \in [\beta]^{(\alpha^! i)}$ 
    using  $\langle H \subseteq \beta \rangle \langle H' \subseteq H \rangle \langle \text{finite } H' \rangle$  card $H'$  nsets-def by fastforce
qed
qed

```

90.2 Finite versions of Ramsey’s theorem

To distinguish the finite and infinite ones, lower and upper case names are used (ramsey vs Ramsey).

90.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers

The Erds–Szekeres bound, essentially extracted from the proof

```

fun ES ::  $[nat, nat, nat] \Rightarrow nat$ 
  where ES 0 k l = max k l
    | ES (Suc r) k l =
      (if r=0 then k+l-1
       else if k=0  $\vee$  l=0 then 1 else Suc (ES r (ES (Suc r) (k-1) l) (ES (Suc
r) k (l-1))))

declare ES.simps [simp del]

lemma ES-0 [simp]: ES 0 k l = max k l
  using ES.simps(1) by blast

lemma ES-1 [simp]: ES 1 k l = k+l-1
  using ES.simps(2) [of 0 k l] by simp

lemma ES-2: ES 2 k l = (if k=0  $\vee$  l=0 then 1 else ES 2 (k-1) l + ES 2 k (l-1))
  unfolding numeral-2-eq-2
  by (smt (verit) ES.elims One-nat-def Suc-pred add-gr-0 neq0-conv nat.inject
zero-less-Suc)

```

The Erds–Szekeres upper bound

```

lemma ES2-choose: ES 2 k l = (k+l) choose k
proof (induct n  $\equiv$  k+l arbitrary: k l)
  case 0
  then show ?case
    by (auto simp: ES-2)

```

```

next
  case (Suc n)
  then have  $k > 0 \implies l > 0 \implies ES\ 2\ (k - 1)\ l + ES\ 2\ k\ (l - 1) = k + l$  choose  $k$ 
    using choose-reduce-nat by force
  then show ?case
    by (metis ES-2 Nat.add-0-right binomial-n-0 binomial-n-n gr0I)
qed

```

90.2.2 Trivial cases

Vacuous, since we are dealing with 0-sets!

```

lemma ramsey0:  $\exists N::nat. \text{partn-lst } \{..<N\} [q1, q2]\ 0$ 
  by (force simp: partn-lst-def monochromatic-def ex-in-conv less-Suc-eq nsets-eq-empty-iff)

```

Just the pigeon hole principle, since we are dealing with 1-sets

```

lemma ramsey1-explicit:  $\text{partn-lst } \{..<q0 + q1 - Suc\ 0\} [q0, q1]\ 1$ 
proof -
  have  $\exists i < Suc\ (Suc\ 0). \exists H \in nsets\ \{..<q0 + q1 - 1\} ([q0, q1] ! i). f\ ' nsets\ H\ 1$ 
 $\subseteq \{i\}$ 
    if  $f \in nsets\ \{..<q0 + q1 - 1\} (Suc\ 0) \rightarrow \{..<Suc\ (Suc\ 0)\}$  for  $f$ 
  proof -
    define A where  $A \equiv \lambda i. \{q. q < q0 + q1 - 1 \wedge f\ \{q\} = i\}$ 
    have  $A\ 0 \cup A\ 1 = \{..<q0 + q1 - 1\}$ 
      using that by (auto simp: A-def PiE-iff nsets-one lessThan-Suc-atMost
le-Suc-eq)
    moreover have  $A\ 0 \cap A\ 1 = \{\}$ 
      by (auto simp: A-def)
    ultimately have  $q0 + q1 \leq \text{card } (A\ 0) + \text{card } (A\ 1) + 1$ 
      by (metis card-Un-le card-lessThan le-diff-conv)
    then consider  $\text{card } (A\ 0) \geq q0 \mid \text{card } (A\ 1) \geq q1$ 
      by linarith
    then obtain i where  $i < Suc\ (Suc\ 0) \text{ card } (A\ i) \geq [q0, q1] ! i$ 
      by (metis One-nat-def lessI nth-Cons-0 nth-Cons-Suc zero-less-Suc)
    then obtain B where  $B \subseteq A\ i \text{ card } B = [q0, q1] ! i \text{ finite } B$ 
      by (meson obtain-subset-with-card-n)
    then have  $B \in nsets\ \{..<q0 + q1 - 1\} ([q0, q1] ! i) \wedge f\ ' nsets\ B\ (Suc\ 0) \subseteq$ 
 $\{i\}$ 
      by (auto simp: A-def nsets-def card-1-singleton-iff)
    then show ?thesis
      using  $\langle i < Suc\ (Suc\ 0) \rangle$  by auto
  qed
then show ?thesis
  by (simp add: partn-lst-def monochromatic-def)
qed

```

```

lemma ramsey1:  $\exists N::nat. \text{partn-lst } \{..<N\} [q0, q1]\ 1$ 
  using ramsey1-explicit by blast

```

90.2.3 Ramsey’s theorem with TWO colours and arbitrary exponents (hypergraph version)

lemma *ramsey-induction-step*:

fixes $p::nat$
assumes $p1$: *partn-lst* $\{..<p1\}$ $[q1-1, q2]$ (*Suc* r) **and** $p2$: *partn-lst* $\{..<p2\}$ $[q1, q2-1]$ (*Suc* r)
and p : *partn-lst* $\{..<p\}$ $[p1, p2]$ r
and $q1 > 0$ $q2 > 0$
shows *partn-lst* $\{..<Suc\ p\}$ $[q1, q2]$ (*Suc* r)
proof –
have $\exists i < Suc\ (Suc\ 0). \exists H \in nsets\ \{..p\}\ ([q1, q2] ! i). f \text{ ‘ } nsets\ H\ (Suc\ r) \subseteq \{i\}$
if $f: f \in nsets\ \{..p\}\ (Suc\ r) \rightarrow \{..<Suc\ (Suc\ 0)\}$ **for** f
proof –
define g **where** $g \equiv \lambda R. f\ (insert\ p\ R)$
have $f\ (insert\ p\ i) \in \{..<Suc\ (Suc\ 0)\}$ **if** $i \in nsets\ \{..<p\}\ r$ **for** i
using *that card-insert-if* **by** (*fastforce simp: nsets-def intro!: Pi-mem [OF f]*)
then have $g: g \in nsets\ \{..<p\}\ r \rightarrow \{..<Suc\ (Suc\ 0)\}$
by (*force simp: g-def PiE-iff*)
then obtain $i\ U$ **where** $i: i < Suc\ (Suc\ 0)$ **and** $gi: g \text{ ‘ } nsets\ U\ r \subseteq \{i\}$
and $U: U \in nsets\ \{..<p\}\ ([p1, p2] ! i)$
using p **by** (*auto simp: partn-lst-def monochromatic-def*)
then have $U_{sub}: U \subseteq \{..<p\}$
by (*auto simp: nsets-def*)
consider (*izero*) $i = 0$ | (*ione*) $i = Suc\ 0$
using i **by** *linarith*
then show *?thesis*
proof cases
case izero
then have $U \in nsets\ \{..<p\}\ p1$
using U **by** *simp*
then obtain u **where** $u: bij\ betw\ u\ \{..<p1\}\ U$
using *ex-bij-betw-nat-finite lessThan-atLeast0* **by** (*fastforce simp: nsets-def*)
have $u\text{-}nsets: u \text{ ‘ } X \in nsets\ \{..p\}\ n$ **if** $X \in nsets\ \{..<p1\}\ n$ **for** $X\ n$
proof –
have *inj-on* $u\ X$
using u *that bij-betw-imp-inj-on inj-on-subset* **by** (*force simp: nsets-def*)
then show *?thesis*
using $U_{sub}\ u$ *that bij-betwE*
by (*fastforce simp: nsets-def card-image*)
qed
define h **where** $h \equiv \lambda R. f\ (u \text{ ‘ } R)$
have $h \in nsets\ \{..<p1\}\ (Suc\ r) \rightarrow \{..<Suc\ (Suc\ 0)\}$
unfolding $h\text{-def}$ **using** $f\ u\text{-}nsets$ **by** *auto*
then obtain $j\ V$ **where** $j: j < Suc\ (Suc\ 0)$ **and** $hj: h \text{ ‘ } nsets\ V\ (Suc\ r) \subseteq \{j\}$
and $V: V \in nsets\ \{..<p1\}\ ([q1 - Suc\ 0, q2] ! j)$
using $p1$ **by** (*auto simp: partn-lst-def monochromatic-def*)
then have $V_{sub}: V \subseteq \{..<p1\}$
by (*auto simp: nsets-def*)
have *invinv-eq*: $u \text{ ‘ } inv\ into\ \{..<p1\}\ u \text{ ‘ } X = X$ **if** $X \subseteq u \text{ ‘ } \{..<p1\}$ **for** X

```

    by (simp add: image-inv-into-cancel that)
  let ?W = insert p (u ‘ V)
  consider (jzero) j = 0 | (jone) j = Suc 0
    using j by linarith
  then show ?thesis
  proof cases
    case jzero
    then have V ∈ nsets {..

```

```

    then show ?thesis
      using izero jzero hj Xim invu-nsets unfolding h-def
      by (fastforce simp: image-subset-iff)
  qed
  moreover have insert p (u ‘ V) ∈ nsets {..p} q1
    by (simp add: izero inq1)
  ultimately show ?thesis
    by (metis izero image-subsetI insertI1 nth-Cons-0 zero-less-Suc)
next
  case jone
  then have u ‘ V ∈ nsets {..p} q2
    using V u-nsets by auto
  moreover have f ‘ nsets (u ‘ V) (Suc r) ⊆ {j}
    using hj
    by (force simp: h-def image-subset-iff nsets-def subset-image-inj card-image
dest: finite-imageD)
  ultimately show ?thesis
    using jone not-less-eq by fastforce
  qed
next
  case ione
  then have U ∈ nsets {..

```

```

then have  $V \in \text{nsets } \{..<p2\} (q2 - \text{Suc } 0)$ 
  using  $V$  by simp
then have  $u \text{ ' } V \in \text{nsets } \{..<p\} (q2 - \text{Suc } 0)$ 
  using  $u\text{-nsets } [of - q2 - \text{Suc } 0] \text{ nsets-mono } [OF Vsub] Usub u$ 
  unfolding bij-betw-def nsets-def by blast
then have  $\text{inq1}: ?W \in \text{nsets } \{..p\} q2$ 
  unfolding nsets-def using  $\langle q2 > 0 \rangle \text{ card-insert-if}$  by fastforce
have  $\text{invu-nsets}: \text{inv-into } \{..<p2\} u \text{ ' } X \in \text{nsets } V r$ 
  if  $X \in \text{nsets } (u \text{ ' } V) r$  for  $X r$ 
proof -
  have  $X \subseteq u \text{ ' } V \wedge \text{finite } X \wedge \text{card } X = r$ 
    using nsets-def that by auto
  then have  $[\text{simp}]: \text{card } (\text{inv-into } \{..<p2\} u \text{ ' } X) = \text{card } X$ 
    by (meson Vsub bij-betw-def bij-betw-inv-into card-image image-mono
inj-on-subset u)
  show ?thesis
    using that u Vsub by (fastforce simp: nsets-def bij-betw-def)
qed
have  $f X = i$  if  $X: X \in \text{nsets } ?W (\text{Suc } r)$  for  $X$ 
proof (cases p ∈ X)
  case True
    then have  $Xp: X - \{p\} \in \text{nsets } (u \text{ ' } V) r$ 
      using  $X$  by (auto simp: nsets-def)
    moreover have  $u \text{ ' } V \subseteq U$ 
      using  $Vsub \text{ bij-betwE } u$  by blast
    ultimately have  $X - \{p\} \in \text{nsets } U r$ 
      by (meson in-mono nsets-mono)
    then have  $g (X - \{p\}) = i$ 
      using  $gi$  by blast
    have  $f X = i$ 
      using  $gi \text{ True } \langle X - \{p\} \in \text{nsets } U r \rangle \text{ insert-Diff}$ 
      by (fastforce simp: g-def image-subset-iff)
    then show ?thesis
      by (simp add: f X = i g (X - {p}) = i)
  next
    case False
    then have  $Xim: X \in \text{nsets } (u \text{ ' } V) (\text{Suc } r)$ 
      using  $X$  by (auto simp: nsets-def subset-insert)
    then have  $u \text{ ' } \text{inv-into } \{..<p2\} u \text{ ' } X = X$ 
      using  $Vsub \text{ bij-betw-imp-inj-on } u$ 
      by (fastforce simp: nsets-def image-mono invinv-eq subset-trans)
    then show ?thesis
      using ione jone hj Xim invu-nsets unfolding h-def
      by (fastforce simp: image-subset-iff)
qed
moreover have  $\text{insert } p (u \text{ ' } V) \in \text{nsets } \{..p\} q2$ 
  by (simp add: ione inq1)
ultimately show ?thesis
  by (metis ione image-subsetI insertI1 lessI nth-Cons-0 nth-Cons-Suc)

```

```

next
  case jzero
  then have u ' V ∈ nsets {..p} q1
    using V u-nsets by auto
  moreover have f ' nsets (u ' V) (Suc r) ⊆ {j}
    using hj unfolding h-def image-subset-iff nsets-def
    apply (clarsimp simp add: h-def image-subset-iff nsets-def)
    by (metis card-image finite-imageD subset-image-inj)
  ultimately show ?thesis
    using jzero not-less-eq by fastforce
qed
qed
qed
then show ?thesis
  using lessThan-Suc lessThan-Suc-atMost
  by (auto simp: partn-lst-def monochromatic-def insert-commute)
qed

proposition ramsey2-full: partn-lst {.. $ES$  r q1 q2} [q1,q2] r
proof (induction r arbitrary: q1 q2)
  case 0
  then show ?case
    by (auto simp: partn-lst-def monochromatic-def less-Suc-eq ex-in-conv nsets-eq-empty-iff)
next
  case (Suc r)
  note outer = this
  show ?case
  proof (cases r = 0)
    case True
    then show ?thesis
      using ramsey1-explicit by (force simp: ES.simps)
  next
    case False
    then have r > 0
      by simp
    show ?thesis
      using Suc.prem
  proof (induct k ≡ q1 + q2 arbitrary: q1 q2)
    case 0
    with partn-lst-0 show ?case by auto
  next
    case (Suc k)
    consider q1 = 0 ∨ q2 = 0 | q1 ≠ 0 q2 ≠ 0 by auto
    then show ?case
  proof cases
    case 1
    with False partn-lst-0 partn-lst-0' show ?thesis
      by blast
  next

```



```

define  $p1$  where  $p1 \equiv ES (Suc\ r) (q1-1)\ q2$ 
define  $p2$  where  $p2 \equiv ES (Suc\ r)\ q1\ (q2-1)$ 
define  $p$  where  $p \equiv ES\ r\ p1\ p2$ 
case 2
with  $Suc$  have  $k = (q1-1) + q2\ k = q1 + (q2 - 1)$  by auto
then have  $p1$ :  $partn\text{-}lst\ \{..<p1\}\ [q1-1,q2]\ (Suc\ r)$ 
      and  $p2$ :  $partn\text{-}lst\ \{..<p2\}\ [q1,q2-1]\ (Suc\ r)$ 
      using  $Suc.hyps$  unfolding  $p1\text{-}def\ p2\text{-}def$  by blast+
then have  $p$ :  $partn\text{-}lst\ \{..<p\}\ [p1,p2]\ r$ 
      using outer Suc.prems unfolding  $p\text{-}def$  by auto
show ?thesis
      using ramsey-induction-step  $[OF\ p1\ p2\ p]\ 2\ ES.simps(2)\ False\ p1\text{-}def\ p2\text{-}def\ p\text{-}def$  by auto
qed
qed
qed
qed

```

90.2.4 Full Ramsey’s theorem with multiple colours and arbitrary exponents

```

theorem ramsey-full:  $\exists N::nat.\ partn\text{-}lst\ \{..<N\}\ qs\ r$ 
proof (induction k  $\equiv$  length qs arbitrary: qs)
  case 0
  then show ?case
    by (rule-tac x = r in exI) (simp add: partn-lst-def)
  next
  case ( $Suc\ k$ )
  note  $IH = this$ 
  show ?case
  proof (cases k)
    case 0
    with  $Suc$  obtain  $q$  where  $qs = [q]$ 
    by (metis length-0-conv length-Suc-conv)
    then show ?thesis
    by (rule-tac x = q in exI) (auto simp: partn-lst-def monochromatic-def func-set-to-empty-iff)
  next
  case ( $Suc\ k'$ )
  then obtain  $q1\ q2\ l$  where  $qs = q1 \# q2 \# l$ 
    by (metis Suc.hyps(2) length-Suc-conv)
  then obtain  $q::nat$  where  $q$ :  $partn\text{-}lst\ \{..<q\}\ [q1,q2]\ r$ 
    using ramsey2-full by blast
  then obtain  $p::nat$  where  $p$ :  $partn\text{-}lst\ \{..<p\}\ (q \# l)\ r$ 
    using  $IH\ \langle qs = q1 \# q2 \# l \rangle$  by fastforce
  have  $keq$ :  $Suc\ (length\ l) = k$ 
    using  $IH\ qs$  by auto
  show ?thesis
  proof (intro exI conjI)

```

```

show partn-lst  $\{..<p\}$  qs r
proof (auto simp: partn-lst-def)
  fix f
  assume f:  $f \in \text{nsets } \{..<p\} \text{ } r \rightarrow \{..<\text{length } \text{qs}\}$ 
  define g where  $g \equiv \lambda X. \text{if } f \text{ } X < \text{Suc } (\text{Suc } 0) \text{ then } 0 \text{ else } f \text{ } X - \text{Suc } 0$ 
  have  $g \in \text{nsets } \{..<p\} \text{ } r \rightarrow \{..<k\}$ 
    unfolding g-def using f Suc IH
    by (auto simp: Pi-def not-less)
  then obtain i U where  $i: i < k$  and  $g i: g \text{ ' } \text{nsets } U \text{ } r \subseteq \{i\}$ 
    and  $U: U \in \text{nsets } \{..<p\} \text{ } ((q\#l) ! i)$ 
    using p keq by (auto simp: partn-lst-def monochromatic-def)
  show  $\exists i < \text{length } \text{qs}. \text{monochromatic } \{..<p\} \text{ } (q\#l) i \text{ } r \text{ } f \text{ } i$ 
  proof (cases i = 0)
    case True
      then have  $U \in \text{nsets } \{..<p\} \text{ } q$  and  $f01: f \text{ ' } \text{nsets } U \text{ } r \subseteq \{0, \text{Suc } 0\}$ 
        using U gi unfolding g-def by (auto simp: image-subset-iff)
      then obtain u where  $u: \text{bij-betw } u \text{ } \{..<q\} \text{ } U$ 
        using ex-bij-betw-nat-finite lessThan-atLeast0 by (fastforce simp:
nsets-def)
      then have  $U_{\text{sub}}: U \subseteq \{..<p\}$ 
        by (smt (verit) U mem-Collect-eq nsets-def)
      have  $u\text{-nsets}: u \text{ ' } X \in \text{nsets } \{..<p\} \text{ } n \text{ if } X \in \text{nsets } \{..<q\} \text{ } n \text{ for } X \text{ } n$ 
        proof –
          have inj-on  $u \text{ } X$ 
            using u that bij-betw-imp-inj-on inj-on-subset
            by (force simp: nsets-def)
          then show ?thesis
            using  $U_{\text{sub}} u \text{ that } \text{bij-betwE}$ 
            by (fastforce simp: nsets-def card-image)
        qed
      define h where  $h \equiv \lambda X. f \text{ } (u \text{ ' } X)$ 
      have  $f \text{ } (u \text{ ' } X) < \text{Suc } (\text{Suc } 0) \text{ if } X \in \text{nsets } \{..<q\} \text{ } r \text{ for } X$ 
        proof –
          have  $u \text{ ' } X \in \text{nsets } U \text{ } r$ 
            using u u-nsets that by (auto simp: nsets-def bij-betwE subset-eq)
          then show ?thesis
            using f01 by auto
        qed
      then have  $h \in \text{nsets } \{..<q\} \text{ } r \rightarrow \{..<\text{Suc } (\text{Suc } 0)\}$ 
        unfolding h-def by blast
      then obtain j V where  $j: j < \text{Suc } (\text{Suc } 0)$  and  $h j: h \text{ ' } \text{nsets } V \text{ } r \subseteq \{j\}$ 
        and  $V: V \in \text{nsets } \{..<q\} \text{ } ([q1, q2] ! j)$ 
        using q by (auto simp: partn-lst-def monochromatic-def)
      show ?thesis
        unfolding monochromatic-def
      proof (intro exI conjI bexI)
        show  $j < \text{length } \text{qs}$ 
          using Suc Suc.hyps(2) j by linarith
        have  $\text{nsets } (u \text{ ' } V) \text{ } r \subseteq (\lambda x. (u \text{ ' } x)) \text{ ' } \text{nsets } V \text{ } r$ 

```

```

    apply (clarsimp simp add: nsets-def image-iff)
    by (metis card-image finite-imageD subset-image-inj)
  then have f ‘ nsets (u ‘ V) r ⊆ h ‘ nsets V r
    by (auto simp: h-def)
  then show f ‘ nsets (u ‘ V) r ⊆ {j}
    using hj by auto
  show (u ‘ V) ∈ nsets {..\bigwedge A. \llbracket A \in [U]^r \rrbracket \implies f A = \text{Suc } i
by (metis Suc-pred diff-0-eq-0 g-def gi image-subset-iff not-gr0 singletonD)
show ?thesis
  unfolding monochromatic-def
proof (intro exI conjI bexI)
  show  $\text{Suc } i < \text{length } qs$ 
    using Suc.hyps(2) i by auto
  show f ‘ nsets U r ⊆ {Suc i}
    using False by (auto simp: eq)
  show U ∈ nsets {..

```

90.2.5 Simple graph version

This is the most basic version in terms of cliques and independent sets, i.e. the version for graphs and 2 colours.

definition *clique* $V E \longleftrightarrow (\forall v \in V. \forall w \in V. v \neq w \longrightarrow \{v, w\} \in E)$

definition *indep* $V E \longleftrightarrow (\forall v \in V. \forall w \in V. v \neq w \longrightarrow \{v, w\} \notin E)$

lemma *clique-Un*: $\llbracket \text{clique } K F; \text{clique } L F; \forall v \in K. \forall w \in L. v \neq w \longrightarrow \{v, w\} \in F \rrbracket \implies \text{clique } (K \cup L) F$
 by (metis UnE clique-def doubleton-eq-iff)

lemma *null-clique[simp]*: $\text{clique } \{\} E$ **and** *null-indep[simp]*: $\text{indep } \{\} E$
 by (auto simp: clique-def indep-def)

lemma *smaller-clique*: $\llbracket \text{clique } R E; R' \subseteq R \rrbracket \implies \text{clique } R' E$
 by (auto simp: clique-def)

lemma *smaller-indep*: $\llbracket \text{indep } R E; R' \subseteq R \rrbracket \implies \text{indep } R' E$
 by (auto simp: indep-def)

lemma *ramsey2*:

$\exists r \geq 1. \forall (V :: 'a \text{ set}) (E :: 'a \text{ set set}). \text{finite } V \wedge \text{card } V \geq r \longrightarrow$
 $(\exists R \subseteq V. \text{card } R = m \wedge \text{clique } R \ E \vee \text{card } R = n \wedge \text{indep } R \ E)$

proof –

obtain N **where** $N \geq \text{Suc } 0$ **and** N : *partn-lst* $\{..<N\}$ $[m,n]$ 2
using *ramsey2-full nat-le-linear partn-lst-greater-resource* **by** *blast*
have $\exists R \subseteq V. \text{card } R = m \wedge \text{clique } R \ E \vee \text{card } R = n \wedge \text{indep } R \ E$
if *finite* V $N \leq \text{card } V$ **for** $V :: 'a \text{ set}$ **and** $E :: 'a \text{ set set}$

proof –

from *that*

obtain v **where** u : *inj-on* v $\{..<N\}$ v ‘ $\{..<N\} \subseteq V$

by (*metis card-le-inj card-lessThan finite-lessThan*)

define f **where** $f \equiv \lambda e. \text{if } v \text{ ‘ } e \in E \text{ then } 0 \text{ else } \text{Suc } 0$

have f : $f \in \text{nsets } \{..<N\}$ 2 $\rightarrow \{..<\text{Suc } (\text{Suc } 0)\}$

by (*simp add: f-def*)

then obtain i U **where** i : $i < 2$ **and** gi : f ‘ $\text{nsets } U$ 2 $\subseteq \{i\}$

and U : $U \in \text{nsets } \{..<N\}$ $([m,n] ! i)$

using N *numeral-2-eq-2* **by** (*auto simp: partn-lst-def monochromatic-def*)

show *?thesis*

proof (*intro exI conjI*)

show v ‘ $U \subseteq V$

using U u **by** (*auto simp: image-subset-iff nsets-def*)

show $\text{card } (v \text{ ‘ } U) = m \wedge \text{clique } (v \text{ ‘ } U) \ E \vee \text{card } (v \text{ ‘ } U) = n \wedge \text{indep } (v \text{ ‘ } U) \ E$

using i *unfolding numeral-2-eq-2*

using gi U u

unfolding *image-subset-iff nsets-2-eq clique-def indep-def less-Suc-eq*

by (*auto simp: f-def nsets-def card-image inj-on-subset split: if-splits*)

qed

qed

then show *?thesis*

using $\langle \text{Suc } 0 \leq N \rangle$ **by** *auto*

qed

90.3 Preliminaries for the infinitary version

90.3.1 “Axiom” of Dependent Choice

primrec *choice* :: $('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a) \text{ set} \Rightarrow \text{nat} \Rightarrow 'a$

where — An integer-indexed chain of choices

choice-0: $\text{choice } P \ r \ 0 = (\text{SOME } x. P \ x)$

| *choice-Suc*: $\text{choice } P \ r \ (\text{Suc } n) = (\text{SOME } y. P \ y \wedge (\text{choice } P \ r \ n, y) \in r)$

lemma *choice-n*:

assumes $P0$: $P \ x0$

and $P\text{step}$: $\bigwedge x. P \ x \Longrightarrow \exists y. P \ y \wedge (x, y) \in r$

shows $P \ (\text{choice } P \ r \ n)$

proof (*induct n*)

case 0

show *?case* **by** (*force intro: someI P0*)

```

next
  case Suc
  then show ?case by (auto intro: someI2-ex [OF Pstep])
qed

lemma dependent-choice:
  assumes trans: trans r
  and P0: P x0
  and Pstep:  $\bigwedge x. P\ x \implies \exists y. P\ y \wedge (x, y) \in r$ 
  obtains f :: nat  $\Rightarrow$  'a where  $\bigwedge n. P\ (f\ n)$  and  $\bigwedge n\ m. n < m \implies (f\ n, f\ m) \in r$ 
proof
  fix n
  show P (choice P r n)
    by (blast intro: choice-n [OF P0 Pstep])
next
  fix n m :: nat
  assume n < m
  from Pstep [OF choice-n [OF P0 Pstep]] have (choice P r k, choice P r (Suc
k))  $\in r$  for k
    by (auto intro: someI2-ex)
  then show (choice P r n, choice P r m)  $\in r$ 
    by (auto intro: less-Suc-induct [OF <n < m>] transD [OF trans])
qed

```

90.3.2 Partition functions

definition *part-fn* :: *nat* \Rightarrow *nat* \Rightarrow '*a* *set* \Rightarrow ('*a* *set* \Rightarrow *nat*) \Rightarrow *bool*
 — the function *f* partitions the *r*-subsets of the typically infinite set *Y* into *s* distinct categories.
 where *part-fn* *r* *s* *Y* *f* $\longleftrightarrow (f \in \text{nsets } Y\ r \rightarrow \{..<s\})$

For induction, we decrease the value of *r* in partitions.

lemma *part-fn-Suc-imp-part-fn*:
 $\llbracket \text{infinite } Y; \text{part-fn } (Suc\ r)\ s\ Y\ f; y \in Y \rrbracket \implies \text{part-fn } r\ s\ (Y - \{y\})\ (\lambda u. f\ (\text{insert } y\ u))$
 by (*simp* add: *part-fn-def* *nsets-def* *Pi-def* *subset-Diff-insert*)

lemma *part-fn-subset*: *part-fn* *r* *s* *YY* *f* $\implies Y \subseteq YY \implies \text{part-fn } r\ s\ Y\ f$
 unfolding *part-fn-def* *nsets-def* by *blast*

90.4 Ramsey’s Theorem: Infinitary Version

lemma *Ramsey-induction*:
 fixes *s* *r* :: *nat*
 and *YY* :: '*a* *set*
 and *f* :: '*a* *set* \Rightarrow *nat*
 assumes *infinite* *YY* *part-fn* *r* *s* *YY* *f*
 shows $\exists Y'\ t'. Y' \subseteq YY \wedge \text{infinite } Y' \wedge t' < s \wedge (\forall X. X \subseteq Y' \wedge \text{finite } X \wedge \text{card } X = r \longrightarrow f\ X = t')$
 using *assms*

```

proof (induct r arbitrary: YY f)
  case 0
  then show ?case
    by (auto simp add: part-fn-def card-eq-0-iff cong: conj-cong)
next
  case (Suc r)
  show ?case
  proof –
    from Suc.premis infinite-imp-nonempty obtain yy where yy: yy ∈ YY
    by blast
    let ?ramr = {((y, Y, t), (y', Y', t')). y' ∈ Y ∧ Y' ⊆ Y}
    let ?propr = λ(y, Y, t).
      y ∈ YY ∧ y ∉ Y ∧ Y ⊆ YY ∧ infinite Y ∧ t < s
      ∧ (∀ X. X ⊆ Y ∧ finite X ∧ card X = r ⟶ (f ∘ insert y) X = t)
    from Suc.premis have infYY': infinite (YY − {yy}) by auto
    from Suc.premis have partf': part-fn r s (YY − {yy}) (f ∘ insert yy)
    by (simp add: o-def part-fn-Suc-imp-part-fn yy)
    have transr: trans ?ramr by (force simp: trans-def)
    from Suc.hyps [OF infYY' partf']
    obtain Y0 and t0 where Y0 ⊆ YY − {yy} infinite Y0 t0 < s
      X ⊆ Y0 ∧ finite X ∧ card X = r ⟶ (f ∘ insert yy) X = t0 for X
    by blast
    with yy have propr0: ?propr(yy, Y0, t0) by blast
    have proprstep: ∃ y. ?propr y ∧ (x, y) ∈ ?ramr if x: ?propr x for x
    proof (cases x)
      case (fields yx Yx tx)
      with x obtain yx' where yx': yx' ∈ Yx
      by (blast dest: infinite-imp-nonempty)
      from fields x have infYx': infinite (Yx − {yx'}) by auto
      with fields x yx' Suc.premis have partfx': part-fn r s (Yx − {yx'}) (f ∘ insert
yx')
      by (simp add: o-def part-fn-Suc-imp-part-fn part-fn-subset [where YY=YY
and Y=Yx])
      from Suc.hyps [OF infYx' partfx'] obtain Y' and t'
      where Y': Y' ⊆ Yx − {yx'} infinite Y' t' < s
      X ⊆ Y' ∧ finite X ∧ card X = r ⟶ (f ∘ insert yx') X = t' for X
      by blast
      from fields x Y' yx' have ?propr (yx', Y', t') ∧ (x, (yx', Y', t')) ∈ ?ramr
      by blast
    then show ?thesis ..
  qed
  from dependent-choice [OF transr propr0 proprstep]
  obtain g where pg: ?propr (g n) and rg: n < m ⟶ (g n, g m) ∈ ?ramr for
n m :: nat
  by blast
  let ?gy = fst ∘ g
  let ?gt = snd ∘ snd ∘ g
  have rangeg: ∃ k. range ?gt ⊆ {..proof (intro exI subsetI)

```

```

fix  $x$ 
assume  $x \in \text{range } ?gt$ 
then obtain  $n$  where  $x = ?gt\ n \ ..$ 
with  $pg\ [of\ n]$  show  $x \in \{..<s\}$  by (cases g n) auto
qed
from range g have finite (range ?gt)
by (simp add: finite-nat-iff-bounded)
then obtain  $s'$  and  $n'$  where  $s': s' = ?gt\ n'$  and infeqs': infinite  $\{n. ?gt\ n = s'\}$ 
by (rule inf-img-fin-domE) (auto simp add: vimage-def intro: infinite-UNIV-nat)
with  $pg\ [of\ n']$  have less':  $s' < s$  by (cases g n') auto
have inj-gy: inj ?gy
proof (rule linorder-injI)
  fix  $m\ m' :: \text{nat}$ 
  assume  $m < m'$ 
  from rg [OF this] pg [of m] show  $?gy\ m \neq ?gy\ m'$ 
  by (cases g m, cases g m') auto
qed
show ?thesis
proof (intro exI conjI)
  from  $pg$  show  $?gy\ ' \{n. ?gt\ n = s'\} \subseteq YY$ 
  by (auto simp add: Let-def split-beta)
  from infeqs' show infinite  $(?gy\ ' \{n. ?gt\ n = s'\})$ 
  by (blast intro: inj-gy [THEN inj-on-subset] dest: finite-imageD)
  show  $s' < s$  by (rule less')
  show  $\forall X. X \subseteq ?gy\ ' \{n. ?gt\ n = s'\} \wedge \text{finite } X \wedge \text{card } X = \text{Suc } r \longrightarrow f\ X$ 
   $= s'$ 
  proof –
    have  $f\ X = s'$ 
    if  $X: X \subseteq ?gy\ ' \{n. ?gt\ n = s'\}$ 
    and cardX: finite X card X = Suc r
    for  $X$ 
  proof –
    from  $X$  obtain  $AA$  where  $AA: AA \subseteq \{n. ?gt\ n = s'\}$  and Xeq:  $X = ?gy\ ' AA$ 
    by (auto simp add: subset-image-iff)
    with cardX have  $AA \neq \{\}$  by auto
    then have AAleast: (LEAST x. x ∈ AA)  $\in AA$  by (auto intro: LeastI-ex)
    show ?thesis
    proof (cases g (LEAST x. x ∈ AA))
      case (fields ya Ya ta)
        with AAleast Xeq have  $ya: ya \in X$  by (force intro!: rev-image-eqI)
        then have  $f\ X = f\ (\text{insert } ya\ (X - \{ya\}))$  by (simp add: insert-absorb)
        also have  $\dots = ta$ 
        proof –
          have  $*$ :  $X - \{ya\} \subseteq Ya$ 
          proof
            fix  $x$  assume  $x: x \in X - \{ya\}$ 
            then obtain  $a'$  where xeq:  $x = ?gy\ a'$  and  $a': a' \in AA$ 

```

```

      by (auto simp add: Xeq)
    with fields x have  $a' \neq (LEAST x. x \in AA)$  by auto
  with Least-le [of  $\lambda x. x \in AA$ , OF a'] have  $(LEAST x. x \in AA) < a'$ 
    by arith
  from xeq fields rg [OF this] show  $x \in Ya$  by auto
qed
have  $card (X - \{ya\}) = r$ 
  by (simp add: cardX ya)
with pg [of  $LEAST x. x \in AA$ ] fields cardX * show ?thesis
  by (auto simp del: insert-Diff-single)
qed
also from AA AAleast fields have  $\dots = s'$  by auto
finally show ?thesis .
qed
qed
then show ?thesis by blast
qed
qed
qed
qed

```

theorem Ramsey:

```

  fixes s r :: nat
  and Z :: 'a set
  and f :: 'a set  $\Rightarrow$  nat
  shows
    [[infinite Z;
       $\forall X. X \subseteq Z \wedge finite X \wedge card X = r \longrightarrow f X < s$ ]
     $\implies \exists Y t. Y \subseteq Z \wedge infinite Y \wedge t < s$ 
       $\wedge (\forall X. X \subseteq Y \wedge finite X \wedge card X = r \longrightarrow f X = t)$ ]
  by (blast intro: Ramsey-induction [unfolded part-fn-def nsets-def])

```

corollary Ramsey2:

```

  fixes s :: nat
  and Z :: 'a set
  and f :: 'a set  $\Rightarrow$  nat
  assumes infZ: infinite Z
  and part:  $\forall x \in Z. \forall y \in Z. x \neq y \longrightarrow f \{x, y\} < s$ 
  shows  $\exists Y t. Y \subseteq Z \wedge infinite Y \wedge t < s \wedge (\forall x \in Y. \forall y \in Y. x \neq y \longrightarrow f \{x, y\} = t)$ 
  proof -
    from part have part2:  $\forall X. X \subseteq Z \wedge finite X \wedge card X = 2 \longrightarrow f X < s$ 
      by (fastforce simp: eval-nat-numeral card-Suc-eq)
    obtain Y t where *:
       $Y \subseteq Z \wedge infinite Y \wedge t < s \wedge (\forall X. X \subseteq Y \wedge finite X \wedge card X = 2 \longrightarrow f X = t)$ 
      by (insert Ramsey [OF infZ part2]) auto
    then have  $\forall x \in Y. \forall y \in Y. x \neq y \longrightarrow f \{x, y\} = t$  by auto
    with * show ?thesis by iprover
  qed

```


qed

corollary *Ramsey-nsets*:

fixes $f :: 'a \text{ set} \Rightarrow \text{nat}$
 assumes $\text{infinite } Z \text{ f ' nsets } Z \text{ r} \subseteq \{..<s\}$
 obtains $Y \text{ t where } Y \subseteq Z \text{ infinite } Y \text{ t} < s \text{ f ' nsets } Y \text{ r} \subseteq \{t\}$
 using *Ramsey* [of $Z \text{ r f s}$] *assms* **by** (auto simp: *nsets-def image-subset-iff*)

90.5 Disjunctive Well-Foundedness

An application of Ramsey’s theorem to program termination. See [4].

definition *disj-wf* :: $('a \times 'a) \text{ set} \Rightarrow \text{bool}$

where $\text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i<n. \text{wf } (T \text{ i})) \wedge r = (\bigcup i<n. T \text{ i}))$

definition *transition-idx* :: $(\text{nat} \Rightarrow 'a) \Rightarrow (\text{nat} \Rightarrow ('a \times 'a) \text{ set}) \Rightarrow \text{nat set} \Rightarrow \text{nat}$

where $\text{transition-idx } s \text{ T } A = (\text{LEAST } k. \exists i \text{ j. } A = \{i, j\} \wedge i < j \wedge (s \text{ j}, s \text{ i}) \in T \text{ k})$

lemma *transition-idx-less*:

assumes $i < j \text{ (s j, s i)} \in T \text{ k } k < n$
 shows $\text{transition-idx } s \text{ T } \{i, j\} < n$

proof –

from *assms*(1,2) **have** $\text{transition-idx } s \text{ T } \{i, j\} \leq k$
 by (simp add: *transition-idx-def*, blast intro: *Least-le*)
 with *assms*(3) **show** *?thesis* **by** simp

qed

lemma *transition-idx-in*:

assumes $i < j \text{ (s j, s i)} \in T \text{ k}$
 shows $(s \text{ j}, s \text{ i}) \in T \text{ (transition-idx } s \text{ T } \{i, j\})$
 using *assms*
 by (simp add: *transition-idx-def doubleton-eq-iff conj-disj-distribR cong: conj-cong*)
 (erule *LeastI*)

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf*: $\text{disj-wf } r \longleftrightarrow (\exists T. \exists n::\text{nat}. (\forall i<n. \text{wf}(T \text{ i})) \wedge r \subseteq (\bigcup i<n. T \text{ i}))$

proof –

have *: $\bigwedge T \text{ n. } [\forall i<n. \text{wf } (T \text{ i}); r \subseteq \bigcup (T \text{ i} \text{ ' } \{..<n\})]$
 $\implies (\forall i<n. \text{wf } (T \text{ i} \cap r)) \wedge r = (\bigcup i<n. T \text{ i} \cap r)$

by (force simp: *wf-Int1*)

show *?thesis*

unfolding *disj-wf-def* **by** auto (*metis* *)

qed

theorem *trans-disj-wf-implies-wf*:

assumes *trans* r

```

    and disj-wf r
  shows wf r
proof (simp only: wf-iff-no-infinite-down-chain, rule notI)
  assume  $\exists s. \forall i. (s (Suc\ i), s\ i) \in r$ 
  then obtain s where sSuc:  $\forall i. (s (Suc\ i), s\ i) \in r$  ..
  from  $\langle disj\text{-}wf\ r \rangle$  obtain T and n :: nat where wfT:  $\forall k < n. wf(T\ k)$  and r: r
= ( $\bigcup_{k < n. T\ k}$ )
  by (auto simp add: disj-wf-def)
  have s-in-T:  $\exists k. (s\ j, s\ i) \in T\ k \wedge k < n$  if i < j for i j
proof -
  from  $\langle i < j \rangle$  have  $(s\ j, s\ i) \in r$ 
  proof (induct rule: less-Suc-induct)
    case 1
    then show ?case by (simp add: sSuc)
  next
    case 2
    with  $\langle trans\ r \rangle$  show ?case
    unfolding trans-def by blast
  qed
  then show ?thesis by (auto simp add: r)
qed
have  $i < j \implies transition\text{-}idx\ s\ T\ \{i, j\} < n$  for i j
  using s-in-T transition-idx-less by blast
then have trless:  $i \neq j \implies transition\text{-}idx\ s\ T\ \{i, j\} < n$  for i j
  by (metis doubleton-eq-iff less-linear)
have  $\exists K\ k. K \subseteq UNIV \wedge infinite\ K \wedge k < n \wedge$ 
  ( $\forall i \in K. \forall j \in K. i \neq j \longrightarrow transition\text{-}idx\ s\ T\ \{i, j\} = k$ )
  by (rule Ramsey2) (auto intro: trless infinite-UNIV-nat)
then obtain K and k where infK: infinite K and k < n
  and allk:  $\forall i \in K. \forall j \in K. i \neq j \longrightarrow transition\text{-}idx\ s\ T\ \{i, j\} = k$ 
  by auto
have  $(s\ (enumerate\ K\ (Suc\ m)), s\ (enumerate\ K\ m)) \in T\ k$  for m :: nat
proof -
  let ?j = enumerate K (Suc m)
  let ?i = enumerate K m
  have ij: ?i < ?j by (simp add: enumerate-step infK)
  have ?j ∈ K ?i ∈ K by (simp-all add: enumerate-in-set infK)
  with ij have k: k = transition-idx s T {?i, ?j} by (simp add: allk)
  from s-in-T [OF ij] obtain k' where  $(s\ ?j, s\ ?i) \in T\ k' \wedge k' < n$  by blast
  then show  $(s\ ?j, s\ ?i) \in T\ k$  by (simp add: k transition-idx-in ij)
qed
then have  $\neg wf\ (T\ k)$ 
  unfolding wf-iff-no-infinite-down-chain by iprover
with wfT  $\langle k < n \rangle$  show False by blast
qed
end

```

91 Modulo and congruence on the reals

```
theory Real-Mod
  imports Complex-Main
begin
```

definition $rmod :: real \Rightarrow real \Rightarrow real$ (**infixl** $\langle rmod \rangle$ 70) **where**
 $x \ rmod \ y = x - |y| * of_int \lfloor x / |y| \rfloor$

lemma $rmod_conv_frac$: $y \neq 0 \implies x \ rmod \ y = frac \ (x / |y|) * |y|$
by (*simp add: rmod-def frac-def algebra-simps*)

lemma $rmod_conv_frac'$: $x \ rmod \ y = (if \ y = 0 \ then \ x \ else \ frac \ (x / |y|) * |y|)$
by (*simp add: rmod-def frac-def algebra-simps*)

lemma $rmod_rmod$ [*simp*]: $(x \ rmod \ y) \ rmod \ y = x \ rmod \ y$
by (*simp add: rmod-conv-fac'*)

lemma $rmod_0_right$ [*simp*]: $x \ rmod \ 0 = x$
by (*simp add: rmod-def*)

lemma $rmod_less$: $m > 0 \implies x \ rmod \ m < m$
by (*simp add: rmod-conv-fac' frac-lt-1*)

lemma $rmod_less_abs$: $m \neq 0 \implies x \ rmod \ m < |m|$
by (*simp add: rmod-conv-fac' frac-lt-1*)

lemma $rmod_le$: $m > 0 \implies x \ rmod \ m \leq m$
by (*intro less-imp-le rmod-less*)

lemma $rmod_nonneg$: $m \neq 0 \implies x \ rmod \ m \geq 0$
unfolding $rmod_def$
by (*metis abs-le-zero-iff diff-ge-0-iff-ge floor-divide-lower linorder-not-le mult.commute*)

lemma $rmod_unique$:
assumes $z \in \{0..<|y|\}$ $x = z + of_int \ n * y$
shows $x \ rmod \ y = z$

proof –

have $(x - z) / y = of_int \ n$

using *assms* **by** *auto*

hence $(x - z) / |y| = of_int \ ((if \ y > 0 \ then \ 1 \ else \ -1) * n)$

using *assms*(1) **by** (*cases y 0 :: real rule: linorder-cases*) (*auto split: if-splits*)

also have $\dots \in \mathbb{Z}$

by *auto*

finally have $frac \ (x / |y|) = z / |y|$

using *assms*(1) **by** (*subst frac-unique-iff*) (*auto simp: field-simps*)

thus *?thesis*

using *assms*(1) **by** (*auto simp: rmod-conv-frac'*)
qed

lemma *rmod-0* [*simp*]: $0 \text{ rmod } z = 0$
by (*simp add: rmod-def*)

lemma *rmod-add*: $(x \text{ rmod } z + y \text{ rmod } z) \text{ rmod } z = (x + y) \text{ rmod } z$
proof (*cases z = 0*)
case [*simp*]: *False*
show ?thesis
proof (*rule sym, rule rmod-unique*)
define *n* **where** $n = (\text{if } z > 0 \text{ then } 1 \text{ else } -1) * (\lfloor x / |z| \rfloor + \lfloor y / |z| \rfloor +$
 $\lfloor (x + y - (|z| * \text{real-of-int } \lfloor x / |z| \rfloor + |z| * \text{real-of-int } \lfloor y / |z| \rfloor)) / |z| \rfloor)$
show $x + y = (x \text{ rmod } z + y \text{ rmod } z) \text{ rmod } z + \text{real-of-int } n * z$
by (*simp add: rmod-def algebra-simps n-def*)
qed (*auto simp: rmod-less-abs rmod-nonneg*)
qed *auto*

lemma *rmod-diff*: $(x \text{ rmod } z - y \text{ rmod } z) \text{ rmod } z = (x - y) \text{ rmod } z$
proof (*cases z = 0*)
case [*simp*]: *False*
show ?thesis
proof (*rule sym, rule rmod-unique*)
define *n* **where** $n = (\text{if } z > 0 \text{ then } 1 \text{ else } -1) * (\lfloor x / |z| \rfloor +$
 $\lfloor (x + |z| * \text{real-of-int } \lfloor y / |z| \rfloor - (y + |z| * \text{real-of-int } \lfloor x / |z| \rfloor)) / |z| \rfloor - \lfloor y$
 $/ |z| \rfloor)$
show $x - y = (x \text{ rmod } z - y \text{ rmod } z) \text{ rmod } z + \text{real-of-int } n * z$
by (*simp add: rmod-def algebra-simps n-def*)
qed (*auto simp: rmod-less-abs rmod-nonneg*)
qed *auto*

lemma *rmod-self* [*simp*]: $x \text{ rmod } x = 0$
by (*cases x 0 :: real rule: linorder-cases*) (*auto simp: rmod-conv-frac*)

lemma *rmod-self-multiple-int* [*simp*]: $(\text{of-int } n * x) \text{ rmod } x = 0$
by (*cases x 0 :: real rule: linorder-cases*) (*auto simp: rmod-conv-frac*)

lemma *rmod-self-multiple-nat* [*simp*]: $(\text{of-nat } n * x) \text{ rmod } x = 0$
by (*cases x 0 :: real rule: linorder-cases*) (*auto simp: rmod-conv-frac*)

lemma *rmod-self-multiple-numeral* [*simp*]: $(\text{numeral } n * x) \text{ rmod } x = 0$
by (*cases x 0 :: real rule: linorder-cases*) (*auto simp: rmod-conv-frac*)

lemma *rmod-self-multiple-int'* [*simp*]: $(x * \text{of-int } n) \text{ rmod } x = 0$
by (*cases x 0 :: real rule: linorder-cases*) (*auto simp: rmod-conv-frac*)

lemma *rmod-self-multiple-nat'* [*simp*]: $(x * \text{of-nat } n) \text{ rmod } x = 0$
by (*cases x 0 :: real rule: linorder-cases*) (*auto simp: rmod-conv-frac*)

lemma *rmod-self-multiple-numeral'* [simp]: $(x * \text{numeral } n) \text{ rmod } x = 0$
by (cases $x \ 0$:: real rule: linorder-cases) (auto simp: rmod-conv-frac)

lemma *rmod-idem* [simp]: $x \in \{0..<|y|\} \implies x \text{ rmod } y = x$
by (rule rmod-unique[of - - 0]) auto

definition *rcong* :: real \Rightarrow real \Rightarrow real \Rightarrow bool
 $(\langle (\langle \text{indent}=1 \text{ notation}=\text{mixfix } rcong \rangle [- = -] '(\text{' rmod -}')) \rangle)$
where $[x = y] \text{ (rmod } m) \longleftrightarrow x \text{ rmod } m = y \text{ rmod } m$

named-theorems *rcong-intros*

lemma *rcong-0-right* [simp]: $[x = y] \text{ (rmod } 0) \longleftrightarrow x = y$
by (simp add: rcong-def)

lemma *rcong-0-iff*: $[x = 0] \text{ (rmod } m) \longleftrightarrow x \text{ rmod } m = 0$
and *rcong-0-iff'*: $[0 = x] \text{ (rmod } m) \longleftrightarrow x \text{ rmod } m = 0$
by (simp-all add: rcong-def)

lemma *rcong-refl* [simp, intro!, rcong-intros]: $[x = x] \text{ (rmod } m)$
by (simp add: rcong-def)

lemma *rcong-sym*: $[y = x] \text{ (rmod } m) \implies [x = y] \text{ (rmod } m)$
by (simp add: rcong-def)

lemma *rcong-sym-iff*: $[y = x] \text{ (rmod } m) \longleftrightarrow [x = y] \text{ (rmod } m)$
unfolding *rcong-def* **by** (simp add: eq-commute del: rmod-idem)

lemma *rcong-trans* [trans]: $[x = y] \text{ (rmod } m) \implies [y = z] \text{ (rmod } m) \implies [x = z] \text{ (rmod } m)$
by (simp add: rcong-def)

lemma *rcong-add* [rcong-intros]:
 $[a = b] \text{ (rmod } m) \implies [c = d] \text{ (rmod } m) \implies [a + c = b + d] \text{ (rmod } m)$
unfolding *rcong-def* **using** *rmod-add* **by** metis

lemma *rcong-diff* [rcong-intros]:
 $[a = b] \text{ (rmod } m) \implies [c = d] \text{ (rmod } m) \implies [a - c = b - d] \text{ (rmod } m)$
unfolding *rcong-def* **using** *rmod-diff* **by** metis

lemma *rcong-uminus* [rcong-intros]:
 $[a = b] \text{ (rmod } m) \implies [-a = -b] \text{ (rmod } m)$
using *rcong-diff*[of 0 0 $m \ a \ b$] **by** simp

lemma *rcong-uminus-uminus-iff* [simp]: $[-x = -y] \text{ (rmod } m) \longleftrightarrow [x = y] \text{ (rmod } m)$
using *rcong-uminus minus-minus* **by** metis

lemma *rcong-uminus-left-iff*: $[-x = y] \text{ (rmod } m) \longleftrightarrow [x = -y] \text{ (rmod } m)$
using *rcong-uminus minus-minus by metis*

lemma *rcong-add-right-cancel [simp]*: $[a + c = b + c] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$
using *rcong-add[of a b m c c] rcong-add[of a + c b + c m -c -c] by auto*

lemma *rcong-add-left-cancel [simp]*: $[c + a = c + b] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$
by (*subst (1 2) add.commute simp*)

lemma *rcong-diff-right-cancel [simp]*: $[a - c = b - c] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$
by (*metis rcong-add-left-cancel uminus-add-conv-diff*)

lemma *rcong-diff-left-cancel [simp]*: $[c - a = c - b] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$
by (*metis minus-diff-eq rcong-diff-right-cancel rcong-uminus-uminus-iff*)

lemma *rcong-rmod-right-iff [simp]*: $[a = (b \text{ rmod } m)] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$
and *rcong-rmod-left-iff [simp]*: $[(a \text{ rmod } m) = b] \text{ (rmod } m) \longleftrightarrow [a = b] \text{ (rmod } m)$
by (*simp-all add: rcong-def*)

lemma *rcong-rmod-left [rcong-intros]*: $[a = b] \text{ (rmod } m) \implies [(a \text{ rmod } m) = b] \text{ (rmod } m)$
and *rcong-rmod-right [rcong-intros]*: $[a = b] \text{ (rmod } m) \implies [a = (b \text{ rmod } m)] \text{ (rmod } m)$
by *simp-all*

lemma *rcong-mult-of-int-0-left-left [rcong-intros]*: $[0 = \text{of-int } n * m] \text{ (rmod } m)$
and *rcong-mult-of-int-0-right-left [rcong-intros]*: $[0 = m * \text{of-int } n] \text{ (rmod } m)$
and *rcong-mult-of-int-0-left-right [rcong-intros]*: $[\text{of-int } n * m = 0] \text{ (rmod } m)$
and *rcong-mult-of-int-0-right-right [rcong-intros]*: $[m * \text{of-int } n = 0] \text{ (rmod } m)$
by (*simp-all add: rcong-def*)

lemma *rcong-altdef*: $[a = b] \text{ (rmod } m) \longleftrightarrow (\exists n. b = a + \text{of-int } n * m)$

proof (*cases m = 0*)

case *False*

show *?thesis*

proof

assume $[a = b] \text{ (rmod } m)$

hence $[a - b = b - b] \text{ (rmod } m)$

by (*intro rcong-intros*)

hence $(a - b) \text{ rmod } m = 0$

by (*simp add: rcong-def*)

then obtain *n* **where** $\text{of-int } n = (a - b) / |m|$

```

    using False by (auto simp: rmod-conv-fraction elim!: Ints-cases)
  thus  $\exists n. b = a + \text{of-int } n * m$  using False
    by (intro exI[of - if  $m > 0$  then  $-n$  else  $n$ ]) (auto simp: field-simps)
next
  assume  $\exists n. b = a + \text{of-int } n * m$ 
  then obtain  $n$  where  $n: b = a + \text{of-int } n * m$ 
    by auto
  have  $[a + 0 = a + \text{of-int } n * m] \text{ (rmod } m)$ 
    by (intro rcong-intros)
  with  $n$  show  $[a = b] \text{ (rmod } m)$ 
    by simp
qed
qed auto

```

lemma *rcong-conv-diff-rmod-eq-0*: $[x = y] \text{ (rmod } m) \longleftrightarrow (x - y) \text{ rmod } m = 0$
 by (*metis cancel-comm-monoid-add-class.diff-cancel rcong-0-iff rcong-diff-right-cancel*)

lemma *rcong-imp-eq*:
 assumes $[x = y] \text{ (rmod } m)$ $|x - y| < |m|$
 shows $x = y$
proof –
 from *assms* obtain n where $n: y = x + \text{of-int } n * m$
 unfolding *rcong-altdef* by blast
 have $\text{of-int } |n| * |m| = |x - y|$
 by (*simp add: n abs-mult*)
 also have $\dots < 1 * |m|$
 using *assms*(2) by simp
 finally have $n = 0$
 by (*subst (asm) mult-less-cancel-right*) auto
 with n show ?thesis
 by simp
qed

lemma *rcong-mult-modulus*:
 assumes $[a = b] \text{ (rmod } (m / c))$ $c \neq 0$
 shows $[a * c = b * c] \text{ (rmod } m)$
proof –
 from *assms* obtain k where $k: b = a + \text{of-int } k * (m / c)$
 by (*auto simp: rcong-altdef*)
 have $b * c = (a + \text{of-int } k * (m / c)) * c$
 by (*simp only: k*)
 also have $\dots = a * c + \text{of-int } k * m$
 using *assms*(2) by (*auto simp: divide-simps*)
 finally show ?thesis
 unfolding *rcong-altdef* by blast
qed

lemma *rcong-divide-modulus*:
 assumes $[a = b] \text{ (rmod } (m * c))$ $c \neq 0$

shows $[a / c = b / c] (rmod\ m)$
using *rcong-mult-modulus*[of $a\ b\ m\ 1 / c$] *assms* **by** (*auto simp: field-simps*)

lemma *sin-rmod* [*simp*]: $\sin (x\ rmod\ (2 * \pi)) = \sin x$
and *cos-rmod* [*simp*]: $\cos (x\ rmod\ (2 * \pi)) = \cos x$
by (*simp-all add: rmod-def sin-diff cos-diff*)

lemma *tan-rmod* [*simp*]: $\tan (x\ rmod\ (2 * \pi)) = \tan x$
and *cot-rmod* [*simp*]: $\cot (x\ rmod\ (2 * \pi)) = \cot x$
and *cis-rmod* [*simp*]: $\text{cis} (x\ rmod\ (2 * \pi)) = \text{cis} x$
and *rcis-rmod* [*simp*]: $\text{rcis}\ r\ (x\ rmod\ (2 * \pi)) = \text{rcis}\ r\ x$
by (*simp-all add: tan-def cot-def complex-eq-iff*)

lemma *cis-eq-iff*: $\text{cis}\ a = \text{cis}\ b \longleftrightarrow [a = b] (rmod\ (2 * \pi))$

proof –

have $\text{cis}\ a = \text{cis}\ b \longleftrightarrow \sin a = \sin b \wedge \cos a = \cos b$
by (*auto simp: complex-eq-iff*)
also have $\dots \longleftrightarrow (\exists x. a = b + 2 * \pi * \text{real-of-int}\ x)$
by (*rule sin-cos-eq-iff*)
also have $\dots \longleftrightarrow [b = a] (rmod\ (2 * \pi))$
by (*simp add: rcong-altdef mult-ac*)
finally show *?thesis*
by (*simp add: rcong-sym-iff*)

qed

lemma *cis-eq-1-iff*: $\text{cis}\ a = 1 \longleftrightarrow (\exists n. a = \text{of-int}\ n * (2 * \pi))$

proof –

have $\text{cis}\ 0 = \text{cis}\ a \longleftrightarrow [0 = a] (rmod\ (2 * \pi))$
by (*rule cis-eq-iff*)
also have $\dots \longleftrightarrow (\exists n. a = \text{of-int}\ n * (2 * \pi))$
unfolding *rcong-altdef* **by** *simp*
finally show *?thesis*
by *simp*

qed

lemma *cis-cong*:

assumes $[a = b] (rmod\ 2 * \pi)$
shows $\text{cis}\ a = \text{cis}\ b$
using *cis-eq-iff* *assms* **by** *blast*

lemma *rcis-cong*:

assumes $[a = b] (rmod\ 2 * \pi)$
shows $\text{rcis}\ r\ a = \text{rcis}\ r\ b$
using *assms* **by** (*auto simp: rcis-def intro!: cis-cong*)

lemma *sin-rcong*: $[x = y] (rmod\ (2 * \pi)) \implies \sin x = \sin y$

and *cos-rcong*: $[x = y] (rmod\ (2 * \pi)) \implies \cos x = \cos y$
using *cis-cong*[of $x\ y$] **by** (*simp-all add: complex-eq-iff*)


```

lemma sin-eq-sin-conv-rmod:
  assumes  $\sin x = \sin y$ 
  shows  $[x = y] \text{ (rmod } 2 * \pi) \vee [x = \pi - y] \text{ (rmod } 2 * \pi)$ 
proof –
  have  $0 = \sin y - \sin x$ 
  using assms by simp
  hence  $\sin ((y - x) / 2) = 0 \vee \cos ((y + x) / 2) = 0$ 
  unfolding sin-diff-sin by simp
  hence  $(\exists i. y = x + \text{real-of-int } i * (2 * \pi)) \vee$ 
     $(\exists i. \pi - y = x + \text{real-of-int } (-i) * (2 * \pi))$ 
  unfolding sin-zero-iff-int2 cos-zero-iff-int2
  by (auto simp: algebra-simps)
  thus ?thesis
  unfolding rcong-altdef by blast
qed

```

```

lemma cos-eq-cos-conv-rmod:
  assumes  $\cos x = \cos y$ 
  shows  $[x = y] \text{ (rmod } 2 * \pi) \vee [x = -y] \text{ (rmod } 2 * \pi)$ 
proof –
  have  $0 = \cos y - \cos x$ 
  using assms by simp
  hence  $\sin ((y + x) / 2) = 0 \vee \sin ((x - y) / 2) = 0$ 
  unfolding cos-diff-cos by simp
  hence  $(\exists i. -y = x + \text{real-of-int } (-i) * (2 * \pi)) \vee$ 
     $(\exists i. y = x + \text{real-of-int } (-i) * (2 * \pi))$ 
  unfolding sin-zero-iff-int2
  by (auto simp: algebra-simps)
  thus ?thesis
  unfolding rcong-altdef by blast
qed

```

```

lemma sin-eq-sin-conv-rmod-iff:
   $\sin x = \sin y \longleftrightarrow [x = y] \text{ (rmod } 2 * \pi) \vee [x = \pi - y] \text{ (rmod } 2 * \pi)$ 
  by (metis sin-eq-sin-conv-rmod sin-pi-minus sin-rcong)

```

```

lemma cos-eq-cos-conv-rmod-iff:
   $\cos x = \cos y \longleftrightarrow [x = y] \text{ (rmod } 2 * \pi) \vee [x = -y] \text{ (rmod } 2 * \pi)$ 
  by (metis cos-eq-cos-conv-rmod cos-minus cos-rcong)

```

end

92 Generic reflection and reification

```

theory Reflection
imports Main
begin

```

```

ML-file  $\langle \sim \sim / \text{src} / \text{HOL} / \text{Tools} / \text{reflection.ML} \rangle$ 

```

```

method-setup reify = ⟨
  Attrib.thms --
  Scan.option (Scan.lift (Args.$$$ () |-- Args.term --| Scan.lift (Args.$$$ )))
>>
  (fn (user-egs, to) => fn ctxt => SIMPLE-METHOD' (Reflection.default-reify-tac
ctxt user-egs to))
⟩ partial automatic reification

method-setup reflection = ⟨
  let
    fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ();
    val onlyN = only;
    val rulesN = rules;
    val any-keyword = keyword onlyN || keyword rulesN;
    val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm);
    val terms = thms >> map (Thm.term-of o Drule.dest-term);
  in
    thms -- Scan.optional (keyword rulesN |-- thms) [] --
    Scan.option (keyword onlyN |-- Args.term) >>
    (fn ((user-egs, user-thms), to) => fn ctxt =>
      SIMPLE-METHOD' (Reflection.default-reflection-tac ctxt user-thms user-egs
to))
  end
⟩ partial automatic reflection

end

theory Rewrite
imports Main
begin

consts rewrite-HOLE :: 'a::{} (⟨⊔⟩)

lemma eta-expand:
  fixes f :: 'a::{} ⇒ 'b::{}
  shows f ≡ λx. f x .

lemma imp-cong-eq:
  (PROP A ⇒ (PROP B ⇒ PROP C) ≡ (PROP B' ⇒ PROP C')) ≡
  ((PROP B ⇒ PROP A ⇒ PROP C) ≡ (PROP B' ⇒ PROP A ⇒ PROP C'))
apply (intro Pure.equal-intr-rule)
apply (drule (1) cut-rl; drule Pure.equal-elim-rule1 Pure.equal-elim-rule2;
assumption)+
apply (drule Pure.equal-elim-rule1 Pure.equal-elim-rule2; assumption)+
done

```

```
ML-file <conv.ML>
ML-file <rewrite.ML>
```

```
end
```

93 Assigning lengths to types by type classes

```
theory Type-Length
imports Numeral-Type
begin
```

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in `Numeral_Type.thy`.

```
class len0 =
  fixes len-of :: 'a itself  $\Rightarrow$  nat

syntax -type-length :: type  $\Rightarrow$  nat ( $\langle (1LENGTH/(1'(-')))\rangle$ )
syntax-consts -type-length  $\Rightarrow$  len-of
translations LENGTH('a)  $\rightarrow$  CONST len-of TYPE('a)
print-translation <
  let
    fun len-of-itself-tr' ctxt [Const (const-syntax <Pure.type>, Type (-, [T]))] =
      Syntax.const syntax-const <-type-length> $ Syntax-Phases.term-of-typ ctxt T
    in [(const-syntax <len-of>, len-of-itself-tr')] end
  >
```

Some theorems are only true on words with length greater 0.

```
class len = len0 +
  assumes len-gt-0 [iff]: 0 < LENGTH('a)
begin
```

```
lemma len-not-eq-0 [simp]:
  LENGTH('a)  $\neq$  0
  by simp
```

```
end
```

```
instantiation num0 and num1 :: len0
begin
```

```
definition len-num0: len-of (- :: num0 itself) = 0
definition len-num1: len-of (- :: num1 itself) = 1
```

```
instance ..
```

```
end
```

```
instantiation bit0 and bit1 :: (len0) len0
```

begin

definition *len-bit0*: *len-of* ($- :: 'a::\text{len0}$ *bit0* *itself*) = $2 * \text{LENGTH}('a)$

definition *len-bit1*: *len-of* ($- :: 'a::\text{len0}$ *bit1* *itself*) = $2 * \text{LENGTH}('a) + 1$

instance ..

end

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len*

by *standard simp*

instance *bit0* :: (*len*) *len*

by *standard simp*

instance *bit1* :: (*len0*) *len*

by *standard simp*

instantiation *Enum.finite-1* :: *len*

begin

definition

len-of-finite-1 ($x :: \text{Enum.finite-1}$ *itself*) $\equiv (1 :: \text{nat})$

instance

by *standard (auto simp: len-of-finite-1-def)*

end

instantiation *Enum.finite-2* :: *len*

begin

definition

len-of-finite-2 ($x :: \text{Enum.finite-2}$ *itself*) $\equiv (2 :: \text{nat})$

instance

by *standard (auto simp: len-of-finite-2-def)*

end

instantiation *Enum.finite-3* :: *len*

begin

definition

len-of-finite-3 ($x :: \text{Enum.finite-3}$ *itself*) $\equiv (4 :: \text{nat})$

instance

by *standard (auto simp: len-of-finite-3-def)*

end

lemma *length-less-eq-Suc-0-iff* [simp]:
 $\langle \text{LENGTH}('a::\text{len}) \leq \text{Suc } 0 \longleftrightarrow \text{LENGTH}('a) = \text{Suc } 0 \rangle$
by (simp add: le-Suc-eq)

lemma *length-not-greater-eq-2-iff* [simp]:
 $\langle \neg 2 \leq \text{LENGTH}('a::\text{len}) \longleftrightarrow \text{LENGTH}('a) = \text{Suc } 0 \rangle$
by (auto simp add: not-le dest: less-2-cases)

lemma *less-eq-decr-length-iff* [simp]:
 $\langle n \leq \text{LENGTH}('a::\text{len}) - \text{Suc } 0 \longleftrightarrow n < \text{LENGTH}('a) \rangle$
by (cases $\langle \text{LENGTH}('a) \rangle$) (simp-all add: less-Suc-eq le-less)

lemma *decr-length-less-iff* [simp]:
 $\langle \text{LENGTH}('a::\text{len}) - \text{Suc } 0 < n \longleftrightarrow \text{LENGTH}('a) \leq n \rangle$
by (cases $\langle \text{LENGTH}('a) \rangle$) auto

context *linordered-idom*
begin

lemma *two-less-eq-exp-length* [simp]:
 $\langle 2 \leq 2 \wedge \text{LENGTH}('b::\text{len}) \rangle$
using *mult-left-mono* [of 1 $\langle 2 \wedge (\text{LENGTH}('b::\text{len}) - 1) \rangle 2]$
by (cases $\langle \text{LENGTH}('b::\text{len}) \rangle$) simp-all

end

end

94 Saturated arithmetic

theory *Saturated*
imports *Natural-Type Type-Length*
begin

94.1 The type of saturated naturals

typedef (overloaded) $('a::\text{len}) \text{ sat} = \{.. \text{LENGTH}('a)\}$
morphisms *nat-of Abs-sat*
by *auto*

lemma *sat-eqI*:
 $\text{nat-of } m = \text{nat-of } n \implies m = n$
by (simp add: nat-of-inject)

lemma *sat-eq-iff*:
 $m = n \longleftrightarrow \text{nat-of } m = \text{nat-of } n$
by (simp add: nat-of-inject)

lemma *Abs-sat-nat-of* [*code abstype*]:

Abs-sat (*nat-of* *n*) = *n*
by (*fact nat-of-inverse*)

definition *Abs-sat'* :: *nat* \Rightarrow *'a::len* *sat* **where**

Abs-sat' *n* = *Abs-sat* (*min* (*LENGTH*(*'a*)) *n*)

lemma *nat-of-Abs-sat'* [*simp*]:

nat-of (*Abs-sat'* *n* :: (*'a::len*) *sat*) = *min* (*LENGTH*(*'a*)) *n*
unfolding *Abs-sat'-def* **by** (*rule Abs-sat-inverse*) *simp*

lemma *nat-of-le-len-of* [*simp*]:

nat-of (*n* :: (*'a::len*) *sat*) \leq *LENGTH*(*'a*)
using *nat-of* [**where** *x* = *n*] **by** *simp*

lemma *min-len-of-nat-of* [*simp*]:

min (*LENGTH*(*'a*)) (*nat-of* (*n*::(*'a::len*) *sat*)) = *nat-of* *n*
by (*rule min.absorb2* [*OF nat-of-le-len-of*])

lemma *min-nat-of-len-of* [*simp*]:

min (*nat-of* (*n*::(*'a::len*) *sat*)) (*LENGTH*(*'a*)) = *nat-of* *n*
by (*subst min.commute*) *simp*

lemma *Abs-sat'-nat-of* [*simp*]:

Abs-sat' (*nat-of* *n*) = *n*
by (*simp add: Abs-sat'-def nat-of-inverse*)

instantiation *sat* :: (*len*) *linorder*

begin

definition

less-eq-sat-def: $x \leq y \longleftrightarrow \text{nat-of } x \leq \text{nat-of } y$

definition

less-sat-def: $x < y \longleftrightarrow \text{nat-of } x < \text{nat-of } y$

instance

by *standard*

(*auto simp add: less-eq-sat-def less-sat-def not-le sat-eq-iff min.coboundedI1 mult.commute*)

end

instantiation *sat* :: (*len*) {*minus*, *comm-semiring-1*}

begin

definition

0 = *Abs-sat'* *0*

definition

$$1 = \text{Abs-sat}' 1$$
lemma *nat-of-zero-sat* [*simp*, *code abstract*]:

$$\text{nat-of } 0 = 0$$
by (*simp add: zero-sat-def*)

lemma *nat-of-one-sat* [*simp*, *code abstract*]:

$$\text{nat-of } 1 = \min 1 (\text{LENGTH}('a))$$
by (*simp add: one-sat-def*)
definition

$$x + y = \text{Abs-sat}' (\text{nat-of } x + \text{nat-of } y)$$
lemma *nat-of-plus-sat* [*simp*, *code abstract*]:

$$\text{nat-of } (x + y) = \min (\text{nat-of } x + \text{nat-of } y) (\text{LENGTH}('a))$$
by (*simp add: plus-sat-def*)
definition

$$x - y = \text{Abs-sat}' (\text{nat-of } x - \text{nat-of } y)$$
lemma *nat-of-minus-sat* [*simp*, *code abstract*]:

$$\text{nat-of } (x - y) = \text{nat-of } x - \text{nat-of } y$$
proof –

from *nat-of-le-len-of* [*of x*] **have** $\text{nat-of } x - \text{nat-of } y \leq \text{LENGTH}('a)$ **by** *arith*
then show *?thesis* **by** (*simp add: minus-sat-def*)

qed
definition

$$x * y = \text{Abs-sat}' (\text{nat-of } x * \text{nat-of } y)$$
lemma *nat-of-times-sat* [*simp*, *code abstract*]:

$$\text{nat-of } (x * y) = \min (\text{nat-of } x * \text{nat-of } y) (\text{LENGTH}('a))$$
by (*simp add: times-sat-def*)
instance
proof
fix *a b c* :: *'a::len sat*
show $a * b * c = a * (b * c)$
proof(*cases a = 0*)

case *True* **thus** *?thesis* **by** (*simp add: sat-eq-iff*)

next
case *False* **show** *?thesis*
proof(*cases c = 0*)

case *True* **thus** *?thesis* **by** (*simp add: sat-eq-iff*)

next
case *False* **with** $\langle a \neq 0 \rangle$ **show** *?thesis*
by (*simp add: sat-eq-iff nat-mult-min-left nat-mult-min-right mult.assoc*)

```

min.assoc min.absorb2)
  qed
  qed
  show 1 * a = a
    by (simp add: sat-eq-iff min-def not-le not-less)
  show (a + b) * c = a * c + b * c
  proof(cases c = 0)
    case True thus ?thesis by (simp add: sat-eq-iff)
  next
    case False thus ?thesis
      by (simp add: sat-eq-iff nat-mult-min-left add-mult-distrib min-add-distrib-left
min-add-distrib-right min.assoc)
  qed
qed (simp-all add: sat-eq-iff mult.commute)

end

instantiation sat :: (len) ordered-comm-semiring
begin

instance
  by standard
  (auto simp add: less-eq-sat-def less-sat-def not-le sat-eq-iff min.coboundedI1
mult.commute)

end

lemma Abs-sat'-eq-of-nat: Abs-sat' n = of-nat n
  by (rule sat-eqI, induct n, simp-all)

abbreviation Sat :: nat  $\Rightarrow$  'a::len sat where
  Sat  $\equiv$  of-nat

lemma nat-of-Sat [simp]:
  nat-of (Sat n :: ('a::len) sat) = min (LENGTH('a)) n
  by (rule nat-of-Abs-sat' [unfolded Abs-sat'-eq-of-nat])

lemma [code-abbrev]:
  of-nat (numeral k) = (numeral k :: 'a::len sat)
  by simp

context
begin

qualified definition sat-of-nat :: nat  $\Rightarrow$  ('a::len) sat
  where [code-abbrev]: sat-of-nat = of-nat

lemma [code abstract]:
  nat-of (sat-of-nat n :: ('a::len) sat) = min (LENGTH('a)) n

```



```

    by (simp add: sat-of-nat-def)

end

instance sat :: (len) finite
proof
  show finite (UNIV::'a sat set)
    unfolding type-definition.univ [OF type-definition-sat]
    using finite by simp
qed

instantiation sat :: (len) equal
begin

definition HOL.equal A B  $\longleftrightarrow$  nat-of A = nat-of B

instance
  by standard (simp add: equal-sat-def nat-of-inject)

end

instantiation sat :: (len) {bounded-lattice, distrib-lattice}
begin

definition (inf :: 'a sat  $\Rightarrow$  'a sat  $\Rightarrow$  'a sat) = min
definition (sup :: 'a sat  $\Rightarrow$  'a sat  $\Rightarrow$  'a sat) = max
definition bot = (0 :: 'a sat)
definition top = Sat (LENGTH('a))

instance
  by standard
    (simp-all add: inf-sat-def sup-sat-def bot-sat-def top-sat-def max-min-distrib2,
     simp-all add: less-eq-sat-def)

end

instantiation sat :: (len) {Inf, Sup}
begin

global-interpretation Inf-sat: semilattice-neutr-set min <top :: 'a sat>
  defines Inf-sat = Inf-sat.F
  by standard (simp add: min-def)

global-interpretation Sup-sat: semilattice-neutr-set max <bot :: 'a sat>
  defines Sup-sat = Sup-sat.F
  by standard (simp add: max-def bot.extremum-unique)

instance ..

```

end

instance *sat* :: (*len*) *complete-lattice*

proof

fix *x* :: 'a *sat*

fix *A* :: 'a *sat set*

note *finite*

moreover assume $x \in A$

ultimately show $\text{Inf } A \leq x$

by (*induct A*) (*auto intro: min.coboundedI2*)

next

fix *z* :: 'a *sat*

fix *A* :: 'a *sat set*

note *finite*

moreover assume $z: \bigwedge x. x \in A \implies z \leq x$

ultimately show $z \leq \text{Inf } A$ **by** (*induct A*) *simp-all*

next

fix *x* :: 'a *sat*

fix *A* :: 'a *sat set*

note *finite*

moreover assume $x \in A$

ultimately show $x \leq \text{Sup } A$

by (*induct A*) (*auto intro: max.coboundedI2*)

next

fix *z* :: 'a *sat*

fix *A* :: 'a *sat set*

note *finite*

moreover assume $z: \bigwedge x. x \in A \implies x \leq z$

ultimately show $\text{Sup } A \leq z$ **by** (*induct A*) *auto*

next

show $\text{Inf } \{\} = (\text{top}::'a \text{ sat})$

by (*auto simp: top-sat-def*)

show $\text{Sup } \{\} = (\text{bot}::'a \text{ sat})$

by (*auto simp: bot-sat-def*)

qed

94.2 Enumeration

lemma *inj-on-sat-of-nat*:

shows *inj-on* (*of-nat* :: $\text{nat} \Rightarrow 'a::\text{len sat}$) $\{0..\text{LENGTH}('a)\}$

by (*rule inj-onI*) (*simp add: sat-eq-iff*)

lemma *UNIV-sat-eq-of-nat*:

shows (*UNIV* :: 'a::len sat set) = *of-nat* ' $\{0..\text{LENGTH}('a)\}$ (**is** ?*lhs* = ?*rhs*)

proof –

have $x \in ?\text{rhs}$ **for** $x :: 'a \text{ sat}$

by (*simp add: image-eqI* [**where** $x=\text{nat-of } x$] *sat-eq-iff*)

then show ?*thesis*

by *blast*

qed

instantiation *sat* :: (*len*) *enum*
begin

definition *enum-sat* :: 'a *sat list* **where**
enum-sat = *map of-nat* [0..*Suc*(*LENGTH*('a))]

definition *enum-all-sat* :: ('a *sat* \Rightarrow *bool*) \Rightarrow *bool* **where**
enum-all-sat = *All*

definition *enum-ex-sat* :: ('a *sat* \Rightarrow *bool*) \Rightarrow *bool* **where**
enum-ex-sat = *Ex*

instance

proof *intro-classes*

show *UNIV* = *set* (*enum-class.enum* :: 'a *sat list*)
by (*simp only: enum-sat-def UNIV-sat-eq-of-nat set-map flip: atLeastAtMost-upt*)
show *distinct* (*enum-class.enum* :: 'a *sat list*)
by (*clarsimp simp: enum-sat-def distinct-map inj-on-sat-of-nat sat-eq-iff*)
qed (*simp-all add: enum-all-sat-def enum-ex-sat-def*)

end

lemma *enum-sat-code* [*code*]:

fixes *P* :: 'a::*len* *sat* \Rightarrow *bool*
shows *Enum.enum-all* *P* \longleftrightarrow *list-all* *P* *Enum.enum*
and *Enum.enum-ex* *P* \longleftrightarrow *list-ex* *P* *Enum.enum*
by (*simp-all add: enum-all-sat-def enum-ex-sat-def enum-UNIV list-all-iff list-ex-iff*)

end

95 Set Idioms

theory *Set-Idioms*

imports *Countable-Set*

begin

95.1 Idioms for being a suitable union/intersection of something

definition *union-of* :: ('a *set set* \Rightarrow *bool*) \Rightarrow ('a *set* \Rightarrow *bool*) \Rightarrow 'a *set* \Rightarrow *bool*
(infixr <union'-of> 60)
where *P union-of Q* $\equiv \lambda S. \exists \mathcal{U}. P \mathcal{U} \wedge \mathcal{U} \subseteq \text{Collect } Q \wedge \bigcup \mathcal{U} = S$

definition *intersection-of* :: ('a *set set* \Rightarrow *bool*) \Rightarrow ('a *set* \Rightarrow *bool*) \Rightarrow 'a *set* \Rightarrow *bool*
(infixr <intersection'-of> 60)
where *P intersection-of Q* $\equiv \lambda S. \exists \mathcal{U}. P \mathcal{U} \wedge \mathcal{U} \subseteq \text{Collect } Q \wedge \bigcap \mathcal{U} = S$

definition *arbitrary*:: 'a set set \Rightarrow bool **where** *arbitrary* $\mathcal{U} \equiv \text{True}$

lemma *union-of-inc*: $\llbracket P \{S\}; Q \ S \rrbracket \Longrightarrow (P \text{ union-of } Q) \ S$
by (*auto simp: union-of-def*)

lemma *intersection-of-inc*:
 $\llbracket P \{S\}; Q \ S \rrbracket \Longrightarrow (P \text{ intersection-of } Q) \ S$
by (*auto simp: intersection-of-def*)

lemma *union-of-mono*:
 $\llbracket (P \text{ union-of } Q) \ S; \bigwedge x. Q \ x \Longrightarrow Q' \ x \rrbracket \Longrightarrow (P \text{ union-of } Q') \ S$
by (*auto simp: union-of-def*)

lemma *intersection-of-mono*:
 $\llbracket (P \text{ intersection-of } Q) \ S; \bigwedge x. Q \ x \Longrightarrow Q' \ x \rrbracket \Longrightarrow (P \text{ intersection-of } Q') \ S$
by (*auto simp: intersection-of-def*)

lemma *all-union-of*:
 $(\forall S. (P \text{ union-of } Q) \ S \longrightarrow R \ S) \longleftrightarrow (\forall T. P \ T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcup T))$
by (*auto simp: union-of-def*)

lemma *all-intersection-of*:
 $(\forall S. (P \text{ intersection-of } Q) \ S \longrightarrow R \ S) \longleftrightarrow (\forall T. P \ T \wedge T \subseteq \text{Collect } Q \longrightarrow R(\bigcap T))$
by (*auto simp: intersection-of-def*)

lemma *intersection-ofE*:
 $\llbracket (P \text{ intersection-of } Q) \ S; \bigwedge T. \llbracket P \ T; T \subseteq \text{Collect } Q \rrbracket \Longrightarrow R(\bigcap T) \rrbracket \Longrightarrow R \ S$
by (*auto simp: intersection-of-def*)

lemma *union-of-empty*:
 $P \ \{\} \Longrightarrow (P \text{ union-of } Q) \ \{\}$
by (*auto simp: union-of-def*)

lemma *intersection-of-empty*:
 $P \ \{\} \Longrightarrow (P \text{ intersection-of } Q) \ \text{UNIV}$
by (*auto simp: intersection-of-def*)

The arbitrary and finite cases

lemma *arbitrary-union-of-alt*:
 $(\text{arbitrary union-of } Q) \ S \longleftrightarrow (\forall x \in S. \exists U. Q \ U \wedge x \in U \wedge U \subseteq S)$
(is ?lhs = ?rhs)

proof
assume *?lhs*
then show *?rhs*
by (*force simp: union-of-def arbitrary-def*)
next
assume *?rhs*

then have $\{U. Q \ U \wedge U \subseteq S\} \subseteq \text{Collect } Q \cup \{U. Q \ U \wedge U \subseteq S\} = S$
by *auto*
then show *?lhs*
unfolding *union-of-def arbitrary-def* **by** *blast*
qed

lemma *arbitrary-union-of-empty [simp]: (arbitrary union-of P) {}*
by (*force simp: union-of-def arbitrary-def*)

lemma *arbitrary-intersection-of-empty [simp]:*
(arbitrary intersection-of P) UNIV
by (*force simp: intersection-of-def arbitrary-def*)

lemma *arbitrary-union-of-inc:*
 $P \ S \implies (\text{arbitrary union-of } P) \ S$
by (*force simp: union-of-inc arbitrary-def*)

lemma *arbitrary-intersection-of-inc:*
 $P \ S \implies (\text{arbitrary intersection-of } P) \ S$
by (*force simp: intersection-of-inc arbitrary-def*)

lemma *arbitrary-union-of-complement:*
 $(\text{arbitrary union-of } P) \ S \longleftrightarrow (\text{arbitrary intersection-of } (\lambda S. P(- S))) (- S)$
(is ?lhs = ?rhs)

proof
assume *?lhs*
then obtain \mathcal{U} **where** $\mathcal{U} \subseteq \text{Collect } P \ S = \bigcup \mathcal{U}$
by (*auto simp: union-of-def arbitrary-def*)
then show *?rhs*
unfolding *intersection-of-def arbitrary-def*
by (*rule-tac x=uminus 'U in exI*) *auto*
next
assume *?rhs*
then obtain \mathcal{U} **where** $\mathcal{U} \subseteq \{S. P (- S)\} \cap \mathcal{U} = - S$
by (*auto simp: union-of-def intersection-of-def arbitrary-def*)
then show *?lhs*
unfolding *union-of-def arbitrary-def*
by (*rule-tac x=uminus 'U in exI*) *auto*
qed

lemma *arbitrary-intersection-of-complement:*
 $(\text{arbitrary intersection-of } P) \ S \longleftrightarrow (\text{arbitrary union-of } (\lambda S. P(- S))) (- S)$
by (*simp add: arbitrary-union-of-complement*)

lemma *arbitrary-union-of-idempot [simp]:*
 $\text{arbitrary union-of arbitrary union-of } P = \text{arbitrary union-of } P$
proof –
have $1: \exists \mathcal{U}' \subseteq \text{Collect } P. \bigcup \mathcal{U}' = \bigcup \mathcal{U}$ **if** $\mathcal{U} \subseteq \{S. \exists \mathcal{V} \subseteq \text{Collect } P. \bigcup \mathcal{V} = S\}$ **for**
 \mathcal{U}

proof –
 let $?W = \{V. \exists \mathcal{V}. \mathcal{V} \subseteq \text{Collect } P \wedge V \in \mathcal{V} \wedge (\exists S \in \mathcal{U}. \bigcup \mathcal{V} = S)\}$
 have *: $\bigwedge x U. \llbracket x \in U; U \in \mathcal{U} \rrbracket \implies x \in \bigcup ?W$
 using *that*
 apply *simp*
 apply (*drule subsetD, assumption, auto*)
 done
 show *?thesis*
 apply (*rule-tac* $x = \{V. \exists \mathcal{V}. \mathcal{V} \subseteq \text{Collect } P \wedge V \in \mathcal{V} \wedge (\exists S \in \mathcal{U}. \bigcup \mathcal{V} = S)\}$ in
exI)
 using *that* by (*blast intro: **)
 qed
 have $\mathcal{U} \subseteq \{S. \exists \mathcal{U}' \subseteq \{S. \exists \mathcal{U} \subseteq \text{Collect } P. \bigcup \mathcal{U} = S\}. \bigcup \mathcal{U}' = \bigcup \mathcal{U} \text{ if } \mathcal{U} \subseteq \text{Collect } P \text{ for } \mathcal{U}\}$
 by (*metis (mono-tags, lifting) union-of-def arbitrary-union-of-inc that*)
 show *?thesis*
 unfolding *union-of-def arbitrary-def* by (*force simp: 1 2*)
 qed

lemma *arbitrary-intersection-of-idempot*:
 $\text{arbitrary intersection-of arbitrary intersection-of } P = \text{arbitrary intersection-of } P$
(is ?lhs = ?rhs)
proof –
 have – *?lhs = – ?rhs*
 unfolding *arbitrary-intersection-of-complement* by *simp*
 then show *?thesis*
 by *simp*
 qed

lemma *arbitrary-union-of-Union*:
 $(\bigwedge S. S \in \mathcal{U} \implies (\text{arbitrary union-of } P) S) \implies (\text{arbitrary union-of } P) (\bigcup \mathcal{U})$
 by (*metis union-of-def arbitrary-def arbitrary-union-of-idempot mem-Collect-eq subsetI*)

lemma *arbitrary-union-of-Un*:
 $\llbracket (\text{arbitrary union-of } P) S; (\text{arbitrary union-of } P) T \rrbracket$
 $\implies (\text{arbitrary union-of } P) (S \cup T)$
 using *arbitrary-union-of-Union* [of $\{S, T\}$] by *auto*

lemma *arbitrary-intersection-of-Inter*:
 $(\bigwedge S. S \in \mathcal{U} \implies (\text{arbitrary intersection-of } P) S) \implies (\text{arbitrary intersection-of } P) (\bigcap \mathcal{U})$
 by (*metis intersection-of-def arbitrary-def arbitrary-intersection-of-idempot mem-Collect-eq subsetI*)

lemma *arbitrary-intersection-of-Int*:
 $\llbracket (\text{arbitrary intersection-of } P) S; (\text{arbitrary intersection-of } P) T \rrbracket$
 $\implies (\text{arbitrary intersection-of } P) (S \cap T)$
 using *arbitrary-intersection-of-Inter* [of $\{S, T\}$] by *auto*

lemma *arbitrary-union-of-Int-eq*:

$$(\forall S T. (\text{arbitrary union-of } P) S \wedge (\text{arbitrary union-of } P) T \longrightarrow (\text{arbitrary union-of } P) (S \cap T)) \\ \longleftrightarrow (\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary union-of } P) (S \cap T)) \quad (\text{is } ?lhs = ?rhs)$$

proof

assume *?lhs*

then show *?rhs*

by (*simp add: arbitrary-union-of-inc*)

next

assume *R: ?rhs*

show *?lhs*

proof *clarify*

fix *S :: 'a set* and *T :: 'a set*

assume *(arbitrary union-of P) S* and *(arbitrary union-of P) T*

then obtain *U V* where **: U ⊆ Collect P ∪ U = S* *V ⊆ Collect P ∪ V = T*

by (*auto simp: union-of-def*)

then have *(arbitrary union-of P) (∪ C∈U. ∪ D∈V. C ∩ D)*

using *R* by (*blast intro: arbitrary-union-of-Union*)

then show *(arbitrary union-of P) (S ∩ T)*

by (*simp add: Int-UN-distrib2 **)

qed

qed

lemma *arbitrary-intersection-of-Un-eq*:

$$(\forall S T. (\text{arbitrary intersection-of } P) S \wedge (\text{arbitrary intersection-of } P) T \longrightarrow (\text{arbitrary intersection-of } P) (S \cup T)) \longleftrightarrow \\ (\forall S T. P S \wedge P T \longrightarrow (\text{arbitrary intersection-of } P) (S \cup T))$$

apply (*simp add: arbitrary-intersection-of-complement*)

using *arbitrary-union-of-Int-eq* [*of λS. P (− S)*]

by (*metis (no-types, lifting) arbitrary-def double-compl union-of-inc*)

lemma *finite-union-of-empty* [*simp*]: *(finite union-of P) {}*

by (*simp add: union-of-empty*)

lemma *finite-intersection-of-empty* [*simp*]: *(finite intersection-of P) UNIV*

by (*simp add: intersection-of-empty*)

lemma *finite-union-of-inc*:

P S \implies *(finite union-of P) S*

by (*simp add: union-of-inc*)

lemma *finite-intersection-of-inc*:

P S \implies *(finite intersection-of P) S*

by (*simp add: intersection-of-inc*)

lemma *finite-union-of-complement*:

(finite union-of P) S \longleftrightarrow *(finite intersection-of (λS. P(− S))) (− S)*

unfolding *union-of-def intersection-of-def*

apply *safe*
apply (*rule-tac* $x = \text{uminus } 'U \text{ in } exI, \text{fastforce}) +$
done

lemma *finite-intersection-of-complement*:
 $(\text{finite intersection-of } P) S \longleftrightarrow (\text{finite union-of } (\lambda S. P(- S))) (- S)$
by (*simp add: finite-union-of-complement*)

lemma *finite-union-of-idempot* [*simp*]:
 $\text{finite union-of finite union-of } P = \text{finite union-of } P$
proof –
have ($\text{finite union-of } P$) S **if** S : ($\text{finite union-of finite union-of } P$) S **for** S
proof –
obtain \mathcal{U} **where** $\text{finite } \mathcal{U} S = \bigcup \mathcal{U}$ **and** \mathcal{U} : $\forall U \in \mathcal{U}. \exists U. \text{finite } U \wedge (U \subseteq \text{Collect } P) \wedge \bigcup \mathcal{U} = U$
using S **unfolding** *union-of-def* **by** (*auto simp: subset-eq*)
then obtain f **where** $\forall U \in \mathcal{U}. \text{finite } (f U) \wedge (f U \subseteq \text{Collect } P) \wedge \bigcup (f U) = U$
by *metis*
then show *?thesis*
unfolding *union-of-def* $\langle S = \bigcup \mathcal{U} \rangle$
by (*rule-tac* $x = \text{snd } 'Sigma \mathcal{U} f \text{ in } exI$) (*fastforce simp: \langle finite \mathcal{U} \rangle*)
qed
moreover
have ($\text{finite union-of finite union-of } P$) S **if** ($\text{finite union-of } P$) S **for** S
by (*simp add: finite-union-of-inc that*)
ultimately show *?thesis*
by *force*
qed

lemma *finite-intersection-of-idempot* [*simp*]:
 $\text{finite intersection-of finite intersection-of } P = \text{finite intersection-of } P$
by (*force simp: finite-intersection-of-complement*)

lemma *finite-union-of-Union*:
 $\llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{finite union-of } P) S \rrbracket \implies (\text{finite union-of } P) (\bigcup \mathcal{U})$
using *finite-union-of-idempot [of P]*
by (*metis mem-Collect-eq subsetI union-of-def*)

lemma *finite-union-of-Un*:
 $\llbracket (\text{finite union-of } P) S; (\text{finite union-of } P) T \rrbracket \implies (\text{finite union-of } P) (S \cup T)$
by (*auto simp: union-of-def*)

lemma *finite-intersection-of-Inter*:
 $\llbracket \text{finite } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{finite intersection-of } P) S \rrbracket \implies (\text{finite intersection-of } P) (\bigcap \mathcal{U})$
using *finite-intersection-of-idempot [of P]*
by (*metis intersection-of-def mem-Collect-eq subsetI*)

lemma *finite-intersection-of-Int*:

$\llbracket (\text{finite intersection-of } P) \ S; (\text{finite intersection-of } P) \ T \rrbracket$
 $\implies (\text{finite intersection-of } P) \ (S \cap T)$
by (*auto simp: intersection-of-def*)

lemma *finite-union-of-Int-eq*:

$(\forall S \ T. (\text{finite union-of } P) \ S \wedge (\text{finite union-of } P) \ T \longrightarrow (\text{finite union-of } P) \ (S \cap T))$

$\longleftrightarrow (\forall S \ T. P \ S \wedge P \ T \longrightarrow (\text{finite union-of } P) \ (S \cap T))$

(is ?lhs = ?rhs)

proof

assume *?lhs*

then show *?rhs*

by (*simp add: finite-union-of-inc*)

next

assume *R: ?rhs*

show *?lhs*

proof clarify

fix *S :: 'a set* **and** *T :: 'a set*

assume $(\text{finite union-of } P) \ S$ **and** $(\text{finite union-of } P) \ T$

then obtain *U V* **where** $U \subseteq \text{Collect } P \cup U = S$ *finite U V* $V \subseteq \text{Collect } P \cup V = T$ *finite V*

by (*auto simp: union-of-def*)

then have $(\text{finite union-of } P) \ (\bigcup C \in U. \bigcup D \in V. C \cap D)$

using *R*

by (*blast intro: finite-union-of-Union*)

then show $(\text{finite union-of } P) \ (S \cap T)$

by (*simp add: Int-UN-distrib2 **)

qed

qed

lemma *finite-intersection-of-Un-eq*:

$(\forall S \ T. (\text{finite intersection-of } P) \ S \wedge$

$(\text{finite intersection-of } P) \ T$

$\longrightarrow (\text{finite intersection-of } P) \ (S \cup T)) \longleftrightarrow$

$(\forall S \ T. P \ S \wedge P \ T \longrightarrow (\text{finite intersection-of } P) \ (S \cup T))$

apply (*simp add: finite-intersection-of-complement*)

using *finite-union-of-Int-eq* [*of* $\lambda S. P \ (- S)$]

by (*metis (no-types, lifting) double-compl*)

abbreviation *finite'* $:: 'a \text{ set} \Rightarrow \text{bool}$

where *finite'* *A* $\equiv \text{finite } A \wedge A \neq \{\}$

lemma *finite'-intersection-of-Int*:

$\llbracket (\text{finite}' \text{ intersection-of } P) \ S; (\text{finite}' \text{ intersection-of } P) \ T \rrbracket$

$\implies (\text{finite}' \text{ intersection-of } P) \ (S \cap T)$

by (*auto simp: intersection-of-def*)

lemma *finite'-intersection-of-inc*:

$P \ S \Longrightarrow (\text{finite' intersection-of } P) \ S$
by (*simp add: intersection-of-inc*)

95.2 The “Relative to” operator

A somewhat cheap but handy way of getting localized forms of various topological concepts (open, closed, borel, fsigma, gdelta etc.)

definition *relative-to* :: [*'a set* \Rightarrow *bool*, *'a set*, *'a set*] \Rightarrow *bool* (**infixl** \langle *relative'-to* \rangle 55)

where $P \text{ relative-to } S \equiv \lambda T. \exists U. P \ U \wedge S \cap U = T$

lemma *relative-to-UNIV* [*simp*]: $(P \text{ relative-to } \text{UNIV}) \ S \longleftrightarrow P \ S$
by (*simp add: relative-to-def*)

lemma *relative-to-imp-subset*:
 $(P \text{ relative-to } S) \ T \Longrightarrow T \subseteq S$
by (*auto simp: relative-to-def*)

lemma *all-relative-to*: $(\forall S. (P \text{ relative-to } U) \ S \longrightarrow Q \ S) \longleftrightarrow (\forall S. P \ S \longrightarrow Q (U \cap S))$
by (*auto simp: relative-to-def*)

lemma *relative-toE*: $\llbracket (P \text{ relative-to } U) \ S; \bigwedge S. P \ S \Longrightarrow Q (U \cap S) \rrbracket \Longrightarrow Q \ S$
by (*auto simp: relative-to-def*)

lemma *relative-to-inc*:
 $P \ S \Longrightarrow (P \text{ relative-to } U) \ (U \cap S)$
by (*auto simp: relative-to-def*)

lemma *relative-to-relative-to* [*simp*]:
 $P \text{ relative-to } S \text{ relative-to } T = P \text{ relative-to } (S \cap T)$
unfolding *relative-to-def*
by *auto*

lemma *relative-to-compl*:
 $S \subseteq U \Longrightarrow ((P \text{ relative-to } U) \ (U - S) \longleftrightarrow ((\lambda c. P(- \ c)) \text{ relative-to } U) \ S)$
unfolding *relative-to-def*
by (*metis Diff-Diff-Int Diff-eq double-compl inf.absorb-iff2*)

lemma *relative-to-subset-trans*:
 $\llbracket (P \text{ relative-to } U) \ S; S \subseteq T; T \subseteq U \rrbracket \Longrightarrow (P \text{ relative-to } T) \ S$
unfolding *relative-to-def* **by** *auto*

lemma *relative-to-mono*:
 $\llbracket (P \text{ relative-to } U) \ S; \bigwedge S. P \ S \Longrightarrow Q \ S \rrbracket \Longrightarrow (Q \text{ relative-to } U) \ S$
unfolding *relative-to-def* **by** *auto*

lemma *relative-to-subset-inc*: $\llbracket S \subseteq U; P \ S \rrbracket \Longrightarrow (P \text{ relative-to } U) \ S$
unfolding *relative-to-def* **by** *auto*

lemma *relative-to-Int*:

$$\begin{aligned} & \llbracket (P \text{ relative-to } S) \ C; (P \text{ relative-to } S) \ D; \bigwedge X \ Y. \llbracket P \ X; P \ Y \rrbracket \implies P(X \cap Y) \rrbracket \\ & \implies (P \text{ relative-to } S) \ (C \cap D) \end{aligned}$$

unfolding *relative-to-def* **by** *auto*

lemma *relative-to-Un*:

$$\begin{aligned} & \llbracket (P \text{ relative-to } S) \ C; (P \text{ relative-to } S) \ D; \bigwedge X \ Y. \llbracket P \ X; P \ Y \rrbracket \implies P(X \cup Y) \rrbracket \\ & \implies (P \text{ relative-to } S) \ (C \cup D) \end{aligned}$$

unfolding *relative-to-def* **by** *auto*

lemma *arbitrary-union-of-relative-to*:

$$\begin{aligned} & ((\text{arbitrary union-of } P) \text{ relative-to } U) = (\text{arbitrary union-of } (P \text{ relative-to } U)) \\ & (\text{is } ?lhs = ?rhs) \end{aligned}$$

proof –

have *?rhs S if L: ?lhs S for S*

proof –

obtain \mathcal{U} **where** $S = U \cap \bigcup \mathcal{U} \ \mathcal{U} \subseteq \text{Collect } P$

using *L* **unfolding** *relative-to-def union-of-def* **by** *auto*

then show *?thesis*

unfolding *relative-to-def union-of-def arbitrary-def*

by (*rule-tac* $x=(\lambda X. U \cap X)$ ‘ \mathcal{U} in *exI*) *auto*

qed

moreover have *?lhs S if R: ?rhs S for S*

proof –

obtain \mathcal{U} **where** $S = \bigcup \mathcal{U} \ \forall T \in \mathcal{U}. \exists V. P \ V \wedge U \cap V = T$

using *R* **unfolding** *relative-to-def union-of-def* **by** *auto*

then obtain f **where** $f: \bigwedge T. T \in \mathcal{U} \implies P \ (f \ T) \wedge T. T \in \mathcal{U} \implies U \cap (f \ T)$

$= T$

by *metis*

then have $\exists \mathcal{U}' \subseteq \text{Collect } P. \bigcup \mathcal{U}' = \bigcup (f \ ' \ \mathcal{U})$

by (*metis image-subset-iff mem-Collect-eq*)

moreover have *eq*: $U \cap \bigcup (f \ ' \ \mathcal{U}) = \bigcup \mathcal{U}$

using f **by** *auto*

ultimately show *?thesis*

unfolding *relative-to-def union-of-def arbitrary-def* ‘ $S = \bigcup \mathcal{U}$ ’

by *metis*

qed

ultimately show *?thesis*

by *blast*

qed

lemma *finite-union-of-relative-to*:

$$\begin{aligned} & ((\text{finite union-of } P) \text{ relative-to } U) = (\text{finite union-of } (P \text{ relative-to } U)) \ (\text{is } ?lhs \\ & = ?rhs) \end{aligned}$$

proof –

have *?rhs S if L: ?lhs S for S*

proof –

obtain \mathcal{U} **where** $S = U \cap \bigcup \mathcal{U} \ \mathcal{U} \subseteq \text{Collect } P$ *finite* \mathcal{U}

```

    using L unfolding relative-to-def union-of-def by auto
  then show ?thesis
    unfolding relative-to-def union-of-def
    by (rule-tac x=( $\lambda X. U \cap X$ ) ‘ $\mathcal{U}$  in exI) auto
qed
moreover have ?lhs S if R: ?rhs S for S
proof -
  obtain  $\mathcal{U}$  where  $S = \bigcup \mathcal{U} \ \forall T \in \mathcal{U}. \exists V. P \ V \wedge U \cap V = T$  finite  $\mathcal{U}$ 
    using R unfolding relative-to-def union-of-def by auto
  then obtain f where  $f: \bigwedge T. T \in \mathcal{U} \implies P \ (f \ T) \wedge T. T \in \mathcal{U} \implies U \cap (f \ T)$ 
= T
    by metis
  then have  $\exists \mathcal{U}' \subseteq \text{Collect } P. \bigcup \mathcal{U}' = \bigcup (f \ ' \ \mathcal{U})$ 
    by (metis image-subset-iff mem-Collect-eq)
  moreover have eq:  $U \cap \bigcup (f \ ' \ \mathcal{U}) = \bigcup \mathcal{U}$ 
    using f by auto
  ultimately show ?thesis
    using ‘finite  $\mathcal{U}$ ’ f
    unfolding relative-to-def union-of-def ‘ $S = \bigcup \mathcal{U}$ ’
    by (rule-tac x= $\bigcup (f \ ' \ \mathcal{U})$  in exI) (metis finite-imageI image-subsetI mem-Collect-eq)
qed
ultimately show ?thesis
  by blast
qed

```

lemma countable-union-of-relative-to:

((countable union-of P) relative-to U) = (countable union-of (P relative-to U))
(is ?lhs = ?rhs)

proof –

have ?rhs S if L: ?lhs S for S

proof –

obtain \mathcal{U} where $S = U \cap \bigcup \mathcal{U} \ \mathcal{U} \subseteq \text{Collect } P$ countable \mathcal{U}

using L unfolding relative-to-def union-of-def by auto

then show ?thesis

unfolding relative-to-def union-of-def

by (rule-tac x=($\lambda X. U \cap X$) ‘ \mathcal{U} in exI) auto

qed

moreover have ?lhs S if R: ?rhs S for S

proof –

obtain \mathcal{U} where $S = \bigcup \mathcal{U} \ \forall T \in \mathcal{U}. \exists V. P \ V \wedge U \cap V = T$ countable \mathcal{U}

using R unfolding relative-to-def union-of-def by auto

then obtain f where $f: \bigwedge T. T \in \mathcal{U} \implies P \ (f \ T) \wedge T. T \in \mathcal{U} \implies U \cap (f \ T)$

= T

by metis

then have $\exists \mathcal{U}' \subseteq \text{Collect } P. \bigcup \mathcal{U}' = \bigcup (f \ ' \ \mathcal{U})$

by (metis image-subset-iff mem-Collect-eq)

moreover have eq: $U \cap \bigcup (f \ ' \ \mathcal{U}) = \bigcup \mathcal{U}$

using f by auto

ultimately show ?thesis

```

    using ⟨countable  $\mathcal{U}$ ⟩  $f$ 
    unfolding relative-to-def union-of-def ⟨ $S = \bigcup \mathcal{U}$ ⟩
    by (rule-tac  $x = \bigcup (f \text{ ‘ } \mathcal{U})$  in  $exI$ ) (metis countable-image image-subsetI
mem-Collect-eq)
  qed
  ultimately show ?thesis
    by blast
  qed

```

lemma *arbitrary-intersection-of-relative-to:*

$((\text{arbitrary intersection-of } P) \text{ relative-to } U) = ((\text{arbitrary intersection-of } (P \text{ relative-to } U)) \text{ relative-to } U)$ (is ?lhs = ?rhs)

proof –

have ?rhs S if L : ?lhs S for S

proof –

obtain \mathcal{U} where \mathcal{U} : $S = U \cap \bigcap \mathcal{U}$ $\mathcal{U} \subseteq \text{Collect } P$

using L unfolding relative-to-def intersection-of-def by auto

show ?thesis

unfolding relative-to-def intersection-of-def arbitrary-def

proof (intro exI $conjI$)

show $U \cap (\bigcap X \in \mathcal{U}. U \cap X) = S \cap U \text{ ‘ } \mathcal{U} \subseteq \{T. \exists Ua. P \ Ua \wedge U \cap Ua = T\}$

using \mathcal{U} by blast+

qed auto

qed

moreover have ?lhs S if R : ?rhs S for S

proof –

obtain \mathcal{U} where $S = U \cap \bigcap \mathcal{U} \forall T \in \mathcal{U}. \exists V. P \ V \wedge U \cap V = T$

using R unfolding relative-to-def intersection-of-def by auto

then obtain f where f : $\bigwedge T. T \in \mathcal{U} \implies P \ (f \ T) \wedge T. T \in \mathcal{U} \implies U \cap (f \ T) = T$

by metis

then have $f \text{ ‘ } \mathcal{U} \subseteq \text{Collect } P$

by auto

moreover have eq: $U \cap \bigcap (f \text{ ‘ } \mathcal{U}) = U \cap \bigcap \mathcal{U}$

using f by auto

ultimately show ?thesis

unfolding relative-to-def intersection-of-def arbitrary-def ⟨ $S = U \cap \bigcap \mathcal{U}$ ⟩

by auto

qed

ultimately show ?thesis

by blast

qed

lemma *finite-intersection-of-relative-to:*

$((\text{finite intersection-of } P) \text{ relative-to } U) = ((\text{finite intersection-of } (P \text{ relative-to } U)) \text{ relative-to } U)$ (is ?lhs = ?rhs)

proof –

```

have ?rhs S if L: ?lhs S for S
proof -
  obtain  $\mathcal{U}$  where  $\mathcal{U}: S = U \cap \bigcap \mathcal{U} \mathcal{U} \subseteq \text{Collect } P \text{ finite } \mathcal{U}$ 
  using L unfolding relative-to-def intersection-of-def by auto
  show ?thesis
  unfolding relative-to-def intersection-of-def
  proof (intro exI conjI)
    show  $U \cap (\bigcap_{X \in \mathcal{U}} U \cap X) = S \cap U \mathcal{U} \subseteq \{T. \exists Ua. P Ua \wedge U \cap Ua = T\}$ 
    using  $\mathcal{U}$  by blast+
    show finite  $((\cap) U \mathcal{U})$ 
    by (simp add:  $\langle \text{finite } \mathcal{U} \rangle$ )
  qed auto
qed
moreover have ?lhs S if R: ?rhs S for S
proof -
  obtain  $\mathcal{U}$  where  $S = U \cap \bigcap \mathcal{U} \forall T \in \mathcal{U}. \exists V. P V \wedge U \cap V = T \text{ finite } \mathcal{U}$ 
  using R unfolding relative-to-def intersection-of-def by auto
  then obtain f where  $f: \bigwedge T. T \in \mathcal{U} \implies P (f T) \bigwedge T. T \in \mathcal{U} \implies U \cap (f T) = T$ 
  by metis
  then have  $f \mathcal{U} \subseteq \text{Collect } P$ 
  by auto
  moreover have eq:  $U \cap \bigcap (f \mathcal{U}) = U \cap \bigcap \mathcal{U}$ 
  using f by auto
  ultimately show ?thesis
  unfolding relative-to-def intersection-of-def  $\langle S = U \cap \bigcap \mathcal{U} \rangle$ 
  using  $\langle \text{finite } \mathcal{U} \rangle$ 
  by auto
qed
ultimately show ?thesis
by blast
qed

```

lemma *countable-intersection-of-relative-to:*

$((\text{countable intersection-of } P) \text{ relative-to } U) = ((\text{countable intersection-of } (P \text{ relative-to } U)) \text{ relative-to } U)$ (is ?lhs = ?rhs)

```

proof -
  have ?rhs S if L: ?lhs S for S
  proof -
    obtain  $\mathcal{U}$  where  $\mathcal{U}: S = U \cap \bigcap \mathcal{U} \mathcal{U} \subseteq \text{Collect } P \text{ countable } \mathcal{U}$ 
    using L unfolding relative-to-def intersection-of-def by auto
    show ?thesis
    unfolding relative-to-def intersection-of-def
    proof (intro exI conjI)
      show  $U \cap (\bigcap_{X \in \mathcal{U}} U \cap X) = S \cap U \mathcal{U} \subseteq \{T. \exists Ua. P Ua \wedge U \cap Ua = T\}$ 
      using  $\mathcal{U}$  by blast+
      show countable  $((\cap) U \mathcal{U})$ 

```

```

    by (simp add: ‹countable  $\mathcal{U}$ ›)
  qed auto
qed
moreover have ?lhs  $S$  if  $R$ : ?rhs  $S$  for  $S$ 
proof -
  obtain  $\mathcal{U}$  where  $S = U \cap \bigcap \mathcal{U} \ \forall T \in \mathcal{U}. \exists V. P \ V \wedge U \cap V = T$  countable  $\mathcal{U}$ 
  using  $R$  unfolding relative-to-def intersection-of-def by auto
  then obtain  $f$  where  $f: \bigwedge T. T \in \mathcal{U} \implies P \ (f \ T) \wedge T. T \in \mathcal{U} \implies U \cap (f \ T)$ 
=  $T$ 
    by metis
  then have  $f \text{ ‘ } \mathcal{U} \subseteq \text{Collect } P$ 
    by auto
  moreover have eq:  $U \cap \bigcap (f \text{ ‘ } \mathcal{U}) = U \cap \bigcap \mathcal{U}$ 
    using  $f$  by auto
  ultimately show ?thesis
    unfolding relative-to-def intersection-of-def ‹ $S = U \cap \bigcap \mathcal{U}$ ›
    using ‹countable  $\mathcal{U}$ › countable-image
    by auto
qed
ultimately show ?thesis
  by blast
qed

lemma countable-union-of-empty [simp]: (countable union-of  $P$ ) {}
  by (simp add: union-of-empty)

lemma countable-intersection-of-empty [simp]: (countable intersection-of  $P$ )  $UNIV$ 
  by (simp add: intersection-of-empty)

lemma countable-union-of-inc:  $P \ S \implies (\text{countable union-of } P) \ S$ 
  by (simp add: union-of-inc)

lemma countable-intersection-of-inc:  $P \ S \implies (\text{countable intersection-of } P) \ S$ 
  by (simp add: intersection-of-inc)

lemma countable-union-of-complement:
  (countable union-of  $P$ )  $S \longleftrightarrow (\text{countable intersection-of } (\lambda S. P(-S))) \ (-S)$ 
  (is ?lhs=?rhs)
proof
  assume ?lhs
  then obtain  $\mathcal{U}$  where countable  $\mathcal{U}$  and  $\mathcal{U}: \mathcal{U} \subseteq \text{Collect } P \ \bigcup \mathcal{U} = S$ 
    by (metis union-of-def)
  define  $\mathcal{U}'$  where  $\mathcal{U}' \equiv (\lambda C. -C) \text{ ‘ } \mathcal{U}$ 
  have  $\mathcal{U}' \subseteq \{S. P \ (-S)\} \ \bigcap \mathcal{U}' = -S$ 
    using  $\mathcal{U}'$ -def  $\mathcal{U}$  by auto
  then show ?rhs
    unfolding intersection-of-def by (metis  $\mathcal{U}'$ -def ‹countable  $\mathcal{U}$ › countable-image)
next
  assume ?rhs

```

then obtain \mathcal{U} where countable \mathcal{U} and \mathcal{U} : $\mathcal{U} \subseteq \{S. P(-S)\} \cap \mathcal{U} = -S$
 by (metis intersection-of-def)
 define \mathcal{U}' where $\mathcal{U}' \equiv (\lambda C. -C) \cdot \mathcal{U}$
 have $\mathcal{U}' \subseteq \text{Collect } P \cup \mathcal{U}' = S$
 using \mathcal{U}' -def \mathcal{U} by auto
 then show ?lhs
 unfolding union-of-def
 by (metis \mathcal{U}' -def countable \mathcal{U} countable-image)
 qed

lemma countable-intersection-of-complement:
 (countable intersection-of P) $S \longleftrightarrow$ (countable union-of $(\lambda S. P(-S))$) $(-S)$
 by (simp add: countable-union-of-complement)

lemma countable-union-of-explicit:
 assumes $P \{\}$
 shows (countable union-of P) $S \longleftrightarrow$
 ($\exists T. (\forall n::\text{nat}. P(T\ n)) \wedge \bigcup(\text{range } T) = S$) (is ?lhs=?rhs)
 proof
 assume ?lhs
 then obtain \mathcal{U} where countable \mathcal{U} and \mathcal{U} : $\mathcal{U} \subseteq \text{Collect } P \cup \mathcal{U} = S$
 by (metis union-of-def)
 then show ?rhs
 by (metis SUP-bot Sup-empty assms from-nat-into mem-Collect-eq range-from-nat-into subsetD)
 next
 assume ?rhs
 then show ?lhs
 by (metis countableI-type countable-image image-subset-iff mem-Collect-eq union-of-def)
 qed

lemma countable-union-of-ascending:
 assumes empty: $P \{\}$ and $Un: \bigwedge T\ U. \llbracket P\ T; P\ U \rrbracket \implies P(T \cup U)$
 shows (countable union-of P) $S \longleftrightarrow$
 ($\exists T. (\forall n. P(T\ n)) \wedge (\forall n. T\ n \subseteq T(\text{Suc } n)) \wedge \bigcup(\text{range } T) = S$) (is ?lhs=?rhs)
 proof
 assume ?lhs
 then obtain T where $T: \bigwedge n::\text{nat}. P(T\ n) \cup (\text{range } T) = S$
 by (meson empty countable-union-of-explicit)
 have $P(\bigcup(T \cdot \{..n\}))$ for n
 by (induction n) (auto simp: atMost-Suc $Un\ T$)
 with T show ?rhs
 by (rule-tac $x=\lambda n. \bigcup_{k \leq n}. T\ k$ in exI) force
 next
 assume ?rhs
 then show ?lhs
 using empty countable-union-of-explicit by auto
 qed

lemma *countable-union-of-idem* [simp]:
countable union-of countable union-of P = countable union-of P (is ?lhs=?rhs)
proof
 fix S
 show (countable union-of countable union-of P) S = (countable union-of P) S
proof
 assume L: ?lhs S
 then obtain U where countable U and U: $U \subseteq \text{Collect } (\text{countable union-of } P)$
 $P) \bigcup U = S$
 by (metis union-of-def)
 then have $\forall U \in \mathcal{U}. \exists \mathcal{V}. \text{countable } \mathcal{V} \wedge \mathcal{V} \subseteq \text{Collect } P \wedge U = \bigcup \mathcal{V}$
 by (metis Ball-Collect union-of-def)
 then obtain F where F: $\forall U \in \mathcal{U}. \text{countable } (F \ U) \wedge F \ U \subseteq \text{Collect } P \wedge U$
 $= \bigcup (F \ U)$
 by metis
 have countable $(\bigcup (F \ ' \mathcal{U}))$
 using F ‹countable U› by blast
 moreover have $\bigcup (F \ ' \mathcal{U}) \subseteq \text{Collect } P$
 by (simp add: Sup-le-iff F)
 moreover have $\bigcup (\bigcup (F \ ' \mathcal{U})) = S$
 by auto (metis Union-iff F U(2))+
 ultimately show ?rhs S
 by (meson union-of-def)
qed (simp add: countable-union-of-inc)
qed

lemma *countable-intersection-of-idem* [simp]:
countable intersection-of countable intersection-of P =
countable intersection-of P
 by (force simp: countable-intersection-of-complement)

lemma *countable-union-of-Union*:
 $\llbracket \text{countable } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{countable union-of } P) \ S \rrbracket$
 $\implies (\text{countable union-of } P) (\bigcup \mathcal{U})$
 by (metis Ball-Collect countable-union-of-idem union-of-def)

lemma *countable-union-of-UN*:
 $\llbracket \text{countable } I; \bigwedge i. i \in I \implies (\text{countable union-of } P) (U \ i) \rrbracket$
 $\implies (\text{countable union-of } P) (\bigcup_{i \in I} U \ i)$
 by (metis (mono-tags, lifting) countable-image countable-union-of-Union imageE)

lemma *countable-union-of-Un*:
 $\llbracket (\text{countable union-of } P) \ S; (\text{countable union-of } P) \ T \rrbracket$
 $\implies (\text{countable union-of } P) (S \cup T)$
 by (smt (verit) Union-Un-distrib countable-Un le-sup-iff union-of-def)

lemma *countable-intersection-of-Inter*:
 $\llbracket \text{countable } \mathcal{U}; \bigwedge S. S \in \mathcal{U} \implies (\text{countable intersection-of } P) \ S \rrbracket$

$\implies (\text{countable intersection-of } P) (\bigcap \mathcal{U})$
by (*metis countable-intersection-of-idem intersection-of-def mem-Collect-eq subsetI*)

lemma *countable-intersection-of-INT*:

$\llbracket \text{countable } I; \bigwedge i. i \in I \implies (\text{countable intersection-of } P) (U\ i) \rrbracket$
 $\implies (\text{countable intersection-of } P) (\bigcap_{i \in I}. U\ i)$
by (*metis (mono-tags, lifting) countable-image countable-intersection-of-Inter imageE*)

lemma *countable-intersection-of-inter*:

$\llbracket (\text{countable intersection-of } P)\ S; (\text{countable intersection-of } P)\ T \rrbracket$
 $\implies (\text{countable intersection-of } P) (S \cap T)$
by (*simp add: countable-intersection-of-complement countable-union-of-Un*)

lemma *countable-union-of-Int*:

assumes S : (*countable union-of* P) S **and** T : (*countable union-of* P) T
and Int : $\bigwedge S\ T. P\ S \wedge P\ T \implies P(S \cap T)$
shows (*countable union-of* P) $(S \cap T)$

proof –

obtain \mathcal{U} **where** *countable* \mathcal{U} **and** \mathcal{U} : $\mathcal{U} \subseteq \text{Collect } P \cup \mathcal{U} = S$
using S **by** (*metis union-of-def*)
obtain \mathcal{V} **where** *countable* \mathcal{V} **and** \mathcal{V} : $\mathcal{V} \subseteq \text{Collect } P \cup \mathcal{V} = T$
using T **by** (*metis union-of-def*)
have $\bigwedge U\ V. \llbracket U \in \mathcal{U}; V \in \mathcal{V} \rrbracket \implies (\text{countable union-of } P) (U \cap V)$
using $\mathcal{U}\ \mathcal{V}$ **by** (*metis Ball-Collect countable-union-of-inc local.Int*)
then have (*countable union-of* P) $(\bigcup U \in \mathcal{U}. \bigcup V \in \mathcal{V}. U \cap V)$
by (*meson <countable* \mathcal{U} *<countable* \mathcal{V} *countable-union-of-UN*)
moreover have $S \cap T = (\bigcup U \in \mathcal{U}. \bigcup V \in \mathcal{V}. U \cap V)$
by (*simp add: $\mathcal{U}\ \mathcal{V}$*)
ultimately show *?thesis*
by *presburger*

qed

lemma *countable-intersection-of-union*:

assumes S : (*countable intersection-of* P) S **and** T : (*countable intersection-of* P) T
and Un : $\bigwedge S\ T. P\ S \wedge P\ T \implies P(S \cup T)$
shows (*countable intersection-of* P) $(S \cup T)$
by (*metis (mono-tags, lifting) Compl-Int S T Un compl-sup countable-intersection-of-complement countable-union-of-Int*)

end

96 Signed division: negative results rounded towards zero rather than minus infinity.

theory *Signed-Division*

```

imports Main
begin

class signed-divide =
  fixes signed-divide ::  $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$  (infixl  $\langle sdiv \rangle$  70)

class signed-modulo =
  fixes signed-modulo ::  $\langle 'a \Rightarrow 'a \Rightarrow 'a \rangle$  (infixl  $\langle smod \rangle$  70)

class signed-division = comm-semiring-1-cancel + signed-divide + signed-modulo
+
  assumes sdiv-mult-smod-eq:  $\langle a \text{ sdiv } b * b + a \text{ smod } b = a \rangle$ 
begin

lemma mult-sdiv-smod-eq:
   $\langle b * (a \text{ sdiv } b) + a \text{ smod } b = a \rangle$ 
  using sdiv-mult-smod-eq [of a b] by (simp add: ac-simps)

lemma smod-sdiv-mult-eq:
   $\langle a \text{ smod } b + a \text{ sdiv } b * b = a \rangle$ 
  using sdiv-mult-smod-eq [of a b] by (simp add: ac-simps)

lemma smod-mult-sdiv-eq:
   $\langle a \text{ smod } b + b * (a \text{ sdiv } b) = a \rangle$ 
  using sdiv-mult-smod-eq [of a b] by (simp add: ac-simps)

lemma minus-sdiv-mult-eq-smod:
   $\langle a - a \text{ sdiv } b * b = a \text{ smod } b \rangle$ 
  by (rule add-implies-diff [symmetric]) (fact smod-sdiv-mult-eq)

lemma minus-mult-sdiv-eq-smod:
   $\langle a - b * (a \text{ sdiv } b) = a \text{ smod } b \rangle$ 
  by (rule add-implies-diff [symmetric]) (fact smod-mult-sdiv-eq)

lemma minus-smod-eq-sdiv-mult:
   $\langle a - a \text{ smod } b = a \text{ sdiv } b * b \rangle$ 
  by (rule add-implies-diff [symmetric]) (fact sdiv-mult-smod-eq)

lemma minus-smod-eq-mult-sdiv:
   $\langle a - a \text{ smod } b = b * (a \text{ sdiv } b) \rangle$ 
  by (rule add-implies-diff [symmetric]) (fact mult-sdiv-smod-eq)

end

```

The following specification of division is named “T-division” in [2]. It is motivated by ISO C99, which in turn adopted the typical behavior of hardware modern in the beginning of the 1990ies; but note ISO C99 describes the instance on machine words, not mathematical integers.

instantiation *int* :: signed-division

begin

definition *signed-divide-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ sdiv } l = \text{sgn } k * \text{sgn } l * (|k| \text{ div } |l|) \rangle$ **for** $k \ l :: \text{int}$

definition *signed-modulo-int* :: $\langle \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \rangle$
where $\langle k \text{ smod } l = \text{sgn } k * (|k| \text{ mod } |l|) \rangle$ **for** $k \ l :: \text{int}$

instance *by standard*

(*simp add: signed-divide-int-def signed-modulo-int-def div-abs-eq mod-abs-eq algebra-simps*)

end

lemma *divide-int-eq-signed-divide-int*:
 $\langle k \text{ div } l = k \text{ sdiv } l - \text{of_bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
by (*simp add: div-eq-div-abs [of k l] signed-divide-int-def*)

lemma *signed-divide-int-eq-divide-int*:
 $\langle k \text{ sdiv } l = k \text{ div } l + \text{of_bool } (l \neq 0 \wedge \text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
by (*simp add: divide-int-eq-signed-divide-int*)

lemma *modulo-int-eq-signed-modulo-int*:
 $\langle k \text{ mod } l = k \text{ smod } l + l * \text{of_bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
by (*simp add: mod-eq-mod-abs [of k l] signed-modulo-int-def*)

lemma *signed-modulo-int-eq-modulo-int*:
 $\langle k \text{ smod } l = k \text{ mod } l - l * \text{of_bool } (\text{sgn } k \neq \text{sgn } l \wedge \neg l \text{ dvd } k) \rangle$
for $k \ l :: \text{int}$
by (*simp add: modulo-int-eq-signed-modulo-int*)

lemma *sdiv-int-div-0*:
 $(x :: \text{int}) \text{ sdiv } 0 = 0$
by (*clarsimp simp: signed-divide-int-def*)

lemma *sdiv-int-0-div* [*simp*]:
 $0 \text{ sdiv } (x :: \text{int}) = 0$
by (*clarsimp simp: signed-divide-int-def*)

lemma *smod-int-alt-def*:
 $(a :: \text{int}) \text{ smod } b = \text{sgn } (a) * (\text{abs } a \text{ mod } \text{abs } b)$
by (*fact signed-modulo-int-def*)

lemma *int-sdiv-simps* [*simp*]:
 $(a :: \text{int}) \text{ sdiv } 1 = a$
 $(a :: \text{int}) \text{ sdiv } 0 = 0$

$(a :: \text{int}) \text{ sdiv } -1 = -a$
by (*auto simp: signed-divide-int-def sgn-if*)

lemma *smod-int-mod-0* [*simp*]:
 $x \text{ smod } (0 :: \text{int}) = x$
by (*clarsimp simp: signed-modulo-int-def abs-mult-sgn ac-simps*)

lemma *smod-int-0-mod* [*simp*]:
 $0 \text{ smod } (x :: \text{int}) = 0$
by (*clarsimp simp: smod-int-alt-def*)

lemma *sgn-sdiv-eq-sgn-mult*:
 $a \text{ sdiv } b \neq 0 \implies \text{sgn } ((a :: \text{int}) \text{ sdiv } b) = \text{sgn } (a * b)$
by (*auto simp: signed-divide-int-def sgn-div-eq-sgn-mult sgn-mult*)

lemma *int-sdiv-same-is-1* [*simp*]:
assumes $a \neq 0$
shows $((a :: \text{int}) \text{ sdiv } b = a) = (b = 1)$
proof –
have $b = 1$ **if** $a \text{ sdiv } b = a$
proof –
have $b > 0$
by (*smt (verit, ccfv-threshold) assms mult-cancel-left2 sgn-if sgn-mult sgn-sdiv-eq-sgn-mult that*)
then show ?thesis
by (*smt (verit) assms dvd-eq-mod-eq-0 int-div-less-self of-bool-eq(1,2) sgn-if signed-divide-int-eq-divide-int that zdiv-zminus1-eq-if*)
qed
then show ?thesis
by auto
qed

lemma *int-sdiv-negated-is-minus1* [*simp*]:
 $a \neq 0 \implies ((a :: \text{int}) \text{ sdiv } b = -a) = (b = -1)$
using *int-sdiv-same-is-1* [*of - -b*]
using *signed-divide-int-def* **by fastforce**

lemma *sdiv-int-range*:
 $\langle a \text{ sdiv } b \in \{-|a|..|a|\} \rangle$ **for** $a \ b :: \text{int}$
using *zdiv-mono2* [*of* $\langle |a| \rangle$ 1 $\langle |b| \rangle$]
by (*cases* $\langle b = 0 \rangle$; *cases* $\langle \text{sgn } b = \text{sgn } a \rangle$)
(auto simp add: signed-divide-int-def pos-imp-zdiv-nonneg-iff dest!: sgn-not-eq-imp intro: order-trans [of - 0])

lemma *smod-int-range*:
 $\langle a \text{ smod } b \in \{-|b| + 1..|b| - 1\} \rangle$
if $\langle b \neq 0 \rangle$ **for** $a \ b :: \text{int}$
proof –
define $m \ n$ **where** $\langle m = \text{nat } |a| \rangle \ \langle n = \text{nat } |b| \rangle$

then have $\langle |a| = \text{int } m \rangle \langle |b| = \text{int } n \rangle$
by *simp-all*
with that have $\langle n > 0 \rangle$
by *simp*
with *signed-modulo-int-def* [of a b] $\langle |a| = \text{int } m \rangle \langle |b| = \text{int } n \rangle$
show *?thesis*
by (*auto simp add: sgn-if diff-le-eq int-one-le-iff-zero-less simp flip: of-nat-mod of-nat-diff*)
qed

lemma *smod-int-compares*:

$\llbracket 0 \leq a; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b < b$
 $\llbracket 0 \leq a; 0 < b \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; 0 < b \rrbracket \implies -b < (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; 0 < b \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b < -b$
 $\llbracket 0 \leq a; b < 0 \rrbracket \implies 0 \leq (a :: \text{int}) \text{ smod } b$
 $\llbracket a \leq 0; b < 0 \rrbracket \implies (a :: \text{int}) \text{ smod } b \leq 0$
 $\llbracket a \leq 0; b < 0 \rrbracket \implies b \leq (a :: \text{int}) \text{ smod } b$
using *smod-int-range* [where a=a and b=b]
by (*auto simp: add1-zle-eq smod-int-alt-def sgn-if*)

lemma *smod-mod-positive*:

$\llbracket 0 \leq (a :: \text{int}); 0 \leq b \rrbracket \implies a \text{ smod } b = a \text{ mod } b$
by (*clarsimp simp: smod-int-alt-def zsgn-def*)

lemma *minus-sdiv-eq* [simp]:

$\langle -k \text{ sdiv } l = -(k \text{ sdiv } l) \rangle$ **for** $k \ l :: \text{int}$
by (*simp add: signed-divide-int-def*)

lemma *sdiv-minus-eq* [simp]:

$\langle k \text{ sdiv } -l = -(k \text{ sdiv } l) \rangle$ **for** $k \ l :: \text{int}$
by (*simp add: signed-divide-int-def*)

lemma *sdiv-int-numeral-numeral* [simp]:

$\langle \text{numeral } m \text{ sdiv numeral } n = \text{numeral } m \text{ div } (\text{numeral } n :: \text{int}) \rangle$
by (*simp add: signed-divide-int-def*)

lemma *minus-smod-eq* [simp]:

$\langle -k \text{ smod } l = -(k \text{ smod } l) \rangle$ **for** $k \ l :: \text{int}$
by (*simp add: smod-int-alt-def*)

lemma *smod-minus-eq* [simp]:

$\langle k \text{ smod } -l = k \text{ smod } l \rangle$ **for** $k \ l :: \text{int}$
by (*simp add: smod-int-alt-def*)

lemma *smod-int-numeral-numeral* [simp]:

$\langle \text{numeral } m \text{ smod numeral } n = \text{numeral } m \text{ mod } (\text{numeral } n :: \text{int}) \rangle$
by (*simp add: smod-int-alt-def*)

end

97 State monad

theory *State-Monad*
imports *Monad-Syntax*
begin

datatype (*'s*, *'a*) *state* = *State* (*run-state*: *'s* \Rightarrow (*'a* \times *'s*))

lemma *set-state-iff*: $x \in \text{set-state } m \iff (\exists s\ s'. \text{run-state } m\ s = (x, s'))$
by (*cases m*) (*simp add: prod-set-defs eq-fst-iff*)

lemma *pred-stateI*[*intro*]:
assumes $\bigwedge a\ s\ s'. \text{run-state } m\ s = (a, s') \implies P\ a$
shows *pred-state P m*

proof (*subst state.pred-set, rule*)

fix *x*

assume $x \in \text{set-state } m$

then obtain *s s'* **where** $\text{run-state } m\ s = (x, s')$

by (*auto simp: set-state-iff*)

with *assms* **show** $P\ x$.

qed

lemma *pred-stateD*[*dest*]:
assumes *pred-state P m* $\text{run-state } m\ s = (a, s')$
shows $P\ a$

proof (*rule state.exhaust[of m]*)

fix *f*

assume $m = \text{State } f$

with *assms* **have** *pred-fun* ($\lambda_. \text{True}$) (*pred-prod P top*) *f*

by (*metis state.pred-inject*)

moreover have $f\ s = (a, s')$

using *assms* **unfolding** $\langle m = \rightarrow \rangle$ **by** *auto*

ultimately show $P\ a$

unfolding *pred-prod-beta pred-fun-def*

by (*metis fst-conv*)

qed

lemma *pred-state-run-state*: $\text{pred-state } P\ m \implies P\ (\text{fst } (\text{run-state } m\ s))$
by (*meson pred-stateD prod.exhaust-sel*)

definition *state-io-rel* :: $(\text{'s} \Rightarrow \text{'s} \Rightarrow \text{bool}) \Rightarrow (\text{'s}, \text{'a})\ \text{state} \Rightarrow \text{bool}$ **where**
state-io-rel P m = $(\forall s. P\ s\ (\text{snd } (\text{run-state } m\ s)))$

lemma *state-io-relI*[*intro*]:
assumes $\bigwedge a\ s\ s'. \text{run-state } m\ s = (a, s') \implies P\ s\ s'$
shows *state-io-rel P m*

using *assms* **unfolding** *state-io-rel-def*
by (*metis prod.collapse*)

lemma *state-io-relD[dest]*:
assumes *state-io-rel P m run-state m s = (a, s')*
shows *P s s'*
using *assms* **unfolding** *state-io-rel-def*
by (*metis snd-conv*)

lemma *state-io-rel-mono[mono]*: $P \leq Q \implies \text{state-io-rel } P \leq \text{state-io-rel } Q$
by *blast*

lemma *state-ext*:
assumes $\bigwedge s. \text{run-state } m \ s = \text{run-state } n \ s$
shows $m = n$
using *assms*
by (*cases m; cases n*) *auto*

context **begin**

qualified definition *return* :: $'a \Rightarrow ('s, 'a) \text{ state}$ **where**
return a = State (Pair a)

lemma *run-state-return[simp]*: $\text{run-state } (\text{return } x) \ s = (x, s)$
unfolding *return-def*
by *simp*

qualified definition *ap* :: $('s, 'a \Rightarrow 'b) \text{ state} \Rightarrow ('s, 'a) \text{ state} \Rightarrow ('s, 'b) \text{ state}$
where
ap f x = State ($\lambda s. \text{case run-state } f \ s \text{ of } (g, s') \Rightarrow \text{case run-state } x \ s' \text{ of } (y, s'') \Rightarrow (g \ y, s'')$)

lemma *run-state-ap[simp]*:
 $\text{run-state } (\text{ap } f \ x) \ s = (\text{case run-state } f \ s \text{ of } (g, s') \Rightarrow \text{case run-state } x \ s' \text{ of } (y, s'') \Rightarrow (g \ y, s''))$
unfolding *ap-def* **by** *auto*

qualified definition *bind* :: $('s, 'a) \text{ state} \Rightarrow ('a \Rightarrow ('s, 'b) \text{ state}) \Rightarrow ('s, 'b) \text{ state}$
where
bind x f = State ($\lambda s. \text{case run-state } x \ s \text{ of } (a, s') \Rightarrow \text{run-state } (f \ a) \ s'$)

lemma *run-state-bind[simp]*:
 $\text{run-state } (\text{bind } x \ f) \ s = (\text{case run-state } x \ s \text{ of } (a, s') \Rightarrow \text{run-state } (f \ a) \ s')$
unfolding *bind-def* **by** *auto*

adhoc-overloading *Monad-Syntax.bind* \equiv *bind*

lemma *bind-left-identity[simp]*: $\text{bind } (\text{return } a) \ f = f \ a$
unfolding *return-def bind-def* **by** *simp*

lemma *bind-right-identity*[simp]: $\text{bind } m \text{ return} = m$
unfolding *return-def bind-def* **by** *simp*

lemma *bind-assoc*[simp]: $\text{bind } (\text{bind } m f) g = \text{bind } m (\lambda x. \text{bind } (f x) g)$
unfolding *bind-def* **by** (*auto split: prod.splits*)

lemma *bind-predI*[intro]:
assumes *pred-state* $(\lambda x. \text{pred-state } P (f x)) m$
shows *pred-state* $P (\text{bind } m f)$
apply (*rule pred-stateI*)
unfolding *bind-def*
using *assms* **by** (*auto split: prod.splits*)

qualified definition *get* :: $('s, 's) \text{ state}$ **where**
get = *State* $(\lambda s. (s, s))$

lemma *run-state-get*[simp]: $\text{run-state } \text{get } s = (s, s)$
unfolding *get-def* **by** *simp*

qualified definition *set* :: $'s \Rightarrow ('s, \text{unit}) \text{ state}$ **where**
set $s' = \text{State } (\lambda -. ((), s'))$

lemma *run-state-set*[simp]: $\text{run-state } (\text{set } s') s = ((), s')$
unfolding *set-def* **by** *simp*

lemma *get-set*[simp]: $\text{bind } \text{get } \text{set} = \text{return } ()$
unfolding *bind-def get-def set-def return-def*
by *simp*

lemma *set-set*[simp]: $\text{bind } (\text{set } s) (\lambda -. \text{set } s') = \text{set } s'$
unfolding *bind-def set-def*
by *simp*

lemma *get-bind-set*[simp]: $\text{bind } \text{get } (\lambda s. \text{bind } (\text{set } s) (f s)) = \text{bind } \text{get } (\lambda s. f s ())$
unfolding *bind-def get-def set-def*
by *simp*

lemma *get-const*[simp]: $\text{bind } \text{get } (\lambda -. m) = m$
unfolding *get-def bind-def*
by *simp*

fun *traverse-list* :: $('a \Rightarrow ('b, 'c) \text{ state}) \Rightarrow 'a \text{ list} \Rightarrow ('b, 'c \text{ list}) \text{ state}$ **where**
traverse-list - [] = *return* [] |
traverse-list $f (x \# xs) = \text{do } \{$
 $x \leftarrow f x;$
 $xs \leftarrow \text{traverse-list } f xs;$
 $\text{return } (x \# xs)$
 $\}$

lemma *traverse-list-app*[simp]: *traverse-list* *f* (*xs* @ *ys*) = *do* {
 xs ← *traverse-list* *f* *xs*;
 ys ← *traverse-list* *f* *ys*;
 return (*xs* @ *ys*)
 }
by (*induction xs*) *auto*

lemma *traverse-comp*[simp]: *traverse-list* (*g* ∘ *f*) *xs* = *traverse-list* *g* (*map f xs*)
by (*induction xs*) *auto*

abbreviation *mono-state* :: ('s::preorder, 'a) *state* ⇒ *bool* **where**
mono-state ≡ *state-io-rel* (≤)

abbreviation *strict-mono-state* :: ('s::preorder, 'a) *state* ⇒ *bool* **where**
strict-mono-state ≡ *state-io-rel* (<)

corollary *strict-mono-implies-mono*: *strict-mono-state m* ⇒ *mono-state m*
unfolding *state-io-rel-def*
by (*simp add: less-imp-le*)

lemma *return-mono*[simp, intro]: *mono-state* (*return x*)
unfolding *return-def* **by** *auto*

lemma *get-mono*[simp, intro]: *mono-state* *get*
unfolding *get-def* **by** *auto*

lemma *put-mono*:
 assumes $\bigwedge x. s' \geq x$
 shows *mono-state* (*set s'*)
using *assms* **unfolding** *set-def*
by *auto*

lemma *map-mono*[intro]: *mono-state m* ⇒ *mono-state* (*map-state f m*)
by (*auto intro!*: *state-io-relI split: prod.splits simp: map-prod-def state.map-sel*)

lemma *map-strict-mono*[intro]: *strict-mono-state m* ⇒ *strict-mono-state* (*map-state f m*)
by (*auto intro!*: *state-io-relI split: prod.splits simp: map-prod-def state.map-sel*)

lemma *bind-mono-strong*:
 assumes *mono-state m*
 assumes $\bigwedge x s s'. \text{run-state } m \ s = (x, s') \Rightarrow \text{mono-state } (f \ x)$
 shows *mono-state* (*bind m f*)
unfolding *bind-def*
apply (*rule state-io-relI*)
using *assms* **by** (*auto split: prod.splits dest!: state-io-relD intro: order-trans*)

lemma *bind-strict-mono-strong1*:

assumes *mono-state m*
assumes $\bigwedge x\ s\ s'. \text{run-state } m\ s = (x, s') \implies \text{strict-mono-state } (f\ x)$
shows *strict-mono-state (bind m f)*
unfolding *bind-def*
apply (*rule state-io-relI*)
using *assms* **by** (*auto split: prod.splits dest!: state-io-relD intro: le-less-trans*)

lemma *bind-strict-mono-strong2*:
assumes *strict-mono-state m*
assumes $\bigwedge x\ s\ s'. \text{run-state } m\ s = (x, s') \implies \text{mono-state } (f\ x)$
shows *strict-mono-state (bind m f)*
unfolding *bind-def*
apply (*rule state-io-relI*)
using *assms* **by** (*auto split: prod.splits dest!: state-io-relD intro: less-le-trans*)

corollary *bind-strict-mono-strong*:
assumes *strict-mono-state m*
assumes $\bigwedge x\ s\ s'. \text{run-state } m\ s = (x, s') \implies \text{strict-mono-state } (f\ x)$
shows *strict-mono-state (bind m f)*
using *assms* **by** (*auto intro: bind-strict-mono-strong1 strict-mono-implies-mono*)

qualified definition *update* :: $('s \Rightarrow 's) \Rightarrow ('s, \text{unit}) \text{ state}$ **where**
update f = bind get (set \circ f)

lemma *update-id[simp]*: *update* $(\lambda x. x) = \text{return } ()$
unfolding *update-def return-def get-def set-def bind-def*
by *auto*

lemma *update-comp[simp]*: *bind* (*update f*) $(\lambda-. \text{update } g) = \text{update } (g \circ f)$
unfolding *update-def return-def get-def set-def bind-def*
by *auto*

lemma *set-update[simp]*: *bind* (*set s*) $(\lambda-. \text{update } f) = \text{set } (f\ s)$
unfolding *set-def update-def bind-def get-def set-def*
by *simp*

lemma *set-bind-update[simp]*: *bind* (*set s*) $(\lambda-. \text{bind } (\text{update } f)\ g) = \text{bind } (\text{set } (f\ s))\ g$
unfolding *set-def update-def bind-def get-def set-def*
by *simp*

lemma *update-mono*:
assumes $\bigwedge x. x \leq f\ x$
shows *mono-state (update f)*
using *assms* **unfolding** *update-def get-def set-def bind-def*
by (*auto intro!: state-io-relI*)

lemma *update-strict-mono*:
assumes $\bigwedge x. x < f\ x$

```

  shows strict-mono-state (update f)
using assms unfolding update-def get-def set-def bind-def
by (auto intro!: state-io-relI)

end

end

```

```

theory Comparator
  imports Main
begin

```

98 Comparators on linear quasi-orders

98.1 Basic properties

```

datatype cmp = Less | Equiv | Greater

```

```

locale comparator =
  fixes cmp :: 'a  $\Rightarrow$  'a  $\Rightarrow$  cmp'
  assumes refl [simp]: ' $\bigwedge a. \text{cmp } a \ a = \text{Equiv}$ '
    and trans-equiv: ' $\bigwedge a \ b \ c. \text{cmp } a \ b = \text{Equiv} \Longrightarrow \text{cmp } b \ c = \text{Equiv} \Longrightarrow \text{cmp } a \ c = \text{Equiv}$ '
  assumes trans-less: ' $\text{cmp } a \ b = \text{Less} \Longrightarrow \text{cmp } b \ c = \text{Less} \Longrightarrow \text{cmp } a \ c = \text{Less}$ '
    and greater-iff-sym-less: ' $\bigwedge b \ a. \text{cmp } b \ a = \text{Greater} \longleftrightarrow \text{cmp } a \ b = \text{Less}$ '
begin

```

Dual properties

```

lemma trans-greater:
  ' $\text{cmp } a \ c = \text{Greater}$ ' if ' $\text{cmp } a \ b = \text{Greater}$ ' ' $\text{cmp } b \ c = \text{Greater}$ '
  using that greater-iff-sym-less trans-less by blast

```

```

lemma less-iff-sym-greater:
  ' $\text{cmp } b \ a = \text{Less} \longleftrightarrow \text{cmp } a \ b = \text{Greater}$ '
  by (simp add: greater-iff-sym-less)

```

The equivalence part

```

lemma sym:
  ' $\text{cmp } b \ a = \text{Equiv} \longleftrightarrow \text{cmp } a \ b = \text{Equiv}$ '
  by (metis (full-types) comp.exhaust greater-iff-sym-less)

```

```

lemma reflp:
  'reflp ( $\lambda a \ b. \text{cmp } a \ b = \text{Equiv}$ )'
  by (rule reflpI) simp

```

```

lemma symp:
  'symp ( $\lambda a \ b. \text{cmp } a \ b = \text{Equiv}$ )'
  by (rule sympI) (simp add: sym)

```

lemma *transp*:

⟨*transp* ($\lambda a b. \text{cmp } a b = \text{Equiv}$)⟩
by (*rule transpI*) (*fact trans-equiv*)

lemma *equivp*:

⟨*equivp* ($\lambda a b. \text{cmp } a b = \text{Equiv}$)⟩
using *reflp sym transp* **by** (*rule equivpI*)

The strict part

lemma *irreflp-less*:

⟨*irreflp* ($\lambda a b. \text{cmp } a b = \text{Less}$)⟩
by (*rule irreflpI*) *simp*

lemma *irreflp-greater*:

⟨*irreflp* ($\lambda a b. \text{cmp } a b = \text{Greater}$)⟩
by (*rule irreflpI*) *simp*

lemma *asym-less*:

⟨*cmp* $b a \neq \text{Less}$ ⟩ **if** ⟨*cmp* $a b = \text{Less}$ ⟩
using *that greater-iff-sym-less* **by** *force*

lemma *asym-greater*:

⟨*cmp* $b a \neq \text{Greater}$ ⟩ **if** ⟨*cmp* $a b = \text{Greater}$ ⟩
using *that greater-iff-sym-less* **by** *force*

lemma *asym-less*:

⟨*asym* ($\lambda a b. \text{cmp } a b = \text{Less}$)⟩
using *irreflp-less* **by** (*auto dest: asym-less*)

lemma *asym-greater*:

⟨*asym* ($\lambda a b. \text{cmp } a b = \text{Greater}$)⟩
using *irreflp-greater* **by** (*auto dest: asym-greater*)

lemma *trans-equiv-less*:

⟨*cmp* $a c = \text{Less}$ ⟩ **if** ⟨*cmp* $a b = \text{Equiv}$ ⟩ **and** ⟨*cmp* $b c = \text{Less}$ ⟩
using *that*
by (*metis (full-types) comp.exhaust greater-iff-sym-less trans-equiv trans-less*)

lemma *trans-less-equiv*:

⟨*cmp* $a c = \text{Less}$ ⟩ **if** ⟨*cmp* $a b = \text{Less}$ ⟩ **and** ⟨*cmp* $b c = \text{Equiv}$ ⟩
using *that*
by (*metis (full-types) comp.exhaust greater-iff-sym-less trans-equiv trans-less*)

lemma *trans-equiv-greater*:

⟨*cmp* $a c = \text{Greater}$ ⟩ **if** ⟨*cmp* $a b = \text{Equiv}$ ⟩ **and** ⟨*cmp* $b c = \text{Greater}$ ⟩
using *that* **by** (*simp add: sym [of a b] greater-iff-sym-less trans-less-equiv*)

lemma *trans-greater-equiv*:

$\langle \text{cmp } a \ c = \text{Greater} \rangle$ **if** $\langle \text{cmp } a \ b = \text{Greater} \rangle$ **and** $\langle \text{cmp } b \ c = \text{Equiv} \rangle$
using that by (*simp add: sym [of b c] greater-iff-sym-less trans-equiv-less*)

lemma *transp-less*:
 $\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b = \text{Less}) \rangle$
by (*rule transpI*) (*fact trans-less*)

lemma *transp-greater*:
 $\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b = \text{Greater}) \rangle$
by (*rule transpI*) (*fact trans-greater*)

The reflexive part

lemma *reflp-not-less*:
 $\langle \text{reflp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Less}) \rangle$
by (*rule reflpI*) *simp*

lemma *reflp-not-greater*:
 $\langle \text{reflp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Greater}) \rangle$
by (*rule reflpI*) *simp*

lemma *quasisym-not-less*:
 $\langle \text{cmp } a \ b = \text{Equiv} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Less} \rangle$ **and** $\langle \text{cmp } b \ a \neq \text{Less} \rangle$
using that *comp.exhaust greater-iff-sym-less* **by** *auto*

lemma *quasisym-not-greater*:
 $\langle \text{cmp } a \ b = \text{Equiv} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Greater} \rangle$ **and** $\langle \text{cmp } b \ a \neq \text{Greater} \rangle$
using that *comp.exhaust greater-iff-sym-less* **by** *auto*

lemma *trans-not-less*:
 $\langle \text{cmp } a \ c \neq \text{Less} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Less} \rangle$ $\langle \text{cmp } b \ c \neq \text{Less} \rangle$
using that by (*metis comp.exhaust greater-iff-sym-less trans-equiv trans-less*)

lemma *trans-not-greater*:
 $\langle \text{cmp } a \ c \neq \text{Greater} \rangle$ **if** $\langle \text{cmp } a \ b \neq \text{Greater} \rangle$ $\langle \text{cmp } b \ c \neq \text{Greater} \rangle$
using that *greater-iff-sym-less trans-not-less* **by** *blast*

lemma *transp-not-less*:
 $\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Less}) \rangle$
by (*rule transpI*) (*fact trans-not-less*)

lemma *transp-not-greater*:
 $\langle \text{transp } (\lambda a \ b. \text{cmp } a \ b \neq \text{Greater}) \rangle$
by (*rule transpI*) (*fact trans-not-greater*)

Substitution under equivalences

lemma *equiv-subst-left*:
 $\langle \text{cmp } z \ y = \text{comp} \longleftrightarrow \text{cmp } x \ y = \text{comp} \rangle$ **if** $\langle \text{cmp } z \ x = \text{Equiv} \rangle$ **for** *comp*
proof –
from that have $\langle \text{cmp } x \ z = \text{Equiv} \rangle$

```

    by (simp add: sym)
  with that show ?thesis
    by (cases comp) (auto intro: trans-equiv trans-equiv-less trans-equiv-greater)
qed

```

```

lemma equiv-subst-right:
   $\langle \text{cmp } x \ z = \text{comp} \longleftrightarrow \text{cmp } x \ y = \text{comp} \rangle$  if  $\langle \text{cmp } z \ y = \text{Equiv} \rangle$  for comp
proof –
  from that have  $\langle \text{cmp } y \ z = \text{Equiv} \rangle$ 
    by (simp add: sym)
  with that show ?thesis
    by (cases comp) (auto intro: trans-equiv trans-less-equiv trans-greater-equiv)
qed

end

```

```

typedef 'a comparator =  $\langle \{ \text{cmp} :: 'a \Rightarrow 'a \Rightarrow \text{comp. comparator cmp} \} \rangle$ 
morphisms compare Abs-comparator
proof –
  have  $\langle \text{comparator } (\lambda - . \text{Equiv}) \rangle$ 
    by standard simp-all
  then show ?thesis
    by auto
qed

```

```

setup-lifting type-definition-comparator

```

```

global-interpretation compare: comparator  $\langle \text{compare cmp} \rangle$ 
  using compare [of cmp] by simp

```

```

lift-definition flat ::  $\langle 'a \text{ comparator} \rangle$ 
  is  $\langle \lambda - . \text{Equiv} \rangle$  by standard simp-all

```

```

instantiation comparator :: (linorder) default
begin

```

```

lift-definition default-comparator ::  $\langle 'a \text{ comparator} \rangle$ 
  is  $\langle \lambda x \ y. \text{if } x < y \text{ then Less else if } x > y \text{ then Greater else Equiv} \rangle$ 
  by standard (auto split: if-splits)

```

```

instance ..

```

```

end

```

```

lemma compare-default-eq-Less-iff [simp]:
   $\langle \text{compare default } x \ y = \text{Less} \longleftrightarrow x < y \rangle$ 
  by transfer simp

```

```

lemma compare-default-eq-Equiv-iff [simp]:

```

⟨compare default $x\ y = \text{Equiv} \longleftrightarrow x = y$ ⟩
by transfer auto

lemma compare-default-eq-Greater-iff [simp]:
 ⟨compare default $x\ y = \text{Greater} \longleftrightarrow x > y$ ⟩
by transfer auto

A rudimentary quickcheck setup

instantiation comparator :: (enum) equal
begin

lift-definition equal-comparator :: ⟨'a comparator \Rightarrow 'a comparator \Rightarrow bool⟩
is ⟨ $\lambda f\ g.\ \forall x \in \text{set Enum.enum. } f\ x = g\ x$ ⟩ .

instance
by (standard; transfer) (auto simp add: enum-UNIV)

end

lemma [code nbe]:
 ⟨HOL.equal (cmp :: 'a::enum comparator) cmp \longleftrightarrow True⟩
by (fact equal-refl)

lemma [code]:
 ⟨HOL.equal cmp1 cmp2 \longleftrightarrow Enum.enum-all ($\lambda x.$ compare cmp1 $x =$ compare
 cmp2 x)⟩
by transfer (simp add: enum-UNIV)

instantiation comparator :: ({linorder, typerep}) full-exhaustive
begin

definition full-exhaustive-comparator ::
 ⟨('a comparator \times (unit \Rightarrow term) \Rightarrow (bool \times term list) option)
 \Rightarrow natural \Rightarrow (bool \times term list) option⟩
where ⟨full-exhaustive-comparator $f\ s =$
 Quickcheck-Exhaustive.orelse
 (f (flat, ($\lambda u.$ Code-Evaluation.Const (STR "Comparator.flat") TYPEREPA ('a
 comparator))))
 (f (default, ($\lambda u.$ Code-Evaluation.Const (STR "HOL.default-class.default")
 TYPEREPA ('a comparator))))⟩

instance ..

end

98.2 Fundamental comparator combinators

lift-definition reversed :: ⟨'a comparator \Rightarrow 'a comparator⟩
is ⟨ $\lambda \text{cmp}\ a\ b.\ \text{cmp}\ b\ a$ ⟩

proof –


```

fix cmp :: ⟨'a ⇒ 'a ⇒ comp⟩
assume ⟨comparator cmp⟩
then interpret comparator cmp .
show ⟨comparator (λa b. cmp b a)⟩
  by standard (auto intro: trans-equiv trans-less simp: greater-iff-sym-less)
qed

```

```

lemma compare-reversed-apply [simp]:
  ⟨compare (reversed cmp) x y = compare cmp y x⟩
by transfer simp

```

```

lift-definition key :: ⟨('b ⇒ 'a) ⇒ 'a comparator ⇒ 'b comparator⟩
is ⟨λf cmp a b. cmp (f a) (f b)⟩
proof –
  fix cmp :: ⟨'a ⇒ 'a ⇒ comp⟩ and f :: ⟨'b ⇒ 'a⟩
  assume ⟨comparator cmp⟩
  then interpret comparator cmp .
  show ⟨comparator (λa b. cmp (f a) (f b))⟩
    by standard (auto intro: trans-equiv trans-less simp: greater-iff-sym-less)
qed

```

```

lemma compare-key-apply [simp]:
  ⟨compare (key f cmp) x y = compare cmp (f x) (f y)⟩
by transfer simp

```

```

lift-definition prod-lex :: ⟨'a comparator ⇒ 'b comparator ⇒ ('a × 'b) comparator⟩
is ⟨λf g (a, c) (b, d). case f a b of Less ⇒ Less | Equiv ⇒ g c d | Greater ⇒ Greater⟩
proof –
  fix f :: ⟨'a ⇒ 'a ⇒ comp⟩ and g :: ⟨'b ⇒ 'b ⇒ comp⟩
  assume ⟨comparator f⟩
  then interpret f: comparator f .
  assume ⟨comparator g⟩
  then interpret g: comparator g .
  define h where ⟨h = (λ(a, c) (b, d). case f a b of Less ⇒ Less | Equiv ⇒ g c d | Greater ⇒ Greater)⟩
  then have h-apply [simp]: ⟨h (a, c) (b, d) = (case f a b of Less ⇒ Less | Equiv ⇒ g c d | Greater ⇒ Greater)⟩ for a b c d
    by simp
  have h-equiv: ⟨h p q = Equiv ⟷ f (fst p) (fst q) = Equiv ∧ g (snd p) (snd q) = Equiv⟩ for p q
    by (cases p; cases q) (simp split: comp.split)
  have h-less: ⟨h p q = Less ⟷ f (fst p) (fst q) = Less ∨ f (fst p) (fst q) = Equiv ∧ g (snd p) (snd q) = Less⟩ for p q
    by (cases p; cases q) (simp split: comp.split)
  have h-greater: ⟨h p q = Greater ⟷ f (fst p) (fst q) = Greater ∨ f (fst p) (fst q) = Equiv ∧ g (snd p) (snd q) = Greater⟩ for p q
    by (cases p; cases q) (simp split: comp.split)
  have ⟨comparator h⟩

```

```

apply standard
apply (simp-all add: h-equiv h-less h-greater f.greater-iff-sym-less g.greater-iff-sym-less
f.sym g.sym)
apply (auto intro: f.trans-equiv g.trans-equiv f.trans-less g.trans-less f.trans-less-equiv
f.trans-equiv-less)
done
then show  $\langle \text{comparator } (\lambda(a, c) (b, d). \text{case } f \text{ a b of } \text{Less} \Rightarrow \text{Less}$ 
 $| \text{Equiv} \Rightarrow g \text{ c d}$ 
 $| \text{Greater} \Rightarrow \text{Greater}) \rangle$ 
by (simp add: h-def)
qed

```

```

lemma compare-prod-lex-apply:
 $\langle \text{compare } (\text{prod-lex } \text{cmp1 } \text{cmp2}) \text{ p q} =$ 
 $(\text{case compare } (\text{key fst } \text{cmp1}) \text{ p q of } \text{Less} \Rightarrow \text{Less} | \text{Equiv} \Rightarrow \text{compare } (\text{key snd}$ 
 $\text{cmp2}) \text{ p q} | \text{Greater} \Rightarrow \text{Greater}) \rangle$ 
by transfer (simp add: split-def)

```

98.3 Direct implementations for linear orders on selected types

```

definition comparator-bool ::  $\langle \text{bool comparator} \rangle$ 
where [simp, code-abbrev]:  $\langle \text{comparator-bool} = \text{default} \rangle$ 

```

```

lemma compare-comparator-bool [code abstract]:
 $\langle \text{compare } \text{comparator-bool} = (\lambda p \text{ q.}$ 
 $\text{if } p \text{ then if } q \text{ then } \text{Equiv} \text{ else } \text{Greater}$ 
 $\text{else if } q \text{ then } \text{Less} \text{ else } \text{Equiv}) \rangle$ 
by (auto simp add: fun-eq-iff)

```

```

definition raw-comparator-nat ::  $\langle \text{nat} \Rightarrow \text{nat} \Rightarrow \text{comp} \rangle$ 
where [simp]:  $\langle \text{raw-comparator-nat} = \text{compare default} \rangle$ 

```

```

lemma default-comparator-nat [simp, code]:
 $\langle \text{raw-comparator-nat } (0::\text{nat}) \text{ } 0 = \text{Equiv} \rangle$ 
 $\langle \text{raw-comparator-nat } (\text{Suc } m) \text{ } 0 = \text{Greater} \rangle$ 
 $\langle \text{raw-comparator-nat } 0 \text{ } (\text{Suc } n) = \text{Less} \rangle$ 
 $\langle \text{raw-comparator-nat } (\text{Suc } m) \text{ } (\text{Suc } n) = \text{raw-comparator-nat } m \text{ } n \rangle$ 
by (transfer; simp)+

```

```

definition comparator-nat ::  $\langle \text{nat comparator} \rangle$ 
where [simp, code-abbrev]:  $\langle \text{comparator-nat} = \text{default} \rangle$ 

```

```

lemma compare-comparator-nat [code abstract]:
 $\langle \text{compare } \text{comparator-nat} = \text{raw-comparator-nat} \rangle$ 
by simp

```

```

definition comparator-linordered-group ::  $\langle 'a::\text{linordered-ab-group-add comparator} \rangle$ 
where [simp, code-abbrev]:  $\langle \text{comparator-linordered-group} = \text{default} \rangle$ 

```

lemma *comparator-linordered-group* [code abstract]:

⟨compare comparator-linordered-group = (λa b.
 let c = a - b in if c < 0 then Less
 else if c = 0 then Equiv else Greater)⟩

proof (rule ext)+

fix a b :: 'a

show ⟨compare comparator-linordered-group a b =
 (let c = a - b in if c < 0 then Less
 else if c = 0 then Equiv else Greater)⟩

by (simp add: Let-def not-less) (transfer; auto)

qed

end

theory *Sorting-Algorithms*

imports *Main Multiset Comparator*

begin

99 Stably sorted lists

abbreviation (input) *stable-segment* :: ⟨'a comparator ⇒ 'a ⇒ 'a list ⇒ 'a list⟩

where ⟨stable-segment cmp x ≡ filter (λy. compare cmp x y = Equiv)⟩

fun *sorted* :: ⟨'a comparator ⇒ 'a list ⇒ bool⟩

where *sorted-Nil*: ⟨sorted cmp [] ⟷ True⟩

| *sorted-single*: ⟨sorted cmp [x] ⟷ True⟩

| *sorted-rec*: ⟨sorted cmp (y # x # xs) ⟷ compare cmp y x ≠ Greater ∧ sorted
 cmp (x # xs)⟩

lemma *sorted-ConsI*:

⟨sorted cmp (x # xs)⟩ if ⟨sorted cmp xs⟩

and ⟨∧y ys. xs = y # ys ⟹ compare cmp x y ≠ Greater⟩

using that by (cases xs) simp-all

lemma *sorted-Cons-imp-sorted*:

⟨sorted cmp xs⟩ if ⟨sorted cmp (x # xs)⟩

using that by (cases xs) simp-all

lemma *sorted-Cons-imp-not-less*:

⟨compare cmp y x ≠ Greater⟩ if ⟨sorted cmp (y # xs)⟩

and ⟨x ∈ set xs⟩

using that by (induction xs arbitrary: y) (auto dest: compare.trans-not-greater)

lemma *sorted-induct* [consumes 1, case-names Nil Cons, induct pred: sorted]:

⟨P xs⟩ if ⟨sorted cmp xs⟩ and ⟨P []⟩

and *: ⟨∧x xs. sorted cmp xs ⟹ P xs

⟹ (∧y. y ∈ set xs ⟹ compare cmp x y ≠ Greater) ⟹ P (x # xs)⟩

```

using ⟨sorted cmp xs⟩ proof (induction xs)
  case Nil
  show ?case
  by (rule ⟨P []⟩)
next
  case (Cons x xs)
  from ⟨sorted cmp (x # xs)⟩ have ⟨sorted cmp xs⟩
  by (cases xs) simp-all
  moreover have ⟨P xs⟩ using ⟨sorted cmp xs⟩
  by (rule Cons.IH)
  moreover have ⟨compare cmp x y ≠ Greater⟩ if ⟨y ∈ set xs⟩ for y
  using that ⟨sorted cmp (x # xs)⟩ proof (induction xs)
    case Nil
    then show ?case
    by simp
  next
    case (Cons z zs)
    then show ?case
    proof (cases zs)
      case Nil
      with Cons.prem1 show ?thesis
      by simp
    next
      case (Cons w ws)
      with Cons.prem1 have ⟨compare cmp z w ≠ Greater⟩ ⟨compare cmp x z ≠
Greater⟩
      by auto
      then have ⟨compare cmp x w ≠ Greater⟩
      by (auto dest: compare.trans-not-greater)
      with Cons show ?thesis
      using Cons.prem1 Cons.IH by auto
    qed
  qed
  ultimately show ?case
  by (rule *)
qed

```

lemma sorted-induct-remove1 [consumes 1, case-names Nil minimum]:

```

  ⟨P xs⟩ if ⟨sorted cmp xs⟩ and ⟨P []⟩
  and *: ⟨ $\bigwedge x xs.$  sorted cmp xs  $\implies$  P (remove1 x xs)
     $\implies x \in \text{set } xs \implies \text{hd } (\text{stable-segment cmp } x xs) = x \implies (\bigwedge y. y \in \text{set } xs \implies$ 
compare cmp x y ≠ Greater)
     $\implies P xs$ ⟩

```

using ⟨sorted cmp xs⟩ **proof** (induction xs)

```

  case Nil
  show ?case
  by (rule ⟨P []⟩)
next

```

```

  case (Cons x xs)

```

```

then have ⟨sorted cmp (x # xs)⟩
  by (simp add: sorted-ConsI)
moreover note Cons.IH
moreover have ⟨ $\bigwedge y. \text{compare cmp } x \ y = \text{Greater} \implies y \in \text{set } xs \implies \text{False}$ ⟩
  using Cons.hyps by simp
ultimately show ?case
  by (auto intro!: * [of ⟨x # xs⟩ x]) blast
qed

```

```

lemma sorted-remove1:
  ⟨sorted cmp (remove1 x xs)⟩ if ⟨sorted cmp xs⟩
proof (cases ⟨x ∈ set xs⟩)
  case False
  with that show ?thesis
    by (simp add: remove1-idem)
next
  case True
  with that show ?thesis proof (induction xs)
    case Nil
    then show ?case
      by simp
  next
    case (Cons y ys)
    show ?case proof (cases ⟨x = y⟩)
      case True
      with Cons.hyps show ?thesis
        by simp
    next
      case False
      then have ⟨sorted cmp (remove1 x ys)⟩
        using Cons.IH Cons.prem by auto
      then have ⟨sorted cmp (y # remove1 x ys)⟩
      proof (rule sorted-ConsI)
        fix z zs
        assume ⟨remove1 x ys = z # zs⟩
        with ⟨x ≠ y⟩ have ⟨z ∈ set ys⟩
          using notin-set-remove1 [of z ys x] by auto
        then show ⟨compare cmp y z ≠ Greater⟩
          by (rule Cons.hyps(2))
      qed
    with False show ?thesis
      by simp
  qed
qed
qed
qed

```

```

lemma sorted-stable-segment:
  ⟨sorted cmp (stable-segment cmp x xs)⟩
proof (induction xs)

```

```

case Nil
show ?case
  by simp
next
case (Cons y ys)
then show ?case
  by (auto intro!: sorted-ConsI simp add: filter-eq-Cons-iff compare.sym)
    (auto dest: compare.trans-equiv simp add: compare.sym compare.greater-iff-sym-less)

```

qed

```

primrec insort :: ⟨'a comparator ⇒ 'a ⇒ 'a list ⇒ 'a list⟩
  where ⟨insort cmp y [] = [y]⟩
  | ⟨insort cmp y (x # xs) = (if compare cmp y x ≠ Greater
    then y # x # xs
    else x # insort cmp y xs)⟩

```

```

lemma mset-insort [simp]:
  ⟨mset (insort cmp x xs) = add-mset x (mset xs)⟩
by (induction xs) simp-all

```

```

lemma length-insort [simp]:
  ⟨length (insort cmp x xs) = Suc (length xs)⟩
by (induction xs) simp-all

```

```

lemma sorted-insort:
  ⟨sorted cmp (insort cmp x xs)⟩ if ⟨sorted cmp xs⟩
using that proof (induction xs)
  case Nil
  then show ?case
    by simp
next
case (Cons y ys)
then show ?case by (cases ys)
  (auto, simp-all add: compare.greater-iff-sym-less)
qed

```

```

lemma stable-insort-equiv:
  ⟨stable-segment cmp y (insort cmp x xs) = x # stable-segment cmp y xs⟩
  if ⟨compare cmp y x = Equiv⟩
proof (induction xs)
  case Nil
  from that show ?case
    by simp
next
case (Cons z xs)
  moreover from that have ⟨compare cmp y z = Equiv ⇒ compare cmp z x =
    Equiv⟩
  by (auto intro: compare.trans-equiv simp add: compare.sym)

```

ultimately show ?case
 using that by (auto simp add: compare.greater-iff-sym-less)
 qed

lemma stable-insort-not-equiv:
 $\langle \text{stable-segment cmp } y (\text{insort cmp } x \text{ } xs) = \text{stable-segment cmp } y \text{ } xs \rangle$
 if $\langle \text{compare cmp } y \text{ } x \neq \text{Equiv} \rangle$
 using that by (induction xs) simp-all

lemma remove1-insort-same-eq [simp]:
 $\langle \text{remove1 } x (\text{insort cmp } x \text{ } xs) = xs \rangle$
 by (induction xs) simp-all

lemma insort-eq-ConsI:
 $\langle \text{insort cmp } x \text{ } xs = x \# xs \rangle$
 if $\langle \text{sorted cmp } xs \rangle \wedge \langle y. y \in \text{set } xs \implies \text{compare cmp } x \text{ } y \neq \text{Greater} \rangle$
 using that by (induction xs) (simp-all add: compare.greater-iff-sym-less)

lemma remove1-insort-not-same-eq [simp]:
 $\langle \text{remove1 } y (\text{insort cmp } x \text{ } xs) = \text{insort cmp } x (\text{remove1 } y \text{ } xs) \rangle$
 if $\langle \text{sorted cmp } xs \rangle \wedge \langle x \neq y \rangle$
 using that proof (induction xs)
 case Nil
 then show ?case
 by simp
 next
 case (Cons z zs)
 show ?case
 proof (cases $\langle \text{compare cmp } x \text{ } z = \text{Greater} \rangle$)
 case True
 with Cons show ?thesis
 by simp
 next
 case False
 then have $\langle \text{compare cmp } x \text{ } y \neq \text{Greater} \rangle$ if $\langle y \in \text{set } zs \rangle$ for y
 using that Cons.hyps
 by (auto dest: compare.trans-not-greater)
 with Cons show ?thesis
 by (simp add: insort-eq-ConsI)
 qed
 qed

lemma insort-remove1-same-eq:
 $\langle \text{insort cmp } x (\text{remove1 } x \text{ } xs) = xs \rangle$
 if $\langle \text{sorted cmp } xs \rangle$ and $\langle x \in \text{set } xs \rangle$ and $\langle \text{hd } (\text{stable-segment cmp } x \text{ } xs) = x \rangle$
 using that proof (induction xs)
 case Nil
 then show ?case
 by simp

```

next
  case (Cons y ys)
  then have ⟨compare cmp x y ≠ Less⟩
    by (auto simp add: compare.greater-iff-sym-less)
  then consider ⟨compare cmp x y = Greater⟩ | ⟨compare cmp x y = Equiv⟩
    by (cases ⟨compare cmp x y⟩) auto
  then show ?case proof cases
    case 1
    with Cons.prem1 Cons.IH show ?thesis
      by auto
    next
    case 2
    with Cons.prem2 have ⟨x = y⟩
      by simp
    with Cons.hyps show ?thesis
      by (simp add: insort-eq-ConsI)
  qed
qed

lemma sorted-append-iff:
  ⟨sorted cmp (xs @ ys) ⟷ sorted cmp xs ∧ sorted cmp ys
  ∧ (∀ x ∈ set xs. ∀ y ∈ set ys. compare cmp x y ≠ Greater)⟩ (is ⟨?P ⟷ ?R ∧
  ?S ∧ ?Q⟩)
proof
  assume ?P
  have ?R
    using ⟨?P⟩ by (induction xs)
    (auto simp add: sorted-Cons-imp-not-less,
    auto simp add: sorted-Cons-imp-sorted intro: sorted-ConsI)
  moreover have ?S
    using ⟨?P⟩ by (induction xs) (auto dest: sorted-Cons-imp-sorted)
  moreover have ?Q
    using ⟨?P⟩ by (induction xs) (auto simp add: sorted-Cons-imp-not-less,
    simp add: sorted-Cons-imp-sorted)
  ultimately show ⟨?R ∧ ?S ∧ ?Q⟩
    by simp
next
  assume ⟨?R ∧ ?S ∧ ?Q⟩
  then have ?R ?S ?Q
    by simp-all
  then show ?P
    by (induction xs)
    (auto simp add: append-eq-Cons-conv intro!: sorted-ConsI)
qed

definition sort :: ⟨'a comparator ⇒ 'a list ⇒ 'a list⟩
  where ⟨sort cmp xs = foldr (insort cmp) xs []⟩

lemma sort-simps [simp]:

```


$\langle \text{sort cmp } [] = [] \rangle$
 $\langle \text{sort cmp } (x \# xs) = \text{insort cmp } x (\text{sort cmp } xs) \rangle$
by (*simp-all add: sort-def*)

lemma *mset-sort* [*simp*]:
 $\langle \text{mset } (\text{sort cmp } xs) = \text{mset } xs \rangle$
by (*induction xs*) *simp-all*

lemma *length-sort* [*simp*]:
 $\langle \text{length } (\text{sort cmp } xs) = \text{length } xs \rangle$
by (*induction xs*) *simp-all*

lemma *sorted-sort* [*simp*]:
 $\langle \text{sorted cmp } (\text{sort cmp } xs) \rangle$
by (*induction xs*) (*simp-all add: sorted-insort*)

lemma *stable-sort*:
 $\langle \text{stable-segment cmp } x (\text{sort cmp } xs) = \text{stable-segment cmp } x xs \rangle$
by (*induction xs*) (*simp-all add: stable-insort-equiv stable-insort-not-equiv*)

lemma *sort-remove1-eq* [*simp*]:
 $\langle \text{sort cmp } (\text{remove1 } x xs) = \text{remove1 } x (\text{sort cmp } xs) \rangle$
by (*induction xs*) *simp-all*

lemma *set-insort* [*simp*]:
 $\langle \text{set } (\text{insort cmp } x xs) = \text{insert } x (\text{set } xs) \rangle$
by (*induction xs*) *auto*

lemma *set-sort* [*simp*]:
 $\langle \text{set } (\text{sort cmp } xs) = \text{set } xs \rangle$
by (*induction xs*) *auto*

lemma *sort-eqI*:
 $\langle \text{sort cmp } ys = xs \rangle$
if *permutation*: $\langle \text{mset } ys = \text{mset } xs \rangle$
and *sorted*: $\langle \text{sorted cmp } xs \rangle$
and *stable*: $\langle \bigwedge y. y \in \text{set } ys \implies$
 $\text{stable-segment cmp } y ys = \text{stable-segment cmp } y xs \rangle$

proof –

have *stable'*: $\langle \text{stable-segment cmp } y ys =$
 $\text{stable-segment cmp } y xs \rangle$ **for** *y*

proof (*cases* $\langle \exists x \in \text{set } ys. \text{compare cmp } y x = \text{Equiv} \rangle$)

case *True*

then obtain *z* **where** $\langle z \in \text{set } ys \rangle$ **and** $\langle \text{compare cmp } y z = \text{Equiv} \rangle$

by *auto*

then have $\langle \text{compare cmp } y x = \text{Equiv} \iff \text{compare cmp } z x = \text{Equiv} \rangle$ **for** *x*

by (*meson compare.sym compare.trans-equiv*)

moreover have $\langle \text{stable-segment cmp } z ys =$

$\text{stable-segment cmp } z xs \rangle$

```

    using ⟨ $z \in \text{set } ys$ ⟩ by (rule stable)
  ultimately show ?thesis
    by simp
next
  case False
  moreover from permutation have ⟨ $\text{set } ys = \text{set } xs$ ⟩
    by (rule mset-eq-setD)
  ultimately show ?thesis
    by simp
qed
show ?thesis
using sorted permutation stable' proof (induction xs arbitrary: ys rule: sorted-induct-remove1)
  case Nil
  then show ?case
    by simp
next
  case (minimum x xs)
  from ⟨ $\text{mset } ys = \text{mset } xs$ ⟩ have ys: ⟨ $\text{set } ys = \text{set } xs$ ⟩
    by (rule mset-eq-setD)
  then have ⟨ $\text{compare cmp } x \ y \neq \text{Greater}$ ⟩ if ⟨ $y \in \text{set } ys$ ⟩ for y
    using that minimum.hyps by simp
  from minimum.prem1 have stable: ⟨ $\text{stable-segment cmp } x \ ys = \text{stable-segment}$ 
     $\text{cmp } x \ xs$ ⟩
    by simp
  have ⟨ $\text{sort cmp } (\text{remove1 } x \ ys) = \text{remove1 } x \ xs$ ⟩
    by (rule minimum.IH) (simp-all add: minimum.prem1 filter-remove1)
  then have ⟨ $\text{remove1 } x \ (\text{sort cmp } ys) = \text{remove1 } x \ xs$ ⟩
    by simp
  then have ⟨ $\text{insort cmp } x \ (\text{remove1 } x \ (\text{sort cmp } ys)) =$ 
     $\text{insort cmp } x \ (\text{remove1 } x \ xs)$ ⟩
    by simp
  also from minimum.hyps ys stable have ⟨ $\text{insort cmp } x \ (\text{remove1 } x \ (\text{sort cmp}$ 
     $ys)) = \text{sort cmp } ys$ ⟩
    by (simp add: stable-sort insort-remove1-same-eq)
  also from minimum.hyps have ⟨ $\text{insort cmp } x \ (\text{remove1 } x \ xs) = xs$ ⟩
    by (simp add: insort-remove1-same-eq)
  finally show ?case .
qed
qed

lemma filter-insort:
  ⟨ $\text{filter } P \ (\text{insort cmp } x \ xs) = \text{insort cmp } x \ (\text{filter } P \ xs)$ ⟩
  if ⟨ $\text{sorted cmp } xs$ ⟩ and ⟨ $P \ x$ ⟩
  using that by (induction xs)
    (auto simp add: compare.trans-not-greater insort-eq-ConsI)

lemma filter-insort-triv:
  ⟨ $\text{filter } P \ (\text{insort cmp } x \ xs) = \text{filter } P \ xs$ ⟩
  if ⟨ $\neg P \ x$ ⟩

```

using that by (induction xs) simp-all

lemma filter-sort:

⟨filter P (sort cmp xs) = sort cmp (filter P xs)⟩
 by (induction xs) (auto simp add: filter-insort filter-insort-triv)

100 Alternative sorting algorithms

100.1 Quicksort

definition quicksort :: ⟨'a comparator ⇒ 'a list ⇒ 'a list⟩

where quicksort-is-sort [simp]: ⟨quicksort = sort⟩

lemma sort-by-quicksort:

⟨sort = quicksort⟩
 by simp

lemma sort-by-quicksort-rec:

⟨sort cmp xs = sort cmp [x ← xs. compare cmp x (xs ! (length xs div 2)) = Less]
 @ stable-segment cmp (xs ! (length xs div 2)) xs
 @ sort cmp [x ← xs. compare cmp x (xs ! (length xs div 2)) = Greater]⟩ (is ⟨- =
 ?rhs⟩)

proof (rule sort-eqI)

show ⟨mset xs = mset ?rhs⟩

by (rule multiset-eqI) (auto simp add: compare.sym intro: comp.exhaust)

next

show ⟨sorted cmp ?rhs⟩

by (auto simp add: sorted-append-iff sorted-stable-segment compare.equiv-subst-right
 dest: compare.trans-greater)

next

let ?pivot = ⟨xs ! (length xs div 2)⟩

fix l

have ⟨compare cmp x ?pivot = comp ∧ compare cmp l x = Equiv

⟷ compare cmp l ?pivot = comp ∧ compare cmp l x = Equiv⟩ for x comp

proof -

have ⟨compare cmp x ?pivot = comp ⟷ compare cmp l ?pivot = comp⟩

if ⟨compare cmp l x = Equiv⟩

using that by (simp add: compare.equiv-subst-left compare.sym)

then show ?thesis by blast

qed

then show ⟨stable-segment cmp l xs = stable-segment cmp l ?rhs⟩

by (simp add: stable-sort compare.sym [of - ?pivot])

(cases ⟨compare cmp l ?pivot⟩, simp-all)

qed

context

begin

qualified definition partition :: ⟨'a comparator ⇒ 'a ⇒ 'a list ⇒ 'a list × 'a list

× 'a list›
where ⟨partition cmp pivot xs =
 ([x ← xs. compare cmp x pivot = Less], stable-segment cmp pivot xs, [x ← xs.
 compare cmp x pivot = Greater])⟩

qualified lemma partition-code [code]:

⟨partition cmp pivot [] = ([], [], [])⟩
 ⟨partition cmp pivot (x # xs) =
 (let (lts, eqs, gts) = partition cmp pivot xs
 in case compare cmp x pivot of
 Less ⇒ (x # lts, eqs, gts)
 | Equiv ⇒ (lts, x # eqs, gts)
 | Greater ⇒ (lts, eqs, x # gts))⟩
using comp.exhaust **by** (auto simp add: partition-def Let-def compare.sym [of -
 pivot])

lemma quicksort-code [code]:

⟨quicksort cmp xs =
 (case xs of
 [] ⇒ []
 | [x] ⇒ xs
 | [x, y] ⇒ (if compare cmp x y ≠ Greater then xs else [y, x])
 | - ⇒
 let (lts, eqs, gts) = partition cmp (xs ! (length xs div 2)) xs
 in quicksort cmp lts @ eqs @ quicksort cmp gts)⟩

proof (cases ⟨length xs ≥ 3⟩)

case False

then have ⟨length xs ∈ {0, 1, 2}⟩

by (auto simp add: not-le le-less less-antisym)

then consider ⟨xs = []⟩ | x **where** ⟨xs = [x]⟩ | x y **where** ⟨xs = [x, y]⟩

by (auto simp add: length-Suc-conv numeral-2-eq-2)

then show ?thesis

by cases simp-all

next

case True

then obtain x y z zs **where** ⟨xs = x # y # z # zs⟩

by (metis le-0-eq length-0-conv length-Cons list.exhaust not-less-eq-eq numeral-3-eq-3)

moreover have ⟨quicksort cmp xs =

(let (lts, eqs, gts) = partition cmp (xs ! (length xs div 2)) xs

in quicksort cmp lts @ eqs @ quicksort cmp gts)⟩

using sort-by-quicksort-rec [of cmp xs] **by** (simp add: partition-def)

ultimately show ?thesis

by simp

qed

end

100.2 Mergesort

definition *mergesort* :: $\langle 'a \text{ comparator} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$
where *mergesort-is-sort* [*simp*]: $\langle \text{mergesort} = \text{sort} \rangle$

lemma *sort-by-mergesort*:
 $\langle \text{sort} = \text{mergesort} \rangle$
by *simp*

context
fixes *cmp* :: $\langle 'a \text{ comparator} \rangle$
begin

qualified function *merge* :: $\langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \rangle$
where $\langle \text{merge } [] \text{ } ys = ys \rangle$
 $\mid \langle \text{merge } xs \text{ } [] = xs \rangle$
 $\mid \langle \text{merge } (x \# xs) (y \# ys) = (\text{if compare cmp } x \text{ } y = \text{Greater}$
 $\text{then } y \# \text{merge } (x \# xs) \text{ } ys \text{ else } x \# \text{merge } xs (y \# ys)) \rangle$
by *pat-completeness auto*

qualified termination by *lexicographic-order*

lemma *mset-merge*:
 $\langle \text{mset } (\text{merge } xs \text{ } ys) = \text{mset } xs + \text{mset } ys \rangle$
by (*induction xs ys rule: merge.induct*) *simp-all*

lemma *merge-eq-Cons-imp*:
 $\langle xs \neq [] \wedge z = \text{hd } xs \vee ys \neq [] \wedge z = \text{hd } ys \rangle$
if $\langle \text{merge } xs \text{ } ys = z \# zs \rangle$
using that by (*induction xs ys rule: merge.induct*) (*auto split: if-splits*)

lemma *filter-merge*:
 $\langle \text{filter } P (\text{merge } xs \text{ } ys) = \text{merge } (\text{filter } P \text{ } xs) (\text{filter } P \text{ } ys) \rangle$
if $\langle \text{sorted cmp } xs \rangle$ **and** $\langle \text{sorted cmp } ys \rangle$
using that proof (*induction xs ys rule: merge.induct*)
case (1 *ys*)
then show ?*case*
by *simp*
next
case (2 *xs*)
then show ?*case*
by *simp*
next
case (3 *x xs y ys*)
show ?*case*
proof (*cases* $\langle \text{compare cmp } x \text{ } y = \text{Greater} \rangle$)
case *True*
with 3 **have** *hyp*: $\langle \text{filter } P (\text{merge } (x \# xs) \text{ } ys) =$
 $\text{merge } (\text{filter } P (x \# xs)) (\text{filter } P \text{ } ys) \rangle$
by (*simp add: sorted-Cons-imp-sorted*)

```

show ?thesis
proof (cases ⟨ $\neg P\ x \wedge P\ y$ ⟩)
  case False
  with ⟨compare cmp x y = Greater⟩ show ?thesis
    by (auto simp add: hyp)
next
  case True
  from ⟨compare cmp x y = Greater⟩ 3.prem
  have *: ⟨compare cmp z y = Greater⟩ if ⟨ $z \in \text{set } (\text{filter } P\ xs)$ ⟩ for z
  using that by (auto dest: compare.trans-not-greater sorted-Cons-imp-not-less)
  from ⟨compare cmp x y = Greater⟩ show ?thesis
    by (cases ⟨filter P xs⟩) (simp-all add: hyp *)
qed
next
  case False
  with 3 have hyp: ⟨filter P (merge xs (y # ys)) =
    merge (filter P xs) (filter P (y # ys))⟩
    by (simp add: sorted-Cons-imp-sorted)
  show ?thesis
  proof (cases ⟨ $P\ x \wedge \neg P\ y$ ⟩)
    case False
    with ⟨compare cmp x y  $\neq$  Greater⟩ show ?thesis
      by (auto simp add: hyp)
    next
      case True
      from ⟨compare cmp x y  $\neq$  Greater⟩ 3.prem
      have *: ⟨compare cmp x z  $\neq$  Greater⟩ if ⟨ $z \in \text{set } (\text{filter } P\ ys)$ ⟩ for z
      using that by (auto dest: compare.trans-not-greater sorted-Cons-imp-not-less)
      from ⟨compare cmp x y  $\neq$  Greater⟩ show ?thesis
        by (cases ⟨filter P ys⟩) (simp-all add: hyp *)
      qed
    qed
  qed
lemma sorted-merge:
  ⟨sorted cmp (merge xs ys)⟩ if ⟨sorted cmp xs⟩ and ⟨sorted cmp ys⟩
using that proof (induction xs ys rule: merge.induct)
  case (1 ys)
  then show ?case
    by simp
  next
    case (2 xs)
    then show ?case
      by simp
  next
    case (3 x xs y ys)
    show ?case
    proof (cases ⟨compare cmp x y = Greater⟩)
      case True

```

```

with  $\exists$  have  $\langle \text{sorted cmp } (\text{merge } (x \# xs) \text{ } ys) \rangle$ 
  by (simp add: sorted-Cons-imp-sorted)
then have  $\langle \text{sorted cmp } (y \# \text{merge } (x \# xs) \text{ } ys) \rangle$ 
proof (rule sorted-ConsI)
  fix  $z \text{ } zs$ 
  assume  $\langle \text{merge } (x \# xs) \text{ } ys = z \# zs \rangle$ 
  with  $\exists(4)$  True show  $\langle \text{compare cmp } y \text{ } z \neq \text{Greater} \rangle$ 
    by (clarsimp simp add: sorted-Cons-imp-sorted dest!: merge-eq-Cons-imp)
      (auto simp add: compare.asym-greater sorted-Cons-imp-not-less)
  qed
with True show ?thesis
  by simp
next
case False
with  $\exists$  have  $\langle \text{sorted cmp } (\text{merge } xs \text{ } (y \# ys)) \rangle$ 
  by (simp add: sorted-Cons-imp-sorted)
then have  $\langle \text{sorted cmp } (x \# \text{merge } xs \text{ } (y \# ys)) \rangle$ 
proof (rule sorted-ConsI)
  fix  $z \text{ } zs$ 
  assume  $\langle \text{merge } xs \text{ } (y \# ys) = z \# zs \rangle$ 
  with  $\exists(3)$  False show  $\langle \text{compare cmp } x \text{ } z \neq \text{Greater} \rangle$ 
    by (clarsimp simp add: sorted-Cons-imp-sorted dest!: merge-eq-Cons-imp)
      (auto simp add: compare.asym-greater sorted-Cons-imp-not-less)
  qed
with False show ?thesis
  by simp
qed
qed

lemma merge-eq-appendI:
   $\langle \text{merge } xs \text{ } ys = xs @ ys \rangle$ 
  if  $\langle \bigwedge x y. x \in \text{set } xs \implies y \in \text{set } ys \implies \text{compare cmp } x \text{ } y \neq \text{Greater} \rangle$ 
using that by (induction xs ys rule: merge.induct) simp-all

lemma merge-stable-segments:
   $\langle \text{merge } (\text{stable-segment cmp } l \text{ } xs) \text{ } (\text{stable-segment cmp } l \text{ } ys) =$ 
     $\text{stable-segment cmp } l \text{ } xs @ \text{stable-segment cmp } l \text{ } ys \rangle$ 
by (rule merge-eq-appendI) (auto dest: compare.trans-equiv-greater)

lemma sort-by-mergesort-rec:
   $\langle \text{sort cmp } xs =$ 
     $\text{merge } (\text{sort cmp } (\text{take } (\text{length } xs \text{ div } 2) \text{ } xs))$ 
     $(\text{sort cmp } (\text{drop } (\text{length } xs \text{ div } 2) \text{ } xs)) \rangle$  (is  $\langle - = ?rhs \rangle$ )
proof (rule sort-eqI)
  have  $\langle \text{mset } (\text{take } (\text{length } xs \text{ div } 2) \text{ } xs) + \text{mset } (\text{drop } (\text{length } xs \text{ div } 2) \text{ } xs) =$ 
     $\text{mset } (\text{take } (\text{length } xs \text{ div } 2) \text{ } xs @ \text{drop } (\text{length } xs \text{ div } 2) \text{ } xs) \rangle$ 
  by (simp only: mset-append)
  then show  $\langle \text{mset } xs = \text{mset } ?rhs \rangle$ 
  by (simp add: mset-merge)

```

```

next
  show ⟨sorted cmp ?rhs⟩
  by (simp add: sorted-merge)
next
  fix l
  have ⟨stable-segment cmp l (take (length xs div 2) xs) @ stable-segment cmp l
(drop (length xs div 2) xs)
  = stable-segment cmp l xs⟩
  by (simp only: filter-append [symmetric] append-take-drop-id)
  have ⟨merge (stable-segment cmp l (take (length xs div 2) xs))
(stable-segment cmp l (drop (length xs div 2) xs)) =
  stable-segment cmp l (take (length xs div 2) xs) @ stable-segment cmp l (drop
(length xs div 2) xs)⟩
  by (rule merge-eq-appendI) (auto simp add: compare.trans-equiv-greater)
  also have ⟨... = stable-segment cmp l xs⟩
  by (simp only: filter-append [symmetric] append-take-drop-id)
  finally show ⟨stable-segment cmp l xs = stable-segment cmp l ?rhs⟩
  by (simp add: stable-sort filter-merge)
qed

lemma mergesort-code [code]:
  ⟨mergesort cmp xs =
    (case xs of
      [] ⇒ []
    | [x] ⇒ xs
    | [x, y] ⇒ (if compare cmp x y ≠ Greater then xs else [y, x])
    | - ⇒
      let
        half = length xs div 2;
        ys = take half xs;
        zs = drop half xs
      in merge (mergesort cmp ys) (mergesort cmp zs))⟩
proof (cases ⟨length xs ≥ 3⟩)
  case False
  then have ⟨length xs ∈ {0, 1, 2}⟩
  by (auto simp add: not-le le-less less-antisym)
  then consider ⟨xs = []⟩ | x where ⟨xs = [x]⟩ | x y where ⟨xs = [x, y]⟩
  by (auto simp add: length-Suc-conv numeral-2-eq-2)
  then show ?thesis
  by cases simp-all
next
  case True
  then obtain x y z zs where ⟨xs = x # y # z # zs⟩
  by (metis le-0-eq length-0-conv length-Cons list.exhaust not-less-eq-eq numeral-3-eq-3)
  moreover have ⟨mergesort cmp xs =
    (let
      half = length xs div 2;
      ys = take half xs;
      zs = drop half xs

```



```

      in merge (mergesort cmp ys) (mergesort cmp zs))
    using sort-by-mergesort-rec [of xs] by (simp add: Let-def)
    ultimately show ?thesis
      by simp
qed

end

```

100.3 Lexicographic products

lemma *sorted-prod-lex-imp-sorted-fst*:

```

  ⟨sorted (key fst cmp1) ps⟩ if ⟨sorted (prod-lex cmp1 cmp2) ps⟩
using that proof (induction rule: sorted-induct)
  case Nil
  then show ?case
    by simp
next
  case (Cons p ps)
  have ⟨compare (key fst cmp1) p q ≠ Greater⟩ if ⟨ps = q # qs⟩ for q qs
    using that Cons.hyps(2) [of q] by (simp add: compare-prod-lex-apply split:
comp.splits)
  with Cons.IH show ?case
    by (rule sorted-ConsI) simp
qed

```

lemma *sorted-prod-lex-imp-sorted-snd*:

```

  ⟨sorted (key snd cmp2) ps⟩ if ⟨sorted (prod-lex cmp1 cmp2) ps⟩ ⟨ $\bigwedge a' b'. (a', b') \in \text{set } ps \implies \text{compare cmp1 } a \ a' = \text{Equiv}$ ⟩
using that proof (induction rule: sorted-induct)
  case Nil
  then show ?case
    by simp
next
  case (Cons p ps)
  then show ?case
    apply (cases p)
    apply (rule sorted-ConsI)
    apply (simp-all add: compare-prod-lex-apply)
    apply (auto cong del: comp.case-cong-weak)
    apply (metis comp.simps(8) compare.equiv-subst-left)
    done
qed

```

lemma *sort-comp-fst-snd-eq-sort-prod-lex*:

```

  ⟨sort (key fst cmp1) ⟩ ∘ sort (key snd cmp2) = sort (prod-lex cmp1 cmp2)⟩ (is
  ⟨sort ?cmp1 ∘ sort ?cmp2 = sort ?cmp⟩)
proof
  fix ps :: ⟨('a × 'b) list⟩
  have ⟨sort ?cmp1 (sort ?cmp2 ps) = sort ?cmp ps⟩

```

```

proof (rule sort-eqI)
  show  $\langle \text{mset} (\text{sort } ?\text{cmp2 } ps) = \text{mset} (\text{sort } ?\text{cmp } ps) \rangle$ 
    by simp
  show  $\langle \text{sorted } ?\text{cmp1 } (\text{sort } ?\text{cmp } ps) \rangle$ 
    by (rule sorted-prod-lex-imp-sorted-fst [of - cmp2]) simp
next
  fix  $p :: \langle 'a \times 'b \rangle$ 
  define  $a \ b$  where  $ab: \langle a = \text{fst } p \rangle \langle b = \text{snd } p \rangle$ 
  moreover assume  $\langle p \in \text{set} (\text{sort } ?\text{cmp2 } ps) \rangle$ 
  ultimately have  $\langle (a, b) \in \text{set} (\text{sort } ?\text{cmp2 } ps) \rangle$ 
    by simp
  let  $?qs = \langle \text{filter } (\lambda(a', -). \text{compare } \text{cmp1 } a \ a' = \text{Equiv}) \ ps \rangle$ 
  have  $\langle \text{sort } ?\text{cmp2 } ?qs = \text{sort } ?\text{cmp } ?qs \rangle$ 
  proof (rule sort-eqI)
    show  $\langle \text{mset } ?qs = \text{mset} (\text{sort } ?\text{cmp } ?qs) \rangle$ 
      by simp
    show  $\langle \text{sorted } ?\text{cmp2 } (\text{sort } ?\text{cmp } ?qs) \rangle$ 
      by (rule sorted-prod-lex-imp-sorted-snd) auto
  next
    fix  $q :: \langle 'a \times 'b \rangle$ 
    define  $c \ d$  where  $\langle c = \text{fst } q \rangle \langle d = \text{snd } q \rangle$ 
    moreover assume  $\langle q \in \text{set } ?qs \rangle$ 
    ultimately have  $\langle (c, d) \in \text{set } ?qs \rangle$ 
      by simp
    from sorted-stable-segment [of ?cmp  $\langle (a, d) \rangle$  ps]
    have  $\langle \text{sorted } ?\text{cmp} (\text{filter } (\lambda(c, b). \text{compare } (\text{prod-lex } \text{cmp1 } \text{cmp2}) (a, d) (c, b) = \text{Equiv}) \ ps) \rangle$ 
      by (simp only: case-prod-unfold prod.collapse)
    also have  $\langle (\lambda(c, b). \text{compare } (\text{prod-lex } \text{cmp1 } \text{cmp2}) (a, d) (c, b) = \text{Equiv}) =$ 
       $\langle \lambda(c, b). \text{compare } \text{cmp1 } a \ c = \text{Equiv} \wedge \text{compare } \text{cmp2 } d \ b = \text{Equiv} \rangle$ 
      by (simp add: fun-eq-iff compare-prod-lex-apply split: comp.split)
    finally have  $*$ :  $\langle \text{sorted } ?\text{cmp} (\text{filter } (\lambda(c, b). \text{compare } \text{cmp1 } a \ c = \text{Equiv} \wedge$ 
       $\text{compare } \text{cmp2 } d \ b = \text{Equiv}) \ ps) \rangle$  .
    let  $?rs = \langle \text{filter } (\lambda(-, d'). \text{compare } \text{cmp2 } d \ d' = \text{Equiv}) \ ?qs \rangle$ 
    have  $\langle \text{sort } ?\text{cmp } ?rs = ?rs \rangle$ 
      by (rule sort-eqI) (use * in  $\langle \text{simp-all add: case-prod-unfold} \rangle$ )
    then show  $\langle \text{filter } (\lambda r. \text{compare } ?\text{cmp2 } q \ r = \text{Equiv}) \ ?qs =$ 
       $\text{filter } (\lambda r. \text{compare } ?\text{cmp2 } q \ r = \text{Equiv}) (\text{sort } ?\text{cmp } ?qs) \rangle$ 
      by (simp add: filter-sort case-prod-unfold flip:  $\langle d = \text{snd } q \rangle$ )
  qed
  then show  $\langle \text{filter } (\lambda q. \text{compare } ?\text{cmp1 } p \ q = \text{Equiv}) (\text{sort } ?\text{cmp2 } ps) =$ 
     $\text{filter } (\lambda q. \text{compare } ?\text{cmp1 } p \ q = \text{Equiv}) (\text{sort } ?\text{cmp } ps) \rangle$ 
    by (simp add: filter-sort case-prod-unfold flip: ab)
  qed
  then show  $\langle (\text{sort } (\text{key } \text{fst } \text{cmp1}) \circ \text{sort } (\text{key } \text{snd } \text{cmp2})) \ ps = \text{sort } (\text{prod-lex}$ 
     $\text{cmp1 } \text{cmp2}) \ ps \rangle$ 
    by simp
qed

```

end

101 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

```
theory Sum-of-Squares
imports Complex-Main
begin
```

```
ML-file <Sum-of-Squares/positivstellensatz.ML>
ML-file <Sum-of-Squares/positivstellensatz-tools.ML>
ML-file <Sum-of-Squares/sum-of-squares.ML>
ML-file <Sum-of-Squares/sos-wrapper.ML>
```

end

```
theory Time-Commands
imports Main
keywords time-fun :: thy-decl
and time-function :: thy-decl
and time-definition :: thy-decl
and time-partial-function :: thy-decl
and equations
and time-fun-0 :: thy-decl
begin
```

```
ML-file Time-Commands-0.ML
ML-file Time-Commands.ML
```

```
declare [[time-prefix = T-]]
```

This theory provides commands for the automatic definition of step-counting running-time functions from HOL functions following the translation described in Section 1.5, Running Time, of the book "Functional Data Structures and Algorithms. A Proof Assistant Approach." See <https://functional-algorithms-verified.org>

Command *time-fun* *f* retrieves the definition of *f* and defines a corresponding step-counting running-time function *T-f*. For all auxiliary functions used by *f* (excluding constructors), running time functions must already have been defined. If the definition of the function requires a manual termination proof, use *time-function* accompanied by a *termination* command.

The pre-defined functions below are assumed to have constant running

time. In fact, we make that constant 0. This does not change the asymptotic running time of user-defined functions using the pre-defined functions because 1 is added for every user-defined function call.

Many of the functions below are polymorphic and reside in type classes. The constant-time assumption is justified only for those types where the hardware offers suitable support, e.g. numeric types. The argument size is implicitly bounded, too.

The constant-time assumption for $(=)$ is justified for recursive data types such as lists and trees as long as the comparison is of the form $t = c$ where c is a constant term, for example $xs = []$.

Users of this running time framework need to ensure that 0-time functions are used only within the above restrictions.

```

time-fun-0 min
time-fun-0 max
time-fun-0 (+)
time-fun-0 (-)
time-fun-0 (*)
time-fun-0 (/)
time-fun-0 (div)
time-fun-0 (<)
time-fun-0 (≤)
time-fun-0 Not
time-fun-0 (∧)
time-fun-0 (∨)
time-fun-0 Num.n numeral-class.n numeral
time-fun-0 (=)

end

```

102 Time functions for various standard library operations. Also defines *itrev*.

```

theory Time-Functions
  imports Time-Commands
begin

time-fun fst
time-fun snd

time-fun (@)

lemma T-append: T-append xs ys = length xs + 1
by(induction xs) auto

class T-size =
  fixes T-size :: 'a ⇒ nat

```

```

instantiation list :: (-) T-size
begin

time-fun length

instance ..

end

abbreviation T-length :: 'a list  $\Rightarrow$  nat where
  T-length  $\equiv$  T-size

lemma T-length: T-length xs = length xs + 1
  by (induction xs) auto

lemmas [simp del] = T-size-list.simps

time-fun map

lemma T-map-simps [simp,code]:
  T-map T-f [] = 1
  T-map T-f (x # xs) = T-f x + T-map T-f xs + 1
by (simp-all add: T-map-def)

lemma T-map: T-map T-f xs = ( $\sum x \leftarrow xs.$  T-f x) + length xs + 1
  by (induction xs) auto

lemmas [simp del] = T-map-simps

time-fun filter

lemma T-filter-simps [code]:
  T-filter T-P [] = 1
  T-filter T-P (x # xs) = T-P x + T-filter T-P xs + 1
by (simp-all add: T-filter-def)

lemma T-filter: T-filter T-P xs = ( $\sum x \leftarrow xs.$  T-P x) + length xs + 1
by (induction xs) (auto simp: T-filter-simps)

time-fun nth

lemma T-nth: n < length xs  $\implies$  T-nth xs n = n + 1
  by (induction xs n rule: T-nth.induct) (auto split: nat.splits)

lemmas [simp del] = T-nth.simps

time-fun take
time-fun drop

```

lemma *T-take*: $T\text{-take } n \text{ } xs = \min n (\text{length } xs) + 1$
by (*induction xs arbitrary: n*) (*auto split: nat.splits*)

lemma *T-drop*: $T\text{-drop } n \text{ } xs = \min n (\text{length } xs) + 1$
by (*induction xs arbitrary: n*) (*auto split: nat.splits*)

time-fun *rev*

lemma *T-rev*: $T\text{-rev } xs \leq (\text{length } xs + 1)^2$
by(*induction xs*) (*auto simp: T-append power2-eq-square*)

fun *itrev* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
itrev [] $ys = ys$ |
itrev ($x \# xs$) $ys = \text{itrev } xs \ (x \# ys)$

lemma *itrev*: $\text{itrev } xs \text{ } ys = \text{rev } xs \ @ \ ys$
by(*induction xs arbitrary: ys*) *auto*

lemma *itrev-Nil*: $\text{itrev } xs \ [] = \text{rev } xs$
by(*simp add: itrev*)

time-fun *itrev*

lemma *T-itrev*: $T\text{-itrev } xs \text{ } ys = \text{length } xs + 1$
by(*induction xs arbitrary: ys*) *auto*

time-fun *tl*

lemma *T-tl*: $T\text{-tl } xs = 0$
by (*cases xs*) *simp-all*

declare *T-tl.simps*[*simp del*]

end

103 A table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtranc-path* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \Rightarrow \text{bool}$
for $r :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
where
base: $\text{rtranc-path } r \text{ } x \ [] \ x$
| *step*: $r \text{ } x \ y \Longrightarrow \text{rtranc-path } r \text{ } y \text{ } ys \ z \Longrightarrow \text{rtranc-path } r \text{ } x \ (y \# ys) \ z$

lemma *rtranclp-eq-rtrancl-path*: $r^{**} x y \longleftrightarrow (\exists xs. \text{rtrancl-path } r x xs y)$

proof

show $\exists xs. \text{rtrancl-path } r x xs y$ **if** $r^{**} x y$

using *that*

proof (*induct rule: converse-rtranclp-induct*)

case *base*

have $\text{rtrancl-path } r y [] y$ **by** (*rule rtrancl-path.base*)

then show *?case* ..

next

case (*step x z*)

from $\langle \exists xs. \text{rtrancl-path } r z xs y \rangle$

obtain *xs* **where** $\text{rtrancl-path } r z xs y$..

with $\langle r x z \rangle$ **have** $\text{rtrancl-path } r x (z \# xs) y$

by (*rule rtrancl-path.step*)

then show *?case* ..

qed

show $r^{**} x y$ **if** $\exists xs. \text{rtrancl-path } r x xs y$

proof –

from *that* **obtain** *xs* **where** $\text{rtrancl-path } r x xs y$..

then show *?thesis*

proof *induct*

case (*base x*)

show *?case*

by (*rule rtranclp.rtrancl-refl*)

next

case (*step x y ys z*)

from $\langle r x y \rangle \langle r^{**} y z \rangle$ **show** *?case*

by (*rule converse-rtranclp-into-rtranclp*)

qed

qed

qed

lemma *rtrancl-path-trans*:

assumes *xy*: $\text{rtrancl-path } r x xs y$

and *yz*: $\text{rtrancl-path } r y ys z$

shows $\text{rtrancl-path } r x (xs @ ys) z$ **using** *xy yz*

proof (*induct arbitrary: z*)

case (*base x*)

then show *?case* **by** *simp*

next

case (*step x y xs*)

then have $\text{rtrancl-path } r y (xs @ ys) z$

by *simp*

with $\langle r x y \rangle$ **have** $\text{rtrancl-path } r x (y \# (xs @ ys)) z$

by (*rule rtrancl-path.step*)

then show *?case* **by** *simp*

qed

lemma *rtrancl-path-appendE*:

assumes *xz*: *rtrancl-path* *r* *x* (*xs* @ *y* # *ys*) *z*
obtains *rtrancl-path* *r* *x* (*xs* @ [*y*]) *y* **and** *rtrancl-path* *r* *y* *ys* *z*
using *xz*
proof (*induct xs arbitrary: x*)
 case *Nil*
 then have *rtrancl-path* *r* *x* (*y* # *ys*) *z* **by** *simp*
 then obtain *xy*: *r* *x* *y* **and** *yz*: *rtrancl-path* *r* *y* *ys* *z*
 by cases auto
 from *xy* **have** *rtrancl-path* *r* *x* [*y*] *y*
 by (*rule rtrancl-path.step* [*OF* - *rtrancl-path.base*])
 then have *rtrancl-path* *r* *x* ([] @ [*y*]) *y* **by** *simp*
 then show thesis using yz by (*rule Nil*)
next
 case (*Cons a as*)
 then have *rtrancl-path* *r* *x* (*a* # (*as* @ *y* # *ys*)) *z* **by** *simp*
 then obtain *xa*: *r* *x* *a* **and** *az*: *rtrancl-path* *r* *a* (*as* @ *y* # *ys*) *z*
 by cases auto
 show thesis
 proof (*rule Cons(1)* [*OF* - *az*])
 assume *rtrancl-path* *r* *y* *ys* *z*
 assume *rtrancl-path* *r* *a* (*as* @ [*y*]) *y*
 with *xa* **have** *rtrancl-path* *r* *x* (*a* # (*as* @ [*y*])) *y*
 by (*rule rtrancl-path.step*)
 then have *rtrancl-path* *r* *x* ((*a* # *as*) @ [*y*]) *y*
 by *simp*
 then show thesis using *rtrancl-path* *r* *y* *ys* *z*
 by (*rule Cons(2)*)
qed
qed

lemma *rtrancl-path-distinct*:

assumes *xy*: *rtrancl-path* *r* *x* *xs* *y*
obtains *xs'* **where** *rtrancl-path* *r* *x* *xs'* *y* **and** *distinct* (*x* # *xs'*) **and** *set xs' ⊆*
set xs
using *xy*
proof (*induct xs rule: measure-induct-rule [of length]*)
 case (*less xs*)
 show ?*case*
 proof (*cases distinct* (*x* # *xs*))
 case *True*
 with *rtrancl-path* *r* *x* *xs* *y* **show** ?*thesis* **by** (*rule less*) *simp*
next
 case *False*
 then have \exists *as bs cs a. x # xs = as @ [a] @ bs @ [a] @ cs
 by (*rule not-distinct-decomp*)
 then obtain *as bs cs a* **where** *xs*: *x* # *xs* = *as* @ [*a*] @ *bs* @ [*a*] @ *cs*
 by *iprover*
 show ?*thesis**


```

proof (cases as)
  case Nil
  with xs have  $x: x = a$  and  $xs: xs = bs @ a \# cs$ 
    by auto
  from  $x \ xs \ \langle rtrancl\text{-}path \ r \ x \ xs \ y \rangle$  have  $cs: rtrancl\text{-}path \ r \ x \ cs \ y \ set \ cs \subseteq set \ xs$ 
    by (auto elim: rtrancl-path-appendE)
  from xs have  $length \ cs < length \ xs$  by simp
  then show ?thesis
    by (rule less(1))(blast intro: cs less(2) order-trans del: subsetI)+
next
  case (Cons d ds)
  with xs have  $xs: xs = ds @ a \# (bs @ [a] @ cs)$ 
    by auto
  with  $\langle rtrancl\text{-}path \ r \ x \ xs \ y \rangle$  obtain  $xa: rtrancl\text{-}path \ r \ x \ (ds @ [a]) \ a$ 
    and  $ay: rtrancl\text{-}path \ r \ a \ (bs @ a \# cs) \ y$ 
    by (auto elim: rtrancl-path-appendE)
  from ay have  $rtrancl\text{-}path \ r \ a \ cs \ y$  by (auto elim: rtrancl-path-appendE)
  with xa have  $xy: rtrancl\text{-}path \ r \ x \ ((ds @ [a]) @ cs) \ y$ 
    by (rule rtrancl-path-trans)
  from xs have  $set: set \ ((ds @ [a]) @ cs) \subseteq set \ xs$  by auto
  from xs have  $length \ ((ds @ [a]) @ cs) < length \ xs$  by simp
  then show ?thesis
    by (rule less(1))(blast intro: xy less(2) set[THEN subsetD])+
qed
qed
qed

inductive rtrancl-tab :: ( $'a \Rightarrow 'a \Rightarrow bool$ )  $\Rightarrow 'a \ list \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ 
  for  $r :: 'a \Rightarrow 'a \Rightarrow bool$ 
where
  base:  $rtrancl\text{-}tab \ r \ xs \ x \ x$ 
  | step:  $x \notin set \ xs \Longrightarrow r \ x \ y \Longrightarrow rtrancl\text{-}tab \ r \ (x \# xs) \ y \ z \Longrightarrow rtrancl\text{-}tab \ r \ xs \ x \ z$ 

lemma rtrancl-path-imp-rtrancl-tab:
  assumes path:  $rtrancl\text{-}path \ r \ x \ xs \ y$ 
  and  $x: distinct \ (x \# xs)$ 
  and  $ys: (\{x\} \cup set \ xs) \cap set \ ys = \{\}$ 
  shows  $rtrancl\text{-}tab \ r \ ys \ x \ y$ 
  using path  $x \ ys$ 
proof (induct arbitrary: ys)
  case base
  show ?case
    by (rule rtrancl-tab.base)
next
  case (step  $x \ y \ zs \ z$ )
  then have  $x \notin set \ ys$ 
    by auto
  from step have  $distinct \ (y \# zs)$ 
    by simp

```

```

moreover from step have  $(\{y\} \cup \text{set } zs) \cap \text{set } (x \# ys) = \{\}$ 
  by auto
ultimately have rtrancl-tab r  $(x \# ys)$  y z
  by (rule step)
with  $\langle x \notin \text{set } ys \rangle \langle r \ x \ y \rangle$  show ?case
  by (rule rtrancl-tab.step)
qed

```

```

lemma rtrancl-tab-imp-rtrancl-path:
  assumes tab: rtrancl-tab r ys x y
  obtains xs where rtrancl-path r x xs y
  using tab
proof induct
  case base
  from rtrancl-path.base show ?case
  by (rule base)
next
  case step
  show ?case
  by (iprover intro: step rtrancl-path.step)
qed

```

```

lemma rtranclp-eq-rtrancl-tab-nil:  $r^{**} \ x \ y \longleftrightarrow \text{rtrancl-tab } r \ [] \ x \ y$ 
proof
  show rtrancl-tab r  $[] \ x \ y$  if  $r^{**} \ x \ y$ 
  proof –
    from that obtain xs where rtrancl-path r x xs y
    by (auto simp add: rtranclp-eq-rtrancl-path)
    then obtain xs' where xs': rtrancl-path r x xs' y and distinct: distinct  $(x \#$ 
xs')
    by (rule rtrancl-path-distinct)
    have  $(\{x\} \cup \text{set } xs') \cap \text{set } [] = \{\}$ 
    by simp
    with xs' distinct show ?thesis
    by (rule rtrancl-path-imp-rtrancl-tab)
  qed
  show  $r^{**} \ x \ y$  if rtrancl-tab r  $[] \ x \ y$ 
  proof –
    from that obtain xs where rtrancl-path r x xs y
    by (rule rtrancl-tab-imp-rtrancl-path)
    then show ?thesis
    by (auto simp add: rtranclp-eq-rtrancl-path)
  qed
qed

```

```

declare rtranclp-rtrancl-eq [code del]
declare rtranclp-eq-rtrancl-tab-nil [THEN iffD2, code-pred-intro]

```

```

code-pred rtranclp

```

```

using rtranclp-eq-rtrancl-tab-nil [THEN iffD1] by fastforce

lemma rtrancl-path-Range:  $\llbracket \text{rtrancl-path } R \ x \ xs \ y; z \in \text{set } xs \rrbracket \implies \text{Range } R \ z$ 
by(induction rule: rtrancl-path.induct) auto

lemma rtrancl-path-Range-end:  $\llbracket \text{rtrancl-path } R \ x \ xs \ y; xs \neq [] \rrbracket \implies \text{Range } R \ y$ 
by(induction rule: rtrancl-path.induct)(auto elim: rtrancl-path.cases)

lemma rtrancl-path-nth:
   $\llbracket \text{rtrancl-path } R \ x \ xs \ y; i < \text{length } xs \rrbracket \implies R \ ((x \# xs) ! i) \ (xs ! i)$ 
proof(induction arbitrary: i rule: rtrancl-path.induct)
  case step thus ?case by(cases i) simp-all
qed simp

lemma rtrancl-path-last:  $\llbracket \text{rtrancl-path } R \ x \ xs \ y; xs \neq [] \rrbracket \implies \text{last } xs = y$ 
by(induction rule: rtrancl-path.induct)(auto elim: rtrancl-path.cases)

lemma rtrancl-path-mono:
   $\llbracket \text{rtrancl-path } R \ x \ p \ y; \bigwedge x \ y. R \ x \ y \implies S \ x \ y \rrbracket \implies \text{rtrancl-path } S \ x \ p \ y$ 
by(induction rule: rtrancl-path.induct)(auto intro: rtrancl-path.intros)

end

```

104 Binary Tree

```

theory Tree
imports Main
begin

datatype 'a tree =
  Leaf ( $\langle \rangle$ ) |
  Node 'a tree (value: 'a) 'a tree ( $\langle \langle \text{indent}=1 \text{ notation}=\langle \text{mixfix Node} \rangle \langle -, / -, / - \rangle \rangle$ )
datatype-compact tree

primrec left :: 'a tree  $\Rightarrow$  'a tree where
  left (Node l v r) = l |
  left Leaf = Leaf

primrec right :: 'a tree  $\Rightarrow$  'a tree where
  right (Node l v r) = r |
  right Leaf = Leaf

  Counting the number of leaves rather than nodes:

fun size1 :: 'a tree  $\Rightarrow$  nat where
  size1  $\langle \rangle$  = 1 |
  size1  $\langle l, x, r \rangle$  = size1 l + size1 r

fun subtrees :: 'a tree  $\Rightarrow$  'a tree set where
  subtrees  $\langle \rangle$  =  $\{\langle \rangle\}$  |

```

$$\text{subtrees } (\langle l, a, r \rangle) = \{\langle l, a, r \rangle\} \cup \text{subtrees } l \cup \text{subtrees } r$$

```
fun mirror :: 'a tree  $\Rightarrow$  'a tree where
  mirror  $\langle \rangle$  = Leaf |
  mirror  $\langle l, x, r \rangle$  =  $\langle \text{mirror } r, x, \text{mirror } l \rangle$ 
```

```
class height = fixes height :: 'a  $\Rightarrow$  nat
```

```
instantiation tree :: (type)height
begin
```

```
fun height-tree :: 'a tree  $\Rightarrow$  nat where
  height Leaf = 0 |
  height (Node l a r) = max (height l) (height r) + 1
```

```
instance ..
```

```
end
```

```
fun min-height :: 'a tree  $\Rightarrow$  nat where
  min-height Leaf = 0 |
  min-height (Node l - r) = min (min-height l) (min-height r) + 1
```

```
fun complete :: 'a tree  $\Rightarrow$  bool where
  complete Leaf = True |
  complete (Node l x r) = (height l = height r  $\wedge$  complete l  $\wedge$  complete r)
```

Almost complete:

```
definition acomplete :: 'a tree  $\Rightarrow$  bool where
  acomplete t = (height t - min-height t  $\leq$  1)
```

Weight balanced:

```
fun wbalanced :: 'a tree  $\Rightarrow$  bool where
  wbalanced Leaf = True |
  wbalanced (Node l x r) = (abs(int(size l) - int(size r))  $\leq$  1  $\wedge$  wbalanced l  $\wedge$  wbalanced r)
```

Internal path length:

```
fun ipl :: 'a tree  $\Rightarrow$  nat where
  ipl Leaf = 0 |
  ipl (Node l - r) = ipl l + size l + ipl r + size r
```

```
fun preorder :: 'a tree  $\Rightarrow$  'a list where
  preorder  $\langle \rangle$  = [] |
  preorder  $\langle l, x, r \rangle$  = x # preorder l @ preorder r
```

```
fun inorder :: 'a tree  $\Rightarrow$  'a list where
  inorder  $\langle \rangle$  = [] |
  inorder  $\langle l, x, r \rangle$  = inorder l @ [x] @ inorder r
```

A linear version avoiding append:

```
fun inorder2 :: 'a tree  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
inorder2  $\langle \rangle$  xs = xs |
inorder2  $\langle l, x, r \rangle$  xs = inorder2 l (x # inorder2 r xs)
```

```
fun postorder :: 'a tree  $\Rightarrow$  'a list where
postorder  $\langle \rangle$  = [] |
postorder  $\langle l, x, r \rangle$  = postorder l @ postorder r @ [x]
```

Binary Search Tree:

```
fun bst-wrt :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a tree  $\Rightarrow$  bool where
bst-wrt P  $\langle \rangle$   $\longleftrightarrow$  True |
bst-wrt P  $\langle l, a, r \rangle$   $\longleftrightarrow$ 
  ( $\forall x \in \text{set-tree } l. P x a$ )  $\wedge$  ( $\forall x \in \text{set-tree } r. P a x$ )  $\wedge$  bst-wrt P l  $\wedge$  bst-wrt P r
```

```
abbreviation bst :: ('a::linorder) tree  $\Rightarrow$  bool where
bst  $\equiv$  bst-wrt (<)
```

```
fun (in linorder) heap :: 'a tree  $\Rightarrow$  bool where
heap Leaf = True |
heap (Node l m r) =
  ( $\forall x \in \text{set-tree } l \cup \text{set-tree } r. m \leq x$ )  $\wedge$  heap l  $\wedge$  heap r
```

104.1 map-tree

```
lemma eq-map-tree-Leaf[simp]: map-tree f t = Leaf  $\longleftrightarrow$  t = Leaf
by (rule tree.map-disc-iff)
```

```
lemma eq-Leaf-map-tree[simp]: Leaf = map-tree f t  $\longleftrightarrow$  t = Leaf
by (cases t) auto
```

104.2 size

```
lemma size1-size: size1 t = size t + 1
by (induction t) simp-all
```

```
lemma size1-ge0[simp]: 0 < size1 t
by (simp add: size1-size)
```

```
lemma eq-size-0[simp]: size t = 0  $\longleftrightarrow$  t = Leaf
by (cases t) auto
```

```
lemma eq-0-size[simp]: 0 = size t  $\longleftrightarrow$  t = Leaf
by (cases t) auto
```

```
lemma neg-Leaf-iff: (t  $\neq$   $\langle \rangle$ ) = ( $\exists l a r. t = \langle l, a, r \rangle$ )
by (cases t) auto
```

```
lemma size-map-tree[simp]: size (map-tree f t) = size t
```

by (*induction t*) *auto*

lemma *size1-map-tree[simp]*: *size1 (map-tree f t) = size1 t*
by (*simp add: size1-size*)

104.3 *set-tree*

lemma *eq-set-tree-empty[simp]*: *set-tree t = {} \longleftrightarrow t = Leaf*
by (*cases t*) *auto*

lemma *eq-empty-set-tree[simp]*: *{ } = set-tree t \longleftrightarrow t = Leaf*
by (*cases t*) *auto*

lemma *finite-set-tree[simp]*: *finite(set-tree t)*
by(*induction t*) *auto*

104.4 *subtrees*

lemma *neg-subtrees-empty[simp]*: *subtrees t \neq { }*
by (*cases t*)(*auto*)

lemma *neg-empty-subtrees[simp]*: *{ } \neq subtrees t*
by (*cases t*)(*auto*)

lemma *size-subtrees*: *s \in subtrees t \implies size s \leq size t*
by(*induction t*)(*auto*)

lemma *set-treeE*: *a \in set-tree t $\implies \exists l r. \langle l, a, r \rangle \in$ subtrees t*
by (*induction t*)(*auto*)

lemma *Node-notin-subtrees-if[simp]*: *a \notin set-tree t \implies Node l a r \notin subtrees t*
by (*induction t*) *auto*

lemma *in-set-tree-if*: *$\langle l, a, r \rangle \in$ subtrees t \implies a \in set-tree t*
by (*metis Node-notin-subtrees-if*)

104.5 *height and min-height*

lemma *eq-height-0[simp]*: *height t = 0 \longleftrightarrow t = Leaf*
by(*cases t*) *auto*

lemma *eq-0-height[simp]*: *0 = height t \longleftrightarrow t = Leaf*
by(*cases t*) *auto*

lemma *height-map-tree[simp]*: *height (map-tree f t) = height t*
by (*induction t*) *auto*

lemma *height-le-size-tree*: *height t \leq size (t::'a tree)*
by (*induction t*) *auto*

```

lemma size1-height: size1 t ≤ 2 ^ height (t::'a tree)
proof(induction t)
  case (Node l a r)
  show ?case
  proof (cases height l ≤ height r)
    case True
    have size1(Node l a r) = size1 l + size1 r by simp
    also have ... ≤ 2 ^ height l + 2 ^ height r using Node.IH by arith
    also have ... ≤ 2 ^ height r + 2 ^ height r using True by simp
    also have ... = 2 ^ height (Node l a r)
      using True by (auto simp: max-def mult-2)
    finally show ?thesis .
  next
  case False
  have size1(Node l a r) = size1 l + size1 r by simp
  also have ... ≤ 2 ^ height l + 2 ^ height r using Node.IH by arith
  also have ... ≤ 2 ^ height l + 2 ^ height l using False by simp
  finally show ?thesis using False by (auto simp: max-def mult-2)
qed
qed simp

```

```

corollary size-height: size t ≤ 2 ^ height (t::'a tree) - 1
using size1-height[of t, unfolded size1-size] by(arith)

```

```

lemma height-subtrees: s ∈ subtrees t ⇒ height s ≤ height t
by (induction t) auto

```

```

lemma min-height-le-height: min-height t ≤ height t
by(induction t) auto

```

```

lemma min-height-map-tree[simp]: min-height (map-tree f t) = min-height t
by (induction t) auto

```

```

lemma min-height-size1: 2 ^ min-height t ≤ size1 t
proof(induction t)
  case (Node l a r)
  have (2::nat) ^ min-height (Node l a r) ≤ 2 ^ min-height l + 2 ^ min-height r
    by (simp add: min-def)
  also have ... ≤ size1(Node l a r) using Node.IH by simp
  finally show ?case .
qed simp

```

104.6 complete

```

lemma complete-iff-height: complete t ⇔ (min-height t = height t)
apply(induction t)
  apply simp
  apply (simp add: min-def max-def)

```

by (*metis le-antisym le-trans min-height-le-height*)

lemma *size1-if-complete*: $\text{complete } t \implies \text{size1 } t = 2^{\text{height } t}$
by (*induction t*) *auto*

lemma *size-if-complete*: $\text{complete } t \implies \text{size } t = 2^{\text{height } t} - 1$
using *size1-if-complete[simplified size1-size]* **by** *fastforce*

lemma *size1-height-if-incomplete*:
 $\neg \text{complete } t \implies \text{size1 } t < 2^{\text{height } t}$
proof(*induction t*)
case *Leaf* **thus** *?case* **by** *simp*
next
case (*Node l x r*)
have 1: *?case* **if** *h*: $\text{height } l < \text{height } r$
using *h size1-height[of l] size1-height[of r] power-strict-increasing[OF h, of 2::nat]*
by(*auto simp: max-def simp del: power-strict-increasing-iff*)
have 2: *?case* **if** *h*: $\text{height } l > \text{height } r$
using *h size1-height[of l] size1-height[of r] power-strict-increasing[OF h, of 2::nat]*
by(*auto simp: max-def simp del: power-strict-increasing-iff*)
have 3: *?case* **if** *h*: $\text{height } l = \text{height } r$ **and** *c*: $\neg \text{complete } l$
using *h size1-height[of r] Node.IH(1)[OF c] by(simp)*
have 4: *?case* **if** *h*: $\text{height } l = \text{height } r$ **and** *c*: $\neg \text{complete } r$
using *h size1-height[of l] Node.IH(2)[OF c] by(simp)*
from 1 2 3 4 *Node.prem*s **show** *?case* **apply** (*simp add: max-def*) **by** *linarith*
qed

lemma *complete-iff-min-height*: $\text{complete } t \iff (\text{height } t = \text{min-height } t)$
by(*auto simp add: complete-iff-height*)

lemma *min-height-size1-if-incomplete*:
 $\neg \text{complete } t \implies 2^{\text{min-height } t} < \text{size1 } t$
proof(*induction t*)
case *Leaf* **thus** *?case* **by** *simp*
next
case (*Node l x r*)
have 1: *?case* **if** *h*: $\text{min-height } l < \text{min-height } r$
using *h min-height-size1[of l] min-height-size1[of r] power-strict-increasing[OF h, of 2::nat]*
by(*auto simp: max-def simp del: power-strict-increasing-iff*)
have 2: *?case* **if** *h*: $\text{min-height } l > \text{min-height } r$
using *h min-height-size1[of l] min-height-size1[of r] power-strict-increasing[OF h, of 2::nat]*
by(*auto simp: max-def simp del: power-strict-increasing-iff*)
have 3: *?case* **if** *h*: $\text{min-height } l = \text{min-height } r$ **and** *c*: $\neg \text{complete } l$
using *h min-height-size1[of r] Node.IH(1)[OF c] by(simp add: complete-iff-min-height)*
have 4: *?case* **if** *h*: $\text{min-height } l = \text{min-height } r$ **and** *c*: $\neg \text{complete } r$


```

    using h min-height-size1 [of l] Node.IH(2)[OF c] by (simp add: complete-iff-min-height)
  from 1 2 3 4 Node.prem1 show ?case
    by (fastforce simp: complete-iff-min-height[THEN iffD1])
qed

```

```

lemma complete-if-size1-height: size1 t = 2 ^ height t  $\implies$  complete t
using size1-height-if-incomplete by fastforce

```

```

lemma complete-if-size1-min-height: size1 t = 2 ^ min-height t  $\implies$  complete t
using min-height-size1-if-incomplete by fastforce

```

```

lemma complete-iff-size1: complete t  $\longleftrightarrow$  size1 t = 2 ^ height t
using complete-if-size1-height size1-if-complete by blast

```

104.7 acomplete

```

lemma acomplete-subtreeL: acomplete (Node l x r)  $\implies$  acomplete l
by (simp add: acomplete-def)

```

```

lemma acomplete-subtreeR: acomplete (Node l x r)  $\implies$  acomplete r
by (simp add: acomplete-def)

```

```

lemma acomplete-subtrees:  $\llbracket$  acomplete t; s  $\in$  subtrees t  $\rrbracket \implies$  acomplete s
using [[simp-depth-limit=1]]
by (induction t arbitrary: s)
  (auto simp add: acomplete-subtreeL acomplete-subtreeR)

```

Balanced trees have optimal height:

```

lemma acomplete-optimal:
fixes t :: 'a tree and t' :: 'b tree
assumes acomplete t size t  $\leq$  size t' shows height t  $\leq$  height t'
proof (cases complete t)
  case True
  have (2::nat) ^ height t  $\leq$  2 ^ height t'
  proof -
    have 2 ^ height t = size1 t
      using True by (simp add: size1-if-complete)
    also have ...  $\leq$  size1 t' using assms(2) by (simp add: size1-size)
    also have ...  $\leq$  2 ^ height t' by (rule size1-height)
    finally show ?thesis .
  qed
  thus ?thesis by (simp)
next
  case False
  have (2::nat) ^ min-height t < 2 ^ height t'
  proof -
    have (2::nat) ^ min-height t < size1 t
      by (rule min-height-size1-if-incomplete[OF False])
    also have ...  $\leq$  size1 t' using assms(2) by (simp add: size1-size)
    also have ...  $\leq$  2 ^ height t' by (rule size1-height)
  qed

```

```

    finally have  $(2::nat) \wedge \text{min-height } t < (2::nat) \wedge \text{height } t'$  .
    thus ?thesis .
qed
hence *:  $\text{min-height } t < \text{height } t'$  by simp
have  $\text{min-height } t + 1 = \text{height } t$ 
  using  $\text{min-height-le-height[of } t] \text{ assms}(1) \text{ False}$ 
  by (simp add: complete-iff-height acomplete-def)
with * show ?thesis by arith
qed

```

104.8 wbalanced

```

lemma wbalanced-subtrees:  $\llbracket \text{wbalanced } t; s \in \text{subtrees } t \rrbracket \implies \text{wbalanced } s$ 
using  $\llbracket \text{simp-depth-limit}=1 \rrbracket$  by (induction t arbitrary: s) auto

```

104.9 ipl

The internal path length of a tree:

```

lemma ipl-if-complete-int:
  complete t  $\implies \text{int}(\text{ipl } t) = (\text{int}(\text{height } t) - 2) * 2^{\text{height } t} + 2$ 
apply (induction t)
  apply simp
  apply simp
  apply (simp add: algebra-simps size-if-complete of-nat-diff)
done

```

104.10 List of entries

```

lemma eq-inorder-Nil[simp]:  $\text{inorder } t = [] \longleftrightarrow t = \text{Leaf}$ 
by (cases t) auto

```

```

lemmas eq-Nil-inorder[simp] = eq-inorder-Nil[THEN eq-iff-swap]

```

```

lemma set-inorder[simp]:  $\text{set } (\text{inorder } t) = \text{set-tree } t$ 
by (induction t) auto

```

```

lemma preorder-eq-Nil-iff[simp]:  $(\text{preorder } t = []) = (t = \langle \rangle)$ 
by (cases t) auto

```

```

lemmas Nil-eq-preorder-iff [simp] = preorder-eq-Nil-iff[THEN eq-iff-swap]

```

```

lemma preorder-eq-Cons-iff:
  preorder t = x # xs  $\longleftrightarrow (\exists l r. t = \langle l, x, r \rangle \wedge xs = \text{preorder } l @ \text{preorder } r)$ 
by (cases t) auto

```

```

lemmas Cons-eq-preorder-iff = preorder-eq-Cons-iff[THEN eq-iff-swap]

```

```

lemma set-preorder[simp]:  $\text{set } (\text{preorder } t) = \text{set-tree } t$ 
by (induction t) auto

```

lemma *set-postorder*[simp]: $\text{set } (\text{postorder } t) = \text{set-tree } t$
by (*induction t*) *auto*

lemma *length-preorder*[simp]: $\text{length } (\text{preorder } t) = \text{size } t$
by (*induction t*) *auto*

lemma *length-inorder*[simp]: $\text{length } (\text{inorder } t) = \text{size } t$
by (*induction t*) *auto*

lemma *length-postorder*[simp]: $\text{length } (\text{postorder } t) = \text{size } t$
by (*induction t*) *auto*

lemma *preorder-map*: $\text{preorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{preorder } t)$
by (*induction t*) *auto*

lemma *inorder-map*: $\text{inorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{inorder } t)$
by (*induction t*) *auto*

lemma *postorder-map*: $\text{postorder } (\text{map-tree } f \ t) = \text{map } f \ (\text{postorder } t)$
by (*induction t*) *auto*

lemma *inorder2-inorder*: $\text{inorder2 } t \ xs = \text{inorder } t \ @ \ xs$
by (*induction t arbitrary: xs*) *auto*

104.11 Binary Search Tree

lemma *bst-wrt-mono*: $(\bigwedge x \ y. P \ x \ y \implies Q \ x \ y) \implies \text{bst-wrt } P \ t \implies \text{bst-wrt } Q \ t$
by (*induction t*) (*auto*)

lemma *bst-wrt-le-if-bst*: $\text{bst } t \implies \text{bst-wrt } (\leq) \ t$
using *bst-wrt-mono less-imp-le* **by** *blast*

lemma *bst-wrt-le-iff-sorted*: $\text{bst-wrt } (\leq) \ t \longleftrightarrow \text{sorted } (\text{inorder } t)$
apply (*induction t*)
apply(*simp*)
by (*fastforce simp: sorted-append intro: less-imp-le less-trans*)

lemma *bst-iff-sorted-wrt-less*: $\text{bst } t \longleftrightarrow \text{sorted-wrt } (<) \ (\text{inorder } t)$
apply (*induction t*)
apply *simp*
apply (*fastforce simp: sorted-wrt-append*)
done

104.12 heap

104.13 mirror

lemma *mirror-Leaf*[simp]: $\text{mirror } t = \langle \rangle \longleftrightarrow t = \langle \rangle$
by (*induction t*) *simp-all*

lemma *Leaf-mirror*[simp]: $\langle \rangle = \text{mirror } t \longleftrightarrow t = \langle \rangle$
using *mirror-Leaf* **by** *fastforce*

lemma *size-mirror*[simp]: $\text{size}(\text{mirror } t) = \text{size } t$
by (*induction t*) *simp-all*

lemma *size1-mirror*[simp]: $\text{size1}(\text{mirror } t) = \text{size1 } t$
by (*simp add: size1-size*)

lemma *height-mirror*[simp]: $\text{height}(\text{mirror } t) = \text{height } t$
by (*induction t*) *simp-all*

lemma *min-height-mirror* [simp]: $\text{min-height } (\text{mirror } t) = \text{min-height } t$
by (*induction t*) *simp-all*

lemma *ipl-mirror* [simp]: $\text{ipl } (\text{mirror } t) = \text{ipl } t$
by (*induction t*) *simp-all*

lemma *inorder-mirror*: $\text{inorder}(\text{mirror } t) = \text{rev}(\text{inorder } t)$
by (*induction t*) *simp-all*

lemma *map-mirror*: $\text{map-tree } f (\text{mirror } t) = \text{mirror } (\text{map-tree } f t)$
by (*induction t*) *simp-all*

lemma *mirror-mirror*[simp]: $\text{mirror}(\text{mirror } t) = t$
by (*induction t*) *simp-all*

end

105 Multiset of Elements of Binary Tree

theory *Tree-Multiset*
imports *Multiset Tree*
begin

Kept separate from theory *HOL-Library.Tree* to avoid importing all of theory *HOL-Library.Multiset* into *HOL-Library.Tree*. Should be merged if *HOL-Library.Multiset* ever becomes part of *Main*.

fun *mset-tree* :: 'a tree \Rightarrow 'a multiset **where**
mset-tree Leaf = {#} |
mset-tree (Node l a r) = {#a#} + *mset-tree* l + *mset-tree* r

fun *subtrees-mset* :: 'a tree \Rightarrow 'a tree multiset **where**
subtrees-mset Leaf = {#Leaf#} |
subtrees-mset (Node l x r) = *add-mset* (Node l x r) (*subtrees-mset* l + *subtrees-mset* r)

lemma *mset-tree-empty-iff*[simp]: $mset-tree\ t = \{\#\} \longleftrightarrow t = Leaf$
by (*cases t*) *auto*

lemma *set-mset-tree*[simp]: $set-mset\ (mset-tree\ t) = set-tree\ t$
by(*induction t*) *auto*

lemma *size-mset-tree*[simp]: $size(mset-tree\ t) = size\ t$
by(*induction t*) *auto*

lemma *mset-map-tree*: $mset-tree\ (map-tree\ f\ t) = image-mset\ f\ (mset-tree\ t)$
by (*induction t*) *auto*

lemma *mset-iff-set-tree*: $x \in \# mset-tree\ t \longleftrightarrow x \in set-tree\ t$
by(*induction t arbitrary: x*) *auto*

lemma *mset-preorder*[simp]: $mset\ (preorder\ t) = mset-tree\ t$
by (*induction t*) (*auto simp: ac-simps*)

lemma *mset-inorder*[simp]: $mset\ (inorder\ t) = mset-tree\ t$
by (*induction t*) (*auto simp: ac-simps*)

lemma *map-mirror*: $mset-tree\ (mirror\ t) = mset-tree\ t$
by (*induction t*) (*simp-all add: ac-simps*)

lemma *in-subtrees-mset-iff*[simp]: $s \in \# subtrees-mset\ t \longleftrightarrow s \in subtrees\ t$
by(*induction t*) *auto*

end

theory *Tree-Real*

imports

Complex-Main

Tree

begin

This theory is separate from *HOL-Library.Tree* because the former is discrete and builds on *Main* whereas this theory builds on *Complex-Main*.

lemma *size1-height-log*: $\log 2\ (size1\ t) \leq height\ t$
by (*simp add: log2-of-power-le size1-height*)

lemma *min-height-size1-log*: $min-height\ t \leq \log 2\ (size1\ t)$
by (*simp add: le-log2-of-power min-height-size1*)

lemma *size1-log-if-complete*: $complete\ t \implies height\ t = \log 2\ (size1\ t)$
by (*simp add: size1-if-complete*)

lemma *min-height-size1-log-if-incomplete*:
 $\neg complete\ t \implies min-height\ t < \log 2\ (size1\ t)$

by (simp add: less-log2-of-power min-height-size1-if-incomplete)

lemma min-height-acomplete: **assumes** acomplete t

shows min-height $t = \text{nat}(\text{floor}(\log 2 (\text{size1 } t)))$

proof cases

assume *: complete t

hence size1 $t = 2^{\text{min-height } t}$

by (simp add: complete-iff-height size1-if-complete)

from log2-of-power-eq[OF this] **show** ?thesis **by** linarith

next

assume *: \neg complete t

hence height $t = \text{min-height } t + 1$

using assms min-height-le-height[of t]

by (auto simp: acomplete-def complete-iff-height)

hence size1 $t < 2^{\text{min-height } t + 1}$ **by** (metis * size1-height-if-incomplete)

from floor-log-nat-eq-iff[OF min-height-size1 this] **show** ?thesis **by** simp

qed

lemma height-acomplete: **assumes** acomplete t

shows height $t = \text{nat}(\text{ceiling}(\log 2 (\text{size1 } t)))$

proof cases

assume *: complete t

hence size1 $t = 2^{\text{height } t}$ **by** (simp add: size1-if-complete)

from log2-of-power-eq[OF this] **show** ?thesis **by** linarith

next

assume *: \neg complete t

hence **: height $t = \text{min-height } t + 1$

using assms min-height-le-height[of t]

by (auto simp add: acomplete-def complete-iff-height)

hence size1 $t \leq 2^{\text{min-height } t + 1}$ **by** (metis size1-height)

from log2-of-power-le[OF this size1-ge0] min-height-size1-log-if-incomplete[OF *]

 **

show ?thesis **by** linarith

qed

lemma acomplete-Node-if-wbal1:

assumes acomplete l acomplete r size $l = \text{size } r + 1$

shows acomplete $\langle l, x, r \rangle$

proof –

from assms(3) **have** [simp]: size1 $l = \text{size1 } r + 1$ **by** (simp add: size1-size)

have nat $\lceil \log 2 (1 + \text{size1 } r) \rceil \geq \text{nat } \lceil \log 2 (\text{size1 } r) \rceil$

by (rule nat-mono[OF ceiling-mono]) simp

hence 1: height(Node l x r) = nat $\lceil \log 2 (1 + \text{size1 } r) \rceil + 1$

using height-acomplete[OF assms(1)] height-acomplete[OF assms(2)]

by (simp del: nat-ceiling-le-eq add: max-def)

have nat $\lfloor \log 2 (1 + \text{size1 } r) \rfloor \geq \text{nat } \lfloor \log 2 (\text{size1 } r) \rfloor$

by (rule nat-mono[OF floor-mono]) simp

hence 2: min-height(Node l x r) = nat $\lfloor \log 2 (\text{size1 } r) \rfloor + 1$

```

    using min-height-acomplete[OF assms(1)] min-height-acomplete[OF assms(2)]
    by (simp)
  have size1 r ≥ 1 by (simp add: size1-size)
  then obtain i where i: 2 ^ i ≤ size1 r size1 r < 2 ^ (i + 1)
    using ex-power-ivl1[of 2 size1 r] by auto
  hence i1: 2 ^ i < size1 r + 1 size1 r + 1 ≤ 2 ^ (i + 1) by auto
  from 1 2 floor-log-nat-eq-if[OF i] ceiling-log-nat-eq-if[OF i1]
  show ?thesis by (simp add: acomplete-def)
qed

```

```

lemma acomplete-sym: acomplete ⟨l, x, r⟩ ⟹ acomplete ⟨r, y, l⟩
by (auto simp: acomplete-def)

```

```

lemma acomplete-Node-if-wbal2:

```

```

assumes acomplete l acomplete r abs(int(size l) - int(size r)) ≤ 1

```

```

shows acomplete ⟨l, x, r⟩

```

```

proof -

```

```

  have size l = size r ∨ (size l = size r + 1 ∨ size r = size l + 1) (is ?A ∨ ?B)

```

```

    using assms(3) by linarith

```

```

  thus ?thesis

```

```

  proof

```

```

    assume ?A

```

```

    thus ?thesis using assms(1,2)

```

```

      apply (simp add: acomplete-def min-def max-def)

```

```

      by (metis assms(1,2) acomplete-optimal le-antisym le-less)

```

```

  next

```

```

    assume ?B

```

```

    thus ?thesis

```

```

      by (meson assms(1,2) acomplete-sym acomplete-Node-if-wbal1)

```

```

  qed

```

```

qed

```

```

lemma acomplete-if-wbalanced: wbalanced t ⟹ acomplete t

```

```

proof (induction t)

```

```

  case Leaf show ?case by (simp add: acomplete-def)

```

```

next

```

```

  case (Node l x r)

```

```

    thus ?case by (simp add: acomplete-Node-if-wbal2)

```

```

qed

```

```

end

```

106 Unordered pairs

```

theory Uprod imports Main begin

```

```

typedef ('a, 'b) commute = {f :: 'a ⇒ 'a ⇒ 'b. ∀ x y. f x y = f y x}
morphisms apply-commute Abs-commute
by auto

```

setup-lifting *type-definition-commute*

lemma *apply-commute-commute*: *apply-commute* $f\ x\ y = \text{apply-commute}\ f\ y\ x$
by(*transfer*) *simp*

context includes *lifting-syntax* **begin**

lift-definition *rel-commute* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('c \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow ('a, 'c) \Rightarrow ('b, 'd) \Rightarrow \text{bool}$
commute $\Rightarrow ('b, 'd) \Rightarrow \text{bool}$
is $\lambda A\ B. A \implies B$.

end

definition *eq-upair* :: $('a \times 'a) \Rightarrow ('a \times 'a) \Rightarrow \text{bool}$
where *eq-upair* = $(\lambda(a, b)\ (c, d). a = c \wedge b = d \vee a = d \wedge b = c)$

lemma *eq-upair-simps* [*simp*]:
 $\text{eq-upair}\ (a, b)\ (c, d) \longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
by(*simp add: eq-upair-def*)

lemma *equivp-eq-upair*: *equivp* *eq-upair*
by(*auto simp add: equivp-def fun-eq-iff*)

quotient-type $'a\ \text{uprod} = 'a \times 'a / \text{eq-upair}$ **by**(*rule equivp-eq-upair*)

lift-definition *Upair* :: $'a \Rightarrow 'a \Rightarrow 'a\ \text{uprod}$ **is** *Pair* **parametric** *Pair-transfer*[*of* $A\ A$ **for** A] .

lemma *uprod-exhaust* [*case-names Upair, cases type: uprod*]:
obtains $a\ b$ **where** $x = \text{Upair}\ a\ b$
by *transfer fastforce*

lemma *Upair-inject* [*simp*]: $\text{Upair}\ a\ b = \text{Upair}\ c\ d \longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
by *transfer auto*

code-datatype *Upair*

lift-definition *case-uprod* :: $('a, 'b) \Rightarrow 'a\ \text{uprod} \Rightarrow 'b$ **is** *case-prod*
parametric *case-prod-transfer*[*of* $A\ A$ **for** A] **by** *auto*

lemma *case-uprod-simps* [*simp, code*]: $\text{case-uprod}\ f\ (\text{Upair}\ x\ y) = \text{apply-commute}\ f\ x\ y$
by *transfer auto*

lemma *uprod-split*: $P\ (\text{case-uprod}\ f\ x) \longleftrightarrow (\forall a\ b. x = \text{Upair}\ a\ b \longrightarrow P\ (\text{apply-commute}\ f\ a\ b))$
by *transfer auto*

lemma *uprod-split-asm*: $P \text{ (case-uprod } f \text{ } x) \longleftrightarrow \neg (\exists a \ b. x = \text{Upair } a \ b \wedge \neg P \text{ (apply-commute } f \ a \ b))$
by *transfer auto*

lift-definition *not-equal* :: $('a, \text{bool}) \text{ commute is } (\neq) \text{ by auto}$

lemma *apply-not-equal* [*simp*]: $\text{apply-commute not-equal } x \ y \longleftrightarrow x \neq y$
by *transfer simp*

definition *proper-uprod* :: $'a \text{ uprod} \Rightarrow \text{bool}$
where *proper-uprod* = *case-uprod not-equal*

lemma *proper-uprod-simps* [*simp*, *code*]: $\text{proper-uprod } (\text{Upair } x \ y) \longleftrightarrow x \neq y$
by(*simp add: proper-uprod-def*)

context includes *lifting-syntax begin*

private lemma *set-uprod-parametric'*:
 $(\text{rel-prod } A \ A \Longrightarrow \text{rel-set } A) \ (\lambda(a, b). \{a, b\}) \ (\lambda(a, b). \{a, b\})$
by *transfer-prover*

lift-definition *set-uprod* :: $'a \text{ uprod} \Rightarrow 'a \text{ set is } \lambda(a, b). \{a, b\}$
parametric *set-uprod-parametric'* **by** *auto*

lemma *set-uprod-simps* [*simp*, *code*]: $\text{set-uprod } (\text{Upair } x \ y) = \{x, y\}$
by *transfer simp*

lemma *finite-set-uprod* [*simp*]: $\text{finite } (\text{set-uprod } x)$
by(*cases x*) *simp*

private lemma *map-uprod-parametric'*:
 $((A \Longrightarrow B) \Longrightarrow \text{rel-prod } A \ A \Longrightarrow \text{rel-prod } B \ B) \ (\lambda f. \text{map-prod } f \ f) \ (\lambda f. \text{map-prod } f \ f)$
by *transfer-prover*

lift-definition *map-uprod* :: $('a \Rightarrow 'b) \Rightarrow 'a \text{ uprod} \Rightarrow 'b \text{ uprod is } \lambda f. \text{map-prod } f \ f$
parametric *map-uprod-parametric'* **by** *auto*

lemma *map-uprod-simps* [*simp*, *code*]: $\text{map-uprod } f \ (\text{Upair } x \ y) = \text{Upair } (f \ x) \ (f \ y)$
by *transfer simp*

private lemma *rel-uprod-transfer'*:
 $((A \Longrightarrow B \Longrightarrow (=)) \Longrightarrow \text{rel-prod } A \ A \Longrightarrow \text{rel-prod } B \ B \Longrightarrow (=))$
 $(\lambda R \ (a, b) \ (c, d). R \ a \ c \wedge R \ b \ d \vee R \ a \ d \wedge R \ b \ c) \ (\lambda R \ (a, b) \ (c, d). R \ a \ c \wedge R \ b \ d \vee R \ a \ d \wedge R \ b \ c)$
by *transfer-prover*

lift-definition $rel\text{-uprod} :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\text{ uprod} \Rightarrow 'b\text{ uprod} \Rightarrow bool$
is $\lambda R (a, b) (c, d). R\ a\ c \wedge R\ b\ d \vee R\ a\ d \wedge R\ b\ c$ **parametric** $rel\text{-uprod-transfer}'$
by *auto*

lemma $rel\text{-uprod-simps}$ [*simp*, *code*]:
 $rel\text{-uprod}\ R\ (Upair\ a\ b)\ (Upair\ c\ d) \longleftrightarrow R\ a\ c \wedge R\ b\ d \vee R\ a\ d \wedge R\ b\ c$
by *transfer auto*

lemma $Upair\text{-parametric}$ [*transfer-rule*]: $(A \implies A \implies rel\text{-uprod}\ A)\ Upair$
 $Upair$
unfolding $rel\text{-fun-def}$ **by** *transfer auto*

lemma $case\text{-uprod-parametric}$ [*transfer-rule*]:
 $(rel\text{-commute}\ A\ B \implies rel\text{-uprod}\ A \implies B)\ case\text{-uprod}\ case\text{-uprod}$
unfolding $rel\text{-fun-def}$ **by** *transfer(force dest: rel-funD)*

end

bnf $uprod: 'a\text{ uprod}$
 $map: map\text{-uprod}$
 $sets: set\text{-uprod}$
 $bd: natLeq$
 $rel: rel\text{-uprod}$

proof –

show $map\text{-uprod}\ id = id$ **unfolding** $fun\text{-eq-iff}$ **by** *transfer auto*
show $map\text{-uprod}\ (g \circ f) = map\text{-uprod}\ g \circ map\text{-uprod}\ f$ **for** $f :: 'a \Rightarrow 'b$ **and** $g :: 'b \Rightarrow 'c$
unfolding $fun\text{-eq-iff}$ **by** *transfer auto*
show $map\text{-uprod}\ f\ x = map\text{-uprod}\ g\ x$ **if** $\bigwedge z. z \in set\text{-uprod}\ x \implies f\ z = g\ z$
for $f :: 'a \Rightarrow 'b$ **and** $g\ x$ **using that** **by** *transfer auto*
show $set\text{-uprod} \circ map\text{-uprod}\ f = (\cdot) f \circ set\text{-uprod}$ **for** $f :: 'a \Rightarrow 'b$ **by** *transfer auto*
show $card\text{-order}\ natLeq$ **by**(*rule natLeq-card-order*)
show $BNF\text{-Cardinal-Arithmetic.cinfinite}\ natLeq$ **by**(*rule natLeq-cinfinite*)
show $regularCard\ natLeq$ **by**(*rule regularCard-natLeq*)
show $ordLess2\ (card\text{-of}\ (set\text{-uprod}\ x))\ natLeq$ **for** $x :: 'a\text{ uprod}$
by (*auto simp flip: finite-iff-ordLess-natLeq*)
show $rel\text{-uprod}\ R\ OO\ rel\text{-uprod}\ S \leq rel\text{-uprod}\ (R\ OO\ S)$
for $R :: 'a \Rightarrow 'b \Rightarrow bool$ **and** $S :: 'b \Rightarrow 'c \Rightarrow bool$ **by**(*rule predicate2I*)(*transfer; auto*)
show $rel\text{-uprod}\ R = (\lambda x\ y. \exists z. set\text{-uprod}\ z \subseteq \{(x, y). R\ x\ y\} \wedge map\text{-uprod}\ fst\ z = x \wedge map\text{-uprod}\ snd\ z = y)$
for $R :: 'a \Rightarrow 'b \Rightarrow bool$ **by** *transfer(auto simp add: fun-eq-iff)*
qed

lemma $pred\text{-uprod-code}$ [*simp*, *code*]: $pred\text{-uprod}\ P\ (Upair\ x\ y) \longleftrightarrow P\ x \wedge P\ y$
by(*simp add: pred-uprod-def*)

instantiation *uprod* :: (*equal*) *equal* **begin**

definition *equal-uprod* :: 'a *uprod* \Rightarrow 'a *uprod* \Rightarrow bool
where *equal-uprod* = (=)

lemma *equal-uprod-code* [*code*]:

HOL.equal (*Upair* *x y*) (*Upair* *z u*) \longleftrightarrow *x* = *z* \wedge *y* = *u* \vee *x* = *u* \wedge *y* = *z*

unfolding *equal-uprod-def* **by** *simp*

instance **by** *standard*(*simp add: equal-uprod-def*)
end

quickcheck-generator *uprod constructors: Upair*

lemma *UNIV-uprod*: *UNIV* = ($\lambda x.$ *Upair* *x x*) ‘ *UNIV* \cup ($\lambda(x, y).$ *Upair* *x y*) ‘
Sigma UNIV ($\lambda x.$ *UNIV* – {*x*})

apply(*rule set-eqI*)

subgoal **for** *x* **by**(*cases x*) *auto*

done

context **begin**

private lift-definition *upair-inv* :: 'a *uprod* \Rightarrow 'a

is $\lambda(x, y).$ *if* *x* = *y* *then* *x* *else* *undefined* **by** *auto*

lemma *finite-UNIV-prod* [*simp*]:

finite (*UNIV* :: 'a *uprod set*) \longleftrightarrow *finite* (*UNIV* :: 'a *set*) (**is** ?*lhs* = ?*rhs*)

proof

assume ?*lhs*

hence *finite* (*range* ($\lambda x :: 'a.$ *Upair* *x x*)) **by**(*rule finite-subset[rotated]*) *simp*

hence *finite* (*upair-inv* ‘ *range* ($\lambda x :: 'a.$ *Upair* *x x*)) **by**(*rule finite-imageI*)

also have *upair-inv* (*Upair* *x x*) = *x* **for** *x* :: 'a **by** *transfer simp*

then have *upair-inv* ‘ *range* ($\lambda x :: 'a.$ *Upair* *x x*) = *UNIV* **by**(*auto simp add: image-image*)

finally show ?*rhs* .

qed(*simp add: UNIV-uprod*)

end

lemma *card-UNIV-uprod*:

card (*UNIV* :: 'a *uprod set*) = *card* (*UNIV* :: 'a *set*) * (*card* (*UNIV* :: 'a *set*) + 1) *div* 2

(**is** ?*UPROD* = ?*A* * - *div* -)

proof(*cases finite* (*UNIV* :: 'a *set*))

case *True*

from *True* **obtain** *f* :: nat \Rightarrow 'a **where** *bij*: *bij-betw* *f* {0..*?A*} *UNIV*

by (*blast dest: ex-bij-betw-nat-finite*)

hence [*simp*]: *f* ‘ {0..*?A*} = *UNIV* **by**(*rule bij-betw-imp-surj-on*)

have *UNIV* = ($\lambda(x, y).$ *Upair* (*f x*) (*f y*)) ‘ (*SIGMA* *x*:{0..*?A*}. {..*x*})

apply(*rule set-eqI*)

```

    subgoal for x
      apply(cases x)
      apply(clarsimp)
      subgoal for a b
        apply(cases inv-into {0..?A} f a ≤ inv-into {0..?A} f b)
        subgoal by(rule rev-image-eqI[where x=(inv-into {0..?A} f -, inv-into
{0..?A} f -)])
          (auto simp add: inv-into-into[where A={0..?A} and f=f,
simplified] intro: f-inv-into-f[where f=f, symmetric])
        subgoal
          apply(simp only: not-le)
          apply(drule less-imp-le)
          apply(rule rev-image-eqI[where x=(inv-into {0..?A} f -, inv-into
{0..?A} f -)])
          apply(auto simp add: inv-into-into[where A={0..?A} and f=f, simpli-
fied] intro: f-inv-into-f[where f=f, symmetric])
        done
      done
    done
  done
  hence ?UPROD = card ... by simp
  also have ... = card (SIGMA x:{0..?A}. {..x})
    apply(rule card-image)
    using bij[THEN bij-betw-imp-inj-on]
    by(simp add: inj-on-def Ball-def)(metis leD le-eq-less-or-eq le-less-trans)
  also have ... = sum Suc {0..?A}
    by (subst card-SigmaI) simp-all
  also have ... = sum of-nat {Suc 0..?A}
    using sum.atLeastLessThan-reindex [symmetric, of Suc 0 ?A id]
    by (simp del: sum.op-ivl-Suc add: atLeastLessThanSuc-atLeastAtMost)
  also have ... = ?A * (?A + 1) div 2
    using gauss-sum-from-Suc-0 [of ?A, where ?a = nat] by simp
  finally show ?thesis .
qed simp

end

```

107 A type of finite bit strings

```

theory Word
imports
  HOL-Library.Type-Length
begin

```

107.1 Preliminaries

```

lemma signed-take-bit-decr-length-iff:
  ⟨signed-take-bit (LENGTH('a::len) - Suc 0) k = signed-take-bit (LENGTH('a)
- Suc 0) l

```

$\longleftrightarrow \text{take-bit LENGTH}('a) \ k = \text{take-bit LENGTH}('a) \ l$
by (*simp add: signed-take-bit-eq-iff-take-bit-eq*)

107.2 Fundamentals

107.2.1 Type definition

quotient-type (overloaded) $'a \text{ word} = \text{int} / \langle \lambda k \ l. \text{take-bit LENGTH}('a) \ k = \text{take-bit LENGTH}('a::\text{len}) \ l \rangle$

morphisms *rep Word* **by** (*auto intro!: equivpI reflpI sympI transpI*)

hide-const (**open**) *rep* — only for foundational purpose

hide-const (**open**) *Word* — only for code generation

107.2.2 Basic arithmetic

instantiation *word* :: (*len*) *comm-ring-1*
begin

lift-definition *zero-word* :: $\langle 'a \text{ word} \rangle$
is 0 .

lift-definition *one-word* :: $\langle 'a \text{ word} \rangle$
is 1 .

lift-definition *plus-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle (+) \rangle$
by (*auto simp: take-bit-eq-mod intro: mod-add-cong*)

lift-definition *minus-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle (-) \rangle$
by (*auto simp: take-bit-eq-mod intro: mod-diff-cong*)

lift-definition *uminus-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *uminus*
by (*auto simp: take-bit-eq-mod intro: mod-minus-cong*)

lift-definition *times-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle (*) \rangle$
by (*auto simp: take-bit-eq-mod intro: mod-mult-cong*)

instance
by (*standard; transfer*) (*simp-all add: algebra-simps*)

end

context
includes *lifting-syntax*
notes
power-transfer [*transfer-rule*]

```

    transfer-rule-of-bool [transfer-rule]
    transfer-rule-numeral [transfer-rule]
    transfer-rule-of-nat [transfer-rule]
    transfer-rule-of-int [transfer-rule]
begin

lemma power-transfer-word [transfer-rule]:
  ⟨(pcr-word == => (=) == => pcr-word) (⌈) (⌋)⟩
  by transfer-prover

lemma [transfer-rule]:
  ⟨((=) == => pcr-word) of-bool of-bool⟩
  by transfer-prover

lemma [transfer-rule]:
  ⟨((=) == => pcr-word) numeral numeral⟩
  by transfer-prover

lemma [transfer-rule]:
  ⟨((=) == => pcr-word) int of-nat⟩
  by transfer-prover

lemma [transfer-rule]:
  ⟨((=) == => pcr-word) (λk. k) of-int⟩
proof -
  have ⟨((=) == => pcr-word) of-int of-int⟩
    by transfer-prover
  then show ?thesis by (simp add: id-def)
qed

lemma [transfer-rule]:
  ⟨(pcr-word == => (↔)) even ((dvd) 2 :: 'a::len word ⇒ bool)⟩
proof -
  have even-word-unfold: even k ↔ (∃ l. take-bit LENGTH('a) k = take-bit
    LENGTH('a) (2 * l)) (is ?P ↔ ?Q)
    for k :: int
  by (metis dvd-triv-left evenE even-take-bit-eq len-not-eq-0)
  show ?thesis
    unfolding even-word-unfold [abs-def] dvd-def [where ?'a = 'a word, abs-def]
    by transfer-prover
qed

end

lemma exp-eq-zero-iff [simp]:
  ⟨2 ^ n = (0 :: 'a::len word) ↔ n ≥ LENGTH('a)⟩
  by transfer auto

lemma word-exp-length-eq-0 [simp]:

```

$\langle (2 :: 'a::len\ word) \wedge LENGTH('a) = 0 \rangle$
by *simp*

107.2.3 Basic tool setup

ML-file $\langle Tools/word-lib.ML \rangle$

107.2.4 Basic code generation setup

context

begin

qualified lift-definition *the-int* :: $\langle 'a::len\ word \Rightarrow int \rangle$
is $\langle take-bit\ LENGTH('a) \rangle$.

end

lemma [*code abstype*]:
 $\langle Word.Word\ (Word.the-int\ w) = w \rangle$
by *transfer simp*

lemma *Word-eq-word-of-int* [*code-post, simp*]:
 $\langle Word.Word = of-int \rangle$
by (*rule; transfer*) *simp*

quickcheck-generator *word*
constructors:
 $\langle 0 :: 'a::len\ word \rangle,$
 $\langle numeral :: num \Rightarrow 'a::len\ word \rangle$

instantiation *word* :: (*len*) *equal*
begin

lift-definition *equal-word* :: $\langle 'a\ word \Rightarrow 'a\ word \Rightarrow bool \rangle$
is $\langle \lambda k\ l.\ take-bit\ LENGTH('a)\ k = take-bit\ LENGTH('a)\ l \rangle$
by *simp*

instance

by (*standard; transfer*) *rule*

end

lemma [*code*]:
 $\langle HOL.equal\ v\ w \longleftrightarrow HOL.equal\ (Word.the-int\ v)\ (Word.the-int\ w) \rangle$
by *transfer (simp add: equal)*

lemma [*code*]:
 $\langle Word.the-int\ 0 = 0 \rangle$
by *transfer simp*

lemma [code]:
 $\langle \text{Word.the-int } 1 = 1 \rangle$
by *transfer simp*

lemma [code]:
 $\langle \text{Word.the-int } (v + w) = \text{take-bit } \text{LENGTH}('a) (\text{Word.the-int } v + \text{Word.the-int } w) \rangle$
for $v w :: \langle 'a::\text{len word} \rangle$
by *transfer (simp add: take-bit-add)*

lemma [code]:
 $\langle \text{Word.the-int } (- w) = (\text{let } k = \text{Word.the-int } w \text{ in if } w = 0 \text{ then } 0 \text{ else } 2^{\wedge} \text{LENGTH}('a) - k) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by *transfer (auto simp: take-bit-eq-mod zmod-zminus1-eq-if)*

lemma [code]:
 $\langle \text{Word.the-int } (v - w) = \text{take-bit } \text{LENGTH}('a) (\text{Word.the-int } v - \text{Word.the-int } w) \rangle$
for $v w :: \langle 'a::\text{len word} \rangle$
by *transfer (simp add: take-bit-diff)*

lemma [code]:
 $\langle \text{Word.the-int } (v * w) = \text{take-bit } \text{LENGTH}('a) (\text{Word.the-int } v * \text{Word.the-int } w) \rangle$
for $v w :: \langle 'a::\text{len word} \rangle$
by *transfer (simp add: take-bit-mult)*

107.2.5 Basic conversions

abbreviation *word-of-nat* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \rangle$
where $\langle \text{word-of-nat} \equiv \text{of-nat} \rangle$

abbreviation *word-of-int* :: $\langle \text{int} \Rightarrow 'a::\text{len word} \rangle$
where $\langle \text{word-of-int} \equiv \text{of-int} \rangle$

lemma *word-of-nat-eq-iff*:
 $\langle \text{word-of-nat } m = (\text{word-of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit } \text{LENGTH}('a) m = \text{take-bit } \text{LENGTH}('a) n \rangle$
by *transfer (simp add: take-bit-of-nat)*

lemma *word-of-int-eq-iff*:
 $\langle \text{word-of-int } k = (\text{word-of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit } \text{LENGTH}('a) k = \text{take-bit } \text{LENGTH}('a) l \rangle$
by *transfer rule*

lemma *word-of-nat-eq-0-iff*:
 $\langle \text{word-of-nat } n = (0 :: 'a::\text{len word}) \longleftrightarrow 2^{\wedge} \text{LENGTH}('a) \text{ dvd } n \rangle$
using *word-of-nat-eq-iff* [where ?'a = 'a, of n 0] **by** *(simp add: take-bit-eq-0-iff)*

lemma *word-of-int-eq-0-iff*:

$\langle \text{word-of-int } k = (0 :: 'a::\text{len word}) \longleftrightarrow 2 \wedge \text{LENGTH}('a) \text{ dvd } k \rangle$

using *word-of-int-eq-iff* [**where** $?'a = 'a$, of k 0] **by** (*simp add: take-bit-eq-0-iff*)

context *semiring-1*

begin

lift-definition *unsigned* :: $\langle 'b::\text{len word} \Rightarrow 'a \rangle$

is $\langle \text{of-nat} \circ \text{nat} \circ \text{take-bit LENGTH}('b) \rangle$

by *simp*

lemma *unsigned-0* [*simp*]:

$\langle \text{unsigned } 0 = 0 \rangle$

by *transfer simp*

lemma *unsigned-1* [*simp*]:

$\langle \text{unsigned } 1 = 1 \rangle$

by *transfer simp*

lemma *unsigned-numeral* [*simp*]:

$\langle \text{unsigned} (\text{numeral } n :: 'b::\text{len word}) = \text{of-nat} (\text{take-bit LENGTH}('b) (\text{numeral } n)) \rangle$

by *transfer (simp add: nat-take-bit-eq)*

lemma *unsigned-neg-numeral* [*simp*]:

$\langle \text{unsigned} (- \text{numeral } n :: 'b::\text{len word}) = \text{of-nat} (\text{nat} (\text{take-bit LENGTH}('b) (- \text{numeral } n))) \rangle$

by *transfer simp*

end

context *semiring-1*

begin

lemma *unsigned-of-nat*:

$\langle \text{unsigned} (\text{word-of-nat } n :: 'b::\text{len word}) = \text{of-nat} (\text{take-bit LENGTH}('b) n) \rangle$

by *transfer (simp add: nat-eq-iff take-bit-of-nat)*

lemma *unsigned-of-int*:

$\langle \text{unsigned} (\text{word-of-int } k :: 'b::\text{len word}) = \text{of-nat} (\text{nat} (\text{take-bit LENGTH}('b) k)) \rangle$

by *transfer simp*

end

context *semiring-char-0*

begin

lemma *unsigned-word-eqI*:

$\langle v = w \rangle$ **if** $\langle \text{unsigned } v = \text{unsigned } w \rangle$
using that by transfer (*simp add: eq-nat-nat-iff*)

lemma *word-eq-iff-unsigned*:
 $\langle v = w \longleftrightarrow \text{unsigned } v = \text{unsigned } w \rangle$
by (*auto intro: unsigned-word-eqI*)

lemma *inj-unsigned* [*simp*]:
 $\langle \text{inj unsigned} \rangle$
by (*rule injI*) (*simp add: unsigned-word-eqI*)

lemma *unsigned-eq-0-iff*:
 $\langle \text{unsigned } w = 0 \longleftrightarrow w = 0 \rangle$
using *word-eq-iff-unsigned* [*of w 0*] **by** *simp*

end

context *ring-1*
begin

lift-definition *signed* :: $\langle 'b::\text{len word} \Rightarrow 'a \rangle$
is $\langle \text{of-int} \circ \text{signed-take-bit } (\text{LENGTH}('b) - \text{Suc } 0) \rangle$
by (*simp flip: signed-take-bit-decr-length-iff*)

lemma *signed-0* [*simp*]:
 $\langle \text{signed } 0 = 0 \rangle$
by *transfer simp*

lemma *signed-1* [*simp*]:
 $\langle \text{signed } (1 :: 'b::\text{len word}) = (\text{if } \text{LENGTH}('b) = 1 \text{ then } - 1 \text{ else } 1) \rangle$
by (*transfer fixing: uminus; cases* $\langle \text{LENGTH}('b) \rangle$) (*auto dest: gr0-implies-Suc*)

lemma *signed-minus-1* [*simp*]:
 $\langle \text{signed } (- 1 :: 'b::\text{len word}) = - 1 \rangle$
by (*transfer fixing: uminus*) *simp*

lemma *signed-numeral* [*simp*]:
 $\langle \text{signed } (\text{numeral } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - 1) (\text{numeral } n)) \rangle$
by *transfer simp*

lemma *signed-neg-numeral* [*simp*]:
 $\langle \text{signed } (- \text{numeral } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - 1) (- \text{numeral } n)) \rangle$
by *transfer simp*

lemma *signed-of-nat*:
 $\langle \text{signed } (\text{word-of-nat } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH}('b) - \text{Suc } 0) (\text{int } n)) \rangle$

```

by transfer simp

lemma signed-of-int:
   $\langle \text{signed } (\text{word-of-int } n :: 'b::\text{len word}) = \text{of-int } (\text{signed-take-bit } (\text{LENGTH } 'b) - \text{Suc } 0) \ n) \rangle$ 
  by transfer simp

end

context ring-char-0
begin

lemma signed-word-eqI:
   $\langle v = w \rangle$  if  $\langle \text{signed } v = \text{signed } w \rangle$ 
  using that by transfer (simp flip: signed-take-bit-decr-length-iff)

lemma word-eq-iff-signed:
   $\langle v = w \iff \text{signed } v = \text{signed } w \rangle$ 
  by (auto intro: signed-word-eqI)

lemma inj-signed [simp]:
   $\langle \text{inj signed} \rangle$ 
  by (rule injI (simp add: signed-word-eqI))

lemma signed-eq-0-iff:
   $\langle \text{signed } w = 0 \iff w = 0 \rangle$ 
  using word-eq-iff-signed [of w 0] by simp

end

abbreviation unat ::  $\langle 'a::\text{len word} \Rightarrow \text{nat} \rangle$ 
  where  $\langle \text{unat} \equiv \text{unsigned} \rangle$ 

abbreviation uint ::  $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$ 
  where  $\langle \text{uint} \equiv \text{unsigned} \rangle$ 

abbreviation sint ::  $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$ 
  where  $\langle \text{sint} \equiv \text{signed} \rangle$ 

abbreviation ucast ::  $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$ 
  where  $\langle \text{ucast} \equiv \text{unsigned} \rangle$ 

abbreviation scast ::  $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$ 
  where  $\langle \text{scast} \equiv \text{signed} \rangle$ 

context
  includes lifting-syntax
begin

```

lemma [transfer-rule]:
 $\langle (pcr\text{-}word ==> (=)) (nat \circ take\text{-}bit\ LENGTH('a)) (unat :: 'a::len\ word \Rightarrow nat) \rangle$
using unsigned.transfer [where ?'a = nat] **by** simp

lemma [transfer-rule]:
 $\langle (pcr\text{-}word ==> (=)) (take\text{-}bit\ LENGTH('a)) (uint :: 'a::len\ word \Rightarrow int) \rangle$
using unsigned.transfer [where ?'a = int] **by** (simp add: comp-def)

lemma [transfer-rule]:
 $\langle (pcr\text{-}word ==> (=)) (signed\text{-}take\text{-}bit\ (LENGTH('a) - Suc\ 0)) (sint :: 'a::len\ word \Rightarrow int) \rangle$
using signed.transfer [where ?'a = int] **by** simp

lemma [transfer-rule]:
 $\langle (pcr\text{-}word ==> pcr\text{-}word) (take\text{-}bit\ LENGTH('a)) (ucast :: 'a::len\ word \Rightarrow 'b::len\ word) \rangle$
proof (rule rel-funI)
fix k :: int **and** w :: $\langle 'a\ word \rangle$
assume $\langle pcr\text{-}word\ k\ w \rangle$
then have $\langle w = word\text{-}of\text{-}int\ k \rangle$
by (simp add: pcr-word-def cr-word-def relcompp-apply)
moreover have $\langle pcr\text{-}word\ (take\text{-}bit\ LENGTH('a)\ k)\ (ucast\ (word\text{-}of\text{-}int\ k :: 'a\ word)) \rangle$
by transfer (simp add: pcr-word-def cr-word-def relcompp-apply)
ultimately show $\langle pcr\text{-}word\ (take\text{-}bit\ LENGTH('a)\ k)\ (ucast\ w) \rangle$
by simp
qed

lemma [transfer-rule]:
 $\langle (pcr\text{-}word ==> pcr\text{-}word) (signed\text{-}take\text{-}bit\ (LENGTH('a) - Suc\ 0)) (scast :: 'a::len\ word \Rightarrow 'b::len\ word) \rangle$
proof (rule rel-funI)
fix k :: int **and** w :: $\langle 'a\ word \rangle$
assume $\langle pcr\text{-}word\ k\ w \rangle$
then have $\langle w = word\text{-}of\text{-}int\ k \rangle$
by (simp add: pcr-word-def cr-word-def relcompp-apply)
moreover have $\langle pcr\text{-}word\ (signed\text{-}take\text{-}bit\ (LENGTH('a) - Suc\ 0)\ k)\ (scast\ (word\text{-}of\text{-}int\ k :: 'a\ word)) \rangle$
by transfer (simp add: pcr-word-def cr-word-def relcompp-apply)
ultimately show $\langle pcr\text{-}word\ (signed\text{-}take\text{-}bit\ (LENGTH('a) - Suc\ 0)\ k)\ (scast\ w) \rangle$
by simp
qed

end

lemma of-nat-unat [simp]:
 $\langle of\text{-}nat\ (unat\ w) = unsigned\ w \rangle$

by *transfer simp*

lemma *of-int-uint* [*simp*]:
 $\langle \text{of-int } (\text{uint } w) = \text{unsigned } w \rangle$
by *transfer simp*

lemma *of-int-sint* [*simp*]:
 $\langle \text{of-int } (\text{sint } a) = \text{signed } a \rangle$
by *transfer (simp-all add: take-bit-signed-take-bit)*

lemma *nat-uint-eq* [*simp*]:
 $\langle \text{nat } (\text{uint } w) = \text{unat } w \rangle$
by *transfer simp*

lemma *nat-of-natural-unsigned-eq* [*simp*]:
 $\langle \text{nat-of-natural } (\text{unsigned } w) = \text{unat } w \rangle$
by *transfer simp*

lemma *int-of-integer-unsigned-eq* [*simp*]:
 $\langle \text{int-of-integer } (\text{unsigned } w) = \text{uint } w \rangle$
by *transfer simp*

lemma *int-of-integer-signed-eq* [*simp*]:
 $\langle \text{int-of-integer } (\text{signed } w) = \text{sint } w \rangle$
by *transfer simp*

lemma *sgn-uint-eq* [*simp*]:
 $\langle \text{sgn } (\text{uint } w) = \text{of-bool } (w \neq 0) \rangle$
by *transfer (simp add: less-le)*

Aliases only for code generation

context
begin

qualified lift-definition *of-int* :: $\langle \text{int} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \text{take-bit } \text{LENGTH}('a) \rangle$.

qualified lift-definition *of-nat* :: $\langle \text{nat} \Rightarrow 'a::\text{len word} \rangle$
is $\langle \text{int} \circ \text{take-bit } \text{LENGTH}('a) \rangle$.

qualified lift-definition *the-nat* :: $\langle 'a::\text{len word} \Rightarrow \text{nat} \rangle$
is $\langle \text{nat} \circ \text{take-bit } \text{LENGTH}('a) \rangle$ **by** *simp*

qualified lift-definition *the-signed-int* :: $\langle 'a::\text{len word} \Rightarrow \text{int} \rangle$
is $\langle \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle$ **by** *(simp add: signed-take-bit-decr-length-iff)*

qualified lift-definition *cast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
is $\langle \text{take-bit } \text{LENGTH}('a) \rangle$ **by** *simp*

qualified lift-definition *signed-cast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
is $\langle \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle$ **by** $(\text{metis signed-take-bit-decr-length-iff})$

end

lemma [*code-abbrev, simp*]:
 $\langle \text{Word.the-int} = \text{uint} \rangle$
by *transfer rule*

lemma [*code*]:
 $\langle \text{Word.the-int } (\text{Word.of-int } k :: 'a::\text{len word}) = \text{take-bit LENGTH}('a) k \rangle$
by *transfer simp*

lemma [*code-abbrev, simp*]:
 $\langle \text{Word.of-int} = \text{word-of-int} \rangle$
by $(\text{rule}; \text{transfer}) \text{ simp}$

lemma [*code*]:
 $\langle \text{Word.the-int } (\text{Word.of-nat } n :: 'a::\text{len word}) = \text{take-bit LENGTH}('a) (\text{int } n) \rangle$
by *transfer (simp add: take-bit-of-nat)*

lemma [*code-abbrev, simp*]:
 $\langle \text{Word.of-nat} = \text{word-of-nat} \rangle$
by $(\text{rule}; \text{transfer}) (\text{simp add: take-bit-of-nat})$

lemma [*code*]:
 $\langle \text{Word.the-nat } w = \text{nat } (\text{Word.the-int } w) \rangle$
by *transfer simp*

lemma [*code-abbrev, simp*]:
 $\langle \text{Word.the-nat} = \text{unat} \rangle$
by $(\text{rule}; \text{transfer}) \text{ simp}$

lemma [*code*]:
 $\langle \text{Word.the-signed-int } w = (\text{let } k = \text{Word.the-int } w$
 $\text{in if bit } k (\text{LENGTH}('a) - \text{Suc } 0) \text{ then } k + \text{push-bit LENGTH}('a) (- 1) \text{ else } k) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by *transfer (simp add: bit-simps signed-take-bit-eq-take-bit-add)*

lemma [*code-abbrev, simp*]:
 $\langle \text{Word.the-signed-int} = \text{sint} \rangle$
by $(\text{rule}; \text{transfer}) \text{ simp}$

lemma [*code*]:
 $\langle \text{Word.the-int } (\text{Word.cast } w :: 'b::\text{len word}) = \text{take-bit LENGTH}('b) (\text{Word.the-int } w) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by *transfer simp*

```

lemma [code-abbrev, simp]:
  ⟨ Word.cast = ucast ⟩
  by (rule; transfer) simp

```

```

lemma [code]:
  ⟨ Word.the-int ( Word.signed-cast w :: 'b::len word ) = take-bit LENGTH('b) ( Word.the-signed-int w ) ⟩
  for w :: ⟨ 'a::len word ⟩
  by transfer simp

```

```

lemma [code-abbrev, simp]:
  ⟨ Word.signed-cast = scast ⟩
  by (rule; transfer) simp

```

```

lemma [code]:
  ⟨ unsigned w = of-nat ( nat ( Word.the-int w ) ) ⟩
  by transfer simp

```

```

lemma [code]:
  ⟨ signed w = of-int ( Word.the-signed-int w ) ⟩
  by transfer simp

```

107.3 Elementary case distinctions

```

lemma word-length-one [case-names zero minus-one length-beyond]:
  fixes w :: ⟨ 'a::len word ⟩
  obtains (zero) ⟨ LENGTH('a) = Suc 0 ⟩ ⟨ w = 0 ⟩
  | (minus-one) ⟨ LENGTH('a) = Suc 0 ⟩ ⟨ w = - 1 ⟩
  | (length-beyond) ⟨ 2 ≤ LENGTH('a) ⟩
proof (cases ⟨ 2 ≤ LENGTH('a) ⟩)
  case True
  with length-beyond show ?thesis .
next
  case False
  then have ⟨ LENGTH('a) = Suc 0 ⟩
  by simp
  then have ⟨ w = 0 ∨ w = - 1 ⟩
  by transfer auto
  with ⟨ LENGTH('a) = Suc 0 ⟩ zero minus-one show ?thesis
  by blast
qed

```

107.3.1 Basic ordering

```

instantiation word :: (len) linorder
begin

```

```

lift-definition less-eq-word :: 'a word ⇒ 'a word ⇒ bool
  is λ a b. take-bit LENGTH('a) a ≤ take-bit LENGTH('a) b

```

```

by simp

lift-definition less-word :: 'a word  $\Rightarrow$  'a word  $\Rightarrow$  bool
is  $\lambda a b.$  take-bit LENGTH('a) a < take-bit LENGTH('a) b
by simp

instance
by (standard; transfer) auto

end

interpretation word-order: ordering-top  $\langle (\leq) \rangle \langle (<) \rangle \langle - 1 :: 'a::len \text{ word} \rangle$ 
by (standard; transfer) (simp add: take-bit-eq-mod zmod-minus1)

interpretation word-coorder: ordering-top  $\langle (\geq) \rangle \langle (>) \rangle \langle 0 :: 'a::len \text{ word} \rangle$ 
by (standard; transfer) simp

lemma word-of-nat-less-eq-iff:
 $\langle \text{word-of-nat } m \leq (\text{word-of-nat } n :: 'a::len \text{ word}) \longleftrightarrow \text{take-bit LENGTH('a) } m$ 
 $\leq \text{take-bit LENGTH('a) } n \rangle$ 
by transfer (simp add: take-bit-of-nat)

lemma word-of-int-less-eq-iff:
 $\langle \text{word-of-int } k \leq (\text{word-of-int } l :: 'a::len \text{ word}) \longleftrightarrow \text{take-bit LENGTH('a) } k \leq$ 
 $\text{take-bit LENGTH('a) } l \rangle$ 
by transfer rule

lemma word-of-nat-less-iff:
 $\langle \text{word-of-nat } m < (\text{word-of-nat } n :: 'a::len \text{ word}) \longleftrightarrow \text{take-bit LENGTH('a) } m$ 
 $< \text{take-bit LENGTH('a) } n \rangle$ 
by transfer (simp add: take-bit-of-nat)

lemma word-of-int-less-iff:
 $\langle \text{word-of-int } k < (\text{word-of-int } l :: 'a::len \text{ word}) \longleftrightarrow \text{take-bit LENGTH('a) } k <$ 
 $\text{take-bit LENGTH('a) } l \rangle$ 
by transfer rule

lemma word-le-def [code]:
 $a \leq b \longleftrightarrow \text{uint } a \leq \text{uint } b$ 
by transfer rule

lemma word-less-def [code]:
 $a < b \longleftrightarrow \text{uint } a < \text{uint } b$ 
by transfer rule

lemma word-greater-zero-iff:
 $\langle a > 0 \longleftrightarrow a \neq 0 \rangle$  for  $a :: 'a::len \text{ word}$ 
by transfer (simp add: less-le)

```


lemma *of-nat-word-less-iff*:

$\langle \text{of-nat } m \leq (\text{of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a) } m \leq \text{take-bit LENGTH('a) } n \rangle$
by *transfer (simp add: take-bit-of-nat)*

lemma *of-nat-word-less-iff*:

$\langle \text{of-nat } m < (\text{of-nat } n :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a) } m < \text{take-bit LENGTH('a) } n \rangle$
by *transfer (simp add: take-bit-of-nat)*

lemma *of-int-word-less-iff*:

$\langle \text{of-int } k \leq (\text{of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a) } k \leq \text{take-bit LENGTH('a) } l \rangle$
by *transfer rule*

lemma *of-int-word-less-iff*:

$\langle \text{of-int } k < (\text{of-int } l :: 'a::\text{len word}) \longleftrightarrow \text{take-bit LENGTH('a) } k < \text{take-bit LENGTH('a) } l \rangle$
by *transfer rule*

instantiation *word* :: (*len*) *order-bot*
begin

lift-definition *bot-word* :: $\langle 'a \text{ word} \rangle$
is 0 .

instance

by (*standard; transfer*) *simp*

end

lemma *bot-word-eq*:

$\langle \text{bot} = (0 :: 'a::\text{len word}) \rangle$
by *transfer rule*

instantiation *word* :: (*len*) *order-top*
begin

lift-definition *top-word* :: $\langle 'a \text{ word} \rangle$
is $\langle - 1 \rangle$.

instance

by (*standard; transfer*) (*simp add: take-bit-int-less-eq-mask*)

end

lemma *top-word-eq*:

$\langle \text{top} = (- 1 :: 'a::\text{len word}) \rangle$
by *transfer rule*

107.4 Enumeration

lemma *inj-on-word-of-nat*:

$\langle \text{inj-on } (\text{word-of-nat} :: \text{nat} \Rightarrow 'a::\text{len word}) \{0..<2 \wedge \text{LENGTH}('a)\} \rangle$
by (rule *inj-onI*; transfer) (simp-all add: take-bit-int-eq-self)

lemma *UNIV-word-eq-word-of-nat*:

$\langle (\text{UNIV} :: 'a::\text{len word set}) = \text{word-of-nat } ' \{0..<2 \wedge \text{LENGTH}('a)\} \rangle$ (is $\langle - = ?A \rangle$)

proof

show $\langle \text{word-of-nat } ' \{0..<2 \wedge \text{LENGTH}('a)\} \subseteq \text{UNIV} \rangle$

by *simp*

show $\langle \text{UNIV} \subseteq ?A \rangle$

proof

fix $w :: 'a \text{ word}$

show $\langle w \in (\text{word-of-nat } ' \{0..<2 \wedge \text{LENGTH}('a)\} :: 'a \text{ word set}) \rangle$

by (rule *image-eqI* [of - - $\langle \text{unat } w \rangle$; transfer] simp-all

qed

qed

instantiation *word* :: (len) *enum*

begin

definition *enum-word* :: $\langle 'a \text{ word list} \rangle$

where $\langle \text{enum-word} = \text{map word-of-nat } [0..<2 \wedge \text{LENGTH}('a)] \rangle$

definition *enum-all-word* :: $\langle ('a \text{ word} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$

where $\langle \text{enum-all-word} = \text{All} \rangle$

definition *enum-ex-word* :: $\langle ('a \text{ word} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$

where $\langle \text{enum-ex-word} = \text{Ex} \rangle$

instance

by *standard*

(simp-all add: enum-all-word-def enum-ex-word-def enum-word-def distinct-map
inj-on-word-of-nat flip: UNIV-word-eq-word-of-nat)

end

lemma [code]:

$\langle \text{Enum.enum-all } P \longleftrightarrow \text{list-all } P \text{ Enum.enum} \rangle$

$\langle \text{Enum.enum-ex } P \longleftrightarrow \text{list-ex } P \text{ Enum.enum} \rangle$ **for** $P :: 'a::\text{len word} \Rightarrow \text{bool}$

by (simp-all add: enum-all-word-def enum-ex-word-def enum-UNIV list-all-iff
list-ex-iff)

107.5 Bit-wise operations

The following specification of word division just lifts the pre-existing division on integers named “F-Division” in [2].

instantiation *word* :: (*len*) *semiring-modulo*
begin

lift-definition *divide-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda a \ b. \text{take-bit } LENGTH('a) \ a \ \text{div} \ \text{take-bit } LENGTH('a) \ b \rangle$
by *simp*

lift-definition *modulo-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda a \ b. \text{take-bit } LENGTH('a) \ a \ \text{mod} \ \text{take-bit } LENGTH('a) \ b \rangle$
by *simp*

instance proof

show $a \ \text{div} \ b * b + a \ \text{mod} \ b = a$ **for** $a \ b :: 'a \text{ word}$
proof *transfer*
fix $k \ l :: \text{int}$
define $r :: \text{int}$ **where** $r = 2 \wedge LENGTH('a)$
then have $r: \text{take-bit } LENGTH('a) \ k = k \ \text{mod} \ r$ **for** k
by (*simp add: take-bit-eq-mod*)
have $k \ \text{mod} \ r = ((k \ \text{mod} \ r) \ \text{div} \ (l \ \text{mod} \ r) * (l \ \text{mod} \ r)$
 $+ (k \ \text{mod} \ r) \ \text{mod} \ (l \ \text{mod} \ r)) \ \text{mod} \ r$
by (*simp add: div-mult-mod-eq*)
also have $\dots = (((k \ \text{mod} \ r) \ \text{div} \ (l \ \text{mod} \ r) * (l \ \text{mod} \ r)) \ \text{mod} \ r$
 $+ (k \ \text{mod} \ r) \ \text{mod} \ (l \ \text{mod} \ r)) \ \text{mod} \ r$
by (*simp add: mod-add-left-eq*)
also have $\dots = (((k \ \text{mod} \ r) \ \text{div} \ (l \ \text{mod} \ r) * l) \ \text{mod} \ r$
 $+ (k \ \text{mod} \ r) \ \text{mod} \ (l \ \text{mod} \ r)) \ \text{mod} \ r$
by (*simp add: mod-mult-right-eq*)
finally have $k \ \text{mod} \ r = ((k \ \text{mod} \ r) \ \text{div} \ (l \ \text{mod} \ r) * l$
 $+ (k \ \text{mod} \ r) \ \text{mod} \ (l \ \text{mod} \ r)) \ \text{mod} \ r$
by (*simp add: mod-simps*)
with r **show** $\text{take-bit } LENGTH('a) \ (\text{take-bit } LENGTH('a) \ k \ \text{div} \ \text{take-bit}$
 $LENGTH('a) \ l * l$
 $+ \text{take-bit } LENGTH('a) \ k \ \text{mod} \ \text{take-bit } LENGTH('a) \ l) = \text{take-bit } LENGTH('a)$
 k
by *simp*
qed
qed
end

lemma *unat-div-distrib*:

$\langle \text{unat} \ (v \ \text{div} \ w) = \text{unat} \ v \ \text{div} \ \text{unat} \ w \rangle$
proof *transfer*
fix $k \ l$
have $\langle \text{nat} \ (\text{take-bit } LENGTH('a) \ k) \ \text{div} \ \text{nat} \ (\text{take-bit } LENGTH('a) \ l) \leq \text{nat}$
 $(\text{take-bit } LENGTH('a) \ k) \rangle$
by (*rule div-le-dividend*)
also have $\langle \text{nat} \ (\text{take-bit } LENGTH('a) \ k) < 2 \wedge LENGTH('a) \rangle$
by (*simp add: nat-less-iff*)

finally show $\langle (nat \circ take-bit\ LENGTH('a)) (take-bit\ LENGTH('a)\ k\ div\ take-bit\ LENGTH('a)\ l) =$
 $(nat \circ take-bit\ LENGTH('a))\ k\ div\ (nat \circ take-bit\ LENGTH('a))\ l \rangle$
by (*simp add: nat-take-bit-eq div-int-pos-iff nat-div-distrib take-bit-nat-eq-self-iff*)
qed

lemma *unat-mod-distrib*:

$\langle unat\ (v\ mod\ w) = unat\ v\ mod\ unat\ w \rangle$

proof *transfer*

fix $k\ l$

have $\langle nat\ (take-bit\ LENGTH('a)\ k)\ mod\ nat\ (take-bit\ LENGTH('a)\ l) \leq nat\ (take-bit\ LENGTH('a)\ k) \rangle$

by (*rule mod-less-eq-dividend*)

also have $\langle nat\ (take-bit\ LENGTH('a)\ k) < 2^{\wedge LENGTH('a)} \rangle$

by (*simp add: nat-less-iff*)

finally show $\langle (nat \circ take-bit\ LENGTH('a)) (take-bit\ LENGTH('a)\ k\ mod\ take-bit\ LENGTH('a)\ l) =$

$(nat \circ take-bit\ LENGTH('a))\ k\ mod\ (nat \circ take-bit\ LENGTH('a))\ l \rangle$

by (*simp add: nat-take-bit-eq mod-int-pos-iff less-le nat-mod-distrib take-bit-nat-eq-self-iff*)
qed

instance *word* :: (len) *semiring-parity*

by (*standard; transfer*)

(*auto simp: mod-2-eq-odd take-bit-Suc elim: evenE dest: le-Suc-ex*)

lemma *word-bit-induct* [*case-names zero even odd*]:

$\langle P\ a \rangle$ **if** *word-zero*: $\langle P\ 0 \rangle$

and *word-even*: $\langle \bigwedge a. P\ a \implies 0 < a \implies a < 2^{\wedge (LENGTH('a) - Suc\ 0)} \implies P\ (2 * a) \rangle$

and *word-odd*: $\langle \bigwedge a. P\ a \implies a < 2^{\wedge (LENGTH('a) - Suc\ 0)} \implies P\ (1 + 2 * a) \rangle$

for P **and** $a :: 'a :: len\ word$

proof –

define $m :: nat$ **where** $\langle m = LENGTH('a) - Suc\ 0 \rangle$

then have $l: \langle LENGTH('a) = Suc\ m \rangle$

by *simp*

define $n :: nat$ **where** $\langle n = unat\ a \rangle$

then have $\langle n < 2^{\wedge LENGTH('a)} \rangle$

by *transfer (simp add: take-bit-eq-mod)*

then have $\langle n < 2 * 2^{\wedge m} \rangle$

by (*simp add: l*)

then have $\langle P\ (of-nat\ n) \rangle$

proof (*induction n rule: nat-bit-induct*)

case *zero*

show *?case*

by *simp (rule word-zero)*

next

case (*even n*)

then have $\langle n < 2^{\wedge m} \rangle$

```

    by simp
  with even.IH have ⟨P (of-nat n)⟩
    by simp
  moreover from ⟨n < 2 ^ m⟩ even.hyps have ⟨0 < (of-nat n :: 'a word)⟩
    by (auto simp: word-greater-zero-iff l word-of-nat-eq-0-iff)
  moreover from ⟨n < 2 ^ m⟩ have ⟨(of-nat n :: 'a word) < 2 ^ (LENGTH('a)
- Suc 0)⟩
    using of-nat-word-less-iff [where ?'a = 'a, of n ⟨2 ^ m⟩]
    by (simp add: l take-bit-eq-mod)
  ultimately have ⟨P (2 * of-nat n)⟩
    by (rule word-even)
  then show ?case
    by simp
next
case (odd n)
then have ⟨Suc n ≤ 2 ^ m⟩
  by simp
with odd.IH have ⟨P (of-nat n)⟩
  by simp
moreover from ⟨Suc n ≤ 2 ^ m⟩ have ⟨(of-nat n :: 'a word) < 2 ^ (LENGTH('a)
- Suc 0)⟩
  using of-nat-word-less-iff [where ?'a = 'a, of n ⟨2 ^ m⟩]
  by (simp add: l take-bit-eq-mod)
ultimately have ⟨P (1 + 2 * of-nat n)⟩
  by (rule word-odd)
then show ?case
  by simp
qed
moreover have ⟨of-nat (nat (uint a)) = a⟩
  by transfer simp
ultimately show ?thesis
  by (simp add: n-def)
qed

lemma bit-word-half-eq:
  ⟨(of-bool b + a * 2) div 2 = a⟩
  if ⟨a < 2 ^ (LENGTH('a) - Suc 0)⟩
  for a :: ⟨'a::len word⟩
proof (cases ⟨2 ≤ LENGTH('a::len)⟩)
case False
with that show ?thesis
  by transfer simp
next
case True
obtain n where length: ⟨LENGTH('a) = Suc n⟩
  by (cases ⟨LENGTH('a)⟩) simp-all
show ?thesis proof (cases b)
case False
moreover have ⟨a * 2 div 2 = a⟩

```

```

using that proof transfer
  fix  $k :: \text{int}$ 
  from length have  $\langle k * 2 \bmod 2^{\text{LENGTH}('a)} = (k \bmod 2^n) * 2 \rangle$ 
    by simp
  moreover assume  $\langle \text{take-bit LENGTH}('a) k < \text{take-bit LENGTH}('a) (2^{\text{LENGTH}('a)} - \text{Suc } 0) \rangle$ 
  with  $\langle \text{LENGTH}('a) = \text{Suc } n \rangle$  have  $\langle \text{take-bit LENGTH}('a) k = \text{take-bit } n k \rangle$ 
    by (auto simp: take-bit-Suc-from-most)
  ultimately have  $\langle \text{take-bit LENGTH}('a) (k * 2) = \text{take-bit LENGTH}('a) k * 2 \rangle$ 
    by (simp add: take-bit-eq-mod)
  with True show  $\langle \text{take-bit LENGTH}('a) (\text{take-bit LENGTH}('a) (k * 2) \text{ div } \text{take-bit LENGTH}('a) 2) = \text{take-bit LENGTH}('a) k \rangle$ 
    by simp
qed
ultimately show ?thesis
  by simp
next
  case True
  moreover have  $\langle (1 + a * 2) \text{ div } 2 = a \rangle$ 
  using that proof transfer
    fix  $k :: \text{int}$ 
    from length have  $\langle (1 + k * 2) \bmod 2^{\text{LENGTH}('a)} = 1 + (k \bmod 2^n) * 2 \rangle$ 
      using pos-zmod-mult-2 [of  $\langle 2^n \rangle k$ ] by (simp add: ac-simps)
      moreover assume  $\langle \text{take-bit LENGTH}('a) k < \text{take-bit LENGTH}('a) (2^{\text{LENGTH}('a)} - \text{Suc } 0) \rangle$ 
      with  $\langle \text{LENGTH}('a) = \text{Suc } n \rangle$  have  $\langle \text{take-bit LENGTH}('a) k = \text{take-bit } n k \rangle$ 
        by (auto simp: take-bit-Suc-from-most)
      ultimately have  $\langle \text{take-bit LENGTH}('a) (1 + k * 2) = 1 + \text{take-bit LENGTH}('a) k * 2 \rangle$ 
        by (simp add: take-bit-eq-mod)
      with True show  $\langle \text{take-bit LENGTH}('a) (\text{take-bit LENGTH}('a) (1 + k * 2) \text{ div } \text{take-bit LENGTH}('a) 2) = \text{take-bit LENGTH}('a) k \rangle$ 
        by (auto simp: take-bit-Suc)
    qed
  ultimately show ?thesis
    by simp
qed
qed

lemma even-mult-exp-div-word-iff:
   $\langle \text{even } (a * 2^m \text{ div } 2^n) \longleftrightarrow \neg (m \leq n \wedge n < \text{LENGTH}('a) \wedge \text{odd } (a \text{ div } 2^{(n-m)})) \rangle$  for  $a :: \langle 'a :: \text{len word} \rangle$ 
by transfer
  (auto simp flip: drop-bit-eq-div simp add: even-drop-bit-iff-not-bit bit-take-bit-iff,

```

simp-all flip: push-bit-eq-mult add: bit-push-bit-iff-int)

instantiation *word* :: (*len*) *semiring-bits*
begin

lift-definition *bit-word* :: $\langle 'a \text{ word} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$

is $\langle \lambda k n. n < \text{LENGTH}('a) \wedge \text{bit } k n \rangle$

proof

fix *k l* :: *int* **and** *n* :: *nat*

assume *: $\langle \text{take-bit } \text{LENGTH}('a) k = \text{take-bit } \text{LENGTH}('a) l \rangle$

show $\langle n < \text{LENGTH}('a) \wedge \text{bit } k n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } l n \rangle$

proof (*cases* $\langle n < \text{LENGTH}('a) \rangle$)

case *True*

from * **have** $\langle \text{bit } (\text{take-bit } \text{LENGTH}('a) k) n \longleftrightarrow \text{bit } (\text{take-bit } \text{LENGTH}('a) l) n \rangle$

l) *n*

by *simp*

then show *?thesis*

by (*simp add: bit-take-bit-iff*)

next

case *False*

then show *?thesis*

by *simp*

qed

qed

instance proof

show $\langle P a \rangle$ **if** *stable*: $\langle \bigwedge a. a \text{ div } 2 = a \Longrightarrow P a \rangle$

and *rec*: $\langle \bigwedge a b. P a \Longrightarrow (\text{of-bool } b + 2 * a) \text{ div } 2 = a \Longrightarrow P (\text{of-bool } b + 2 * a) \rangle$

for *P* **and** *a* :: $\langle 'a \text{ word} \rangle$

proof (*induction a rule: word-bit-induct*)

case *zero*

have $\langle 0 \text{ div } 2 = (0 :: 'a \text{ word}) \rangle$

by *transfer simp*

with *stable* [*of 0*] **show** *?case*

by *simp*

next

case (*even a*)

with *rec* [*of a False*] **show** *?case*

using *bit-word-half-eq* [*of a False*] **by** (*simp add: ac-simps*)

next

case (*odd a*)

with *rec* [*of a True*] **show** *?case*

using *bit-word-half-eq* [*of a True*] **by** (*simp add: ac-simps*)

qed

show $\langle \text{bit } a n \longleftrightarrow \text{odd } (a \text{ div } 2 \wedge n) \rangle$ **for** *a* :: $\langle 'a \text{ word} \rangle$ **and** *n*

by *transfer (simp flip: drop-bit-eq-div add: drop-bit-take-bit bit-iff-odd-drop-bit)*

show $\langle a \text{ div } 0 = 0 \rangle$

for *a* :: $\langle 'a \text{ word} \rangle$

```

    by transfer simp
  show ⟨a div 1 = a⟩
    for a :: ⟨'a word⟩
    by transfer simp
  show ⟨0 div a = 0⟩
    for a :: ⟨'a word⟩
    by transfer simp
  show ⟨a mod b div b = 0⟩
    for a b :: ⟨'a word⟩
    by (simp add: word-eq-iff-unsigned [where ?'a = nat] unat-div-distrib unat-mod-distrib)
  show ⟨a div 2 div 2 ^ n = a div 2 ^ Suc n⟩
    for a :: ⟨'a word⟩ and m n :: nat
    apply transfer
    using drop-bit-eq-div [symmetric, where ?'a = int, of - 1]
    apply (auto simp: not-less take-bit-drop-bit ac-simps simp flip: drop-bit-eq-div
simp del: power.simps)
    apply (simp add: drop-bit-take-bit)
    done
  show ⟨even (2 * a div 2 ^ Suc n) ⟷ even (a div 2 ^ n)⟩ if ⟨2 ^ Suc n ≠ (0 :: 'a
word)⟩
    for a :: ⟨'a word⟩ and n :: nat
    using that by transfer
    (simp add: even-drop-bit-iff-not-bit bit-simps flip: drop-bit-eq-div del: power.simps)
qed

end

lemma bit-word-eqI:
  ⟨a = b⟩ if ⟨ $\bigwedge n. n < \text{LENGTH}('a) \implies \text{bit } a \ n \longleftrightarrow \text{bit } b \ n$ ⟩
  for a b :: ⟨'a::len word⟩
  using that by transfer (auto simp: nat-less-le bit-eq-iff bit-take-bit-iff)

lemma bit-imp-le-length: ⟨n < LENGTH('a)⟩ if ⟨bit w n⟩ for w :: ⟨'a::len word⟩
  by (meson bit-word.rep-eq that)

lemma not-bit-length [simp]:
  ⟨ $\neg \text{bit } w \ \text{LENGTH}('a)$ ⟩ for w :: ⟨'a::len word⟩
  using bit-imp-le-length by blast

lemma finite-bit-word [simp]:
  ⟨finite {n. bit w n}⟩
  for w :: ⟨'a::len word⟩
  by (metis bit-imp-le-length bounded-nat-set-is-finite mem-Collect-eq)

lemma bit-numeral-word-iff [simp]:
  ⟨bit (numeral w :: 'a::len word) n
  ⟷ n < LENGTH('a) ∧ bit (numeral w :: int) n⟩
  by transfer simp

```


lemma *bit-neg-numeral-word-iff* [*simp*]:
 $\langle \text{bit } (- \text{ numeral } w :: 'a::\text{len word}) \ n \rangle$
 $\longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } (- \text{ numeral } w :: \text{int}) \ n \rangle$
by *transfer simp*

instantiation *word* :: (*len*) *ring-bit-operations*
begin

lift-definition *not-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *not*
by (*simp add: take-bit-not-iff*)

lift-definition *and-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *and*
by *simp*

lift-definition *or-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *or*
by *simp*

lift-definition *xor-word* :: $\langle 'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *xor*
by *simp*

lift-definition *mask-word* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \rangle$
is *mask*
.

lift-definition *set-bit-word* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *set-bit*
by (*simp add: set-bit-def*)

lift-definition *unset-bit-word* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *unset-bit*
by (*simp add: unset-bit-def*)

lift-definition *flip-bit-word* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *flip-bit*
by (*simp add: flip-bit-def*)

lift-definition *push-bit-word* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is *push-bit*

proof –
show $\langle \text{take-bit } \text{LENGTH}('a) \ (\text{push-bit } n \ k) = \text{take-bit } \text{LENGTH}('a) \ (\text{push-bit } n \ l) \rangle$
if $\langle \text{take-bit } \text{LENGTH}('a) \ k = \text{take-bit } \text{LENGTH}('a) \ l \rangle$ **for** $k \ l :: \text{int}$ **and** $n :: \text{nat}$
by (*metis le-add2 push-bit-take-bit take-bit-tightened that*)
qed

lift-definition *drop-bit-word* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda n. \text{drop-bit } n \circ \text{take-bit } \text{LENGTH}('a) \rangle$
by (*simp add: take-bit-eq-mod*)

lift-definition *take-bit-word* :: $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a \text{ word} \rangle$
is $\langle \lambda n. \text{take-bit } (\min \text{LENGTH}('a) \ n) \rangle$
by (*simp add: ac-simps*) (*simp only: flip: take-bit-take-bit*)

context
includes *bit-operations-syntax*
begin

instance proof
fix *v w* :: $\langle 'a \text{ word} \rangle$ **and** *n m* :: *nat*
show $\langle \text{NOT } v = - \ v - 1 \rangle$
by *transfer (simp add: not-eq-complement)*
show $\langle v \ \text{AND} \ w = \text{of_bool } (\text{odd } v \wedge \text{odd } w) + 2 * (v \ \text{div} \ 2 \ \text{AND} \ w \ \text{div} \ 2) \rangle$
apply *transfer*
by (*rule bit-eqI*) (*auto simp: even-bit-succ-iff bit-simps bit-0 simp flip: bit-Suc*)
show $\langle v \ \text{OR} \ w = \text{of_bool } (\text{odd } v \vee \text{odd } w) + 2 * (v \ \text{div} \ 2 \ \text{OR} \ w \ \text{div} \ 2) \rangle$
apply *transfer*
by (*rule bit-eqI*) (*auto simp: even-bit-succ-iff bit-simps bit-0 simp flip: bit-Suc*)
show $\langle v \ \text{XOR} \ w = \text{of_bool } (\text{odd } v \neq \text{odd } w) + 2 * (v \ \text{div} \ 2 \ \text{XOR} \ w \ \text{div} \ 2) \rangle$
apply *transfer*
apply (*rule bit-eqI*)
subgoal for *k l n*
apply (*cases n*)
apply (*auto simp: even-bit-succ-iff bit-simps bit-0 even-xor-iff simp flip: bit-Suc*)
done
done
show $\langle \text{mask } n = 2^{\wedge} n - (1 :: 'a \text{ word}) \rangle$
by *transfer (simp flip: mask-eq-exp-minus-1)*
show $\langle \text{set-bit } n \ v = v \ \text{OR} \ \text{push-bit } n \ 1 \rangle$
by *transfer (simp add: set-bit-eq-or)*
show $\langle \text{unset-bit } n \ v = (v \ \text{OR} \ \text{push-bit } n \ 1) \ \text{XOR} \ \text{push-bit } n \ 1 \rangle$
by *transfer (simp add: unset-bit-eq-or-xor)*
show $\langle \text{flip-bit } n \ v = v \ \text{XOR} \ \text{push-bit } n \ 1 \rangle$
by *transfer (simp add: flip-bit-eq-xor)*
show $\langle \text{push-bit } n \ v = v * 2^{\wedge} n \rangle$
by *transfer (simp add: push-bit-eq-mult)*
show $\langle \text{drop-bit } n \ v = v \ \text{div} \ 2^{\wedge} n \rangle$
by *transfer (simp add: drop-bit-take-bit flip: drop-bit-eq-div)*
show $\langle \text{take-bit } n \ v = v \ \text{mod} \ 2^{\wedge} n \rangle$
by *transfer (simp flip: take-bit-eq-mod)*
qed
end

end

lemma [code]:
 $\langle \text{push-bit } n \ w = w * 2^n \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
by (fact push-bit-eq-mult)

lemma [code]:
 $\langle \text{Word.the-int } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{Word.the-int } w) \rangle$
by transfer (simp add: drop-bit-take-bit min-def le-less less-diff-conv)

lemma [code]:
 $\langle \text{Word.the-int } (\text{take-bit } n \ w) = (\text{if } n < \text{LENGTH}('a::\text{len}) \text{ then } \text{take-bit } n \ (\text{Word.the-int } w) \text{ else } \text{Word.the-int } w) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by transfer (simp add: not-le not-less ac-simps min-absorb2)

lemma [code-abbrev]:
 $\langle \text{push-bit } n \ 1 = (2 :: 'a::\text{len word})^n \rangle$
by (fact push-bit-of-1)

context
includes bit-operations-syntax
begin

lemma [code]:
 $\langle \text{NOT } w = \text{Word.of-int } (\text{NOT } (\text{Word.the-int } w)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by transfer (simp add: take-bit-not-take-bit)

lemma [code]:
 $\langle \text{Word.the-int } (v \ \text{AND } w) = \text{Word.the-int } v \ \text{AND } \text{Word.the-int } w \rangle$
by transfer simp

lemma [code]:
 $\langle \text{Word.the-int } (v \ \text{OR } w) = \text{Word.the-int } v \ \text{OR } \text{Word.the-int } w \rangle$
by transfer simp

lemma [code]:
 $\langle \text{Word.the-int } (v \ \text{XOR } w) = \text{Word.the-int } v \ \text{XOR } \text{Word.the-int } w \rangle$
by transfer simp

lemma [code]:
 $\langle \text{Word.the-int } (\text{mask } n :: 'a::\text{len word}) = \text{mask } (\text{min } \text{LENGTH}('a) \ n) \rangle$
by transfer simp

lemma [code]:
 $\langle \text{set-bit } n \ w = w \ \text{OR } \text{push-bit } n \ 1 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
by (fact set-bit-eq-or)

```

lemma [code]:
  ⟨unset-bit n w = w AND NOT (push-bit n 1)⟩ for w :: ⟨'a::len word⟩
  by (fact unset-bit-eq-and-not)

lemma [code]:
  ⟨flip-bit n w = w XOR push-bit n 1⟩ for w :: ⟨'a::len word⟩
  by (fact flip-bit-eq-xor)

context
  includes lifting-syntax
begin

lemma set-bit-word-transfer [transfer-rule]:
  ⟨((=) ==> pcr-word ==> pcr-word) set-bit set-bit⟩
  by (unfold set-bit-def) transfer-prover

lemma unset-bit-word-transfer [transfer-rule]:
  ⟨((=) ==> pcr-word ==> pcr-word) unset-bit unset-bit⟩
  by (unfold unset-bit-def) transfer-prover

lemma flip-bit-word-transfer [transfer-rule]:
  ⟨((=) ==> pcr-word ==> pcr-word) flip-bit flip-bit⟩
  by (unfold flip-bit-def) transfer-prover

lemma signed-take-bit-word-transfer [transfer-rule]:
  ⟨((=) ==> pcr-word ==> pcr-word)
    (λn k. signed-take-bit n (take-bit LENGTH('a::len) k))
    (signed-take-bit :: nat ⇒ 'a word ⇒ 'a word)⟩
  proof –
    let ?K = ⟨λn (k :: int). take-bit (min LENGTH('a) n) k OR of-bool (n <
      LENGTH('a) ∧ bit k n) * NOT (mask n)⟩
    let ?W = ⟨λn (w :: 'a word). take-bit n w OR of-bool (bit w n) * NOT (mask
      n)⟩
    have ⟨((=) ==> pcr-word ==> pcr-word) ?K ?W⟩
      by transfer-prover
    also have ⟨?K = (λn k. signed-take-bit n (take-bit LENGTH('a::len) k))⟩
      by (simp add: fun-eq-iff signed-take-bit-def bit-take-bit-iff ac-simps)
    also have ⟨?W = signed-take-bit⟩
      by (simp add: fun-eq-iff signed-take-bit-def)
    finally show ?thesis .
  qed

end

end

```

107.6 Conversions including casts

107.6.1 Generic unsigned conversion

context *semiring-bits*

begin

lemma *bit-unsigned-iff* [*bit-simps*]:

$\langle \text{bit } (\text{unsigned } w) \ n \longleftrightarrow \text{possible-bit } \text{TYPE}('a) \ n \wedge \text{bit } w \ n \rangle$

for $w :: \langle 'b :: \text{len word} \rangle$

by (*transfer fixing: bit*) (*simp add: bit-of-nat-iff bit-nat-iff bit-take-bit-iff*)

end

lemma *possible-bit-word*[*simp*]:

$\langle \text{possible-bit } \text{TYPE}(('a :: \text{len}) \text{ word}) \ m \longleftrightarrow m < \text{LENGTH}('a) \rangle$

by (*simp add: possible-bit-def linorder-not-le*)

context *semiring-bit-operations*

begin

lemma *unsigned-minus-1-eq-mask*:

$\langle \text{unsigned } (- \ 1 :: 'b :: \text{len word}) = \text{mask } \text{LENGTH}('b) \rangle$

by (*transfer fixing: mask*) (*simp add: nat-mask-eq of-nat-mask-eq*)

lemma *unsigned-push-bit-eq*:

$\langle \text{unsigned } (\text{push-bit } n \ w) = \text{take-bit } \text{LENGTH}('b) \ (\text{push-bit } n \ (\text{unsigned } w)) \rangle$

for $w :: \langle 'b :: \text{len word} \rangle$

proof (*rule bit-eqI*)

fix m

assume $\langle \text{possible-bit } \text{TYPE}('a) \ m \rangle$

show $\langle \text{bit } (\text{unsigned } (\text{push-bit } n \ w)) \ m = \text{bit } (\text{take-bit } \text{LENGTH}('b) \ (\text{push-bit } n \ (\text{unsigned } w))) \ m \rangle$

proof (*cases* $\langle n \leq m \rangle$)

case *True*

with $\langle \text{possible-bit } \text{TYPE}('a) \ m \rangle$ **have** $\langle \text{possible-bit } \text{TYPE}('a) \ (m - n) \rangle$

by (*simp add: possible-bit-less-imp*)

with *True* **show** *?thesis*

by (*simp add: bit-unsigned-iff bit-push-bit-iff Bit-Operations.bit-push-bit-iff bit-take-bit-iff not-le ac-simps*)

next

case *False*

then show *?thesis*

by (*simp add: not-le bit-unsigned-iff bit-push-bit-iff Bit-Operations.bit-push-bit-iff bit-take-bit-iff*)

qed

qed

lemma *unsigned-take-bit-eq*:

$\langle \text{unsigned } (\text{take-bit } n \ w) = \text{take-bit } n \ (\text{unsigned } w) \rangle$

```

for  $w :: \langle 'b :: \text{len word} \rangle$ 
by (rule bit-eqI) (simp add: bit-unsigned-iff bit-take-bit-iff Bit-Operations.bit-take-bit-iff)

```

```

end

```

```

context linordered-euclidean-semiring-bit-operations
begin

```

```

lemma unsigned-drop-bit-eq:

```

```

   $\langle \text{unsigned} (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{take-bit } \text{LENGTH}('b) \ (\text{unsigned } w)) \rangle$ 
  for  $w :: \langle 'b :: \text{len word} \rangle$ 
  by (rule bit-eqI) (auto simp: bit-unsigned-iff bit-take-bit-iff bit-drop-bit-eq Bit-Operations.bit-drop-bit-eq
    possible-bit-def dest: bit-imp-le-length)

```

```

end

```

```

lemma ucast-drop-bit-eq:

```

```

   $\langle \text{ucast} (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{ucast } w :: 'b :: \text{len word}) \rangle$ 
  if  $\langle \text{LENGTH}('a) \leq \text{LENGTH}('b) \rangle$  for  $w :: \langle 'a :: \text{len word} \rangle$ 
  by (rule bit-word-eqI) (use that in  $\langle \text{auto simp: bit-unsigned-iff bit-drop-bit-eq dest: bit-imp-le-length} \rangle$ )

```

```

context semiring-bit-operations
begin

```

```

context
  includes bit-operations-syntax
begin

```

```

lemma unsigned-and-eq:

```

```

   $\langle \text{unsigned} (v \ \text{AND} \ w) = \text{unsigned } v \ \text{AND} \ \text{unsigned } w \rangle$ 
  for  $v \ w :: \langle 'b :: \text{len word} \rangle$ 
  by (simp add: bit-eq-iff bit-simps)

```

```

lemma unsigned-or-eq:

```

```

   $\langle \text{unsigned} (v \ \text{OR} \ w) = \text{unsigned } v \ \text{OR} \ \text{unsigned } w \rangle$ 
  for  $v \ w :: \langle 'b :: \text{len word} \rangle$ 
  by (simp add: bit-eq-iff bit-simps)

```

```

lemma unsigned-xor-eq:

```

```

   $\langle \text{unsigned} (v \ \text{XOR} \ w) = \text{unsigned } v \ \text{XOR} \ \text{unsigned } w \rangle$ 
  for  $v \ w :: \langle 'b :: \text{len word} \rangle$ 
  by (simp add: bit-eq-iff bit-simps)

```

```

end

```

```

end

```

```

context ring-bit-operations

```

```

begin

context
  includes bit-operations-syntax
begin

lemma unsigned-not-eq:
  ⟨unsigned (NOT w) = take-bit LENGTH('b) (NOT (unsigned w))⟩
  for w :: ⟨'b::len word⟩
  by (simp add: bit-eq-iff bit-simps)

end

end

context linordered-euclidean-semiring
begin

lemma unsigned-greater-eq [simp]:
  ⟨0 ≤ unsigned w⟩ for w :: ⟨'b::len word⟩
  by (transfer fixing: less-eq) simp

lemma unsigned-less [simp]:
  ⟨unsigned w < 2 ^ LENGTH('b)⟩ for w :: ⟨'b::len word⟩
  by (transfer fixing: less) simp

end

context linordered-semidom
begin

lemma word-less-eq-iff-unsigned:
  a ≤ b ⟷ unsigned a ≤ unsigned b
  by (transfer fixing: less-eq) (simp add: nat-le-eq-zle)

lemma word-less-iff-unsigned:
  a < b ⟷ unsigned a < unsigned b
  by (transfer fixing: less) (auto dest: preorder-class.le-less-trans [OF take-bit-nonnegative])

end

107.6.2 Generic signed conversion

context ring-bit-operations
begin

lemma bit-signed-iff [bit-simps]:
  ⟨bit (signed w) n ⟷ possible-bit TYPE('a) n ∧ bit w (min (LENGTH('b) -
    Suc 0) n)⟩

```

```

for  $w :: \langle 'b :: \text{len word} \rangle$ 
by (transfer fixing: bit)
    (auto simp: bit-of-int-iff Bit-Operations.bit-signed-take-bit-iff min-def)

lemma signed-push-bit-eq:
   $\langle \text{signed } (\text{push-bit } n \ w) = \text{signed-take-bit } (\text{LENGTH('b)} - \text{Suc } 0) (\text{push-bit } n$ 
    (signed  $w :: 'a$ )  $\rangle$ 
  for  $w :: \langle 'b :: \text{len word} \rangle$ 
  by (simp add: bit-eq-iff bit-simps possible-bit-less-imp min-less-iff-disj auto)

lemma signed-take-bit-eq:
   $\langle \text{signed } (\text{take-bit } n \ w) = (\text{if } n < \text{LENGTH('b)} \text{ then } \text{take-bit } n \ (\text{signed } w) \text{ else}$ 
    (signed  $w$ )  $\rangle$ 
  for  $w :: \langle 'b :: \text{len word} \rangle$ 
  by (transfer fixing: take-bit; cases LENGTH('b))
    (auto simp: Bit-Operations.signed-take-bit-take-bit Bit-Operations.take-bit-signed-take-bit
      take-bit-of-int min-def less-Suc-eq)

context
  includes bit-operations-syntax
begin

lemma signed-not-eq:
   $\langle \text{signed } (\text{NOT } w) = \text{signed-take-bit } \text{LENGTH('b)} (\text{NOT } (\text{signed } w)) \rangle$ 
  for  $w :: \langle 'b :: \text{len word} \rangle$ 
  by (simp add: bit-eq-iff bit-simps possible-bit-less-imp min-less-iff-disj)
    (auto simp: min-def)

lemma signed-and-eq:
   $\langle \text{signed } (v \ \text{AND } w) = \text{signed } v \ \text{AND } \text{signed } w \rangle$ 
  for  $v \ w :: \langle 'b :: \text{len word} \rangle$ 
  by (rule bit-eqI) (simp add: bit-signed-iff bit-and-iff Bit-Operations.bit-and-iff)

lemma signed-or-eq:
   $\langle \text{signed } (v \ \text{OR } w) = \text{signed } v \ \text{OR } \text{signed } w \rangle$ 
  for  $v \ w :: \langle 'b :: \text{len word} \rangle$ 
  by (rule bit-eqI) (simp add: bit-signed-iff bit-or-iff Bit-Operations.bit-or-iff)

lemma signed-xor-eq:
   $\langle \text{signed } (v \ \text{XOR } w) = \text{signed } v \ \text{XOR } \text{signed } w \rangle$ 
  for  $v \ w :: \langle 'b :: \text{len word} \rangle$ 
  by (rule bit-eqI) (simp add: bit-signed-iff bit-xor-iff Bit-Operations.bit-xor-iff)

end

end

```


107.6.3 More**lemma** *sint-greater-eq*: $\langle - (2 \wedge (\text{LENGTH}('a) - \text{Suc } 0)) \leq \text{sint } w \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$ **proof** (cases $\langle \text{bit } w (\text{LENGTH}('a) - \text{Suc } 0) \rangle$)**case** *True***then show** *?thesis***by** transfer (simp add: signed-take-bit-eq-if-negative minus-exp-eq-not-mask or-greater-eq ac-simps)**next****have** *: $\langle - (2 \wedge (\text{LENGTH}('a) - \text{Suc } 0)) \leq (0::\text{int}) \rangle$ **by** simp**case** *False***then show** *?thesis***by** transfer (auto simp: signed-take-bit-eq intro: order-trans *)**qed****lemma** *sint-less*: $\langle \text{sint } w < 2 \wedge (\text{LENGTH}('a) - \text{Suc } 0) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$ **by** (cases $\langle \text{bit } w (\text{LENGTH}('a) - \text{Suc } 0) \rangle$; transfer)

(simp-all add: signed-take-bit-eq signed-take-bit-def not-eq-complement mask-eq-exp-minus-1 OR-upper)

lemma *uint-div-distrib*: $\langle \text{uint } (v \text{ div } w) = \text{uint } v \text{ div uint } w \rangle$ **by** (metis int-ops(8) of-nat-unat unat-div-distrib)**lemma** *unat-drop-bit-eq*: $\langle \text{unat } (\text{drop-bit } n \ w) = \text{drop-bit } n \ (\text{unat } w) \rangle$ **by** (rule bit-eqI) (simp add: bit-unsigned-iff bit-drop-bit-eq)**lemma** *uint-mod-distrib*: $\langle \text{uint } (v \text{ mod } w) = \text{uint } v \text{ mod uint } w \rangle$ **by** (metis int-ops(9) of-nat-unat unat-mod-distrib)**context** *semiring-bit-operations***begin****lemma** *unsigned-ucast-eq*: $\langle \text{unsigned } (\text{ucast } w :: 'c::\text{len word}) = \text{take-bit } \text{LENGTH}('c) \ (\text{unsigned } w) \rangle$ **for** $w :: \langle 'b::\text{len word} \rangle$ **by** (rule bit-eqI) (simp add: bit-unsigned-iff Word.bit-unsigned-iff bit-take-bit-iff not-le)**end****context** *ring-bit-operations***begin****lemma** *signed-ucast-eq*:

$\langle \text{signed } (\text{ucast } w :: 'c::\text{len word}) = \text{signed-take-bit } (\text{LENGTH}('c) - \text{Suc } 0) (\text{unsigned } w) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
by (*simp add: bit-eq-iff bit-simps min-less-iff-disj*)

lemma *signed-scast-eq*:

$\langle \text{signed } (\text{scast } w :: 'c::\text{len word}) = \text{signed-take-bit } (\text{LENGTH}('c) - \text{Suc } 0) (\text{signed } w) \rangle$
for $w :: \langle 'b::\text{len word} \rangle$
by (*simp add: bit-eq-iff bit-simps min-less-iff-disj*)

end

lemma *uint-nonnegative*: $0 \leq \text{uint } w$

by (*fact unsigned-greater-eq*)

lemma *uint-bounded*: $\text{uint } w < 2^{\text{LENGTH}('a)}$

for $w :: 'a::\text{len word}$

by (*fact unsigned-less*)

lemma *uint-idem*: $\text{uint } w \bmod 2^{\text{LENGTH}('a)} = \text{uint } w$

for $w :: 'a::\text{len word}$

by (*transfer (simp add: take-bit-eq-mod)*)

lemma *word-uint-eqI*: $\text{uint } a = \text{uint } b \implies a = b$

by (*fact unsigned-word-eqI*)

lemma *word-uint-eq-iff*: $a = b \longleftrightarrow \text{uint } a = \text{uint } b$

by (*fact word-eq-iff-unsigned*)

lemma *uint-word-of-int-eq*:

$\langle \text{uint } (\text{word-of-int } k :: 'a::\text{len word}) = \text{take-bit } \text{LENGTH}('a) \ k \rangle$

by (*transfer rule*)

lemma *uint-word-of-int*: $\text{uint } (\text{word-of-int } k :: 'a::\text{len word}) = k \bmod 2^{\text{LENGTH}('a)}$

by (*simp add: uint-word-of-int-eq take-bit-eq-mod*)

lemma *word-of-int-uint*: $\text{word-of-int } (\text{uint } w) = w$

by (*transfer simp*)

lemma *word-div-def* [*code*]:

$a \text{ div } b = \text{word-of-int } (\text{uint } a \text{ div } \text{uint } b)$

by (*transfer rule*)

lemma *word-mod-def* [*code*]:

$a \bmod b = \text{word-of-int } (\text{uint } a \bmod \text{uint } b)$

by (*transfer rule*)

lemma *split-word-all*: $(\bigwedge x::'a::\text{len word}. \text{PROP } P \ x) \equiv (\bigwedge x. \text{PROP } P \ (\text{word-of-int } x))$

```

x))
proof
  fix  $x :: 'a \text{ word}$ 
  assume  $\bigwedge x. \text{PROP } P \text{ (word-of-int } x)$ 
  then have  $\text{PROP } P \text{ (word-of-int (uint } x))$  .
  then show  $\text{PROP } P \text{ } x$ 
    by (simp only: word-of-int-uint)
qed

lemma sint-uint:
   $\langle \text{sint } w = \text{signed-take-bit (LENGTH('a) - Suc 0) (uint } w) \rangle$ 
  for  $w :: \langle 'a::\text{len word} \rangle$ 
  by (cases  $\langle \text{LENGTH('a)} \rangle$ ; transfer) (simp-all add: signed-take-bit-take-bit)

lemma unat-eq-nat-uint:
   $\langle \text{unat } w = \text{nat (uint } w) \rangle$ 
  by simp

lemma ucast-eq:
   $\langle \text{ucast } w = \text{word-of-int (uint } w) \rangle$ 
  by transfer simp

lemma scast-eq:
   $\langle \text{scast } w = \text{word-of-int (sint } w) \rangle$ 
  by transfer simp

lemma uint-0-eq:
   $\langle \text{uint } 0 = 0 \rangle$ 
  by (fact unsigned-0)

lemma uint-1-eq:
   $\langle \text{uint } 1 = 1 \rangle$ 
  by (fact unsigned-1)

lemma word-m1-wi:  $- 1 = \text{word-of-int } (- 1)$ 
  by simp

lemma uint-0-iff:  $\text{uint } x = 0 \longleftrightarrow x = 0$ 
  by (auto simp: unsigned-word-eqI)

lemma unat-0-iff:  $\text{unat } x = 0 \longleftrightarrow x = 0$ 
  by (auto simp: unsigned-word-eqI)

lemma unat-0:  $\text{unat } 0 = 0$ 
  by (fact unsigned-0)

lemma unat-gt-0:  $0 < \text{unat } x \longleftrightarrow x \neq 0$ 
  by (auto simp: unat-0-iff [symmetric])

```

```

lemma ucast-0: ucast 0 = 0
  by (fact unsigned-0)

lemma sint-0: sint 0 = 0
  by (fact signed-0)

lemma scast-0: scast 0 = 0
  by (fact signed-0)

lemma sint-n1: sint (- 1) = - 1
  by (fact signed-minus-1)

lemma scast-n1: scast (- 1) = - 1
  by (fact signed-minus-1)

lemma uint-1: uint (1::'a::len word) = 1
  by (fact uint-1-eq)

lemma unat-1: unat (1::'a::len word) = 1
  by (fact unsigned-1)

lemma ucast-1: ucast (1::'a::len word) = 1
  by (fact unsigned-1)

instantiation word :: (len) size
begin

lift-definition size-word :: ⟨'a word ⇒ nat⟩
  is ⟨λ-. LENGTH('a)⟩ ..

instance ..

end

lemma word-size [code]:
  ⟨size w = LENGTH('a)⟩ for w :: ⟨'a::len word⟩
  by (fact size-word.rep-eq)

lemma word-size-gt-0 [iff]: 0 < size w
  for w :: 'a::len word
  by (simp add: word-size)

lemmas lens-gt-0 = word-size-gt-0 len-gt-0

lemma lens-not-0 [iff]:
  ⟨size w ≠ 0⟩ for w :: ⟨'a::len word⟩
  by auto

lift-definition source-size :: ⟨('a::len word ⇒ 'b) ⇒ nat⟩

```

is $\langle \lambda-. \text{LENGTH}('a) \rangle .$

lift-definition *target-size* :: $\langle ('a \Rightarrow 'b::\text{len word}) \Rightarrow \text{nat} \rangle$
is $\langle \lambda-. \text{LENGTH}('b) \rangle ..$

lift-definition *is-up* :: $\langle ('a::\text{len word} \Rightarrow 'b::\text{len word}) \Rightarrow \text{bool} \rangle$
is $\langle \lambda-. \text{LENGTH}('a) \leq \text{LENGTH}('b) \rangle ..$

lift-definition *is-down* :: $\langle ('a::\text{len word} \Rightarrow 'b::\text{len word}) \Rightarrow \text{bool} \rangle$
is $\langle \lambda-. \text{LENGTH}('a) \geq \text{LENGTH}('b) \rangle ..$

lemma *is-up-eq*:
 $\langle \text{is-up } f \longleftrightarrow \text{source-size } f \leq \text{target-size } f \rangle$
for $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
by (*simp add: source-size.rep-eq target-size.rep-eq is-up.rep-eq*)

lemma *is-down-eq*:
 $\langle \text{is-down } f \longleftrightarrow \text{target-size } f \leq \text{source-size } f \rangle$
for $f :: \langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
by (*simp add: source-size.rep-eq target-size.rep-eq is-down.rep-eq*)

lift-definition *word-int-case* :: $\langle (\text{int} \Rightarrow 'b) \Rightarrow 'a::\text{len word} \Rightarrow 'b \rangle$
is $\langle \lambda f. f \circ \text{take-bit LENGTH}('a) \rangle$ **by** *simp*

lemma *word-int-case-eq-uint* [*code*]:
 $\langle \text{word-int-case } f \ w = f \ (\text{uint } w) \rangle$
by *transfer simp*

translations

case x of *XCONST* *of-int* $y \Rightarrow b \Rightarrow \text{CONST } \text{word-int-case } (\lambda y. b) \ x$
case x of (*XCONST* *of-int* :: $'a$) $y \Rightarrow b \rightarrow \text{CONST } \text{word-int-case } (\lambda y. b) \ x$

107.7 Arithmetic operations

lemma *div-word-self*:
 $\langle w \text{ div } w = 1 \rangle$ **if** $\langle w \neq 0 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
using *that* **by** *transfer simp*

lemma *mod-word-self* [*simp*]:
 $\langle w \text{ mod } w = 0 \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
by (*simp add: word-mod-def*)

lemma *div-word-less*:
 $\langle w \text{ div } v = 0 \rangle$ **if** $\langle w < v \rangle$ **for** $w \ v :: \langle 'a::\text{len word} \rangle$
using *that* **by** *transfer simp*

lemma *mod-word-less*:
 $\langle w \text{ mod } v = w \rangle$ **if** $\langle w < v \rangle$ **for** $w \ v :: \langle 'a::\text{len word} \rangle$
using *div-mult-mod-eq* [*of w v*] **using** *that* **by** (*simp add: div-word-less*)

lemma *div-word-one* [simp]:
 $\langle 1 \text{ div } w = \text{of_bool } (w = 1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
proof *transfer*
fix $k :: \text{int}$
show $\langle \text{take-bit LENGTH('a)} (\text{take-bit LENGTH('a)} 1 \text{ div take-bit LENGTH('a)} k) =$
 $\text{take-bit LENGTH('a)} (\text{of_bool } (\text{take-bit LENGTH('a)} k = \text{take-bit LENGTH('a)} 1)) \rangle$
using *take-bit-nonnegative* [of $\langle \text{LENGTH('a)} \rangle k$]
by (*smt* (*verit*, *best*) *div-by-1 of-bool-eq take-bit-of-0 take-bit-of-1 zdiv-eq-0-iff*)
qed

lemma *mod-word-one* [simp]:
 $\langle 1 \text{ mod } w = 1 - w * \text{of_bool } (w = 1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
using *div-mult-mod-eq* [of 1 w] **by** *auto*

lemma *div-word-by-minus-1-eq* [simp]:
 $\langle w \text{ div } -1 = \text{of_bool } (w = -1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
by (*auto intro: div-word-less simp add: div-word-self word-order.not-eq-extremum*)

lemma *mod-word-by-minus-1-eq* [simp]:
 $\langle w \text{ mod } -1 = w * \text{of_bool } (w < -1) \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$
using *mod-word-less word-order.not-eq-extremum* **by** *fastforce*

Legacy theorems:

lemma *word-add-def* [code]:
 $a + b = \text{word-of-int } (\text{uint } a + \text{uint } b)$
by *transfer* (*simp add: take-bit-add*)

lemma *word-sub-wi* [code]:
 $a - b = \text{word-of-int } (\text{uint } a - \text{uint } b)$
by *transfer* (*simp add: take-bit-diff*)

lemma *word-mult-def* [code]:
 $a * b = \text{word-of-int } (\text{uint } a * \text{uint } b)$
by *transfer* (*simp add: take-bit-eq-mod mod-simps*)

lemma *word-minus-def* [code]:
 $-a = \text{word-of-int } (-\text{uint } a)$
by *transfer* (*simp add: take-bit-minus*)

lemma *word-0-wi*:
 $0 = \text{word-of-int } 0$
by *transfer simp*

lemma *word-1-wi*:
 $1 = \text{word-of-int } 1$
by *transfer simp*

lift-definition *word-succ* :: 'a::len word \Rightarrow 'a word **is** $\lambda x. x + 1$
by (auto simp: take-bit-eq-mod intro: mod-add-cong)

lift-definition *word-pred* :: 'a::len word \Rightarrow 'a word **is** $\lambda x. x - 1$
by (auto simp: take-bit-eq-mod intro: mod-diff-cong)

lemma *word-succ-alt* [code]:
word-succ a = word-of-int (uint a + 1)
by transfer (simp add: take-bit-eq-mod mod-simps)

lemma *word-pred-alt* [code]:
word-pred a = word-of-int (uint a - 1)
by transfer (simp add: take-bit-eq-mod mod-simps)

lemmas *word-arith-wis* =
word-add-def word-sub-wi word-mult-def
word-minus-def word-succ-alt word-pred-alt
word-0-wi word-1-wi

lemma *wi-homs*:
shows *wi-hom-add*: *word-of-int a + word-of-int b = word-of-int (a + b)*
and *wi-hom-sub*: *word-of-int a - word-of-int b = word-of-int (a - b)*
and *wi-hom-mult*: *word-of-int a * word-of-int b = word-of-int (a * b)*
and *wi-hom-neg*: *- word-of-int a = word-of-int (- a)*
and *wi-hom-succ*: *word-succ (word-of-int a) = word-of-int (a + 1)*
and *wi-hom-pred*: *word-pred (word-of-int a) = word-of-int (a - 1)*
by (transfer, simp)+

lemmas *wi-hom-syms* = *wi-homs* [symmetric]

lemmas *word-of-int-homs* = *wi-homs word-0-wi word-1-wi*

lemmas *word-of-int-hom-syms* = *word-of-int-homs* [symmetric]

lemma *double-eq-zero-iff*:
 $\langle 2 * a = 0 \longleftrightarrow a = 0 \vee a = 2 \wedge (\text{LENGTH}('a) - \text{Suc } 0) \rangle$
for *a* :: 'a::len word
proof -
define *n* **where** $\langle n = \text{LENGTH}('a) - \text{Suc } 0 \rangle$
then have *: $\langle \text{LENGTH}('a) = \text{Suc } n \rangle$
by simp
have $\langle a = 0 \rangle$ **if** $\langle 2 * a = 0 \rangle$ **and** $\langle a \neq 2 \wedge (\text{LENGTH}('a) - \text{Suc } 0) \rangle$
using that by transfer
(auto simp: take-bit-eq-0-iff take-bit-eq-mod *)
moreover have $\langle 2 \wedge \text{LENGTH}('a) = (0 :: 'a \text{ word}) \rangle$
by transfer simp
then have $\langle 2 * 2 \wedge (\text{LENGTH}('a) - \text{Suc } 0) = (0 :: 'a \text{ word}) \rangle$
by (simp add: *)

ultimately show ?thesis
 by auto
 qed

107.8 Ordering

instance word :: (len) wellorder

proof

fix P :: 'a word \Rightarrow bool **and** a
 assume *: ($\bigwedge b. (\bigwedge a. a < b \implies P a) \implies P b$)
 have wf (measure unat) ..
 moreover have $\{(a, b :: ('a::len) \text{ word}). a < b\} \subseteq \text{measure unat}$
 by (auto simp add: word-less-iff-unsigned [where ?'a = nat])
 ultimately have wf $\{(a, b :: ('a::len) \text{ word}). a < b\}$
 by (rule wf-subset)
 then show P a using *
 by induction blast
 qed

lemma word-m1-ge [simp]:
 word-pred 0 \geq y
 by transfer (simp add: mask-eq-exp-minus-1)

lemma word-less-alt:
 a < b \longleftrightarrow uint a < uint b
 by (fact word-less-def)

lemma word-zero-le [simp]:
 0 \leq y **for** y :: 'a::len word
 by (fact word-coorder.extremum)

lemma word-n1-ge [simp]:
 y \leq - 1 **for** y :: 'a::len word
 by (fact word-order.extremum)

lemmas word-not-simps [simp] =
 word-zero-le [THEN leD] word-m1-ge [THEN leD] word-n1-ge [THEN leD]

lemma word-gt-0:
 0 < y \longleftrightarrow 0 \neq y
for y :: 'a::len word
 by (simp add: less-le)

lemma word-gt-0-no [simp]:
 $\langle (0 :: 'a::len \text{ word}) < \text{numeral } y \longleftrightarrow (0 :: 'a::len \text{ word}) \neq \text{numeral } y \rangle$
 by (fact word-gt-0)

lemma word-le-nat-alt:
 a \leq b \longleftrightarrow unat a \leq unat b

by *transfer* (*simp add: nat-le-eq-zle*)

lemma *word-less-nat-alt*:

$a < b \longleftrightarrow \text{unat } a < \text{unat } b$

by *transfer* (*auto simp: less-le [of 0]*)

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1*]

lemma *wi-less*:

$(\text{word-of-int } n < (\text{word-of-int } m :: 'a::\text{len word})) =$
 $(n \bmod 2 \wedge \text{LENGTH}('a) < m \bmod 2 \wedge \text{LENGTH}('a))$

by (*simp add: uint-word-of-int word-less-def*)

lemma *wi-le*:

$(\text{word-of-int } n \leq (\text{word-of-int } m :: 'a::\text{len word})) =$
 $(n \bmod 2 \wedge \text{LENGTH}('a) \leq m \bmod 2 \wedge \text{LENGTH}('a))$

by (*simp add: uint-word-of-int word-le-def*)

lift-definition *word-sle* :: $\langle 'a::\text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$

is $\langle \lambda k l. \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) k \leq \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) l \rangle$

by (*simp flip: signed-take-bit-decr-length-iff*)

lift-definition *word-sless* :: $\langle 'a::\text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool} \rangle$

is $\langle \lambda k l. \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) k < \text{signed-take-bit} (\text{LENGTH}('a) - \text{Suc } 0) l \rangle$

by (*simp flip: signed-take-bit-decr-length-iff*)

notation

word-sle $\langle '(\leq s') \rangle$ **and**

word-sle $\langle (\langle \text{notation} = \langle \text{infix } \leq s \rangle \rangle - / \leq s -) \rangle$ [51, 51] 50) **and**

word-sless $\langle '(< s') \rangle$ **and**

word-sless $\langle (\langle \text{notation} = \langle \text{infix } < s \rangle \rangle - / < s -) \rangle$ [51, 51] 50)

notation (*input*)

word-sle $\langle (\langle \text{notation} = \langle \text{infix } \leq s \rangle \rangle - / \leq s -) \rangle$ [51, 51] 50)

lemma *word-sle-eq* [*code*]:

$\langle a \leq_s b \longleftrightarrow \text{sint } a \leq \text{sint } b \rangle$

by *transfer simp*

lemma *word-sless-alt* [*code*]:

$a <_s b \longleftrightarrow \text{sint } a < \text{sint } b$

by *transfer simp*

lemma *signed-ordering*: $\langle \text{ordering word-sle word-sless} \rangle$

by (*standard; transfer*) (*auto simp flip: signed-take-bit-decr-length-iff*)

lemma *signed-linorder*: $\langle \text{class.linorder word-sle word-sless} \rangle$

by (*standard; transfer*) (*auto simp: signed-take-bit-decr-length-iff*)

interpretation *signed: linorder word-sle word-sless*
by (*fact signed-linorder*)

lemma *word-sless-eq*:
 $\langle x <_s y \iff x \leq_s y \wedge x \neq y \rangle$
by (*fact signed.less-le*)

lemma *minus-1-sless-0* [*simp*]:
 $\langle -1 <_s 0 \rangle$
by *transfer simp*

lemma *not-0-sless-minus-1* [*simp*]:
 $\langle \neg 0 <_s -1 \rangle$
by *transfer simp*

lemma *minus-1-sless-eq-0* [*simp*]:
 $\langle -1 \leq_s 0 \rangle$
by *transfer simp*

lemma *not-0-sless-eq-minus-1* [*simp*]:
 $\langle \neg 0 \leq_s -1 \rangle$
by *transfer simp*

107.9 Bit-wise operations

context
includes *bit-operations-syntax*
begin

lemma *take-bit-word-eq-self*:
 $\langle \text{take-bit } n \ w = w \rangle$ **if** $\langle \text{LENGTH}(a) \leq n \rangle$ **for** $w :: \langle a::\text{len word} \rangle$
using that **by** *transfer simp*

lemma *take-bit-length-eq* [*simp*]:
 $\langle \text{take-bit } \text{LENGTH}(a) \ w = w \rangle$ **for** $w :: \langle a::\text{len word} \rangle$
by (*simp add: nat-le-linear take-bit-word-eq-self*)

lemma *bit-word-of-int-iff*:
 $\langle \text{bit } (\text{word-of-int } k :: a::\text{len word}) \ n \iff n < \text{LENGTH}(a) \wedge \text{bit } k \ n \rangle$
by (*simp add: bit-simps*)

lemma *bit-wint-iff*:
 $\langle \text{bit } (\text{uint } w) \ n \iff n < \text{LENGTH}(a) \wedge \text{bit } w \ n \rangle$
for $w :: \langle a::\text{len word} \rangle$
by *transfer (simp add: bit-take-bit-iff)*

lemma *bit-sint-iff*:

$\langle \text{bit } (\text{shint } w) \ n \longleftrightarrow n \geq \text{LENGTH}('a) \wedge \text{bit } w \ (\text{LENGTH}('a) - 1) \vee \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by *transfer (auto simp: bit-signed-take-bit-iff min-def le-less not-less)*

lemma *bit-word-ucast-iff*:

$\langle \text{bit } (\text{ucast } w :: 'b::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}('a) \wedge n < \text{LENGTH}('b) \wedge \text{bit } w \ n \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by *(auto simp add: bit-unsigned-iff bit-imp-le-length)*

lemma *bit-word-scast-iff*:

$\langle \text{bit } (\text{scast } w :: 'b::\text{len word}) \ n \longleftrightarrow$
 $n < \text{LENGTH}('b) \wedge (\text{bit } w \ n \vee \text{LENGTH}('a) \leq n \wedge \text{bit } w \ (\text{LENGTH}('a) -$
 $\text{Suc } 0)) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by *(auto simp add: bit-signed-iff bit-imp-le-length min-def le-less dest: bit-imp-possible-bit)*

lemma *bit-word-iff-drop-bit-and* [code]:

$\langle \text{bit } a \ n \longleftrightarrow \text{drop-bit } n \ a \ \text{AND } 1 = 1 \rangle$ **for** $a :: \langle 'a::\text{len word} \rangle$
by *(simp add: bit-iff-odd-drop-bit odd-iff-mod-2-eq-one and-one-eq)*

lemma

$\text{word-not-def: } \text{NOT } (a::'a::\text{len word}) = \text{word-of-int } (\text{NOT } (\text{uint } a))$
and $\text{word-and-def: } (a::'a \ \text{word}) \ \text{AND } b = \text{word-of-int } (\text{uint } a \ \text{AND } \text{uint } b)$
and $\text{word-or-def: } (a::'a \ \text{word}) \ \text{OR } b = \text{word-of-int } (\text{uint } a \ \text{OR } \text{uint } b)$
and $\text{word-xor-def: } (a::'a \ \text{word}) \ \text{XOR } b = \text{word-of-int } (\text{uint } a \ \text{XOR } \text{uint } b)$
by *(transfer, simp add: take-bit-not-take-bit)+*

definition *even-word* :: $\langle 'a::\text{len word} \Rightarrow \text{bool} \rangle$

where [code-abbrev]: $\langle \text{even-word} = \text{even} \rangle$

lemma *even-word-iff* [code]:

$\langle \text{even-word } a \longleftrightarrow a \ \text{AND } 1 = 0 \rangle$
by *(simp add: and-one-eq even-iff-mod-2-eq-zero even-word-def)*

lemma *map-bit-range-eq-if-take-bit-eq*:

$\langle \text{map } (\text{bit } k) \ [0..<n] = \text{map } (\text{bit } l) \ [0..<n] \rangle$
if $\langle \text{take-bit } n \ k = \text{take-bit } n \ l \rangle$ **for** $k \ l :: \text{int}$
using *that*

proof *(induction n arbitrary: k l)*

case 0

then show *?case*

by *simp*

next

case *(Suc n)*

from *Suc.prem*s **have** $\langle \text{take-bit } n \ (k \ \text{div } 2) = \text{take-bit } n \ (l \ \text{div } 2) \rangle$

by *(simp add: take-bit-Suc)*

then have $\langle \text{map } (\text{bit } (k \ \text{div } 2)) \ [0..<n] = \text{map } (\text{bit } (l \ \text{div } 2)) \ [0..<n] \rangle$

by *(rule Suc.IH)*

```

moreover have  $\langle \text{bit } (r \text{ div } 2) = \text{bit } r \circ \text{Suc} \rangle$  for  $r :: \text{int}$ 
  by (simp add: fun-eq-iff bit-Suc)
moreover from Suc.prems have  $\langle \text{even } k \longleftrightarrow \text{even } l \rangle$ 
  by (metis Zero-neq-Suc even-take-bit-eq)
ultimately show ?case
  by (simp only: map-Suc-upt upt-conv-Cons flip: list.map-comp) (simp add:
bit-0)
qed

```

lemma

```

  take-bit-word-Bit0-eq [simp]:  $\langle \text{take-bit } (\text{numeral } n) (\text{numeral } (\text{num.Bit0 } m)) ::$ 
'a::len word
     $= 2 * \text{take-bit } (\text{pred-numeral } n) (\text{numeral } m) \rangle$  (is ?P)
  and take-bit-word-Bit1-eq [simp]:  $\langle \text{take-bit } (\text{numeral } n) (\text{numeral } (\text{num.Bit1 } m)) ::$ 
'a::len word
     $= 1 + 2 * \text{take-bit } (\text{pred-numeral } n) (\text{numeral } m) \rangle$  (is ?Q)
  and take-bit-word-minus-Bit0-eq [simp]:  $\langle \text{take-bit } (\text{numeral } n) (- \text{numeral } (\text{num.Bit0 } m)) ::$ 
'a::len word
     $= 2 * \text{take-bit } (\text{pred-numeral } n) (- \text{numeral } m) \rangle$  (is ?R)
  and take-bit-word-minus-Bit1-eq [simp]:  $\langle \text{take-bit } (\text{numeral } n) (- \text{numeral } (\text{num.Bit1 } m)) ::$ 
'a::len word
     $= 1 + 2 * \text{take-bit } (\text{pred-numeral } n) (- \text{numeral } (\text{Num.inc } m)) \rangle$  (is ?S)

```

proof –

```

  define  $w :: \langle 'a::len \text{ word} \rangle$ 
    where  $\langle w = \text{numeral } m \rangle$ 
  moreover define  $q :: \text{nat}$ 
    where  $\langle q = \text{pred-numeral } n \rangle$ 
  ultimately have num:
     $\langle \text{numeral } m = w \rangle$ 
     $\langle \text{numeral } (\text{num.Bit0 } m) = 2 * w \rangle$ 
     $\langle \text{numeral } (\text{num.Bit1 } m) = 1 + 2 * w \rangle$ 
     $\langle \text{numeral } (\text{Num.inc } m) = 1 + w \rangle$ 
     $\langle \text{pred-numeral } n = q \rangle$ 
     $\langle \text{numeral } n = \text{Suc } q \rangle$ 
  by (simp-all only: w-def q-def numeral-Bit0 [of m] numeral-Bit1 [of m] ac-simps
    numeral-inc numeral-eq-Suc flip: mult-2)
  have even:  $\langle \text{take-bit } (\text{Suc } q) (2 * w) = 2 * \text{take-bit } q w \rangle$  for  $w :: \langle 'a::len \text{ word} \rangle$ 
    by (rule bit-word-eqI)
    (auto simp: bit-take-bit-iff bit-double-iff)
  have odd:  $\langle \text{take-bit } (\text{Suc } q) (1 + 2 * w) = 1 + 2 * \text{take-bit } q w \rangle$  for  $w :: \langle 'a::len \text{ word} \rangle$ 
    by (rule bit-eqI)
    (auto simp: bit-take-bit-iff bit-double-iff even-bit-succ-iff)
  show ?P
    using even [of w] by (simp add: num)
  show ?Q
    using odd [of w] by (simp add: num)
  show ?R
    using even [of  $\langle - w \rangle$ ] by (simp add: num)

```

```

show ?S
  using odd [of  $\leftarrow (1 + w)$ ] by (simp add: num)
qed

```

107.10 More shift operations

```

lift-definition signed-drop-bit ::  $\langle \text{nat} \Rightarrow 'a \text{ word} \Rightarrow 'a::\text{len word} \rangle$ 
  is  $\langle \lambda n. \text{drop-bit } n \circ \text{signed-take-bit } (\text{LENGTH}('a) - \text{Suc } 0) \rangle$ 
  using signed-take-bit-decr-length-iff
  by (simp add: take-bit-drop-bit) force

```

```

lemma bit-signed-drop-bit-iff [bit-simps]:
   $\langle \text{bit } (\text{signed-drop-bit } m \ w) \ n \longleftrightarrow \text{bit } w \ (if \ \text{LENGTH}('a) - m \leq n \wedge n < \text{LENGTH}('a) \text{ then } \text{LENGTH}('a) - 1 \text{ else } m + n) \rangle$ 
  for  $w :: 'a::\text{len word}$ 
  by transfer (simp add: bit-drop-bit-eq bit-signed-take-bit-iff min-def le-less not-less,
auto)

```

```

lemma [code]:
   $\langle \text{Word.the-int } (\text{signed-drop-bit } n \ w) = \text{take-bit } \text{LENGTH}('a) \ (\text{drop-bit } n \ (\text{Word.the-signed-int } w)) \rangle$ 
  for  $w :: 'a::\text{len word}$ 
  by transfer simp

```

```

lemma signed-drop-bit-of-0 [simp]:
   $\langle \text{signed-drop-bit } n \ 0 = 0 \rangle$ 
  by transfer simp

```

```

lemma signed-drop-bit-of-minus-1 [simp]:
   $\langle \text{signed-drop-bit } n \ (-1) = -1 \rangle$ 
  by transfer simp

```

```

lemma signed-drop-bit-signed-drop-bit [simp]:
   $\langle \text{signed-drop-bit } m \ (\text{signed-drop-bit } n \ w) = \text{signed-drop-bit } (m + n) \ w \rangle$ 
  for  $w :: 'a::\text{len word}$ 

```

```

proof (cases  $\langle \text{LENGTH}('a) \rangle$ )
case 0
  then show ?thesis
    using len-not-eq-0 by blast
next
case (Suc n)
  then show ?thesis
    by (force simp: bit-signed-drop-bit-iff not-le less-diff-conv ac-simps intro!: bit-word-eqI)
qed

```

```

lemma signed-drop-bit-0 [simp]:
   $\langle \text{signed-drop-bit } 0 \ w = w \rangle$ 
  by transfer (simp add: take-bit-signed-take-bit)

```

lemma *sint-signed-drop-bit-eq*:
 $\langle \text{sint } (\text{signed-drop-bit } n \ w) = \text{drop-bit } n \ (\text{sint } w) \rangle$
by (rule *bit-eqI*; cases *n*) (auto simp add: *bit-simps not-less*)

107.11 Single-bit operations

lemma *set-bit-eq-idem-iff*:
 $\langle \text{set-bit } n \ w = w \longleftrightarrow \text{bit } w \ n \vee n \geq \text{LENGTH}('a) \rangle$
for $w :: \langle 'a :: \text{len word} \rangle$
unfolding *bit-eq-iff*
by (auto simp: *bit-simps not-le*)

lemma *unset-bit-eq-idem-iff*:
 $\langle \text{unset-bit } n \ w = w \longleftrightarrow \text{bit } w \ n \longrightarrow n \geq \text{LENGTH}('a) \rangle$
for $w :: \langle 'a :: \text{len word} \rangle$
unfolding *bit-eq-iff*
by (auto simp: *bit-simps dest: bit-imp-le-length*)

lemma *flip-bit-eq-idem-iff*:
 $\langle \text{flip-bit } n \ w = w \longleftrightarrow n \geq \text{LENGTH}('a) \rangle$
for $w :: \langle 'a :: \text{len word} \rangle$
by (simp add: *flip-bit-eq-if set-bit-eq-idem-iff unset-bit-eq-idem-iff*)

107.12 Rotation

lift-definition *word-rotr* :: $\langle \text{nat} \Rightarrow 'a :: \text{len word} \Rightarrow 'a :: \text{len word} \rangle$
is $\langle \lambda n \ k. \text{concat-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a))$
 $(\text{drop-bit } (n \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (n \bmod \text{LENGTH}('a)) \ k) \rangle$
using *take-bit-tightened* **by** *fastforce*

lift-definition *word-rotl* :: $\langle \text{nat} \Rightarrow 'a :: \text{len word} \Rightarrow 'a :: \text{len word} \rangle$
is $\langle \lambda n \ k. \text{concat-bit } (n \bmod \text{LENGTH}('a))$
 $(\text{drop-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) \ k) \rangle$
using *take-bit-tightened* **by** *fastforce*

lift-definition *word-roti* :: $\langle \text{int} \Rightarrow 'a :: \text{len word} \Rightarrow 'a :: \text{len word} \rangle$
is $\langle \lambda r \ k. \text{concat-bit } (\text{LENGTH}('a) - \text{nat } (r \bmod \text{int } \text{LENGTH}('a)))$
 $(\text{drop-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (\text{nat } (r \bmod \text{int } \text{LENGTH}('a))) \ k) \rangle$
by (smt (verit, best) *len-gt-0 nat-le-iff of-nat-0-less-iff pos-mod-bound*
take-bit-tightened)

lemma *word-rotl-eq-word-rotr* [code]:
 $\langle \text{word-rotl } n = (\text{word-rotr } (\text{LENGTH}('a) - n \bmod \text{LENGTH}('a)) :: 'a :: \text{len word}$
 $\Rightarrow 'a \ \text{word}) \rangle$
by (rule *ext*, cases $\langle n \bmod \text{LENGTH}('a) = 0 \rangle$; transfer) *simp-all*

lemma *word-roti-eq-word-rotr-word-rotl* [code]:

```

  ⟨word-roti i w =
    (if i ≥ 0 then word-rotr (nat i) w else word-rotl (nat (− i)) w)⟩
proof (cases ⟨i ≥ 0⟩)
  case True
  moreover define n where ⟨n = nat i⟩
  ultimately have ⟨i = int n⟩
    by simp
  moreover have ⟨word-roti (int n) = (word-rotr n :: - ⇒ 'a word)⟩
    by (rule ext, transfer) (simp add: nat-mod-distrib)
  ultimately show ?thesis
    by simp
next
case False
moreover define n where ⟨n = nat (− i)⟩
ultimately have ⟨i = − int n⟩ ⟨n > 0⟩
  by simp-all
moreover have ⟨word-roti (− int n) = (word-rotl n :: - ⇒ 'a word)⟩
  by (rule ext, transfer)
    (simp add: zmod-zminus1-eq-if flip: of-nat-mod of-nat-diff)
ultimately show ?thesis
  by simp
qed

lemma bit-word-rotr-iff [bit-simps]:
  ⟨bit (word-rotr m w) n ⟷
    n < LENGTH('a) ∧ bit w ((n + m) mod LENGTH('a))⟩
  for w :: ⟨'a::len word⟩
proof transfer
  fix k :: int and m n :: nat
  define q where ⟨q = m mod LENGTH('a)⟩
  have ⟨q < LENGTH('a)⟩
    by (simp add: q-def)
  then have ⟨q ≤ LENGTH('a)⟩
    by simp
  have ⟨m mod LENGTH('a) = q⟩
    by (simp add: q-def)
  moreover have ⟨(n + m) mod LENGTH('a) = (n + q) mod LENGTH('a)⟩
    by (subst mod-add-right-eq [symmetric]) (simp add: ⟨m mod LENGTH('a) =
q⟩)
  moreover have ⟨n < LENGTH('a) ∧
    bit (concat-bit (LENGTH('a) − q) (drop-bit q (take-bit LENGTH('a) k))
(take-bit q k)) n ⟷
    n < LENGTH('a) ∧ bit k ((n + q) mod LENGTH('a))⟩
  using ⟨q < LENGTH('a)⟩
  by (cases ⟨q + n ≥ LENGTH('a)⟩)
    (auto simp: bit-concat-bit-iff bit-drop-bit-eq
      bit-take-bit-iff le-mod-geq ac-simps)
  ultimately show ⟨n < LENGTH('a) ∧
    bit (concat-bit (LENGTH('a) − m mod LENGTH('a))

```

$(\text{drop-bit } (m \bmod \text{LENGTH}('a)) (\text{take-bit } \text{LENGTH}('a) \ k))$
 $(\text{take-bit } (m \bmod \text{LENGTH}('a)) \ k)) \ n$
 $\longleftrightarrow n < \text{LENGTH}('a) \wedge$
 $(n + m) \bmod \text{LENGTH}('a) < \text{LENGTH}('a) \wedge$
 $\text{bit } k ((n + m) \bmod \text{LENGTH}('a))\rangle$
 by *simp*
 qed

lemma *bit-word-rotl-iff* [*bit-simps*]:

$\langle \text{bit } (\text{word-rotl } m \ w) \ n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } w ((n + (\text{LENGTH}('a) - m \bmod \text{LENGTH}('a))) \bmod$
 $\text{LENGTH}('a))\rangle$
 for $w :: \langle 'a :: \text{len } \text{word} \rangle$
 by (*simp add: word-rotl-eq-word-rotr bit-word-rotr-iff*)

lemma *bit-word-roti-iff* [*bit-simps*]:

$\langle \text{bit } (\text{word-roti } k \ w) \ n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } w (\text{nat } ((\text{int } n + k) \bmod \text{int } \text{LENGTH}('a)))\rangle$
 for $w :: \langle 'a :: \text{len } \text{word} \rangle$

proof *transfer*

fix $k \ l :: \text{int}$ and $n :: \text{nat}$
 define m where $\langle m = \text{nat } (k \bmod \text{int } \text{LENGTH}('a)) \rangle$
 have $\langle m < \text{LENGTH}('a) \rangle$
 by (*simp add: nat-less-iff m-def*)
 then have $\langle m \leq \text{LENGTH}('a) \rangle$
 by *simp*
 have $\langle k \bmod \text{int } \text{LENGTH}('a) = \text{int } m \rangle$
 by (*simp add: nat-less-iff m-def*)
 moreover have $\langle (\text{int } n + k) \bmod \text{int } \text{LENGTH}('a) = \text{int } ((n + m) \bmod$
 $\text{LENGTH}('a)) \rangle$
 by (*subst mod-add-right-eq [symmetric] (simp add: of-nat-mod k mod int*
 $\text{LENGTH}('a) = \text{int } m)$)
 moreover have $\langle n < \text{LENGTH}('a) \wedge$
 $\text{bit } (\text{concat-bit } (\text{LENGTH}('a) - m) (\text{drop-bit } m (\text{take-bit } \text{LENGTH}('a) \ l))$
 $(\text{take-bit } m \ l)) \ n \longleftrightarrow$
 $n < \text{LENGTH}('a) \wedge \text{bit } l ((n + m) \bmod \text{LENGTH}('a))\rangle$
 using $\langle m < \text{LENGTH}('a) \rangle$
 by (*cases m + n ≥ LENGTH('a)*)
 (*auto simp: bit-concat-bit-iff bit-drop-bit-eq*
bit-take-bit-iff nat-less-iff not-le not-less ac-simps
le-diff-conv le-mod-geq)
 ultimately show $\langle n < \text{LENGTH}('a)$
 $\wedge \text{bit } (\text{concat-bit } (\text{LENGTH}('a) - \text{nat } (k \bmod \text{int } \text{LENGTH}('a)))$
 $(\text{drop-bit } (\text{nat } (k \bmod \text{int } \text{LENGTH}('a))) (\text{take-bit } \text{LENGTH}('a) \ l))$
 $(\text{take-bit } (\text{nat } (k \bmod \text{int } \text{LENGTH}('a))) \ l)) \ n \longleftrightarrow$
 $n < \text{LENGTH}('a)$
 $\wedge \text{nat } ((\text{int } n + k) \bmod \text{int } \text{LENGTH}('a)) < \text{LENGTH}('a)$
 $\wedge \text{bit } l (\text{nat } ((\text{int } n + k) \bmod \text{int } \text{LENGTH}('a)))\rangle$
 by *simp*

qed

lemma *uint-word-rotr-eq*:

⟨uint (word-rotr n w) = concat-bit (LENGTH('a) - n mod LENGTH('a))
 (drop-bit (n mod LENGTH('a)) (uint w))
 (uint (take-bit (n mod LENGTH('a)) w))⟩
for w :: ⟨'a::len word⟩
by transfer (simp add: take-bit-concat-bit-eq)

lemma [code]:

⟨Word.the-int (word-rotr n w) = concat-bit (LENGTH('a) - n mod LENGTH('a))
 (drop-bit (n mod LENGTH('a)) (Word.the-int w))
 (Word.the-int (take-bit (n mod LENGTH('a)) w))⟩
for w :: ⟨'a::len word⟩
using uint-word-rotr-eq [of n w] **by** simp

107.13 Split and cat operations

lift-definition *word-cat* :: ⟨'a::len word ⇒ 'b::len word ⇒ 'c::len word⟩

is ⟨λk l. concat-bit LENGTH('b) l (take-bit LENGTH('a) k)⟩

by (simp add: bit-eq-iff bit-concat-bit-iff bit-take-bit-iff)

lemma *word-cat-eq*:

⟨(word-cat v w :: 'c::len word) = push-bit LENGTH('b) (ucast v) + ucast w⟩
for v :: ⟨'a::len word⟩ **and** w :: ⟨'b::len word⟩
by transfer (simp add: concat-bit-eq ac-simps)

lemma *word-cat-eq'* [code]:

⟨word-cat a b = word-of-int (concat-bit LENGTH('b) (uint b) (uint a))⟩
for a :: ⟨'a::len word⟩ **and** b :: ⟨'b::len word⟩
by transfer (simp add: concat-bit-take-bit-eq)

lemma *bit-word-cat-iff* [bit-simps]:

⟨bit (word-cat v w :: 'c::len word) n ⟷ n < LENGTH('c) ∧ (if n < LENGTH('b)
 then bit w n else bit v (n - LENGTH('b)))⟩
for v :: ⟨'a::len word⟩ **and** w :: ⟨'b::len word⟩
by transfer (simp add: bit-concat-bit-iff bit-take-bit-iff)

definition *word-split* :: ⟨'a::len word ⇒ 'b::len word × 'c::len word⟩

where ⟨word-split w =
 (ucast (drop-bit LENGTH('c) w) :: 'b::len word, ucast w :: 'c::len word)⟩

definition *word-rcat* :: ⟨'a::len word list ⇒ 'b::len word⟩

where ⟨word-rcat = word-of-int ∘ horner-sum uint (2 ^ LENGTH('a)) ∘ rev⟩

107.14 More on conversions

lemma *int-word-sint*:

⟨sint (word-of-int x :: 'a::len word) = (x + 2 ^ (LENGTH('a) - 1)) mod 2 ^
 LENGTH('a) - 2 ^ (LENGTH('a) - 1)⟩

by *transfer (simp flip: take-bit-eq-mod add: signed-take-bit-eq-take-bit-shift)*

lemma *sint-sbintrunc'*: *sint (word-of-int bin :: 'a word) = signed-take-bit (LENGTH('a::len) - 1) bin*
by *(simp add: signed-of-int)*

lemma *uint-sint*: *uint w = take-bit LENGTH('a) (sint w)*
for *w :: 'a::len word*
by *transfer (simp add: take-bit-signed-take-bit)*

lemma *bintr-uint*: *LENGTH('a) ≤ n ⇒ take-bit n (uint w) = uint w*
for *w :: 'a::len word*
by *transfer (simp add: min-def)*

lemma *wi-bintr*:
 $LENGTH('a::len) ≤ n ⇒$
 $word-of-int (take-bit n w) = (word-of-int w :: 'a word)$
by *transfer simp*

lemma *word-numeral-alt*: *numeral b = word-of-int (numeral b)*
by *simp*

declare *word-numeral-alt* [*symmetric, code-abbrev*]

lemma *word-neg-numeral-alt*: $- numeral b = word-of-int (- numeral b)$
by *simp*

declare *word-neg-numeral-alt* [*symmetric, code-abbrev*]

lemma *uint-bintrunc* [*simp*]:
 $uint (numeral bin :: 'a word) = take-bit LENGTH('a::len) (numeral bin)$
by *transfer rule*

lemma *uint-bintrunc-neg* [*simp*]:
 $uint (- numeral bin :: 'a word) = take-bit LENGTH('a::len) (- numeral bin)$
by *transfer rule*

lemma *sint-sbintrunc* [*simp*]:
 $sint (numeral bin :: 'a word) = signed-take-bit (LENGTH('a::len) - 1) (numeral bin)$
by *transfer simp*

lemma *sint-sbintrunc-neg* [*simp*]:
 $sint (- numeral bin :: 'a word) = signed-take-bit (LENGTH('a::len) - 1) (- numeral bin)$
by *transfer simp*

lemma *unat-bintrunc* [*simp*]:
 $unat (numeral bin :: 'a::len word) = take-bit LENGTH('a) (numeral bin)$

by *transfer simp*

lemma *unat-bintrunc-neg [simp]*:

unat ($-$ numeral *bin* :: '*a*::len word) = *nat* (*take-bit* *LENGTH*('a) ($-$ numeral *bin*))

by *transfer simp*

lemma *size-0-eq*: *size* *w* = 0 \implies *v* = *w*

for *v w* :: '*a*::len word

by *transfer simp*

lemma *uint-ge-0 [iff]*: 0 \leq *uint* *x*

by (*fact unsigned-greater-eq*)

lemma *uint-lt2p [iff]*: *uint* *x* < 2 \wedge *LENGTH*('a)

for *x* :: '*a*::len word

by (*fact unsigned-less*)

lemma *sint-ge*: $-$ (2 \wedge (*LENGTH*('a) $-$ 1)) \leq *sint* *x*

for *x* :: '*a*::len word

using *sint-greater-eq [of x]* **by** *simp*

lemma *sint-lt*: *sint* *x* < 2 \wedge (*LENGTH*('a) $-$ 1)

for *x* :: '*a*::len word

using *sint-less [of x]* **by** *simp*

lemma *uint-m2p-neg*: *uint* *x* $-$ 2 \wedge *LENGTH*('a) < 0

for *x* :: '*a*::len word

by (*simp only: diff-less-0-iff-less uint-lt2p*)

lemma *uint-m2p-not-non-neg*: \neg 0 \leq *uint* *x* $-$ 2 \wedge *LENGTH*('a)

for *x* :: '*a*::len word

by (*simp only: not-le uint-m2p-neg*)

lemma *lt2p-lem*: *LENGTH*('a) \leq *n* \implies *uint* *w* < 2 \wedge *n*

for *w* :: '*a*::len word

using *uint-bounded [of w]* **by** (*rule less-le-trans*) *simp*

lemma *uint-le-0-iff [simp]*: *uint* *x* \leq 0 \iff *uint* *x* = 0

by (*fact uint-ge-0 [THEN leD, THEN antisym-conv1]*)

lemma *uint-nat*: *uint* *w* = *int* (*unat* *w*)

by *transfer simp*

lemma *uint-numeral*: *uint* (numeral *b* :: '*a*::len word) = numeral *b* mod 2 \wedge *LENGTH*('a)

by (*simp flip: take-bit-eq-mod add: of-nat-take-bit*)

lemma *uint-neg-numeral*: *uint* ($-$ numeral *b* :: '*a*::len word) = $-$ numeral *b* mod

$2^{\wedge} \text{LENGTH}('a)$
by (*simp flip: take-bit-eq-mod add: of-nat-take-bit*)

lemma *unat-numeral*: *unat* (*numeral* *b* :: '*a*::len word) = *numeral* *b* mod $2^{\wedge} \text{LENGTH}('a)$
by *transfer* (*simp add: take-bit-eq-mod nat-mod-distrib nat-power-eq*)

lemma *sint-numeral*:
sint (*numeral* *b* :: '*a*::len word) =
 (*numeral* *b* + $2^{\wedge}(\text{LENGTH}('a) - 1)$) mod $2^{\wedge} \text{LENGTH}('a) - 2^{\wedge}(\text{LENGTH}('a) - 1)$
by (*metis int-word-sint word-numeral-alt*)

lemma *word-of-int-0* [*simp, code-post*]: *word-of-int* 0 = 0
by (*fact of-int-0*)

lemma *word-of-int-1* [*simp, code-post*]: *word-of-int* 1 = 1
by (*fact of-int-1*)

lemma *word-of-int-neg-1* [*simp*]: *word-of-int* (− 1) = − 1
by *simp*

lemma *word-of-int-numeral* [*simp*] : (*word-of-int* (*numeral* *bin*) :: '*a*::len word) =
numeral *bin*
by *simp*

lemma *word-int-case-wi*:
word-int-case *f* (*word-of-int* *i* :: '*b*::len word) = *f* (*i* mod $2^{\wedge} \text{LENGTH}('b::len)$)
by (*simp add: uint-word-of-int word-int-case-eq-uint*)

lemma *word-int-split*:
P (*word-int-case* *f* *x*) =
 ($\forall i. x = (\text{word-of-int } i :: 'b::len \text{ word}) \wedge 0 \leq i \wedge i < 2^{\wedge} \text{LENGTH}('b) \longrightarrow P$
 (*f* *i*))
by *transfer* (*auto simp: take-bit-eq-mod*)

lemma *word-int-split-asm*:
P (*word-int-case* *f* *x*) =
 ($\nexists n. x = (\text{word-of-int } n :: 'b::len \text{ word}) \wedge 0 \leq n \wedge n < 2^{\wedge} \text{LENGTH}('b::len)$
 $\wedge \neg P (f \ n)$)
using *word-int-split* **by** *auto*

lemma *uint-range-size*: $0 \leq \text{uint } w \wedge \text{uint } w < 2^{\wedge} \text{size } w$
by *transfer simp*

lemma *sint-range-size*: $-(2^{\wedge}(\text{size } w - \text{Suc } 0)) \leq \text{sint } w \wedge \text{sint } w < 2^{\wedge}(\text{size } w - \text{Suc } 0)$
by (*simp add: word-size sint-greater-eq sint-less*)

lemma *sint-above-size*: $2^{\wedge}(\text{size } w - 1) \leq x \implies \text{sint } w < x$
for $w :: 'a::\text{len word}$
unfolding *word-size* **by** (*rule less-le-trans [OF sint-lt]*)

lemma *sint-below-size*: $x \leq - (2^{\wedge}(\text{size } w - 1)) \implies x \leq \text{sint } w$
for $w :: 'a::\text{len word}$
unfolding *word-size* **by** (*rule order-trans [OF - sint-ge]*)

lemma *word-unat-eq-iff*:
 $\langle v = w \longleftrightarrow \text{unat } v = \text{unat } w \rangle$
for $v w :: \langle 'a::\text{len word} \rangle$
by (*fact word-eq-iff-unsigned*)

107.15 Testing bits

lemma *bin-nth-uint-imp*: $\text{bit } (\text{uint } w) n \implies n < \text{LENGTH}('a)$
for $w :: 'a::\text{len word}$
by (*simp add: bit-uint-iff*)

lemma *bin-nth-sint*:
 $\text{LENGTH}('a) \leq n \implies$
 $\text{bit } (\text{sint } w) n = \text{bit } (\text{sint } w) (\text{LENGTH}('a) - 1)$
for $w :: 'a::\text{len word}$
by (*transfer fixing: n*) (*simp add: bit-signed-take-bit-iff le-diff-conv min-def*)

lemma *num-of-bintr'*:
 $\text{take-bit } (\text{LENGTH}('a::\text{len})) (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$
proof (*transfer fixing: a b*)
assume $\langle \text{take-bit } \text{LENGTH}('a) (\text{numeral } a :: \text{int}) = \text{numeral } b \rangle$
then have $\langle \text{take-bit } \text{LENGTH}('a) (\text{take-bit } \text{LENGTH}('a) (\text{numeral } a :: \text{int})) =$
 $\text{take-bit } \text{LENGTH}('a) (\text{numeral } b) \rangle$
by *simp*
then show $\langle \text{take-bit } \text{LENGTH}('a) (\text{numeral } a :: \text{int}) = \text{take-bit } \text{LENGTH}('a)$
 $(\text{numeral } b) \rangle$
by *simp*
qed

lemma *num-of-sbintr'*:
 $\text{signed-take-bit } (\text{LENGTH}('a::\text{len}) - 1) (\text{numeral } a :: \text{int}) = (\text{numeral } b) \implies$
 $\text{numeral } a = (\text{numeral } b :: 'a \text{ word})$
proof (*transfer fixing: a b*)
assume $\langle \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{numeral } a :: \text{int}) = \text{numeral } b \rangle$
then have $\langle \text{take-bit } \text{LENGTH}('a) (\text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{numeral } a$
 $:: \text{int})) = \text{take-bit } \text{LENGTH}('a) (\text{numeral } b) \rangle$
by *simp*
then show $\langle \text{take-bit } \text{LENGTH}('a) (\text{numeral } a :: \text{int}) = \text{take-bit } \text{LENGTH}('a)$
 $(\text{numeral } b) \rangle$
by (*simp add: take-bit-signed-take-bit*)

qed

lemma *num-abs-bintr*:

(numeral $x :: 'a$ word) =
 word-of-int (take-bit (LENGTH('a::len)) (numeral x))
 by transfer simp

lemma *num-abs-sbintr*:

(numeral $x :: 'a$ word) =
 word-of-int (signed-take-bit (LENGTH('a::len) - 1) (numeral x))
 by transfer (simp add: take-bit-signed-take-bit)

cast – note, no arg for new length, as it’s determined by type of result,
 thus in *cast* $w = w$, the type means cast to length of w !

lemma *bit-ucast-iff*:

$\langle \text{bit } (\text{ucast } a :: 'a::\text{len word}) \ n \longleftrightarrow n < \text{LENGTH}('a::\text{len}) \wedge \text{bit } a \ n \rangle$
 by transfer (simp add: bit-take-bit-iff)

lemma *ucast-id* [simp]: *ucast* $w = w$

by transfer simp

lemma *scast-id* [simp]: *scast* $w = w$

by transfer (simp add: take-bit-signed-take-bit)

lemma *ucast-mask-eq*:

$\langle \text{ucast } (\text{mask } n :: 'b \text{ word}) = \text{mask } (\min \text{LENGTH}('b::\text{len}) \ n) \rangle$
 by (simp add: bit-eq-iff) (auto simp: bit-mask-iff bit-ucast-iff)

— literal u(s)cast

lemma *ucast-bintr* [simp]:

ucast (numeral $w :: 'a::\text{len word}$) =
 word-of-int (take-bit (LENGTH('a)) (numeral w))
 by transfer simp

lemma *scast-sbintr* [simp]:

scast (numeral $w :: 'a::\text{len word}$) =
 word-of-int (signed-take-bit (LENGTH('a) - Suc 0) (numeral w))
 by transfer simp

lemma *source-size*: *source-size* ($c :: 'a::\text{len word} \Rightarrow -$) = LENGTH('a)

by transfer simp

lemma *target-size*: *target-size* ($c :: - \Rightarrow 'b::\text{len word}$) = LENGTH('b)

by transfer simp

lemma *is-down*: *is-down* $c \longleftrightarrow \text{LENGTH}('b) \leq \text{LENGTH}('a)$

for $c :: 'a::\text{len word} \Rightarrow 'b::\text{len word}$

by *transfer simp*

lemma *is-up*: $is-up\ c \longleftrightarrow LENGTH('a) \leq LENGTH('b)$
for $c :: 'a::len\ word \Rightarrow 'b::len\ word$
by *transfer simp*

lemma *is-up-down*:
 $\langle is-up\ c \longleftrightarrow is-down\ d \rangle$
for $c :: \langle 'a::len\ word \Rightarrow 'b::len\ word \rangle$
and $d :: \langle 'b::len\ word \Rightarrow 'a::len\ word \rangle$
by *transfer simp*

context

fixes *dummy-types* :: $\langle 'a::len \times 'b::len \rangle$

begin

private abbreviation (*input*) *UCAST* :: $\langle 'a::len\ word \Rightarrow 'b::len\ word \rangle$
where $\langle UCAST == ucast \rangle$

private abbreviation (*input*) *SCAST* :: $\langle 'a::len\ word \Rightarrow 'b::len\ word \rangle$
where $\langle SCAST == scast \rangle$

lemma *down-cast-same*:
 $\langle UCAST = scast \rangle$ **if** $\langle is-down\ UCAST \rangle$
by (*rule ext, use that in transfer*) (*simp add: take-bit-signed-take-bit*)

lemma *sint-up-scast*:
 $\langle sint\ (SCAST\ w) = sint\ w \rangle$ **if** $\langle is-up\ SCAST \rangle$
using that by transfer (*simp add: min-def Suc-leI le-diff-iff*)

lemma *uint-up-ucast*:
 $\langle uint\ (UCAST\ w) = uint\ w \rangle$ **if** $\langle is-up\ UCAST \rangle$
using that by transfer (*simp add: min-def*)

lemma *ucast-up-ucast*:
 $\langle ucast\ (UCAST\ w) = ucast\ w \rangle$ **if** $\langle is-up\ UCAST \rangle$
using that by transfer (*simp add: ac-simps*)

lemma *ucast-up-ucast-id*:
 $\langle ucast\ (UCAST\ w) = w \rangle$ **if** $\langle is-up\ UCAST \rangle$
using that by (*simp add: ucast-up-ucast*)

lemma *scast-up-scast*:
 $\langle scast\ (SCAST\ w) = scast\ w \rangle$ **if** $\langle is-up\ SCAST \rangle$
using that by transfer (*simp add: ac-simps*)

lemma *scast-up-scast-id*:
 $\langle scast\ (SCAST\ w) = w \rangle$ **if** $\langle is-up\ SCAST \rangle$
using that by (*simp add: scast-up-scast*)

lemma *isduu*:

⟨*is-up UCAST*⟩ **if** ⟨*is-down d*⟩
for $d :: \langle 'b \text{ word} \Rightarrow 'a \text{ word} \rangle$
using *that is-up-down [of UCAST d] by simp*

lemma *isdus*:

⟨*is-up SCAST*⟩ **if** ⟨*is-down d*⟩
for $d :: \langle 'b \text{ word} \Rightarrow 'a \text{ word} \rangle$
using *that is-up-down [of SCAST d] by simp*

lemmas *ucast-down-ucast-id* = *isduu* [*THEN ucast-up-ucast-id*]

lemmas *scast-down-scast-id* = *isdus* [*THEN scast-up-scast-id*]

lemma *up-ucast-surj*:

⟨*surj* (*ucast* :: $'b \text{ word} \Rightarrow 'a \text{ word}$)⟩ **if** ⟨*is-up UCAST*⟩
by (*rule surjI*) (*use that in* ⟨*rule ucast-up-ucast-id*⟩)

lemma *up-scast-surj*:

⟨*surj* (*scast* :: $'b \text{ word} \Rightarrow 'a \text{ word}$)⟩ **if** ⟨*is-up SCAST*⟩
by (*rule surjI*) (*use that in* ⟨*rule scast-up-scast-id*⟩)

lemma *down-ucast-inj*:

⟨*inj-on UCAST A*⟩ **if** ⟨*is-down* (*ucast* :: $'b \text{ word} \Rightarrow 'a \text{ word}$)⟩
by (*rule inj-on-inverseI*) (*use that in* ⟨*rule ucast-down-ucast-id*⟩)

lemma *down-scast-inj*:

⟨*inj-on SCAST A*⟩ **if** ⟨*is-down* (*scast* :: $'b \text{ word} \Rightarrow 'a \text{ word}$)⟩
by (*rule inj-on-inverseI*) (*use that in* ⟨*rule scast-down-scast-id*⟩)

lemma *ucast-down-wi*:

⟨*UCAST* (*word-of-int x*) = *word-of-int x*⟩ **if** ⟨*is-down UCAST*⟩
using *that by transfer simp*

lemma *ucast-down-no*:

⟨*UCAST* (*numeral bin*) = *numeral bin*⟩ **if** ⟨*is-down UCAST*⟩
using *that by transfer simp*

end

lemmas *word-log-defs* = *word-and-def word-or-def word-xor-def word-not-def*

lemma *bit-last-iff*:

⟨*bit w* (*LENGTH* ('a) − *Suc 0*) \longleftrightarrow *sint w* < 0⟩ (**is** ⟨ $?P \longleftrightarrow ?Q$ ⟩)
for $w :: \langle 'a::\text{len word} \rangle$
by (*simp add: bit-unsigned-iff sint-uint*)

lemma *drop-bit-eq-zero-iff-not-bit-last*:

⟨*drop-bit* (*LENGTH* ('a) − *Suc 0*) *w* = 0 \longleftrightarrow \neg *bit w* (*LENGTH* ('a) − *Suc 0*)⟩


```

for  $w :: 'a::len$  word
proof (cases  $\langle LENGTH('a) \rangle$ )
  case (Suc  $n$ )
  then show ?thesis
    apply transfer
    apply (simp add: take-bit-drop-bit)
    by (simp add: bit-iff-odd-drop-bit drop-bit-take-bit odd-iff-mod-2-eq-one)
qed auto

```

```

lemma unat-div:
   $\langle unat (x \div y) = unat x \div unat y \rangle$ 
by (fact unat-div-distrib)

```

```

lemma unat-mod:
   $\langle unat (x \bmod y) = unat x \bmod unat y \rangle$ 
by (fact unat-mod-distrib)

```

107.16 Word Arithmetic

```

lemmas less-eq-word-numeral-numeral [simp] =
  word-le-def [of  $\langle numeral a \rangle \langle numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg
  unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for  $a b$ 
lemmas less-word-numeral-numeral [simp] =
  word-less-def [of  $\langle numeral a \rangle \langle numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg
  unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for  $a b$ 
lemmas less-eq-word-minus-numeral-numeral [simp] =
  word-le-def [of  $\langle - numeral a \rangle \langle numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg
  unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for  $a b$ 
lemmas less-word-minus-numeral-numeral [simp] =
  word-less-def [of  $\langle - numeral a \rangle \langle numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg
  unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for  $a b$ 
lemmas less-eq-word-numeral-minus-numeral [simp] =
  word-le-def [of  $\langle numeral a \rangle \langle - numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg
  unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for  $a b$ 
lemmas less-word-numeral-minus-numeral [simp] =
  word-less-def [of  $\langle numeral a \rangle \langle - numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg
  unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for  $a b$ 
lemmas less-eq-word-minus-numeral-minus-numeral [simp] =
  word-le-def [of  $\langle - numeral a \rangle \langle - numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg
  unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for  $a b$ 
lemmas less-word-minus-numeral-minus-numeral [simp] =
  word-less-def [of  $\langle - numeral a \rangle \langle - numeral b \rangle$ , simplified uint-bintrunc uint-bintrunc-neg

```

```

unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-numeral-minus-1 [simp] =
  word-less-def [of ⟨numeral a⟩ ⟨− 1⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas less-word-minus-numeral-minus-1 [simp] =
  word-less-def [of ⟨− numeral a⟩ ⟨− 1⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b

lemmas sless-eq-word-numeral-numeral [simp] =
  word-sle-eq [of ⟨numeral a⟩ ⟨numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-word-numeral-numeral [simp] =
  word-sless-alt [of ⟨numeral a⟩ ⟨numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-eq-word-minus-numeral-numeral [simp] =
  word-sle-eq [of ⟨− numeral a⟩ ⟨numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-word-minus-numeral-numeral [simp] =
  word-sless-alt [of ⟨− numeral a⟩ ⟨numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-eq-word-numeral-minus-numeral [simp] =
  word-sle-eq [of ⟨numeral a⟩ ⟨− numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-word-numeral-minus-numeral [simp] =
  word-sless-alt [of ⟨numeral a⟩ ⟨− numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-eq-word-minus-numeral-minus-numeral [simp] =
  word-sle-eq [of ⟨− numeral a⟩ ⟨− numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b
lemmas sless-word-minus-numeral-minus-numeral [simp] =
  word-sless-alt [of ⟨− numeral a⟩ ⟨− numeral b⟩, simplified sint-sbintrunc sint-sbintrunc-neg]
  for a b

lemmas div-word-numeral-numeral [simp] =
  word-div-def [of ⟨numeral a⟩ ⟨numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas div-word-minus-numeral-numeral [simp] =
  word-div-def [of ⟨− numeral a⟩ ⟨numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas div-word-numeral-minus-numeral [simp] =
  word-div-def [of ⟨numeral a⟩ ⟨− numeral b⟩, simplified uint-bintrunc uint-bintrunc-neg
unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
  for a b
lemmas div-word-minus-numeral-minus-numeral [simp] =

```

word-div-def [of $\langle - \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *div-word-minus-1-numeral* [simp] =

word-div-def [of $\langle -\ 1 \rangle \langle \text{numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *div-word-minus-1-minus-numeral* [simp] =

word-div-def [of $\langle -\ 1 \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *mod-word-numeral-numeral* [simp] =

word-mod-def [of $\langle \text{numeral } a \rangle \langle \text{numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *mod-word-minus-numeral-numeral* [simp] =

word-mod-def [of $\langle - \text{ numeral } a \rangle \langle \text{numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *mod-word-numeral-minus-numeral* [simp] =

word-mod-def [of $\langle \text{numeral } a \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *mod-word-minus-numeral-minus-numeral* [simp] =

word-mod-def [of $\langle - \text{ numeral } a \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *mod-word-minus-1-numeral* [simp] =

word-mod-def [of $\langle -\ 1 \rangle \langle \text{numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemmas *mod-word-minus-1-minus-numeral* [simp] =

word-mod-def [of $\langle -\ 1 \rangle \langle - \text{ numeral } b \rangle$, simplified *uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]

for $a\ b$

lemma *signed-drop-bit-of-1* [simp]:

$\langle \text{signed-drop-bit } n\ (1 :: 'a::\text{len word}) \rangle = \text{of_bool } (\text{LENGTH}('a) = 1 \vee n = 0)$

apply (*transfer fixing: n*)

apply (*cases* $\langle \text{LENGTH}('a) \rangle$)

apply (*auto simp: take-bit-signed-take-bit*)

apply (*auto simp: take-bit-drop-bit gr0-conv-Suc simp flip: take-bit-eq-self-iff-drop-bit-eq-0*)

done

lemma *take-bit-word-beyond-length-eq*:

$\langle \text{take-bit } n\ w = w \rangle$ **if** $\langle \text{LENGTH}('a) \leq n \rangle$ **for** $w :: \langle 'a::\text{len word} \rangle$

by (*simp add: take-bit-word-eq-self that*)

lemmas *word-div-no* [simp] = *word-div-def* [of numeral *a* numeral *b*] **for** *a b*
lemmas *word-mod-no* [simp] = *word-mod-def* [of numeral *a* numeral *b*] **for** *a b*
lemmas *word-less-no* [simp] = *word-less-def* [of numeral *a* numeral *b*] **for** *a b*
lemmas *word-le-no* [simp] = *word-le-def* [of numeral *a* numeral *b*] **for** *a b*
lemmas *word-sless-no* [simp] = *word-sless-eq* [of numeral *a* numeral *b*] **for** *a b*
lemmas *word-sle-no* [simp] = *word-sle-eq* [of numeral *a* numeral *b*] **for** *a b*

lemma *size-0-same'*: *size w = 0 \implies w = v*
for *v w :: 'a::len word*
by (*unfold word-size*) *simp*

lemmas *size-0-same* = *size-0-same'* [*unfolded word-size*]

lemmas *unat-eq-0* = *unat-0-iff*
lemmas *unat-eq-zero* = *unat-0-iff*

lemma *mask-1*: *mask 1 = 1*
by *simp*

lemma *mask-Suc-0*: *mask (Suc 0) = 1*
by *simp*

lemma *bin-last-bintrunc*: *odd (take-bit l n) \longleftrightarrow l > 0 \wedge odd n*
by *simp*

lemma *push-bit-word-beyond* [simp]:
 $\langle \text{push-bit } n \ w = 0 \rangle$ **if** $\langle \text{LENGTH('a)} \leq n \rangle$ **for** *w :: 'a::len word*
using that by (*transfer fixing: n*) (*simp add: take-bit-push-bit*)

lemma *drop-bit-word-beyond* [simp]:
 $\langle \text{drop-bit } n \ w = 0 \rangle$ **if** $\langle \text{LENGTH('a)} \leq n \rangle$ **for** *w :: 'a::len word*
using that by (*transfer fixing: n*) (*simp add: drop-bit-take-bit*)

lemma *signed-drop-bit-beyond*:
 $\langle \text{signed-drop-bit } n \ w = (\text{if bit } w \ (\text{LENGTH('a)} - \text{Suc } 0) \text{ then } -1 \text{ else } 0) \rangle$
if $\langle \text{LENGTH('a)} \leq n \rangle$ **for** *w :: 'a::len word*
by (*rule bit-word-eqI*) (*simp add: bit-signed-drop-bit-iff that*)

lemma *take-bit-numeral-minus-numeral-word* [simp]:
 $\langle \text{take-bit (numeral } m) \ (- \text{ numeral } n :: 'a::len \text{ word}) =$
 $(\text{case take-bit-num (numeral } m) \ n \text{ of None } \Rightarrow 0 \mid \text{Some } q \Rightarrow \text{take-bit (numeral } m) \ (2 \wedge \text{numeral } m - \text{numeral } q)) \rangle$ **(is** $\langle ?lhs = ?rhs \rangle$)
proof (*cases* $\langle \text{LENGTH('a)} \leq \text{numeral } m \rangle$)
case *True*
then have *: $\langle (\text{take-bit (numeral } m) :: 'a \text{ word} \Rightarrow 'a \text{ word}) = \text{id} \rangle$
by (*simp add: fun-eq-iff take-bit-word-eq-self*)
have **: $\langle 2 \wedge \text{numeral } m = (0 :: 'a \text{ word}) \rangle$
using *True* **by** (*simp flip: exp-eq-zero-iff*)
show *?thesis*

```

    by (auto simp only: * ** split: option.split
      dest!: take-bit-num-eq-None-imp [where ?'a = ⟨'a word⟩] take-bit-num-eq-Some-imp
      [where ?'a = ⟨'a word⟩])
      simp-all
next
case False
then show ?thesis
  by (transfer fixing: m n) simp
qed

```

```

lemma of-nat-inverse:
  ⟨word-of-nat r = a ⟹ r < 2 ^ LENGTH('a) ⟹ unat a = r⟩
  for a :: ⟨'a::len word⟩
  by (metis id-apply of-nat-eq-id take-bit-nat-eq-self-iff unsigned-of-nat)

```

107.17 Transferring goals from words to ints

```

lemma word-ths:
  shows word-succ-p1: word-succ a = a + 1
    and word-pred-m1: word-pred a = a - 1
    and word-pred-succ: word-pred (word-succ a) = a
    and word-succ-pred: word-succ (word-pred a) = a
    and word-mult-succ: word-succ a * b = b + a * b
  by (transfer, simp add: algebra-simps)+

lemma uint-cong: x = y ⟹ uint x = uint y
  by simp

lemma uint-word-ariths:
  fixes a b :: 'a::len word
  shows uint (a + b) = (uint a + uint b) mod 2 ^ LENGTH('a::len)
    and uint (a - b) = (uint a - uint b) mod 2 ^ LENGTH('a)
    and uint (a * b) = uint a * uint b mod 2 ^ LENGTH('a)
    and uint (- a) = - uint a mod 2 ^ LENGTH('a)
    and uint (word-succ a) = (uint a + 1) mod 2 ^ LENGTH('a)
    and uint (word-pred a) = (uint a - 1) mod 2 ^ LENGTH('a)
    and uint (0 :: 'a word) = 0 mod 2 ^ LENGTH('a)
    and uint (1 :: 'a word) = 1 mod 2 ^ LENGTH('a)
  by (simp-all only: word-arith-wis uint-word-of-int-eq flip: take-bit-eq-mod)

```

```

lemma uint-word-arith-bintrs:
  fixes a b :: 'a::len word
  shows uint (a + b) = take-bit (LENGTH('a)) (uint a + uint b)
    and uint (a - b) = take-bit (LENGTH('a)) (uint a - uint b)
    and uint (a * b) = take-bit (LENGTH('a)) (uint a * uint b)
    and uint (- a) = take-bit (LENGTH('a)) (- uint a)
    and uint (word-succ a) = take-bit (LENGTH('a)) (uint a + 1)
    and uint (word-pred a) = take-bit (LENGTH('a)) (uint a - 1)
    and uint (0 :: 'a word) = take-bit (LENGTH('a)) 0

```

and $\text{uint } (1 :: 'a \text{ word}) = \text{take-bit } (\text{LENGTH}('a)) \ 1$
by (*simp-all add: uint-word-ariths take-bit-eq-mod*)

context
fixes $a \ b :: 'a::\text{len word}$
begin

lemma *sint-word-add*: $\text{sint } (a + b) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a + \text{sint } b)$
by *transfer (simp add: signed-take-bit-add)*

lemma *sint-word-diff*: $\text{sint } (a - b) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a - \text{sint } b)$
by *transfer (simp add: signed-take-bit-diff)*

lemma *sint-word-mult*: $\text{sint } (a * b) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a * \text{sint } b)$
by *transfer (simp add: signed-take-bit-mult)*

lemma *sint-word-minus*: $\text{sint } (- a) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (- \text{sint } a)$
by *transfer (simp add: signed-take-bit-minus)*

lemma *sint-word-succ*: $\text{sint } (\text{word-succ } a) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a + 1)$
by (*metis of-int-sint scast-id sint-sbintrunc' wi-hom-succ*)

lemma *sint-word-pred*: $\text{sint } (\text{word-pred } a) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) (\text{sint } a - 1)$
by (*metis of-int-sint scast-id sint-sbintrunc' wi-hom-pred*)

lemma *sint-word-01*:
 $\text{sint } (0 :: 'a \text{ word}) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) \ 0$
 $\text{sint } (1 :: 'a \text{ word}) = \text{signed-take-bit } (\text{LENGTH}('a) - 1) \ 1$
by (*simp-all add: sint-uint*)

end

lemmas *sint-word-ariths* =
sint-word-add sint-word-diff sint-word-mult sint-word-minus
sint-word-succ sint-word-pred sint-word-01

lemma *word-pred-0-n1*: $\text{word-pred } 0 = \text{word-of-int } (- 1)$
unfolding *word-pred-m1* **by** *simp*

lemma *succ-pred-no* [*simp*]:
 $\text{word-succ } (\text{numeral } w) = \text{numeral } w + 1$
 $\text{word-pred } (\text{numeral } w) = \text{numeral } w - 1$

$\text{word-succ } (- \text{ numeral } w) = - \text{ numeral } w + 1$
 $\text{word-pred } (- \text{ numeral } w) = - \text{ numeral } w - 1$
by (*simp-all add: word-succ-p1 word-pred-m1*)

lemma *word-sp-01* [*simp*]:
 $\text{word-succ } (- 1) = 0 \wedge \text{word-succ } 0 = 1 \wedge \text{word-pred } 0 = - 1 \wedge \text{word-pred } 1 = 0$
by (*simp-all add: word-succ-p1 word-pred-m1*)

— alternative approach to lifting arithmetic equalities

lemma *word-of-int-Ex*: $\exists y. x = \text{word-of-int } y$
by (*rule-tac x=uint x in exI simp*)

107.18 Order on fixed-length words

lift-definition *udvd* :: $\langle 'a::\text{len word} \Rightarrow 'a::\text{len word} \Rightarrow \text{bool} \rangle$ (**infixl** $\langle \text{udvd} \rangle$ 50)
is $\langle \lambda k l. \text{take-bit LENGTH('a)} k \text{ dvd take-bit LENGTH('a)} l \rangle$ **by** *simp*

lemma *udvd-iff-dvd*:
 $\langle x \text{ udvd } y \longleftrightarrow \text{unat } x \text{ dvd unat } y \rangle$
by *transfer (simp add: nat-dvd-iff)*

lemma *udvd-iff-dvd-int*:
 $\langle v \text{ udvd } w \longleftrightarrow \text{uint } v \text{ dvd uint } w \rangle$
by *transfer rule*

lemma *udvdI* [*intro*]:
 $\langle v \text{ udvd } w \rangle$ **if** $\langle \text{unat } w = \text{unat } v * \text{unat } u \rangle$
proof —
from *that* **have** $\langle \text{unat } v \text{ dvd unat } w \rangle$..
then show *?thesis*
by (*simp add: udvd-iff-dvd*)
qed

lemma *udvdE* [*elim*]:
fixes $v w :: \langle 'a::\text{len word} \rangle$
assumes $\langle v \text{ udvd } w \rangle$
obtains $u :: \langle 'a \text{ word} \rangle$ **where** $\langle \text{unat } w = \text{unat } v * \text{unat } u \rangle$
proof (*cases* $\langle v = 0 \rangle$)
case *True*
moreover from *True* $\langle v \text{ udvd } w \rangle$ **have** $\langle w = 0 \rangle$
by *transfer simp*
ultimately show *thesis*
using *that* **by** *simp*
next
case *False*
then have $\langle \text{unat } v > 0 \rangle$
by (*simp add: unat-gt-0*)
from $\langle v \text{ udvd } w \rangle$ **have** $\langle \text{unat } v \text{ dvd unat } w \rangle$

```

  by (simp add: udvd-iff-dvd)
then obtain n where ⟨unat w = unat v * n⟩ ..
moreover have ⟨n < 2 ^ LENGTH('a)⟩
proof (rule ccontr)
  assume ⟨¬ n < 2 ^ LENGTH('a)⟩
  then have ⟨n ≥ 2 ^ LENGTH('a)⟩
  by (simp add: not-le)
  then have ⟨unat v * n ≥ 2 ^ LENGTH('a)⟩
  using ⟨unat v > 0⟩ mult-le-mono [of 1 ⟨unat v⟩ ⟨2 ^ LENGTH('a)⟩ n]
  by simp
  with ⟨unat w = unat v * n⟩
  have ⟨unat w ≥ 2 ^ LENGTH('a)⟩
  by simp
  with unsigned-less [of w, where ?'a = nat] show False
  by linarith
qed
ultimately have ⟨unat w = unat v * unat (word-of-nat n :: 'a word)⟩
  by (auto simp: take-bit-nat-eq-self-iff unsigned-of-nat intro: sym)
with that show thesis .
qed

```

```

lemma udvd-imp-mod-eq-0:
  ⟨w mod v = 0⟩ if ⟨v udvd w⟩
  using that by transfer simp

```

```

lemma mod-eq-0-imp-udvd [intro?]:
  ⟨v udvd w⟩ if ⟨w mod v = 0⟩
  by (metis mod-0-imp-dvd that udvd-iff-dvd unat-0 unat-mod-distrib)

```

```

lemma udvd-imp-dvd:
  ⟨v dvd w⟩ if ⟨v udvd w⟩ for v w :: ⟨'a::len word⟩
proof -
  from that obtain u :: ⟨'a word⟩ where ⟨unat w = unat v * unat u⟩ ..
  then have ⟨w = v * u⟩
  by (metis of-nat-mult of-nat-unat word-mult-def word-of-int-uint)
  then show ⟨v dvd w⟩ ..
qed

```

```

lemma exp-dvd-iff-exp-udvd:
  ⟨2 ^ n dvd w ⟷ 2 ^ n udvd w⟩ for v w :: ⟨'a::len word⟩
proof
  assume ⟨2 ^ n udvd w⟩ then show ⟨2 ^ n dvd w⟩
  by (rule udvd-imp-dvd)
next
  assume ⟨2 ^ n dvd w⟩
  then obtain u :: ⟨'a word⟩ where ⟨w = 2 ^ n * u⟩ ..
  then have ⟨w = push-bit n u⟩
  by (simp add: push-bit-eq-mult)
  then show ⟨2 ^ n udvd w⟩

```


by transfer (simp add: take-bit-push-bit dvd-eq-mod-eq-0 flip: take-bit-eq-mod)
qed

lemma *udvd-nat-alt*:

$\langle a \text{ udvd } b \longleftrightarrow (\exists n. \text{unat } b = n * \text{unat } a) \rangle$
by (auto simp: udvd-iff-dvd)

lemma *udvd-unfold-int*:

$\langle a \text{ udvd } b \longleftrightarrow (\exists n \geq 0. \text{uint } b = n * \text{uint } a) \rangle$
unfolding *udvd-iff-dvd-int*
by (metis dvd-div-mult-self dvd-triv-right uint-div-distrib uint-ge-0)

lemma *unat-minus-one*:

$\langle \text{unat } (w - 1) = \text{unat } w - 1 \rangle$ if $\langle w \neq 0 \rangle$
proof –
have $0 \leq \text{uint } w$ by (fact uint-nonnegative)
moreover from that have $0 \neq \text{uint } w$
by (simp add: uint-0-iff)
ultimately have $1 \leq \text{uint } w$
by arith
from *uint-lt2p* [of *w*] have $\text{uint } w - 1 < 2^{\text{LENGTH}('a)}$
by arith
with $\langle 1 \leq \text{uint } w \rangle$ have $(\text{uint } w - 1) \bmod 2^{\text{LENGTH}('a)} = \text{uint } w - 1$
by (auto intro: mod-pos-pos-trivial)
with $\langle 1 \leq \text{uint } w \rangle$ have $\text{nat } ((\text{uint } w - 1) \bmod 2^{\text{LENGTH}('a)}) = \text{nat } (\text{uint } w) - 1$
by (auto simp del: nat-uint-eq)
then show ?thesis
by (metis uint-word-ariths(6) unat-eq-nat-uint word-pred-m1)
qed

lemma *measure-unat*: $p \neq 0 \implies \text{unat } (p - 1) < \text{unat } p$

by (simp add: unat-minus-one) (simp add: unat-0-iff [symmetric])

lemmas *uint-add-ge0* [simp] = *add-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

lemmas *uint-mult-ge0* [simp] = *mult-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

lemma *uint-sub-lt2p* [simp]: $\text{uint } x - \text{uint } y < 2^{\text{LENGTH}('a)}$

for $x :: 'a::\text{len word}$ and $y :: 'b::\text{len word}$
using *uint-ge-0* [of *y*] *uint-lt2p* [of *x*] by arith

107.19 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:

$(\text{uint } x + \text{uint } y < 2^{\text{LENGTH}('a)}) =$
 $(\text{uint } (x + y) = \text{uint } x + \text{uint } y)$
for $x \ y :: 'a::\text{len word}$
by (metis add.right-neutral add-mono-thms-linordered-semiring(1) mod-pos-pos-trivial)

of-nat-0-le-iff uint-lt2p uint-nat uint-word-ariths(1))

lemma *uint-mult-lem:*

$(\text{uint } x * \text{uint } y < 2 \wedge \text{LENGTH}('a)) =$
 $(\text{uint } (x * y) = \text{uint } x * \text{uint } y)$
for $x \ y :: 'a::\text{len word}$
by (*metis mod-pos-pos-trivial uint-lt2p uint-mult-ge0 uint-word-ariths(3)*)

lemma *uint-sub-lem:* $\text{uint } x \geq \text{uint } y \longleftrightarrow \text{uint } (x - y) = \text{uint } x - \text{uint } y$

by (*simp add: uint-word-arith-bintrs take-bit-int-eq-self-iff*)

lemma *uint-add-le:* $\text{uint } (x + y) \leq \text{uint } x + \text{uint } y$

unfolding *uint-word-ariths* **by** (*simp add: zmod-le-nonneg-dividend*)

lemma *uint-sub-ge:* $\text{uint } (x - y) \geq \text{uint } x - \text{uint } y$

by (*smt (verit, ccfv-SIG) uint-nonnegative uint-sub-lem*)

lemma *int-mod-ge:* $\langle a \leq a \bmod n \rangle$ **if** $\langle a < n \rangle \langle 0 < n \rangle$

for $a \ n :: \text{int}$

using *that order.trans* [*of a 0 < a mod n*] **by** (*cases <a < 0>*) *auto*

lemma *mod-add-if-z:*

$\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$
 $(x + y) \bmod z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$
for $x \ y \ z :: \text{int}$
by (*smt (verit, best) minus-mod-self2 mod-pos-pos-trivial*)

lemma *uint-plus-if':*

$\text{uint } (a + b) =$
 $(\text{if } \text{uint } a + \text{uint } b < 2 \wedge \text{LENGTH}('a) \text{ then } \text{uint } a + \text{uint } b$
 $\text{else } \text{uint } a + \text{uint } b - 2 \wedge \text{LENGTH}('a))$
for $a \ b :: 'a::\text{len word}$
using *mod-add-if-z* [*of uint a - uint b*] **by** (*simp add: uint-word-ariths*)

lemma *mod-sub-if-z:*

$\llbracket x < z; y < z; 0 \leq y; 0 \leq x; 0 \leq z \rrbracket \implies$
 $(x - y) \bmod z = (\text{if } y \leq x \text{ then } x - y \text{ else } x - y + z)$
for $x \ y \ z :: \text{int}$
using *mod-pos-pos-trivial* [*of x - y + z z*] **by** (*auto simp: not-le*)

lemma *uint-sub-if':*

$\text{uint } (a - b) =$
 $(\text{if } \text{uint } b \leq \text{uint } a \text{ then } \text{uint } a - \text{uint } b$
 $\text{else } \text{uint } a - \text{uint } b + 2 \wedge \text{LENGTH}('a))$
for $a \ b :: 'a::\text{len word}$
using *mod-sub-if-z* [*of uint a - uint b*] **by** (*simp add: uint-word-ariths*)

lemma *word-of-int-inverse:*

$\text{word-of-int } r = a \implies 0 \leq r \implies r < 2 \wedge \text{LENGTH}('a) \implies \text{uint } a = r$

```

for  $a :: 'a::len$  word
by transfer (simp add: take-bit-int-eq-self)

lemma unat-split:  $P \text{ (unat } x) \longleftrightarrow (\forall n. \text{ of-nat } n = x \wedge n < 2^{\text{LENGTH}('a)} \longrightarrow P\ n)$ 
for  $x :: 'a::len$  word
by (auto simp: unsigned-of-nat take-bit-nat-eq-self)

lemma unat-split-asm:  $P \text{ (unat } x) \longleftrightarrow (\nexists n. \text{ of-nat } n = x \wedge n < 2^{\text{LENGTH}('a)} \wedge \neg P\ n)$ 
for  $x :: 'a::len$  word
using unat-split by auto

lemma un-ui-le:
 $\langle \text{unat } a \leq \text{unat } b \longleftrightarrow \text{uint } a \leq \text{uint } b \rangle$ 
by transfer (simp add: nat-le-iff)

lemma unat-plus-if':
 $\langle \text{unat } (a + b) =$ 
  (if  $\text{unat } a + \text{unat } b < 2^{\text{LENGTH}('a)}$ 
    then  $\text{unat } a + \text{unat } b$ 
    else  $\text{unat } a + \text{unat } b - 2^{\text{LENGTH}('a)}$ )  $\rangle$  for  $a\ b :: 'a::len$  word
apply (auto simp: not-less le-iff-add)
using of-nat-inverse apply force
by (smt (verit, ccfv-SIG) numeral-Bit0 numerals(1) of-nat-0-le-iff of-nat-1 of-nat-add
  of-nat-eq-iff of-nat-power of-nat-unat uint-plus-if')

lemma unat-sub-if-size:
 $\text{unat } (x - y) =$ 
  (if  $\text{unat } y \leq \text{unat } x$ 
    then  $\text{unat } x - \text{unat } y$ 
    else  $\text{unat } x + 2^{\text{size } x} - \text{unat } y$ )
proof -
  { assume  $xy: \neg \text{uint } y \leq \text{uint } x$ 
    have  $\text{nat } (\text{uint } x - \text{uint } y + 2^{\text{LENGTH}('a)}) = \text{nat } (\text{uint } x + 2^{\text{LENGTH}('a)} - \text{uint } y)$ 
    by simp
    also have  $\dots = \text{nat } (\text{uint } x + 2^{\text{LENGTH}('a)}) - \text{nat } (\text{uint } y)$ 
    by (simp add: nat-diff-distrib')
    also have  $\dots = \text{nat } (\text{uint } x) + 2^{\text{LENGTH}('a)} - \text{nat } (\text{uint } y)$ 
    by (simp add: nat-add-distrib nat-power-eq)
    finally have  $\text{nat } (\text{uint } x - \text{uint } y + 2^{\text{LENGTH}('a)}) = \text{nat } (\text{uint } x) + 2^{\text{LENGTH}('a)} - \text{nat } (\text{uint } y)$ 
  }
then show ?thesis
by (metis nat-diff-distrib' uint-range-size uint-sub-if' un-ui-le unat-eq-nat-uint
  word-size)
qed

```

lemmas *unat-sub-if' = unat-sub-if-size [unfolded word-size]*

lemma *uint-split:*

$P \text{ (uint } x) = (\forall i. \text{word-of-int } i = x \wedge 0 \leq i \wedge i < 2^{\text{LENGTH('a)}} \longrightarrow P \ i)$
for $x :: 'a::\text{len word}$
by *transfer (auto simp: take-bit-eq-mod)*

lemma *uint-split-asm:*

$P \text{ (uint } x) = (\nexists i. \text{word-of-int } i = x \wedge 0 \leq i \wedge i < 2^{\text{LENGTH('a)}} \wedge \neg P \ i)$
for $x :: 'a::\text{len word}$
by *(auto simp: unsigned-of-int take-bit-int-eq-self)*

107.20 Some proof tool support

lemma *power-False-cong: False $\implies a \wedge b = c \wedge d$*

by *auto*

lemmas *unat-splits = unat-split unat-split-asm*

lemmas *unat-arith-simps =*

word-le-nat-alt word-less-nat-alt
word-unat-eq-iff
unat-sub-if' unat-plus-if' unat-div unat-mod

lemmas *uint-splits = uint-split uint-split-asm*

lemmas *uint-arith-simps =*

word-le-def word-less-alt
word-uint-eq-iff
uint-sub-if' uint-plus-if'

— *unat-arith-tac*: tactic to reduce word arithmetic to *nat*, try to solve via *arith*

ML \langle

val unat-arith-simpset =

@{context} (TODO: completely explicitly determined simpset *)*
 $|> \text{fold Simplifier.add-simp } @\{\text{thms unat-arith-simps}\}$
 $|> \text{fold Splitter.add-split } @\{\text{thms if-split-asm}\}$
 $|> \text{fold Simplifier.add-cong } @\{\text{thms power-False-cong}\}$
 $|> \text{simpset-of}$

fun unat-arith-tacs ctxt =

let

fun arith-tac' n t =
Arith-Data.arith-tac ctxt n t
handle Cooper.COOPER - => Seq.empty;

in

$[\text{clarify-tac ctxt } 1,$
 $\text{full-simp-tac (put-simpset unat-arith-simpset ctxt) } 1,$
 $\text{ALLGOALS (full-simp-tac$

```

      (put-simpset HOL-ss ctxt
        |> fold Splitter.add-split @{thms unat-splits}
        |> fold Simplifier.add-cong @{thms power-False-cong})),
      rewrite-goals-tac ctxt @{thms word-size},
      ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
        REPEAT (eresolve-tac ctxt [conjE] n) THEN
        REPEAT (dresolve-tac ctxt @{thms of-nat-inverse} n) THEN
      assume-tac ctxt n)),
      TRYALL arith-tac' ]
    end

```

```

fun unat-arith-tac ctxt = SELECT-GOAL (EVERY (unat-arith-tacs ctxt))
>

```

method-setup *unat-arith* =
 ‹Scan.succeed (SIMPLE-METHOD' o unat-arith-tac)›
 solving word arithmetic via natural numbers and arith

— *uint-arith-tac*: reduce to arithmetic on int, try to solve by arith

ML ‹

```

val uint-arith-simpset =
  @{context} (* TODO: completely explicitly determined simpset *)
  |> fold Simplifier.add-simp @{thms uint-arith-simps}
  |> fold Splitter.add-split @{thms if-split-asm}
  |> fold Simplifier.add-cong @{thms power-False-cong}
  |> simpset-of;

```

```

fun uint-arith-tacs ctxt =
  let
    fun arith-tac' n t =
      Arith-Data.arith-tac ctxt n t
      handle Cooper.COOPER - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (put-simpset uint-arith-simpset ctxt) 1,
      ALLGOALS (full-simp-tac
        (put-simpset HOL-ss ctxt
          |> fold Splitter.add-split @{thms uint-splits}
          |> fold Simplifier.add-cong @{thms power-False-cong})),
      rewrite-goals-tac ctxt @{thms word-size},
      ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
        REPEAT (eresolve-tac ctxt [conjE] n) THEN
        REPEAT (dresolve-tac ctxt @{thms word-of-int-inverse} n
          THEN assume-tac ctxt n
          THEN assume-tac ctxt n)),
      TRYALL arith-tac' ]
  end

```

```

fun uint-arith-tac ctxt = SELECT-GOAL (EVERY (uint-arith-tacs ctxt))

```

›

method-setup *uint-arith* =
 ‹*Scan.succeed (SIMPLE-METHOD' o uint-arith-tac)*›
solving word arithmetic via integers and arith

107.21 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*: $x \leq x + y \longleftrightarrow \text{uint } x + \text{uint } y < 2^{\text{size } x}$
 for $x \ y :: 'a::\text{len word}$
 by (*auto simp: word-size word-le-def uint-add-lem uint-sub-lem*)

lemmas *no-olen-add* = *no-plus-overflow-uint-size* [*unfolded word-size*]

lemma *no-ulen-sub*: $x \geq x - y \longleftrightarrow \text{uint } y \leq \text{uint } x$
 for $x \ y :: 'a::\text{len word}$
 by (*auto simp: word-size word-le-def uint-add-lem uint-sub-lem*)

lemma *no-olen-add'*: $x \leq y + x \longleftrightarrow \text{uint } y + \text{uint } x < 2^{\text{LENGTH}('a)}$
 for $x \ y :: 'a::\text{len word}$
 by (*simp add: ac-simps no-olen-add*)

lemmas *olen-add-equiv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*]]

lemmas *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]
lemmas *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]
lemmas *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*]
lemmas *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]
lemmas *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]
lemmas *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

lemma *word-less-sub1*: $x \neq 0 \implies 1 < x \longleftrightarrow 0 < x - 1$
 for $x :: 'a::\text{len word}$
 by (*transfer (simp add: take-bit-decr-eq)*)

lemma *word-le-sub1*: $x \neq 0 \implies 1 \leq x \longleftrightarrow 0 \leq x - 1$
 for $x :: 'a::\text{len word}$
 by (*transfer (simp add: int-one-le-iff-zero-less less-le)*)

lemma *sub-wrap-lt*: $x < x - z \longleftrightarrow x < z$
 for $x \ z :: 'a::\text{len word}$
 by (*meson linorder-not-le word-sub-le-iff*)

lemma *sub-wrap*: $x \leq x - z \longleftrightarrow z = 0 \vee x < z$
 for $x \ z :: 'a::\text{len word}$
 by (*simp add: le-less sub-wrap-lt ac-simps*)

lemma *plus-minus-not-NULL-ab*: $x \leq ab - c \implies c \leq ab \implies c \neq 0 \implies x + c \neq 0$

for $x\ ab\ c :: 'a::len\ word$
by *uint-arith*

lemma *plus-minus-no-overflow-ab*: $x \leq ab - c \implies c \leq ab \implies x \leq x + c$
for $x\ ab\ c :: 'a::len\ word$
by *uint-arith*

lemma *le-minus'*: $a + c \leq b \implies a \leq a + c \implies c \leq b - a$
for $a\ b\ c :: 'a::len\ word$
by *uint-arith*

lemma *le-plus'*: $a \leq b \implies c \leq b - a \implies a + c \leq b$
for $a\ b\ c :: 'a::len\ word$
by *uint-arith*

lemmas *le-plus = le-plus'* [*rotated*]

lemmas *le-minus = leD* [*THEN thin-rl, THEN le-minus'*]

lemma *word-plus-mono-right*: $y \leq z \implies x \leq x + z \implies x + y \leq x + z$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-less-minus-cancel*: $y - x < z - x \implies x \leq z \implies y < z$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-less-minus-mono-left*: $y < z \implies x \leq y \implies y - x < z - x$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-less-minus-mono*: $a < c \implies d < b \implies a - b < a \implies c - d < c$
 $\implies a - b < c - d$
for $a\ b\ c\ d :: 'a::len\ word$
by *uint-arith*

lemma *word-le-minus-cancel*: $y - x \leq z - x \implies x \leq z \implies y \leq z$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-le-minus-mono-left*: $y \leq z \implies x \leq y \implies y - x \leq z - x$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-le-minus-mono*:
 $a \leq c \implies d \leq b \implies a - b \leq a \implies c - d \leq c \implies a - b \leq c - d$
for $a\ b\ c\ d :: 'a::len\ word$
by *uint-arith*

lemma *plus-le-left-cancel-wrap*: $x + y' < x \implies x + y < x \implies x + y' < x + y$
 $\longleftrightarrow y' < y$
for $x\ y\ y' :: 'a::\text{len word}$
by *uint-arith*

lemma *plus-le-left-cancel-nowrap*: $x \leq x + y' \implies x \leq x + y \implies x + y' < x + y$
 $\longleftrightarrow y' < y$
for $x\ y\ y' :: 'a::\text{len word}$
by *uint-arith*

lemma *word-plus-mono-right2*: $a \leq a + b \implies c \leq b \implies a \leq a + c$
for $a\ b\ c :: 'a::\text{len word}$
by *uint-arith*

lemma *word-less-add-right*: $x < y - z \implies z \leq y \implies x + z < y$
for $x\ y\ z :: 'a::\text{len word}$
by *uint-arith*

lemma *word-less-sub-right*: $x < y + z \implies y \leq x \implies x - y < z$
for $x\ y\ z :: 'a::\text{len word}$
by *uint-arith*

lemma *word-le-plus-either*: $x \leq y \vee x \leq z \implies y \leq y + z \implies x \leq y + z$
for $x\ y\ z :: 'a::\text{len word}$
by *uint-arith*

lemma *word-less-nowrapI*: $x < z - k \implies k \leq z \implies 0 < k \implies x < x + k$
for $x\ z\ k :: 'a::\text{len word}$
by *uint-arith*

lemma *inc-le*: $i < m \implies i + 1 \leq m$
for $i\ m :: 'a::\text{len word}$
by *uint-arith*

lemma *less-imp-less-eq-dec*:
 $\langle v \leq w - 1 \rangle$ **if** $\langle v < w \rangle$ **for** $v\ w :: 'a::\text{len word}$
using that *proof transfer*
show $\langle \text{take-bit LENGTH('a)}\ k \leq \text{take-bit LENGTH('a)}\ (l - 1) \rangle$
if $\langle \text{take-bit LENGTH('a)}\ k < \text{take-bit LENGTH('a)}\ l \rangle$
for $k\ l :: \text{int}$
using that **by** (cases $\langle \text{take-bit LENGTH('a)}\ l = 0 \rangle$)
(auto simp add: take-bit-decr-eq)
qed

lemma *inc-less-eq-triv-imp*:
 $\langle w = -1 \rangle$ **if** $\langle w + 1 \leq w \rangle$ **for** $w :: 'a::\text{len word}$
proof (rule ccontr)
assume $\langle w \neq -1 \rangle$
with that **show** *False*

by transfer (auto simp add: take-bit-eq-mask-iff dest: take-bit-decr-eq)
qed

lemma less-eq-dec-triv-imp:
 $\langle w = 0 \rangle$ if $\langle w \leq w - 1 \rangle$ for $w :: \langle 'a::len \text{ word} \rangle$
proof (rule ccontr)
 assume $\langle w \neq 0 \rangle$
 with that show False
 by transfer (auto simp add: take-bit-eq-mask-iff dest: take-bit-decr-eq)
 qed

lemma inc-less-eq-iff:
 $\langle v + 1 \leq w \iff v = -1 \vee v < w \rangle$ for $v w :: \langle 'a::len \text{ word} \rangle$
 by (auto intro: inc-less-eq-triv-imp inc-le)

lemma less-eq-dec-iff:
 $\langle v \leq w - 1 \iff w = 0 \vee v < w \rangle$ for $v w :: \langle 'a::len \text{ word} \rangle$
 by (auto intro: less-eq-dec-triv-imp less-imp-less-eq-dec)

lemma inc-i: $1 \leq i \implies i < m \implies 1 \leq i + 1 \wedge i + 1 \leq m$
 for $i m :: 'a::len \text{ word}$
 by uint-arith

lemma dec-less-imp-less-eq:
 $\langle v \leq w \rangle$ if $\langle v - 1 < w \rangle$ for $v w :: \langle 'a::len \text{ word} \rangle$
 using that inc-le [of $\langle v - 1 \rangle w$] by simp

lemma less-inc-imp-less-eq:
 $\langle v \leq w \rangle$ if $\langle v < w + 1 \rangle$ for $v w :: \langle 'a::len \text{ word} \rangle$
 using that less-imp-less-eq-dec [of $v \langle w + 1 \rangle$] by simp

lemma less-eq-dec-self-iff-eq:
 $\langle w \leq w - 1 \iff w = 0 \rangle$ for $w :: \langle 'a::len \text{ word} \rangle$
 using less-eq-dec-iff [of $w w$] by simp

lemma inc-less-eq-self-iff-eq:
 $\langle w + 1 \leq w \iff w = -1 \rangle$ for $w :: \langle 'a::len \text{ word} \rangle$
 using inc-less-eq-triv-imp [of w] by auto

lemma udvd-incr-lem:
 $\llbracket up < uq; up = ua + n * uint K; uq = ua + n' * uint K \rrbracket$
 $\implies up + uint K \leq uq$
 by auto (metis int-distrib(1) linorder-not-less mult.left-neutral mult-right-mono
 uint-nonnegative zless-imp-add1-zle)

lemma udvd-incr':
 $p < q \implies uint p = ua + n * uint K \implies$
 $uint q = ua + n' * uint K \implies p + K \leq q$
 unfolding word-less-alt word-le-def

by (metis (full-types) order-trans udvd-incr-lem uint-add-le)

lemma *udvd-decr'*:
 assumes $p < q$ uint $p = ua + n * uint\ K$ uint $q = ua + n' * uint\ K$
 shows uint $q = ua + n' * uint\ K \implies p \leq q - K$
proof –
 have $\bigwedge w\ v. \text{uint}\ (w::'a\ \text{word}) \leq \text{uint}\ v + \text{uint}\ (w - v)$
 by (metis (no-types) add-diff-cancel-left' diff-add-cancel uint-add-le)
 moreover have uint $K + \text{uint}\ p \leq \text{uint}\ q$
 using assms by (metis (no-types) add-diff-cancel-left' diff-add-cancel udvd-incr-lem word-less-def)
 ultimately show ?thesis
 by (meson add-le-cancel-left order-trans word-less-eq-iff-unsigned)
qed

lemmas *udvd-incr-lem0* = *udvd-incr-lem* [where $ua=0$, unfolded *add-0-left*]
lemmas *udvd-incr0* = *udvd-incr'* [where $ua=0$, unfolded *add-0-left*]
lemmas *udvd-decr0* = *udvd-decr'* [where $ua=0$, unfolded *add-0-left*]

lemma *udvd-minus-le'*: $xy < k \implies z\ \text{udvd}\ xy \implies z\ \text{udvd}\ k \implies xy \leq k - z$
 unfolding *udvd-unfold-int*
 by (meson *udvd-decr0*)

lemma *udvd-incr2-K*:
 $p < a + s \implies a \leq a + s \implies K\ \text{udvd}\ s \implies K\ \text{udvd}\ p - a \implies a \leq p \implies$
 $0 < K \implies p \leq p + K \wedge p + K \leq a + s$
 unfolding *udvd-unfold-int*
 by (smt (verit, best) diff-add-cancel leD *udvd-incr-lem uint-plus-if'* word-less-eq-iff-unsigned word-sub-le)

107.22 Arithmetic type class instantiations

lemmas *word-le-0-iff* [simp] =
word-zero-le [THEN *leD*, THEN *antisym-conv1*]

lemma *word-of-int-nat*: $0 \leq x \implies \text{word-of-int}\ x = \text{of-nat}\ (\text{nat}\ x)$
 by *simp*

note that *iszero-def* is only for class *comm-semiring-1-cancel*, which requires word length ≥ 1 , ie $'a::\text{len}\ \text{word}$

lemma *iszero-word-no* [simp]:
iszero (numeral $\text{bin} :: 'a::\text{len}\ \text{word}$) =
iszero (take-bit *LENGTH* ('a) (numeral $\text{bin} :: \text{int}$))
 by (metis *iszero-def uint-0-iff uint-bintrunc*)

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [simp] =
eq-numeral-iff-iszero [where $'a='a::\text{len}\ \text{word}$]

lemma *word-less-eq-imp-half-less-eq*:
 $\langle v \text{ div } 2 \leq w \text{ div } 2 \rangle$ **if** $\langle v \leq w \rangle$ **for** $v \ w :: \langle 'a :: \text{len word} \rangle$
using *that* **by** (*simp add: word-le-nat-alt unat-div div-le-mono*)

lemma *word-half-less-imp-less-eq*:
 $\langle v \leq w \rangle$ **if** $\langle v \text{ div } 2 < w \text{ div } 2 \rangle$ **for** $v \ w :: \langle 'a :: \text{len word} \rangle$
using *that* *linorder-linear word-less-eq-imp-half-less-eq* **by** *fastforce*

107.23 Word and nat

lemma *word-nchotomy*: $\forall w :: 'a :: \text{len word}. \exists n. w = \text{of-nat } n \wedge n < 2^{\text{LENGTH}('a)}$
by (*metis of-nat-unat ucast-id unsigned-less*)

lemma *of-nat-eq*: $\text{of-nat } n = w \longleftrightarrow (\exists q. n = \text{unat } w + q * 2^{\text{LENGTH}('a)})$
for $w :: 'a :: \text{len word}$
using *mod-div-mult-eq* [*of-nat-2-LENGTH('a), symmetric*]
by (*auto simp flip: take-bit-eq-mod simp add: unsigned-of-nat*)

lemma *of-nat-eq-size*: $\text{of-nat } n = w \longleftrightarrow (\exists q. n = \text{unat } w + q * 2^{\text{size } w})$
unfolding *word-size* **by** (*rule of-nat-eq*)

lemma *of-nat-0*: $\text{of-nat } m = (0 :: 'a :: \text{len word}) \longleftrightarrow (\exists q. m = q * 2^{\text{LENGTH}('a)})$
by (*simp add: of-nat-eq*)

lemma *of-nat-2p* [*simp*]: $\text{of-nat } (2^{\text{LENGTH}('a)}) = (0 :: 'a :: \text{len word})$
by (*fact mult-1* [*symmetric, THEN iffD2* [*OF of-nat-0 exI*]])

lemma *of-nat-gt-0*: $\text{of-nat } k \neq 0 \implies 0 < k$
by (*cases k*) *auto*

lemma *of-nat-neq-0*: $0 < k \implies k < 2^{\text{LENGTH}('a :: \text{len})} \implies \text{of-nat } k \neq (0 :: 'a \text{ word})$
by (*auto simp : of-nat-0*)

lemma *Abs-fnat-hom-add*: $\text{of-nat } a + \text{of-nat } b = \text{of-nat } (a + b)$
by *simp*

lemma *Abs-fnat-hom-mult*: $\text{of-nat } a * \text{of-nat } b = (\text{of-nat } (a * b) :: 'a :: \text{len word})$
by (*simp add: wi-hom-mult*)

lemma *Abs-fnat-hom-Suc*: $\text{word-succ } (\text{of-nat } a) = \text{of-nat } (\text{Suc } a)$
by *transfer* (*simp add: ac-simps*)

lemma *Abs-fnat-hom-0*: $(0 :: 'a :: \text{len word}) = \text{of-nat } 0$
by *simp*

lemma *Abs-fnat-hom-1*: $(1 :: 'a :: \text{len word}) = \text{of-nat } (\text{Suc } 0)$
by *simp*

lemmas *Abs-fnat-homs* =
Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc
Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*: $a + b = \text{of-nat } (\text{unat } a + \text{unat } b)$
by *simp*

lemma *word-arith-nat-mult*: $a * b = \text{of-nat } (\text{unat } a * \text{unat } b)$
by *simp*

lemma *word-arith-nat-Suc*: $\text{word-succ } a = \text{of-nat } (\text{Suc } (\text{unat } a))$
by (*subst Abs-fnat-hom-Suc [symmetric]*) *simp*

lemma *word-arith-nat-div*: $a \text{ div } b = \text{of-nat } (\text{unat } a \text{ div } \text{unat } b)$
by (*metis of-int-of-nat-eq of-nat-unat of-nat-div word-div-def*)

lemma *word-arith-nat-mod*: $a \text{ mod } b = \text{of-nat } (\text{unat } a \text{ mod } \text{unat } b)$
by (*metis of-int-of-nat-eq of-nat-mod of-nat-unat word-mod-def*)

lemmas *word-arith-nat-defs* =
word-arith-nat-add word-arith-nat-mult
word-arith-nat-Suc Abs-fnat-hom-0
Abs-fnat-hom-1 word-arith-nat-div
word-arith-nat-mod

lemma *unat-of-nat*:
 $\langle \text{unat } (\text{word-of-nat } x :: 'a::\text{len word}) = x \text{ mod } 2 \wedge \text{LENGTH}('a) \rangle$
by *transfer (simp flip: take-bit-eq-mod add: nat-take-bit-eq)*

lemma *unat-cong*: $x = y \implies \text{unat } x = \text{unat } y$
by (*fact arg-cong*)

lemmas *unat-word-ariths* = *word-arith-nat-defs*
[*THEN trans [OF unat-cong unat-of-nat]*]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
[*unfolded linorder-not-less [symmetric] Not-eq-iff*]

lemma *unat-add-lem*:
 $\text{unat } x + \text{unat } y < 2 \wedge \text{LENGTH}('a) \longleftrightarrow \text{unat } (x + y) = \text{unat } x + \text{unat } y$
for $x \ y :: 'a::\text{len word}$
by (*metis mod-less unat-word-ariths(1) unsigned-less*)

lemma *unat-mult-lem*:
 $\text{unat } x * \text{unat } y < 2 \wedge \text{LENGTH}('a) \longleftrightarrow \text{unat } (x * y) = \text{unat } x * \text{unat } y$
for $x \ y :: 'a::\text{len word}$
by (*metis mod-less unat-word-ariths(2) unsigned-less*)

lemma *le-no-overflow*: $x \leq b \implies a \leq a + b \implies x \leq a + b$

```

for  $a\ b\ x :: 'a::len\ word$ 
using word-le-plus-either by blast

lemma uint-div:
   $\langle uint\ (x\ div\ y) = uint\ x\ div\ uint\ y \rangle$ 
by (fact uint-div-distrib)

lemma uint-mod:
   $\langle uint\ (x\ mod\ y) = uint\ x\ mod\ uint\ y \rangle$ 
by (fact uint-mod-distrib)

lemma no-plus-overflow-unat-size:  $x \leq x + y \longleftrightarrow unat\ x + unat\ y < 2^{\wedge size\ x}$ 
for  $x\ y :: 'a::len\ word$ 
unfolding word-size by unat-arith

lemmas no-olen-add-nat =
  no-plus-overflow-unat-size [unfolded word-size]

lemmas unat-plus-simple =
  trans [OF no-olen-add-nat unat-add-lem]

lemma word-div-mult:  $\llbracket 0 < y; unat\ x * unat\ y < 2^{\wedge LENGTH('a)} \rrbracket \Longrightarrow x * y\ div\ y = x$ 
for  $x\ y :: 'a::len\ word$ 
by (simp add: unat-eq-zero unat-mult-lem word-arith-nat-div)

lemma div-lt':  $i \leq k\ div\ x \Longrightarrow unat\ i * unat\ x < 2^{\wedge LENGTH('a)}$ 
for  $i\ k\ x :: 'a::len\ word$ 
by unat-arith (meson le-less-trans less-mult-imp-div-less not-le unsigned-less)

lemmas div-lt'' = order-less-imp-le [THEN div-lt']

lemma div-lt-mult:  $\llbracket i < k\ div\ x; 0 < x \rrbracket \Longrightarrow i * x < k$ 
for  $i\ k\ x :: 'a::len\ word$ 
by (metis div-le-mono div-lt'' not-le unat-div word-div-mult word-less-iff-unsigned)

lemma div-le-mult:  $\llbracket i \leq k\ div\ x; 0 < x \rrbracket \Longrightarrow i * x \leq k$ 
for  $i\ k\ x :: 'a::len\ word$ 
by (metis div-lt' less-mult-imp-div-less not-less unat-arith-simps(2) unat-div unat-mult-lem)

lemma div-lt-uint':  $i \leq k\ div\ x \Longrightarrow uint\ i * uint\ x < 2^{\wedge LENGTH('a)}$ 
for  $i\ k\ x :: 'a::len\ word$ 
unfolding uint-nat
by (metis div-lt' int-ops(7) of-nat-unat uint-mult-lem unat-mult-lem)

lemmas div-lt-uint'' = order-less-imp-le [THEN div-lt-uint']

lemma word-le-exists':  $x \leq y \Longrightarrow \exists z. y = x + z \wedge uint\ x + uint\ z < 2^{\wedge LENGTH('a)}$ 

```

```

for  $x\ y\ z :: 'a::len\ word$ 
by (metis add.commute diff-add-cancel no-olen-add)

lemmas plus-minus-not-NULL = order-less-imp-le [THEN plus-minus-not-NULL-ab]

lemmas plus-minus-no-overflow =
  order-less-imp-le [THEN plus-minus-no-overflow-ab]

lemmas mcs = word-less-minus-cancel word-less-minus-mono-left
  word-le-minus-cancel word-le-minus-mono-left

lemmas word-l-diffs = mcs [where  $y = w + x$ , unfolded add-diff-cancel] for  $w\ x$ 
lemmas word-diff-ls = mcs [where  $z = w + x$ , unfolded add-diff-cancel] for  $w\ x$ 
lemmas word-plus-mcs = word-diff-ls [where  $y = v + x$ , unfolded add-diff-cancel]
for  $v\ x$ 

lemma le-unat-uo:
   $\langle y \leq unat\ z \implies unat\ (word-of-nat\ y :: 'a\ word) = y \rangle$ 
  for  $z :: 'a::len\ word$ 
  by transfer (simp add: nat-take-bit-eq take-bit-nat-eq-self-iff le-less-trans)

lemmas thd = times-div-less-eq-dividend

lemmas uno-simps [THEN le-unat-uo] = mod-le-divisor div-le-dividend

lemma word-mod-div-equality:  $(n\ div\ b) * b + (n\ mod\ b) = n$ 
  for  $n\ b :: 'a::len\ word$ 
  by (fact div-mult-mod-eq)

lemma word-div-mult-le:  $a\ div\ b * b \leq a$ 
  for  $a\ b :: 'a::len\ word$ 
  by (metis div-le-mult mult-not-zero order.not-eq-order-implies-strict order-refl
word-zero-le)

lemma word-mod-less-divisor:  $0 < n \implies m\ mod\ n < n$ 
  for  $m\ n :: 'a::len\ word$ 
  by (simp add: unat-arith-simps)

lemma word-of-int-power-hom:  $word-of-int\ a \wedge n = (word-of-int\ (a \wedge n) :: 'a::len$ 
word)
  by (induct n (simp-all add: wi-hom-mult [symmetric]))

lemma word-arith-power-alt:  $a \wedge n = (word-of-int\ (uint\ a \wedge n) :: 'a::len\ word)$ 
  by (simp add : word-of-int-power-hom [symmetric])

lemma unatSuc:  $1 + n \neq 0 \implies unat\ (1 + n) = Suc\ (unat\ n)$ 
  for  $n :: 'a::len\ word$ 
  by unat-arith

```

107.24 Cardinality, finiteness of set of words

lemma *inj-on-word-of-int*: $\langle \text{inj-on } (\text{word-of-int} :: \text{int} \Rightarrow 'a \text{ word}) \{0..<2^{\text{LENGTH}('a::\text{len})}\} \rangle$
unfolding *inj-on-def*
by (*metis atLeastLessThan-iff word-of-int-inverse*)

lemma *range-uint*: $\langle \text{range } (\text{uint} :: 'a \text{ word} \Rightarrow \text{int}) = \{0..<2^{\text{LENGTH}('a::\text{len})}\} \rangle$
apply *transfer*
apply (*auto simp: image-iff*)
apply (*metis take-bit-int-eq-self-iff*)
done

lemma *UNIV-eq*: $\langle (\text{UNIV} :: 'a \text{ word set}) = \text{word-of-int } ' \{0..<2^{\text{LENGTH}('a::\text{len})}\} \rangle$
by (*auto simp: image-iff*) (*metis atLeastLessThan-iff linorder-not-le uint-split*)

lemma *card-word*: $\text{CARD}('a \text{ word}) = 2^{\text{LENGTH}('a::\text{len})}$
by (*simp add: UNIV-eq card-image inj-on-word-of-int*)

lemma *card-word-size*: $\text{CARD}('a \text{ word}) = 2^{\text{size } x}$
for $x :: 'a::\text{len} \text{ word}$
unfolding *word-size* **by** (*rule card-word*)

end

instance *word* :: $(\text{len}) \text{ finite}$
by *standard* (*simp add: UNIV-eq*)

107.25 Bitwise Operations on Words

context
includes *bit-operations-syntax*
begin

lemma *word-wi-log-defs*:
 $\text{NOT } (\text{word-of-int } a) = \text{word-of-int } (\text{NOT } a)$
 $\text{word-of-int } a \text{ AND } \text{word-of-int } b = \text{word-of-int } (a \text{ AND } b)$
 $\text{word-of-int } a \text{ OR } \text{word-of-int } b = \text{word-of-int } (a \text{ OR } b)$
 $\text{word-of-int } a \text{ XOR } \text{word-of-int } b = \text{word-of-int } (a \text{ XOR } b)$
by (*transfer, rule refl*)**+**

lemma *word-no-log-defs* [*simp*]:
 $\text{NOT } (\text{numeral } a) = \text{word-of-int } (\text{NOT } (\text{numeral } a))$
 $\text{NOT } (- \text{numeral } a) = \text{word-of-int } (\text{NOT } (- \text{numeral } a))$
 $\text{numeral } a \text{ AND } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ AND } \text{numeral } b)$
 $\text{numeral } a \text{ AND } - \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ AND } - \text{numeral } b)$
 $- \text{numeral } a \text{ AND } \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ AND } \text{numeral } b)$
 $- \text{numeral } a \text{ AND } - \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ AND } - \text{numeral } b)$
 $\text{numeral } a \text{ OR } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ OR } \text{numeral } b)$
 $\text{numeral } a \text{ OR } - \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ OR } - \text{numeral } b)$
 $- \text{numeral } a \text{ OR } \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ OR } \text{numeral } b)$

$- \text{numeral } a \text{ OR } - \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ OR } - \text{numeral } b)$
 $\text{numeral } a \text{ XOR } \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ XOR } \text{numeral } b)$
 $\text{numeral } a \text{ XOR } - \text{numeral } b = \text{word-of-int } (\text{numeral } a \text{ XOR } - \text{numeral } b)$
 $- \text{numeral } a \text{ XOR } \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ XOR } \text{numeral } b)$
 $- \text{numeral } a \text{ XOR } - \text{numeral } b = \text{word-of-int } (- \text{numeral } a \text{ XOR } - \text{numeral } b)$
by (transfer, rule refl)+

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [simp]:

$\text{NOT } (1::'a::\text{len word}) = -2$
 $1 \text{ AND } \text{numeral } b = \text{word-of-int } (1 \text{ AND } \text{numeral } b)$
 $1 \text{ AND } - \text{numeral } b = \text{word-of-int } (1 \text{ AND } - \text{numeral } b)$
 $\text{numeral } a \text{ AND } 1 = \text{word-of-int } (\text{numeral } a \text{ AND } 1)$
 $- \text{numeral } a \text{ AND } 1 = \text{word-of-int } (- \text{numeral } a \text{ AND } 1)$
 $1 \text{ OR } \text{numeral } b = \text{word-of-int } (1 \text{ OR } \text{numeral } b)$
 $1 \text{ OR } - \text{numeral } b = \text{word-of-int } (1 \text{ OR } - \text{numeral } b)$
 $\text{numeral } a \text{ OR } 1 = \text{word-of-int } (\text{numeral } a \text{ OR } 1)$
 $- \text{numeral } a \text{ OR } 1 = \text{word-of-int } (- \text{numeral } a \text{ OR } 1)$
 $1 \text{ XOR } \text{numeral } b = \text{word-of-int } (1 \text{ XOR } \text{numeral } b)$
 $1 \text{ XOR } - \text{numeral } b = \text{word-of-int } (1 \text{ XOR } - \text{numeral } b)$
 $\text{numeral } a \text{ XOR } 1 = \text{word-of-int } (\text{numeral } a \text{ XOR } 1)$
 $- \text{numeral } a \text{ XOR } 1 = \text{word-of-int } (- \text{numeral } a \text{ XOR } 1)$
apply (simp-all add: word-uint-eq-iff unsigned-not-eq unsigned-and-eq
 unsigned-or-eq
 unsigned-xor-eq of-nat-take-bit ac-simps unsigned-of-int)
apply (simp-all add: minus-numeral-eq-not-sub-one)
apply (simp-all only: sub-one-eq-not-neg bit.xor-compl-right take-bit-xor bit.double-compl)
apply simp-all
done

Special cases for when one of the arguments equals -1.

lemma *word-bitwise-m1-simps* [simp]:

$\text{NOT } (-1::'a::\text{len word}) = 0$
 $(-1::'a::\text{len word}) \text{ AND } x = x$
 $x \text{ AND } (-1::'a::\text{len word}) = x$
 $(-1::'a::\text{len word}) \text{ OR } x = -1$
 $x \text{ OR } (-1::'a::\text{len word}) = -1$
 $(-1::'a::\text{len word}) \text{ XOR } x = \text{NOT } x$
 $x \text{ XOR } (-1::'a::\text{len word}) = \text{NOT } x$
by (transfer, simp)+

lemma *word-of-int-not-numeral-eq* [simp]:

$\langle (\text{word-of-int } (\text{NOT } (\text{numeral bin})) :: 'a::\text{len word}) = - \text{numeral bin} - 1 \rangle$
by transfer (simp add: not-eq-complement)

lemma *uint-and*:

$\langle \text{uint } (x \text{ AND } y) = \text{uint } x \text{ AND } \text{uint } y \rangle$
by transfer simp

lemma *uint-or*:

$\langle \text{uint } (x \text{ OR } y) = \text{uint } x \text{ OR } \text{uint } y \rangle$

by *transfer simp*

lemma *uint-xor*:

$\langle \text{uint } (x \text{ XOR } y) = \text{uint } x \text{ XOR } \text{uint } y \rangle$

by *transfer simp*

— get from commutativity, associativity etc of *int-and* etc to same for *word-and* etc

lemmas *bwsimps* =

wi-hom-add

word-wi-log-defs

lemma *word-bw-assocs*:

$(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$

$(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$

$(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$

for $x :: 'a::\text{len word}$

by *(fact ac-simps)+*

lemma *word-bw-comms*:

$x \text{ AND } y = y \text{ AND } x$

$x \text{ OR } y = y \text{ OR } x$

$x \text{ XOR } y = y \text{ XOR } x$

for $x :: 'a::\text{len word}$

by *(fact ac-simps)+*

lemma *word-bw-lcs*:

$y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$

$y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$

$y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$

for $x :: 'a::\text{len word}$

by *(fact ac-simps)+*

lemma *word-log-esimps*:

$x \text{ AND } 0 = 0$

$x \text{ AND } -1 = x$

$x \text{ OR } 0 = x$

$x \text{ OR } -1 = -1$

$x \text{ XOR } 0 = x$

$x \text{ XOR } -1 = \text{NOT } x$

$0 \text{ AND } x = 0$

$-1 \text{ AND } x = x$

$0 \text{ OR } x = x$

$-1 \text{ OR } x = -1$

$0 \text{ XOR } x = x$

$-1 \text{ XOR } x = \text{NOT } x$

for $x :: 'a::\text{len word}$

by *simp-all*

lemma *word-not-dist*:

$NOT (x OR y) = NOT x AND NOT y$

$NOT (x AND y) = NOT x OR NOT y$

for $x :: 'a::len\ word$

by *simp-all*

lemma *word-bw-same*:

$x AND x = x$

$x OR x = x$

$x XOR x = 0$

for $x :: 'a::len\ word$

by *simp-all*

lemma *word-ao-absorbs* [*simp*]:

$x AND (y OR x) = x$

$x OR y AND x = x$

$x AND (x OR y) = x$

$y AND x OR x = x$

$(y OR x) AND x = x$

$x OR x AND y = x$

$(x OR y) AND x = x$

$x AND y OR x = x$

for $x :: 'a::len\ word$

by (*auto intro: bit-eqI simp add: bit-and-iff bit-or-iff*)

lemma *word-not-not* [*simp*]: $NOT (NOT x) = x$

for $x :: 'a::len\ word$

by (*fact bit.double-compl*)

lemma *word-ao-dist*: $(x OR y) AND z = x AND z OR y AND z$

for $x :: 'a::len\ word$

by (*fact bit.conj-disj-distrib2*)

lemma *word-oa-dist*: $x AND y OR z = (x OR z) AND (y OR z)$

for $x :: 'a::len\ word$

by (*fact bit.disj-conj-distrib2*)

lemma *word-add-not* [*simp*]: $x + NOT x = -1$

for $x :: 'a::len\ word$

by (*simp add: not-eq-complement*)

lemma *word-plus-and-or* [*simp*]: $(x AND y) + (x OR y) = x + y$

for $x :: 'a::len\ word$

by *transfer* (*simp add: plus-and-or*)

lemma *leoa*: $w = x OR y \implies y = w AND y$

for $x :: 'a::len\ word$

by *auto*

lemma *leao*: $w' = x' \text{ AND } y' \implies x' = x' \text{ OR } w'$
 for $x' :: 'a::\text{len word}$
 by *auto*

lemma *word-ao-equiv*: $w = w \text{ OR } w' \longleftrightarrow w' = w \text{ AND } w'$
 for $w w' :: 'a::\text{len word}$
 by (*auto intro: leoa leao*)

lemma *le-word-or2*: $x \leq x \text{ OR } y$
 for $x y :: 'a::\text{len word}$
 by (*simp add: or-greater-eq uint-or word-le-def*)

lemmas *le-word-or1* = *xtrans*(3) [*OF word-bw-comms* (2) *le-word-or2*]
lemmas *word-and-le1* = *xtrans*(3) [*OF word-ao-absorbs* (4) [*symmetric*] *le-word-or2*]
lemmas *word-and-le2* = *xtrans*(3) [*OF word-ao-absorbs* (8) [*symmetric*] *le-word-or2*]

lemma *bit-horner-sum-bit-word-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{horner-sum of-bool } (2 :: 'a::\text{len word}) \text{ bs}) \text{ } n \rangle$
 $\longleftrightarrow n < \min \text{LENGTH}('a) (\text{length bs}) \wedge \text{bs ! } n \rangle$
 by *transfer (simp add: bit-horner-sum-bit-iff)*

definition *word-reverse* :: $\langle 'a::\text{len word} \Rightarrow 'a \text{ word} \rangle$
 where $\langle \text{word-reverse } w = \text{horner-sum of-bool } 2 (\text{rev } (\text{map } (\text{bit } w) [0..<\text{LENGTH}('a)])) \rangle$

lemma *bit-word-reverse-iff* [*bit-simps*]:
 $\langle \text{bit } (\text{word-reverse } w) \text{ } n \longleftrightarrow n < \text{LENGTH}('a) \wedge \text{bit } w (\text{LENGTH}('a) - \text{Suc } n) \rangle$
 for $w :: \langle 'a::\text{len word} \rangle$
 by (*cases* $\langle n < \text{LENGTH}('a) \rangle$)
 (*simp-all add: word-reverse-def bit-horner-sum-bit-word-iff rev-nth*)

lemma *word-rev-rev* [*simp*] : $\text{word-reverse } (\text{word-reverse } w) = w$
 by (*rule bit-word-eqI*)
 (*auto simp: bit-word-reverse-iff bit-imp-le-length Suc-diff-Suc*)

lemma *word-rev-gal*: $\text{word-reverse } w = u \implies \text{word-reverse } u = w$
 by (*metis word-rev-rev*)

lemma *word-rev-gal'*: $u = \text{word-reverse } w \implies w = \text{word-reverse } u$
 by *simp*

lemma *word-eq-reverseI*:
 $\langle v = w \rangle \text{ if } \langle \text{word-reverse } v = \text{word-reverse } w \rangle$
 by (*metis that word-rev-rev*)

lemma *uint-2p*: $(0 :: 'a::\text{len word}) < 2 \wedge n \implies \text{uint } (2 \wedge n :: 'a::\text{len word}) = 2 \wedge n$
 by (*cases* $\langle n < \text{LENGTH}('a) \rangle$; *transfer; force*)

lemma *word-of-int-2p*: $(\text{word-of-int } (2 \wedge n) :: 'a::\text{len word}) = 2 \wedge n$
by *simp*

107.25.1 shift functions in terms of lists of bools

lemma *drop-bit-word-numeral* [*simp*]:
 $\langle \text{drop-bit (numeral } n) \text{ (numeral } k) =$
 $(\text{word-of-int (drop-bit (numeral } n) \text{ (take-bit LENGTH('a) (numeral } k))) :: 'a::\text{len}$
 $\text{word}) \rangle$
by *transfer simp*

lemma *drop-bit-word-Suc-numeral* [*simp*]:
 $\langle \text{drop-bit (Suc } n) \text{ (numeral } k) =$
 $(\text{word-of-int (drop-bit (Suc } n) \text{ (take-bit LENGTH('a) (numeral } k))) :: 'a::\text{len}$
 $\text{word}) \rangle$
by *transfer simp*

lemma *drop-bit-word-minus-numeral* [*simp*]:
 $\langle \text{drop-bit (numeral } n) \text{ (- numeral } k) =$
 $(\text{word-of-int (drop-bit (numeral } n) \text{ (take-bit LENGTH('a) (- numeral } k))) ::$
 $'a::\text{len word}) \rangle$
by *transfer simp*

lemma *drop-bit-word-Suc-minus-numeral* [*simp*]:
 $\langle \text{drop-bit (Suc } n) \text{ (- numeral } k) =$
 $(\text{word-of-int (drop-bit (Suc } n) \text{ (take-bit LENGTH('a) (- numeral } k))) :: 'a::\text{len}$
 $\text{word}) \rangle$
by *transfer simp*

lemma *signed-drop-bit-word-numeral* [*simp*]:
 $\langle \text{signed-drop-bit (numeral } n) \text{ (numeral } k) =$
 $(\text{word-of-int (drop-bit (numeral } n) \text{ (signed-take-bit (LENGTH('a) - 1) (numeral}$
 $k))) :: 'a::\text{len word}) \rangle$
by *transfer simp*

lemma *signed-drop-bit-word-Suc-numeral* [*simp*]:
 $\langle \text{signed-drop-bit (Suc } n) \text{ (numeral } k) =$
 $(\text{word-of-int (drop-bit (Suc } n) \text{ (signed-take-bit (LENGTH('a) - 1) (numeral}$
 $k))) :: 'a::\text{len word}) \rangle$
by *transfer simp*

lemma *signed-drop-bit-word-minus-numeral* [*simp*]:
 $\langle \text{signed-drop-bit (numeral } n) \text{ (- numeral } k) =$
 $(\text{word-of-int (drop-bit (numeral } n) \text{ (signed-take-bit (LENGTH('a) - 1) (-}$
 $\text{numeral } k))) :: 'a::\text{len word}) \rangle$
by *transfer simp*

lemma *signed-drop-bit-word-Suc-minus-numeral* [*simp*]:
 $\langle \text{signed-drop-bit (Suc } n) \text{ (- numeral } k) =$

(word-of-int (drop-bit (Suc n) (signed-take-bit (LENGTH('a) - 1) (- numeral k)))) :: 'a::len word)›

by transfer simp

lemma take-bit-word-numeral [simp]:

⟨take-bit (numeral n) (numeral k) =

(word-of-int (take-bit (min LENGTH('a) (numeral n)) (numeral k)) :: 'a::len word)›

by transfer rule

lemma take-bit-word-Suc-numeral [simp]:

⟨take-bit (Suc n) (numeral k) =

(word-of-int (take-bit (min LENGTH('a) (Suc n)) (numeral k)) :: 'a::len word)›

by transfer rule

lemma take-bit-word-minus-numeral [simp]:

⟨take-bit (numeral n) (- numeral k) =

(word-of-int (take-bit (min LENGTH('a) (numeral n)) (- numeral k)) :: 'a::len word)›

by transfer rule

lemma take-bit-word-Suc-minus-numeral [simp]:

⟨take-bit (Suc n) (- numeral k) =

(word-of-int (take-bit (min LENGTH('a) (Suc n)) (- numeral k)) :: 'a::len word)›

by transfer rule

lemma signed-take-bit-word-numeral [simp]:

⟨signed-take-bit (numeral n) (numeral k) =

(word-of-int (signed-take-bit (numeral n) (take-bit LENGTH('a) (numeral k)))) :: 'a::len word)›

by transfer rule

lemma signed-take-bit-word-Suc-numeral [simp]:

⟨signed-take-bit (Suc n) (numeral k) =

(word-of-int (signed-take-bit (Suc n) (take-bit LENGTH('a) (numeral k)))) :: 'a::len word)›

by transfer rule

lemma signed-take-bit-word-minus-numeral [simp]:

⟨signed-take-bit (numeral n) (- numeral k) =

(word-of-int (signed-take-bit (numeral n) (take-bit LENGTH('a) (- numeral k)))) :: 'a::len word)›

by transfer rule

lemma signed-take-bit-word-Suc-minus-numeral [simp]:

⟨signed-take-bit (Suc n) (- numeral k) =

(word-of-int (signed-take-bit (Suc n) (take-bit LENGTH('a) (- numeral k)))) :: 'a::len word)›

by *transfer rule*

lemma *False-map2-or*: $\llbracket \text{set } xs \subseteq \{\text{False}\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\vee) \text{ } xs$
 $ys = ys$
 by (*induction xs arbitrary: ys*) (*auto simp: length-Suc-conv*)

lemma *align-lem-or*:

assumes $\text{length } xs = n + m$ $\text{length } ys = n + m$
 and $\text{drop } m \text{ } xs = \text{replicate } n \text{ False take } m \text{ } ys = \text{replicate } m \text{ False}$
 shows $\text{map2 } (\vee) \text{ } xs \text{ } ys = \text{take } m \text{ } xs @ \text{drop } m \text{ } ys$
 using *assms*
proof (*induction xs arbitrary: ys m*)
 case (*Cons a xs*)
 then show ?*case*
 by (*cases m*) (*auto simp: length-Suc-conv False-map2-or*)
qed *auto*

lemma *False-map2-and*: $\llbracket \text{set } xs \subseteq \{\text{False}\}; \text{length } ys = \text{length } xs \rrbracket \implies \text{map2 } (\wedge) \text{ } xs$
 $ys = xs$
 by (*induction xs arbitrary: ys*) (*auto simp: length-Suc-conv*)

lemma *align-lem-and*:

assumes $\text{length } xs = n + m$ $\text{length } ys = n + m$
 and $\text{drop } m \text{ } xs = \text{replicate } n \text{ False take } m \text{ } ys = \text{replicate } m \text{ False}$
 shows $\text{map2 } (\wedge) \text{ } xs \text{ } ys = \text{replicate } (n + m) \text{ False}$
 using *assms*
proof (*induction xs arbitrary: ys m*)
 case (*Cons a xs*)
 then show ?*case*
 by (*cases m*) (*auto simp: length-Suc-conv set-replicate-conv-if False-map2-and*)
qed *auto*

107.25.2 Mask

lemma *minus-1-eq-mask*:

$\langle - 1 = (\text{mask } \text{LENGTH}('a) :: 'a::\text{len word}) \rangle$
 by (*rule bit-eqI*) (*simp add: bit-exp-iff bit-mask-iff*)

lemma *mask-eq-decr-exp*:

$\langle \text{mask } n = 2^{\wedge} n - (1 :: 'a::\text{len word}) \rangle$
 by (*fact mask-eq-exp-minus-1*)

lemma *mask-Suc-rec*:

$\langle \text{mask } (\text{Suc } n) = 2 * \text{mask } n + (1 :: 'a::\text{len word}) \rangle$
 by (*simp add: mask-eq-exp-minus-1*)

context

begin

qualified lemma *bit-mask-iff* [*bit-simps*]:

$\langle \text{bit } (\text{mask } m :: 'a::\text{len word}) \ n \longleftrightarrow n < \min \text{LENGTH}('a) \ m \rangle$

by (*simp add: bit-mask-iff not-le*)

end

lemma *mask-bin*: $\text{mask } n = \text{word-of-int } (\text{take-bit } n \ (- \ 1))$

by *transfer simp*

lemma *and-mask-bintr*: $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \ (\text{uint } w))$

by (*transfer (simp add: ac-simps take-bit-eq-mask)*)

lemma *and-mask-wi*: $\text{word-of-int } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \ i)$

by (*simp add: take-bit-eq-mask of-int-and-eq of-int-mask-eq*)

lemma *and-mask-wi'*:

$\text{word-of-int } i \text{ AND } \text{mask } n = (\text{word-of-int } (\text{take-bit } (\min \text{LENGTH}('a) \ n) \ i) :: 'a::\text{len word})$

by (*auto simp: and-mask-wi min-def wi-bintr*)

lemma *and-mask-no*: $\text{numeral } i \text{ AND } \text{mask } n = \text{word-of-int } (\text{take-bit } n \ (\text{numeral } i))$

unfolding *word-numeral-alt* **by** (*rule and-mask-wi*)

lemma *and-mask-mod-2p*: $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{uint } w \bmod 2^n)$

by (*simp only: and-mask-bintr take-bit-eq-mod*)

lemma *uint-mask-eq*:

$\langle \text{uint } (\text{mask } n :: 'a::\text{len word}) = \text{mask } (\min \text{LENGTH}('a) \ n) \rangle$

by *transfer simp*

lemma *and-mask-lt-2p*: $\text{uint } (w \text{ AND } \text{mask } n) < 2^n$

by (*metis take-bit-eq-mask take-bit-int-less-exp unsigned-take-bit-eq*)

lemma *mask-eq-iff*: $w \text{ AND } \text{mask } n = w \longleftrightarrow \text{uint } w < 2^n$

by (*metis and-mask-bintr and-mask-lt-2p take-bit-int-eq-self take-bit-nonnegative uint-sint word-of-int-uint*)

lemma *and-mask-dvd*: $2^n \text{ dvd } \text{uint } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$

by (*simp flip: take-bit-eq-mask take-bit-eq-mod unsigned-take-bit-eq add: dvd-eq-mod-eq-0 uint-0-iff*)

lemma *and-mask-dvd-nat*: $2^n \text{ dvd } \text{unat } w \longleftrightarrow w \text{ AND } \text{mask } n = 0$

by (*simp flip: take-bit-eq-mask take-bit-eq-mod unsigned-take-bit-eq add: dvd-eq-mod-eq-0 unat-0-iff uint-0-iff*)

lemma *word-2p-lem*: $n < \text{size } w \implies w < 2^n = (\text{uint } w < 2^n)$

for $w :: 'a::\text{len word}$

by *transfer simp*

lemma *less-mask-eq*:
fixes $x :: 'a::len\ word$
assumes $x < 2 \wedge n$ **shows** $x \text{ AND } mask\ n = x$
by (*metis (no-types) assms lt2p-lem mask-eq-iff not-less word-2p-lem word-size*)

lemmas *mask-eq-iff-w2p* = *trans* [*OF* *mask-eq-iff word-2p-lem* [*symmetric*]]

lemmas *and-mask-less'* = *iffD2* [*OF* *word-2p-lem and-mask-lt-2p*, *simplified word-size*]

lemma *and-mask-less-size*: $n < size\ x \implies x \text{ AND } mask\ n < 2 \wedge n$
for $x :: 'a::len\ word$
unfolding *word-size* **by** (*erule and-mask-less'*)

lemma *word-mod-2p-is-mask* [*OF refl*]: $c = 2 \wedge n \implies c > 0 \implies x \bmod c = x$
AND *mask n*
for $c\ x :: 'a::len\ word$
by (*auto simp: word-mod-def uint-2p and-mask-mod-2p*)

lemma *mask-eqs*:
 $(a \text{ AND } mask\ n) + b \text{ AND } mask\ n = a + b \text{ AND } mask\ n$
 $a + (b \text{ AND } mask\ n) \text{ AND } mask\ n = a + b \text{ AND } mask\ n$
 $(a \text{ AND } mask\ n) - b \text{ AND } mask\ n = a - b \text{ AND } mask\ n$
 $a - (b \text{ AND } mask\ n) \text{ AND } mask\ n = a - b \text{ AND } mask\ n$
 $a * (b \text{ AND } mask\ n) \text{ AND } mask\ n = a * b \text{ AND } mask\ n$
 $(b \text{ AND } mask\ n) * a \text{ AND } mask\ n = b * a \text{ AND } mask\ n$
 $(a \text{ AND } mask\ n) + (b \text{ AND } mask\ n) \text{ AND } mask\ n = a + b \text{ AND } mask\ n$
 $(a \text{ AND } mask\ n) - (b \text{ AND } mask\ n) \text{ AND } mask\ n = a - b \text{ AND } mask\ n$
 $(a \text{ AND } mask\ n) * (b \text{ AND } mask\ n) \text{ AND } mask\ n = a * b \text{ AND } mask\ n$
 $-(a \text{ AND } mask\ n) \text{ AND } mask\ n = -a \text{ AND } mask\ n$
 $word\ succ\ (a \text{ AND } mask\ n) \text{ AND } mask\ n = word\ succ\ a \text{ AND } mask\ n$
 $word\ pred\ (a \text{ AND } mask\ n) \text{ AND } mask\ n = word\ pred\ a \text{ AND } mask\ n$
using *word-of-int-Ex* [**where** $x=a$] *word-of-int-Ex* [**where** $x=b$]
unfolding *take-bit-eq-mask* [*symmetric*]
by (*transfer; simp add: take-bit-eq-mod mod-simps*)+

lemma *mask-power-eq*: $(x \text{ AND } mask\ n) \wedge^k \text{ AND } mask\ n = x \wedge^k \text{ AND } mask\ n$
for $x :: 'a::len\ word$
using *word-of-int-Ex* [**where** $x=x$]
unfolding *take-bit-eq-mask* [*symmetric*]
by (*transfer; simp add: take-bit-eq-mod mod-simps*)+

lemma *mask-full* [*simp*]: $mask\ LENGTH('a) = (-\ 1 :: 'a::len\ word)$
by *transfer simp*

107.25.3 Slices

definition *slice1* :: $\langle nat \Rightarrow 'a::len\ word \Rightarrow 'b::len\ word \rangle$
where $\langle slice1\ n\ w = (if\ n < LENGTH('a)$

then *ucast* (*drop-bit* (*LENGTH*('a) - n) w)
 else *push-bit* (n - *LENGTH*('a)) (*ucast* w))

lemma *bit-slice1-iff* [*bit-simps*]:

⟨*bit* (*slice1* m w :: 'b::len word) n \longleftrightarrow m - *LENGTH*('a) \leq n \wedge n < min
LENGTH('b) m
 \wedge *bit* w (n + (*LENGTH*('a) - m) - (m - *LENGTH*('a)))⟩
for w :: 'a::len word
by (*auto simp: slice1-def bit-ucast-iff bit-drop-bit-eq bit-push-bit-iff not-less not-le*
ac-simps
dest: bit-imp-le-length)

definition *slice* :: nat \Rightarrow 'a::len word \Rightarrow 'b::len word
where ⟨*slice* n = *slice1* (*LENGTH*('a) - n)⟩

lemma *bit-slice-iff* [*bit-simps*]:

⟨*bit* (*slice* m w :: 'b::len word) n \longleftrightarrow n < min *LENGTH*('b) (*LENGTH*('a) -
 m) \wedge *bit* w (n + *LENGTH*('a) - (*LENGTH*('a) - m))⟩
for w :: 'a::len word
by (*simp add: slice-def word-size bit-slice1-iff*)

lemma *slice1-0* [*simp*] : *slice1* n 0 = 0
unfolding *slice1-def* **by** *simp*

lemma *slice-0* [*simp*] : *slice* n 0 = 0
unfolding *slice-def* **by** *auto*

lemma *ucast-slice1*: *ucast* w = *slice1* (*size* w) w
unfolding *slice1-def* **by** (*simp add: size-word.rep-eq*)

lemma *ucast-slice*: *ucast* w = *slice* 0 w
by (*simp add: slice-def slice1-def*)

lemma *slice-id*: *slice* 0 t = t
by (*simp only: ucast-slice [symmetric] ucast-id*)

lemma *rev-slice1*:

⟨*slice1* n (*word-reverse* w :: 'b::len word) = *word-reverse* (*slice1* k w :: 'a::len
 word)⟩
if ⟨n + k = *LENGTH*('a) + *LENGTH*('b)⟩
proof (*rule bit-word-eqI*)
fix m
assume *: ⟨m < *LENGTH*('a)⟩
from that **have** **: ⟨*LENGTH*('b) = n + k - *LENGTH*('a)⟩
by *simp*
show ⟨*bit* (*slice1* n (*word-reverse* w :: 'b word) :: 'a word) m \longleftrightarrow *bit* (*word-reverse*
 (*slice1* k w :: 'a word)) m⟩
unfolding *bit-slice1-iff bit-word-reverse-iff*
using * **

by (cases $\langle n \leq \text{LENGTH}('a) \rangle$; cases $\langle k \leq \text{LENGTH}('a) \rangle$) auto
qed

lemma *rev-slice*:

$n + k + \text{LENGTH}('a::\text{len}) = \text{LENGTH}('b::\text{len}) \implies$
 $\text{slice } n (\text{word-reverse } (w::'b \text{ word})) = \text{word-reverse } (\text{slice } k w :: 'a \text{ word})$
unfolding *slice-def word-size*
by (*simp add: rev-slice1*)

107.25.4 Recast

definition *recast* :: $\langle 'a::\text{len word} \Rightarrow 'b::\text{len word} \rangle$
where $\langle \text{recast} = \text{slice1 } \text{LENGTH}('b) \rangle$

lemma *bit-recast-iff* [*bit-simps*]:

$\langle \text{bit } (\text{recast } w :: 'b::\text{len word}) \ n \longleftrightarrow \text{LENGTH}('b) - \text{LENGTH}('a) \leq n \wedge n < \text{LENGTH}('b)$
 $\wedge \text{bit } w (n + (\text{LENGTH}('a) - \text{LENGTH}('b)) - (\text{LENGTH}('b) - \text{LENGTH}('a))) \rangle$
for $w :: \langle 'a::\text{len word} \rangle$
by (*simp add: recast-def bit-slice1-iff*)

lemma *recast-slice1* [*OF refl*]: $\text{rc} = \text{recast } w \implies \text{slice1 } (\text{size } \text{rc}) w = \text{rc}$
by (*simp add: recast-def word-size*)

lemma *recast-rev-ucast* [*OF refl refl refl*]:

$\text{cs} = [\text{rc}, \text{uc}] \implies \text{rc} = \text{recast } (\text{word-reverse } w) \implies \text{uc} = \text{ucast } w \implies$
 $\text{rc} = \text{word-reverse } \text{uc}$
by (*metis rev-slice1 recast-slice1 ucast-slice1 word-size*)

lemma *recast-ucast*: $\text{recast } w = \text{word-reverse } (\text{ucast } (\text{word-reverse } w))$
using *recast-rev-ucast* [*of word-reverse w*] **by** *simp*

lemma *ucast-recast*: $\text{ucast } w = \text{word-reverse } (\text{recast } (\text{word-reverse } w))$
by (*fact recast-rev-ucast* [*THEN word-rev-gal*])

lemma *ucast-rev-recast*: $\text{ucast } (\text{word-reverse } w) = \text{word-reverse } (\text{recast } w)$
by (*fact recast-ucast* [*THEN word-rev-gal*])

linking recast and cast via shift

lemmas *wsst-TYs* = *source-size target-size word-size*

lemmas *sym-notr* =

not-iff [*THEN iffD2*, *THEN not-sym*, *THEN not-iff* [*THEN iffD1*]]

107.26 Split and cat

lemmas *word-split-bin'* = *word-split-def*

lemmas *word-cat-bin'* = *word-cat-eq*

— this odd result is analogous to *ucast-id*, result to the length given by the result type

lemma *word-cat-id*: *word-cat a b = b*
by *transfer (simp add: take-bit-concat-bit-eq)*

lemma *word-cat-split-alt*: $\llbracket \text{size } w \leq \text{size } u + \text{size } v; \text{word-split } w = (u, v) \rrbracket \implies$
word-cat u v = w
unfolding *word-split-def*
by (*rule bit-word-eqI*) (*auto simp: bit-word-cat-iff not-less word-size bit-ucast-iff bit-drop-bit-eq*)

lemmas *word-cat-split-size* = *sym [THEN [2] word-cat-split-alt [symmetric]]*

107.26.1 Split and slice

lemma *split-slices*:
assumes *word-split w = (u, v)*
shows *u = slice (size v) w \wedge v = slice 0 w*
unfolding *word-size*
proof (*intro conjI*)
have $\S: \bigwedge n. \llbracket \text{ucast } (\text{drop-bit } \text{LENGTH}('b) \ w) = u; \text{LENGTH}('c) < \text{LENGTH}('b) \rrbracket$
 $\implies \neg \text{bit } u \ n$
by (*metis bit-take-bit-iff bit-word-of-int-iff diff-is-0-eq' drop-bit-take-bit less-imp-le less-nat-zero-code of-int-uint unsigned-drop-bit-eq*)
show *u = slice LENGTH('b) w*
proof (*rule bit-word-eqI*)
show *bit u n = bit ((slice LENGTH('b) w)::'a word) n if n < LENGTH('a)*
for *n*
using *assms bit-imp-le-length*
unfolding *word-split-def bit-slice-iff*
by (*fastforce simp: \S ac-simps word-size bit-ucast-iff bit-drop-bit-eq*)
qed
show *v = slice 0 w*
by (*metis Pair-inject assms ucast-slice word-split-bin'*)
qed

lemma *slice-cat1* [*OF refl*]:
 $\llbracket \text{wc} = \text{word-cat } a \ b; \text{size } a + \text{size } b \leq \text{size } \text{wc} \rrbracket \implies \text{slice } (\text{size } b) \ \text{wc} = a$
by (*rule bit-word-eqI*) (*auto simp: bit-slice-iff bit-word-cat-iff word-size*)

lemmas *slice-cat2* = *trans [OF slice-id word-cat-id]*

lemma *cat-slices*:
 $\llbracket a = \text{slice } n \ c; b = \text{slice } 0 \ c; n = \text{size } b; \text{size } c \leq \text{size } a + \text{size } b \rrbracket \implies \text{word-cat } a \ b = c$
by (*rule bit-word-eqI*) (*auto simp: bit-slice-iff bit-word-cat-iff word-size*)

lemma *word-split-cat-alt*:
assumes $w = \text{word-cat } u \ v$ **and** *size*: $\text{size } u + \text{size } v \leq \text{size } w$
shows $\text{word-split } w = (u, v)$
proof –
have $\text{ucast } ((\text{drop-bit } \text{LENGTH}('c) (\text{word-cat } u \ v))::'a \ \text{word}) = u \ \text{ucast } ((\text{word-cat } u \ v)::'a \ \text{word}) = v$
using *assms*
by (*auto simp: word-size bit-ucast-iff bit-drop-bit-eq bit-word-cat-iff intro: bit-eqI*)
then show *?thesis*
by (*simp add: assms(1) word-split-bin'*)
qed

lemma *horner-sum-uint-exp-Cons-eq*:
 $\langle \text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) (w \# \text{ws}) =$
 $\text{concat-bit } \text{LENGTH}('a) (\text{uint } w) (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \text{ws}) \rangle$
for $\text{ws} :: \langle 'a::\text{len word list} \rangle$
by (*simp add: bintr-uint concat-bit-eq push-bit-eq-mult*)

lemma *bit-horner-sum-uint-exp-iff*:
 $\langle \text{bit } (\text{horner-sum uint } (2 \wedge \text{LENGTH}('a)) \text{ws}) \ n \longleftrightarrow$
 $n \ \text{div} \ \text{LENGTH}('a) < \text{length } \text{ws} \wedge \text{bit } (\text{ws} \ ! \ (n \ \text{div} \ \text{LENGTH}('a))) \ (n \ \text{mod} \ \text{LENGTH}('a)) \rangle$
for $\text{ws} :: \langle 'a::\text{len word list} \rangle$
proof (*induction ws arbitrary: n*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons w ws*)
then show *?case*
by (*cases* $\langle n \geq \text{LENGTH}('a) \rangle$)
(simp-all only: horner-sum-uint-exp-Cons-eq, simp-all add: bit-concat-bit-iff
le-div-geq le-mod-geq bit-uint-iff Cons)
qed

107.27 Rotation

lemma *word-rotr-word-rotr-eq*: $\langle \text{word-rotr } m (\text{word-rotr } n \ w) = \text{word-rotr } (m + n) \ w \rangle$
by (*rule bit-word-eqI*) (*simp add: bit-word-rotr-iff ac-simps mod-add-right-eq*)

lemma *word-rot-lem*: $\llbracket l + k = d + k \ \text{mod} \ l; n < l \rrbracket \implies ((d + n) \ \text{mod} \ l) = n$ **for** $l::\text{nat}$
by (*metis (no-types, lifting) add.commute add.right-neutral add-diff-cancel-left' mod-if mod-mult-div-eq mod-mult-self2 mod-self*)

lemma *word-rot-rl* [*simp*]: $\langle \text{word-rotl } k (\text{word-rotr } k \ v) = v \rangle$
proof (*rule bit-word-eqI*)
show $\text{bit } (\text{word-rotl } k (\text{word-rotr } k \ v)) \ n = \text{bit } v \ n$ **if** $n < \text{LENGTH}('a)$ **for** n

using that
 by (auto simp: word-rot-lem word-rotl-eq-word-rotr word-rotr-word-rotr-eq bit-word-rotr-iff
 algebra-simps split: nat-diff-split)
 qed

lemma word-rot-lr [simp]: $\langle \text{word-rotr } k (\text{word-rotl } k v) = v \rangle$
proof (rule bit-word-eqI)
 show bit (word-rotr k (word-rotl k v)) n = bit v n if $n < \text{LENGTH}('a)$ for n
 using that
 by (auto simp: word-rot-lem word-rotl-eq-word-rotr word-rotr-word-rotr-eq bit-word-rotr-iff
 algebra-simps split: nat-diff-split)
 qed

lemma word-rot-gal:
 $\langle \text{word-rotr } n v = w \longleftrightarrow \text{word-rotl } n w = v \rangle$
 by auto

lemma word-rot-gal':
 $\langle w = \text{word-rotr } n v \longleftrightarrow v = \text{word-rotl } n w \rangle$
 by auto

lemma word-reverse-word-rotl:
 $\langle \text{word-reverse } (\text{word-rotl } n w) = \text{word-rotr } n (\text{word-reverse } w) \rangle$ (is $\langle ?lhs = ?rhs \rangle$)
proof (rule bit-word-eqI)
 fix m
 assume $\langle m < \text{LENGTH}('a) \rangle$
 then have $\langle \text{int } (\text{LENGTH}('a) - \text{Suc } ((m + n) \bmod \text{LENGTH}('a))) =$
 $\text{int } ((\text{LENGTH}('a) + \text{LENGTH}('a) - \text{Suc } (m + n \bmod \text{LENGTH}('a))) \bmod$
 $\text{LENGTH}('a)) \rangle$
 unfolding of-nat-diff of-nat-mod
 apply (simp add: Suc-le-eq add-less-le-mono of-nat-mod algebra-simps)
 apply (simp only: mod-diff-left-eq [symmetric, of $\langle \text{int } \text{LENGTH}('a) * 2 \rangle$]
 mod-mult-self1-is-0 diff-0 minus-mod-int-eq)
 apply (simp add: mod-simps)
 done
 then have $\langle \text{LENGTH}('a) - \text{Suc } ((m + n) \bmod \text{LENGTH}('a)) =$
 $(\text{LENGTH}('a) + \text{LENGTH}('a) - \text{Suc } (m + n \bmod \text{LENGTH}('a))) \bmod$
 $\text{LENGTH}('a) \rangle$
 by simp
 with $\langle m < \text{LENGTH}('a) \rangle$ show $\langle \text{bit } ?lhs m \longleftrightarrow \text{bit } ?rhs m \rangle$
 by (simp add: bit-simps)
 qed

lemma word-reverse-word-rotr:
 $\langle \text{word-reverse } (\text{word-rotr } n w) = \text{word-rotl } n (\text{word-reverse } w) \rangle$
 by (rule word-eq-reverseI) (simp add: word-reverse-word-rotl)

lemma word-rotl-rev:
 $\langle \text{word-rotl } n w = \text{word-reverse } (\text{word-rotr } n (\text{word-reverse } w)) \rangle$

```

  by (simp add: word-reverse-word-rotr)

lemma word-rotr-rev:
  ⟨word-rotr n w = word-reverse (word-rotr n (word-reverse w))⟩
  by (simp add: word-reverse-word-rotr)

lemma word-roti-0 [simp]: word-roti 0 w = w
  by transfer simp

lemma word-roti-add: word-roti (m + n) w = word-roti m (word-roti n w)
  by (rule bit-word-eqI)
  (simp add: bit-word-roti-iff nat-less-iff mod-simps ac-simps)

lemma word-roti-conv-mod':
  word-roti n w = word-roti (n mod int (size w)) w
  by transfer simp

lemmas word-roti-conv-mod = word-roti-conv-mod' [unfolded word-size]

end

107.27.1 "Word rotation commutes with bit-wise operations

locale word-rotate
begin

context
  includes bit-operations-syntax
begin

lemma word-rot-logs:
  word-rotr n (NOT v) = NOT (word-rotr n v)
  word-rotr n (NOT v) = NOT (word-rotr n v)
  word-rotr n (x AND y) = word-rotr n x AND word-rotr n y
  word-rotr n (x AND y) = word-rotr n x AND word-rotr n y
  word-rotr n (x OR y) = word-rotr n x OR word-rotr n y
  word-rotr n (x OR y) = word-rotr n x OR word-rotr n y
  word-rotr n (x XOR y) = word-rotr n x XOR word-rotr n y
  word-rotr n (x XOR y) = word-rotr n x XOR word-rotr n y
  by (rule bit-word-eqI, auto simp: bit-word-rotr-iff bit-and-iff bit-or-iff
    bit-xor-iff bit-not-iff algebra-simps not-le)+

end

end

lemmas word-rot-logs = word-rotate.word-rot-logs

lemma word-rotx-0 [simp]: word-rotx i 0 = 0 ∧ word-rotl i 0 = 0

```

by *transfer simp-all*

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*
by *transfer simp*

declare *word-roti-eq-word-rotr-word-rotl* [*simp*]

107.28 Maximum machine word

context

includes *bit-operations-syntax*

begin

lemma *word-int-cases*:

fixes $x :: 'a::\text{len word}$

obtains n **where** $x = \text{word-of-int } n$ **and** $0 \leq n$ **and** $n < 2^{\text{LENGTH}('a)}$

by (*rule that [of <uint x>] simp-all*)

lemma *word-nat-cases* [*cases type: word*]:

fixes $x :: 'a::\text{len word}$

obtains n **where** $x = \text{of-nat } n$ **and** $n < 2^{\text{LENGTH}('a)}$

by (*rule that [of <unat x>] simp-all*)

lemma *max-word-max* [*intro!*]:

$\langle n \leq -1 \rangle$ **for** $n :: \langle 'a::\text{len word} \rangle$

by (*fact word-order.extremum*)

lemma *word-of-int-2p-len*: $\text{word-of-int } (2^{\text{LENGTH}('a)}) = (0 :: 'a::\text{len word})$

by *simp*

lemma *word-pow-0*: $(2 :: 'a::\text{len word})^{\text{LENGTH}('a)} = 0$

by (*fact word-exp-length-eq-0*)

lemma *max-word-wrap*:

$\langle x + 1 = 0 \implies x = -1 \rangle$ **for** $x :: \langle 'a::\text{len word} \rangle$

by (*simp add: eq-neg-iff-add-eq-0*)

lemma *word-and-max*:

$\langle x \text{ AND } -1 = x \rangle$ **for** $x :: \langle 'a::\text{len word} \rangle$

by (*fact word-log-esimps*)

lemma *word-or-max*:

$\langle x \text{ OR } -1 = -1 \rangle$ **for** $x :: \langle 'a::\text{len word} \rangle$

by (*fact word-log-esimps*)

lemma *word-ao-dist2*: $x \text{ AND } (y \text{ OR } z) = x \text{ AND } y \text{ OR } x \text{ AND } z$

for $x y z :: 'a::\text{len word}$

by (*fact bit.conj-disj-distrib*)

lemma *word-oa-dist2*: $x \text{ OR } y \text{ AND } z = (x \text{ OR } y) \text{ AND } (x \text{ OR } z)$
for $x \ y \ z :: 'a::\text{len word}$
by (*fact bit.disj-conj-distrib*)

lemma *word-and-not [simp]*: $x \text{ AND NOT } x = 0$
for $x :: 'a::\text{len word}$
by (*fact bit.conj-cancel-right*)

lemma *word-or-not [simp]*:
 $\langle x \text{ OR NOT } x = -1 \rangle$ **for** $x :: \langle 'a::\text{len word} \rangle$
by (*fact bit.disj-cancel-right*)

lemma *word-xor-and-or*: $x \text{ XOR } y = x \text{ AND NOT } y \text{ OR NOT } x \text{ AND } y$
for $x \ y :: 'a::\text{len word}$
by (*fact bit.xor-def*)

lemma *uint-lt-0 [simp]*: $\text{uint } x < 0 = \text{False}$
by (*simp add: linorder-not-less*)

lemma *word-less-1 [simp]*: $x < 1 \longleftrightarrow x = 0$
for $x :: 'a::\text{len word}$
by (*simp add: word-less-nat-alt unat-0-iff*)

lemma *uint-plus-if-size*:
 $\text{uint } (x + y) =$
 (*if* $\text{uint } x + \text{uint } y < 2^{\text{size } x}$
 then $\text{uint } x + \text{uint } y$
 else $\text{uint } x + \text{uint } y - 2^{\text{size } x}$)
by (*simp add: take-bit-eq-mod word-size uint-word-of-int-eq uint-plus-if'*)

lemma *unat-plus-if-size*:
 $\text{unat } (x + y) =$
 (*if* $\text{unat } x + \text{unat } y < 2^{\text{size } x}$
 then $\text{unat } x + \text{unat } y$
 else $\text{unat } x + \text{unat } y - 2^{\text{size } x}$)
for $x \ y :: 'a::\text{len word}$
by (*simp add: size-word.rep-eq unat-arith-simps*)

lemma *word-neq-0-conv*: $w \neq 0 \longleftrightarrow 0 < w$
for $w :: 'a::\text{len word}$
by (*fact word-coorder.not-eq-extremum*)

lemma *max-lt*: $\text{unat } (\max a \ b \ \text{div } c) = \text{unat } (\max a \ b) \ \text{div } \text{unat } c$
for $c :: 'a::\text{len word}$
by (*fact unat-div*)

lemma *uint-sub-if-size*:
 $\text{uint } (x - y) =$
 (*if* $\text{uint } y \leq \text{uint } x$


```

    then uint x - uint y
    else uint x - uint y + 2size x
  by (simp add: size-word.rep-eq uint-sub-if')

```

```

lemma unat-sub:
  ⟨unat (a - b) = unat a - unat b⟩
  if ⟨b ≤ a⟩
  by (meson that unat-sub-if-size word-le-nat-alt)

```

```

lemmas word-less-sub1-numberof [simp] = word-less-sub1 [of numeral w] for w
lemmas word-le-sub1-numberof [simp] = word-le-sub1 [of numeral w] for w

```

```

lemma word-of-int-minus: word-of-int (2LENGTH('a) - i) = (word-of-int (-i)::'a::len
word)
  by simp

```

```

lemma word-of-int-inj:
  ⟨(word-of-int x :: 'a::len word) = word-of-int y ⟷ x = y⟩
  if ⟨0 ≤ x ∧ x < 2LENGTH('a)⟩ ⟨0 ≤ y ∧ y < 2LENGTH('a)⟩
  using that by (transfer fixing: x y) (simp add: take-bit-int-eq-self)

```

```

lemma word-le-less-eq: x ≤ y ⟷ x = y ∨ x < y
  for x y :: 'z::len word
  by (auto simp: order-class.le-less)

```

```

lemma mod-plus-cong:
  fixes b b' :: int
  assumes 1: b = b'
  and 2: x mod b' = x' mod b'
  and 3: y mod b' = y' mod b'
  and 4: x' + y' = z'
  shows (x + y) mod b = z' mod b'
proof -
  from 1 2[symmetric] 3[symmetric]
  have (x + y) mod b = (x' mod b' + y' mod b') mod b'
    by (simp add: mod-add-eq)
  also have ... = (x' + y') mod b'
    by (simp add: mod-add-eq)
  finally show ?thesis
    by (simp add: 4)
qed

```

```

lemma mod-minus-cong:
  fixes b b' :: int
  assumes b = b'
  and x mod b' = x' mod b'
  and y mod b' = y' mod b'
  and x' - y' = z'
  shows (x - y) mod b = z' mod b'

```

```

using assms [symmetric] by (auto intro: mod-diff-cong)

lemma word-induct-less [case-names zero less]:
   $\langle P \, m \rangle$  if zero:  $\langle P \, 0 \rangle$  and less:  $\langle \bigwedge n. n < m \implies P \, n \implies P \, (1 + n) \rangle$ 
  for  $m :: \langle 'a :: \text{len word} \rangle$ 
proof –
  define  $q$  where  $\langle q = \text{unat } m \rangle$ 
  with less have  $\langle \bigwedge n. n < \text{word-of-nat } q \implies P \, n \implies P \, (1 + n) \rangle$ 
  by simp
  then have  $\langle P \, (\text{word-of-nat } q :: 'a \text{ word}) \rangle$ 
  proof (induction q)
    case 0
    show ?case
    by (simp add: zero)
  next
    case (Suc q)
    show ?case
    proof (cases  $\langle 1 + \text{word-of-nat } q = (0 :: 'a \text{ word}) \rangle$ )
      case True
      then show ?thesis
      by (simp add: zero)
    next
      case False
      then have  $*$ :  $\langle \text{word-of-nat } q < (\text{word-of-nat } (\text{Suc } q) :: 'a \text{ word}) \rangle$ 
      by (simp add: unatSuc word-less-nat-alt)
      then have  $**$ :  $\langle n < (1 + \text{word-of-nat } q :: 'a \text{ word}) \longleftrightarrow n \leq (\text{word-of-nat } q$ 
       $:: 'a \text{ word}) \rangle$  for  $n$ 
      by (metis (no-types, lifting) add.commute inc-le le-less-trans not-less
      of-nat-Suc)
      have  $\langle P \, (\text{word-of-nat } q) \rangle$ 
      by (simp add: ** Suc.IH Suc.prems)
      with  $*$  have  $\langle P \, (1 + \text{word-of-nat } q) \rangle$ 
      by (rule Suc.prems)
      then show ?thesis
      by simp
    qed
  qed
  with  $\langle q = \text{unat } m \rangle$  show ?thesis
  by simp
qed

lemma word-induct:  $P \, 0 \implies (\bigwedge n. P \, n \implies P \, (1 + n)) \implies P \, m$ 
  for  $P :: 'a :: \text{len word} \Rightarrow \text{bool}$ 
  by (rule word-induct-less)

lemma word-induct2 [case-names zero suc, induct type]:  $P \, 0 \implies (\bigwedge n. 1 + n \neq$ 
   $0 \implies P \, n \implies P \, (1 + n)) \implies P \, n$ 
  for  $P :: 'b :: \text{len word} \Rightarrow \text{bool}$ 
  by (induction rule: word-induct-less; force)

```

107.29 Recursion combinator for words

definition $\text{word-rec} :: 'a \Rightarrow ('b::\text{len word} \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b \text{ word} \Rightarrow 'a$
where $\text{word-rec forZero forSuc } n = \text{rec-nat forZero (forSuc } \circ \text{ of-nat) (unat } n)$

lemma word-rec-0 [simp]: $\text{word-rec } z \ s \ 0 = z$
by (simp add: word-rec-def)

lemma word-rec-Suc [simp]: $1 + n \neq 0 \implies \text{word-rec } z \ s \ (1 + n) = s \ n \ (\text{word-rec } z \ s \ n)$
for $n :: 'a::\text{len word}$
by (simp add: unatSuc word-rec-def)

lemma word-rec-Pred : $n \neq 0 \implies \text{word-rec } z \ s \ n = s \ (n - 1) \ (\text{word-rec } z \ s \ (n - 1))$
by (metis add.commute diff-add-cancel word-rec-Suc)

lemma word-rec-in : $f \ (\text{word-rec } z \ (\lambda-. f) \ n) = \text{word-rec } (f \ z) \ (\lambda-. f) \ n$
by (induct n) simp-all

lemma word-rec-in2 : $f \ n \ (\text{word-rec } z \ f \ n) = \text{word-rec } (f \ 0 \ z) \ (f \circ (+) \ 1) \ n$
by (induct n) simp-all

lemma word-rec-twice :
 $m \leq n \implies \text{word-rec } z \ f \ n = \text{word-rec } (\text{word-rec } z \ f \ (n - m)) \ (f \circ (+) \ (n - m))$
 m

proof (induction n arbitrary: z f)

case zero

then show ?case

by (metis diff-0-right word-le-0-iff word-rec-0)

next

case (suc n z f)

show ?case

proof (cases $1 + (n - m) = 0$)

case True

then show ?thesis

by (simp add: add-diff-eq)

next

case False

then have eq: $1 + n - m = 1 + (n - m)$

by simp

with False **have** $m \leq n$

using inc-le linorder-not-le suc.premis word-le-minus-mono-left **by** fastforce

with False suc.hyps **show** ?thesis

using suc.IH [of $f \ 0 \ z \ f \circ (+) \ 1$]

by (simp add: word-rec-in2 eq add.assoc o-def)

qed

qed

lemma word-rec-id : $\text{word-rec } z \ (\lambda-. \text{id}) \ n = z$

```

by (induct n) auto

lemma word-rec-id-eq: ( $\bigwedge m. m < n \implies f\ m = id$ )  $\implies$  word-rec z f n = z
by (induction n) (auto simp: unatSuc unat-arith-simps(2))

lemma word-rec-max:
  assumes  $\forall m \geq n. m \neq -1 \longrightarrow f\ m = id$ 
  shows word-rec z f (-1) = word-rec z f n
proof -
  have  $\S: \bigwedge m. \llbracket m < -1 - n \rrbracket \implies (f \circ (+)\ n)\ m = id$ 
  using assms
  by (metis (mono-tags, lifting) add.commute add-diff-cancel-left' comp-apply
less-le olen-add-eqv plus-minus-no-overflow word-n1-ge)
  have word-rec z f (-1) = word-rec (word-rec z f (-1 - (-1 - n))) (f  $\circ$  (+) (-1 - (-1 - n))) (-1 - n)
  by (meson word-n1-ge word-rec-twice)
  also have  $\dots = \text{word-rec } z\ f\ n$ 
  by (metis (no-types, lifting)  $\S$  diff-add-cancel minus-diff-eq uminus-add-conv-diff
word-rec-id-eq)
  finally show ?thesis .
qed

end

```

107.30 Some more naive computations rules

```

lemma drop-bit-of-minus-1-eq [simp]:
   $\langle \text{drop-bit } n\ (-1 :: 'a::\text{len word}) = \text{mask } (\text{LENGTH('a)} - n) \rangle$ 
by (rule bit-word-eqI) (auto simp add: bit-simps)

context
includes bit-operations-syntax
begin

lemma word-cat-eq-push-bit-or:
   $\langle \text{word-cat } v\ w = (\text{push-bit } \text{LENGTH('b)}\ (\text{ucast } v)\ \text{OR } \text{ucast } w :: 'c::\text{len word}) \rangle$ 
for  $v :: \langle 'a::\text{len word} \rangle$  and  $w :: \langle 'b::\text{len word} \rangle$ 
by transfer (simp add: concat-bit-def ac-simps)

end

context semiring-bit-operations
begin

lemma of-nat-take-bit-numeral-eq [simp]:
   $\langle \text{of-nat } (\text{take-bit } m\ (\text{numeral } n)) = \text{take-bit } m\ (\text{numeral } n) \rangle$ 
by (simp add: of-nat-take-bit)

end

```

context *ring-bit-operations*

begin

lemma *signed-take-bit-of-int*:

$\langle \text{signed-take-bit } n \text{ (of-int } k) = \text{of-int (signed-take-bit } n \text{ } k) \rangle$

by (*rule bit-eqI*) (*simp add: bit-simps*)

lemma *of-int-signed-take-bit*:

$\langle \text{of-int (signed-take-bit } n \text{ } k) = \text{signed-take-bit } n \text{ (of-int } k) \rangle$

by (*simp add: signed-take-bit-of-int*)

lemma *of-int-take-bit-minus-numeral-eq* [*simp*]:

$\langle \text{of-int (take-bit } m \text{ (numeral } n)) = \text{take-bit } m \text{ (numeral } n) \rangle$

$\langle \text{of-int (take-bit } m \text{ (} - \text{ numeral } n)) = \text{take-bit } m \text{ (} - \text{ numeral } n) \rangle$

by (*simp-all add: of-int-take-bit*)

end

context

includes *bit-operations-syntax*

begin

lemma *concat-bit-numeral-of-one-1* [*simp*]:

$\langle \text{concat-bit (numeral } m) \text{ } 1 \text{ } l = 1 \text{ OR push-bit (numeral } m) \text{ } l \rangle$

by (*rule bit-eqI*) (*auto simp add: bit-simps*)

lemma *concat-bit-of-one-2* [*simp*]:

$\langle \text{concat-bit } n \text{ } k \text{ } 1 = \text{set-bit } n \text{ (take-bit } n \text{ } k) \rangle$

by (*rule bit-eqI*) (*auto simp add: bit-simps*)

lemma *concat-bit-numeral-of-minus-one-1* [*simp*]:

$\langle \text{concat-bit (numeral } m) \text{ (} - \text{ } 1) \text{ } l = \text{push-bit (numeral } m) \text{ } l \text{ OR mask (numeral } m) \rangle$

by (*rule bit-eqI*) (*auto simp add: bit-simps*)

lemma *concat-bit-numeral-of-minus-one-2* [*simp*]:

$\langle \text{concat-bit (numeral } m) \text{ } k \text{ (} - \text{ } 1) = \text{take-bit (numeral } m) \text{ } k \text{ OR NOT (mask (numeral } m))} \rangle$

by (*rule bit-eqI*) (*auto simp add: bit-simps*)

lemma *concat-bit-numeral* [*simp*]:

$\langle \text{concat-bit (numeral } m) \text{ (numeral } n) \text{ (numeral } q) = \text{take-bit (numeral } m) \text{ (numeral } n) \text{ OR push-bit (numeral } m) \text{ (numeral } q) \rangle$

$\langle \text{concat-bit (numeral } m) \text{ (} - \text{ numeral } n) \text{ (numeral } q) = \text{take-bit (numeral } m) \text{ (} - \text{ numeral } n) \text{ OR push-bit (numeral } m) \text{ (numeral } q) \rangle$

$\langle \text{concat-bit (numeral } m) \text{ (numeral } n) \text{ (} - \text{ numeral } q) = \text{take-bit (numeral } m) \text{ (numeral } n) \text{ OR push-bit (numeral } m) \text{ (} - \text{ numeral } q) \rangle$

$\langle \text{concat-bit (numeral } m) \text{ (} - \text{ numeral } n) \text{ (} - \text{ numeral } q) = \text{take-bit (numeral } m) \text{ (} - \text{ numeral } n) \text{ OR push-bit (numeral } m) \text{ (} - \text{ numeral } q) \rangle$

```
(- numeral n) OR push-bit (numeral m) (- numeral q)⟩
  by (fact concat-bit-def)+
```

```
end
```

```
lemma word-cat-0-left [simp]:
  ⟨word-cat 0 w = ucast w⟩
  by (simp add: word-cat-eq)
```

107.31 Executable intervals

```
instance word :: (len) ⟨{interval-top, interval-bot}⟩
  by standard
  (simp-all add: less-eq-dec-self-iff-eq inc-less-eq-self-iff-eq less-inc-imp-less-eq
  dec-less-imp-less-eq)
```

107.32 Tool support

```
ML-file ⟨Tools/smt-word.ML⟩
```

```
end
```

108 The Field of Integers mod 2

```
theory Z2
imports Main
begin
```

Note that in most cases *bool* is appropriate when a binary type is needed; the type provided here, for historical reasons named *bit*, is only needed if proper field operations are required.

```
typedef bit = ⟨UNIV :: bool set⟩ ..
```

```
instantiation bit :: zero-neq-one
begin
```

```
definition zero-bit :: bit
  where ⟨0 = Abs-bit False⟩
```

```
definition one-bit :: bit
  where ⟨1 = Abs-bit True⟩
```

```
instance
  by standard (simp add: zero-bit-def one-bit-def Abs-bit-inject)
```

```
end
```

```
free-constructors case-bit for ⟨0::bit⟩ | ⟨1::bit⟩
proof —
```

```

fix P :: bool
fix a :: bit
assume  $\langle a = 0 \implies P \rangle$  and  $\langle a = 1 \implies P \rangle$ 
then show P
  by (cases a) (auto simp add: zero-bit-def one-bit-def Abs-bit-inject)
qed simp

lemma bit-not-zero-iff [simp]:
   $\langle a \neq 0 \longleftrightarrow a = 1 \rangle$  for a :: bit
  by (cases a) simp-all

lemma bit-not-one-iff [simp]:
   $\langle a \neq 1 \longleftrightarrow a = 0 \rangle$  for a :: bit
  by (cases a) simp-all

instantiation bit :: semidom-modulo
begin

definition plus-bit ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$ 
  where  $\langle a + b = \text{Abs-bit} (\text{Rep-bit } a \neq \text{Rep-bit } b) \rangle$ 

definition minus-bit ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$ 
  where [simp]:  $\langle \text{minus-bit} = \text{plus} \rangle$ 

definition times-bit ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$ 
  where  $\langle a * b = \text{Abs-bit} (\text{Rep-bit } a \wedge \text{Rep-bit } b) \rangle$ 

definition divide-bit ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$ 
  where [simp]:  $\langle \text{divide-bit} = \text{times} \rangle$ 

definition modulo-bit ::  $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$ 
  where  $\langle a \bmod b = \text{Abs-bit} (\text{Rep-bit } a \wedge \neg \text{Rep-bit } b) \rangle$ 

instance
  by standard
  (auto simp flip: Rep-bit-inject
    simp add: zero-bit-def one-bit-def plus-bit-def times-bit-def modulo-bit-def Abs-bit-inverse
    Rep-bit-inverse)

end

lemma bit-2-eq-0 [simp]:
   $\langle 2 = (0::\text{bit}) \rangle$ 
  by (simp flip: one-add-one add: zero-bit-def plus-bit-def)

instance bit :: semiring-parity
  apply standard
  apply (auto simp flip: Rep-bit-inject simp add: modulo-bit-def Abs-bit-inverse
    Rep-bit-inverse)

```

```

    apply (auto simp add: zero-bit-def one-bit-def Abs-bit-inverse Rep-bit-inverse)
  done

lemma Abs-bit-eq-of-bool [code-abbrev]:
  ⟨Abs-bit = of-bool⟩
  by (simp add: fun-eq-iff zero-bit-def one-bit-def)

lemma Rep-bit-eq-odd:
  ⟨Rep-bit = odd⟩
proof -
  have ⟨¬ Rep-bit 0⟩
    by (simp only: zero-bit-def) (subst Abs-bit-inverse, auto)
  then show ?thesis
    by (auto simp flip: Rep-bit-inject simp add: fun-eq-iff)
qed

lemma Rep-bit-iff-odd [code-abbrev]:
  ⟨Rep-bit b ⟷ odd b⟩
  by (simp add: Rep-bit-eq-odd)

lemma Not-Rep-bit-iff-even [code-abbrev]:
  ⟨¬ Rep-bit b ⟷ even b⟩
  by (simp add: Rep-bit-eq-odd)

lemma Not-Not-Rep-bit [code-unfold]:
  ⟨¬ ¬ Rep-bit b ⟷ Rep-bit b⟩
  by simp

code-datatype ⟨0::bit⟩ ⟨1::bit⟩

lemma Abs-bit-code [code]:
  ⟨Abs-bit False = 0⟩
  ⟨Abs-bit True = 1⟩
  by (simp-all add: Abs-bit-eq-of-bool)

lemma Rep-bit-code [code]:
  ⟨Rep-bit 0 ⟷ False⟩
  ⟨Rep-bit 1 ⟷ True⟩
  by (simp-all add: Rep-bit-eq-odd)

context zero-neq-one
begin

abbreviation of-bit :: ⟨bit ⇒ 'a⟩
  where ⟨of-bit b ≡ of-bool (odd b)⟩

end

context

```


begin

qualified lemma *bit-eq-iff*:

$\langle a = b \longleftrightarrow (\text{even } a \longleftrightarrow \text{even } b) \rangle$ **for** $a \ b :: \text{bit}$
by (*cases a; cases b*) *simp-all*

end

lemma *modulo-bit-unfold* [*simp, code*]:

$\langle a \bmod b = \text{of-bool } (\text{odd } a \wedge \text{even } b) \rangle$ **for** $a \ b :: \text{bit}$
by (*simp add: modulo-bit-def Abs-bit-eq-of-bool Rep-bit-eq-odd*)

lemma *power-bit-unfold* [*simp*]:

$\langle a \wedge^n = \text{of-bool } (\text{odd } a \vee n = 0) \rangle$ **for** $a :: \text{bit}$
by (*cases a*) *simp-all*

instantiation *bit* :: *field*

begin

definition *uminus-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \rangle$

where [*simp*]: $\langle \text{uminus-bit} = \text{id} \rangle$

definition *inverse-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \rangle$

where [*simp*]: $\langle \text{inverse-bit} = \text{id} \rangle$

instance

apply *standard*

apply *simp-all*

apply (*simp only: Z2.bit-eq-iff even-add even-zero refl*)

done

end

instantiation *bit* :: *semiring-bits*

begin

definition *bit-bit* :: $\langle \text{bit} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$

where [*simp*]: $\langle \text{bit-bit } b \ n \longleftrightarrow \text{odd } b \wedge n = 0 \rangle$

instance

by *standard*

(*auto intro: Abs-bit-induct simp add: Abs-bit-eq-of-bool*)

end

instantiation *bit* :: *ring-bit-operations*

begin

context

includes *bit-operations-syntax*
begin

definition *not-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{NOT } b = \text{of-bool } (\text{even } b) \rangle$ **for** $b :: \text{bit}$

definition *and-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \text{ AND } c = \text{of-bool } (\text{odd } b \wedge \text{odd } c) \rangle$ **for** $b \ c :: \text{bit}$

definition *or-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \text{ OR } c = \text{of-bool } (\text{odd } b \vee \text{odd } c) \rangle$ **for** $b \ c :: \text{bit}$

definition *xor-bit* :: $\langle \text{bit} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle b \text{ XOR } c = \text{of-bool } (\text{odd } b \neq \text{odd } c) \rangle$ **for** $b \ c :: \text{bit}$

definition *mask-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{mask } n = (\text{of-bool } (n > 0)) :: \text{bit} \rangle$

definition *set-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{set-bit } n \ b = \text{of-bool } (n = 0 \vee \text{odd } b) \rangle$ **for** $b :: \text{bit}$

definition *unset-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{unset-bit } n \ b = \text{of-bool } (n > 0 \wedge \text{odd } b) \rangle$ **for** $b :: \text{bit}$

definition *flip-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{flip-bit } n \ b = \text{of-bool } ((n = 0) \neq \text{odd } b) \rangle$ **for** $b :: \text{bit}$

definition *push-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{push-bit } n \ b = \text{of-bool } (\text{odd } b \wedge n = 0) \rangle$ **for** $b :: \text{bit}$

definition *drop-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{drop-bit } n \ b = \text{of-bool } (\text{odd } b \wedge n = 0) \rangle$ **for** $b :: \text{bit}$

definition *take-bit-bit* :: $\langle \text{nat} \Rightarrow \text{bit} \Rightarrow \text{bit} \rangle$
where [*simp*]: $\langle \text{take-bit } n \ b = \text{of-bool } (\text{odd } b \wedge n > 0) \rangle$ **for** $b :: \text{bit}$

end

instance
by *standard auto*

end

lemma *add-bit-eq-xor* [*simp*, *code*]:
 $\langle (+) = (\text{Bit-Operations.xor} :: \text{bit} \Rightarrow -) \rangle$
by (*auto simp add: fun-eq-iff*)

lemma *mult-bit-eq-and* [*simp*, *code*]:
 $\langle (*) = (\text{Bit-Operations.and} :: \text{bit} \Rightarrow -) \rangle$

by (*simp add: fun-eq-iff*)

lemma *bit-numeral-even* [*simp*]:
 $\langle \text{numeral } (\text{Num.Bit0 } n) = (0 :: \text{bit}) \rangle$
by (*simp only: Z2.bit-eq-iff even-numeral*) *simp*

lemma *bit-numeral-odd* [*simp*]:
 $\langle \text{numeral } (\text{Num.Bit1 } n) = (1 :: \text{bit}) \rangle$
by (*simp only: Z2.bit-eq-iff odd-numeral*) *simp*

end

109 Pointwise order on product types

theory *Product-Order*
imports *Product-Plus*
begin

109.1 Pointwise ordering

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
 $x \leq y \longleftrightarrow \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$

definition
 $(x :: 'a \times 'b) < y \longleftrightarrow x \leq y \wedge \neg y \leq x$

instance ..

end

lemma *fst-mono*: $x \leq y \implies \text{fst } x \leq \text{fst } y$
unfolding *less-eq-prod-def* **by** *simp*

lemma *snd-mono*: $x \leq y \implies \text{snd } x \leq \text{snd } y$
unfolding *less-eq-prod-def* **by** *simp*

lemma *Pair-mono*: $x \leq x' \implies y \leq y' \implies (x, y) \leq (x', y')$
unfolding *less-eq-prod-def* **by** *simp*

lemma *Pair-le* [*simp*]: $(a, b) \leq (c, d) \longleftrightarrow a \leq c \wedge b \leq d$
unfolding *less-eq-prod-def* **by** *simp*

lemma *atLeastAtMost-prod-eq*: $\{a..b\} = \{\text{fst } a.. \text{fst } b\} \times \{\text{snd } a.. \text{snd } b\}$
by (*auto simp: less-eq-prod-def*)

```

instance prod :: (preorder, preorder) preorder
proof
  fix x y z :: 'a × 'b
  show x < y  $\longleftrightarrow$  x ≤ y ∧ ¬ y ≤ x
    by (rule less-prod-def)
  show x ≤ x
    unfolding less-eq-prod-def
    by fast
  assume x ≤ y and y ≤ z thus x ≤ z
    unfolding less-eq-prod-def
    by (fast elim: order-trans)
qed

```

```

instance prod :: (order, order) order
  by standard auto

```

109.2 Binary infimum and supremum

```

instantiation prod :: (inf, inf) inf
begin

```

```

definition inf x y = (inf (fst x) (fst y), inf (snd x) (snd y))

```

```

lemma inf-Pair-Pair [simp]: inf (a, b) (c, d) = (inf a c, inf b d)
  unfolding inf-prod-def by simp

```

```

lemma fst-inf [simp]: fst (inf x y) = inf (fst x) (fst y)
  unfolding inf-prod-def by simp

```

```

lemma snd-inf [simp]: snd (inf x y) = inf (snd x) (snd y)
  unfolding inf-prod-def by simp

```

```

instance ..

```

```

end

```

```

instance prod :: (semilattice-inf, semilattice-inf) semilattice-inf
  by standard auto

```

```

instantiation prod :: (sup, sup) sup
begin

```

```

definition
  sup x y = (sup (fst x) (fst y), sup (snd x) (snd y))

```

```

lemma sup-Pair-Pair [simp]: sup (a, b) (c, d) = (sup a c, sup b d)
  unfolding sup-prod-def by simp

```

lemma *fst-sup* [*simp*]: $\text{fst } (\text{sup } x \ y) = \text{sup } (\text{fst } x) (\text{fst } y)$
unfolding *sup-prod-def* **by** *simp*

lemma *snd-sup* [*simp*]: $\text{snd } (\text{sup } x \ y) = \text{sup } (\text{snd } x) (\text{snd } y)$
unfolding *sup-prod-def* **by** *simp*

instance ..

end

instance *prod* :: (*semilattice-sup*, *semilattice-sup*) *semilattice-sup*
by *standard auto*

instance *prod* :: (*lattice*, *lattice*) *lattice* ..

instance *prod* :: (*distrib-lattice*, *distrib-lattice*) *distrib-lattice*
by *standard (auto simp add: sup-inf-distrib1)*

109.3 Top and bottom elements

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ..

end

lemma *fst-top* [*simp*]: $\text{fst } \text{top} = \text{top}$
unfolding *top-prod-def* **by** *simp*

lemma *snd-top* [*simp*]: $\text{snd } \text{top} = \text{top}$
unfolding *top-prod-def* **by** *simp*

lemma *Pair-top-top*: $(\text{top}, \text{top}) = \text{top}$
unfolding *top-prod-def* **by** *simp*

instance *prod* :: (*order-top*, *order-top*) *order-top*
by *standard (auto simp add: top-prod-def)*

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ..

end

lemma *fst-bot* [*simp*]: *fst bot = bot*
unfolding *bot-prod-def* **by** *simp*

lemma *snd-bot* [*simp*]: *snd bot = bot*
unfolding *bot-prod-def* **by** *simp*

lemma *Pair-bot-bot*: (*bot, bot*) = *bot*
unfolding *bot-prod-def* **by** *simp*

instance *prod* :: (*order-bot, order-bot*) *order-bot*
by *standard* (*auto simp add: bot-prod-def*)

instance *prod* :: (*bounded-lattice, bounded-lattice*) *bounded-lattice* ..

instance *prod* :: (*boolean-algebra, boolean-algebra*) *boolean-algebra*
by *standard* (*auto simp add: prod-eqI diff-eq*)

109.4 Complete lattice operations

instantiation *prod* :: (*Inf, Inf*) *Inf*
begin

definition *Inf A* = (*INF x ∈ A. fst x, INF x ∈ A. snd x*)

instance ..

end

instantiation *prod* :: (*Sup, Sup*) *Sup*
begin

definition *Sup A* = (*SUP x ∈ A. fst x, SUP x ∈ A. snd x*)

instance ..

end

instance *prod* :: (*conditionally-complete-lattice, conditionally-complete-lattice*)
conditionally-complete-lattice
by *standard* (*force simp: less-eq-prod-def Inf-prod-def Sup-prod-def bdd-below-def*
bdd-above-def
intro!: cInf-lower cSup-upper cInf-greatest cSup-least) +

instance *prod* :: (*complete-lattice, complete-lattice*) *complete-lattice*
by *standard* (*simp-all add: less-eq-prod-def Inf-prod-def Sup-prod-def*
INF-lower SUP-upper le-INF-iff SUP-le-iff bot-prod-def top-prod-def)

lemma *fst-Inf*: $\text{fst } (\text{Inf } A) = (\text{INF } x \in A. \text{fst } x)$
by (*simp add: Inf-prod-def*)

lemma *fst-INF*: $\text{fst } (\text{INF } x \in A. f x) = (\text{INF } x \in A. \text{fst } (f x))$
by (*simp add: fst-Inf image-image*)

lemma *fst-Sup*: $\text{fst } (\text{Sup } A) = (\text{SUP } x \in A. \text{fst } x)$
by (*simp add: Sup-prod-def*)

lemma *fst-SUP*: $\text{fst } (\text{SUP } x \in A. f x) = (\text{SUP } x \in A. \text{fst } (f x))$
by (*simp add: fst-Sup image-image*)

lemma *snd-Inf*: $\text{snd } (\text{Inf } A) = (\text{INF } x \in A. \text{snd } x)$
by (*simp add: Inf-prod-def*)

lemma *snd-INF*: $\text{snd } (\text{INF } x \in A. f x) = (\text{INF } x \in A. \text{snd } (f x))$
by (*simp add: snd-Inf image-image*)

lemma *snd-Sup*: $\text{snd } (\text{Sup } A) = (\text{SUP } x \in A. \text{snd } x)$
by (*simp add: Sup-prod-def*)

lemma *snd-SUP*: $\text{snd } (\text{SUP } x \in A. f x) = (\text{SUP } x \in A. \text{snd } (f x))$
by (*simp add: snd-Sup image-image*)

lemma *INF-Pair*: $(\text{INF } x \in A. (f x, g x)) = (\text{INF } x \in A. f x, \text{INF } x \in A. g x)$
by (*simp add: Inf-prod-def image-image*)

lemma *SUP-Pair*: $(\text{SUP } x \in A. (f x, g x)) = (\text{SUP } x \in A. f x, \text{SUP } x \in A. g x)$
by (*simp add: Sup-prod-def image-image*)

Alternative formulations for set infima and suprema over the product of two complete lattices:

lemma *INF-prod-alt-def*:
 $\text{Inf } (f \text{ ‘ } A) = (\text{Inf } ((\text{fst} \circ f) \text{ ‘ } A), \text{Inf } ((\text{snd} \circ f) \text{ ‘ } A))$
by (*simp add: Inf-prod-def image-image*)

lemma *SUP-prod-alt-def*:
 $\text{Sup } (f \text{ ‘ } A) = (\text{Sup } ((\text{fst} \circ f) \text{ ‘ } A), \text{Sup } ((\text{snd} \circ f) \text{ ‘ } A))$
by (*simp add: Sup-prod-def image-image*)

109.5 Complete distributive lattices

instance *prod* :: (*complete-distrib-lattice*, *complete-distrib-lattice*) *complete-distrib-lattice*

proof

fix *A*::(*a*×*b*) *set set*
show $\text{Inf } (\text{Sup } \text{ ‘ } A) \leq \text{Sup } (\text{Inf } \text{ ‘ } \{f \text{ ‘ } A \mid f. \forall Y \in A. f Y \in Y\})$
by (*simp add: Inf-prod-def Sup-prod-def INF-SUP-set image-image*)
qed

109.6 Bekic’s Theorem

Simultaneous fixed points over pairs can be written in terms of separate fixed points. Transliterated from HOLCF.Fix by Peter Gammie

lemma *lfp-prod*:

```

fixes  $F :: 'a::\text{complete-lattice} \times 'b::\text{complete-lattice} \Rightarrow 'a \times 'b$ 
assumes mono F
shows  $\text{lfp } F = (\text{lfp } (\lambda x. \text{fst } (F (x, \text{lfp } (\lambda y. \text{snd } (F (x, y)))))),$ 
       $(\text{lfp } (\lambda y. \text{snd } (F (\text{lfp } (\lambda x. \text{fst } (F (x, \text{lfp } (\lambda y. \text{snd } (F (x, y)))))), y))))))$ 
       $(\text{is } \text{lfp } F = (?x, ?y))$ 
proof(rule lfp-eqI[OF assms])
  have  $1: \text{fst } (F (?x, ?y)) = ?x$ 
    by (rule trans [symmetric, OF lfp-unfold])
    (blast intro!: monoI monoD[OF assms(1)] fst-mono snd-mono Pair-mono lfp-mono) +
  have  $2: \text{snd } (F (?x, ?y)) = ?y$ 
    by (rule trans [symmetric, OF lfp-unfold])
    (blast intro!: monoI monoD[OF assms(1)] fst-mono snd-mono Pair-mono lfp-mono) +
  from  $1\ 2$  show  $F (?x, ?y) = (?x, ?y)$  by (simp add: prod-eq-iff)
next
  fix  $z$  assume  $F\text{-}z: F\ z = z$ 
  obtain  $x\ y$  where  $z = (x, y)$  by (rule prod.exhaust)
  from  $F\text{-}z\ z$  have  $F\text{-}x: \text{fst } (F (x, y)) = x$  by simp
  from  $F\text{-}z\ z$  have  $F\text{-}y: \text{snd } (F (x, y)) = y$  by simp
  let  $?y1 = \text{lfp } (\lambda y. \text{snd } (F (x, y)))$ 
  have  $?y1 \leq y$  by (rule lfp-lowerbound, simp add: F-y)
  hence  $\text{fst } (F (x, ?y1)) \leq \text{fst } (F (x, y))$ 
    by (simp add: assms fst-mono monoD)
  hence  $\text{fst } (F (x, ?y1)) \leq x$  using  $F\text{-}x$  by simp
  hence  $1: ?x \leq x$  by (simp add: lfp-lowerbound)
  hence  $\text{snd } (F (?x, y)) \leq \text{snd } (F (x, y))$ 
    by (simp add: assms snd-mono monoD)
  hence  $\text{snd } (F (?x, y)) \leq y$  using  $F\text{-}y$  by simp
  hence  $2: ?y \leq y$  by (simp add: lfp-lowerbound)
  show  $(?x, ?y) \leq z$  using  $z\ 1\ 2$  by simp
qed

```

lemma *gfp-prod*:

```

fixes  $F :: 'a::\text{complete-lattice} \times 'b::\text{complete-lattice} \Rightarrow 'a \times 'b$ 
assumes mono F
shows  $\text{gfp } F = (\text{gfp } (\lambda x. \text{fst } (F (x, \text{gfp } (\lambda y. \text{snd } (F (x, y)))))),$ 
       $(\text{gfp } (\lambda y. \text{snd } (F (\text{gfp } (\lambda x. \text{fst } (F (x, \text{gfp } (\lambda y. \text{snd } (F (x, y)))))), y))))))$ 
       $(\text{is } \text{gfp } F = (?x, ?y))$ 
proof(rule gfp-eqI[OF assms])
  have  $1: \text{fst } (F (?x, ?y)) = ?x$ 
    by (rule trans [symmetric, OF gfp-unfold])
    (blast intro!: monoI monoD[OF assms(1)] fst-mono snd-mono Pair-mono gfp-mono) +

```



```

have 2: snd (F (?x, ?y)) = ?y
by (rule trans [symmetric, OF gfp-unfold])
      (blast intro!: monoI monoD[OF assms(1)] fst-mono snd-mono Pair-mono
gfp-mono)+
from 1 2 show F (?x, ?y) = (?x, ?y) by (simp add: prod-eq-iff)
next
fix z assume F-z: F z = z
obtain x y where z: z = (x, y) by (rule prod.exhaust)
from F-z z have F-x: fst (F (x, y)) = x by simp
from F-z z have F-y: snd (F (x, y)) = y by simp
let ?y1 = gfp (λy. snd (F (x, y)))
have y ≤ ?y1 by (rule gfp-upperbound, simp add: F-y)
hence fst (F (x, y)) ≤ fst (F (x, ?y1))
      by (simp add: assms fst-mono monoD)
hence x ≤ fst (F (x, ?y1)) using F-x by simp
hence 1: x ≤ ?x by (simp add: gfp-upperbound)
hence snd (F (x, y)) ≤ snd (F (?x, y))
      by (simp add: assms snd-mono monoD)
hence y ≤ snd (F (?x, y)) using F-y by simp
hence 2: y ≤ ?y by (simp add: gfp-upperbound)
show z ≤ (?x, ?y) using z 1 2 by simp
qed

end

```

110 Finite Lattices

```

theory Finite-Lattice
imports Product-Order
begin

```

110.1 Finite Complete Lattices

A non-empty finite lattice is a complete lattice. Since types are never empty in Isabelle/HOL, a type of classes *finite* and *lattice* should also have class *complete-lattice*. A type class is defined that extends classes *finite* and *lattice* with the operators *bot*, *top*, *Inf*, and *Sup*, along with assumptions that define these operators in terms of the ones of classes *finite* and *lattice*. The resulting class is a subclass of *complete-lattice*.

```

class finite-lattice-complete = finite + lattice + bot + top + Inf + Sup +
  assumes bot-def: bot = Inf-fin UNIV
  assumes top-def: top = Sup-fin UNIV
  assumes Inf-def: Inf A = Finite-Set.fold inf top A
  assumes Sup-def: Sup A = Finite-Set.fold sup bot A

```

The definitional assumptions on the operators *bot* and *top* of class *finite-lattice-complete* ensure that they yield bottom and top.

```

lemma finite-lattice-complete-bot-least: (bot::'a::finite-lattice-complete) ≤ x

```

```

by (auto simp: bot-def intro: Inf-fin.coboundedI)

instance finite-lattice-complete  $\subseteq$  order-bot
  by standard (auto simp: finite-lattice-complete-bot-least)

lemma finite-lattice-complete-top-greatest:  $(top :: 'a :: \text{finite-lattice-complete}) \geq x$ 
  by (auto simp: top-def Sup-fin.coboundedI)

instance finite-lattice-complete  $\subseteq$  order-top
  by standard (auto simp: finite-lattice-complete-top-greatest)

instance finite-lattice-complete  $\subseteq$  bounded-lattice ..

  The definitional assumptions on the operators Inf and Sup of class finite-lattice-complete ensure that they yield infimum and supremum.

lemma finite-lattice-complete-Inf-empty:  $\text{Inf } \{\} = (top :: 'a :: \text{finite-lattice-complete})$ 
  by (simp add: Inf-def)

lemma finite-lattice-complete-Sup-empty:  $\text{Sup } \{\} = (bot :: 'a :: \text{finite-lattice-complete})$ 
  by (simp add: Sup-def)

lemma finite-lattice-complete-Inf-insert:
  fixes  $A :: 'a :: \text{finite-lattice-complete}$  set
  shows  $\text{Inf } (\text{insert } x \ A) = \text{inf } x \ (\text{Inf } A)$ 
proof –
  interpret  $\text{comp-fun-idem inf} :: 'a \Rightarrow -$ 
  by (fact comp-fun-idem-inf)
  show ?thesis by (simp add: Inf-def)
qed

lemma finite-lattice-complete-Sup-insert:
  fixes  $A :: 'a :: \text{finite-lattice-complete}$  set
  shows  $\text{Sup } (\text{insert } x \ A) = \text{sup } x \ (\text{Sup } A)$ 
proof –
  interpret  $\text{comp-fun-idem sup} :: 'a \Rightarrow -$ 
  by (fact comp-fun-idem-sup)
  show ?thesis by (simp add: Sup-def)
qed

lemma finite-lattice-complete-Inf-lower:
   $(x :: 'a :: \text{finite-lattice-complete}) \in A \implies \text{Inf } A \leq x$ 
  using finite [of A]
  by (induct A) (auto simp add: finite-lattice-complete-Inf-insert intro: le-infI2)

lemma finite-lattice-complete-Inf-greatest:
   $\forall x :: 'a :: \text{finite-lattice-complete} \in A. z \leq x \implies z \leq \text{Inf } A$ 
  using finite [of A]
  by (induct A) (auto simp add: finite-lattice-complete-Inf-empty finite-lattice-complete-Inf-insert)

```

lemma *finite-lattice-complete-Sup-upper*:
 $(x :: 'a :: \text{finite-lattice-complete}) \in A \implies \text{Sup } A \geq x$
using *finite* [of *A*]
by (*induct A*) (*auto simp add: finite-lattice-complete-Sup-insert intro: le-supI2*)

lemma *finite-lattice-complete-Sup-least*:
 $\forall x :: 'a :: \text{finite-lattice-complete} \in A. z \geq x \implies z \geq \text{Sup } A$
using *finite* [of *A*]
by (*induct A*) (*auto simp add: finite-lattice-complete-Sup-empty finite-lattice-complete-Sup-insert*)

instance *finite-lattice-complete* \subseteq *complete-lattice*
proof

qed (*auto simp*:
finite-lattice-complete-Inf-lower
finite-lattice-complete-Inf-greatest
finite-lattice-complete-Sup-upper
finite-lattice-complete-Sup-least
finite-lattice-complete-Inf-empty
finite-lattice-complete-Sup-empty)

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
 $(\text{bot} :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete})) =$
 Inf-fin UNIV
by (*metis Inf-fin.coboundedI UNIV-I bot.extremum-uniqueI finite-UNIV*)

lemma *finite-top-prod*:
 $(\text{top} :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete})) =$
 Sup-fin UNIV
by (*metis Sup-fin.coboundedI UNIV-I top.extremum-uniqueI finite-UNIV*)

lemma *finite-Inf-prod*:
 $\text{Inf}(A :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete}) \text{ set}) =$
 $\text{Finite-Set.fold inf top } A$
by (*metis Inf-fold-inf finite*)

lemma *finite-Sup-prod*:
 $\text{Sup}(A :: ('a :: \text{finite-lattice-complete} \times 'b :: \text{finite-lattice-complete}) \text{ set}) =$
 $\text{Finite-Set.fold sup bot } A$
by (*metis Sup-fold-sup finite*)

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
by *standard* (*auto simp: finite-bot-prod finite-top-prod finite-Inf-prod finite-Sup-prod*)

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

lemma *finite-bot-fun*: $(\text{bot} :: ('a :: \text{finite} \Rightarrow 'b :: \text{finite-lattice-complete})) = \text{Inf-fin UNIV}$
by (*metis Inf-UNIV Inf-fin-Inf empty-not-UNIV finite*)

lemma *finite-top-fun*: (*top* :: ('a::finite \Rightarrow 'b::finite-lattice-complete)) = *Sup-fin UNIV*

by (*metis Sup-UNIV Sup-fin-Sup empty-not-UNIV finite*)

lemma *finite-Inf-fun*:

Inf (*A*::('a::finite \Rightarrow 'b::finite-lattice-complete) *set*) =

Finite-Set.fold inf top A

by (*metis Inf-fold-inf finite*)

lemma *finite-Sup-fun*:

Sup (*A*::('a::finite \Rightarrow 'b::finite-lattice-complete) *set*) =

Finite-Set.fold sup bot A

by (*metis Sup-fold-sup finite*)

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*

by *standard* (*auto simp: finite-bot-fun finite-top-fun finite-Inf-fun finite-Sup-fun*)

110.2 Finite Distributive Lattices

A finite distributive lattice is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

class *finite-distrib-lattice-complete* =

distrib-lattice + *finite-lattice-complete*

lemma *finite-distrib-lattice-complete-sup-Inf*:

sup (*x*::'a::finite-distrib-lattice-complete) (*Inf A*) = (*INF* *y*∈*A*. *sup x y*)

using *finite*

by (*induct A rule: finite-induct*) (*simp-all add: sup-inf-distrib1*)

lemma *finite-distrib-lattice-complete-inf-Sup*:

inf (*x*::'a::finite-distrib-lattice-complete) (*Sup A*) = (*SUP* *y*∈*A*. *inf x y*)

using *finite* [*of A*] **by** *induct* (*simp-all add: inf-sup-distrib1*)

context *finite-distrib-lattice-complete*

begin

subclass *finite-distrib-lattice*

proof –

show *class.finite-distrib-lattice Inf Sup inf* (\leq) ($<$) *sup bot top*

proof

show *bot* = *Inf UNIV*

unfolding *bot-def top-def Inf-def*

using *Inf-fin.eq-fold Inf-fin.insert inf.absorb2* **by** *force*

next

show *top* = *Sup UNIV*

unfolding *bot-def top-def Sup-def*

using *Sup-fin.eq-fold Sup-fin.insert* **by** *force*

next

show *Inf {}* = *Sup UNIV*

unfolding *Inf-def Sup-def bot-def top-def*

```

    using Sup-fin.eq-fold Sup-fin.insert by force
  next
  show Sup {} = Inf UNIV
    unfolding Inf-def Sup-def bot-def top-def
    using Inf-fin.eq-fold Inf-fin.insert inf.absorb2 by force
  next
  interpret comp-fun-idem-inf: comp-fun-idem inf
    by (fact comp-fun-idem-inf)
  show Inf (insert a A) = inf a (Inf A) for a A
    using comp-fun-idem-inf.fold-insert-idem Inf-def finite by simp
  next
  interpret comp-fun-idem-sup: comp-fun-idem sup
    by (fact comp-fun-idem-sup)
  show Sup (insert a A) = sup a (Sup A) for a A
    using comp-fun-idem-sup.fold-insert-idem Sup-def finite by simp
qed
qed
end

```

instance *finite-distrib-lattice-complete* \subseteq *complete-distrib-lattice* ..

The product of two finite distributive lattices is already a finite distributive lattice.

```

instance prod ::
  (finite-distrib-lattice-complete, finite-distrib-lattice-complete)
  finite-distrib-lattice-complete
..

```

Functions with a finite domain and with a finite distributive lattice as codomain already form a finite distributive lattice.

```

instance fun ::
  (finite, finite-distrib-lattice-complete) finite-distrib-lattice-complete
..

```

110.3 Linear Orders

A linear order is a distributive lattice. A type class is defined that extends class *linorder* with the operators *inf* and *sup*, along with assumptions that define these operators in terms of the ones of class *linorder*. The resulting class is a subclass of *distrib-lattice*.

```

class linorder-lattice = linorder + inf + sup +
  assumes inf-def: inf x y = (if x ≤ y then x else y)
  assumes sup-def: sup x y = (if x ≥ y then x else y)

```

The definitional assumptions on the operators *inf* and *sup* of class *linorder-lattice* ensure that they yield infimum and supremum and that they distribute over each other.

lemma *linorder-lattice-inf-le1*: *inf* (x::'a::linorder-lattice) y ≤ x

```

unfolding inf-def by (metis (full-types) linorder-linear)

lemma linorder-lattice-inf-le2:  $\inf (x::'a::\text{linorder-lattice})\ y \leq y$ 
unfolding inf-def by (metis (full-types) linorder-linear)

lemma linorder-lattice-inf-greatest:
 $(x::'a::\text{linorder-lattice}) \leq y \implies x \leq z \implies x \leq \inf y\ z$ 
unfolding inf-def by (metis (full-types))

lemma linorder-lattice-sup-ge1:  $\sup (x::'a::\text{linorder-lattice})\ y \geq x$ 
unfolding sup-def by (metis (full-types) linorder-linear)

lemma linorder-lattice-sup-ge2:  $\sup (x::'a::\text{linorder-lattice})\ y \geq y$ 
unfolding sup-def by (metis (full-types) linorder-linear)

lemma linorder-lattice-sup-least:
 $(x::'a::\text{linorder-lattice}) \geq y \implies x \geq z \implies x \geq \sup y\ z$ 
by (auto simp: sup-def)

lemma linorder-lattice-sup-inf-distrib1:
 $\sup (x::'a::\text{linorder-lattice})\ (\inf y\ z) = \inf (\sup x\ y)\ (\sup x\ z)$ 
by (auto simp: inf-def sup-def)

instance linorder-lattice  $\subseteq$  distrib-lattice
proof
qed (auto simp:
  linorder-lattice-inf-le1
  linorder-lattice-inf-le2
  linorder-lattice-inf-greatest
  linorder-lattice-sup-ge1
  linorder-lattice-sup-ge2
  linorder-lattice-sup-least
  linorder-lattice-sup-inf-distrib1)

```

110.4 Finite Linear Orders

A (non-empty) finite linear order is a complete linear order.

```
class finite-linorder-complete = linorder-lattice + finite-lattice-complete
```

```
instance finite-linorder-complete  $\subseteq$  complete-linorder ..
```

A (non-empty) finite linear order is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

```
instance finite-linorder-complete  $\subseteq$  finite-distrib-lattice-complete ..
```

```
end
```

111 Lexicographic order on lists

theory *List-Lexorder*

imports *Main*

begin

instantiation *list* :: (*ord*) *ord*

begin

definition

list-less-def: $xs < ys \longleftrightarrow (xs, ys) \in \text{lexord } \{(u, v). u < v\}$

definition

list-le-def: $(xs :: \text{list}) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

instance ..

end

instance *list* :: (*order*) *order*

proof

let ?*r* = $\{(u, v::'a). u < v\}$

have *tr*: *trans* ?*r*

using *trans-def* **by** *fastforce*

have §: *False*

if $(xs, ys) \in \text{lexord } ?r$ **for** *xs* *ys* :: '*a* *list*

proof –

have $(xs, xs) \in \text{lexord } ?r$

using *that* *transD* [*OF* *lexord-transI* [*OF* *tr*]] **by** *blast*

then show *False*

by (*meson case-prodD lexord-irreflexive less-irrefl mem-Collect-eq*)

qed

show $xs \leq xs$ **for** *xs* :: '*a* *list* **by** (*simp add: list-le-def*)

show $xs \leq zs$ **if** $xs \leq ys$ **and** $ys \leq zs$ **for** *xs* *ys* *zs* :: '*a* *list*

using *that* *transD* [*OF* *lexord-transI* [*OF* *tr*]] **by** (*auto simp add: list-le-def list-less-def*)

show $xs = ys$ **if** $xs \leq ys$ $ys \leq xs$ **for** *xs* *ys* :: '*a* *list*

using § *that* *list-le-def list-less-def* **by** *blast*

show $xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs$ **for** *xs* *ys* :: '*a* *list*

by (*auto simp add: list-less-def list-le-def dest: §*)

qed

instance *list* :: (*linorder*) *linorder*

proof

fix *xs* *ys* :: '*a* *list*

have *total* ($\{(u, v::'a). u < v\}$)

by (*rule total-lexord*) (*auto simp: total-on-def*)

then show $xs \leq ys \vee ys \leq xs$

by (*auto simp add: total-on-def list-le-def list-less-def*)

qed

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition (*inf* :: 'a *list* \Rightarrow -) = *min*

definition (*sup* :: 'a *list* \Rightarrow -) = *max*

instance

by *standard* (*auto simp add: inf-list-def sup-list-def max-min-distrib2*)

end

lemma *not-less-Nil* [*simp*]: $\neg x < []$
by (*simp add: list-less-def*)

lemma *Nil-less-Cons* [*simp*]: $[] < a \# x$
by (*simp add: list-less-def*)

lemma *Cons-less-Cons* [*simp*]: $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$
by (*simp add: list-less-def*)

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
unfolding *list-le-def* **by** *auto*

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = $[]$

instance

by *standard* (*simp add: bot-list-def*)

end

lemma *less-list-code* [*code*]:
 $xs < ([] :: 'a :: \{equal, order\} list) \longleftrightarrow False$
 $[] < (x :: 'a :: \{equal, order\}) \# xs \longleftrightarrow True$
 $(x :: 'a :: \{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
by *simp-all*

lemma *less-eq-list-code* [*code*]:


```

 $x \# xs \leq ([::'a::\{equal, order\} list) \longleftrightarrow False$ 
 $[] \leq (xs::'a::\{equal, order\} list) \longleftrightarrow True$ 
 $(x::'a::\{equal, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$ 
by simp-all

```

end

112 Lexicographic order on lists

This version prioritises length and can yield wellorderings

```

theory List-Lenlexorder
imports Main
begin

```

```

instantiation list :: (ord) ord
begin

```

definition

```

 $list-less-def: xs < ys \longleftrightarrow (xs, ys) \in lenlex \{(u, v). u < v\}$ 

```

definition

```

 $list-le-def: (xs :: - list) \leq ys \longleftrightarrow xs < ys \vee xs = ys$ 

```

instance ..

end

```

instance list :: (order) order

```

proof

```

  have tr:  $trans \{(u, v::'a). u < v\}$ 
    using trans-def by fastforce
  have §: False
    if  $(xs,ys) \in lenlex \{(u, v). u < v\}$   $(ys,xs) \in lenlex \{(u, v). u < v\}$  for  $xs\ ys :: 'a\ list$ 
  proof –
    have  $(xs,xs) \in lenlex \{(u, v). u < v\}$ 
      using that transD [OF lenlex-transI [OF tr]] by blast
    then show False
      by (meson case-prodD lenlex-irreflexive less-irrefl mem-Collect-eq)
  qed
  show  $xs \leq xs$  for  $xs :: 'a\ list$  by (simp add: list-le-def)
  show  $xs \leq zs$  if  $xs \leq ys$  and  $ys \leq zs$  for  $xs\ ys\ zs :: 'a\ list$ 
    using that transD [OF lenlex-transI [OF tr]] by (auto simp add: list-le-def list-less-def)
  show  $xs = ys$  if  $xs \leq ys$   $ys \leq xs$  for  $xs\ ys :: 'a\ list$ 
    using § that list-le-def list-less-def by blast
  show  $xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs$  for  $xs\ ys :: 'a\ list$ 

```

by (auto simp add: list-less-def list-le-def dest: §)
qed

instance list :: (linorder) linorder

proof

fix xs ys :: 'a list

have total (lenlex {(u, v::'a). u < v})

by (rule total-lenlex) (auto simp: total-on-def)

then show $xs \leq ys \vee ys \leq xs$

by (auto simp add: total-on-def list-le-def list-less-def)

qed

instance list :: (wellorder) wellorder

proof

fix P :: 'a list \Rightarrow bool and a

assume $\bigwedge x. (\bigwedge y. y < x \Longrightarrow P y) \Longrightarrow P x$

then show $P a$

unfolding list-less-def by (metis wf-lenlex wf-induct wf-lenlex wf)

qed

instantiation list :: (linorder) distrib-lattice

begin

definition (inf :: 'a list \Rightarrow -) = min

definition (sup :: 'a list \Rightarrow -) = max

instance

by standard (auto simp add: inf-list-def sup-list-def max-min-distrib2)

end

lemma not-less-Nil [simp]: $\neg x < []$

by (simp add: list-less-def)

lemma Nil-less-Cons [simp]: $[] < a \# x$

by (simp add: list-less-def)

lemma Cons-less-Cons: $a \# x < b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x < y)$

using lenlex-length

by (fastforce simp: list-less-def Cons-lenlex-iff)

lemma le-Nil [simp]: $x \leq [] \longleftrightarrow x = []$

unfolding list-le-def by (cases x) auto

lemma Nil-le-Cons [simp]: $[] \leq x$

unfolding list-le-def by (cases x) auto

lemma *Cons-le-Cons*: $a \# x \leq b \# y \longleftrightarrow \text{length } x < \text{length } y \vee \text{length } x = \text{length } y \wedge (a < b \vee a = b \wedge x \leq y)$

by (*auto simp: list-le-def Cons-less-Cons*)

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
by *standard (simp add: bot-list-def)*

end

end

113 Prefix order on lists as order class instance

theory *Prefix-Order*
imports *Sublist*
begin

instantiation *list* :: (*type*) *order*
begin

definition $xs \leq ys \equiv \text{prefix } xs \text{ } ys$ **for** $xs \text{ } ys :: 'a \text{ list}$

definition $xs < ys \equiv xs \leq ys \wedge \neg (ys \leq xs)$ **for** $xs \text{ } ys :: 'a \text{ list}$

instance
by *standard (auto simp: less-eq-list-def less-list-def)*

end

lemma *less-list-def'*: $xs < ys \longleftrightarrow \text{strict-prefix } xs \text{ } ys$ **for** $xs \text{ } ys :: 'a \text{ list}$
by (*simp add: less-eq-list-def order.strict-iff-order prefix-order.less-le*)

lemmas *prefixI* [*intro?*] = *prefixI* [*folded less-eq-list-def*]
lemmas *prefixE* [*elim?*] = *prefixE* [*folded less-eq-list-def*]
lemmas *strict-prefixI'* [*intro?*] = *strict-prefixI'* [*folded less-list-def*]
lemmas *strict-prefixE'* [*elim?*] = *strict-prefixE'* [*folded less-list-def*]
lemmas *strict-prefixI* [*intro?*] = *strict-prefixI* [*folded less-list-def*]
lemmas *strict-prefixE* [*elim?*] = *strict-prefixE* [*folded less-list-def*]
lemmas *Nil-prefix* [*iff*] = *Nil-prefix* [*folded less-eq-list-def*]
lemmas *prefix-Nil* [*simp*] = *prefix-Nil* [*folded less-eq-list-def*]
lemmas *prefix-snoc* [*simp*] = *prefix-snoc* [*folded less-eq-list-def*]
lemmas *Cons-prefix-Cons* [*simp*] = *Cons-prefix-Cons* [*folded less-eq-list-def*]
lemmas *same-prefix-prefix* [*simp*] = *same-prefix-prefix* [*folded less-eq-list-def*]
lemmas *same-prefix-nil* [*iff*] = *same-prefix-nil* [*folded less-eq-list-def*]
lemmas *prefix-prefix* [*simp*] = *prefix-prefix* [*folded less-eq-list-def*]

```

lemmas prefix-Cons = prefix-Cons [folded less-eq-list-def]
lemmas prefix-length-le = prefix-length-le [folded less-eq-list-def]
lemmas strict-prefix-simps [simp, code] = strict-prefix-simps [folded less-list-def]
lemmas not-prefix-induct [consumes 1, case-names Nil Neq Eq] =
  not-prefix-induct [folded less-eq-list-def]

end

```

114 Lexicographic order on product types

```

theory Product-Lexorder
imports Main
begin

```

```

instantiation prod :: (ord, ord) ord
begin

```

```

definition
   $x \leq y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x \leq \text{snd } y$ 

```

```

definition
   $x < y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x \leq \text{fst } y \wedge \text{snd } x < \text{snd } y$ 

```

```

instance ..

```

```

end

```

```

lemma less-eq-prod-simp [simp, code]:
   $(x1, y1) \leq (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$ 
  by (simp add: less-eq-prod-def)

```

```

lemma less-prod-simp [simp, code]:
   $(x1, y1) < (x2, y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$ 
  by (simp add: less-prod-def)

```

A stronger version for partial orders.

```

lemma less-prod-def':
  fixes x y :: 'a::order × 'b::ord
  shows  $x < y \longleftrightarrow \text{fst } x < \text{fst } y \vee \text{fst } x = \text{fst } y \wedge \text{snd } x < \text{snd } y$ 
  by (auto simp add: less-prod-def le-less)

```

```

instance prod :: (preorder, preorder) preorder
  by standard (auto simp: less-eq-prod-def less-prod-def less-le-not-le intro: order-trans)

```

```

instance prod :: (order, order) order
  by standard (auto simp add: less-eq-prod-def)

```

```

instance prod :: (linorder, linorder) linorder

```

```

by standard (auto simp: less-eq-prod-def)

instantiation prod :: (linorder, linorder) distrib-lattice
begin

definition
  (inf :: 'a × 'b ⇒ - ⇒ -) = min

definition
  (sup :: 'a × 'b ⇒ - ⇒ -) = max

instance
  by standard (auto simp add: inf-prod-def sup-prod-def max-min-distrib2)

end

instantiation prod :: (bot, bot) bot
begin

definition
  bot = (bot, bot)

instance ..

end

instance prod :: (order-bot, order-bot) order-bot
  by standard (auto simp add: bot-prod-def)

instantiation prod :: (top, top) top
begin

definition
  top = (top, top)

instance ..

end

instance prod :: (order-top, order-top) order-top
  by standard (auto simp add: top-prod-def)

instance prod :: (wellorder, wellorder) wellorder
proof
  fix P :: 'a × 'b ⇒ bool and z :: 'a × 'b
  assume P:  $\bigwedge x. (\bigwedge y. y < x \implies P\ y) \implies P\ x$ 
  show P z
  proof (induct z)
    case (Pair a b)

```

```

show  $P(a, b)$ 
proof (induct a arbitrary: b rule: less-induct)
  case (less a1) note  $a_1 = \text{this}$ 
  show  $P(a_1, b)$ 
  proof (induct b rule: less-induct)
    case (less b1) note  $b_1 = \text{this}$ 
    show  $P(a_1, b_1)$ 
    proof (rule P)
      fix  $p$  assume  $p: p < (a_1, b_1)$ 
      show  $P p$ 
      proof (cases fst p < a1)
        case True
          then have  $P(\text{fst } p, \text{snd } p)$  by (rule a1)
          then show ?thesis by simp
        next
          case False
          with  $p$  have  $1: a_1 = \text{fst } p$  and  $2: \text{snd } p < b_1$ 
            by (simp-all add: less-prod-def')
          from  $2$  have  $P(a_1, \text{snd } p)$  by (rule b1)
          with  $1$  show ?thesis by simp
      qed
    qed
  qed
qed
qed
qed
qed

```

Legacy lemma bindings

```

lemmas prod-le-def = less-eq-prod-def
lemmas prod-less-def = less-prod-def
lemmas prod-less-eq = less-prod-def'

```

end

115 Subsequence Ordering

```

theory Subseq-Order
imports Sublist
begin

```

This theory defines subsequence ordering on lists. A list ys is a subsequence of a list xs , iff one obtains ys by erasing some elements from xs .

115.1 Definitions and basic lemmas

```

instantiation list :: (type) ord
begin

```

```

definition less-eq-list

```

where $\langle xs \leq ys \longleftrightarrow \text{subseq } xs \text{ } ys \rangle$ **for** $xs \text{ } ys :: \langle 'a \text{ list} \rangle$

definition *less-list*

where $\langle xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs \rangle$ **for** $xs \text{ } ys :: \langle 'a \text{ list} \rangle$

instance ..

end

instance *list* :: (type) order

proof

fix $xs \text{ } ys \text{ } zs :: 'a \text{ list}$

show $xs < ys \longleftrightarrow xs \leq ys \wedge \neg ys \leq xs$

unfolding *less-list-def* ..

show $xs \leq xs$

by (*simp add: less-eq-list-def*)

show $xs = ys$ **if** $xs \leq ys$ **and** $ys \leq xs$

using *that* **unfolding** *less-eq-list-def*

by (*rule subseq-order.antisym*)

show $xs \leq zs$ **if** $xs \leq ys$ **and** $ys \leq zs$

using *that* **unfolding** *less-eq-list-def*

by (*rule subseq-order.order-trans*)

qed

lemmas *less-eq-list-induct* [*consumes 1, case-names empty drop take*] =
list-emb.induct [*of (=), folded less-eq-list-def*]

lemma *less-eq-list-empty* [*code*]:

$\langle [] \leq xs \longleftrightarrow \text{True} \rangle$

by (*simp add: less-eq-list-def*)

lemma *less-eq-list-below-empty* [*code*]:

$\langle x \# xs \leq [] \longleftrightarrow \text{False} \rangle$

by (*simp add: less-eq-list-def*)

lemma *le-list-Cons2-iff* [*simp, code*]:

$\langle x \# xs \leq y \# ys \longleftrightarrow (\text{if } x = y \text{ then } xs \leq ys \text{ else } x \# xs \leq ys) \rangle$

by (*simp add: less-eq-list-def*)

lemma *less-list-empty* [*simp*]:

$\langle [] < xs \longleftrightarrow xs \neq [] \rangle$

by (*metis less-eq-list-def list-emb-Nil order-less-le*)

lemma *less-list-empty-Cons* [*code*]:

$\langle [] < x \# xs \longleftrightarrow \text{True} \rangle$

by *simp-all*

lemma *less-list-below-empty* [*simp, code*]:

$\langle xs < [] \longleftrightarrow \text{False} \rangle$

```

by (metis list-emb-Nil less-eq-list-def less-list-def)

lemma less-list-Cons2-iff [code]:
  ⟨ $x \# xs < y \# ys \longleftrightarrow (if\ x = y\ then\ xs < ys\ else\ x \# xs \leq ys)$ ⟩
by (simp add: less-le)

lemmas less-eq-list-drop = list-emb.list-emb-Cons [of (=), folded less-eq-list-def]
lemmas le-list-map = subseq-map [folded less-eq-list-def]
lemmas le-list-filter = subseq-filter [folded less-eq-list-def]
lemmas le-list-length = list-emb-length [of (=), folded less-eq-list-def]

lemma less-list-length:  $xs < ys \implies length\ xs < length\ ys$ 
by (metis list-emb-length subseq-same-length le-neq-implies-less less-list-def less-eq-list-def)

lemma less-list-drop:  $xs < ys \implies xs < x \# ys$ 
by (unfold less-le less-eq-list-def) (auto)

lemma less-list-take-iff:  $x \# xs < x \# ys \longleftrightarrow xs < ys$ 
by (metis subseq-Cons2-iff less-list-def less-eq-list-def)

lemma less-list-drop-many:  $xs < ys \implies xs < zs @ ys$ 
by (metis subseq-append-le-same-iff subseq-drop-many order-less-le
self-append-conv2 less-eq-list-def)

lemma less-list-take-many-iff:  $zs @ xs < zs @ ys \longleftrightarrow xs < ys$ 
by (metis less-list-def less-eq-list-def subseq-append)

lemma less-list-rev-take:  $xs @ zs < ys @ zs \longleftrightarrow xs < ys$ 
by (unfold less-le less-eq-list-def) auto

end

```

116 Records based on BNF/datatype machinery

```

theory Datatype-Records
imports Main
keywords datatype-record :: thy-defn
begin

```

This theory provides an alternative, stripped-down implementation of records based on the machinery of the **datatype** package.

It supports:

- similar declaration syntax as records
- record creation and update syntax (using $\langle \dots \rangle$ brackets)
- regular datatype features (e.g. dead type variables etc.)
- “after-the-fact” registration of single-free-constructor types as records

Caveats:

- there is no compatibility layer; importing this theory will disrupt existing syntax
- extensible records are not supported

nonterminal

ident **and**
field-type **and**
field-types **and**
field **and**
fields **and**
field-update **and**
field-updates

open-bundle *datatype-record-syntax*
begin

unbundle *no record-syntax*

syntax

-constify :: *id* => *ident* (⟨-⟩)
-constify :: *longid* => *ident* (⟨-⟩)

-datatype-field :: *ident* => '*a*' => *field* (⟨(⟨indent=2 notation=⟨infix
field value⟩- =/ -)⟩)
:: *field* => *fields* (⟨-⟩)
-datatype-fields :: *field* => *fields* => *fields* (⟨-,/ -⟩)
-datatype-record :: *fields* => '*a*' (⟨(⟨indent=3 notation=⟨mixfix
datatype record value⟩(|-|)⟩)
-datatype-field-update :: *ident* => '*a*' => *field-update* (⟨(⟨indent=2 nota-
tion=⟨infix *field update*⟩- =/ -)⟩)
:: *field-update* => *field-updates* (⟨-⟩)
-datatype-field-updates :: *field-update* => *field-updates* => *field-updates* (⟨-,/ -⟩)
-datatype-record-update :: '*a*' => *field-updates* => '*b*' (⟨(⟨open-block nota-
tion=⟨mixfix *datatype record update*⟩-/(3(|-|)⟩) [900, 0] 900)

syntax (ASCII)

-datatype-record :: *fields* => '*a*' (⟨(⟨indent=3 notation=⟨mixfix
datatype record value⟩'(|-|')⟩)
-datatype-record-update :: '*a*' => *field-updates* => '*b*' (⟨(⟨open-block nota-
tion=⟨mixfix *datatype record update*⟩-/(3'(|-|')⟩) [900, 0] 900)

end

named-theorems *datatype-record-update*

ML-file ⟨*datatype-records.ML*⟩

```

setup ‹Datatype-Records.setup›
end

```

117 Implementation of mappings with Association Lists

```

theory AList-Mapping
  imports AList Mapping
begin

```

lift-definition *Mapping* :: $('a \times 'b)$ list \Rightarrow $('a, 'b)$ mapping **is** map-of .

code-datatype *Mapping*

lemma *lookup-Mapping* [simp, code]: *Mapping.lookup* (*Mapping xs*) = map-of *xs*
by transfer rule

lemma *keys-Mapping* [simp, code]: *Mapping.keys* (*Mapping xs*) = set (map fst *xs*)
by transfer (simp add: dom-map-of-conv-image-fst)

lemma *empty-Mapping* [code]: *Mapping.empty* = *Mapping []*
by transfer simp

lemma *is-empty-Mapping* [code]: *Mapping.is-empty* (*Mapping xs*) \longleftrightarrow List.null *xs*
by (cases *xs*) (simp-all add: is-empty-def)

lemma *update-Mapping* [code]: *Mapping.update* *k v* (*Mapping xs*) = *Mapping (AList.update k v xs)*
by transfer (simp add: update-conv')

lemma *delete-Mapping* [code]: *Mapping.delete* *k* (*Mapping xs*) = *Mapping (AList.delete k xs)*
by transfer (simp add: delete-conv')

lemma *ordered-keys-Mapping* [code]:
Mapping.ordered-keys (*Mapping xs*) = sort (remdups (map fst *xs*))
by (simp only: ordered-keys-def keys-Mapping sorted-list-of-set-sort-remdups) simp

lemma *entries-Mapping* [code]:
Mapping.entries (*Mapping xs*) = set (*AList.clearjunk xs*)
by transfer (fact graph-map-of)

lemma *ordered-entries-Mapping* [code]:
Mapping.ordered-entries (*Mapping xs*) = sort-key fst (*AList.clearjunk xs*)

proof –
have *distinct*: distinct (sort-key fst (*AList.clearjunk xs*))
using *distinct-clearjunk distinct-map distinct-sort* **by** blast

note *folding-Map-graph.idem-if-sorted-distinct*[**where** *?m=map-of xs, OF - sorted-sort-key distinct*]

then show *?thesis*
unfolding *ordered-entries-def*
by (*transfer fixing: xs*) (*auto simp: graph-map-of*)
qed

lemma *fold-Mapping* [code]:

Mapping.fold f (Mapping xs) a = List.fold (case-prod f) (sort-key fst (AList.clearjunk xs)) a
by (*simp add: Mapping.fold-def ordered-entries-Mapping*)

lemma *size-Mapping* [code]: *Mapping.size (Mapping xs) = length (remdups (map fst xs))*

by (*simp add: size-def length-remdups-card-conv dom-map-of-conv-image-fst*)

lemma *tabulate-Mapping* [code]: *Mapping.tabulate ks f = Mapping (map ($\lambda k. (k, f k)$) ks)*

by *transfer (simp add: map-of-map-restrict)*

lemma *bulkload-Mapping* [code]:

*Mapping.bulkload vs = Mapping (map ($\lambda n. (n, vs ! n)$) [0..*length vs*])*

by *transfer (simp add: map-of-map-restrict fun-eq-iff)*

lemma *equal-Mapping* [code]:

HOL.equal (Mapping xs) (Mapping ys) \longleftrightarrow

(let ks = map fst xs; ls = map fst ys

in ($\forall l \in \text{set } ls. l \in \text{set } ks$) \wedge ($\forall k \in \text{set } ks. k \in \text{set } ls \wedge \text{map-of } xs \ k = \text{map-of } ys \ k$))

proof –

have *: (*a, b*) $\in \text{set } xs \implies a \in \text{fst} \text{ ‘ set } xs$ **for** *a b xs*

by (*auto simp add: image-def intro!: bexI*)

show *?thesis*

apply *transfer*

apply (*auto intro!: map-of-eqI*)

apply (*auto dest!: map-of-eq-dom intro: **)

done

qed

lemma *map-values-Mapping* [code]:

Mapping.map-values f (Mapping xs) = Mapping (map ($\lambda(x,y). (x, f \ x \ y)$) xs)

for *f :: 'c \Rightarrow 'a \Rightarrow 'b and xs :: ('c \times 'a) list*

apply *transfer*

apply (*rule ext*)

subgoal for *f xs x by (induct xs) auto*

done

lemma *combine-with-key-code* [code]:

Mapping.combine-with-key f (Mapping xs) (Mapping ys) =

```

    Mapping.tabulate (remdups (map fst xs @ map fst ys))
      (λx. the (combine-options (f x) (map-of xs x) (map-of ys x)))
  apply transfer
  apply (rule ext)
  apply (rule sym)
  subgoal for f xs ys x
    apply (cases map-of xs x; cases map-of ys x; simp)
    apply (force simp: map-of-eq-None-iff combine-options-def option.the-def
o-def image-iff
      dest: map-of-SomeD split: option.splits)+
  done
done

```

```

lemma combine-code [code]:
  Mapping.combine f (Mapping xs) (Mapping ys) =
    Mapping.tabulate (remdups (map fst xs @ map fst ys))
      (λx. the (combine-options f (map-of xs x) (map-of ys x)))
  apply transfer
  apply (rule ext)
  apply (rule sym)
  subgoal for f xs ys x
    apply (cases map-of xs x; cases map-of ys x; simp)
    apply (force simp: map-of-eq-None-iff combine-options-def option.the-def
o-def image-iff
      dest: map-of-SomeD split: option.splits)+
  done
done

```

```

lemma map-of-filter-distinct:
  assumes distinct (map fst xs)
  shows map-of (filter P xs) x =
    (case map-of xs x of
      None ⇒ None
    | Some y ⇒ if P (x,y) then Some y else None)
  using assms
  by (auto simp: map-of-eq-None-iff filter-map distinct-map-filter dest: map-of-SomeD
    simp del: map-of-eq-Some-iff intro!: map-of-is-SomeI split: option.splits)

```

```

lemma filter-Mapping [code]:
  Mapping.filter P (Mapping xs) = Mapping (filter (λ(k,v). P k v) (AList.clearjunk
xs))
  apply transfer
  apply (rule ext)
  apply (subst map-of-filter-distinct)
  apply (simp-all add: map-of-clearjunk split: option.split)
  done

```

```

lemma [code nbe]: HOL.equal (x :: ('a, 'b) mapping) x ⟷ True
  by (fact equal-refl)

```

end

theory *Code-Abstract-Char*

imports

Main

HOL-Library.Char-ord

begin

definition *Chr* :: $\langle \text{integer} \Rightarrow \text{char} \rangle$

where [*simp*]: $\langle \text{Chr} = \text{char-of} \rangle$

lemma *char-of-integer-of-char* [*code abstype*]:

$\langle \text{Chr} (\text{integer-of-char } c) = c \rangle$

by (*simp add: integer-of-char-def*)

lemma *char-of-integer-code* [*code*]:

$\langle \text{integer-of-char} (\text{char-of-integer } k) = (\text{if } 0 \leq k \wedge k < 256 \text{ then } k \text{ else } k \bmod 256) \rangle$

by (*simp add: integer-of-char-def char-of-integer-def integer-eq-iff integer-less-eq-iff integer-less-iff*)

lemma *of-char-code* [*code*]:

$\langle \text{of-char } c = \text{of-nat} (\text{nat-of-integer} (\text{integer-of-char } c)) \rangle$

proof –

have $\langle \text{int-of-integer} (\text{of-char } c) = \text{of-char } c \rangle$

by (*cases c simp*)

then show *?thesis*

by (*simp add: integer-of-char-def nat-of-integer-def of-nat-of-char*)

qed

definition *byte* :: $\langle \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{integer} \rangle$

where [*simp*]: $\langle \text{byte } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ b7 = \text{horner-sum of-bool } 2 \ [b0, b1, b2, b3, b4, b5, b6, b7] \rangle$

lemma *byte-code* [*code*]:

$\langle \text{byte } b0 \ b1 \ b2 \ b3 \ b4 \ b5 \ b6 \ b7 = ($

let

s0 = if b0 then 1 else 0;

s1 = if b1 then s0 + 2 else s0;

s2 = if b2 then s1 + 4 else s1;

s3 = if b3 then s2 + 8 else s2;

s4 = if b4 then s3 + 16 else s3;

s5 = if b5 then s4 + 32 else s4;

s6 = if b6 then s5 + 64 else s5;

s7 = if b7 then s6 + 128 else s6

in s7) \rangle

by *simp*

lemma *Char-code* [code]:

$\langle \text{integer-of-char } (\text{Char } b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7) = \text{byte } b0\ b1\ b2\ b3\ b4\ b5\ b6\ b7 \rangle$
by (*simp add: integer-of-char-def*)

lemma *digit-0-code* [code]:

$\langle \text{digit0 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 0 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *digit-1-code* [code]:

$\langle \text{digit1 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 1 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *digit-2-code* [code]:

$\langle \text{digit2 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 2 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *digit-3-code* [code]:

$\langle \text{digit3 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 3 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *digit-4-code* [code]:

$\langle \text{digit4 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 4 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *digit-5-code* [code]:

$\langle \text{digit5 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 5 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *digit-6-code* [code]:

$\langle \text{digit6 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 6 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *digit-7-code* [code]:

$\langle \text{digit7 } c \longleftrightarrow \text{bit } (\text{integer-of-char } c)\ 7 \rangle$
by (*cases c*) (*simp add: integer-of-char-def*)

lemma *case-char-code* [code]:

$\langle \text{case-char } f\ c = f\ (\text{digit0 } c)\ (\text{digit1 } c)\ (\text{digit2 } c)\ (\text{digit3 } c)\ (\text{digit4 } c)\ (\text{digit5 } c)\ (\text{digit6 } c)\ (\text{digit7 } c) \rangle$
by (*fact char.case-eq-if*)

lemma *rec-char-code* [code]:

$\langle \text{rec-char } f\ c = f\ (\text{digit0 } c)\ (\text{digit1 } c)\ (\text{digit2 } c)\ (\text{digit3 } c)\ (\text{digit4 } c)\ (\text{digit5 } c)\ (\text{digit6 } c)\ (\text{digit7 } c) \rangle$
by (*cases c*) *simp*

lemma *char-of-code* [code]:

$\langle \text{integer-of-char } (\text{char-of } a) =$
 $\text{byte } (\text{bit } a \ 0) (\text{bit } a \ 1) (\text{bit } a \ 2) (\text{bit } a \ 3) (\text{bit } a \ 4) (\text{bit } a \ 5) (\text{bit } a \ 6) (\text{bit } a \ 7) \rangle$
 $\text{by } (\text{simp add: char-of-def integer-of-char-def})$

lemma *ascii-of-code* [code]:

$\langle \text{integer-of-char } (\text{String.ascii-of } c) = (\text{let } k = \text{integer-of-char } c \text{ in if } k < 128 \text{ then } k \text{ else } k - 128) \rangle$

proof (cases $\langle \text{of-char } c < (128 :: \text{integer}) \rangle$)

case *True*

moreover have $\langle (\text{of-nat } 0 :: \text{integer}) \leq \text{of-char } c \rangle$

by *simp*

then have $\langle (0 :: \text{integer}) \leq \text{of-char } c \rangle$

by (*simp only: of-nat-0 of-nat-of-char*)

ultimately show ?thesis

by (*simp add: Let-def integer-of-char-def take-bit-eq-mod integer-eq-iff integer-less-eq-iff integer-less-iff*)

next

case *False*

then have $\langle (128 :: \text{integer}) \leq \text{of-char } c \rangle$

by *simp*

moreover have $\langle \text{of-nat } (\text{of-char } c) < (\text{of-nat } 256 :: \text{integer}) \rangle$

by (*simp only: of-nat-less-iff*) *simp*

then have $\langle \text{of-char } c < (256 :: \text{integer}) \rangle$

by (*simp add: of-nat-of-char*)

moreover define $k :: \text{integer}$ where $\langle k = \text{of-char } c - 128 \rangle$

then have $\langle \text{of-char } c = k + 128 \rangle$

by *simp*

ultimately show ?thesis

by (*simp add: Let-def integer-of-char-def take-bit-eq-mod integer-eq-iff integer-less-eq-iff integer-less-iff*)

qed

lemma *equal-char-code* [code]:

$\langle \text{HOL.equal } c \ d \longleftrightarrow \text{integer-of-char } c = \text{integer-of-char } d \rangle$

by (*simp add: integer-of-char-def equal*)

lemma *less-eq-char-code* [code]:

$\langle c \leq d \longleftrightarrow \text{integer-of-char } c \leq \text{integer-of-char } d \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)

proof –

have $\langle ?P \longleftrightarrow \text{of-nat } (\text{of-char } c) \leq (\text{of-nat } (\text{of-char } d) :: \text{integer}) \rangle$

by (*simp add: less-eq-char-def*)

also have $\langle \dots \longleftrightarrow ?Q \rangle$

by (*simp add: of-nat-of-char integer-of-char-def*)

finally show ?thesis .

qed

lemma *less-char-code* [code]:

$\langle c < d \longleftrightarrow \text{integer-of-char } c < \text{integer-of-char } d \rangle$ (is $\langle ?P \longleftrightarrow ?Q \rangle$)

proof –

```

have ⟨?P  $\longleftrightarrow$  of-nat (of-char c) < (of-nat (of-char d) :: integer)⟩
  by (simp add: less-char-def)
also have ⟨...  $\longleftrightarrow$  ?Q⟩
  by (simp add: of-nat-of-char integer-of-char-def)
finally show ?thesis .
qed

```

lemma absdef-simps:

```

⟨horner-sum of-bool 2 [] = (0 :: integer)⟩
⟨horner-sum of-bool 2 (False # bs) = (0 :: integer)  $\longleftrightarrow$  horner-sum of-bool 2 bs
= (0 :: integer)⟩
⟨horner-sum of-bool 2 (True # bs) = (1 :: integer)  $\longleftrightarrow$  horner-sum of-bool 2 bs
= (0 :: integer)⟩
⟨horner-sum of-bool 2 (False # bs) = (numeral (Num.Bit0 n) :: integer)  $\longleftrightarrow$ 
horner-sum of-bool 2 bs = (numeral n :: integer)⟩
⟨horner-sum of-bool 2 (True # bs) = (numeral (Num.Bit1 n) :: integer)  $\longleftrightarrow$ 
horner-sum of-bool 2 bs = (numeral n :: integer)⟩
by auto (auto simp only: numeral-Bit0 [of n] numeral-Bit1 [of n] mult-2 [symmetric]
add commute [of - 1] add.left-cancel mult-cancel-left)

```

local-setup ⟨

```

  let
    val simps = @{thms absdef-simps integer-of-char-def of-char-Char numeral-One}
    fun prove-eqn lthy n lhs def-eqn =
      let
        val eqn = (HOLogic.mk-Trueprop o HOLogic.mk-eq)
          (term ⟨integer-of-char⟩ $ lhs, HOLogic.mk-number typ ⟨integer⟩ n)
      in
        Goal.prove-future lthy [] [] eqn (fn {context = ctxt, ...} =>
          unfold-tac ctxt (def-eqn :: simps))
      end
    fun define n =
      let
        val s = Char- ^ String-Syntax.hex n;
        val b = Binding.name s;
        val b-def = Thm.def-binding b;
        val b-code = Binding.name (s ^ -code);
      in
        Local-Theory.define ((b, Mixfix.NoSyn),
          ((Binding.empty, []), HOLogic.mk-char n))
          #-> (fn (lhs, (-, raw-def-eqn)) =>
            Local-Theory.note ((b-def, @{attributes [code-abbrev]}), [HOLogic.mk-obj-eq
              raw-def-eqn])
              #-> (fn (-, [def-eqn]) => ‘(fn lthy => prove-eqn lthy n lhs def-eqn)’)
              #-> (fn raw-code-eqn => Local-Theory.note ((b-code, []), [raw-code-eqn]))
              #-> (fn (-, [code-eqn]) => Code.declare-abstract-eqn code-eqn))
      end
    in
      fold define (0 upto 255)

```



```

end
>

```

```

code-identifier
  code-module Code-Abstract-Char  $\rightarrow$ 
    (SML) Str and (OCaml) Str and (Haskell) Str and (Scala) Str
end

```

118 Avoidance of pattern matching on natural numbers

```

theory Code-Abstract-Nat
imports Main
begin

```

When natural numbers are implemented in another than the conventional inductive *0/Suc* representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

118.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

```

lemma [code, code-unfold]:
  case-nat = ( $\lambda f g n. \text{if } n = 0 \text{ then } f \text{ else } g (n - 1)$ )
  by (auto simp add: fun-eq-iff dest!: gr0-implies-Suc)

```

118.2 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

```

lemma Suc-if-eq:
  assumes  $\bigwedge n. f (Suc\ n) \equiv h\ n$ 
  assumes  $f\ 0 \equiv g$ 
  shows  $f\ n \equiv \text{if } n = 0 \text{ then } g \text{ else } h\ (n - 1)$ 
  by (rule eq-reflection) (cases n, insert assms, simp-all)

```

The rule above is built into a preprocessor that is plugged into the code generator.

```

setup <
  let

```

```

val Suc-if-eq = Thm.incr-indexes 1 @ {thm Suc-if-eq};

fun remove-suc ctxt thms =
  let
    val vname = singleton (Name.variant-list (map fst
      (fold (Term.add-var-names o Thm.full-prop-of) thms [])) n;
    val cv = Thm.ctrm-of ctxt (Var ((vname, 0), HOLogic.natT));
    val lhs-of = Thm.dest-arg1 o Thm.cprop-of;
    val rhs-of = Thm.dest-arg o Thm.cprop-of;
    fun find-vars ct = (case Thm.term-of ct of
      (Const (const-name <Suc>, -) $ Var -) => [(cv, snd (Thm.dest-comb ct))]
    | - $ - =>
      let val (ct1, ct2) = Thm.dest-comb ct
      in
        map (apfst (fn ct => Thm.apply ct ct2)) (find-vars ct1) @
        map (apfst (Thm.apply ct1)) (find-vars ct2)
      end
    | - => []);
    val eqs = maps
      (fn thm => map (pair thm) (find-vars (lhs-of thm))) thms;
    fun mk-thms (thm, (ct, cv')) =
      let
        val thm' =
          Thm.implies-elim
            (Conv.fconv-rule (Thm.beta-conversion true)
              (Thm.instantiate'
                [SOME (Thm.ctyp-of-ctrm ct)] [SOME (Thm.lambda cv ct),
                  SOME (Thm.lambda cv' (rhs-of thm)), NONE, SOME cv']
                Suc-if-eq)) (Thm.forall-intr cv' thm)
        in
          case map-filter (fn thm'' =>
            SOME (thm'', singleton
              (Variable.trade (K (fn [thm'''] => [thm''' RS thm']))
                (Variable.declare-thm thm'' ctxt)) thm''))
            handle THM - => NONE) thms of
            [] => NONE
          | thmps =>
              let val (thms1, thms2) = split-list thmps
              in SOME (subtract Thm.eq-thm (thm :: thms1) thms @ thms2) end
        end
      in get-first mk-thms eqs end;

fun eqn-suc-base-preproc ctxt thms =
  let
    val dest = fst o Logic.dest-equals o Thm.prop-of;
    val contains-suc = exists-Const (fn (c, -) => c = const-name <Suc>);
    in
      if forall (can dest) thms andalso exists (contains-suc o dest) thms
      then thms |> perhaps-loop (remove-suc ctxt) |> (Option.map o map) Drule.zero-var-indexes
    end
  end

```

```

      else NONE
    end;

val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;

in

  Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)

end
>

```

118.3 Candidates which need special treatment

lemma *drop-bit-int-code* [code]:
 $\langle \text{drop-bit } n \ k = k \text{ div } 2^{\wedge n} \rangle$ **for** $k :: \text{int}$
by (fact drop-bit-eq-div)

lemma *take-bit-num-code* [code]:
 $\langle \text{take-bit-num } n \ \text{Num.One} =$
 $(\text{case } n \text{ of } 0 \Rightarrow \text{None} \mid \text{Suc } n \Rightarrow \text{Some Num.One}) \rangle$
 $\langle \text{take-bit-num } n \ (\text{Num.Bit0 } m) =$
 $(\text{case } n \text{ of } 0 \Rightarrow \text{None} \mid \text{Suc } n \Rightarrow (\text{case take-bit-num } n \ m \text{ of } \text{None} \Rightarrow \text{None} \mid$
 $\text{Some } q \Rightarrow \text{Some } (\text{Num.Bit0 } q))) \rangle$
 $\langle \text{take-bit-num } n \ (\text{Num.Bit1 } m) =$
 $(\text{case } n \text{ of } 0 \Rightarrow \text{None} \mid \text{Suc } n \Rightarrow \text{Some } (\text{case take-bit-num } n \ m \text{ of } \text{None} \Rightarrow$
 $\text{Num.One} \mid \text{Some } q \Rightarrow \text{Num.Bit1 } q))) \rangle$
by (cases n; simp)+

end

119 Implementation of natural numbers as binary numerals

theory *Code-Binary-Nat*
imports *Code-Abstract-Nat*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. This theory refines the representation of natural numbers for code generation to use binary numerals, which do not grow linear in size but logarithmic.

119.1 Representation

code-datatype *0::nat nat-of-num*

lemma *[code]*:
 $\text{num-of-nat } 0 = \text{Num.One}$
 $\text{num-of-nat } (\text{nat-of-num } k) = k$
by (*simp-all add: nat-of-num-inverse*)

lemma *[code]*:
 $(1::\text{nat}) = \text{Numeral1}$
by *simp*

lemma *[code-abbrev]*: $\text{Numeral1} = (1::\text{nat})$
by *simp*

lemma *[code]*:
 $\text{Suc } n = n + 1$
by *simp*

119.2 Basic arithmetic

context
begin

lemma *plus-nat-code [code]*:
 $0 + n = (n::\text{nat})$
 $m + 0 = (m::\text{nat})$
 $\text{nat-of-num } k + \text{nat-of-num } l = \text{nat-of-num } (k + l)$
by (*simp-all add: nat-of-num-numeral*)

Bounded subtraction needs some auxiliary

qualified definition $\text{dup} :: \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{dup } n = n + n$

lemma *dup-code [code]*:
 $\text{dup } 0 = 0$
 $\text{dup } (\text{nat-of-num } k) = \text{nat-of-num } (\text{Num.Bit0 } k)$
by (*simp-all add: dup-def numeral-Bit0*)

qualified definition $\text{sub} :: \text{num} \Rightarrow \text{num} \Rightarrow \text{nat option}$ **where**
 $\text{sub } k \ l = (\text{if } k \geq l \text{ then } \text{Some } (\text{numeral } k - \text{numeral } l) \text{ else } \text{None})$

lemma *sub-code [code]*:
 $\text{sub } \text{Num.One } \text{Num.One} = \text{Some } 0$
 $\text{sub } (\text{Num.Bit0 } m) \ \text{Num.One} = \text{Some } (\text{nat-of-num } (\text{Num.BitM } m))$
 $\text{sub } (\text{Num.Bit1 } m) \ \text{Num.One} = \text{Some } (\text{nat-of-num } (\text{Num.Bit0 } m))$
 $\text{sub } \text{Num.One } (\text{Num.Bit0 } n) = \text{None}$
 $\text{sub } \text{Num.One } (\text{Num.Bit1 } n) = \text{None}$
 $\text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit0 } n) = \text{map-option } \text{dup } (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit1 } n) = \text{map-option } \text{dup } (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit1 } m) (\text{Num.Bit0 } n) = \text{map-option } (\lambda q. \text{dup } q + 1) (\text{sub } m \ n)$
 $\text{sub } (\text{Num.Bit0 } m) (\text{Num.Bit1 } n) = (\text{case } \text{sub } m \ n \text{ of } \text{None} \Rightarrow \text{None}$
 $\quad | \text{Some } q \Rightarrow \text{if } q = 0 \text{ then } \text{None} \text{ else } \text{Some } (\text{dup } q - 1))$

by (*auto simp: nat-of-num-numeral Num.dbl-def Num.dbl-inc-def Num.dbl-dec-def*
Let-def le-imp-diff-is-add BitM-plus-one sub-def dup-def
sub-non-positive nat-add-distrib sub-non-negative)

lemma *minus-nat-code* [*code*]:

$0 - n = (0::nat)$

$m - 0 = (m::nat)$

$\text{nat-of-num } k - \text{nat-of-num } l = (\text{case sub } k \text{ } l \text{ of None } \Rightarrow 0 \mid \text{Some } j \Rightarrow j)$

by (*simp-all add: nat-of-num-numeral sub-non-positive sub-def*)

lemma *times-nat-code* [*code*]:

$0 * n = (0::nat)$

$m * 0 = (0::nat)$

$\text{nat-of-num } k * \text{nat-of-num } l = \text{nat-of-num } (k * l)$

by (*simp-all add: nat-of-num-numeral*)

lemma *equal-nat-code* [*code*]:

$\text{HOL.equal } 0 \text{ } (0::nat) \longleftrightarrow \text{True}$

$\text{HOL.equal } 0 \text{ } (\text{nat-of-num } l) \longleftrightarrow \text{False}$

$\text{HOL.equal } (\text{nat-of-num } k) \text{ } 0 \longleftrightarrow \text{False}$

$\text{HOL.equal } (\text{nat-of-num } k) \text{ } (\text{nat-of-num } l) \longleftrightarrow \text{HOL.equal } k \text{ } l$

by (*simp-all add: nat-of-num-numeral equal*)

lemma *equal-nat-refl* [*code nbe*]:

$\text{HOL.equal } (n::nat) \text{ } n \longleftrightarrow \text{True}$

by (*rule equal-refl*)

lemma *less-eq-nat-code* [*code*]:

$0 \leq (n::nat) \longleftrightarrow \text{True}$

$\text{nat-of-num } k \leq 0 \longleftrightarrow \text{False}$

$\text{nat-of-num } k \leq \text{nat-of-num } l \longleftrightarrow k \leq l$

by (*simp-all add: nat-of-num-numeral*)

lemma *less-nat-code* [*code*]:

$(m::nat) < 0 \longleftrightarrow \text{False}$

$0 < \text{nat-of-num } l \longleftrightarrow \text{True}$

$\text{nat-of-num } k < \text{nat-of-num } l \longleftrightarrow k < l$

by (*simp-all add: nat-of-num-numeral*)

lemma *divmod-nat-code* [*code*]:

$\text{Euclidean-Rings.divmod-nat } 0 \text{ } n = (0, 0)$

$\text{Euclidean-Rings.divmod-nat } m \text{ } 0 = (0, m)$

$\text{Euclidean-Rings.divmod-nat } (\text{nat-of-num } k) \text{ } (\text{nat-of-num } l) = \text{divmod } k \text{ } l$

by (*simp-all add: Euclidean-Rings.divmod-nat-def nat-of-num-numeral*)

end

119.3 Conversions

```

lemma of-nat-code [code]:
  of-nat 0 = 0
  of-nat (nat-of-num k) = numeral k
  by (simp-all add: nat-of-num-numeral)

```

code-identifier

```

code-module Code-Binary-Nat  $\rightarrow$ 
  (SML) Arith and (OCaml) Arith and (Haskell) Arith

```

end

120 Code generation of prolog programs

```

theory Code-Prolog
imports Main
keywords values-prolog :: diag
begin

```

ML-file $\langle \sim \sim / \text{src} / \text{HOL} / \text{Tools} / \text{Predicate-Compile} / \text{code-prolog.ML} \rangle$

121 Setup for Numerals

```

setup  $\langle \text{Predicate-Compile-Data.ignore-consts } [\text{const-name } \langle \text{numeral} \rangle] \rangle$ 
setup  $\langle \text{Predicate-Compile-Data.keep-functions } [\text{const-name } \langle \text{numeral} \rangle] \rangle$ 
end

```

122 Implementation of integer numbers by target-language integers

```

theory Code-Target-Int
imports Main
begin

```

code-datatype *int-of-integer*

```

context
includes integer.lifting
begin

```

```

lemma [code]:
  integer-of-int (int-of-integer k) = k
  by transfer rule

```

lemma [code]:
 $Int.Pos = int-of-integer \circ integer-of-num$
by transfer (simp add: fun-eq-iff)

lemma [code]:
 $Int.Neg = int-of-integer \circ uminus \circ integer-of-num$
by transfer (simp add: fun-eq-iff)

lemma [code-abbrev]:
 $int-of-integer (numeral\ k) = Int.Pos\ k$
by transfer simp

lemma [code-abbrev]:
 $int-of-integer (-\ numeral\ k) = Int.Neg\ k$
by transfer simp

context
begin

qualified definition $positive :: num \Rightarrow int$
where [simp]: $positive = numeral$

qualified definition $negative :: num \Rightarrow int$
where [simp]: $negative = uminus \circ numeral$

lemma [code-computation-unfold]:
 $numeral = positive$
 $Int.Pos = positive$
 $Int.Neg = negative$
by (simp-all add: fun-eq-iff)

end

lemma [code, symmetric, code-post]:
 $0 = int-of-integer\ 0$
by transfer simp

lemma [code, symmetric, code-post]:
 $1 = int-of-integer\ 1$
by transfer simp

lemma [code-post]:
 $int-of-integer (-\ 1) = -\ 1$
by simp

lemma [code]:
 $k + l = int-of-integer (of-int\ k + of-int\ l)$
by transfer simp

lemma [code]:
 $- k = \text{int-of-integer } (- \text{ of-int } k)$
by *transfer simp*

lemma [code]:
 $k - l = \text{int-of-integer } (\text{of-int } k - \text{of-int } l)$
by *transfer simp*

lemma [code]:
 $\text{Int.dup } k = \text{int-of-integer } (\text{Code-Numeral.dup } (\text{of-int } k))$
by *transfer simp*

declare [[code drop: *Int.sub*]]

lemma [code]:
 $k * l = \text{int-of-integer } (\text{of-int } k * \text{of-int } l)$
by *simp*

lemma [code]:
 $k \text{ div } l = \text{int-of-integer } (\text{of-int } k \text{ div } \text{of-int } l)$
by *simp*

lemma [code]:
 $k \text{ mod } l = \text{int-of-integer } (\text{of-int } k \text{ mod } \text{of-int } l)$
by *simp*

lemma [code]:
 $\text{divmod } m \ n = \text{map-prod int-of-integer int-of-integer } (\text{divmod } m \ n)$
unfolding *prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv*
by *transfer simp*

lemma [code]:
 $\text{HOL.equal } k \ l = \text{HOL.equal } (\text{of-int } k :: \text{integer}) \ (\text{of-int } l)$
by *transfer (simp add: equal)*

lemma [code]:
 $k \leq l \longleftrightarrow (\text{of-int } k :: \text{integer}) \leq \text{of-int } l$
by *transfer rule*

lemma [code]:
 $k < l \longleftrightarrow (\text{of-int } k :: \text{integer}) < \text{of-int } l$
by *transfer rule*

lemma *gcd-int-of-integer* [code]:
 $\text{gcd } (\text{int-of-integer } x) \ (\text{int-of-integer } y) = \text{int-of-integer } (\text{gcd } x \ y)$
by *transfer rule*

lemma *lcm-int-of-integer* [code]:
 $\text{lcm } (\text{int-of-integer } x) \ (\text{int-of-integer } y) = \text{int-of-integer } (\text{lcm } x \ y)$


```

    by transfer rule

end

lemma (in ring-1) of-int-code-if:
  of-int k = (if k = 0 then 0
    else if k < 0 then - of-int (- k)
    else let
      l = 2 * of-int (k div 2);
      j = k mod 2
      in if j = 0 then l else l + 1)
proof -
  from div-mult-mod-eq have *: of-int k = of-int (k div 2 * 2 + k mod 2) by simp
  show ?thesis
    by (simp add: Let-def of-int-add [symmetric]) (simp add: * mult.commute)
qed

declare of-int-code-if [code]

lemma [code]:
  nat = nat-of-integer ∘ of-int
  including integer.lifting by transfer (simp add: fun-eq-iff)

definition char-of-int :: int ⇒ char
  where [code-abbrev]: char-of-int = char-of

definition int-of-char :: char ⇒ int
  where [code-abbrev]: int-of-char = of-char

lemma [code]:
  char-of-int = char-of-integer ∘ integer-of-int
  including integer.lifting unfolding char-of-integer-def char-of-int-def
  by transfer (simp add: fun-eq-iff)

lemma [code]:
  int-of-char = int-of-integer ∘ integer-of-char
  including integer.lifting unfolding integer-of-char-def int-of-char-def
  by transfer (simp add: fun-eq-iff)

context
  includes integer.lifting and bit-operations-syntax
begin

lemma [code]:
  ⟨bit (int-of-integer k) n ⟷ bit k n⟩
  by transfer rule

lemma [code]:
  ⟨NOT (int-of-integer k) = int-of-integer (NOT k)⟩

```

by *transfer rule*

lemma [code]:

$\langle \text{int-of-integer } k \text{ AND int-of-integer } l = \text{int-of-integer } (k \text{ AND } l) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{int-of-integer } k \text{ OR int-of-integer } l = \text{int-of-integer } (k \text{ OR } l) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{int-of-integer } k \text{ XOR int-of-integer } l = \text{int-of-integer } (k \text{ XOR } l) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{push-bit } n (\text{int-of-integer } k) = \text{int-of-integer } (\text{push-bit } n k) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{drop-bit } n (\text{int-of-integer } k) = \text{int-of-integer } (\text{drop-bit } n k) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{take-bit } n (\text{int-of-integer } k) = \text{int-of-integer } (\text{take-bit } n k) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{mask } n = \text{int-of-integer } (\text{mask } n) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{set-bit } n (\text{int-of-integer } k) = \text{int-of-integer } (\text{set-bit } n k) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{unset-bit } n (\text{int-of-integer } k) = \text{int-of-integer } (\text{unset-bit } n k) \rangle$

by *transfer rule*

lemma [code]:

$\langle \text{flip-bit } n (\text{int-of-integer } k) = \text{int-of-integer } (\text{flip-bit } n k) \rangle$

by *transfer rule*

end

code-identifier

code-module *Code-Target-Int* \rightarrow

(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

```

theory Code-Real-Approx-By-Float
imports Complex-Main Code-Target-Int
begin

```

WARNING! This theory implements mathematical reals by machine reals (floats). This is inconsistent. See the proof of False at the end of the theory, where an equality on mathematical reals is (incorrectly) disproved by mapping it to machine reals.

The **value** command cannot display real results yet.

The only legitimate use of this theory is as a tool for code generation purposes.

```

context
begin

```

```

qualified definition real-of-integer :: integer  $\Rightarrow$  real
  where [code-abbrev]: real-of-integer = of-int  $\circ$  int-of-integer

end

```

```

code-datatype Code-Real-Approx-By-Float.real-of-integer  $\langle (/) \rangle$  :: real  $\Rightarrow$  real  $\Rightarrow$ 
real

```

```

lemma [code-unfold del]: numeral k  $\equiv$  real-of-rat (numeral k)
  by simp

```

```

lemma [code-unfold del]:  $-$  numeral k  $\equiv$  real-of-rat ( $-$  numeral k)
  by simp

```

```

context
begin

```

```

qualified definition real-of-int ::  $\langle \text{int} \Rightarrow \text{real} \rangle$ 
  where [code-abbrev]:  $\langle \text{real-of-int} = \text{of-int} \rangle$ 

```

```

lemma [code]: real-of-int = Code-Real-Approx-By-Float.real-of-integer  $\circ$  integer-of-int
  by (simp add: fun-eq-iff Code-Real-Approx-By-Float.real-of-integer-def real-of-int-def)

```

```

qualified definition exp-real ::  $\langle \text{real} \Rightarrow \text{real} \rangle$ 
  where [code-abbrev, code del]:  $\langle \text{exp-real} = \text{exp} \rangle$ 

```

```

qualified definition sin-real ::  $\langle \text{real} \Rightarrow \text{real} \rangle$ 
  where [code-abbrev, code del]:  $\langle \text{sin-real} = \text{sin} \rangle$ 

```

```

qualified definition cos-real ::  $\langle \text{real} \Rightarrow \text{real} \rangle$ 
  where [code-abbrev, code del]:  $\langle \text{cos-real} = \text{cos} \rangle$ 

```

```

qualified definition tan-real ::  $\langle \text{real} \Rightarrow \text{real} \rangle$ 

```

```

where [code-abbrev, code del]: ⟨tan-real = tan⟩

end

lemma [code]: ⟨r - s = r + (- s)⟩ for r s :: real
  by (fact diff-conv-add-uminus)

lemma [code]: ⟨inverse r = 1 / r⟩ for r :: real
  by (fact inverse-eq-divide)

lemma [code]: ⟨Ratreal r = (let (p, q) = quotient-of r in real-of-int p / real-of-int q)⟩
  by (cases r) (simp add: quotient-of-Fract of-rat-rat)

declare [[code drop:
  ⟨HOL.equal :: real ⇒ real ⇒ bool⟩
  ⟨(≤) :: real ⇒ real ⇒ bool⟩
  ⟨(<) :: real ⇒ real ⇒ bool⟩
  ⟨(+) :: real ⇒ real ⇒ real⟩
  ⟨uminus :: real ⇒ real⟩
  ⟨(*) :: real ⇒ real ⇒ real⟩
  sqrt
  ⟨ln :: real ⇒ real⟩
  pi
  arcsin
  arccos
  arctan]]

code-reserved (SML) Real

code-printing
  type-constructor real ↪
    (SML) real
    and (OCaml) float
    and (Haskell) Prelude.Double
| constant 0 :: real ↪
  (SML) 0.0
  and (OCaml) 0.0
  and (Haskell) 0.0
| constant 1 :: real ↪
  (SML) 1.0
  and (OCaml) 1.0
  and (Haskell) 1.0
| constant HOL.equal :: real ⇒ real ⇒ bool ↪
  (SML) Real.== ((-), (-))
  and (OCaml) Pervasives.(=)
  and (Haskell) infix 4 ==
| class-instance real :: HOL.equal => (Haskell) -
| constant (≤) :: real ⇒ real ⇒ bool ↪

```

```

    (SML) Real.<= ((-), (-))
    and (OCaml) Pervasives.(<=)
    and (Haskell) infix 4 <=
| constant (<) :: real ⇒ real ⇒ bool ⇐
    (SML) Real.< ((-), (-))
    and (OCaml) Pervasives.<
    and (Haskell) infix 4 <
| constant (+) :: real ⇒ real ⇒ real ⇐
    (SML) Real.+ ((-), (-))
    and (OCaml) Pervasives.(+.)
    and (Haskell) infixl 6 +
| constant (*) :: real ⇒ real ⇒ real ⇐
    (SML) Real.* ((-), (-))
    and (Haskell) infixl 7 *
| constant uminus :: real ⇒ real ⇐
    (SML) Real.~
    and (OCaml) Pervasives.(~-. )
    and (Haskell) negate
| constant (-) :: real ⇒ real ⇒ real ⇐
    (SML) Real.- ((-), (-))
    and (OCaml) Pervasives.(-. )
    and (Haskell) infixl 6 -
| constant (/) :: real ⇒ real ⇒ real ⇐
    (SML) Real.'/ ((-), (-))
    and (OCaml) Pervasives.( '/. )
    and (Haskell) infixl 7 /
| constant sqrt :: real ⇒ real ⇐
    (SML) Math.sqrt
    and (OCaml) Pervasives.sqrt
    and (Haskell) Prelude.sqrt
| constant Code-Real-Approx-By-Float.exp-real ⇐
    (SML) Math.exp
    and (OCaml) Pervasives.exp
    and (Haskell) Prelude.exp
| constant ln ⇐
    (SML) Math.ln
    and (OCaml) Pervasives.ln
    and (Haskell) Prelude.log
| constant Code-Real-Approx-By-Float.sin-real ⇐
    (SML) Math.sin
    and (OCaml) Pervasives.sin
    and (Haskell) Prelude.sin
| constant Code-Real-Approx-By-Float.cos-real ⇐
    (SML) Math.cos
    and (OCaml) Pervasives.cos
    and (Haskell) Prelude.cos
| constant Code-Real-Approx-By-Float.tan-real ⇐
    (SML) Math.tan
    and (OCaml) Pervasives.tan

```

```

    and (Haskell) Prelude.tan
| constant pi  $\rightarrow$ 
    (SML) Math.pi

    and (Haskell) Prelude.pi
| constant arcsin  $\rightarrow$ 
    (SML) Math.asin
    and (OCaml) Pervasives.asin
    and (Haskell) Prelude.asin
| constant arccos  $\rightarrow$ 
    (SML) Math.scos
    and (OCaml) Pervasives.acos
    and (Haskell) Prelude.acos
| constant arctan  $\rightarrow$ 
    (SML) Math.atan
    and (OCaml) Pervasives.atan
    and (Haskell) Prelude.atan
| constant Code-Real-Approx-By-Float.real-of-integer  $\rightarrow$ 
    (SML) Real.fromInt
    and (OCaml) Pervasives.float/ (Big'-int.to'-int (-))
    and (Haskell) Prelude.fromIntegral (-)

```

```

notepad
begin
  have  $\cos(\pi/2) = 0$  by (rule cos-pi-half)
  moreover have  $\cos(\pi/2) \neq 0$  by eval
  ultimately have False by blast
end

end

```

123 Implementation of natural numbers by target-language integers

```

theory Code-Target-Nat
imports Code-Abstract-Nat
begin

```

123.1 Implementation for *nat*

```

context
includes integer.lifting
begin

```

```

lift-definition Nat :: integer  $\Rightarrow$  nat
is nat
.

```

lemma [code-post]:

Nat 0 = 0

Nat 1 = 1

Nat (numeral k) = numeral k

by (transfer, simp)+

lemma [code-abbrev]:

integer-of-nat = of-nat

by transfer rule

lemma [code-unfold]:

Int.nat (int-of-integer k) = nat-of-integer k

by transfer rule

lemma [code abstype]:

Code-Target-Nat.Nat (integer-of-nat n) = n

by transfer simp

lemma [code abstract]:

integer-of-nat (nat-of-integer k) = max 0 k

by transfer auto

lemma [code-abbrev]:

nat-of-integer (numeral k) = nat-of-num k

by transfer (simp add: nat-of-num-numeral)

context

begin

qualified definition *natural :: num \Rightarrow nat*

where [simp]: *natural = nat-of-num*

lemma [code-computation-unfold]:

numeral = natural

nat-of-num = natural

by (simp-all add: nat-of-num-numeral)

end

lemma [code abstract]:

integer-of-nat (nat-of-num n) = integer-of-num n

by (simp add: nat-of-num-numeral integer-of-nat-numeral)

lemma [code abstract]:

integer-of-nat 0 = 0

by transfer simp

lemma [code abstract]:

integer-of-nat 1 = 1

by *transfer simp*

lemma [code]:

Suc n = n + 1

by *simp*

lemma [code abstract]:

integer-of-nat (m + n) = of-nat m + of-nat n

by *transfer simp*

lemma [code abstract]:

integer-of-nat (m - n) = max 0 (of-nat m - of-nat n)

by *transfer simp*

lemma [code abstract]:

*integer-of-nat (m * n) = of-nat m * of-nat n*

by *transfer (simp add: of-nat-mult)*

lemma [code abstract]:

integer-of-nat (m div n) = of-nat m div of-nat n

by *transfer (simp add: zdiv-int)*

lemma [code abstract]:

integer-of-nat (m mod n) = of-nat m mod of-nat n

by *transfer (simp add: zmod-int)*

context

includes *integer.lifting*

begin

lemma *divmod-nat-code* [code]:

Euclidean-Rings.divmod-nat m n = (

let k = integer-of-nat m; l = integer-of-nat n

in map-prod nat-of-integer nat-of-integer

(if k = 0 then (0, 0)

else if l = 0 then (0, k) else

Code-Numeral.divmod-abs k l))

by (*simp add: prod-eq-iff Let-def Euclidean-Rings.divmod-nat-def; transfer*)

(*simp add: nat-div-distrib nat-mod-distrib*)

end

lemma [code]:

divmod m n = map-prod nat-of-integer nat-of-integer (divmod m n)

by (*simp only: prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv; transfer*)

(*simp-all only: nat-div-distrib nat-mod-distrib*

zero-le-numeral nat-numeral)


```

lemma [code]:
  HOL.equal m n = HOL.equal (of-nat m :: integer) (of-nat n)
  by transfer (simp add: equal)

lemma [code]:
   $m \leq n \longleftrightarrow (\text{of-nat } m :: \text{integer}) \leq \text{of-nat } n$ 
  by simp

lemma [code]:
   $m < n \longleftrightarrow (\text{of-nat } m :: \text{integer}) < \text{of-nat } n$ 
  by simp

lemma num-of-nat-code [code]:
  num-of-nat = num-of-integer  $\circ$  of-nat
  by transfer (simp add: fun-eq-iff)

end

lemma (in semiring-1) of-nat-code-if:
  of-nat n = (if n = 0 then 0
    else let
      (m, q) = Euclidean-Rings.divmod-nat n 2;
      m' = 2 * of-nat m
      in if q = 0 then m' else m' + 1)
  by (cases n)
  (simp-all add: Let-def Euclidean-Rings.divmod-nat-def ac-simps
    flip: of-nat-numeral of-nat-mult minus-mod-eq-mult-div)

declare of-nat-code-if [code]

definition int-of-nat :: nat  $\Rightarrow$  int where
  [code-abbrev]: int-of-nat = of-nat

lemma [code]:
  int-of-nat n = int-of-integer (of-nat n)
  by (simp add: int-of-nat-def)

lemma [code abstract]:
  integer-of-nat (nat k) = max 0 (integer-of-int k)
  including integer.lifting by transfer auto

definition char-of-nat :: nat  $\Rightarrow$  char
  where [code-abbrev]: char-of-nat = char-of

definition nat-of-char :: char  $\Rightarrow$  nat
  where [code-abbrev]: nat-of-char = of-char

lemma [code]:
  char-of-nat = char-of-integer  $\circ$  integer-of-nat

```

including *integer.lifting* **unfolding** *char-of-integer-def char-of-nat-def*
by *transfer (simp add: fun-eq-iff)*

lemma [*code abstract*]:
integer-of-nat (nat-of-char c) = integer-of-char c
by (*cases c*) (*simp add: nat-of-char-def integer-of-char-def integer-of-nat-eq-of-nat*)

lemma *term-of-nat-code* [*code*]:
 — Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such
 that reconstructed terms can be fed back to the code generator
term-of-class.term-of n =
Code-Evaluation.App
(Code-Evaluation.Const (STR "Code-Numeral.nat-of-integer")
(typerep.Typerep (STR "fun")
[typerep.Typerep (STR "Code-Numeral.integer") [],
typerep.Typerep (STR "Nat.nat") []])
(term-of-class.term-of (integer-of-nat n))
by (*simp add: term-of-anything*)

lemma *nat-of-integer-code-post* [*code-post*]:
nat-of-integer 0 = 0
nat-of-integer 1 = 1
nat-of-integer (numeral k) = numeral k
including *integer.lifting* **by** (*transfer, simp*)+

context
includes *integer.lifting* **and** *bit-operations-syntax*
begin

lemma [*code*]:
 $\langle \text{bit } m \ n \longleftrightarrow \text{bit } (\text{integer-of-nat } m) \ n \rangle$
by *transfer (simp add: bit-simps)*

lemma [*code*]:
 $\langle \text{integer-of-nat } (m \text{ AND } n) = \text{integer-of-nat } m \text{ AND } \text{integer-of-nat } n \rangle$
by *transfer (simp add: of-nat-and-eq)*

lemma [*code*]:
 $\langle \text{integer-of-nat } (m \text{ OR } n) = \text{integer-of-nat } m \text{ OR } \text{integer-of-nat } n \rangle$
by *transfer (simp add: of-nat-or-eq)*

lemma [*code*]:
 $\langle \text{integer-of-nat } (m \text{ XOR } n) = \text{integer-of-nat } m \text{ XOR } \text{integer-of-nat } n \rangle$
by *transfer (simp add: of-nat-xor-eq)*

lemma [*code*]:
 $\langle \text{integer-of-nat } (\text{push-bit } n \ m) = \text{push-bit } n \ (\text{integer-of-nat } m) \rangle$
by *transfer (simp add: of-nat-push-bit)*

lemma [code]:
 $\langle \text{integer-of-nat } (\text{drop-bit } n \ m) = \text{drop-bit } n \ (\text{integer-of-nat } m) \rangle$
by transfer (simp add: of-nat-drop-bit)

lemma [code]:
 $\langle \text{integer-of-nat } (\text{take-bit } n \ m) = \text{take-bit } n \ (\text{integer-of-nat } m) \rangle$
by transfer (simp add: of-nat-take-bit)

lemma [code]:
 $\langle \text{integer-of-nat } (\text{mask } n) = \text{mask } n \rangle$
by transfer (simp add: of-nat-mask-eq)

lemma [code]:
 $\langle \text{integer-of-nat } (\text{set-bit } n \ m) = \text{set-bit } n \ (\text{integer-of-nat } m) \rangle$
by transfer (simp add: of-nat-set-bit-eq)

lemma [code]:
 $\langle \text{integer-of-nat } (\text{unset-bit } n \ m) = \text{unset-bit } n \ (\text{integer-of-nat } m) \rangle$
by transfer (simp add: of-nat-unset-bit-eq)

lemma [code]:
 $\langle \text{integer-of-nat } (\text{flip-bit } n \ m) = \text{flip-bit } n \ (\text{integer-of-nat } m) \rangle$
by transfer (simp add: of-nat-flip-bit-eq)

end

code-identifier

code-module *Code-Target-Nat* \hookrightarrow
 (SML) *Arith* **and** (OCaml) *Arith* **and** (Haskell) *Arith*

end

124 Implementation of natural and integer numbers by target-language integers

theory *Code-Target-Numeral*
imports *Code-Target-Nat* *Code-Target-Int*
begin

end

125 Preprocessor setup for floats implemented by target language numerals

theory *Code-Target-Numeral-Float*
imports *Float* *Code-Target-Numeral*
begin

```

lemma numeral-float-computation-unfold [code-computation-unfold]:
  ⟨numeral k = Float (int-of-integer (Code-Numeral.positive k)) 0⟩
  ⟨− numeral k = Float (int-of-integer (Code-Numeral.negative k)) 0⟩
  by (simp-all add: Float.compute-float-numeral Float.compute-float-neg-numeral)

end

```

```

theory Complex-Order
  imports Complex-Main
begin

```

```

instantiation complex :: order begin

```

```

definition ⟨x < y ⟷ Re x < Re y ∧ Im x = Im y⟩

```

```

definition ⟨x ≤ y ⟷ Re x ≤ Re y ∧ Im x = Im y⟩

```

```

instance

```

```

  apply standard

```

```

  by (auto simp: less-complex-def less-eq-complex-def complex-eq-iff)

```

```

end

```

```

lemma nonnegative-complex-is-real: ⟨(x::complex) ≥ 0 ⟹ x ∈ ℝ⟩

```

```

  by (simp add: complex-is-Real-iff less-eq-complex-def)

```

```

lemma complex-is-real-iff-compare0: ⟨(x::complex) ∈ ℝ ⟷ x ≤ 0 ∨ x ≥ 0⟩

```

```

  using complex-is-Real-iff less-eq-complex-def by auto

```

```

instance complex :: ordered-comm-ring

```

```

  apply standard

```

```

  by (auto simp: less-complex-def less-eq-complex-def complex-eq-iff mult-left-mono

```

```

  mult-right-mono)

```

```

instance complex :: ordered-real-vector

```

```

  apply standard

```

```

  by (auto simp: less-complex-def less-eq-complex-def mult-left-mono mult-right-mono)

```

```

instance complex :: ordered-cancel-comm-semiring

```

```

  by standard

```

```

end

```

126 Abstract type of association lists with unique keys

```

theory DAList

```

```

imports AList

```

begin

This was based on some existing fragments in the AFP-Collection framework.

126.1 Preliminaries

lemma *distinct-map-fst-filter*:

distinct (map fst xs) \implies distinct (map fst (List.filter P xs))
by (*induct xs*) *auto*

126.2 Type ('key, 'value) alist

typedef ('key, 'value) *alist* = {*xs* :: ('key \times 'value) list. (*distinct* \circ *map fst*) *xs*}
morphisms *impl-of Alist*

proof

show [] \in {*xs*. (*distinct* \circ *map fst*) *xs*}
by *simp*

qed

setup-lifting *type-definition-alist*

lemma *alist-ext*: *impl-of xs = impl-of ys \implies xs = ys*
by (*simp add: impl-of-inject*)

lemma *alist-eq-iff*: *xs = ys \longleftrightarrow impl-of xs = impl-of ys*
by (*simp add: impl-of-inject*)

lemma *impl-of-distinct* [*simp, intro*]: *distinct (map fst (impl-of xs))*
using *impl-of[of xs]* **by** *simp*

lemma *impl-of-Alist*:

$\langle \text{impl-of } (Alist\ xs) = xs \rangle$ **if** $\langle \text{distinct } (map\ fst\ xs) \rangle$
using *Alist-inverse [of xs]* **that** **by** *simp*

lemma *Alist-impl-of* [*code abstype*]: *Alist (impl-of xs) = xs*
by (*rule impl-of-inverse*)

126.3 Primitive operations

lift-definition *lookup* :: ('key, 'value) *alist* \Rightarrow 'key \Rightarrow 'value *option* **is** *map-of* .

lift-definition *empty* :: ('key, 'value) *alist* **is** []
by *simp*

lift-definition *update* :: 'key \Rightarrow 'value \Rightarrow ('key, 'value) *alist* \Rightarrow ('key, 'value) *alist*
is *AList.update*
by (*simp add: distinct-update*)

lift-definition *delete* :: $'key \Rightarrow ('key, 'value) \text{ alist} \Rightarrow ('key, 'value) \text{ alist}$
is *AList.delete*
by (*simp add: distinct-delete*)

lift-definition *map-entry* ::
 $'key \Rightarrow ('value \Rightarrow 'value) \Rightarrow ('key, 'value) \text{ alist} \Rightarrow ('key, 'value) \text{ alist}$
is *AList.map-entry*
by (*simp add: distinct-map-entry*)

lift-definition *filter* :: $('key \times 'value \Rightarrow \text{bool}) \Rightarrow ('key, 'value) \text{ alist} \Rightarrow ('key, 'value) \text{ alist}$
is *List.filter*
by (*simp add: distinct-map-fst-filter*)

lift-definition *map-default* ::
 $'key \Rightarrow 'value \Rightarrow ('value \Rightarrow 'value) \Rightarrow ('key, 'value) \text{ alist} \Rightarrow ('key, 'value) \text{ alist}$
is *AList.map-default*
by (*simp add: distinct-map-default*)

126.4 Abstract operation properties

lemma *lookup-empty* [*simp*]: *lookup empty k = None*
by (*simp add: empty-def lookup-def Alist-inverse*)

lemma *lookup-update*:
 $\text{lookup} (\text{update } k1 \ v \ xs) \ k2 = (\text{if } k1 = k2 \text{ then } \text{Some } v \text{ else } \text{lookup } xs \ k2)$
by(*transfer*)(*simp add: update-conv'*)

lemma *lookup-update-eq* [*simp*]:
 $k1 = k2 \implies \text{lookup} (\text{update } k1 \ v \ xs) \ k2 = \text{Some } v$
by(*simp add: lookup-update*)

lemma *lookup-update-neq* [*simp*]:
 $k1 \neq k2 \implies \text{lookup} (\text{update } k1 \ v \ xs) \ k2 = \text{lookup } xs \ k2$
by(*simp add: lookup-update*)

lemma *update-update-eq* [*simp*]:
 $k1 = k2 \implies \text{update } k2 \ v2 (\text{update } k1 \ v1 \ xs) = \text{update } k2 \ v2 \ xs$
by(*transfer*)(*simp add: update-conv'*)

lemma *lookup-delete* [*simp*]: *lookup (delete k al) = (lookup al)(k := None)*
by (*simp add: lookup-def delete-def Alist-inverse distinct-delete delete-conv'*)

126.5 Further operations

126.5.1 Equality

instantiation *alist* :: (*equal, equal*) *equal*
begin

definition *HOL.equal* (*xs* :: ('a, 'b) alist) *ys* == *impl-of xs = impl-of ys*

instance

by *standard* (*simp add: equal-alist-def impl-of-inject*)

end

126.5.2 Size

instantiation *alist* :: (type, type) size

begin

definition *size* (*al* :: ('a, 'b) alist) = *length (impl-of al)*

instance ..

end

126.6 Quickcheck generators

context

includes *state-combinator-syntax* and *term-syntax*

begin

definition

valterm-empty :: ('key :: typerep, 'value :: typerep) alist × (unit ⇒ Code-Evaluation.term)
where *valterm-empty* = Code-Evaluation.valtermify empty

definition

valterm-update :: 'key :: typerep × (unit ⇒ Code-Evaluation.term) ⇒
'value :: typerep × (unit ⇒ Code-Evaluation.term) ⇒
('key, 'value) alist × (unit ⇒ Code-Evaluation.term) ⇒
('key, 'value) alist × (unit ⇒ Code-Evaluation.term) **where**
[*code-unfold*]: *valterm-update* *k v a* = Code-Evaluation.valtermify update {·} *k* {·}
v {·} *a*

fun *random-aux-alist*

where

random-aux-alist *i j* =
(if *i* = 0 then Pair *valterm-empty*
else Quickcheck-Random.collapse
(Random.select-weight
[(*i*, Quickcheck-Random.random *j* ○→ (λ*k*. Quickcheck-Random.random *j*
○→
(λ*v*. random-aux-alist (*i* - 1) *j* ○→ (λ*a*. Pair (valterm-update *k v a*))))),
(1, Pair *valterm-empty*)]))

end

instantiation *alist* :: (random, random) random

begin

definition *random-alist*

where

random-alist i = random-aux-alist i i

instance ..

end

instantiation *alist* :: (*exhaustive*, *exhaustive*) *exhaustive*

begin

fun *exhaustive-alist* ::

((*'a*, *'b*) *alist* \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*

where

exhaustive-alist f i =

(*if i = 0 then None*

else

case f empty of

Some ts \Rightarrow Some ts

| *None \Rightarrow*

exhaustive-alist

($\lambda a.$ *Quickcheck-Exhaustive.exhaustive*

($\lambda k.$ *Quickcheck-Exhaustive.exhaustive* ($\lambda v.$ *f (update k v a)*) (*i - 1*))

(*i - 1*))

(*i - 1*))

instance ..

end

instantiation *alist* :: (*full-exhaustive*, *full-exhaustive*) *full-exhaustive*

begin

fun *full-exhaustive-alist* ::

((*'a*, *'b*) *alist* \times (*unit* \Rightarrow *term*) \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow

(*bool* \times *term list*) *option*

where

full-exhaustive-alist f i =

(*if i = 0 then None*

else

case f valterm-empty of

Some ts \Rightarrow Some ts

| *None \Rightarrow*

full-exhaustive-alist

($\lambda a.$

Quickcheck-Exhaustive.full-exhaustive

($\lambda k.$ *Quickcheck-Exhaustive.full-exhaustive* ($\lambda v.$ *f (valterm-update k v*

$a)) (i - 1))$
 $(i - 1))$
 $(i - 1))$

instance ..

end

127 alist is a BNF

lift-bnf (*dead* 'k, *set*: 'v) *alist* [*wits*: [] :: ('k × 'v) list] **for** *map*: map *rel*: rel
by *auto*

hide-const *valterm-empty valterm-update random-aux-alist*

hide-fact (**open**) *lookup-def empty-def update-def delete-def map-entry-def filter-def map-default-def*

hide-const (**open**) *impl-of lookup empty update delete map-entry filter map-default map set rel*

end

128 Multisets partially implemented by association lists

theory *DAList-Multiset*

imports *Multiset DAList*

begin

Raw operations on lists

definition *join-raw* ::

$('key \Rightarrow 'val \times 'val \Rightarrow 'val) \Rightarrow$

$('key \times 'val) list \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list$

where *join-raw* *f xs ys* = *foldr* $(\lambda(k, v). \text{map-default } k \ v \ (\lambda v'. f \ k \ (v', v))) \ ys \ xs$

lemma *join-raw-Nil* [*simp*]: *join-raw* *f xs* [] = *xs*

by (*simp add: join-raw-def*)

lemma *join-raw-Cons* [*simp*]:

join-raw *f xs* ((*k*, *v*) # *ys*) = *map-default* *k v* $(\lambda v'. f \ k \ (v', v))$ (*join-raw* *f xs ys*)

by (*simp add: join-raw-def*)

lemma *map-of-join-raw*:

assumes *distinct* (*map fst ys*)

shows *map-of* (*join-raw* *f xs ys*) *x* =

(*case map-of xs x of*

None \Rightarrow *map-of* *ys x*

```

    | Some v  $\Rightarrow$  (case map-of ys x of None  $\Rightarrow$  Some v | Some v'  $\Rightarrow$  Some (f x (v,
v'))))
  using assms
  apply (induct ys)
  apply (auto simp add: map-of-map-default split: option.split)
  apply (metis map-of-eq-None-iff option.simps(2) weak-map-of-SomeI)
  apply (metis Some-eq-map-of-iff map-of-eq-None-iff option.simps(2))
  done

```

```

lemma distinct-join-row:
  assumes distinct (map fst xs)
  shows distinct (map fst (join-row f xs ys))
  using assms
proof (induct ys)
  case Nil
  then show ?case by simp
next
  case (Cons y ys)
  then show ?case by (cases y) (simp add: distinct-map-default)
qed

```

definition *subtract-entries-row* xs ys = foldr ($\lambda(k, v). \text{AList.map-entry } k (\lambda v'. v' - v)$) ys xs

```

lemma map-of-subtract-entries-row:
  assumes distinct (map fst ys)
  shows map-of (subtract-entries-row xs ys) x =
    (case map-of xs x of
      None  $\Rightarrow$  None
    | Some v  $\Rightarrow$  (case map-of ys x of None  $\Rightarrow$  Some v | Some v'  $\Rightarrow$  Some (v - v')))
  using assms
  unfolding subtract-entries-row-def
  apply (induct ys)
  apply auto
  apply (simp split: option.split)
  apply (simp add: map-of-map-entry)
  apply (auto split: option.split)
  apply (metis map-of-eq-None-iff option.simps(3) option.simps(4))
  apply (metis map-of-eq-None-iff option.simps(4) option.simps(5))
  done

```

```

lemma distinct-subtract-entries-row:
  assumes distinct (map fst xs)
  shows distinct (map fst (subtract-entries-row xs ys))
  using assms
  unfolding subtract-entries-row-def
  by (induct ys) (auto simp add: distinct-map-entry)

```

Operations on alists with distinct keys

lift-definition $join :: ('a \Rightarrow 'b \times 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{ alist} \Rightarrow ('a, 'b) \text{ alist} \Rightarrow ('a, 'b) \text{ alist}$
is *join-raw*
by (*simp add: distinct-join-raw*)

lift-definition $subtract\text{-}entries :: ('a, ('b :: \text{minus})) \text{ alist} \Rightarrow ('a, 'b) \text{ alist} \Rightarrow ('a, 'b) \text{ alist}$
is *subtract-entries-raw*
by (*simp add: distinct-subtract-entries-raw*)

Implementing multisets by means of association lists

definition $count\text{-}of :: ('a \times \text{nat}) \text{ list} \Rightarrow 'a \Rightarrow \text{nat}$
where $count\text{-}of \text{ xs } x = (\text{case map-of xs } x \text{ of } None \Rightarrow 0 \mid \text{Some } n \Rightarrow n)$

lemma *count-of-multiset: finite {x. 0 < count-of xs x}*
proof –
let $?A = \{x::'a. 0 < (\text{case map-of xs } x \text{ of } None \Rightarrow 0::\text{nat} \mid \text{Some } n \Rightarrow n)\}$
have $?A \subseteq \text{dom } (\text{map-of xs})$
proof
fix x
assume $x \in ?A$
then have $0 < (\text{case map-of xs } x \text{ of } None \Rightarrow 0::\text{nat} \mid \text{Some } n \Rightarrow n)$
by *simp*
then have $\text{map-of xs } x \neq None$
by (*cases map-of xs x auto*)
then show $x \in \text{dom } (\text{map-of xs})$
by *auto*
qed
with *finite-dom-map-of [of xs]* **have** *finite ?A*
by (*auto intro: finite-subset*)
then show *?thesis*
by (*simp add: count-of-def fun-eq-iff*)
qed

lemma *count-simps [simp]:*
 $count\text{-}of [] = (\lambda_. 0)$
 $count\text{-}of ((x, n) \# xs) = (\lambda y. \text{if } x = y \text{ then } n \text{ else } count\text{-}of \text{ xs } y)$
by (*simp-all add: count-of-def fun-eq-iff*)

lemma *count-of-empty: $x \notin \text{fst } \text{'set xs} \implies count\text{-}of \text{ xs } x = 0$*
by (*induct xs (simp-all add: count-of-def)*)

lemma *count-of-filter: $count\text{-}of (\text{List.filter } (P \circ \text{fst}) \text{ xs}) x = (\text{if } P \text{ } x \text{ then } count\text{-}of \text{ xs } x \text{ else } 0)$*
by (*induct xs auto*)

lemma *count-of-map-default [simp]:*
 $count\text{-}of (\text{map-default } x \text{ } b (\lambda x. x + b) \text{ xs}) y =$
 $(\text{if } x = y \text{ then } count\text{-}of \text{ xs } x + b \text{ else } count\text{-}of \text{ xs } y)$

unfolding *count-of-def* **by** (*simp add: map-of-map-default split: option.split*)

lemma *count-of-join-row*:

distinct (map fst ys) \implies

count-of xs x + count-of ys x = count-of (join-row ($\lambda x (x, y). x + y$) xs ys) x

unfolding *count-of-def* **by** (*simp add: map-of-join-row split: option.split*)

lemma *count-of-subtract-entries-row*:

distinct (map fst ys) \implies

count-of xs x - count-of ys x = count-of (subtract-entries-row xs ys) x

unfolding *count-of-def* **by** (*simp add: map-of-subtract-entries-row split: option.split*)

Code equations for multiset operations

definition *Bag* :: ('a, nat) alist \Rightarrow 'a multiset

where *Bag xs* = *Abs-multiset (count-of (DAList.impl-of xs))*

code-datatype *Bag*

lemma *count-Bag* [*simp, code*]: *count (Bag xs) = count-of (DAList.impl-of xs)*

by (*simp add: Bag-def count-of-multiset*)

lemma *Bag-eq*:

$\langle Bag\ ms = (\sum (a, n) \leftarrow alist.impl\ of\ ms.\ replicate\ mset\ n\ a) \rangle$

for *ms* :: $\langle ('a, nat)\ alist \rangle$

proof –

have *: $\langle count\ of\ xs\ a = count\ (\sum (a, n) \leftarrow xs.\ replicate\ mset\ n\ a)\ a \rangle$

if $\langle distinct\ (map\ fst\ xs) \rangle$

for *xs* **and** *a* :: 'a

using that proof (*induction xs*)

case *Nil*

then show ?*case*

by *simp*

next

case (*Cons xn xs*)

then show ?*case* **by** (*cases xn*)

(*auto simp add: count-eq-zero-iff if-split-mem2 image-iff*)

qed

show ?*thesis*

by (*rule multiset-eqI (simp add: *)*)

qed

lemma *Mempty-Bag* [*code*]: $\{\#\} = Bag\ (DAList.empty)$

by (*simp add: multiset-eq-iff alist.Alist-inverse DAList.empty-def*)

lift-definition *is-empty-Bag-impl* :: ('a, nat) alist \Rightarrow bool **is**

$\lambda xs.\ list\ all\ (\lambda x.\ snd\ x = 0)\ xs$.

lemma *is-empty-Bag* [*code*]: *Multiset.is-empty (Bag xs) \longleftrightarrow is-empty-Bag-impl xs*

proof –

have *Multiset.is-empty* (*Bag xs*) $\longleftrightarrow (\forall x. \text{count } (\text{Bag } xs) \ x = 0)$
unfolding *Multiset.is-empty-def multiset-eq-iff* **by** *simp*
also have $\dots \longleftrightarrow (\forall x \in \text{fst } ' \text{set } (\text{alist.impl-of } xs). \text{count } (\text{Bag } xs) \ x = 0)$
proof (*intro iffI allI ballI*)
fix *x* **assume** *A*: $\forall x \in \text{fst } ' \text{set } (\text{alist.impl-of } xs). \text{count } (\text{Bag } xs) \ x = 0$
thus $\text{count } (\text{Bag } xs) \ x = 0$
proof (*cases x ∈ fst ' set (alist.impl-of xs)*)
case *False*
thus *?thesis* **by** (*force simp: count-of-def split: option.splits*)
qed (*insert A, auto*)
qed *simp-all*
also have $\dots \longleftrightarrow \text{list-all } (\lambda x. \text{snd } x = 0) (\text{alist.impl-of } xs)$
by (*auto simp: count-of-def list-all-def*)
finally show *?thesis* **by** (*simp add: is-empty-Bag-impl.rep-eq*)
qed

lemma *union-Bag* [*code*]: $\text{Bag } xs + \text{Bag } ys = \text{Bag } (\text{join } (\lambda x \ (n1, n2). \ n1 + n2) \ xs \ ys)$
by (*rule multiset-eqI*)
(simp add: count-of-join-raw alist.Alist-inverse distinct-join-raw join-def)

lemma *add-mset-Bag* [*code*]: $\text{add-mset } x \ (\text{Bag } xs) = \text{Bag } (\text{join } (\lambda x \ (n1, n2). \ n1 + n2) \ (\text{DAList.update } x \ 1 \ \text{DAList.empty}) \ xs)$
unfolding *add-mset-add-single[of x Bag xs] union-Bag[symmetric]*
by (*simp add: multiset-eq-iff update.rep-eq empty.rep-eq*)

lemma *minus-Bag* [*code*]: $\text{Bag } xs - \text{Bag } ys = \text{Bag } (\text{subtract-entries } xs \ ys)$
by (*rule multiset-eqI*)
(simp add: count-of-subtract-entries-raw alist.Alist-inverse distinct-subtract-entries-raw subtract-entries-def)

lemma *filter-Bag* [*code*]: $\text{filter-mset } P \ (\text{Bag } xs) = \text{Bag } (\text{DAList.filter } (P \circ \text{fst}) \ xs)$
by (*rule multiset-eqI*) *(simp add: count-of-filter DAList.filter.rep-eq)*

lemma *mset-eq* [*code*]: $\text{HOL.equal } (m1 :: 'a :: \text{equal multiset}) \ m2 \longleftrightarrow m1 \subseteq\# m2 \wedge m2 \subseteq\# m1$
by (*metis equal-multiset-def subset-mset.order-eq-iff*)

By default the code for $<$ is $(xs < ys) = (xs \leq ys \wedge xs \neq ys)$. With equality implemented by \leq , this leads to three calls of \leq . Here is a more efficient version:

lemma *mset-less*[*code*]: $xs \subset\# (ys :: 'a \text{ multiset}) \longleftrightarrow xs \subseteq\# ys \wedge \neg ys \subseteq\# xs$
by (*rule subset-mset.less-le-not-le*)

lemma *mset-less-eq-Bag0*:
 $\text{Bag } xs \subseteq\# A \longleftrightarrow (\forall (x, n) \in \text{set } (\text{DAList.impl-of } xs). \text{count-of } (\text{DAList.impl-of } xs) \ x \leq \text{count } A \ x)$

```

    (is ?lhs  $\longleftrightarrow$  ?rhs)
  proof
    assume ?lhs
    then show ?rhs by (auto simp add: subseteq-mset-def)
  next
    assume ?rhs
    show ?lhs
    proof (rule mset-subset-eqI)
      fix x
      from ⟨?rhs⟩ have count-of (DAList.impl-of xs) x  $\leq$  count A x
      by (cases x  $\in$  fst ‘ set (DAList.impl-of xs)) (auto simp add: count-of-empty)
      then show count (Bag xs) x  $\leq$  count A x by (simp add: subset-mset-def)
    qed
  qed

lemma mset-less-eq-Bag [code]:
  Bag xs  $\subseteq\#$  (A :: 'a multiset)  $\longleftrightarrow$  ( $\forall (x, n) \in$  set (DAList.impl-of xs).  $n \leq$  count
  A x)
proof –
  {
    fix x n
    assume (x,n)  $\in$  set (DAList.impl-of xs)
    then have count-of (DAList.impl-of xs) x = n
    proof transfer
      fix x n
      fix xs :: ('a  $\times$  nat) list
      show (distinct  $\circ$  map fst) xs  $\implies$  (x, n)  $\in$  set xs  $\implies$  count-of xs x = n
      proof (induct xs)
        case Nil
        then show ?case by simp
      next
        case (Cons ym ys)
        obtain y m where ym: ym = (y,m) by force
        note Cons = Cons[unfolded ym]
        show ?case
        proof (cases x = y)
          case False
          with Cons show ?thesis
            unfolding ym by auto
        next
          case True
          with Cons(2–3) have m = n by force
          with True show ?thesis
            unfolding ym by auto
        qed
      qed
    qed
  }
then show ?thesis

```

unfolding *mset-less-eq-Bag0* **by** *auto*
qed

declare *inter-mset-def* [*code*]
declare *union-mset-def* [*code*]
declare *mset.simps* [*code*]

fun *fold-impl* :: (*'a* \Rightarrow *nat* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow *'b* \Rightarrow (*'a* \times *nat*) *list* \Rightarrow *'b*
where
 fold-impl *fn* *e* ((*a,n*) # *ms*) = (*fold-impl* *fn* ((*fn* *a* *n*) *e*) *ms*)
 | *fold-impl* *fn* *e* [] = *e*

context
begin

qualified definition *fold* :: (*'a* \Rightarrow *nat* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow *'b* \Rightarrow (*'a*, *nat*) *alist* \Rightarrow *'b*
 where *fold* *f* *e* *al* = *fold-impl* *f* *e* (*DAList.impl-of* *al*)

end

context *comp-fun-commute*
begin

lemma *DAList-Multiset-fold*:

assumes *fn*: $\bigwedge a\ n\ x. \text{fn } a\ n\ x = (f\ a\ \sim n)\ x$
 shows *fold-mset* *f* *e* (*Bag* *al*) = *DAList-Multiset.fold* *fn* *e* *al*
 unfolding *DAList-Multiset.fold-def*
proof (*induct* *al*)
 fix *ys*
 let *?inv* = {*xs* :: (*'a* \times *nat*) *list*. (*distinct* \circ *map fst*) *xs*}
 note *cs*[*simp del*] = *count-simps*
 have *count*[*simp*]: $\bigwedge x. \text{count } (\text{Abs-multiset } (\text{count-of } x)) = \text{count-of } x$
 by (*rule Abs-multiset-inverse*) (*simp add: count-of-multiset*)
 assume *ys*: *ys* \in *?inv*
 then show *fold-mset* *f* *e* (*Bag* (*Alist* *ys*)) = *fold-impl* *fn* *e* (*DAList.impl-of* (*Alist* *ys*))
 unfolding *Bag-def* **unfolding** *Alist-inverse*[*OF* *ys*]
 proof (*induct* *ys* *arbitrary: e* *rule: list.induct*)
 case *Nil*
 show *?case*
 by (*rule trans*[*OF* *arg-cong*[*of* - {#} *fold-mset* *f* *e*, *OF* *multiset-eqI*]])
 (*auto*, *simp add: cs*)
 next
 case (*Cons* *pair* *ys* *e*)
 obtain *a* *n* **where** *pair*: *pair* = (*a,n*)
 by *force*
 from *fn*[*of* *a* *n*] **have** [*simp*]: *fn* *a* *n* = (*f* *a* $\sim n$)
 by *auto*

```

have inv: ys ∈ ?inv
  using Cons(2) by auto
note IH = Cons(1)[OF inv]
define Ys where Ys = Abs-multiset (count-of ys)
have id: Abs-multiset (count-of ((a, n) # ys)) = (((+) {# a #})  $\widetilde{\sim}$  n) Ys
  unfolding Ys-def
proof (rule multiset-eqI, unfold count)
  fix c
  show count-of ((a, n) # ys) c =
    count (((+) {# a #})  $\widetilde{\sim}$  n) (Abs-multiset (count-of ys)) c (is ?l = ?r)
  proof (cases c = a)
    case False
    then show ?thesis
      unfolding cs by (induct n) auto
  next
    case True
    then have ?l = n by (simp add: cs)
    also have n = ?r unfolding True
    proof (induct n)
      case 0
      from Cons(2)[unfolded pair] have a ∉ fst ‘ set ys by auto
      then show ?case by (induct ys) (simp, auto simp: cs)
    next
      case Suc
      then show ?case by simp
    qed
    finally show ?thesis .
  qed
qed
show ?case
  unfolding pair
  apply (simp add: IH[symmetric])
  unfolding id Ys-def[symmetric]
  apply (induct n)
  apply (auto simp: fold-mset-fun-left-comm[symmetric])
  done
qed
qed

end

context
begin

private lift-definition single-alist-entry :: 'a ⇒ 'b ⇒ ('a, 'b) alist is λa b. [(a,
b)]
  by auto

lemma image-mset-Bag [code]:

```



```

image-mset f (Bag ms) =
  DAList-Multiset.fold (λa n m. Bag (single-alist-entry (f a) n) + m) {#} ms
unfolding image-mset-def
proof (rule comp-fun-commute.DAList-Multiset-fold, unfold-locales, (auto simp:
ac-simps)[1])
  fix a n m
  show Bag (single-alist-entry (f a) n) + m = ((add-mset ∘ f) a  $\frown$  n) m (is ?l
= ?r)
  proof (rule multiset-eqI)
    fix x
    have count ?r x = (if x = f a then n + count m x else count m x)
    by (induct n) auto
    also have ... = count ?l x
    by (simp add: single-alist-entry.rep-eq)
    finally show count ?l x = count ?r x ..
  qed
qed

end

```

— we cannot use $\lambda a n. (+) (a * n)$ for folding, since $(*)$ is not defined in *comm-monoid-add*

```

lemma sum-mset-Bag[code]: sum-mset (Bag ms) = DAList-Multiset.fold (λa n.
((+) a)  $\frown$  n) 0 ms
  unfolding sum-mset.eq-fold
  apply (rule comp-fun-commute.DAList-Multiset-fold)
  apply unfold-locales
  apply (auto simp: ac-simps)
  done

```

— we cannot use $\lambda a n. (*) (a \frown n)$ for folding, since (\frown) is not defined in *comm-monoid-mult*

```

lemma prod-mset-Bag[code]: prod-mset (Bag ms) = DAList-Multiset.fold (λa n.
((*) a)  $\frown$  n) 1 ms
  unfolding prod-mset.eq-fold
  apply (rule comp-fun-commute.DAList-Multiset-fold)
  apply unfold-locales
  apply (auto simp: ac-simps)
  done

```

```

lemma size-fold: size A = fold-mset (λ-. Suc) 0 A (is - = fold-mset ?f -)

```

```

proof -
  interpret comp-fun-commute ?f by standard auto
  show ?thesis by (induct A) auto
qed

```

```

lemma size-Bag[code]: size (Bag ms) = DAList-Multiset.fold (λa n. (+) n) 0 ms
  unfolding size-fold
proof (rule comp-fun-commute.DAList-Multiset-fold, unfold-locales, simp)
  fix a n x
  show n + x = (Suc  $\frown$  n) x

```

```

    by (induct n) auto
qed

```

```

lemma set-mset-fold: set-mset A = fold-mset insert {} A (is - = fold-mset ?f - -)
proof -
  interpret comp-fun-commute ?f by standard auto
  show ?thesis by (induct A) auto
qed

```

```

lemma set-mset-Bag[code]:
  set-mset (Bag ms) = DAList-Multiset.fold (λa n. (if n = 0 then (λm. m) else
insert a)) {} ms
  unfolding set-mset-fold
proof (rule comp-fun-commute.DAList-Multiset-fold, unfold-locales, (auto simp:
ac-simps)[I])
  fix a n x
  show (if n = 0 then λm. m else insert a) x = (insert a  $\smile$  n) x (is ?l n = ?r n)
  proof (cases n)
    case 0
    then show ?thesis by simp
  next
    case (Suc m)
    then have ?l n = insert a x by simp
    moreover have ?r n = insert a x unfolding Suc by (induct m) auto
    ultimately show ?thesis by auto
  qed
qed

```

```

lemma sorted-list-of-multiset-Bag [code]:
  ⟨sorted-list-of-multiset (Bag ms) = concat (map (λ(a, n). replicate n a)
(sort-key fst (DAList.impl-of ms)))⟩ (is ⟨?lhs = ?rhs⟩)
proof -
  have *: ⟨sorted (concat (map (λ(a, n). replicate n a) ans))⟩
    if ⟨sorted (map fst ans)⟩
    for ans :: ⟨('a × nat) list⟩
    using that by (induction ans) (auto simp add: sorted-append)
  have ⟨mset ?rhs = mset ?lhs⟩
    by (simp add: Bag-eq mset-concat comp-def split-def flip: sum-mset-sum-list)
  moreover have ⟨sorted ?rhs⟩
    by (rule *) simp
  ultimately have ⟨sort ?lhs = ?rhs⟩
    by (rule properties-for-sort)
  then show ?thesis
    by simp
qed

```

```

instantiation multiset :: (exhaustive) exhaustive
begin

```

definition *exhaustive-multiset* ::
 (*'a multiset* \Rightarrow (*bool* \times *term list*) *option*) \Rightarrow *natural* \Rightarrow (*bool* \times *term list*) *option*
where *exhaustive-multiset* *f i* = *Quickcheck-Exhaustive.exhaustive* ($\lambda xs. f$ (*Bag* *xs*)) *i*

instance ..

end

end

129 Implementation of Red-Black Trees

theory *RBT-Impl*

imports *Main*

begin

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

129.1 Datatype of RB trees

datatype *color* = *R* | *B*

datatype (*'a, 'b*) *rbt* = *Empty* | *Branch color ('a, 'b) rbt 'a 'b ('a, 'b) rbt*

lemma *rbt-cases*:

obtains (*Empty*) *t = Empty*
 | (*Red*) *l k v r* **where** *t = Branch R l k v r*
 | (*Black*) *l k v r* **where** *t = Branch B l k v r*

proof (*cases t*)

case *Empty* **with that show thesis by** *blast*

next

case (*Branch c*) **with that show thesis by** (*cases c*) *blast+*

qed

129.2 Tree properties

129.2.1 Content of a tree

primrec *entries* :: (*'a, 'b*) *rbt* \Rightarrow (*'a* \times *'b*) *list*

where

entries Empty = []

| *entries (Branch - l k v r)* = *entries l* @ (*k, v*) # *entries r*

abbreviation (*input*) *entry-in-tree* :: *'a* \Rightarrow *'b* \Rightarrow (*'a, 'b*) *rbt* \Rightarrow *bool*

where

entry-in-tree k v t \equiv (*k, v*) \in *set (entries t)*

definition $keys :: ('a, 'b) rbt \Rightarrow 'a \text{ list}$ **where**
 $keys\ t = map\ fst\ (entries\ t)$

lemma $keys-simps$ $[simp]$:
 $keys\ Empty = []$
 $keys\ (Branch\ c\ l\ k\ v\ r) = keys\ l @ k \# keys\ r$
by $(simp-all\ add: keys-def)$

lemma $entry-in-tree-keys$:
assumes $(k, v) \in set\ (entries\ t)$
shows $k \in set\ (keys\ t)$
proof –
from $assms$ **have** $fst\ (k, v) \in fst\ 'set\ (entries\ t)$ **by** $(rule\ imageI)$
then show $?thesis$ **by** $(simp\ add: keys-def)$
qed

lemma $keys-entries$:
 $k \in set\ (keys\ t) \longleftrightarrow (\exists v. (k, v) \in set\ (entries\ t))$
by $(auto\ intro: entry-in-tree-keys)\ (auto\ simp\ add: keys-def)$

lemma $non-empty-rbt-keys$:
 $t \neq rbt.Empty \Longrightarrow keys\ t \neq []$
by $(cases\ t)\ simp-all$

129.2.2 Search tree properties

context ord **begin**

definition $rbt-less :: 'a \Rightarrow ('a, 'b) rbt \Rightarrow bool$
where
 $rbt-less-prop: rbt-less\ k\ t \longleftrightarrow (\forall x \in set\ (keys\ t). x < k)$

abbreviation $rbt-less-symbol$ **(infix** $\langle | \langle \rangle$ 50)
where $t | \langle \rangle x \equiv rbt-less\ x\ t$

definition $rbt-greater :: 'a \Rightarrow ('a, 'b) rbt \Rightarrow bool$ **(infix** $\langle \rangle | \rangle$ 50)
where
 $rbt-greater-prop: rbt-greater\ k\ t = (\forall x \in set\ (keys\ t). k < x)$

lemma $rbt-less-simps$ $[simp]$:
 $Empty | \langle \rangle k = True$
 $Branch\ c\ lt\ kt\ v\ rt | \langle \rangle k \longleftrightarrow kt < k \wedge lt | \langle \rangle k \wedge rt | \langle \rangle k$
by $(auto\ simp\ add: rbt-less-prop)$

lemma $rbt-greater-simps$ $[simp]$:
 $k \langle \rangle | Empty = True$
 $k \langle \rangle | (Branch\ c\ lt\ kt\ v\ rt) \longleftrightarrow k < kt \wedge k \langle \rangle | lt \wedge k \langle \rangle | rt$
by $(auto\ simp\ add: rbt-greater-prop)$

lemmas *rbt-ord-props* = *rbt-less-prop* *rbt-greater-prop*

lemmas *rbt-greater-nit* = *rbt-greater-prop* *entry-in-tree-keys*

lemmas *rbt-less-nit* = *rbt-less-prop* *entry-in-tree-keys*

lemma (*in order*)

shows *rbt-less-eq-trans*: $l \mid\!\!\ll u \implies u \leq v \implies l \mid\!\!\ll v$

and *rbt-less-trans*: $t \mid\!\!\ll x \implies x < y \implies t \mid\!\!\ll y$

and *rbt-greater-eq-trans*: $u \leq v \implies v \ll\!\!\mid r \implies u \ll\!\!\mid r$

and *rbt-greater-trans*: $x < y \implies y \ll\!\!\mid t \implies x \ll\!\!\mid t$

by (*auto simp: rbt-ord-props*)

primrec *rbt-sorted* :: (*'a*, *'b*) *rbt* \Rightarrow *bool*

where

rbt-sorted *Empty* = *True*

| *rbt-sorted* (*Branch* *c l k v r*) = $(l \mid\!\!\ll k \wedge k \ll\!\!\mid r \wedge \text{rbt-sorted } l \wedge \text{rbt-sorted } r)$

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:

rbt-sorted *t* \implies *List.sorted* (*map fst* (*entries* *t*))

by (*induct* *t*) (*force simp: sorted-append rbt-ord-props dest!: entry-in-tree-keys*)⁺

lemma *distinct-entries*:

rbt-sorted *t* \implies *distinct* (*map fst* (*entries* *t*))

by (*induct* *t*) (*force simp: sorted-append rbt-ord-props dest!: entry-in-tree-keys*)⁺

lemma *distinct-keys*:

rbt-sorted *t* \implies *distinct* (*keys* *t*)

by (*simp add: distinct-entries keys-def*)

129.2.3 Tree lookup

primrec (*in ord*) *rbt-lookup* :: (*'a*, *'b*) *rbt* \Rightarrow *'a* \multimap *'b*

where

rbt-lookup *Empty* *k* = *None*

| *rbt-lookup* (*Branch* - *l x y r*) *k* =

(if $k < x$ then *rbt-lookup* *l* *k* else if $x < k$ then *rbt-lookup* *r* *k* else *Some* *y*)

lemma *rbt-lookup-keys*: *rbt-sorted* *t* \implies *dom* (*rbt-lookup* *t*) = *set* (*keys* *t*)

by (*induct* *t*) (*auto simp: dom-def rbt-greater-prop rbt-less-prop*)

lemma *dom-rbt-lookup-Branch*:

rbt-sorted (*Branch* *c t1 k v t2*) \implies

dom (*rbt-lookup* (*Branch* *c t1 k v t2*))

= *Set.insert* *k* (*dom* (*rbt-lookup* *t1*) \cup *dom* (*rbt-lookup* *t2*))

proof –

```

  assume rbt-sorted (Branch c t1 k v t2)
  then show ?thesis by (simp add: rbt-lookup-keys)
qed

```

```

lemma finite-dom-rbt-lookup [simp, intro!]: finite (dom (rbt-lookup t))
proof (induct t)
  case Empty then show ?case by simp
next
  case (Branch color t1 a b t2)
  let ?A = Set.insert a (dom (rbt-lookup t1)  $\cup$  dom (rbt-lookup t2))
  have dom (rbt-lookup (Branch color t1 a b t2))  $\subseteq$  ?A by (auto split: if-split-asm)
  moreover from Branch have finite (insert a (dom (rbt-lookup t1)  $\cup$  dom
(rbt-lookup t2))) by simp
  ultimately show ?case by (rule finite-subset)
qed

```

```
end
```

```
context ord begin
```

```

lemma rbt-lookup-rbt-less[simp]: t  $\ll$  k  $\implies$  rbt-lookup t k = None
by (induct t) auto

```

```

lemma rbt-lookup-rbt-greater[simp]: k  $\ll$  t  $\implies$  rbt-lookup t k = None
by (induct t) auto

```

```

lemma rbt-lookup-Empty: rbt-lookup Empty = Map.empty
by (rule ext) simp

```

```
end
```

```
context linorder begin
```

```

lemma map-of-entries:
  rbt-sorted t  $\implies$  map-of (entries t) = rbt-lookup t
proof (induct t)
  case Empty thus ?case by (simp add: rbt-lookup-Empty)
next
  case (Branch c t1 k v t2)
  have rbt-lookup (Branch c t1 k v t2) = rbt-lookup t2 ++ [k  $\mapsto$  v] ++ rbt-lookup
t1
  proof (rule ext)
    fix x
    from Branch have RBT-SORTED: rbt-sorted (Branch c t1 k v t2) by simp
    let ?thesis = rbt-lookup (Branch c t1 k v t2) x = (rbt-lookup t2 ++ [k  $\mapsto$  v]
++ rbt-lookup t1) x

```

```

    have DOM-T1:  $\neg k'. k' \in \text{dom} (\text{rbt-lookup } t1) \implies k > k'$ 
    proof -

```

```

    fix k'
    from RBT-SORTED have t1 |« k by simp
    with rbt-less-prop have  $\forall k' \in \text{set } (\text{keys } t1). k > k'$  by auto
    moreover assume  $k' \in \text{dom } (\text{rbt-lookup } t1)$ 
    ultimately show  $k > k'$  using rbt-lookup-keys RBT-SORTED by auto
  qed

  have DOM-T2:  $\forall k'. k' \in \text{dom } (\text{rbt-lookup } t2) \implies k < k'$ 
  proof -
    fix k'
    from RBT-SORTED have k «| t2 by simp
    with rbt-greater-prop have  $\forall k' \in \text{set } (\text{keys } t2). k < k'$  by auto
    moreover assume  $k' \in \text{dom } (\text{rbt-lookup } t2)$ 
    ultimately show  $k < k'$  using rbt-lookup-keys RBT-SORTED by auto
  qed

  {
    assume C:  $x < k$ 
    hence rbt-lookup (Branch c t1 k v t2) x = rbt-lookup t1 x by simp
    moreover from C have  $x \notin \text{dom } [k \mapsto v]$  by simp
    moreover have  $x \notin \text{dom } (\text{rbt-lookup } t2)$ 
    proof
      assume  $x \in \text{dom } (\text{rbt-lookup } t2)$ 
      with DOM-T2 have  $k < x$  by blast
      with C show False by simp
    qed
    ultimately have ?thesis by (simp add: map-add-upd-left map-add-dom-app-simps)
  } moreover {
    assume [simp]:  $x = k$ 
    hence rbt-lookup (Branch c t1 k v t2) x =  $[k \mapsto v] x$  by simp
    moreover have  $x \notin \text{dom } (\text{rbt-lookup } t1)$ 
    proof
      assume  $x \in \text{dom } (\text{rbt-lookup } t1)$ 
      with DOM-T1 have  $k > x$  by blast
      thus False by simp
    qed
    ultimately have ?thesis by (simp add: map-add-upd-left map-add-dom-app-simps)
  } moreover {
    assume C:  $x > k$ 
    hence rbt-lookup (Branch c t1 k v t2) x = rbt-lookup t2 x by (simp add:
less-not-sym[of k x])
    moreover from C have  $x \notin \text{dom } [k \mapsto v]$  by simp
    moreover have  $x \notin \text{dom } (\text{rbt-lookup } t1)$  proof
      assume  $x \in \text{dom } (\text{rbt-lookup } t1)$ 
      with DOM-T1 have  $k > x$  by simp
      with C show False by simp
    qed
    ultimately have ?thesis by (simp add: map-add-upd-left map-add-dom-app-simps)
  } ultimately show ?thesis using less-linear by blast

```

```

qed
also from Branch
  have rbt-lookup t2 ++ [k ↦ v] ++ rbt-lookup t1 = map-of (entries (Branch c
t1 k v t2)) by simp
  finally show ?case by simp
qed

```

```

lemma rbt-lookup-in-tree: rbt-sorted t  $\implies$  rbt-lookup t k = Some v  $\longleftrightarrow$  (k, v)  $\in$ 
set (entries t)
  by (simp add: map-of-entries [symmetric] distinct-entries)

```

```

lemma set-entries-inject:
  assumes rbt-sorted: rbt-sorted t1 rbt-sorted t2
  shows set (entries t1) = set (entries t2)  $\longleftrightarrow$  entries t1 = entries t2
proof –
  from rbt-sorted have distinct (map fst (entries t1))
    distinct (map fst (entries t2))
  by (auto intro: distinct-entries)
  with rbt-sorted show ?thesis
  by (auto intro: map-sorted-distinct-set-unique rbt-sorted-entries simp add: dis-
tinct-map)
qed

```

```

lemma entries-eqI:
  assumes rbt-sorted: rbt-sorted t1 rbt-sorted t2
  assumes rbt-lookup: rbt-lookup t1 = rbt-lookup t2
  shows entries t1 = entries t2
proof –
  from rbt-sorted rbt-lookup have map-of (entries t1) = map-of (entries t2)
  by (simp add: map-of-entries)
  with rbt-sorted have set (entries t1) = set (entries t2)
  by (simp add: map-of-inject-set distinct-entries)
  with rbt-sorted show ?thesis by (simp add: set-entries-inject)
qed

```

```

lemma entries-rbt-lookup:
  assumes rbt-sorted t1 rbt-sorted t2
  shows entries t1 = entries t2  $\longleftrightarrow$  rbt-lookup t1 = rbt-lookup t2
  using assms by (auto intro: entries-eqI simp add: map-of-entries [symmetric])

```

```

lemma rbt-lookup-from-in-tree:
  assumes rbt-sorted t1 rbt-sorted t2
  and  $\bigwedge v. (k, v) \in \text{set} (\text{entries } t1) \longleftrightarrow (k, v) \in \text{set} (\text{entries } t2)$ 
  shows rbt-lookup t1 k = rbt-lookup t2 k
proof –
  from assms have  $k \in \text{dom} (\text{rbt-lookup } t1) \longleftrightarrow k \in \text{dom} (\text{rbt-lookup } t2)$ 
  by (simp add: keys-entries rbt-lookup-keys)
  with assms show ?thesis by (auto simp add: rbt-lookup-in-tree [symmetric])
qed

```


end

129.2.4 Red-black properties

primrec *color-of* :: (*'a*, *'b*) *rbt* \Rightarrow *color*
where

color-of *Empty* = *B*
| *color-of* (*Branch* *c* - - -) = *c*

primrec *bheight* :: (*'a*, *'b*) *rbt* \Rightarrow *nat*
where

bheight *Empty* = 0
| *bheight* (*Branch* *c* *lt* *k* *v* *rt*) = (if *c* = *B* then *Suc* (*bheight* *lt*) else *bheight* *lt*)

primrec *inv1* :: (*'a*, *'b*) *rbt* \Rightarrow *bool*
where

inv1 *Empty* = *True*
| *inv1* (*Branch* *c* *lt* *k* *v* *rt*) \longleftrightarrow *inv1* *lt* \wedge *inv1* *rt* \wedge (*c* = *B* \vee *color-of* *lt* = *B* \wedge *color-of* *rt* = *B*)

primrec *inv1l* :: (*'a*, *'b*) *rbt* \Rightarrow *bool* — Weaker version
where

inv1l *Empty* = *True*
| *inv1l* (*Branch* *c* *l* *k* *v* *r*) = (*inv1* *l* \wedge *inv1* *r*)
lemma [*simp*]: *inv1* *t* \Longrightarrow *inv1l* *t* **by** (*cases* *t*) *simp*+

primrec *inv2* :: (*'a*, *'b*) *rbt* \Rightarrow *bool*
where

inv2 *Empty* = *True*
| *inv2* (*Branch* *c* *lt* *k* *v* *rt*) = (*inv2* *lt* \wedge *inv2* *rt* \wedge *bheight* *lt* = *bheight* *rt*)

context *ord* **begin**

definition *is-rbt* :: (*'a*, *'b*) *rbt* \Rightarrow *bool* **where**

is-rbt *t* \longleftrightarrow *inv1* *t* \wedge *inv2* *t* \wedge *color-of* *t* = *B* \wedge *rbt-sorted* *t*

lemma *is-rbt-rbt-sorted* [*simp*]:

is-rbt *t* \Longrightarrow *rbt-sorted* *t* **by** (*simp* *add*: *is-rbt-def*)

theorem *Empty-is-rbt* [*simp*]:

is-rbt *Empty* **by** (*simp* *add*: *is-rbt-def*)

end

129.3 Insertion

The function definitions are based on the book by Okasaki.

fun

$balance :: ('a, 'b) \text{ rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$
where
 $balance (Branch R a w x b) s t (Branch R c y z d) = Branch R (Branch B a w x b) s t (Branch B c y z d) \mid$
 $balance (Branch R (Branch R a w x b) s t c) y z d = Branch R (Branch B a w x b) s t (Branch B c y z d) \mid$
 $balance (Branch R a w x (Branch R b s t c)) y z d = Branch R (Branch B a w x b) s t (Branch B c y z d) \mid$
 $balance a w x (Branch R b s t (Branch R c y z d)) = Branch R (Branch B a w x b) s t (Branch B c y z d) \mid$
 $balance a w x (Branch R (Branch R b s t c) y z d) = Branch R (Branch B a w x b) s t (Branch B c y z d) \mid$
 $balance a s t b = Branch B a s t b$

lemma *balance-inv1*: $\llbracket inv1 l \mid l; inv1 r \rrbracket \Longrightarrow inv1 (balance l k v r)$
by (*induct l k v r rule: balance.induct*) *auto*

lemma *balance-bheight*: $bheight l = bheight r \Longrightarrow bheight (balance l k v r) = Suc (bheight l)$
by (*induct l k v r rule: balance.induct*) *auto*

lemma *balance-inv2*:
assumes $inv2 l \mid inv2 r \mid bheight l = bheight r$
shows $inv2 (balance l k v r)$
using *assms*
by (*induct l k v r rule: balance.induct*) *auto*

context *ord* **begin**

lemma *balance-rbt-greater[simp]*: $(v \ll \mid balance a k x b) = (v \ll \mid a \wedge v \ll \mid b \wedge v < k)$
by (*induct a k x b rule: balance.induct*) *auto*

lemma *balance-rbt-less[simp]*: $(balance a k x b \mid \ll v) = (a \mid \ll v \wedge b \mid \ll v \wedge k < v)$
by (*induct a k x b rule: balance.induct*) *auto*

end

lemma (*in linorder*) *balance-rbt-sorted*:
fixes $k :: 'a$
assumes $rbt\text{-}sorted l \mid rbt\text{-}sorted r \mid \ll k k \ll \mid r$
shows $rbt\text{-}sorted (balance l k v r)$
using *assms* **proof** (*induct l k v r rule: balance.induct*)
case (*2-2 a x w b y t c z s va vb vd vc*)
hence $y < z \wedge z \ll \mid Branch B va vb vd vc$
by (*auto simp add: rbt-ord-props*)
hence $y \ll \mid (Branch B va vb vd vc)$ **by** (*blast dest: rbt-greater-trans*)
with *2-2* **show** *?case* **by** *simp*
next

```

case (3-2  $va\ vb\ vd\ vc\ x\ w\ b\ y\ s\ c\ z$ )
from 3-2 have  $x < y \wedge \text{Branch } B\ va\ vb\ vd\ vc \mid \ll x$ 
by simp
hence  $\text{Branch } B\ va\ vb\ vd\ vc \mid \ll y$  by (blast dest: rbt-less-trans)
with 3-2 show ?case by simp
next
case (3-3  $x\ w\ b\ y\ s\ c\ z\ t\ va\ vb\ vd\ vc$ )
from 3-3 have  $y < z \wedge z \ll \mid \text{Branch } B\ va\ vb\ vd\ vc$  by simp
hence  $y \ll \mid \text{Branch } B\ va\ vb\ vd\ vc$  by (blast dest: rbt-greater-trans)
with 3-3 show ?case by simp
next
case (3-4  $vd\ ve\ vg\ vf\ x\ w\ b\ y\ s\ c\ z\ t\ va\ vb\ vii\ vc$ )
hence  $x < y \wedge \text{Branch } B\ vd\ ve\ vg\ vf \mid \ll x$  by simp
hence 1:  $\text{Branch } B\ vd\ ve\ vg\ vf \mid \ll y$  by (blast dest: rbt-less-trans)
from 3-4 have  $y < z \wedge z \ll \mid \text{Branch } B\ va\ vb\ vii\ vc$  by simp
hence  $y \ll \mid \text{Branch } B\ va\ vb\ vii\ vc$  by (blast dest: rbt-greater-trans)
with 1 3-4 show ?case by simp
next
case (4-2  $va\ vb\ vd\ vc\ x\ w\ b\ y\ s\ c\ z\ t\ dd$ )
hence  $x < y \wedge \text{Branch } B\ va\ vb\ vd\ vc \mid \ll x$  by simp
hence  $\text{Branch } B\ va\ vb\ vd\ vc \mid \ll y$  by (blast dest: rbt-less-trans)
with 4-2 show ?case by simp
next
case (5-2  $x\ w\ b\ y\ s\ c\ z\ t\ va\ vb\ vd\ vc$ )
hence  $y < z \wedge z \ll \mid \text{Branch } B\ va\ vb\ vd\ vc$  by simp
hence  $y \ll \mid \text{Branch } B\ va\ vb\ vd\ vc$  by (blast dest: rbt-greater-trans)
with 5-2 show ?case by simp
next
case (5-3  $va\ vb\ vd\ vc\ x\ w\ b\ y\ s\ c\ z\ t$ )
hence  $x < y \wedge \text{Branch } B\ va\ vb\ vd\ vc \mid \ll x$  by simp
hence  $\text{Branch } B\ va\ vb\ vd\ vc \mid \ll y$  by (blast dest: rbt-less-trans)
with 5-3 show ?case by simp
next
case (5-4  $va\ vb\ vg\ vc\ x\ w\ b\ y\ s\ c\ z\ t\ vd\ ve\ vii\ vf$ )
hence  $x < y \wedge \text{Branch } B\ va\ vb\ vg\ vc \mid \ll x$  by simp
hence 1:  $\text{Branch } B\ va\ vb\ vg\ vc \mid \ll y$  by (blast dest: rbt-less-trans)
from 5-4 have  $y < z \wedge z \ll \mid \text{Branch } B\ vd\ ve\ vii\ vf$  by simp
hence  $y \ll \mid \text{Branch } B\ vd\ ve\ vii\ vf$  by (blast dest: rbt-greater-trans)
with 1 5-4 show ?case by simp
qed simp+
```

lemma *entries-balance* [*simp*]:

entries (*balance* $l\ k\ v\ r$) = *entries* $l\ @\ (k, v) \#$ *entries* r
by (*induct* $l\ k\ v\ r$ *rule: balance.induct*) *auto*

lemma *keys-balance* [*simp*]:

keys (*balance* $l\ k\ v\ r$) = *keys* $l\ @\ k \#$ *keys* r
by (*simp* *add: keys-def*)

lemma *balance-in-tree*:

entry-in-tree $k\ x$ (*balance* $l\ v\ y\ r$) \longleftrightarrow *entry-in-tree* $k\ x\ l \vee k = v \wedge x = y \vee$
entry-in-tree $k\ x\ r$
by (*auto simp add: keys-def*)

lemma (*in linorder*) *rbt-lookup-balance[simp]*:

fixes $k :: 'a$

assumes *rbt-sorted* $l\ rbt\text{-sorted}\ r\ l \mid \ll k\ k \mid r$

shows *rbt-lookup* (*balance* $l\ k\ v\ r$) $x =$ *rbt-lookup* (*Branch* $B\ l\ k\ v\ r$) x

by (*rule rbt-lookup-from-in-tree*) (*auto simp: assms balance-in-tree balance-rbt-sorted*)

primrec *paint* :: *color* \Rightarrow ($'a, 'b$) *rbt* \Rightarrow ($'a, 'b$) *rbt*

where

paint $c\ Empty = Empty$

\mid *paint* $c\ (Branch\ -\ l\ k\ v\ r) = Branch\ c\ l\ k\ v\ r$

lemma *paint-inv1l[simp]*: *inv1l* $t \Longrightarrow$ *inv1l* (*paint* $c\ t$) **by** (*cases t*) *auto*

lemma *paint-inv1[simp]*: *inv1* $t \Longrightarrow$ *inv1* (*paint* $B\ t$) **by** (*cases t*) *auto*

lemma *paint-inv2[simp]*: *inv2* $t \Longrightarrow$ *inv2* (*paint* $c\ t$) **by** (*cases t*) *auto*

lemma *paint-color-of[simp]*: *color-of* (*paint* $B\ t$) $= B$ **by** (*cases t*) *auto*

lemma *paint-in-tree[simp]*: *entry-in-tree* $k\ x$ (*paint* $c\ t$) $=$ *entry-in-tree* $k\ x\ t$ **by**
(*cases t*) *auto*

context *ord* **begin**

lemma *paint-rbt-sorted[simp]*: *rbt-sorted* $t \Longrightarrow$ *rbt-sorted* (*paint* $c\ t$) **by** (*cases t*) *auto*

lemma *paint-rbt-lookup[simp]*: *rbt-lookup* (*paint* $c\ t$) $=$ *rbt-lookup* t **by** (*rule ext*)
(*cases t, auto*)

lemma *paint-rbt-greater[simp]*: $(v \ll \mid paint\ c\ t) = (v \ll \mid t)$ **by** (*cases t*) *auto*

lemma *paint-rbt-less[simp]*: $(paint\ c\ t \mid \ll v) = (t \mid \ll v)$ **by** (*cases t*) *auto*

fun

rbt-ins :: ($'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$) \Rightarrow $'a \Rightarrow 'b \Rightarrow$ ($'a, 'b$) *rbt* \Rightarrow ($'a, 'b$) *rbt*

where

rbt-ins $f\ k\ v\ Empty = Branch\ R\ Empty\ k\ v\ Empty \mid$

rbt-ins $f\ k\ v\ (Branch\ B\ l\ x\ y\ r) =$ (*if* $k < x$ *then* *balance* (*rbt-ins* $f\ k\ v\ l$) $x\ y\ r$
else if $k > x$ *then* *balance* $l\ x\ y$ (*rbt-ins* $f\ k\ v\ r$)
else *Branch* $B\ l\ x$ (*f* $k\ y\ v$) r) \mid

rbt-ins $f\ k\ v\ (Branch\ R\ l\ x\ y\ r) =$ (*if* $k < x$ *then* *Branch* R (*rbt-ins* $f\ k\ v\ l$) $x\ y\ r$
else if $k > x$ *then* *Branch* $R\ l\ x\ y$ (*rbt-ins* $f\ k\ v\ r$)
else *Branch* $R\ l\ x$ (*f* $k\ y\ v$) r)

lemma *ins-inv1-inv2*:

assumes *inv1* $t\ inv2\ t$

shows *inv2* (*rbt-ins* $f\ k\ x\ t$) *bheight* (*rbt-ins* $f\ k\ x\ t$) $=$ *bheight* t

color-of $t = B \Longrightarrow$ *inv1* (*rbt-ins* $f\ k\ x\ t$) *inv1l* (*rbt-ins* $f\ k\ x\ t$)

using *assms*

by (*induct f k x t rule: rbt-ins.induct*) (*auto simp: balance-inv1 balance-inv2*)

balance-bheight)

end

context *linorder* **begin**

lemma *ins-rbt-greater*[*simp*]: $(v \ll \mid \text{rbt-ins } f \ (k :: 'a) \ x \ t) = (v \ll \mid t \wedge k > v)$

by (*induct* *f k x t rule: rbt-ins.induct*) *auto*

lemma *ins-rbt-less*[*simp*]: $(\text{rbt-ins } f \ k \ x \ t \mid \ll v) = (t \mid \ll v \wedge k < v)$

by (*induct* *f k x t rule: rbt-ins.induct*) *auto*

lemma *ins-rbt-sorted*[*simp*]: $\text{rbt-sorted } t \implies \text{rbt-sorted } (\text{rbt-ins } f \ k \ x \ t)$

by (*induct* *f k x t rule: rbt-ins.induct*) (*auto simp: balance-rbt-sorted*)

lemma *keys-ins*: $\text{set } (\text{keys } (\text{rbt-ins } f \ k \ v \ t)) = \{ k \} \cup \text{set } (\text{keys } t)$

by (*induct* *f k v t rule: rbt-ins.induct*) *auto*

lemma *rbt-lookup-ins*:

fixes *k* :: *'a*

assumes *rbt-sorted t*

shows *rbt-lookup* (*rbt-ins f k v t*) *x* = $((\text{rbt-lookup } t)(k \mid \rightarrow \text{case } \text{rbt-lookup } t \ k$

of None $\Rightarrow v$

$\mid \text{Some } w \Rightarrow f \ k \ w \ v)) \ x$

using *assms* **by** (*induct* *f k v t rule: rbt-ins.induct*) *auto*

end

context *ord* **begin**

definition *rbt-insert-with-key* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$

where *rbt-insert-with-key* *f k v t* = *paint B* (*rbt-ins f k v t*)

definition *rbt-insertw-def*: *rbt-insert-with* *f* = *rbt-insert-with-key* ($\lambda \cdot. f$)

definition *rbt-insert* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**

rbt-insert = *rbt-insert-with-key* ($\lambda \cdot. \text{nv. nv}$)

end

context *linorder* **begin**

lemma *rbt-insertwk-rbt-sorted*: $\text{rbt-sorted } t \implies \text{rbt-sorted } (\text{rbt-insert-with-key } f \ (k :: 'a) \ x \ t)$

by (*auto simp: rbt-insert-with-key-def*)

theorem *rbt-insertwk-is-rbt*:

assumes *inv: is-rbt t*

shows *is-rbt* (*rbt-insert-with-key f k x t*)

using *assms*

unfolding *rbt-insert-with-key-def is-rbt-def*
by (*auto simp: ins-inv1-inv2*)

lemma *rbt-lookup-rbt-insertwk:*

assumes *rbt-sorted t*

shows *rbt-lookup (rbt-insert-with-key f k v t) x = ((rbt-lookup t)(k | \rightarrow case
 rbt-lookup t k of None \Rightarrow v
 | Some w \Rightarrow f k w v)) x*

unfolding *rbt-insert-with-key-def* **using** *assms*

by (*simp add: rbt-lookup-ins*)

lemma *rbt-insertw-rbt-sorted: rbt-sorted t \Longrightarrow rbt-sorted (rbt-insert-with f k v t)*

by (*simp add: rbt-insertwk-rbt-sorted rbt-insertw-def*)

theorem *rbt-insertw-is-rbt: is-rbt t \Longrightarrow is-rbt (rbt-insert-with f k v t)*

by (*simp add: rbt-insertwk-is-rbt rbt-insertw-def*)

lemma *rbt-lookup-rbt-insertw:*

is-rbt t \Longrightarrow

rbt-lookup (rbt-insert-with f k v t) =

*(rbt-lookup t)(k \mapsto (if $k \in \text{dom (rbt-lookup t)}$ then $f (\text{the (rbt-lookup t k)}) v$
 else v))*

by (*rule ext, cases rbt-lookup t k (auto simp: rbt-lookup-rbt-insertwk dom-def
 rbt-insertw-def)*)

lemma *rbt-insert-rbt-sorted: rbt-sorted t \Longrightarrow rbt-sorted (rbt-insert k v t)*

by (*simp add: rbt-insertwk-rbt-sorted rbt-insert-def*)

theorem *rbt-insert-is-rbt [simp]: is-rbt t \Longrightarrow is-rbt (rbt-insert k v t)*

by (*simp add: rbt-insertwk-is-rbt rbt-insert-def*)

lemma *rbt-lookup-rbt-insert: is-rbt t \Longrightarrow rbt-lookup (rbt-insert k v t) = (rbt-lookup
 t)(k \mapsto v)*

by (*rule ext (simp add: rbt-insert-def rbt-lookup-rbt-insertwk split: option.split)*)

end

129.4 Deletion

lemma *bheight-paintR'[simp]: color-of t = B \Longrightarrow bheight (paint R t) = bheight t
 - 1*

by (*cases t rule: rbt-cases*) *auto*

The function definitions are based on the Haskell code by Stefan Kahrs
 at <http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html>.

fun

balance-left :: ('a,'b) rbt \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a,'b) rbt \Rightarrow ('a,'b) rbt

where

balance-left (Branch R a k x b) s y c = Branch R (Branch B a k x b) s y c |

balance-left bl k x (Branch B a s y b) = balance bl k x (Branch R a s y b) |

*balance-left bl k x (Branch R (Branch B a s y b) t z c) = Branch R (Branch B bl
 k x a) s y (balance b t z (paint R c)) |*

balance-left *t k x s* = *Empty*

lemma *balance-left-inv2-with-inv1*:

assumes *inv2* *lt* *inv2* *rt* *bheight* *lt* + 1 = *bheight* *rt* *inv1* *rt*

shows *bheight* (*balance-left* *lt k v* *rt*) = *bheight* *lt* + 1

and *inv2* (*balance-left* *lt k v* *rt*)

using *assms*

by (*induct* *lt k v* *rt* *rule*: *balance-left.induct*) (*auto simp*: *balance-inv2 balance-bheight*)

lemma *balance-left-inv2-app*:

assumes *inv2* *lt* *inv2* *rt* *bheight* *lt* + 1 = *bheight* *rt* *color-of* *rt* = *B*

shows *inv2* (*balance-left* *lt k v* *rt*)

bheight (*balance-left* *lt k v* *rt*) = *bheight* *rt*

using *assms*

by (*induct* *lt k v* *rt* *rule*: *balance-left.induct*) (*auto simp add*: *balance-inv2 balance-bheight*)⁺

lemma *balance-left-inv1*: $\llbracket \text{inv1 } a; \text{inv1 } b; \text{color-of } b = B \rrbracket \implies \text{inv1 } (\text{balance-left } a \text{ } k \text{ } b)$

by (*induct* *a k x b* *rule*: *balance-left.induct*) (*simp add*: *balance-inv1*)⁺

lemma *balance-left-inv1l*: $\llbracket \text{inv1l } lt; \text{inv1 } rt \rrbracket \implies \text{inv1l } (\text{balance-left } lt \text{ } k \text{ } x \text{ } rt)$

by (*induct* *lt k x rt* *rule*: *balance-left.induct*) (*auto simp*: *balance-inv1*)

lemma (*in linorder*) *balance-left-rbt-sorted*:

$\llbracket \text{rbt-sorted } l; \text{rbt-sorted } r; \text{rbt-less } k \text{ } l; k \ll r \rrbracket \implies \text{rbt-sorted } (\text{balance-left } l \text{ } k \text{ } v \text{ } r)$

apply (*induct* *l k v r* *rule*: *balance-left.induct*)

apply (*auto simp*: *balance-rbt-sorted*)

apply (*unfold* *rbt-greater-prop* *rbt-less-prop*)

by *force*⁺

context *order* **begin**

lemma *balance-left-rbt-greater*:

fixes *k* :: 'a

assumes *k* \ll *a* *k* \ll *b* *k* < *x*

shows *k* \ll *balance-left* *a* *x* *t* *b*

using *assms*

by (*induct* *a x t b* *rule*: *balance-left.induct*) *auto*

lemma *balance-left-rbt-less*:

fixes *k* :: 'a

assumes *a* \ll *k* *b* \ll *k* *x* < *k*

shows *balance-left* *a* *x* *t* *b* \ll *k*

using *assms*

by (*induct* *a x t b* *rule*: *balance-left.induct*) *auto*

end

lemma *balance-left-in-tree*:

assumes *inv1l l inv1 r bheight l + 1 = bheight r*

shows *entry-in-tree k v (balance-left l a b r) = (entry-in-tree k v l ∨ k = a ∧ v = b ∨ entry-in-tree k v r)*

using *assms*

by (*induct l k v r rule: balance-left.induct*) (*auto simp: balance-in-tree*)

fun

balance-right :: (*'a, 'b*) *r*bt ⇒ *'a* ⇒ *'b* ⇒ (*'a, 'b*) *r*bt ⇒ (*'a, 'b*) *r*bt

where

balance-right a k x (Branch R b s y c) = Branch R a k x (Branch B b s y c) |

balance-right (Branch B a k x b) s y bl = balance (Branch R a k x b) s y bl |

balance-right (Branch R a k x (Branch B b s y c)) t z bl = Branch R (balance (paint R a) k x b) s y (Branch B c t z bl) |

balance-right t k x s = Empty

lemma *balance-right-inv2-with-inv1*:

assumes *inv2 lt inv2 rt bheight lt = bheight rt + 1 inv1 lt*

shows *inv2 (balance-right lt k v rt) ∧ bheight (balance-right lt k v rt) = bheight lt*

using *assms*

by (*induct lt k v rt rule: balance-right.induct*) (*auto simp: balance-inv2 balance-bheight*)

lemma *balance-right-inv1*: $\llbracket \text{inv1 } a; \text{inv1l } b; \text{color-of } a = B \rrbracket \Longrightarrow \text{inv1 } (\text{balance-right } a \text{ k x b})$

by (*induct a k x b rule: balance-right.induct*) (*simp add: balance-inv1*)⁺

lemma *balance-right-inv1l*: $\llbracket \text{inv1 lt}; \text{inv1l rt} \rrbracket \Longrightarrow \text{inv1l } (\text{balance-right lt k x rt})$

by (*induct lt k x rt rule: balance-right.induct*) (*auto simp: balance-inv1*)

lemma (*in linorder*) *balance-right-rbt-sorted*:

$\llbracket \text{rbt-sorted l}; \text{rbt-sorted r}; \text{rbt-less k l}; k \ll r \rrbracket \Longrightarrow \text{rbt-sorted } (\text{balance-right l k v r})$

apply (*induct l k v r rule: balance-right.induct*)

apply (*auto simp: balance-rbt-sorted*)

apply (*unfold rbt-less-prop rbt-greater-prop*)

by *force*⁺

context *order* **begin**

lemma *balance-right-rbt-greater*:

fixes *k* :: *'a*

assumes $k \ll a \text{ k} \ll b \text{ k} < x$

shows $k \ll \text{balance-right } a \text{ x t b}$

using *assms* **by** (*induct a x t b rule: balance-right.induct*) *auto*

lemma *balance-right-rbt-less*:

fixes *k* :: *'a*


```

assumes  $a \ll k \ b \ll k \ x < k$ 
shows  $\text{balance-right } a \ x \ t \ b \ll k$ 
using assms by (induct  $a \ x \ t \ b$  rule: balance-right.induct) auto

end

lemma balance-right-in-tree:
  assumes  $\text{inv1 } l \ \text{inv1l } r \ \text{bheight } l = \text{bheight } r + 1 \ \text{inv2 } l \ \text{inv2 } r$ 
  shows  $\text{entry-in-tree } x \ y \ (\text{balance-right } l \ k \ v \ r) = (\text{entry-in-tree } x \ y \ l \vee x = k \wedge$ 
 $y = v \vee \text{entry-in-tree } x \ y \ r)$ 
  using assms by (induct  $l \ k \ v \ r$  rule: balance-right.induct) (auto simp: balance-in-tree)

fun
  combine ::  $('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ 
where
  combine Empty  $x = x$ 
  | combine  $x$  Empty  $= x$ 
  | combine  $(\text{Branch } R \ a \ k \ x \ b) \ (\text{Branch } R \ c \ s \ y \ d) = (\text{case } (\text{combine } b \ c) \text{ of}$ 
 $\text{Branch } R \ b2 \ t \ z \ c2 \Rightarrow (\text{Branch } R \ (\text{Branch } R \ a \ k \ x$ 
 $b2) \ t \ z \ (\text{Branch } R \ c2 \ s \ y \ d)) \mid$ 
 $\text{bc} \Rightarrow \text{Branch } R \ a \ k \ x \ (\text{Branch } R \ bc \ s \ y \ d))$ 
  | combine  $(\text{Branch } B \ a \ k \ x \ b) \ (\text{Branch } B \ c \ s \ y \ d) = (\text{case } (\text{combine } b \ c) \text{ of}$ 
 $\text{Branch } R \ b2 \ t \ z \ c2 \Rightarrow \text{Branch } R \ (\text{Branch } B \ a \ k \ x \ b2)$ 
 $t \ z \ (\text{Branch } B \ c2 \ s \ y \ d)) \mid$ 
 $\text{bc} \Rightarrow \text{balance-left } a \ k \ x \ (\text{Branch } B \ bc \ s \ y \ d))$ 
  | combine  $a \ (\text{Branch } R \ b \ k \ x \ c) = \text{Branch } R \ (\text{combine } a \ b) \ k \ x \ c$ 
  | combine  $(\text{Branch } R \ a \ k \ x \ b) \ c = \text{Branch } R \ a \ k \ x \ (\text{combine } b \ c)$ 

lemma combine-inv2:
  assumes  $\text{inv2 } lt \ \text{inv2 } rt \ \text{bheight } lt = \text{bheight } rt$ 
  shows  $\text{bheight } (\text{combine } lt \ rt) = \text{bheight } lt \ \text{inv2 } (\text{combine } lt \ rt)$ 
using assms
by (induct  $lt \ rt$  rule: combine.induct)
  (auto simp: balance-left-inv2-app split: rbt.splits color.splits)

lemma combine-inv1:
  assumes  $\text{inv1 } lt \ \text{inv1 } rt$ 
  shows  $\text{color-of } lt = B \Longrightarrow \text{color-of } rt = B \Longrightarrow \text{inv1 } (\text{combine } lt \ rt)$ 
 $\text{inv1l } (\text{combine } lt \ rt)$ 
using assms
by (induct  $lt \ rt$  rule: combine.induct)
  (auto simp: balance-left-inv1 split: rbt.splits color.splits)

context linorder begin

lemma combine-rbt-greater[simp]:
  fixes  $k :: 'a$ 
  assumes  $k \ll l \ k \ll r$ 
  shows  $k \ll \text{combine } l \ r$ 

```

```

using assms
by (induct l r rule: combine.induct)
    (auto simp: balance-left-rbt-greater split:rbt.splits color.splits)

lemma combine-rbt-less[simp]:
  fixes k :: 'a
  assumes l |« k r |« k
  shows combine l r |« k
using assms
by (induct l r rule: combine.induct)
    (auto simp: balance-left-rbt-less split:rbt.splits color.splits)

lemma combine-rbt-sorted:
  fixes k :: 'a
  assumes rbt-sorted l rbt-sorted r l |« k k «| r
  shows rbt-sorted (combine l r)
using assms proof (induct l r rule: combine.induct)
  case (∃ a x v b c y w d)
    hence ineqs: a |« x x «| b b |« k k «| c c |« y y «| d
    by auto
    with ∃
    show ?case
      by (cases combine b c rule: rbt-cases)
      (auto, (metis combine-rbt-greater combine-rbt-less ineqs ineqs rbt-less-simps(2)
rbt-greater-simps(2) rbt-greater-trans rbt-less-trans)+)
  next
    case (! a x v b c y w d)
    hence x < k ∧ rbt-greater k c by simp
    hence rbt-greater x c by (blast dest: rbt-greater-trans)
    with ! have 2: rbt-greater x (combine b c) by (simp add: combine-rbt-greater)
    from ! have k < y ∧ rbt-less k b by simp
    hence rbt-less y b by (blast dest: rbt-less-trans)
    with ! have 3: rbt-less y (combine b c) by (simp add: combine-rbt-less)
    show ?case
    proof (cases combine b c rule: rbt-cases)
      case Empty
      from ! have x < y ∧ rbt-greater y d by auto
      hence rbt-greater x d by (blast dest: rbt-greater-trans)
      with ! Empty have rbt-sorted a and rbt-sorted (Branch B Empty y w d)
      and rbt-less x a and rbt-greater x (Branch B Empty y w d) by auto
      with Empty show ?thesis by (simp add: balance-left-rbt-sorted)
    next
      case (Red lta va ka rta)
      with 2 ! have x < va ∧ rbt-less x a by simp
      hence 5: rbt-less va a by (blast dest: rbt-less-trans)
      from Red 3 ! have va < y ∧ rbt-greater y d by simp
      hence rbt-greater va d by (blast dest: rbt-greater-trans)
      with Red 2 3 ! 5 show ?thesis by simp
    next

```

```

    case (Black lta va ka rta)
    from 4 have  $x < y \wedge \text{rbt-greater } y \ d$  by auto
    hence  $\text{rbt-greater } x \ d$  by (blast dest: rbt-greater-trans)
    with Black 2 3 4 have  $\text{rbt-sorted } a$  and  $\text{rbt-sorted } (\text{Branch } B \ (\text{combine } b \ c) \ y \ w \ d)$ 
    and  $\text{rbt-less } x \ a$  and  $\text{rbt-greater } x \ (\text{Branch } B \ (\text{combine } b \ c) \ y \ w \ d)$  by auto
    with Black show ?thesis by (simp add: balance-left-rbt-sorted)
  qed
next
  case (5 va vb vd vc b x w c)
  hence  $k < x \wedge \text{rbt-less } k \ (\text{Branch } B \ va \ vb \ vd \ vc)$  by simp
  hence  $\text{rbt-less } x \ (\text{Branch } B \ va \ vb \ vd \ vc)$  by (blast dest: rbt-less-trans)
  with 5 show ?case by (simp add: combine-rbt-less)
next
  case (6 a x v b va vb vd vc)
  hence  $x < k \wedge \text{rbt-greater } k \ (\text{Branch } B \ va \ vb \ vd \ vc)$  by simp
  hence  $\text{rbt-greater } x \ (\text{Branch } B \ va \ vb \ vd \ vc)$  by (blast dest: rbt-greater-trans)
  with 6 show ?case by (simp add: combine-rbt-greater)
qed simp+

```

end

lemma combine-in-tree:

```

  assumes  $\text{inv2 } l \ \text{inv2 } r \ \text{bheight } l = \text{bheight } r \ \text{inv1 } l \ \text{inv1 } r$ 
  shows  $\text{entry-in-tree } k \ v \ (\text{combine } l \ r) = (\text{entry-in-tree } k \ v \ l \vee \text{entry-in-tree } k \ v \ r)$ 
using assms
proof (induct l r rule: combine.induct)
  case (4 - - b c)
  hence  $a: \text{bheight } (\text{combine } b \ c) = \text{bheight } b$  by (simp add: combine-inv2)
  from 4 have  $b: \text{inv1 } l \ (\text{combine } b \ c)$  by (simp add: combine-inv1)

  show ?case
  proof (cases combine b c rule: rbt-cases)
    case Empty
    with 4 a show ?thesis by (auto simp: balance-left-in-tree)
  next
    case (Red lta ka va rta)
    with 4 show ?thesis by auto
  next
    case (Black lta ka va rta)
    with a b 4 show ?thesis by (auto simp: balance-left-in-tree)
  qed
qed (auto split: rbt.splits color.splits)

```

context ord begin

fun

```

  rbt-del-from-left ::  $'a \Rightarrow ('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$  and
  rbt-del-from-right ::  $'a \Rightarrow ('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$  and

```

$rbt-del :: 'a \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt$
where
 $rbt-del\ x\ Empty = Empty \mid$
 $rbt-del\ x\ (Branch\ c\ a\ y\ s\ b) =$
 $(if\ x < y\ then\ rbt-del-from-left\ x\ a\ y\ s\ b$
 $\quad else\ (if\ x > y\ then\ rbt-del-from-right\ x\ a\ y\ s\ b\ else\ combine\ a\ b)) \mid$
 $rbt-del-from-left\ x\ (Branch\ B\ lt\ z\ v\ rt)\ y\ s\ b = balance-left\ (rbt-del\ x\ (Branch\ B$
 $lt\ z\ v\ rt))\ y\ s\ b \mid$
 $rbt-del-from-left\ x\ a\ y\ s\ b = Branch\ R\ (rbt-del\ x\ a)\ y\ s\ b \mid$
 $rbt-del-from-right\ x\ a\ y\ s\ (Branch\ B\ lt\ z\ v\ rt) = balance-right\ a\ y\ s\ (rbt-del\ x$
 $(Branch\ B\ lt\ z\ v\ rt)) \mid$
 $rbt-del-from-right\ x\ a\ y\ s\ b = Branch\ R\ a\ y\ s\ (rbt-del\ x\ b)$

end

context *linorder* **begin**

lemma

assumes $inv2\ lt\ inv1\ lt$

shows

$\llbracket inv2\ rt; bheight\ lt = bheight\ rt; inv1\ rt \rrbracket \implies$

$inv2\ (rbt-del-from-left\ x\ lt\ k\ v\ rt) \wedge$

$bheight\ (rbt-del-from-left\ x\ lt\ k\ v\ rt) = bheight\ lt \wedge$

$(color-of\ lt = B \wedge color-of\ rt = B \wedge inv1\ (rbt-del-from-left\ x\ lt\ k\ v\ rt) \vee$

$(color-of\ lt \neq B \vee color-of\ rt \neq B) \wedge inv1l\ (rbt-del-from-left\ x\ lt\ k\ v\ rt))$

and $\llbracket inv2\ rt; bheight\ lt = bheight\ rt; inv1\ rt \rrbracket \implies$

$inv2\ (rbt-del-from-right\ x\ lt\ k\ v\ rt) \wedge$

$bheight\ (rbt-del-from-right\ x\ lt\ k\ v\ rt) = bheight\ lt \wedge$

$(color-of\ lt = B \wedge color-of\ rt = B \wedge inv1\ (rbt-del-from-right\ x\ lt\ k\ v\ rt) \vee$

$(color-of\ lt \neq B \vee color-of\ rt \neq B) \wedge inv1l\ (rbt-del-from-right\ x\ lt\ k\ v\ rt))$

and $rbt-del-inv1-inv2: inv2\ (rbt-del\ x\ lt) \wedge (color-of\ lt = R \wedge bheight\ (rbt-del\ x$
 $lt) = bheight\ lt \wedge inv1\ (rbt-del\ x\ lt))$

$\vee color-of\ lt = B \wedge bheight\ (rbt-del\ x\ lt) = bheight\ lt - 1 \wedge inv1l\ (rbt-del\ x\ lt))$

using *assms*

proof (*induct* $x\ lt\ k\ v\ rt$ **and** $x\ lt\ k\ v\ rt$ **and** $x\ lt$ *rule: rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)

case ($2\ y\ c - y'$)

have $y = y' \vee y < y' \vee y > y'$ **by** *auto*

thus *?case proof (elim disjE)*

assume $y = y'$

with 2 **show** *?thesis by (cases c) (simp add: combine-inv2 combine-inv1)+*

next

assume $y < y'$

with 2 **show** *?thesis by (cases c) auto*

next

assume $y' < y$

with 2 **show** *?thesis by (cases c) auto*

qed

next

case ($3\ y\ lt\ z\ v\ rta\ y'\ ss\ bb$)

thus ?case **by** (cases color-of (Branch B lt z v rta) = B \wedge color-of bb = B) (simp
 add: balance-left-inv2-with-inv1 balance-left-inv1 balance-left-inv1l)+
next
 case (5 y a y' ss lt z v rta)
thus ?case **by** (cases color-of a = B \wedge color-of (Branch B lt z v rta) = B) (simp
 add: balance-right-inv2-with-inv1 balance-right-inv1 balance-right-inv1l)+
next
 case (6-1 y a y' ss) **thus** ?case **by** (cases color-of a = B \wedge color-of Empty = B)
 simp+
qed auto

lemma

rbt-del-from-left-rbt-less: $\llbracket lt \mid \ll v; rt \mid \ll v; k < v \rrbracket \implies \text{rbt-del-from-left } x \text{ lt } k \text{ y } rt$
 $\mid \ll v$
and rbt-del-from-right-rbt-less: $\llbracket lt \mid \ll v; rt \mid \ll v; k < v \rrbracket \implies \text{rbt-del-from-right } x$
 $lt \text{ k } y \text{ rt } \mid \ll v$
and rbt-del-rbt-less: $lt \mid \ll v \implies \text{rbt-del } x \text{ lt } \mid \ll v$
by (induct x lt k y rt **and** x lt k y rt **and** x lt rule: rbt-del-from-left-rbt-del-from-right-rbt-del.induct)
 (auto simp: balance-left-rbt-less balance-right-rbt-less)

lemma rbt-del-from-left-rbt-greater: $\llbracket v \mid \ll lt; v \mid \ll rt; k > v \rrbracket \implies v \mid \ll \text{rbt-del-from-left}$
 $x \text{ lt } k \text{ y } rt$

and rbt-del-from-right-rbt-greater: $\llbracket v \mid \ll lt; v \mid \ll rt; k > v \rrbracket \implies v \mid \ll \text{rbt-del-from-right}$
 $x \text{ lt } k \text{ y } rt$

and rbt-del-rbt-greater: $v \mid \ll lt \implies v \mid \ll \text{rbt-del } x \text{ lt}$
by (induct x lt k y rt **and** x lt k y rt **and** x lt rule: rbt-del-from-left-rbt-del-from-right-rbt-del.induct)
 (auto simp: balance-left-rbt-greater balance-right-rbt-greater)

lemma $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll k; k \mid \ll rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-left}$
 $x \text{ lt } k \text{ y } rt)$

and $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll k; k \mid \ll rt \rrbracket \implies \text{rbt-sorted } (\text{rbt-del-from-right}$
 $x \text{ lt } k \text{ y } rt)$

and rbt-del-rbt-sorted: $\text{rbt-sorted } lt \implies \text{rbt-sorted } (\text{rbt-del } x \text{ lt})$

proof (induct x lt k y rt **and** x lt k y rt **and** x lt rule: rbt-del-from-left-rbt-del-from-right-rbt-del.induct)

case (3 x lta zz v rta yy ss bb)

from 3 have Branch B lta zz v rta $\mid \ll yy$ **by** simp

hence rbt-del x (Branch B lta zz v rta) $\mid \ll yy$ **by** (rule rbt-del-rbt-less)

with 3 show ?case **by** (simp add: balance-left-rbt-sorted)

next

case (4-2 x vaa vbb vdd vc yy ss bb)

hence Branch R vaa vbb vdd vc $\mid \ll yy$ **by** simp

hence rbt-del x (Branch R vaa vbb vdd vc) $\mid \ll yy$ **by** (rule rbt-del-rbt-less)

with 4-2 show ?case **by** simp

next

case (5 x aa yy ss lta zz v rta)

hence yy $\mid \ll$ Branch B lta zz v rta **by** simp

hence yy $\mid \ll$ rbt-del x (Branch B lta zz v rta) **by** (rule rbt-del-rbt-greater)

with 5 show ?case **by** (simp add: balance-right-rbt-sorted)

next

case (6-2 x aa yy ss vaa vbb vdd vc)
 hence $yy \ll \text{Branch } R \text{ } vaa \text{ } vbb \text{ } vdd \text{ } vc$ **by** *simp*
 hence $yy \ll \text{rbt-del } x \text{ (Branch } R \text{ } vaa \text{ } vbb \text{ } vdd \text{ } vc)$ **by** (rule *rbt-del-rbt-greater*)
 with 6-2 **show** ?case **by** *simp*
qed (auto *simp*: *combine-rbt-sorted*)

lemma $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll kt; kt \mid \ll rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; x < kt \rrbracket \implies \text{entry-in-tree } k \text{ } v \text{ (rbt-del-from-left } x \text{ } lt \text{ } kt \text{ } y \text{ } rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ (Branch } c \text{ } lt \text{ } kt \text{ } y \text{ } rt)))$

and $\llbracket \text{rbt-sorted } lt; \text{rbt-sorted } rt; lt \mid \ll kt; kt \mid \ll rt; \text{inv1 } lt; \text{inv1 } rt; \text{inv2 } lt; \text{inv2 } rt; \text{bheight } lt = \text{bheight } rt; x > kt \rrbracket \implies \text{entry-in-tree } k \text{ } v \text{ (rbt-del-from-right } x \text{ } lt \text{ } kt \text{ } y \text{ } rt) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ (Branch } c \text{ } lt \text{ } kt \text{ } y \text{ } rt)))$

and *rbt-del-in-tree*: $\llbracket \text{rbt-sorted } t; \text{inv1 } t; \text{inv2 } t \rrbracket \implies \text{entry-in-tree } k \text{ } v \text{ (rbt-del } x \text{ } t) = (\text{False} \vee (x \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } t))$

proof (*induct* $x \text{ } lt \text{ } kt \text{ } y \text{ } rt$ **and** $x \text{ } lt \text{ } kt \text{ } y \text{ } rt$ **and** $x \text{ } t$ rule: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)

case (2 xx c aa yy ss bb)
 have $xx = yy \vee xx < yy \vee xx > yy$ **by** *auto*
 from this 2 **show** ?case **proof** (*elim disjE*)
 assume $xx = yy$
 with 2 **show** ?thesis **proof** (cases $xx = k$)
 case *True*
 from 2 $\langle xx = yy \rangle \langle xx = k \rangle$ **have** *rbt-sorted* (Branch c aa yy ss bb) $\wedge k = yy$
by *simp*
 hence $\neg \text{entry-in-tree } k \text{ } v \text{ } aa \neg \text{entry-in-tree } k \text{ } v \text{ } bb$ **by** (auto *simp*: *rbt-less-nit* *rbt-greater-prop*)
 with $\langle xx = yy \rangle$ 2 $\langle xx = k \rangle$ **show** ?thesis **by** (*simp add*: *combine-in-tree*)
qed (*simp add*: *combine-in-tree*)
qed *simp*+

next

case (3 xx lta zz vv rta yy ss bb)
 define mt **where** [*simp*]: $mt = \text{Branch } B \text{ } lta \text{ } zz \text{ } vv \text{ } rta$
 from 3 **have** $\text{inv2 } mt \wedge \text{inv1 } mt$ **by** *simp*
 hence $\text{inv2 } (\text{rbt-del } xx \text{ } mt) \wedge (\text{color-of } mt = R \wedge \text{bheight } (\text{rbt-del } xx \text{ } mt) = \text{bheight } mt \wedge \text{inv1 } (\text{rbt-del } xx \text{ } mt) \vee \text{color-of } mt = B \wedge \text{bheight } (\text{rbt-del } xx \text{ } mt) = \text{bheight } mt - 1 \wedge \text{inv1l } (\text{rbt-del } xx \text{ } mt))$ **by** (*blast dest*: *rbt-del-inv1-inv2*)
 with 3 **have** 4: $\text{entry-in-tree } k \text{ } v \text{ (rbt-del-from-left } xx \text{ } mt \text{ } yy \text{ } ss \text{ } bb) = (\text{False} \vee xx \neq k \wedge \text{entry-in-tree } k \text{ } v \text{ } mt \vee (k = yy \wedge v = ss) \vee \text{entry-in-tree } k \text{ } v \text{ } bb)$ **by** (*simp add*: *balance-left-in-tree*)
 thus ?case **proof** (cases $xx = k$)
 case *True*
 from 3 *True* **have** $yy \ll bb \wedge yy > k$ **by** *simp*
 hence $k \ll bb$ **by** (*blast dest*: *rbt-greater-trans*)
 with 3 4 *True* **show** ?thesis **by** (auto *simp*: *rbt-greater-nit*)
qed *auto*

next

case (4-1 xx yy ss bb)
 show ?case **proof** (cases $xx = k$)
 case *True*

```

    with 4-1 have yy «| bb ∧ k < yy by simp
    hence k «| bb by (blast dest: rbt-greater-trans)
    with 4-1 ⟨xx = k⟩
    have entry-in-tree k v (Branch R Empty yy ss bb) = entry-in-tree k v Empty by
    (auto simp: rbt-greater-nit)
    thus ?thesis by auto
  qed simp+
next
case (4-2 xx vaa vbb vdd vc yy ss bb)
thus ?case proof (cases xx = k)
  case True
  with 4-2 have k < yy ∧ yy «| bb by simp
  hence k «| bb by (blast dest: rbt-greater-trans)
  with True 4-2 show ?thesis by (auto simp: rbt-greater-nit)
qed auto
next
case (5 xx aa yy ss lta zz vv rta)
define mt where [simp]: mt = Branch B lta zz vv rta
from 5 have inv2 mt ∧ inv1 mt by simp
hence inv2 (rbt-del xx mt) ∧ (color-of mt = R ∧ bheight (rbt-del xx mt) = bheight
mt ∧ inv1 (rbt-del xx mt) ∨ color-of mt = B ∧ bheight (rbt-del xx mt) = bheight
mt - 1 ∧ inv1l (rbt-del xx mt)) by (blast dest: rbt-del-inv1-inv2)
with 5 have 3: entry-in-tree k v (rbt-del-from-right xx aa yy ss mt) = (entry-in-tree
k v aa ∨ (k = yy ∧ v = ss) ∨ False ∨ xx ≠ k ∧ entry-in-tree k v mt) by (simp
add: balance-right-in-tree)
thus ?case proof (cases xx = k)
  case True
  from 5 True have aa |« yy ∧ yy < k by simp
  hence aa |« k by (blast dest: rbt-less-trans)
  with 3 5 True show ?thesis by (auto simp: rbt-less-nit)
qed auto
next
case (6-1 xx aa yy ss)
show ?case proof (cases xx = k)
  case True
  with 6-1 have aa |« yy ∧ k > yy by simp
  hence aa |« k by (blast dest: rbt-less-trans)
  with 6-1 ⟨xx = k⟩ show ?thesis by (auto simp: rbt-less-nit)
qed simp
next
case (6-2 xx aa yy ss vaa vbb vdd vc)
thus ?case proof (cases xx = k)
  case True
  with 6-2 have k > yy ∧ aa |« yy by simp
  hence aa |« k by (blast dest: rbt-less-trans)
  with True 6-2 show ?thesis by (auto simp: rbt-less-nit)
qed auto
qed simp

```

definition (in *ord*) *rbt-delete* **where**
rbt-delete *k t* = *paint B (rbt-del k t)*

theorem *rbt-delete-is-rbt* [*simp*]: **assumes** *is-rbt t* **shows** *is-rbt (rbt-delete k t)*

proof –

from *assms* **have** *inv2 t* **and** *inv1 t* **unfolding** *is-rbt-def* **by** *auto*
hence *inv2 (rbt-del k t) ∧ (color-of t = R ∧ bheight (rbt-del k t) = bheight t ∧*
inv1 (rbt-del k t) ∨ color-of t = B ∧ bheight (rbt-del k t) = bheight t - 1 ∧ inv1
(rbt-del k t)) **by** (rule *rbt-del-inv1-inv2*)
hence *inv2 (rbt-del k t) ∧ inv1 (rbt-del k t)* **by** (cases *color-of t*) *auto*
with *assms* **show** ?thesis
unfolding *is-rbt-def rbt-delete-def*
by (auto intro: *paint-rbt-sorted rbt-del-rbt-sorted*)

qed

lemma *rbt-delete-in-tree*:

assumes *is-rbt t*
shows *entry-in-tree k v (rbt-delete x t) = (x ≠ k ∧ entry-in-tree k v t)*
using *assms* **unfolding** *is-rbt-def rbt-delete-def*
by (auto simp: *rbt-del-in-tree*)

lemma *rbt-lookup-rbt-delete*:

assumes *is-rbt: is-rbt t*
shows *rbt-lookup (rbt-delete k t) = (rbt-lookup t)|‘(-{k})*
proof
fix *x*
show *rbt-lookup (rbt-delete k t) x = (rbt-lookup t |‘(-{k})) x*
proof (cases *x = k*)
assume *x = k*
with *is-rbt* **show** ?thesis
by (cases *rbt-lookup (rbt-delete k t) k*) (auto simp: *rbt-lookup-in-tree rbt-delete-in-tree*)
next
assume *x ≠ k*
thus ?thesis
by auto (metis *is-rbt rbt-delete-is-rbt rbt-delete-in-tree is-rbt-rbt-sorted rbt-lookup-from-in-tree*)

qed

qed

end

129.5 Modifying existing entries

context *ord* **begin**

primrec

rbt-map-entry :: ‘*a* ⇒ (‘*b* ⇒ ‘*b*) ⇒ (‘*a*, ‘*b*) *rbt* ⇒ (‘*a*, ‘*b*) *rbt*

where

rbt-map-entry k f Empty = *Empty*

| *rbt-map-entry k f (Branch c lt x v rt)* =

(if $k < x$ then Branch c (rbt-map-entry k f lt) x v rt
 else if $k > x$ then (Branch c lt x v (rbt-map-entry k f rt))
 else Branch c lt x (f v) rt)

lemma *rbt-map-entry-color-of*: color-of (rbt-map-entry k f t) = color-of t **by**
 (induct t) simp+

lemma *rbt-map-entry-inv1*: inv1 (rbt-map-entry k f t) = inv1 t **by** (induct t) (simp
 add: rbt-map-entry-color-of)+

lemma *rbt-map-entry-inv2*: inv2 (rbt-map-entry k f t) = inv2 t bheight (rbt-map-entry
 k f t) = bheight t **by** (induct t) simp+

lemma *rbt-map-entry-rbt-greater*: rbt-greater a (rbt-map-entry k f t) = rbt-greater
 a t **by** (induct t) simp+

lemma *rbt-map-entry-rbt-less*: rbt-less a (rbt-map-entry k f t) = rbt-less a t **by**
 (induct t) simp+

lemma *rbt-map-entry-rbt-sorted*: rbt-sorted (rbt-map-entry k f t) = rbt-sorted t
by (induct t) (simp-all add: rbt-map-entry-rbt-less rbt-map-entry-rbt-greater)

theorem *rbt-map-entry-is-rbt* [simp]: is-rbt (rbt-map-entry k f t) = is-rbt t

unfolding *is-rbt-def* **by** (simp add: rbt-map-entry-inv2 rbt-map-entry-color-of rbt-map-entry-rbt-sorted
 rbt-map-entry-inv1)

end

theorem (in *linorder*) *rbt-lookup-rbt-map-entry*:

$\text{rbt-lookup (rbt-map-entry } k \text{ } f \text{ } t) = (\text{rbt-lookup } t)(k := \text{map-option } f \text{ (rbt-lookup } t$
 $k))$

by (induct t) (auto split: option.splits simp add: fun-eq-iff)

129.6 Mapping all entries

primrec

$\text{map} :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'c) \text{rbt}$

where

$\text{map } f \text{ Empty} = \text{Empty}$

$| \text{map } f (\text{Branch } c \text{ } lt \text{ } k \text{ } v \text{ } rt) = \text{Branch } c \text{ (map } f \text{ } lt) \text{ } k \text{ (} f \text{ } k \text{ } v) \text{ (map } f \text{ } rt)$

lemma *map-entries* [simp]: entries (map f t) = List.map ($\lambda(k, v). (k, f \text{ } k \text{ } v)$)
 (entries t)

by (induct t) auto

lemma *map-keys* [simp]: keys (map f t) = keys t **by** (simp add: keys-def split-def)

lemma *map-color-of*: color-of (map f t) = color-of t **by** (induct t) simp+

lemma *map-inv1*: inv1 (map f t) = inv1 t **by** (induct t) (simp add: map-color-of)+

lemma *map-inv2*: inv2 (map f t) = inv2 t bheight (map f t) = bheight t **by** (induct
 t) simp+

context *ord* **begin**

lemma *map-rbt-greater*: rbt-greater k (map f t) = rbt-greater k t **by** (induct t)

simp+
lemma *map-rbt-less*: *rbt-less* *k* (*map* *f* *t*) = *rbt-less* *k* *t* **by** (*induct* *t*) *simp*+
lemma *map-rbt-sorted*: *rbt-sorted* (*map* *f* *t*) = *rbt-sorted* *t* **by** (*induct* *t*) (*simp*
add: *map-rbt-less* *map-rbt-greater*) +
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map* *f* *t*) = *is-rbt* *t*
unfolding *is-rbt-def* **by** (*simp* *add*: *map-inv1* *map-inv2* *map-rbt-sorted* *map-color-of*)
end

theorem (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup* (*map* *f* *t*) *x* = *map-option* (*f* *x*)
(*rbt-lookup* *t* *x*)
by (*induct* *t*) (*auto* *simp*: *antisym-conv3*)

hide-const (**open**) *map*

129.7 Folding over entries

definition *fold* :: (*'a* \Rightarrow *'b* \Rightarrow *'c* \Rightarrow *'c*) \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow *'c* \Rightarrow *'c* **where**
fold *f* *t* = *List.fold* (*case-prod* *f*) (*entries* *t*)

lemma *fold-simps* [*simp*]:
fold *f* *Empty* = *id*
fold *f* (*Branch* *c* *lt* *k* *v* *rt*) = *fold* *f* *rt* \circ *f* *k* *v* \circ *fold* *f* *lt*
by (*simp-all* *add*: *fold-def* *fun-eq-iff*)

lemma *fold-code* [*code*]:
fold *f* *Empty* *x* = *x*
fold *f* (*Branch* *c* *lt* *k* *v* *rt*) *x* = *fold* *f* *rt* (*f* *k* *v* (*fold* *f* *lt* *x*))
by(*simp-all*)

— fold with continuation predicate

fun *foldi* :: (*'c* \Rightarrow *bool*) \Rightarrow (*'a* \Rightarrow *'b* \Rightarrow *'c* \Rightarrow *'c*) \Rightarrow (*'a* :: *linorder*, *'b*) *rbt* \Rightarrow *'c* \Rightarrow *'c*

where

foldi *c* *f* *Empty* *s* = *s* |
foldi *c* *f* (*Branch* *col* *l* *k* *v* *r*) *s* = (
 if (*c* *s*) *then*
 let *s'* = *foldi* *c* *f* *l* *s* *in*
 if (*c* *s'*) *then*
 foldi *c* *f* *r* (*f* *k* *v* *s'*)
 else *s'*
 else
 s
)

129.8 Bulkloading a tree

definition (**in** *ord*) *rbt-bulkload* :: (*'a* \times *'b*) *list* \Rightarrow (*'a*, *'b*) *rbt* **where**
rbt-bulkload *xs* = *foldr* ($\lambda(k, v). \text{rbt-insert } k \ v$) *xs* *Empty*

context *linorder* **begin**

lemma *rbt-bulkload-is-rbt* [*simp*, *intro*]:

is-rbt (*rbt-bulkload* *xs*)

unfolding *rbt-bulkload-def* **by** (*induct* *xs*) *auto*

lemma *rbt-lookup-rbt-bulkload*:

rbt-lookup (*rbt-bulkload* *xs*) = *map-of* *xs*

proof –

obtain *ys* **where** *ys* = *rev* *xs* **by** *simp*

have $\bigwedge t. \text{is-rbt } t \implies$

rbt-lookup (*List.fold* (*case-prod* *rbt-insert*) *ys* *t*) = *rbt-lookup* *t* ++ *map-of* (*rev* *ys*)

by (*induct* *ys*) (*simp-all* *add: rbt-bulkload-def rbt-lookup-rbt-insert case-prod-beta*)

from *this* *Empty-is-rbt* **have**

rbt-lookup (*List.fold* (*case-prod* *rbt-insert*) (*rev* *xs*) *Empty*) = *rbt-lookup* *Empty* ++ *map-of* *xs*

by (*simp* *add: ys = rev xs*)

then show *?thesis* **by** (*simp* *add: rbt-bulkload-def rbt-lookup-Empty foldr-conv-fold*)
qed

end

129.9 Building a RBT from a sorted list

These functions have been adapted from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *rbtreeify-f* :: *nat* \Rightarrow (*'a* \times *'b*) *list* \Rightarrow (*'a*, *'b*) *rbt* \times (*'a* \times *'b*) *list*

and *rbtreeify-g* :: *nat* \Rightarrow (*'a* \times *'b*) *list* \Rightarrow (*'a*, *'b*) *rbt* \times (*'a* \times *'b*) *list*

where

rbtreeify-f *n* *kvs* =

(if *n* = 0 then (*Empty*, *kvs*)

else if *n* = 1 then

case *kvs* of (*k*, *v*) # *kvs'* \Rightarrow (*Branch* *R* *Empty* *k* *v* *Empty*, *kvs'*)

else if (*n* mod 2 = 0) then

case *rbtreeify-f* (*n* div 2) *kvs* of (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow

apfst (*Branch* *B* *t1* *k* *v*) (*rbtreeify-g* (*n* div 2) *kvs'*)

else case *rbtreeify-f* (*n* div 2) *kvs* of (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow

apfst (*Branch* *B* *t1* *k* *v*) (*rbtreeify-f* (*n* div 2) *kvs'*))

| *rbtreeify-g* *n* *kvs* =

(if *n* = 0 \vee *n* = 1 then (*Empty*, *kvs*)

else if *n* mod 2 = 0 then

case *rbtreeify-g* (*n* div 2) *kvs* of (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow

apfst (*Branch* *B* *t1* *k* *v*) (*rbtreeify-g* (*n* div 2) *kvs'*)

else case *rbtreeify-f* (*n* div 2) *kvs* of (*t1*, (*k*, *v*) # *kvs'*) \Rightarrow

apfst (*Branch* *B* *t1* *k* *v*) (*rbtreeify-g* (*n* div 2) *kvs'*))

definition *rbtreeify* :: ('a × 'b) list ⇒ ('a, 'b) rbt
where *rbtreeify* kvs = fst (rbtreeify-g (Suc (length kvs)) kvs)

declare *rbtreeify-f.simps* [simp del] *rbtreeify-g.simps* [simp del]

lemma *rbtreeify-f-code* [code]:

rbtreeify-f n kvs =
 (if n = 0 then (Empty, kvs)
 else if n = 1 then
 case kvs of (k, v) # kvs' ⇒
 (Branch R Empty k v Empty, kvs')
 else let (n', r) = Euclidean-Rings.divmod-nat n 2 in
 if r = 0 then
 case *rbtreeify-f* n' kvs of (t1, (k, v) # kvs') ⇒
 apfst (Branch B t1 k v) (*rbtreeify-g* n' kvs')
 else case *rbtreeify-f* n' kvs of (t1, (k, v) # kvs') ⇒
 apfst (Branch B t1 k v) (*rbtreeify-f* n' kvs')
by (subst *rbtreeify-f.simps*) (simp only: Let-def Euclidean-Rings.divmod-nat-def prod.case)

lemma *rbtreeify-g-code* [code]:

rbtreeify-g n kvs =
 (if n = 0 ∨ n = 1 then (Empty, kvs)
 else let (n', r) = Euclidean-Rings.divmod-nat n 2 in
 if r = 0 then
 case *rbtreeify-g* n' kvs of (t1, (k, v) # kvs') ⇒
 apfst (Branch B t1 k v) (*rbtreeify-g* n' kvs')
 else case *rbtreeify-f* n' kvs of (t1, (k, v) # kvs') ⇒
 apfst (Branch B t1 k v) (*rbtreeify-g* n' kvs')
by(subst *rbtreeify-g.simps*)(simp only: Let-def Euclidean-Rings.divmod-nat-def prod.case)

lemma *Suc-double-half*: Suc (2 * n) div 2 = n
by simp

lemma *div2-plus-div2*: n div 2 + n div 2 = (n :: nat) - n mod 2
by arith

lemma *rbtreeify-f-rec-aux-lemma*:

[[k - n div 2 = Suc k'; n ≤ k; n mod 2 = Suc 0]]
 ⇒ k' - n div 2 = k - n
apply(rule add-right-imp-eq[where a = n - n div 2])
apply(subst add-diff-assoc2, arith)
apply(simp add: *div2-plus-div2*)
done

lemma *rbtreeify-f-simps*:

rbtreeify-f 0 kvs = (Empty, kvs)
rbtreeify-f (Suc 0) ((k, v) # kvs) =
 (Branch R Empty k v Empty, kvs)

```

0 < n ==> rbtreeify-f (2 * n) kvs =
  (case rbtreeify-f n kvs of (t1, (k, v) # kvs') =>
    apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
0 < n ==> rbtreeify-f (Suc (2 * n)) kvs =
  (case rbtreeify-f n kvs of (t1, (k, v) # kvs') =>
    apfst (Branch B t1 k v) (rbtreeify-f n kvs'))
by(subst (1) rbtreeify-f.simps, simp add: Suc-double-half)+

```

lemma *rbtreeify-g-simps*:

```

rbtreeify-g 0 kvs = (Empty, kvs)
rbtreeify-g (Suc 0) kvs = (Empty, kvs)
0 < n ==> rbtreeify-g (2 * n) kvs =
  (case rbtreeify-g n kvs of (t1, (k, v) # kvs') =>
    apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
0 < n ==> rbtreeify-g (Suc (2 * n)) kvs =
  (case rbtreeify-f n kvs of (t1, (k, v) # kvs') =>
    apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
by(subst (1) rbtreeify-g.simps, simp add: Suc-double-half)+

```

declare *rbtreeify-f-simps*[simp] *rbtreeify-g-simps*[simp]

lemma *length-rbtreeify-f*: $n \leq \text{length } kvs$

```

==> length (snd (rbtreeify-f n kvs)) = length kvs - n
and length-rbtreeify-g: [ 0 < n; n ≤ Suc (length kvs) ]
==> length (snd (rbtreeify-g n kvs)) = Suc (length kvs) - n
proof(induction n kvs and n kvs rule: rbtreeify-f-rbtreeify-g.induct)

```

case (1 n kvs)

show ?case

proof(cases n ≤ 1)

case True thus ?thesis using 1.prem

by(cases n kvs rule: nat.exhaust[case-product list.exhaust]) auto

next

case False

hence $n \neq 0$ $n \neq 1$ by simp-all

note IH = 1.IH[OF this]

show ?thesis

proof(cases n mod 2 = 0)

case True

hence length (snd (rbtreeify-f n kvs)) =

length (snd (rbtreeify-f (2 * (n div 2)) kvs))

by(metis minus-nat.diff-0 minus-mod-eq-mult-div [symmetric])

also from 1.prem False obtain k v kvs'

where kvs: kvs = (k, v) # kvs' by(cases kvs) auto

also have $0 < n \text{ div } 2$ using False by(simp)

note rbtreeify-f-simps(3)[OF this]

also note kvs[symmetric]

also let ?rest1 = snd (rbtreeify-f (n div 2) kvs)

from 1.prem have $n \text{ div } 2 \leq \text{length } kvs$ by simp

with True have len: length ?rest1 = length kvs - n div 2 by(rule IH)

```

with 1.prem False obtain t1 k' v' kvs''
  where kvs'': rbtreeify-f (n div 2) kvs = (t1, (k', v') # kvs'')
  by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm)
note this also note prod.case also note list.simps(5)
also note prod.case also note snd-apfst
also have  $0 < n \text{ div } 2 \text{ } n \text{ div } 2 \leq \text{Suc } (\text{length } kvs'')$ 
  using len 1.prem False unfolding kvs'' by simp-all
with True kvs''[symmetric] refl refl
have  $\text{length } (\text{snd } (\text{rbtreeify-g } (n \text{ div } 2) kvs'')) =$ 
   $\text{Suc } (\text{length } kvs'') - n \text{ div } 2$  by(rule IH)
finally show ?thesis using len[unfolded kvs''] 1.prem True
by(simp add: Suc-diff-le[symmetric] mult-2[symmetric] minus-mod-eq-mult-div
  [symmetric])
next
  case False
  hence  $\text{length } (\text{snd } (\text{rbtreeify-f } n kvs)) =$ 
     $\text{length } (\text{snd } (\text{rbtreeify-f } (\text{Suc } (2 * (n \text{ div } 2))) kvs))$ 
    by (simp add: mod-eq-0-iff-dvd)
  also from 1.prem  $\langle \neg n \leq 1 \rangle$  obtain k v kvs'
    where kvs: kvs = (k, v) # kvs' by(cases kvs auto)
  also have  $0 < n \text{ div } 2$  using  $\langle \neg n \leq 1 \rangle$  by(simp)
  note rbtreeify-f-simps(4)[OF this]
  also note kvs[symmetric]
  also let ?rest1 = snd (rbtreeify-f (n div 2) kvs)
  from 1.prem have  $n \text{ div } 2 \leq \text{length } kvs$  by simp
  with False have len:  $\text{length } ?rest1 = \text{length } kvs - n \text{ div } 2$  by(rule IH)
  with 1.prem  $\langle \neg n \leq 1 \rangle$  obtain t1 k' v' kvs''
    where kvs'': rbtreeify-f (n div 2) kvs = (t1, (k', v') # kvs'')
    by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm)
  note this also note prod.case also note list.simps(5)
  also note prod.case also note snd-apfst
  also have  $n \text{ div } 2 \leq \text{length } kvs''$ 
    using len 1.prem False unfolding kvs'' by simp arith
  with False kvs''[symmetric] refl refl
  have  $\text{length } (\text{snd } (\text{rbtreeify-f } (n \text{ div } 2) kvs'')) = \text{length } kvs'' - n \text{ div } 2$ 
    by(rule IH)
  finally show ?thesis using len[unfolded kvs''] 1.prem False
    by simp(rule rbtreeify-f-rec-aux-lemma[OF sym])
qed
qed
next
  case ( $2 \mid n$ ) kvs)
  show ?case
  proof(cases n > 1)
    case False with  $\langle 0 < n \rangle$  show ?thesis
      by(cases n kvs rule: nat.exhaust[case-product list.exhaust]) simp-all
  next
    case True
    hence  $\neg (n = 0 \vee n = 1)$  by simp

```

```

note  $IH = 2.IH[OF\ this]$ 
show  $?thesis$ 
proof( $cases\ n\ mod\ 2 = 0$ )
  case  $True$ 
    hence  $length\ (snd\ (rbtreeify-g\ n\ kvs)) =$ 
       $length\ (snd\ (rbtreeify-g\ (2 * (n\ div\ 2))\ kvs))$ 
      by( $metis\ minus-nat.diff-0\ minus-mod-eq-mult-div\ [symmetric]$ )
    also from  $2.prem\ True$  obtain  $k\ v\ kvs'$ 
      where  $kvs: kvs = (k, v) \# kvs'$  by( $cases\ kvs$ ) auto
    also have  $0 < n\ div\ 2$  using  $\langle 1 < n \rangle$  by( $simp$ )
    note  $rbtreeify-g-simps(3)[OF\ this]$ 
    also note  $kvs[symmetric]$ 
    also let  $?rest1 = snd\ (rbtreeify-g\ (n\ div\ 2)\ kvs)$ 
    from  $2.prem\ \langle 1 < n \rangle$ 
    have  $0 < n\ div\ 2\ n\ div\ 2 \leq Suc\ (length\ kvs)$  by  $simp-all$ 
    with  $True$  have  $len: length\ ?rest1 = Suc\ (length\ kvs) - n\ div\ 2$  by( $rule\ IH$ )
    with  $2.prem$  obtain  $t1\ k'\ v'\ kvs''$ 
      where  $kvs'': rbtreeify-g\ (n\ div\ 2)\ kvs = (t1, (k', v') \# kvs'')$ 
      by( $cases\ ?rest1$ )( $auto\ simp\ add: snd-def\ split: prod.split-asm$ )
    note this also note  $prod.case$  also note  $list.simps(5)$ 
    also note  $prod.case$  also note  $snd-apfst$ 
    also have  $n\ div\ 2 \leq Suc\ (length\ kvs'')$ 
      using  $len\ 2.prem$  unfolding  $kvs''$  by  $simp$ 
    with  $True\ kvs''[symmetric]$   $refl\ refl\ \langle 0 < n\ div\ 2 \rangle$ 
    have  $length\ (snd\ (rbtreeify-g\ (n\ div\ 2)\ kvs'')) = Suc\ (length\ kvs'') - n\ div\ 2$ 
      by( $rule\ IH$ )
    finally show  $?thesis$  using  $len[unfolded\ kvs'']\ 2.prem\ True$ 
    by( $simp\ add: Suc-diff-le[symmetric]\ mult-2[symmetric]\ minus-mod-eq-mult-div$ 
 $[symmetric]$ )
  next
    case  $False$ 
    hence  $length\ (snd\ (rbtreeify-g\ n\ kvs)) =$ 
       $length\ (snd\ (rbtreeify-g\ (Suc\ (2 * (n\ div\ 2))))\ kvs))$ 
      by ( $simp\ add: mod-eq-0-iff-dvd$ )
    also from  $2.prem\ \langle 1 < n \rangle$  obtain  $k\ v\ kvs'$ 
      where  $kvs: kvs = (k, v) \# kvs'$  by( $cases\ kvs$ ) auto
    also have  $0 < n\ div\ 2$  using  $\langle 1 < n \rangle$  by( $simp$ )
    note  $rbtreeify-g-simps(4)[OF\ this]$ 
    also note  $kvs[symmetric]$ 
    also let  $?rest1 = snd\ (rbtreeify-f\ (n\ div\ 2)\ kvs)$ 
    from  $2.prem$  have  $n\ div\ 2 \leq length\ kvs$  by  $simp$ 
    with  $False$  have  $len: length\ ?rest1 = length\ kvs - n\ div\ 2$  by( $rule\ IH$ )
    with  $2.prem\ \langle 1 < n \rangle\ False$  obtain  $t1\ k'\ v'\ kvs''$ 
      where  $kvs'': rbtreeify-f\ (n\ div\ 2)\ kvs = (t1, (k', v') \# kvs'')$ 
      by( $cases\ ?rest1$ )( $auto\ simp\ add: snd-def\ split: prod.split-asm, arith$ )
    note this also note  $prod.case$  also note  $list.simps(5)$ 
    also note  $prod.case$  also note  $snd-apfst$ 
    also have  $n\ div\ 2 \leq Suc\ (length\ kvs'')$ 
      using  $len\ 2.prem\ False$  unfolding  $kvs''$  by  $simp\ arith$ 

```

```

with False  $kvs''$ [symmetric] refl refl  $\langle 0 < n \text{ div } 2 \rangle$ 
have length (snd (rbtreeify-g (n div 2)  $kvs''$ )) = Suc (length  $kvs''$ ) - n div 2
by(rule IH)
finally show ?thesis using len[unfolded  $kvs''$ ] 2.prem False
by(simp add: div2-plus-div2)
qed
qed
qed

```

lemma *rbtreeify-induct* [consumes 1, case-names f-0 f-1 f-even f-odd g-0 g-1 g-even g-odd]:

```

fixes P Q
defines f0 == ( $\bigwedge kvs. P \ 0 \ kvs$ )
and f1 == ( $\bigwedge k \ v \ kvs. P \ (Suc \ 0) \ ((k, v) \# \ kvs)$ )
and feven ==
  ( $\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$ 
     $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$ 
     $\implies P \ (2 * n) \ kvs$ )
and fodd ==
  ( $\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$ 
     $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{length } kvs'; P \ n \ kvs' \rrbracket$ 
     $\implies P \ (\text{Suc } (2 * n)) \ kvs$ )
and g0 == ( $\bigwedge kvs. Q \ 0 \ kvs$ )
and g1 == ( $\bigwedge kvs. Q \ (\text{Suc } 0) \ kvs$ )
and geven ==
  ( $\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{Suc } (\text{length } kvs); Q \ n \ kvs;$ 
     $\text{rbtreeify-g } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$ 
     $\implies Q \ (2 * n) \ kvs$ )
and godd ==
  ( $\bigwedge n \ kvs \ t \ k \ v \ kvs'. \llbracket n > 0; n \leq \text{length } kvs; P \ n \ kvs;$ 
     $\text{rbtreeify-f } n \ kvs = (t, (k, v) \# \ kvs'); n \leq \text{Suc } (\text{length } kvs'); Q \ n \ kvs' \rrbracket$ 
     $\implies Q \ (\text{Suc } (2 * n)) \ kvs$ )
shows  $\llbracket n \leq \text{length } kvs;$ 
  PROP f0; PROP f1; PROP feven; PROP fodd;
  PROP g0; PROP g1; PROP geven; PROP godd  $\rrbracket$ 
   $\implies P \ n \ kvs$ 
and  $\llbracket n \leq \text{Suc } (\text{length } kvs);$ 
  PROP f0; PROP f1; PROP feven; PROP fodd;
  PROP g0; PROP g1; PROP geven; PROP godd  $\rrbracket$ 
   $\implies Q \ n \ kvs$ 
proof –
  assume f0: PROP f0 and f1: PROP f1 and feven: PROP feven and fodd:
  PROP fodd
  and g0: PROP g0 and g1: PROP g1 and geven: PROP geven and godd:
  PROP godd
  show  $n \leq \text{length } kvs \implies P \ n \ kvs$  and  $n \leq \text{Suc } (\text{length } kvs) \implies Q \ n \ kvs$ 
  proof(induction rule: rbtreeify-f-rbtreeify-g.induct)
  case (1 n kvs)
  show ?case

```



```

proof(cases  $n \leq 1$ )
  case True thus ?thesis using 1.prem
    by(cases  $n$   $kvs$  rule: nat.exhaust[case-product list.exhaust])
      (auto simp add: f0[unfolded f0-def] f1[unfolded f1-def])
  next
    case False
    hence  $ns: n \neq 0 \wedge n \neq 1$  by simp-all
    hence  $ge0: n \text{ div } 2 > 0$  by simp
    note  $IH = 1.IH[OF ns]$ 
    show ?thesis
    proof(cases  $n \bmod 2 = 0$ )
      case True note  $ge0$ 
      moreover from 1.prem have  $n2: n \text{ div } 2 \leq \text{length } kvs$  by simp
      moreover from True  $n2$  have  $P (n \text{ div } 2) kvs$  by(rule IH)
      moreover from length-rbtreeify-f[OF  $n2$ ]  $ge0$  1.prem obtain  $t \ k \ v \ kvs'$ 
        where  $kvs': \text{rbtreeify-f } (n \text{ div } 2) \ kvs = (t, (k, v) \# kvs')$ 
        by(cases  $\text{snd } (\text{rbtreeify-f } (n \text{ div } 2) \ kvs)$ )
          (auto simp add: snd-def split: prod.split-asm)
      moreover from 1.prem length-rbtreeify-f[OF  $n2$ ]  $ge0$ 
        have  $n2': n \text{ div } 2 \leq \text{Suc } (\text{length } kvs')$  by(simp add:  $kvs'$ )
      moreover from True  $kvs'$ [symmetric] refl refl  $n2'$ 
        have  $Q (n \text{ div } 2) kvs'$  by(rule IH)
      moreover note feven[unfolded feven-def]

      ultimately have  $P (2 * (n \text{ div } 2)) kvs$  by –
      thus ?thesis using True by (metis minus-mod-eq-div-mult [symmetric]
        minus-nat.diff-0 mult commute)
    next
      case False note  $ge0$ 
      moreover from 1.prem have  $n2: n \text{ div } 2 \leq \text{length } kvs$  by simp
      moreover from False  $n2$  have  $P (n \text{ div } 2) kvs$  by(rule IH)
      moreover from length-rbtreeify-f[OF  $n2$ ]  $ge0$  1.prem obtain  $t \ k \ v \ kvs'$ 
        where  $kvs': \text{rbtreeify-f } (n \text{ div } 2) \ kvs = (t, (k, v) \# kvs')$ 
        by(cases  $\text{snd } (\text{rbtreeify-f } (n \text{ div } 2) \ kvs)$ )
          (auto simp add: snd-def split: prod.split-asm)
      moreover from 1.prem length-rbtreeify-f[OF  $n2$ ]  $ge0$  False
        have  $n2': n \text{ div } 2 \leq \text{length } kvs'$  by(simp add:  $kvs'$ ) arith
      moreover from False  $kvs'$ [symmetric] refl refl  $n2'$  have  $P (n \text{ div } 2) kvs'$ 
by(rule IH)
      moreover note fodd[unfolded fodd-def]
      ultimately have  $P (\text{Suc } (2 * (n \text{ div } 2))) kvs$  by –
      thus ?thesis using False
      by simp (metis One-nat-def Suc-eq-plus1-left le-add-diff-inverse mod-less-eq-dividend
        minus-mod-eq-mult-div [symmetric])
    qed
  qed
next
  case ( $2 \ n \ kvs$ )
  show ?case

```

```

proof(cases  $n \leq 1$ )
  case True thus ?thesis using 2.prem
    by(cases  $n$   $kvs$  rule: nat.exhaust[case-product list.exhaust])
      (auto simp add: g0[unfolded g0-def] g1[unfolded g1-def])
  next
    case False
    hence  $ns: \neg (n = 0 \vee n = 1)$  by simp
    hence  $ge0: n \text{ div } 2 > 0$  by simp
    note  $IH = 2.IH[OF ns]$ 
    show ?thesis
    proof(cases  $n \bmod 2 = 0$ )
      case True note  $ge0$ 
      moreover from 2.prem have  $n2: n \text{ div } 2 \leq \text{Suc}(\text{length } kvs)$  by simp
      moreover from True  $n2$  have  $Q(n \text{ div } 2) kvs$  by(rule  $IH$ )
      moreover from length-rbtreeify-g[OF  $ge0$   $n2$ ]  $ge0$  2.prem obtain  $t k v kvs'$ 

      where  $kvs': \text{rbtreeify-g}(n \text{ div } 2) kvs = (t, (k, v) \# kvs')$ 
      by(cases  $snd(\text{rbtreeify-g}(n \text{ div } 2) kvs)$ )
        (auto simp add: snd-def split: prod.split-asm)
      moreover from 2.prem length-rbtreeify-g[OF  $ge0$   $n2$ ]  $ge0$ 
      have  $n2': n \text{ div } 2 \leq \text{Suc}(\text{length } kvs')$  by(simp add:  $kvs'$ )
      moreover from True  $kvs'$ [symmetric] refl refl  $n2'$ 
      have  $Q(n \text{ div } 2) kvs'$  by(rule  $IH$ )
      moreover note given[unfolded given-def]
      ultimately have  $Q(2 * (n \text{ div } 2)) kvs$  by –
      thus ?thesis using True
      by(metis minus-mod-eq-div-mult [symmetric] minus-nat.diff-0 mult.commute)
    next
      case False note  $ge0$ 
      moreover from 2.prem have  $n2: n \text{ div } 2 \leq \text{length } kvs$  by simp
      moreover from False  $n2$  have  $P(n \text{ div } 2) kvs$  by(rule  $IH$ )
      moreover from length-rbtreeify-f[OF  $n2$ ]  $ge0$  2.prem False obtain  $t k v$ 
       $kvs'$ 

      where  $kvs': \text{rbtreeify-f}(n \text{ div } 2) kvs = (t, (k, v) \# kvs')$ 
      by(cases  $snd(\text{rbtreeify-f}(n \text{ div } 2) kvs)$ )
        (auto simp add: snd-def split: prod.split-asm, arith)
      moreover from 2.prem length-rbtreeify-f[OF  $n2$ ]  $ge0$  False
      have  $n2': n \text{ div } 2 \leq \text{Suc}(\text{length } kvs')$  by(simp add:  $kvs'$ ) arith
      moreover from False  $kvs'$ [symmetric] refl refl  $n2'$ 
      have  $Q(n \text{ div } 2) kvs'$  by(rule  $IH$ )
      moreover note godd[unfolded godd-def]
      ultimately have  $Q(\text{Suc}(2 * (n \text{ div } 2))) kvs$  by –
      thus ?thesis using False
      by simp (metis One-nat-def Suc-eq-plus1-left le-add-diff-inverse mod-less-eq-dividend
        minus-mod-eq-mult-div [symmetric])
    qed
  qed
qed
qed

```

```

lemma inv1-rbtreeify-f:  $n \leq \text{length } kvs$ 
   $\implies \text{inv1 } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs))$ 
  and inv1-rbtreeify-g:  $n \leq \text{Suc } (\text{length } kvs)$ 
   $\implies \text{inv1 } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs))$ 
by(induct  $n \text{ } kvs$  and  $n \text{ } kvs$  rule: rbtreeify-induct) simp-all

fun plog2 ::  $\text{nat} \Rightarrow \text{nat}$ 
where plog2  $n = (\text{if } n \leq 1 \text{ then } 0 \text{ else } \text{plog2 } (n \text{ div } 2) + 1)$ 

declare plog2.simps [simp del]

lemma plog2-simps [simp]:
   $\text{plog2 } 0 = 0$   $\text{plog2 } (\text{Suc } 0) = 0$ 
   $0 < n \implies \text{plog2 } (2 * n) = 1 + \text{plog2 } n$ 
   $0 < n \implies \text{plog2 } (\text{Suc } (2 * n)) = 1 + \text{plog2 } n$ 
by(subst plog2.simps, simp add: Suc-double-half)+

lemma bheight-rbtreeify-f:  $n \leq \text{length } kvs$ 
   $\implies \text{bheight } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) = \text{plog2 } n$ 
  and bheight-rbtreeify-g:  $n \leq \text{Suc } (\text{length } kvs)$ 
   $\implies \text{bheight } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = \text{plog2 } n$ 
by(induct  $n \text{ } kvs$  and  $n \text{ } kvs$  rule: rbtreeify-induct) simp-all

lemma bheight-rbtreeify-f-eq-plog2I:
   $\llbracket \text{rbtreeify-f } n \text{ } kvs = (t, kvs'); n \leq \text{length } kvs \rrbracket$ 
   $\implies \text{bheight } t = \text{plog2 } n$ 
using bheight-rbtreeify-f[of  $n \text{ } kvs$ ] by simp

lemma bheight-rbtreeify-g-eq-plog2I:
   $\llbracket \text{rbtreeify-g } n \text{ } kvs = (t, kvs'); n \leq \text{Suc } (\text{length } kvs) \rrbracket$ 
   $\implies \text{bheight } t = \text{plog2 } n$ 
using bheight-rbtreeify-g[of  $n \text{ } kvs$ ] by simp

hide-const (open) plog2

lemma inv2-rbtreeify-f:  $n \leq \text{length } kvs$ 
   $\implies \text{inv2 } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs))$ 
  and inv2-rbtreeify-g:  $n \leq \text{Suc } (\text{length } kvs)$ 
   $\implies \text{inv2 } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs))$ 
by(induct  $n \text{ } kvs$  and  $n \text{ } kvs$  rule: rbtreeify-induct)
  (auto simp add: bheight-rbtreeify-f bheight-rbtreeify-g
   intro: bheight-rbtreeify-f-eq-plog2I bheight-rbtreeify-g-eq-plog2I)

lemma  $n \leq \text{length } kvs \implies \text{True}$ 
  and color-of-rbtreeify-g:
   $\llbracket n \leq \text{Suc } (\text{length } kvs); 0 < n \rrbracket$ 
   $\implies \text{color-of } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = B$ 
by(induct  $n \text{ } kvs$  and  $n \text{ } kvs$  rule: rbtreeify-induct) simp-all

```

lemma *entries-rbtreeify-f-append*:

$n \leq \text{length } kvs$

$\implies \text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) @ \text{snd } (\text{rbtreeify-f } n \text{ } kvs) = kvs$

and *entries-rbtreeify-g-append*:

$n \leq \text{Suc } (\text{length } kvs)$

$\implies \text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) @ \text{snd } (\text{rbtreeify-g } n \text{ } kvs) = kvs$

by(*induction rule: rbtreeify-induct*) *simp-all*

lemma *length-entries-rbtreeify-f*:

$n \leq \text{length } kvs \implies \text{length } (\text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs))) = n$

and *length-entries-rbtreeify-g*:

$n \leq \text{Suc } (\text{length } kvs) \implies \text{length } (\text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs))) = n - 1$

by(*induct rule: rbtreeify-induct*) *simp-all*

lemma *rbtreeify-f-conv-drop*:

$n \leq \text{length } kvs \implies \text{snd } (\text{rbtreeify-f } n \text{ } kvs) = \text{drop } n \text{ } kvs$

using *entries-rbtreeify-f-append*[*of n kvs*]

by(*simp add: append-eq-conv-conj length-entries-rbtreeify-f*)

lemma *rbtreeify-g-conv-drop*:

$n \leq \text{Suc } (\text{length } kvs) \implies \text{snd } (\text{rbtreeify-g } n \text{ } kvs) = \text{drop } (n - 1) \text{ } kvs$

using *entries-rbtreeify-g-append*[*of n kvs*]

by(*simp add: append-eq-conv-conj length-entries-rbtreeify-g*)

lemma *entries-rbtreeify-f [simp]*:

$n \leq \text{length } kvs \implies \text{entries } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) = \text{take } n \text{ } kvs$

using *entries-rbtreeify-f-append*[*of n kvs*]

by(*simp add: append-eq-conv-conj length-entries-rbtreeify-f*)

lemma *entries-rbtreeify-g [simp]*:

$n \leq \text{Suc } (\text{length } kvs) \implies$

$\text{entries } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = \text{take } (n - 1) \text{ } kvs$

using *entries-rbtreeify-g-append*[*of n kvs*]

by(*simp add: append-eq-conv-conj length-entries-rbtreeify-g*)

lemma *keys-rbtreeify-f [simp]*: $n \leq \text{length } kvs$

$\implies \text{keys } (\text{fst } (\text{rbtreeify-f } n \text{ } kvs)) = \text{take } n \text{ } (\text{map } \text{fst } kvs)$

by(*simp add: keys-def take-map*)

lemma *keys-rbtreeify-g [simp]*: $n \leq \text{Suc } (\text{length } kvs)$

$\implies \text{keys } (\text{fst } (\text{rbtreeify-g } n \text{ } kvs)) = \text{take } (n - 1) \text{ } (\text{map } \text{fst } kvs)$

by(*simp add: keys-def take-map*)

lemma *rbtreeify-fD*:

$\llbracket \text{rbtreeify-f } n \text{ } kvs = (t, kvs'); n \leq \text{length } kvs \rrbracket$

$\implies \text{entries } t = \text{take } n \text{ } kvs \wedge kvs' = \text{drop } n \text{ } kvs$

using *rbtreeify-f-conv-drop*[*of n kvs*] *entries-rbtreeify-f*[*of n kvs*] **by** *simp*

lemma *rbtreeify-gD*:

[[*rbtreeify-g* *n kvs* = (*t*, *kvs'*); *n* ≤ *Suc* (*length kvs*)]]
 \impl *entries t* = *take* (*n* − 1) *kvs* ∧ *kvs'* = *drop* (*n* − 1) *kvs*
using *rbtreeify-g-conv-drop*[*of n kvs*] *entries-rbtreeify-g*[*of n kvs*] **by** *simp*

lemma *entries-rbtreeify* [*simp*]: *entries* (*rbtreeify kvs*) = *kvs*
by(*simp add: rbtreeify-def entries-rbtreeify-g*)

context *linorder* **begin**

lemma *rbt-sorted-rbtreeify-f*:

[[*n* ≤ *length kvs*; *sorted* (*map fst kvs*); *distinct* (*map fst kvs*)]]
 \impl *rbt-sorted* (*fst* (*rbtreeify-f n kvs*))
and *rbt-sorted-rbtreeify-g*:
[[*n* ≤ *Suc* (*length kvs*); *sorted* (*map fst kvs*); *distinct* (*map fst kvs*)]]
 \impl *rbt-sorted* (*fst* (*rbtreeify-g n kvs*))

proof(*induction n kvs and n kvs rule: rbtreeify-induct*)

case (*f-even n kvs t k v kvs'*)
from *rbtreeify-fD*[*OF* ⟨*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)⟩ ⟨*n* ≤ *length kvs*⟩]
have *entries t* = *take n kvs*
and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold: kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ⟨*sorted* (*map fst kvs*)⟩ *kvs'*
have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* ≤ *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *k* ≤ *x*)
by(*subst (asm) unfold*)(*auto simp add: sorted-append*)
moreover from ⟨*distinct* (*map fst kvs*)⟩ *kvs'*
have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* ≠ *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *x* ≠ *k*)
by(*subst (asm) unfold*)(*auto intro: rev-image-eqI*)
ultimately have (∀ (*x*, *y*) ∈ *set* (*take n kvs*). *x* < *k*) ∧ (∀ (*x*, *y*) ∈ *set kvs'*. *k* < *x*)

by *fastforce*

hence *fst* (*rbtreeify-f n kvs*) |« *k k* »| *fst* (*rbtreeify-g n kvs'*)
using ⟨*n* ≤ *Suc* (*length kvs'*)⟩ ⟨*n* ≤ *length kvs*⟩ *set-take-subset*[*of n* − 1 *kvs*]
by(*auto simp add: ord.rbt-greater-prop ord.rbt-less-prop take-map split-def*)
moreover from ⟨*sorted* (*map fst kvs*)⟩ ⟨*distinct* (*map fst kvs*)⟩
have *rbt-sorted* (*fst* (*rbtreeify-f n kvs*)) **by**(*rule f-even.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ⟨*sorted* (*map fst kvs*)⟩ ⟨*distinct* (*map fst kvs*)⟩
by(*subst (asm) (1 2) unfold, simp add: sorted-append*)
hence *rbt-sorted* (*fst* (*rbtreeify-g n kvs'*)) **by**(*rule f-even.IH*)
ultimately show ?*case*
using ⟨*0* < *n*⟩ ⟨*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)⟩ **by** *simp*

next

case (*f-odd n kvs t k v kvs'*)
from *rbtreeify-fD*[*OF* ⟨*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)⟩ ⟨*n* ≤ *length kvs*⟩]
have *entries t* = *take n kvs*
and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold: kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ⟨*sorted* (*map fst kvs*)⟩ *kvs'*

have $(\forall (x, y) \in \text{set } (\text{take } n \text{ kvs}). x \leq k) \wedge (\forall (x, y) \in \text{set } \text{kvs}'. k \leq x)$
by(subst (asm) unfold)(auto simp add: sorted-append)
moreover from $\langle \text{distinct } (\text{map fst kvs}) \rangle \text{kvs}'$
have $(\forall (x, y) \in \text{set } (\text{take } n \text{ kvs}). x \neq k) \wedge (\forall (x, y) \in \text{set } \text{kvs}'. x \neq k)$
by(subst (asm) unfold)(auto intro: rev-image-eqI)
ultimately have $(\forall (x, y) \in \text{set } (\text{take } n \text{ kvs}). x < k) \wedge (\forall (x, y) \in \text{set } \text{kvs}'. k < x)$
by fastforce
hence $\text{fst } (\text{rbtreeify-f } n \text{ kvs}) \mid \ll k \mid \ll \text{fst } (\text{rbtreeify-f } n \text{ kvs}')$
using $\langle n \leq \text{length kvs}' \rangle \langle n \leq \text{length kvs} \rangle \text{set-take-subset[of } n \text{ kvs}']$
by(auto simp add: rbt-greater-prop rbt-less-prop take-map split-def)
moreover from $\langle \text{sorted } (\text{map fst kvs}) \rangle \langle \text{distinct } (\text{map fst kvs}) \rangle$
have $\text{rbt-sorted } (\text{fst } (\text{rbtreeify-f } n \text{ kvs}))$ **by**(rule f-odd.IH)
moreover have $\text{sorted } (\text{map fst kvs}') \text{ distinct } (\text{map fst kvs}')$
using $\langle \text{sorted } (\text{map fst kvs}) \rangle \langle \text{distinct } (\text{map fst kvs}) \rangle$
by(subst (asm) (1 2) unfold, simp add: sorted-append)+
hence $\text{rbt-sorted } (\text{fst } (\text{rbtreeify-f } n \text{ kvs}'))$ **by**(rule f-odd.IH)
ultimately show ?case
using $\langle 0 < n \rangle \langle \text{rbtreeify-f } n \text{ kvs} = (t, (k, v) \# \text{kvs}') \rangle$ **by simp**
next
case $(g\text{-even } n \text{ kvs } t \text{ k } v \text{ kvs}')$
from $\text{rbtreeify-gD[OF } \langle \text{rbtreeify-g } n \text{ kvs} = (t, (k, v) \# \text{kvs}') \rangle \langle n \leq \text{Suc } (\text{length kvs}) \rangle]$
have $t: \text{entries } t = \text{take } (n - 1) \text{ kvs}$
and $\text{kvs}': \text{drop } (n - 1) \text{ kvs} = (k, v) \# \text{kvs}'$ **by simp-all**
hence $\text{unfold: kvs} = \text{take } (n - 1) \text{ kvs} @ (k, v) \# \text{kvs}'$ **by**(metis append-take-drop-id)
from $\langle \text{sorted } (\text{map fst kvs}) \rangle \text{kvs}'$
have $(\forall (x, y) \in \text{set } (\text{take } (n - 1) \text{ kvs}). x \leq k) \wedge (\forall (x, y) \in \text{set } \text{kvs}'. k \leq x)$
by(subst (asm) unfold)(auto simp add: sorted-append)
moreover from $\langle \text{distinct } (\text{map fst kvs}) \rangle \text{kvs}'$
have $(\forall (x, y) \in \text{set } (\text{take } (n - 1) \text{ kvs}). x \neq k) \wedge (\forall (x, y) \in \text{set } \text{kvs}'. x \neq k)$
by(subst (asm) unfold)(auto intro: rev-image-eqI)
ultimately have $(\forall (x, y) \in \text{set } (\text{take } (n - 1) \text{ kvs}). x < k) \wedge (\forall (x, y) \in \text{set } \text{kvs}'. k < x)$
by fastforce
hence $\text{fst } (\text{rbtreeify-g } n \text{ kvs}) \mid \ll k \mid \ll \text{fst } (\text{rbtreeify-g } n \text{ kvs}')$
using $\langle n \leq \text{Suc } (\text{length kvs}') \rangle \langle n \leq \text{Suc } (\text{length kvs}) \rangle \text{set-take-subset[of } n - 1 \text{ kvs}]$
by(auto simp add: rbt-greater-prop rbt-less-prop take-map split-def)
moreover from $\langle \text{sorted } (\text{map fst kvs}) \rangle \langle \text{distinct } (\text{map fst kvs}) \rangle$
have $\text{rbt-sorted } (\text{fst } (\text{rbtreeify-g } n \text{ kvs}))$ **by**(rule g-even.IH)
moreover have $\text{sorted } (\text{map fst kvs}') \text{ distinct } (\text{map fst kvs}')$
using $\langle \text{sorted } (\text{map fst kvs}) \rangle \langle \text{distinct } (\text{map fst kvs}) \rangle$
by(subst (asm) (1 2) unfold, simp add: sorted-append)+
hence $\text{rbt-sorted } (\text{fst } (\text{rbtreeify-g } n \text{ kvs}'))$ **by**(rule g-even.IH)
ultimately show ?case **using** $\langle 0 < n \rangle \langle \text{rbtreeify-g } n \text{ kvs} = (t, (k, v) \# \text{kvs}') \rangle$
by simp
next
case $(g\text{-odd } n \text{ kvs } t \text{ k } v \text{ kvs}')$

from *rbtreeify-fD*[*OF* $\langle \text{rbtreeify-f } n \text{ kvs} = (t, (k, v) \# \text{kvs}') \rangle \langle n \leq \text{length kvs} \rangle$]
have *entries* $t = \text{take } n \text{ kvs}$
and kvs' : $\text{drop } n \text{ kvs} = (k, v) \# \text{kvs}'$ **by** *simp-all*
hence *unfold*: $\text{kvs} = \text{take } n \text{ kvs} @ (k, v) \# \text{kvs}'$ **by** (*metis append-take-drop-id*)
from $\langle \text{sorted } (\text{map fst kvs}) \rangle \text{kvs}'$
have $(\forall (x, y) \in \text{set } (\text{take } n \text{ kvs}). x \leq k) \wedge (\forall (x, y) \in \text{set kvs}'. k \leq x)$
by (*subst (asm) unfold*) (*auto simp add: sorted-append*)
moreover from $\langle \text{distinct } (\text{map fst kvs}) \rangle \text{kvs}'$
have $(\forall (x, y) \in \text{set } (\text{take } n \text{ kvs}). x \neq k) \wedge (\forall (x, y) \in \text{set kvs}'. x \neq k)$
by (*subst (asm) unfold*) (*auto intro: rev-image-eqI*)
ultimately have $(\forall (x, y) \in \text{set } (\text{take } n \text{ kvs}). x < k) \wedge (\forall (x, y) \in \text{set kvs}'. k < x)$
by *fastforce*
hence $\text{fst } (\text{rbtreeify-f } n \text{ kvs}) \mid \ll k \mid \ll \text{fst } (\text{rbtreeify-g } n \text{ kvs}')$
using $\langle n \leq \text{Suc } (\text{length kvs}') \rangle \langle n \leq \text{length kvs} \rangle \text{set-take-subset}[of \text{ } n - 1 \text{ kvs}']$
by (*auto simp add: rbt-greater-prop rbt-less-prop take-map split-def*)
moreover from $\langle \text{sorted } (\text{map fst kvs}) \rangle \langle \text{distinct } (\text{map fst kvs}) \rangle$
have *rbt-sorted* ($\text{fst } (\text{rbtreeify-f } n \text{ kvs})$) **by** (*rule g-odd.IH*)
moreover have *sorted* ($\text{map fst kvs}'$) *distinct* ($\text{map fst kvs}'$)
using $\langle \text{sorted } (\text{map fst kvs}) \rangle \langle \text{distinct } (\text{map fst kvs}) \rangle$
by (*subst (asm) (1 2) unfold, simp add: sorted-append*) +
hence *rbt-sorted* ($\text{fst } (\text{rbtreeify-g } n \text{ kvs}')$) **by** (*rule g-odd.IH*)
ultimately show *?case*
using $\langle 0 < n \rangle \langle \text{rbtreeify-f } n \text{ kvs} = (t, (k, v) \# \text{kvs}') \rangle$ **by** *simp*
qed *simp-all*

lemma *rbt-sorted-rbtreeify*:

$\llbracket \text{sorted } (\text{map fst kvs}); \text{distinct } (\text{map fst kvs}) \rrbracket \implies \text{rbt-sorted } (\text{rbtreeify kvs})$
by (*simp add: rbtreeify-def rbt-sorted-rbtreeify-g*)

lemma *is-rbt-rbtreeify*:

$\llbracket \text{sorted } (\text{map fst kvs}); \text{distinct } (\text{map fst kvs}) \rrbracket$
 $\implies \text{is-rbt } (\text{rbtreeify kvs})$
by (*simp add: is-rbt-def rbtreeify-def inv1-rbtreeify-g inv2-rbtreeify-g rbt-sorted-rbtreeify-g color-of-rbtreeify-g*)

lemma *rbt-lookup-rbtreeify*:

$\llbracket \text{sorted } (\text{map fst kvs}); \text{distinct } (\text{map fst kvs}) \rrbracket \implies$
 $\text{rbt-lookup } (\text{rbtreeify kvs}) = \text{map-of kvs}$
by (*simp add: map-of-entries[symmetric] rbt-sorted-rbtreeify*)

end

Functions to compare the height of two rbt trees, taken from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *skip-red* :: $(\text{'a}, \text{'b}) \text{rbt} \Rightarrow (\text{'a}, \text{'b}) \text{rbt}$
where
 $\text{skip-red } (\text{Branch color.R } l \text{ k } v \text{ r}) = l$
 $\mid \text{skip-red } t = t$

definition *skip-black* :: (*'a*, *'b*) *rbt* \Rightarrow (*'a*, *'b*) *rbt*

where

skip-black *t* = (let *t'* = *skip-red* *t* in case *t'* of Branch *color.B l k v r* \Rightarrow *l* | - \Rightarrow *t'*)

datatype *compare* = *LT* | *GT* | *EQ*

partial-function (*tailrec*) *compare-height* :: (*'a*, *'b*) *rbt* \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow *compare*

where

compare-height *sx s t tx* =
 (case (*skip-red* *sx*, *skip-red* *s*, *skip-red* *t*, *skip-red* *tx*) of
 (*Branch* - *sx'* - - -, *Branch* - *s'* - - -, *Branch* - *t'* - - -, *Branch* - *tx'* - - -) \Rightarrow
 compare-height (*skip-black* *sx'*) *s' t' (skip-black tx')*
 | (*-*, *rbt.Empty*, *-*, *Branch* - - - -) \Rightarrow *LT*
 | (*Branch* - - - - -, *-*, *rbt.Empty*, *-*) \Rightarrow *GT*
 | (*Branch* - *sx'* - - -, *Branch* - *s'* - - -, *Branch* - *t'* - - -, *rbt.Empty*) \Rightarrow
 compare-height (*skip-black* *sx'*) *s' t' rbt.Empty*
 | (*rbt.Empty*, *Branch* - *s'* - - -, *Branch* - *t'* - - -, *Branch* - *tx'* - - -) \Rightarrow
 compare-height *rbt.Empty s' t' (skip-black tx')*
 | - \Rightarrow *EQ*)

declare *compare-height.simps* [code]

hide-type (**open**) *compare*

hide-const (**open**)

compare-height skip-black skip-red LT GT EQ case-compare rec-compare

Abs-compare Rep-compare

hide-fact (**open**)

Abs-compare-cases Abs-compare-induct Abs-compare-inject Abs-compare-inverse

Rep-compare Rep-compare-cases Rep-compare-induct Rep-compare-inject Rep-compare-inverse

compare.simps compare.exhaust compare.induct compare.rec compare.simps

compare.size compare.case-cong compare.case-cong-weak compare.case

compare.nchotomy compare.split compare.split-asm compare.eq.refl compare.eq.simps

equal-compare-def

skip-red.simps skip-red.cases skip-red.induct

skip-black-def

compare-height.simps

129.10 union and intersection of sorted associative lists

context *ord* **begin**

function *sunion-with* :: (*'a* \Rightarrow *'b* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow (*'a* \times *'b*) *list* \Rightarrow (*'a* \times *'b*) *list* \Rightarrow (*'a* \times *'b*) *list*

where

sunion-with *f* ((*k*, *v*) # *as*) ((*k'*, *v'*) # *bs*) =

(if *k* > *k'* then (*k'*, *v'*) # *sunion-with* *f* ((*k*, *v*) # *as*) *bs*)


```

    else if  $k < k'$  then  $(k, v) \# \text{sunion-with } f \text{ as } ((k', v') \# bs)$ 
    else  $(k, f\ k\ v\ v') \# \text{sunion-with } f \text{ as } bs$ 
|  $\text{sunion-with } f\ []\ bs = bs$ 
|  $\text{sunion-with } f \text{ as } [] = as$ 
by pat-completeness auto
termination by lexicographic-order

```

```

function sinter-with :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a  $\times$  'b) list  $\Rightarrow$  ('a  $\times$  'b) list  $\Rightarrow$ 
('a  $\times$  'b) list

```

where

```

    sinter-with  $f\ ((k, v) \# as)\ ((k', v') \# bs) =$ 
    (if  $k > k'$  then sinter-with  $f\ ((k, v) \# as)\ bs$ 
    else if  $k < k'$  then sinter-with  $f\ as\ ((k', v') \# bs)$ 
    else  $(k, f\ k\ v\ v') \# \text{sinter-with } f \text{ as } bs$ )
|  $\text{sinter-with } f\ []\ - = []$ 
|  $\text{sinter-with } f\ -\ [] = []$ 
by pat-completeness auto
termination by lexicographic-order

```

end

```

declare ord.sunion-with.simps [code] ord.sinter-with.simps[code]

```

context linorder begin

lemma set-fst-sunion-with:

```

    set (map fst (sunion-with  $f\ xs\ ys$ )) = set (map fst  $xs$ )  $\cup$  set (map fst  $ys$ )
by(induct  $f\ xs\ ys$  rule: sunion-with.induct) auto

```

lemma sorted-sunion-with [simp]:

```

    [[ sorted (map fst  $xs$ ); sorted (map fst  $ys$ ) ]]
     $\impl$  sorted (map fst (sunion-with  $f\ xs\ ys$ ))
by(induct  $f\ xs\ ys$  rule: sunion-with.induct)
(auto simp add: set-fst-sunion-with simp del: set-map)

```

lemma distinct-sunion-with [simp]:

```

    [[ distinct (map fst  $xs$ ); distinct (map fst  $ys$ ); sorted (map fst  $xs$ ); sorted (map fst
 $ys$ ) ]]
     $\impl$  distinct (map fst (sunion-with  $f\ xs\ ys$ ))
proof(induct  $f\ xs\ ys$  rule: sunion-with.induct)
  case (1  $f\ k\ v\ xs\ k'\ v'\ ys$ )
  have [[  $\neg k < k'$ ;  $\neg k' < k$  ]]  $\impl k = k'$  by simp
  thus ?case using 1
    by(auto simp add: set-fst-sunion-with simp del: set-map)
qed simp-all

```

lemma map-of-sunion-with:

```

    [[ sorted (map fst  $xs$ ); sorted (map fst  $ys$ ) ]]
     $\impl \text{map-of } (\text{sunion-with } f\ xs\ ys)\ k =$ 

```

```

  (case map-of xs k of None  $\Rightarrow$  map-of ys k
   | Some v  $\Rightarrow$  case map-of ys k of None  $\Rightarrow$  Some v
   | Some w  $\Rightarrow$  Some (f k v w))
by(induct f xs ys rule: sunion-with.induct)(auto split: option.split dest: map-of-SomeD
bspec)

```

```

lemma set-fst-sinter-with [simp]:
   $\llbracket$  sorted (map fst xs); sorted (map fst ys)  $\rrbracket$ 
 $\impl$  set (map fst (sinter-with f xs ys)) = set (map fst xs)  $\cap$  set (map fst ys)
by(induct f xs ys rule: sinter-with.induct)(auto simp del: set-map)

```

```

lemma set-fst-sinter-with-subset1:
  set (map fst (sinter-with f xs ys))  $\subseteq$  set (map fst xs)
by(induct f xs ys rule: sinter-with.induct) auto

```

```

lemma set-fst-sinter-with-subset2:
  set (map fst (sinter-with f xs ys))  $\subseteq$  set (map fst ys)
by(induct f xs ys rule: sinter-with.induct)(auto simp del: set-map)

```

```

lemma sorted-sinter-with [simp]:
   $\llbracket$  sorted (map fst xs); sorted (map fst ys)  $\rrbracket$ 
 $\impl$  sorted (map fst (sinter-with f xs ys))
by(induct f xs ys rule: sinter-with.induct)(auto simp del: set-map)

```

```

lemma distinct-sinter-with [simp]:
   $\llbracket$  distinct (map fst xs); distinct (map fst ys)  $\rrbracket$ 
 $\impl$  distinct (map fst (sinter-with f xs ys))
proof(induct f xs ys rule: sinter-with.induct)
  case (1 f k v as k' v' bs)
  have  $\llbracket \neg k < k'; \neg k' < k \rrbracket \implies k = k'$  by simp
  thus ?case using 1 set-fst-sinter-with-subset1 [of f as bs]
    set-fst-sinter-with-subset2 [of f as bs]
  by(auto simp del: set-map)
qed simp-all

```

```

lemma map-of-sinter-with:
   $\llbracket$  sorted (map fst xs); sorted (map fst ys)  $\rrbracket$ 
 $\impl$  map-of (sinter-with f xs ys) k =
  (case map-of xs k of None  $\Rightarrow$  None | Some v  $\Rightarrow$  map-option (f k v) (map-of ys
k))
apply(induct f xs ys rule: sinter-with.induct)
apply(auto simp add: map-option-case split: option.splits dest: map-of-SomeD bspec)
done

```

end

```

lemma distinct-map-of-rev: distinct (map fst xs)  $\impl$  map-of (rev xs) = map-of xs
by(induct xs)(auto 4 3 simp add: map-add-def intro!: ext split: option.split intro:
rev-image-eqI)

```

lemma *map-map-filter*:

$\text{map } f \text{ (List.map-filter } g \text{ xs)} = \text{List.map-filter (map-option } f \circ g) \text{ xs}$
by(*auto simp add: List.map-filter-def*)

lemma *map-filter-map-option-const*:

$\text{List.map-filter } (\lambda x. \text{map-option } (\lambda y. f \ x) \ (g \ (f \ x))) \text{ xs} = \text{filter } (\lambda x. g \ x \neq \text{None})$
 $(\text{map } f \text{ xs})$
by(*auto simp add: map-filter-def filter-map o-def*)

lemma *set-map-filter*: $\text{set (List.map-filter } P \text{ xs)} = \text{the } \cdot (P \cdot \text{set xs} - \{\text{None}\})$
by(*auto simp add: List.map-filter-def intro: rev-image-eqI*)

definition *is-rbt-empty* :: $('a, 'b) \text{rbt} \Rightarrow \text{bool}$ **where**

$\text{is-rbt-empty } t \iff (\text{case } t \text{ of RBT-Impl.Empty} \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

lemma *is-rbt-empty-prop[simp]*: $\text{is-rbt-empty } t \iff t = \text{RBT-Impl.Empty}$

by (*auto simp: is-rbt-empty-def split: RBT-Impl.rbt.splits*)

definition *small-rbt* :: $('a, 'b) \text{rbt} \Rightarrow \text{bool}$ **where**

$\text{small-rbt } t \iff \text{bheight } t < 4$

definition *flip-rbt* :: $('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow \text{bool}$ **where**

$\text{flip-rbt } t1 \ t2 \iff \text{bheight } t2 < \text{bheight } t1$

abbreviation (*input*) *MR* **where** $\text{MR } l \ a \ b \ r \equiv \text{Branch RBT-Impl.R } l \ a \ b \ r$

abbreviation (*input*) *MB* **where** $\text{MB } l \ a \ b \ r \equiv \text{Branch RBT-Impl.B } l \ a \ b \ r$

fun *rbt-baliL* :: $('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**

$\text{rbt-baliL } (\text{MR } (\text{MR } t1 \ a \ b \ t2) \ a' \ b' \ t3) \ a'' \ b'' \ t4 = \text{MR } (\text{MB } t1 \ a \ b \ t2) \ a' \ b' \ (\text{MB } t3 \ a'' \ b'' \ t4)$
 $\mid \text{rbt-baliL } (\text{MR } t1 \ a \ b \ (\text{MR } t2 \ a' \ b' \ t3)) \ a'' \ b'' \ t4 = \text{MR } (\text{MB } t1 \ a \ b \ t2) \ a' \ b' \ (\text{MB } t3 \ a'' \ b'' \ t4)$
 $\mid \text{rbt-baliL } t1 \ a \ b \ t2 = \text{MB } t1 \ a \ b \ t2$

fun *rbt-baliR* :: $('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**

$\text{rbt-baliR } t1 \ a \ b \ (\text{MR } t2 \ a' \ b' \ (\text{MR } t3 \ a'' \ b'' \ t4)) = \text{MR } (\text{MB } t1 \ a \ b \ t2) \ a' \ b' \ (\text{MB } t3 \ a'' \ b'' \ t4)$
 $\mid \text{rbt-baliR } t1 \ a \ b \ (\text{MR } (\text{MR } t2 \ a' \ b' \ t3) \ a'' \ b'' \ t4) = \text{MR } (\text{MB } t1 \ a \ b \ t2) \ a' \ b' \ (\text{MB } t3 \ a'' \ b'' \ t4)$
 $\mid \text{rbt-baliR } t1 \ a \ b \ t2 = \text{MB } t1 \ a \ b \ t2$

fun *rbt-baldL* :: $('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**

$\text{rbt-baldL } (\text{MR } t1 \ a \ b \ t2) \ a' \ b' \ t3 = \text{MR } (\text{MB } t1 \ a \ b \ t2) \ a' \ b' \ t3$
 $\mid \text{rbt-baldL } t1 \ a \ b \ (\text{MB } t2 \ a' \ b' \ t3) = \text{rbt-baliR } t1 \ a \ b \ (\text{MR } t2 \ a' \ b' \ t3)$
 $\mid \text{rbt-baldL } t1 \ a \ b \ (\text{MR } (\text{MB } t2 \ a' \ b' \ t3) \ a'' \ b'' \ t4) =$
 $\text{MR } (\text{MB } t1 \ a \ b \ t2) \ a' \ b' \ (\text{rbt-baliR } t3 \ a'' \ b'' \ (\text{paint RBT-Impl.R } t4))$

| *rbt-baldL* *t1 a b t2* = *MR t1 a b t2*

fun *rbt-baldR* :: ('a, 'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* **where**
rbt-baldR t1 a b (MR t2 a' b' t3) = *MR t1 a b (MB t2 a' b' t3)*
 | *rbt-baldR (MB t1 a b t2) a' b' t3* = *rbt-baliL (MR t1 a b t2) a' b' t3*
 | *rbt-baldR (MR t1 a b (MB t2 a' b' t3)) a'' b'' t4* =
MR (rbt-baliL (paint RBT-Impl.R t1) a b t2) a' b' (MB t3 a'' b'' t4)
 | *rbt-baldR t1 a b t2* = *MR t1 a b t2*

fun *rbt-app* :: ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* **where**
rbt-app RBT-Impl.Empty t = *t*
 | *rbt-app t RBT-Impl.Empty* = *t*
 | *rbt-app (MR t1 a b t2) (MR t3 a'' b'' t4)* = (case *rbt-app t2 t3* of
MR u2 a' b' u3 ⇒ (*MR (MR t1 a b u2) a' b' (MR u3 a'' b'' t4)*)
 | *t23* ⇒ *MR t1 a b (MR t23 a'' b'' t4)*)
 | *rbt-app (MB t1 a b t2) (MB t3 a'' b'' t4)* = (case *rbt-app t2 t3* of
MR u2 a' b' u3 ⇒ *MR (MB t1 a b u2) a' b' (MB u3 a'' b'' t4)*
 | *t23* ⇒ *rbt-baldL t1 a b (MB t23 a'' b'' t4)*)
 | *rbt-app t1 (MR t2 a b t3)* = *MR (rbt-app t1 t2) a b t3*
 | *rbt-app (MR t1 a b t2) t3* = *MR t1 a b (rbt-app t2 t3)*

fun *rbt-joinL* :: ('a, 'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* **where**
rbt-joinL l a b r = (if *bheight l* ≥ *bheight r* then *MR l a b r*
 else case *r* of *MB l' a' b' r'* ⇒ *rbt-baliL (rbt-joinL l a b l') a' b' r'*
 | *MR l' a' b' r'* ⇒ *MR (rbt-joinL l a b l') a' b' r'*)

declare *rbt-joinL.simps*[*simp del*]

fun *rbt-joinR* :: ('a, 'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* **where**
rbt-joinR l a b r = (if *bheight l* ≤ *bheight r* then *MR l a b r*
 else case *l* of *MB l' a' b' r'* ⇒ *rbt-baliR l' a' b' (rbt-joinR r' a b r)*
 | *MR l' a' b' r'* ⇒ *MR l' a' b' (rbt-joinR r' a b r)*)

declare *rbt-joinR.simps*[*simp del*]

definition *rbt-join* :: ('a, 'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a, 'b) *rbt* ⇒ ('a, 'b) *rbt* **where**
rbt-join l a b r =
 (let *bhl* = *bheight l*; *bhr* = *bheight r*
 in if *bhl* > *bhr*
 then *paint RBT-Impl.B (rbt-joinR l a b r)*
 else if *bhl* < *bhr*
 then *paint RBT-Impl.B (rbt-joinL l a b r)*
 else *MB l a b r*)

lemma *size-paint*[*simp*]: *size (paint c t)* = *size t*
by (*cases t*) *auto*

lemma *size-baliL*[*simp*]: *size (rbt-baliL t1 a b t2)* = *Suc (size t1 + size t2)*
by (*induction t1 a b t2 rule: rbt-baliL.induct*) *auto*

lemma *size-baliR[simp]*: $\text{size } (\text{rbt-baliR } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$
by (*induction* $t1 \ a \ b \ t2$ *rule*: *rbt-baliR.induct*) *auto*

lemma *size-baldL[simp]*: $\text{size } (\text{rbt-baldL } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$
by (*induction* $t1 \ a \ b \ t2$ *rule*: *rbt-baldL.induct*) *auto*

lemma *size-baldR[simp]*: $\text{size } (\text{rbt-baldR } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$
by (*induction* $t1 \ a \ b \ t2$ *rule*: *rbt-baldR.induct*) *auto*

lemma *size-rbt-app[simp]*: $\text{size } (\text{rbt-app } t1 \ t2) = \text{size } t1 + \text{size } t2$
by (*induction* $t1 \ t2$ *rule*: *rbt-app.induct*)
(auto split: RBT-Impl.rbt.splits RBT-Impl.color.splits)

lemma *size-rbt-joinL[simp]*: $\text{size } (\text{rbt-joinL } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$
by (*induction* $t1 \ a \ b \ t2$ *rule*: *rbt-joinL.induct*)
(auto simp: rbt-joinL.simps split: RBT-Impl.rbt.splits RBT-Impl.color.splits)

lemma *size-rbt-joinR[simp]*: $\text{size } (\text{rbt-joinR } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$
by (*induction* $t1 \ a \ b \ t2$ *rule*: *rbt-joinR.induct*)
(auto simp: rbt-joinR.simps split: RBT-Impl.rbt.splits RBT-Impl.color.splits)

lemma *size-rbt-join[simp]*: $\text{size } (\text{rbt-join } t1 \ a \ b \ t2) = \text{Suc } (\text{size } t1 + \text{size } t2)$
by (*auto simp: rbt-join-def Let-def*)

definition *inv-12* $t \longleftrightarrow \text{inv1 } t \wedge \text{inv2 } t$

lemma *rbt-Node*: $\text{inv-12 } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \Longrightarrow \text{inv-12 } l \wedge \text{inv-12 } r$
by (*auto simp: inv-12-def*)

lemma *paint2*: $\text{paint } c2 \ (\text{paint } c1 \ t) = \text{paint } c2 \ t$
by (*cases* t) *auto*

lemma *inv1-rbt-baliL*: $\text{inv1 } l \Longrightarrow \text{inv1 } r \Longrightarrow \text{inv1 } (\text{rbt-baliL } l \ a \ b \ r)$
by (*induct* $l \ a \ b \ r$ *rule*: *rbt-baliL.induct*) *auto*

lemma *inv1-rbt-baliR*: $\text{inv1 } l \Longrightarrow \text{inv1 } r \Longrightarrow \text{inv1 } (\text{rbt-baliR } l \ a \ b \ r)$
by (*induct* $l \ a \ b \ r$ *rule*: *rbt-baliR.induct*) *auto*

lemma *rbt-bheight-rbt-baliL*: $\text{bheight } l = \text{bheight } r \Longrightarrow \text{bheight } (\text{rbt-baliL } l \ a \ b \ r) = \text{Suc } (\text{bheight } l)$
by (*induct* $l \ a \ b \ r$ *rule*: *rbt-baliL.induct*) *auto*

lemma *rbt-bheight-rbt-baliR*: $\text{bheight } l = \text{bheight } r \Longrightarrow \text{bheight } (\text{rbt-baliR } l \ a \ b \ r) = \text{Suc } (\text{bheight } l)$
by (*induct* $l \ a \ b \ r$ *rule*: *rbt-baliR.induct*) *auto*

lemma *inv2-rbt-baliL*: $\text{inv2 } l \Longrightarrow \text{inv2 } r \Longrightarrow \text{bheight } l = \text{bheight } r \Longrightarrow \text{inv2 } (\text{rbt-baliL } l \ a \ b \ r)$

by (*induct l a b r rule: rbt-baliL.induct*) *auto*

lemma *inv2-rbt-baliR*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r \implies \text{inv2 } (\text{rbt-baliR } l \ a \ b \ r)$

by (*induct l a b r rule: rbt-baliR.induct*) *auto*

lemma *inv-rbt-baliR*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{inv1 } l \implies \text{inv1 } r \implies \text{bheight } l = \text{bheight } r \implies$

$\text{inv1 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{inv2 } (\text{rbt-baliR } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baliR } l \ a \ b \ r) = \text{Suc } (\text{bheight } l)$

by (*induct l a b r rule: rbt-baliR.induct*) *auto*

lemma *inv-rbt-baliL*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{inv1 } l \implies \text{inv1 } r \implies \text{bheight } l = \text{bheight } r \implies$

$\text{inv1 } (\text{rbt-baliL } l \ a \ b \ r) \wedge \text{inv2 } (\text{rbt-baliL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baliL } l \ a \ b \ r) = \text{Suc } (\text{bheight } l)$

by (*induct l a b r rule: rbt-baliL.induct*) *auto*

lemma *inv2-rbt-baldL-inv1*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l + 1 = \text{bheight } r \implies \text{inv1 } r \implies$

$\text{inv2 } (\text{rbt-baldL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldL } l \ a \ b \ r) = \text{bheight } r$

by (*induct l a b r rule: rbt-baldL.induct*) (*auto simp: inv2-rbt-baliR rbt-bheight-rbt-baliR*)

lemma *inv2-rbt-baldL-B*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l + 1 = \text{bheight } r \implies \text{color-of } r = \text{RBT-Impl.B} \implies$

$\text{inv2 } (\text{rbt-baldL } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldL } l \ a \ b \ r) = \text{bheight } r$

by (*induct l a b r rule: rbt-baldL.induct*) (*auto simp add: inv2-rbt-baliR rbt-bheight-rbt-baliR*)

lemma *inv1-rbt-baldL*: $\text{inv1 } l \implies \text{inv1 } r \implies \text{color-of } r = \text{RBT-Impl.B} \implies \text{inv1 } (\text{rbt-baldL } l \ a \ b \ r)$

by (*induct l a b r rule: rbt-baldL.induct*) (*simp-all add: inv1-rbt-baliR*)

lemma *inv1l*: $\text{inv1 } t \implies \text{inv1 } l \ t$

by (*cases t*) *auto*

lemma *neg-Black[simp]*: $(c \neq \text{RBT-Impl.B}) = (c = \text{RBT-Impl.R})$

by (*cases c*) *auto*

lemma *inv1l-rbt-baldL*: $\text{inv1 } l \implies \text{inv1 } r \implies \text{inv1 } (\text{rbt-baldL } l \ a \ b \ r)$

by (*induct l a b r rule: rbt-baldL.induct*) (*auto simp: inv1-rbt-baliR paint2*)

lemma *inv2-rbt-baldR-inv1*: $\text{inv2 } l \implies \text{inv2 } r \implies \text{bheight } l = \text{bheight } r + 1 \implies \text{inv1 } l \implies$

$\text{inv2 } (\text{rbt-baldR } l \ a \ b \ r) \wedge \text{bheight } (\text{rbt-baldR } l \ a \ b \ r) = \text{bheight } l$

by (*induct l a b r rule: rbt-baldR.induct*) (*auto simp: inv2-rbt-baliL rbt-bheight-rbt-baliL*)

lemma *inv1-rbt-baldR*: $\text{inv1 } l \implies \text{inv1 } r \implies \text{color-of } l = \text{RBT-Impl.B} \implies \text{inv1 } (\text{rbt-baldR } l \ a \ b \ r)$

by (*induct l a b r rule: rbt-baldR.induct*) (*simp-all add: inv1-rbt-baliL*)

lemma *inv1l-rbt-baldR*: $inv1\ l \implies inv1\ r \implies inv1\ (rbt-baldR\ l\ a\ b\ r)$
by (*induct* $l\ a\ b\ r$ *rule*: *rbt-baldR.induct*) (*auto simp*: *inv1-rbt-baliL paint2*)

lemma *inv2-rbt-app*: $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r \implies$
 $inv2\ (rbt-app\ l\ r) \wedge bheight\ (rbt-app\ l\ r) = bheight\ l$
by (*induct* $l\ r$ *rule*: *rbt-app.induct*)
(*auto simp*: *inv2-rbt-baldL-B split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *inv1-rbt-app*: $inv1\ l \implies inv1\ r \implies (color-of\ l = RBT-Impl.B \wedge$
 $color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-app\ l\ r)) \wedge inv1\ (rbt-app\ l\ r)$
by (*induct* $l\ r$ *rule*: *rbt-app.induct*)
(*auto simp*: *inv1-rbt-baldL split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *inv-rbt-baldL*: $inv2\ l \implies inv2\ r \implies bheight\ l + 1 = bheight\ r \implies inv1\ l$
 $\implies inv1\ r \implies$
 $inv2\ (rbt-baldL\ l\ a\ b\ r) \wedge bheight\ (rbt-baldL\ l\ a\ b\ r) = bheight\ r \wedge$
 $inv1\ (rbt-baldL\ l\ a\ b\ r) \wedge (color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-baldL\ l\ a\ b$
 $r))$
by (*induct* $l\ a\ b\ r$ *rule*: *rbt-baldL.induct*) (*auto simp*: *inv-rbt-baliR rbt-bheight-rbt-baliR*
paint2)

lemma *inv-rbt-baldR*: $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r + 1 \implies inv1\ l$
 $\implies inv1\ r \implies$
 $inv2\ (rbt-baldR\ l\ a\ b\ r) \wedge bheight\ (rbt-baldR\ l\ a\ b\ r) = bheight\ l \wedge$
 $inv1\ (rbt-baldR\ l\ a\ b\ r) \wedge (color-of\ l = RBT-Impl.B \longrightarrow inv1\ (rbt-baldR\ l\ a\ b$
 $r))$
by (*induct* $l\ a\ b\ r$ *rule*: *rbt-baldR.induct*) (*auto simp*: *inv-rbt-baliL rbt-bheight-rbt-baliL*
paint2)

lemma *inv-rbt-app*: $inv2\ l \implies inv2\ r \implies bheight\ l = bheight\ r \implies inv1\ l \implies$
 $inv1\ r \implies$
 $inv2\ (rbt-app\ l\ r) \wedge bheight\ (rbt-app\ l\ r) = bheight\ l \wedge$
 $inv1\ (rbt-app\ l\ r) \wedge (color-of\ l = RBT-Impl.B \wedge color-of\ r = RBT-Impl.B \longrightarrow$
 $inv1\ (rbt-app\ l\ r))$
by (*induct* $l\ r$ *rule*: *rbt-app.induct*)
(*auto simp*: *inv2-rbt-baldL-B inv-rbt-baldL split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *inv1l-rbt-joinL*: $inv1\ l \implies inv1\ r \implies bheight\ l \leq bheight\ r \implies$
 $inv1\ (rbt-joinL\ l\ a\ b\ r) \wedge$
 $(bheight\ l \neq bheight\ r \wedge color-of\ r = RBT-Impl.B \longrightarrow inv1\ (rbt-joinL\ l\ a\ b\ r))$

proof (*induct* $l\ a\ b\ r$ *rule*: *rbt-joinL.induct*)

case ($1\ l\ a\ b\ r$)

then show ?*case*

by (*auto simp*: *inv1-rbt-baliL rbt-joinL.simps*[*of* $l\ a\ b\ r$]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

qed

lemma *inv1l-rbt-joinR*: $inv1\ l \implies inv2\ l \implies inv1\ r \implies inv2\ r \implies bheight\ l \geq$

$bheight\ r \implies$
 $inv1l\ (rbt-joinR\ l\ a\ b\ r) \wedge$
 $(bheight\ l \neq bheight\ r \wedge color-of\ l = RBT-Impl.B \implies inv1\ (rbt-joinR\ l\ a\ b\ r))$
proof (*induct* $l\ a\ b\ r$ rule: *rbt-joinR.induct*)
case ($1\ l\ a\ b\ r$)
then show ?*case*
by (*fastforce simp: inv1-rbt-baliR rbt-joinR.simps[of l a b r]*
split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)
qed

lemma *bheight-rbt-joinL*: $inv2\ l \implies inv2\ r \implies bheight\ l \leq bheight\ r \implies$
 $bheight\ (rbt-joinL\ l\ a\ b\ r) = bheight\ r$
proof (*induct* $l\ a\ b\ r$ rule: *rbt-joinL.induct*)
case ($1\ l\ a\ b\ r$)
then show ?*case*
by (*auto simp: rbt-bheight-rbt-baliL rbt-joinL.simps[of l a b r]*
split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)
qed

lemma *inv2-rbt-joinL*: $inv2\ l \implies inv2\ r \implies bheight\ l \leq bheight\ r \implies inv2$
 $(rbt-joinL\ l\ a\ b\ r)$
proof (*induct* $l\ a\ b\ r$ rule: *rbt-joinL.induct*)
case ($1\ l\ a\ b\ r$)
then show ?*case*
by (*auto simp: inv2-rbt-baliL bheight-rbt-joinL rbt-joinL.simps[of l a b r]*
split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)
qed

lemma *bheight-rbt-joinR*: $inv2\ l \implies inv2\ r \implies bheight\ l \geq bheight\ r \implies$
 $bheight\ (rbt-joinR\ l\ a\ b\ r) = bheight\ l$
proof (*induct* $l\ a\ b\ r$ rule: *rbt-joinR.induct*)
case ($1\ l\ a\ b\ r$)
then show ?*case*
by (*fastforce simp: rbt-bheight-rbt-baliR rbt-joinR.simps[of l a b r]*
split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)
qed

lemma *inv2-rbt-joinR*: $inv2\ l \implies inv2\ r \implies bheight\ l \geq bheight\ r \implies inv2$
 $(rbt-joinR\ l\ a\ b\ r)$
proof (*induct* $l\ a\ b\ r$ rule: *rbt-joinR.induct*)
case ($1\ l\ a\ b\ r$)
then show ?*case*
by (*fastforce simp: inv2-rbt-baliR bheight-rbt-joinR rbt-joinR.simps[of l a b r]*
split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)
qed

lemma *keys-paint[simp]*: $RBT-Impl.keys\ (paint\ c\ t) = RBT-Impl.keys\ t$
by (*cases t*) *auto*

lemma *keys-rbt-baliL*: $RBT-Impl.keys (rbt-baliL\ l\ a\ b\ r) = RBT-Impl.keys\ l\ @\ a$
 $\# RBT-Impl.keys\ r$

by (cases (l,a,b,r) rule: rbt-baliL.cases) auto

lemma *keys-rbt-baliR*: $RBT-Impl.keys (rbt-baliR\ l\ a\ b\ r) = RBT-Impl.keys\ l\ @\ a$
 $\# RBT-Impl.keys\ r$

by (cases (l,a,b,r) rule: rbt-baliR.cases) auto

lemma *keys-rbt-baldL*: $RBT-Impl.keys (rbt-baldL\ l\ a\ b\ r) = RBT-Impl.keys\ l\ @\ a$
 $\# RBT-Impl.keys\ r$

by (cases (l,a,b,r) rule: rbt-baldL.cases) (auto simp: keys-rbt-baliL keys-rbt-baliR)

lemma *keys-rbt-baldR*: $RBT-Impl.keys (rbt-baldR\ l\ a\ b\ r) = RBT-Impl.keys\ l\ @\ a$
 $\# RBT-Impl.keys\ r$

by (cases (l,a,b,r) rule: rbt-baldR.cases) (auto simp: keys-rbt-baliL keys-rbt-baliR)

lemma *keys-rbt-app*: $RBT-Impl.keys (rbt-app\ l\ r) = RBT-Impl.keys\ l\ @\ RBT-Impl.keys\ r$

by (induction l r rule: rbt-app.induct)

(auto simp: keys-rbt-baldL keys-rbt-baldR split: RBT-Impl.rbt.splits RBT-Impl.color.splits)

lemma *keys-rbt-joinL*: $bheight\ l \leq bheight\ r \implies$

$RBT-Impl.keys (rbt-joinL\ l\ a\ b\ r) = RBT-Impl.keys\ l\ @\ a \# RBT-Impl.keys\ r$

proof (induction l a b r rule: rbt-joinL.induct)

case (1 l a b r)

thus ?case

by (auto simp: keys-rbt-baliL rbt-joinL.simps[of l a b r])

split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)

qed

lemma *keys-rbt-joinR*: $RBT-Impl.keys (rbt-joinR\ l\ a\ b\ r) = RBT-Impl.keys\ l\ @\ a$
 $\# RBT-Impl.keys\ r$

proof (induction l a b r rule: rbt-joinR.induct)

case (1 l a b r)

thus ?case

by (force simp: keys-rbt-baliR rbt-joinR.simps[of l a b r])

split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)

qed

lemma *rbt-set-rbt-baliL*: $set (RBT-Impl.keys (rbt-baliL\ l\ a\ b\ r)) =$

$set (RBT-Impl.keys\ l) \cup \{a\} \cup set (RBT-Impl.keys\ r)$

by (cases (l,a,b,r) rule: rbt-baliL.cases) auto

lemma *set-rbt-joinL*: $set (RBT-Impl.keys (rbt-joinL\ l\ a\ b\ r)) =$

$set (RBT-Impl.keys\ l) \cup \{a\} \cup set (RBT-Impl.keys\ r)$

proof (induction l a b r rule: rbt-joinL.induct)

case (1 l a b r)

thus ?case

by (auto simp: rbt-set-rbt-baliL rbt-joinL.simps[of l a b r])

split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *rbt-set-rbt-baliR*: *set (RBT-Impl.keys (rbt-baliR l a b r)) =*
set (RBT-Impl.keys l) \cup {a} \cup set (RBT-Impl.keys r)
by (*cases (l,a,b,r) rule: rbt-baliR.cases*) *auto*

lemma *set-rbt-joinR*: *set (RBT-Impl.keys (rbt-joinR l a b r)) =*
set (RBT-Impl.keys l) \cup {a} \cup set (RBT-Impl.keys r)
proof (*induction l a b r rule: rbt-joinR.induct*)
case (*1 l a b r*)
thus *?case*
by (*force simp: rbt-set-rbt-baliR rbt-joinR.simps[of l a b r]*
split!: RBT-Impl.rbt.splits RBT-Impl.color.splits)
qed

lemma *set-keys-paint*: *set (RBT-Impl.keys (paint c t)) = set (RBT-Impl.keys t)*
by (*cases t*) *auto*

lemma *set-rbt-join*: *set (RBT-Impl.keys (rbt-join l a b r)) =*
set (RBT-Impl.keys l) \cup {a} \cup set (RBT-Impl.keys r)
by (*simp add: set-rbt-joinL set-rbt-joinR set-keys-paint rbt-join-def Let-def*)

lemma *inv-rbt-join*: *inv-12 l \implies inv-12 r \implies inv-12 (rbt-join l a b r)*
by (*auto simp: rbt-join-def Let-def inv1l-rbt-joinL inv1l-rbt-joinR*
inv2-rbt-joinL inv2-rbt-joinR inv-12-def)

fun *rbt-recolor* :: (*'a, 'b*) *rbt* \Rightarrow (*'a, 'b*) *rbt* **where**
rbt-recolor (Branch RBT-Impl.R t1 k v t2) =
(if color-of t1 = RBT-Impl.B \wedge color-of t2 = RBT-Impl.B then Branch
RBT-Impl.B t1 k v t2
else Branch RBT-Impl.R t1 k v t2)
| *rbt-recolor t = t*

lemma *rbt-recolor*: *inv-12 t \implies inv-12 (rbt-recolor t)*
by (*induction t rule: rbt-recolor.induct*) (*auto simp: inv-12-def*)

fun *rbt-split-min* :: (*'a, 'b*) *rbt* \Rightarrow *'a \times 'b \times ('a, 'b) rbt* **where**
rbt-split-min RBT-Impl.Empty = undefined
| *rbt-split-min (RBT-Impl.Branch - l a b r) =*
(if is-rbt-empty l then (a,b,r) else let (a',b',l') = rbt-split-min l in (a',b',rbt-join
l' a b r))

lemma *rbt-split-min-set*:
rbt-split-min t = (a,b,t') \implies t \neq RBT-Impl.Empty \implies
a \in set (RBT-Impl.keys t) \wedge set (RBT-Impl.keys t) = {a} \cup set (RBT-Impl.keys
t')
by (*induction t arbitrary: t'*) (*auto simp: set-rbt-join split: prod.splits if-splits*)

lemma *rbt-split-min-inv*: $\text{rbt-split-min } t = (a, b, t') \implies \text{inv-12 } t \implies t \neq \text{RBT-Impl.Empty} \implies \text{inv-12 } t'$

by (*induction* t *arbitrary*: t')
(auto simp: inv-rbt-join split: if-splits prod.splits dest: rbt-Node)

definition *rbt-join2* :: $('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**
 $\text{rbt-join2 } l \ r = (\text{if is-rbt-empty } r \text{ then } l \text{ else let } (a, b, r') = \text{rbt-split-min } r \text{ in rbt-join } l \ a \ b \ r')$

lemma *set-rbt-join2[simp]*: $\text{set } (\text{RBT-Impl.keys } (\text{rbt-join2 } l \ r)) = \text{set } (\text{RBT-Impl.keys } l) \cup \text{set } (\text{RBT-Impl.keys } r)$
by (*simp add: rbt-join2-def rbt-split-min-set set-rbt-join split: prod.split*)

lemma *inv-rbt-join2*: $\text{inv-12 } l \implies \text{inv-12 } r \implies \text{inv-12 } (\text{rbt-join2 } l \ r)$
by (*simp add: rbt-join2-def inv-rbt-join rbt-split-min-set rbt-split-min-inv split: prod.split*)

context *ord* **begin**

fun *rbt-split* :: $('a, 'b) \text{rbt} \Rightarrow 'a \Rightarrow ('a, 'b) \text{rbt} \times 'b \text{ option} \times ('a, 'b) \text{rbt}$ **where**
 $\text{rbt-split } \text{RBT-Impl.Empty } k = (\text{RBT-Impl.Empty}, \text{None}, \text{RBT-Impl.Empty})$
 $| \text{rbt-split } (\text{RBT-Impl.Branch } - \ l \ a \ b \ r) \ x =$
 $(\text{if } x < a \text{ then } (\text{case } \text{rbt-split } l \ x \text{ of } (l1, \beta, l2) \Rightarrow (l1, \beta, \text{rbt-join } l2 \ a \ b \ r))$
 $\text{else if } a < x \text{ then } (\text{case } \text{rbt-split } r \ x \text{ of } (r1, \beta, r2) \Rightarrow (\text{rbt-join } l \ a \ b \ r1, \beta, r2))$
 $\text{else } (l, \text{Some } b, r))$

lemma *rbt-split*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{inv-12 } t \implies \text{inv-12 } l \wedge \text{inv-12 } r$
by (*induction* t *arbitrary*: $l \ r$)
(auto simp: set-rbt-join inv-rbt-join rbt-greater-prop rbt-less-prop split: if-splits prod.splits dest!: rbt-Node)

lemma *rbt-split-size*: $(l2, \beta, r2) = \text{rbt-split } t2 \ a \implies \text{size } l2 + \text{size } r2 \leq \text{size } t2$
by (*induction* $t2 \ a$ *arbitrary*: $l2 \ r2$ *rule: rbt-split.induct*) *(auto split: if-splits prod.splits)*

function *rbt-union-rec* :: $('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**
 $\text{rbt-union-rec } f \ t1 \ t2 = (\text{let } (f, t2, t1) =$
 $\text{if flip-rbt } t2 \ t1 \text{ then } (\lambda k \ v \ v'. f \ k \ v' \ v, t1, t2) \text{ else } (f, t2, t1) \text{ in}$
 $\text{if small-rbt } t2 \text{ then } \text{RBT-Impl.fold } (\text{rbt-insert-with-key } f) \ t2 \ t1$
 $\text{else } (\text{case } t1 \text{ of } \text{RBT-Impl.Empty} \Rightarrow t2$
 $| \text{RBT-Impl.Branch } - \ l1 \ a \ b \ r1 \Rightarrow$
 $\text{case } \text{rbt-split } t2 \ a \text{ of } (l2, \beta, r2) \Rightarrow$
 $\text{rbt-join } (\text{rbt-union-rec } f \ l1 \ l2) \ a \ (\text{case } \beta \text{ of } \text{None} \Rightarrow b \mid \text{Some } b' \Rightarrow f \ a \ b$
 $b') \ (\text{rbt-union-rec } f \ r1 \ r2)))$
by *pat-completeness auto*

termination
using *rbt-split-size*
by (*relation measure* $(\lambda(f, t1, t2). \text{size } t1 + \text{size } t2)$) *(fastforce split: if-splits)+*

declare *rbt-union-rec.simps*[simp del]

function *rbt-union-swap-rec* :: (*'a* \Rightarrow *'b* \Rightarrow *'b* \Rightarrow *'b*) \Rightarrow *bool* \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow (*'a*, *'b*) *rbt* \Rightarrow (*'a*, *'b*) *rbt* **where**
rbt-union-swap-rec *f* γ *t1* *t2* = (let (γ , *t2*, *t1*) =
 if *flip-rbt* *t2* *t1* then ($\neg\gamma$, *t1*, *t2*) else (γ , *t2*, *t1*);
f' = (if γ then ($\lambda k v v'. f k v' v$) else *f*) in
 if *small-rbt* *t2* then *RBT-Impl.fold* (*rbt-insert-with-key* *f'*) *t2* *t1*
 else (case *t1* of *RBT-Impl.Empty* \Rightarrow *t2*
 | *RBT-Impl.Branch* - *l1* *a* *b* *r1* \Rightarrow
 case *rbt-split* *t2* *a* of (*l2*, β , *r2*) \Rightarrow
rbt-join (*rbt-union-swap-rec* *f* γ *l1* *l2*) *a* (case β of *None* \Rightarrow *b* | *Some* *b'* \Rightarrow
f' a b b') (*rbt-union-swap-rec* *f* γ *r1* *r2*)))
by *pat-completeness* *auto*
termination
using *rbt-split-size*
by (*relation* *measure* ($\lambda(f,\gamma,t1,t2). \text{size } t1 + \text{size } t2$)) (*fastforce* *split: if-splits*)+

declare *rbt-union-swap-rec.simps*[simp del]

lemma *rbt-union-swap-rec: rbt-union-swap-rec* *f* γ *t1* *t2* =
rbt-union-rec (if γ then ($\lambda k v v'. f k v' v$) else *f*) *t1* *t2*
proof (*induction* *f* γ *t1* *t2* *rule: rbt-union-swap-rec.induct*)
 case (*1 f* γ *t1* *t2*)
 show ?case
 using *1* [*OF refl - refl refl - refl - refl*]
 unfolding *rbt-union-swap-rec.simps*[*of - - t1*] *rbt-union-rec.simps*[*of - t1*]
 by (*auto simp: Let-def split: rbt.splits prod.splits option.splits*)
qed

lemma *rbt-fold-rbt-insert:*
assumes *inv-12* *t2*
shows *inv-12* (*RBT-Impl.fold* (*rbt-insert-with-key* *f*) *t1* *t2*)
proof –
 define *xs* **where** *xs* = *RBT-Impl.entries* *t1*
from *assms* **show** ?thesis
 unfolding *RBT-Impl.fold-def* *xs-def*[*symmetric*]
 by (*induct* *xs* *rule: rev-induct*)
 (*auto simp: inv-12-def rbt-insert-with-key-def ins-inv1-inv2*)
qed

lemma *rbt-union-rec: inv-12* *t1* \Longrightarrow *inv-12* *t2* \Longrightarrow *inv-12* (*rbt-union-rec* *f* *t1* *t2*)
proof (*induction* *f* *t1* *t2* *rule: rbt-union-rec.induct*)
 case (*1 t1* *t2*)
 thus ?case
 by (*auto simp: rbt-union-rec.simps*[*of t1 t2*] *inv-rbt-join* *rbt-split* *rbt-fold-rbt-insert*
split!: RBT-Impl.rbt.splits RBT-Impl.color.splits prod.split if-splits dest:
rbt-Node)

qed

definition *map-filter-inter* $f\ t1\ t2 = \text{List.map-filter } (\lambda(k, v).$

case rbt-lookup t1 k of None \Rightarrow None

| Some v' \Rightarrow Some (k, f k v' v)) (RBT-Impl.entries t2)

function *rbt-inter-rec* $:: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)\ \text{rbt} \Rightarrow ('a, 'b)\ \text{rbt} \Rightarrow ('a,$
'b)\ \text{rbt where

rbt-inter-rec f t1 t2 = (let (f, t2, t1) =

if flip-rbt t2 t1 then $(\lambda k\ v\ v'.\ f\ k\ v'\ v,\ t1,\ t2)$ else $(f,\ t2,\ t1)$ in

if small-rbt t2 then rbtreeify (map-filter-inter f t1 t2)

else case t1 of RBT-Impl.Empty \Rightarrow RBT-Impl.Empty

| RBT-Impl.Branch - l1 a b r1 \Rightarrow

case rbt-split t2 a of $(l2, \beta, r2) \Rightarrow$ let $l' = \text{rbt-inter-rec } f\ l1\ l2$; $r' = \text{rbt-inter-rec } f\ r1\ r2$ in

(case β of None \Rightarrow rbt-join2 l' r' | Some b' \Rightarrow rbt-join l' a (f a b b') r'))

by pat-completeness auto

termination

using rbt-split-size

by (relation measure $(\lambda(f, t1, t2). \text{size } t1 + \text{size } t2))$ (fastforce split: if-splits)+

declare *rbt-inter-rec.simps[simp del]*

function *rbt-inter-swap-rec* $:: ('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow \text{bool} \Rightarrow ('a, 'b)\ \text{rbt} \Rightarrow ('a,$
'b)\ \text{rbt} \Rightarrow ('a, 'b)\ \text{rbt where

rbt-inter-swap-rec f γ t1 t2 = (let $(\gamma, t2, t1) =$

if flip-rbt t2 t1 then $(\neg\gamma, t1, t2)$ else $(\gamma, t2, t1)$;

f' = (if γ then $(\lambda k\ v\ v'.\ f\ k\ v'\ v)$ else f) in

if small-rbt t2 then rbtreeify (map-filter-inter f' t1 t2)

else case t1 of RBT-Impl.Empty \Rightarrow RBT-Impl.Empty

| RBT-Impl.Branch - l1 a b r1 \Rightarrow

case rbt-split t2 a of $(l2, \beta, r2) \Rightarrow$ let $l' = \text{rbt-inter-swap-rec } f\ \gamma\ l1\ l2$; $r' = \text{rbt-inter-swap-rec } f\ \gamma\ r1\ r2$ in

(case β of None \Rightarrow rbt-join2 l' r' | Some b' \Rightarrow rbt-join l' a (f' a b b') r'))

by pat-completeness auto

termination

using rbt-split-size

by (relation measure $(\lambda(f, \gamma, t1, t2). \text{size } t1 + \text{size } t2))$ (fastforce split: if-splits)+

declare *rbt-inter-swap-rec.simps[simp del]*

lemma *rbt-inter-swap-rec: rbt-inter-swap-rec f γ t1 t2 =*

rbt-inter-rec (if γ then $(\lambda k\ v\ v'.\ f\ k\ v'\ v)$ else f) t1 t2

proof (*induction f γ t1 t2 rule: rbt-inter-swap-rec.induct*)

case (1 f γ t1 t2)

show ?case

using 1[OF refl - refl refl - refl - refl]

unfolding rbt-inter-swap-rec.simps[of - - t1] rbt-inter-rec.simps[of - t1]

by (auto simp add: Let-def split: rbt.splits prod.splits option.splits)

qed

lemma *rbt-rbtreeify[simp]: inv-12 (rbtreeify kvs)*

by (*simp add: inv-12-def rbtreeify-def inv1-rbtreeify-g inv2-rbtreeify-g*)

lemma *rbt-inter-rec: inv-12 t1 \implies inv-12 t2 \implies inv-12 (rbt-inter-rec f t1 t2)*

proof(*induction f t1 t2 rule: rbt-inter-rec.induct*)

case (*1 t1 t2*)

thus *?case*

by (*auto simp: rbt-inter-rec.simps[of t1 t2] inv-rbt-join inv-rbt-join2 rbt-split*

Let-def

*split!: RBT-Impl.rbt.splits RBT-Impl.color.splits prod.split if-splits
option.splits dest!: rbt-Node*)

qed

definition *filter-minus t1 t2 = filter ($\lambda(k, -). \text{rbt-lookup } t2 \ k = \text{None}$) (RBT-Impl.entries t1)*

fun *rbt-minus-rec :: ('a, 'b) rbt \Rightarrow ('a, 'b) rbt \Rightarrow ('a, 'b) rbt* **where**

rbt-minus-rec t1 t2 = (if small-rbt t2 then RBT-Impl.fold ($\lambda k - t. \text{rbt-delete } k \ t$) t2 t1

else if small-rbt t1 then rbtreeify (filter-minus t1 t2)

else case t2 of RBT-Impl.Empty \Rightarrow t1

| RBT-Impl.Branch - l2 a b r2 \Rightarrow

case rbt-split t1 a of (l1, -, r1) \Rightarrow rbt-join2 (rbt-minus-rec l1 l2) (rbt-minus-rec r1 r2))

declare *rbt-minus-rec.simps[simp del]*

end

context *linorder* **begin**

lemma *rbt-sorted-entries-right-unique:*

$\llbracket (k, v) \in \text{set } (\text{entries } t); (k, v') \in \text{set } (\text{entries } t);$

$\text{rbt-sorted } t \rrbracket \implies v = v'$

by(*auto dest!: distinct-entries inj-onD[where $x=(k, v)$ and $y=(k, v')$] simp add: distinct-map*)

lemma *rbt-sorted-fold-rbt-insertwk:*

$\text{rbt-sorted } t \implies \text{rbt-sorted } (\text{List.fold } (\lambda(k, v). \text{rbt-insert-with-key } f \ k \ v) \ xs \ t)$

by(*induct xs rule: rev-induct(auto simp add: rbt-insertwk-rbt-sorted)*)

lemma *is-rbt-fold-rbt-insertwk:*

assumes *is-rbt t1*

shows *is-rbt (fold (rbt-insert-with-key f) t2 t1)*

proof –

define *xs* **where** *xs = entries t2*

from *assms* **show** *?thesis unfolding fold-def xs-def[symmetric]*

by(*induct xs rule: rev-induct*)(*auto simp add: rbt-insertwk-is-rbt*)
qed

lemma *rbt-delete: inv-12 t \implies inv-12 (rbt-delete x t)*
using *rbt-del-inv1-inv2[of t x]*
by (*auto simp: inv-12-def rbt-delete-def rbt-del-inv1-inv2*)

lemma *rbt-sorted-delete: rbt-sorted t \implies rbt-sorted (rbt-delete x t)*
by (*auto simp: rbt-delete-def rbt-del-rbt-sorted*)

lemma *rbt-fold-rbt-delete:*
assumes *inv-12 t2*
shows *inv-12 (RBT-Impl.fold ($\lambda k - t.$ rbt-delete k t) t1 t2)*
proof –
define *xs* where *xs = RBT-Impl.entries t1*
from *assms* show *?thesis*
unfolding *RBT-Impl.fold-def xs-def[symmetric]*
by (*induct xs rule: rev-induct*) (*auto simp: rbt-delete*)
qed

lemma *rbt-minus-rec: inv-12 t1 \implies inv-12 t2 \implies inv-12 (rbt-minus-rec t1 t2)*
proof(*induction t1 t2 rule: rbt-minus-rec.induct*)
case (*1 t1 t2*)
thus *?case*
by (*auto simp: rbt-minus-rec.simps[of t1 t2] inv-rbt-join inv-rbt-join2 rbt-split*
rbt-fold-rbt-delete split!: RBT-Impl.rbt.splits RBT-Impl.color.splits prod.split
if-splits
dest: rbt-Node)
qed
end

context *linorder* **begin**

lemma *rbt-sorted-rbt-baliL: rbt-sorted l \implies rbt-sorted r \implies l \ll a \implies a \ll r \implies
rbt-sorted (rbt-baliL l a b r)
using *rbt-greater-trans rbt-less-trans*
by (*cases (l,a,b,r) rule: rbt-baliL.cases*) *fastforce+**

lemma *rbt-lookup-rbt-baliL: rbt-sorted l \implies rbt-sorted r \implies l \ll a \implies a \ll r \implies
rbt-lookup (rbt-baliL l a b r) k =
(if k < a then rbt-lookup l k else if k = a then Some b else rbt-lookup r k)
by (*cases (l,a,b,r) rule: rbt-baliL.cases*) (*auto split!: if-splits*)*

lemma *rbt-sorted-rbt-baliR: rbt-sorted l \implies rbt-sorted r \implies l \ll a \implies a \ll r \implies
rbt-sorted (rbt-baliR l a b r)
using *rbt-greater-trans rbt-less-trans*
by (*cases (l,a,b,r) rule: rbt-baliR.cases*) *fastforce+**

lemma *rbt-lookup-rbt-baliR*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 $\text{rbt-lookup } (\text{rbt-baliR } l \ a \ b \ r) \ k =$
(if $k < a$ *then* $\text{rbt-lookup } l \ k$ *else if* $k = a$ *then* $\text{Some } b$ *else* $\text{rbt-lookup } r \ k$)
by (*cases* (l, a, b, r) *rule*: *rbt-baliR.cases*) (*auto split!*: *if-splits*)

lemma *rbt-sorted-rbt-joinL*: $\text{rbt-sorted } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \implies \text{bheight } l$
 $\leq \text{bheight } r \implies$
 $\text{rbt-sorted } (\text{rbt-joinL } l \ a \ b \ r)$

proof (*induction* $l \ a \ b \ r$ *arbitrary*: c *rule*: *rbt-joinL.induct*)
case $(1 \ l \ a \ b \ r)$
thus *?case*
by (*auto simp*: *rbt-set-rbt-baliL* *rbt-joinL.simps*[*of* $l \ a \ b \ r$] *set-rbt-joinL* *rbt-less-prop*
intro!: *rbt-sorted-rbt-baliL* *split!*: *RBT-Impl.rbt.splits* *RBT-Impl.color.splits*)
qed

lemma *rbt-lookup-rbt-joinL*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 $\text{rbt-lookup } (\text{rbt-joinL } l \ a \ b \ r) \ k =$
(if $k < a$ *then* $\text{rbt-lookup } l \ k$ *else if* $k = a$ *then* $\text{Some } b$ *else* $\text{rbt-lookup } r \ k$)

proof (*induction* $l \ a \ b \ r$ *rule*: *rbt-joinL.induct*)
case $(1 \ l \ a \ b \ r)$
have *less-rbt-joinL*:
 $\text{rbt-sorted } r1 \implies r1 \mid \ll x \implies a \ll \mid r1 \implies a < x \implies \text{rbt-joinL } l \ a \ b \ r1 \mid \ll x$ **for**
 $x \ r1$
using $1(5)$
by (*auto simp*: *rbt-less-prop* *rbt-greater-prop* *set-rbt-joinL*)
show *?case*
using 1 *less-rbt-joinL* *rbt-lookup-rbt-baliL*[*OF* *rbt-sorted-rbt-joinL*[*of* - $l \ a \ b$],
where *?k=k*]
by (*auto simp*: *rbt-joinL.simps*[*of* $l \ a \ b \ r$] *split!*: *if-splits* *rbt.splits* *color.splits*)
qed

lemma *rbt-sorted-rbt-joinR*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 $\text{rbt-sorted } (\text{rbt-joinR } l \ a \ b \ r)$

proof (*induction* $l \ a \ b \ r$ *rule*: *rbt-joinR.induct*)
case $(1 \ l \ a \ b \ r)$
thus *?case*
by (*auto simp*: *rbt-set-rbt-baliR* *rbt-joinR.simps*[*of* $l \ a \ b \ r$] *set-rbt-joinR* *rbt-greater-prop*
intro!: *rbt-sorted-rbt-baliR* *split!*: *RBT-Impl.rbt.splits* *RBT-Impl.color.splits*)
qed

lemma *rbt-lookup-rbt-joinR*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \mid \ll a \implies a \ll \mid r \implies$
 \implies
 $\text{rbt-lookup } (\text{rbt-joinR } l \ a \ b \ r) \ k =$

(if $k < a$ *then* $\text{rbt-lookup } l \ k$ *else if* $k = a$ *then* $\text{Some } b$ *else* $\text{rbt-lookup } r \ k$)
proof (*induction* $l \ a \ b \ r$ *rule*: *rbt-joinR.induct*)

case $(1 \ l \ a \ b \ r)$
have *less-rbt-joinR*:
 $\text{rbt-sorted } l1 \implies x \ll \mid l1 \implies l1 \mid \ll a \implies x < a \implies x \ll \mid \text{rbt-joinR } l1 \ a \ b \ r$ **for**
 $x \ l1$


```

    using 1(6)
    by (auto simp: rbt-less-prop rbt-greater-prop set-rbt-joinR)
  show ?case
    using 1 less-rbt-joinR rbt-lookup-rbt-baliR[OF - rbt-sorted-rbt-joinR[of - r a b],
  where ?k=k]
    by (auto simp: rbt-joinR.simps[of l a b r] split!: if-splits rbt.splits color.splits)
qed

```

lemma *rbt-sorted-paint*: $\text{rbt-sorted } (\text{paint } c \ t) = \text{rbt-sorted } t$
 by (cases t) auto

lemma *rbt-sorted-rbt-join*: $\text{rbt-sorted } (\text{RBT-Impl.Branch } c \ l \ a \ b \ r) \implies$
 $\text{rbt-sorted } (\text{rbt-join } l \ a \ b \ r)$
 by (auto simp: rbt-sorted-paint rbt-sorted-rbt-joinL rbt-sorted-rbt-joinR rbt-join-def
 Let-def)

lemma *rbt-lookup-rbt-join*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies l \ll a \implies a \ll r \implies$
 $\text{rbt-lookup } (\text{rbt-join } l \ a \ b \ r) \ k =$
 $(\text{if } k < a \text{ then } \text{rbt-lookup } l \ k \text{ else if } k = a \text{ then } \text{Some } b \text{ else } \text{rbt-lookup } r \ k)$
 by (auto simp: rbt-join-def Let-def rbt-lookup-rbt-joinL rbt-lookup-rbt-joinR)

lemma *rbt-split-min-rbt-sorted*: $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq$
 $\text{RBT-Impl.Empty} \implies$
 $\text{rbt-sorted } t' \wedge (\forall x \in \text{set } (\text{RBT-Impl.keys } t'). \ a < x)$
 by (induction t arbitrary: t')
 (fastforce simp: rbt-split-min-set rbt-sorted-rbt-join set-rbt-join rbt-less-prop
 rbt-greater-prop
 split: if-splits prod.splits)+

lemma *rbt-split-min-rbt-lookup*: $\text{rbt-split-min } t = (a, b, t') \implies \text{rbt-sorted } t \implies t \neq$
 $\text{RBT-Impl.Empty} \implies$
 $\text{rbt-lookup } t \ k = (\text{if } k < a \text{ then } \text{None} \text{ else if } k = a \text{ then } \text{Some } b \text{ else } \text{rbt-lookup } t' \ k)$
 apply (induction t arbitrary: a b t')
 apply (simp-all split: if-splits prod.splits)
 apply (auto simp: rbt-less-prop rbt-split-min-set rbt-lookup-rbt-join rbt-split-min-rbt-sorted)
 done

lemma *rbt-sorted-rbt-join2*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies$
 $\forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies \text{rbt-sorted}$
 $(\text{rbt-join2 } l \ r)$
 by (simp add: rbt-join2-def rbt-sorted-rbt-join rbt-split-min-set rbt-split-min-rbt-sorted
 set-rbt-join
 rbt-greater-prop rbt-less-prop split: prod.split)

lemma *rbt-lookup-rbt-join2*: $\text{rbt-sorted } l \implies \text{rbt-sorted } r \implies$
 $\forall x \in \text{set } (\text{RBT-Impl.keys } l). \ \forall y \in \text{set } (\text{RBT-Impl.keys } r). \ x < y \implies$
 $\text{rbt-lookup } (\text{rbt-join2 } l \ r) \ k = (\text{case } \text{rbt-lookup } l \ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } r \ k \mid \text{Some } v \Rightarrow \text{Some } v)$

using *rbt-lookup-keys*
by (*fastforce simp: rbt-join2-def rbt-greater-prop rbt-less-prop rbt-lookup-rbt-join*
rbt-split-min-rbt-lookup rbt-split-min-rbt-sorted rbt-split-min-set split: op-
tion.splits prod.splits)

lemma *rbt-split-props*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies$
 $\text{set } (\text{RBT-Impl.keys } l) = \{a \in \text{set } (\text{RBT-Impl.keys } t). a < x\} \wedge$
 $\text{set } (\text{RBT-Impl.keys } r) = \{a \in \text{set } (\text{RBT-Impl.keys } t). x < a\} \wedge$
 $\text{rbt-sorted } l \wedge \text{rbt-sorted } r$
apply (*induction t arbitrary: l r*)
apply (*simp-all split!: prod.splits if-splits*)
apply (*force simp: set-rbt-join rbt-greater-prop rbt-less-prop*
intro: rbt-sorted-rbt-join)
done

lemma *rbt-split-lookup*: $\text{rbt-split } t \ x = (l, \beta, r) \implies \text{rbt-sorted } t \implies$
 $\text{rbt-lookup } t \ k = (\text{if } k < x \text{ then } \text{rbt-lookup } l \ k \text{ else if } k = x \text{ then } \beta \text{ else } \text{rbt-lookup}$
 $r \ k)$

proof (*induction t arbitrary: x l β r*)
case (*Branch c t1 a b t2*)
have $\text{rbt-sorted } r1 \ r1 \mid \ll a \text{ if } \text{rbt-split } t1 \ x = (l, \beta, r1) \text{ for } r1$
using *rbt-split-props Branch(4) that*
by (*fastforce simp: rbt-less-prop*)
moreover have $\text{rbt-sorted } l1 \ a \mid \ll l1 \text{ if } \text{rbt-split } t2 \ x = (l1, \beta, r) \text{ for } l1$
using *rbt-split-props Branch(4) that*
by (*fastforce simp: rbt-greater-prop*)
ultimately show *?case*
using *Branch rbt-lookup-rbt-join[of t1 - a b k] rbt-lookup-rbt-join[of - t2 a b k]*
by (*auto split!: if-splits prod.splits*)
qed *simp*

lemma *rbt-sorted-fold-insertwk*: $\text{rbt-sorted } t \implies$
 $\text{rbt-sorted } (\text{RBT-Impl.fold } (\text{rbt-insert-with-key } f) \ t' \ t)$
by (*induct t' arbitrary: t*)
(simp-all add: rbt-insertwk-rbt-sorted)

lemma *rbt-lookup-iff-keys*:
 $\text{rbt-sorted } t \implies \text{set } (\text{RBT-Impl.keys } t) = \{k. \exists v. \text{rbt-lookup } t \ k = \text{Some } v\}$
 $\text{rbt-sorted } t \implies \text{rbt-lookup } t \ k = \text{None} \iff k \notin \text{set } (\text{RBT-Impl.keys } t)$
 $\text{rbt-sorted } t \implies (\exists v. \text{rbt-lookup } t \ k = \text{Some } v) \iff k \in \text{set } (\text{RBT-Impl.keys } t)$
using *entry-in-tree-keys rbt-lookup-keys[of t]*
by *force*

lemma *rbt-lookup-fold-rbt-insertwk*:
assumes *t1: rbt-sorted t1 and t2: rbt-sorted t2*
shows $\text{rbt-lookup } (\text{fold } (\text{rbt-insert-with-key } f) \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } t2 \ k$
 $\mid \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{Some } v$
 $\mid \text{Some } w \Rightarrow \text{Some } (f \ k \ w \ v))$

```

proof –
  define xs where xs = entries t1
  hence dt1: distinct (map fst xs) using t1 by (simp add: distinct-entries)
  with t2 show ?thesis
    unfolding fold-def map-of-entries[OF t1, symmetric]
      xs-def[symmetric] distinct-map-of-rev[OF dt1, symmetric]
    apply(induct xs rule: rev-induct)
    apply(auto simp add: rbt-lookup-rbt-insertwk rbt-sorted-fold-rbt-insertwk split:
option.splits)
    apply(auto simp add: distinct-map-of-rev intro: rev-image-eqI)
    done
qed

lemma rbt-lookup-union-rec: rbt-sorted t1  $\implies$  rbt-sorted t2  $\implies$ 
rbt-sorted (rbt-union-rec f t1 t2)  $\wedge$  rbt-lookup (rbt-union-rec f t1 t2) k =
(case rbt-lookup t1 k of None  $\implies$  rbt-lookup t2 k
  | Some v  $\implies$  (case rbt-lookup t2 k of None  $\implies$  Some v
    | Some w  $\implies$  Some (f k v w)))

proof(induction f t1 t2 arbitrary: k rule: rbt-union-rec.induct)
  case (1 f t1 t2)
    obtain f' t1' t2' where flip: (f', t2', t1') =
      (if flip-rbt t2 t1 then ( $\lambda k v v'. f k v' v, t1, t2$ ) else (f, t2, t1))
    by fastforce
    have rbt-sorted': rbt-sorted t1' rbt-sorted t2'
      using 1(3,4) flip
      by (auto split: if-splits)
    show ?case
    proof (cases t1')
      case Empty
        show ?thesis
          unfolding rbt-union-rec.simps[of - t1] flip[symmetric]
          using flip rbt-sorted' rbt-split-props[of t2]
          by (auto simp: Empty rbt-lookup-fold-rbt-insertwk
            intro!: rbt-sorted-fold-insertwk split: if-splits option.splits)
      next
        case (Branch c l1 a b r1)
          {
            assume not-small:  $\neg$ small-rbt t2'
            obtain l2  $\beta$  r2 where rbt-split-t2': rbt-split t2' a = (l2,  $\beta$ , r2)
              by (cases rbt-split t2' a) auto
            have rbt-sort: rbt-sorted l1 rbt-sorted r1
              using 1(3,4) flip
              by (auto simp: Branch split: if-splits)
            note rbt-split-t2'-props = rbt-split-props[OF rbt-split-t2' rbt-sorted'(2)]
            have union-l1-l2: rbt-sorted (rbt-union-rec f' l1 l2) rbt-lookup (rbt-union-rec
              f' l1 l2) k =
              (case rbt-lookup l1 k of None  $\implies$  rbt-lookup l2 k
                | Some v  $\implies$  (case rbt-lookup l2 k of None  $\implies$  Some v | Some w  $\implies$  Some (f'
                  k v w))) for k
          }

```

```

using 1(1)[OF flip refl refl - Branch rbt-split-t2'[symmetric]] rbt-sort
rbt-split-t2'-props
by (auto simp: not-small)
have union-r1-r2: rbt-sorted (rbt-union-rec f' r1 r2) rbt-lookup (rbt-union-rec
f' r1 r2) k =
  (case rbt-lookup r1 k of None  $\Rightarrow$  rbt-lookup r2 k
  | Some v  $\Rightarrow$  (case rbt-lookup r2 k of None  $\Rightarrow$  Some v | Some w  $\Rightarrow$  Some (f'
k v w))) for k
using 1(2)[OF flip refl refl - Branch rbt-split-t2'[symmetric]] rbt-sort
rbt-split-t2'-props
by (auto simp: not-small)
have union-l1-l2-keys: set (RBT-Impl.keys (rbt-union-rec f' l1 l2)) =
set (RBT-Impl.keys l1)  $\cup$  set (RBT-Impl.keys l2)
using rbt-sorted'(1) rbt-split-t2'-props
by (auto simp: Branch rbt-lookup-iff-keys(1) union-l1-l2 split: option.splits)
have union-r1-r2-keys: set (RBT-Impl.keys (rbt-union-rec f' r1 r2)) =
set (RBT-Impl.keys r1)  $\cup$  set (RBT-Impl.keys r2)
using rbt-sorted'(1) rbt-split-t2'-props
by (auto simp: Branch rbt-lookup-iff-keys(1) union-r1-r2 split: option.splits)
have union-l1-l2-less: rbt-union-rec f' l1 l2  $\ll$  a
using rbt-sorted'(1) rbt-split-t2'-props
by (auto simp: Branch rbt-less-prop union-l1-l2-keys)
have union-r1-r2-greater: a  $\ll$  rbt-union-rec f' r1 r2
using rbt-sorted'(1) rbt-split-t2'-props
by (auto simp: Branch rbt-greater-prop union-r1-r2-keys)
have rbt-lookup (rbt-union-rec f t1 t2) k =
(case rbt-lookup t1' k of None  $\Rightarrow$  rbt-lookup t2' k
| Some v  $\Rightarrow$  (case rbt-lookup t2' k of None  $\Rightarrow$  Some v | Some w  $\Rightarrow$  Some (f'
k v w)))
using rbt-sorted' union-l1-l2 union-r1-r2 rbt-split-t2'-props
union-l1-l2-less union-r1-r2-greater not-small
by (auto simp: rbt-union-rec.simps[of - t1] flip[symmetric] Branch
rbt-split-t2' rbt-lookup-rbt-join rbt-split-lookup[OF rbt-split-t2'] split:
option.splits)
moreover have rbt-sorted (rbt-union-rec f t1 t2)
using rbt-sorted' rbt-split-t2'-props not-small
by (auto simp: rbt-union-rec.simps[of - t1] flip[symmetric] Branch rbt-split-t2'
union-l1-l2 union-r1-r2 union-l1-l2-keys union-r1-r2-keys rbt-less-prop
rbt-greater-prop intro!: rbt-sorted-rbt-join)
ultimately have ?thesis
using flip
by (auto split: if-splits option.splits)
}
then show ?thesis
unfolding rbt-union-rec.simps[of - t1] flip[symmetric]
using rbt-sorted' flip
by (auto simp: rbt-sorted-fold-insertwk rbt-lookup-fold-rbt-insertwk split: op-
tion.splits)
qed

```

qed

lemma *rbtreeify-map-filter-inter*:

fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$

assumes *rbt-sorted t2*

shows *rbt-sorted (rbtreeify (map-filter-inter f t1 t2))*

rbt-lookup (rbtreeify (map-filter-inter f t1 t2)) k =

(case rbt-lookup t1 k of None \Rightarrow None

| Some v \Rightarrow (case rbt-lookup t2 k of None \Rightarrow None | Some w \Rightarrow Some (f k v w)))

proof –

have *map-of-map-filter*: *map-of (List.map-filter ($\lambda(k, v).$*

case rbt-lookup t1 k of None \Rightarrow None | Some v' \Rightarrow Some (k, f k v' v)) xs) k =

(case rbt-lookup t1 k of None \Rightarrow None

| Some v \Rightarrow (case map-of xs k of None \Rightarrow None | Some w \Rightarrow Some (f k v w)))

for *xs k*

by (*induction xs*) (*auto simp: List.map-filter-def split: option.splits*)

have *map-fst-map-filter*: *map fst (List.map-filter ($\lambda(k, v).$*

case rbt-lookup t1 k of None \Rightarrow None | Some v' \Rightarrow Some (k, f k v' v)) xs) =

filter ($\lambda k. \text{rbt-lookup } t1 \ k \neq \text{None}$) (map fst xs) for xs

by (*induction xs*) (*auto simp: List.map-filter-def split: option.splits*)

have *sorted (map fst (map-filter-inter f t1 t2))*

using *sorted-filter[of id] rbt-sorted-entries[OF assms]*

by (*auto simp: map-filter-inter-def map-fst-map-filter*)

moreover have *distinct (map fst (map-filter-inter f t1 t2))*

using *distinct-filter distinct-entries[OF assms]*

by (*auto simp: map-filter-inter-def map-fst-map-filter*)

ultimately show

rbt-sorted (rbtreeify (map-filter-inter f t1 t2))

rbt-lookup (rbtreeify (map-filter-inter f t1 t2)) k =

(case rbt-lookup t1 k of None \Rightarrow None

| Some v \Rightarrow (case rbt-lookup t2 k of None \Rightarrow None | Some w \Rightarrow Some (f k v w)))

using *rbt-sorted-rbtreeify*

by (*auto simp: rbt-lookup-rbtreeify map-filter-inter-def map-of-map-filter*

map-of-entries[OF assms] split: option.splits)

qed

lemma *rbt-lookup-inter-rec*: *rbt-sorted t1 \Rightarrow rbt-sorted t2 \Rightarrow*

rbt-sorted (rbt-inter-rec f t1 t2) \wedge rbt-lookup (rbt-inter-rec f t1 t2) k =

(case rbt-lookup t1 k of None \Rightarrow None

| Some v \Rightarrow (case rbt-lookup t2 k of None \Rightarrow None | Some w \Rightarrow Some (f k v w)))

proof(*induction f t1 t2 arbitrary: k rule: rbt-inter-rec.induct*)

case (*1 f t1 t2*)

obtain $f' \ t1' \ t2'$ **where** *flip*: ($f', t2', t1' =$

(if flip-rbt t2 t1 then ($\lambda k \ v \ v'. f \ k \ v' \ v, t1, t2$) else ($f, t2, t1$))

by *fastforce*

have *rbt-sorted'*: *rbt-sorted t1' rbt-sorted t2'*

using *1(3,4) flip*

```

  by (auto split: if-splits)
show ?case
proof (cases t1')
  case Empty
  show ?thesis
    unfolding rbt-inter-rec.simps[of - t1] flip[symmetric]
    using flip rbt-sorted' rbt-split-props[of t2] rbtreeify-map-filter-inter[OF rbt-sorted'(2)]
    by (auto simp: Empty split: option.splits)
next
case (Branch c l1 a b r1)
{
  assume not-small:  $\neg$ small-rbt t2'
  obtain l2  $\beta$  r2 where rbt-split-t2': rbt-split t2' a = (l2,  $\beta$ , r2)
  by (cases rbt-split t2' a) auto
  note rbt-split-t2'-props = rbt-split-props[OF rbt-split-t2' rbt-sorted'(2)]
  have rbt-sort: rbt-sorted l1 rbt-sorted r1 rbt-sorted l2 rbt-sorted r2
  using 1(3,4) flip
  by (auto simp: Branch rbt-split-t2'-props split: if-splits)
  have inter-l1-l2: rbt-sorted (rbt-inter-rec f' l1 l2) rbt-lookup (rbt-inter-rec f'
l1 l2) k =
    (case rbt-lookup l1 k of None  $\Rightarrow$  None
    | Some v  $\Rightarrow$  (case rbt-lookup l2 k of None  $\Rightarrow$  None | Some w  $\Rightarrow$  Some (f' k
v w))) for k
    using 1(1)[OF flip refl refl - Branch rbt-split-t2'[symmetric]] rbt-sort
rbt-split-t2'-props
  by (auto simp: not-small)
  have inter-r1-r2: rbt-sorted (rbt-inter-rec f' r1 r2) rbt-lookup (rbt-inter-rec f'
r1 r2) k =
    (case rbt-lookup r1 k of None  $\Rightarrow$  None
    | Some v  $\Rightarrow$  (case rbt-lookup r2 k of None  $\Rightarrow$  None | Some w  $\Rightarrow$  Some (f' k
v w))) for k
    using 1(2)[OF flip refl refl - Branch rbt-split-t2'[symmetric]] rbt-sort
rbt-split-t2'-props
  by (auto simp: not-small)
  have inter-l1-l2-keys: set (RBT-Impl.keys (rbt-inter-rec f' l1 l2)) =
    set (RBT-Impl.keys l1)  $\cap$  set (RBT-Impl.keys l2)
  using inter-l1-l2(1)
  by (auto simp: rbt-lookup-iff-keys(1) inter-l1-l2(2) rbt-sort split: option.splits)
  have inter-r1-r2-keys: set (RBT-Impl.keys (rbt-inter-rec f' r1 r2)) =
    set (RBT-Impl.keys r1)  $\cap$  set (RBT-Impl.keys r2)
  using inter-r1-r2(1)
  by (auto simp: rbt-lookup-iff-keys(1) inter-r1-r2(2) rbt-sort split: op-
tion.splits)
  have inter-l1-l2-less: rbt-inter-rec f' l1 l2  $\ll$  a
  using rbt-sorted'(1) rbt-split-t2'-props
  by (auto simp: Branch rbt-less-prop inter-l1-l2-keys)
  have inter-r1-r2-greater: a  $\ll$  rbt-inter-rec f' r1 r2
  using rbt-sorted'(1) rbt-split-t2'-props
  by (auto simp: Branch rbt-greater-prop inter-r1-r2-keys)
}

```

```

have rbt-lookup-join2: rbt-lookup (rbt-join2 (rbt-inter-rec f' l1 l2) (rbt-inter-rec
f' r1 r2)) k =
  (case rbt-lookup (rbt-inter-rec f' l1 l2) k of None  $\Rightarrow$  rbt-lookup (rbt-inter-rec
f' r1 r2) k
  | Some v  $\Rightarrow$  Some v) for k
  using rbt-lookup-rbt-join2[OF inter-l1-l2(1) inter-r1-r2(1)] rbt-sorted'(1)
  by (fastforce simp: Branch inter-l1-l2-keys inter-r1-r2-keys rbt-less-prop
rbt-greater-prop)
  have rbt-lookup-l1-k: rbt-lookup l1 k = Some v  $\Rightarrow$  k < a for k v
  using rbt-sorted'(1) rbt-lookup-iff-keys(3)
  by (auto simp: Branch rbt-less-prop)
  have rbt-lookup-r1-k: rbt-lookup r1 k = Some v  $\Rightarrow$  a < k for k v
  using rbt-sorted'(1) rbt-lookup-iff-keys(3)
  by (auto simp: Branch rbt-greater-prop)
  have rbt-lookup (rbt-inter-rec f t1 t2) k =
    (case rbt-lookup t1' k of None  $\Rightarrow$  None
    | Some v  $\Rightarrow$  (case rbt-lookup t2' k of None  $\Rightarrow$  None | Some w  $\Rightarrow$  Some (f' k
v w)))
  by (auto simp: Let-def rbt-inter-rec.simps[of - t1] flip[symmetric] not-small
Branch
      rbt-split-t2' rbt-lookup-join2 rbt-lookup-rbt-join inter-l1-l2-less in-
ter-r1-r2-greater
      rbt-split-lookup[OF rbt-split-t2' rbt-sorted'(2)] inter-l1-l2 inter-r1-r2
      split!: if-splits option.splits dest: rbt-lookup-l1-k rbt-lookup-r1-k)
  moreover have rbt-sorted (rbt-inter-rec f t1 t2)
  using rbt-sorted' inter-l1-l2 inter-r1-r2 rbt-split-t2'-props not-small
  by (auto simp: Let-def rbt-inter-rec.simps[of - t1] flip[symmetric] Branch
rbt-split-t2'
      rbt-less-prop rbt-greater-prop inter-l1-l2-less inter-r1-r2-greater
      inter-l1-l2-keys inter-r1-r2-keys intro!: rbt-sorted-rbt-join rbt-sorted-rbt-join2
      split: if-splits option.splits dest!: bspec)
  ultimately have ?thesis
  using flip
  by (auto split: if-splits split: option.splits)
}
then show ?thesis
unfolding rbt-inter-rec.simps[of - t1] flip[symmetric]
using rbt-sorted' flip rbtreeify-map-filter-inter[OF rbt-sorted'(2)]
by (auto split: option.splits)
qed
qed

lemma rbt-lookup-delete:
  assumes inv-12 t rbt-sorted t
  shows rbt-lookup (rbt-delete x t) k = (if x = k then None else rbt-lookup t k)
proof –
  note rbt-sorted-del = rbt-del-rbt-sorted[OF assms(2), of x]
  show ?thesis
  using assms rbt-sorted-del rbt-del-in-tree rbt-lookup-from-in-tree[OF assms(2)]

```

rbt-sorted-del]

by (*fastforce simp: inv-12-def rbt-delete-def rbt-lookup-iff-keys*(2) *keys-entries*)
qed

lemma *fold-rbt-delete*:

assumes *inv-12 t1 rbt-sorted t1 rbt-sorted t2*

shows *inv-12 (RBT-Impl.fold ($\lambda k - t.$ rbt-delete *k t*) t2 t1) \wedge*
*rbt-sorted (RBT-Impl.fold ($\lambda k - t.$ rbt-delete *k t*) t2 t1) \wedge*
*rbt-lookup (RBT-Impl.fold ($\lambda k - t.$ rbt-delete *k t*) t2 t1) *k* =*
*(case rbt-lookup t1 *k* of None \Rightarrow None*
*| Some *v* \Rightarrow (case rbt-lookup t2 *k* of None \Rightarrow Some *v* | - \Rightarrow None))*

proof –

define *xs* **where** *xs* = *RBT-Impl.entries t2*

show *inv-12 (RBT-Impl.fold ($\lambda k - t.$ rbt-delete *k t*) t2 t1) \wedge*
*rbt-sorted (RBT-Impl.fold ($\lambda k - t.$ rbt-delete *k t*) t2 t1) \wedge*
*rbt-lookup (RBT-Impl.fold ($\lambda k - t.$ rbt-delete *k t*) t2 t1) *k* =*
*(case rbt-lookup t1 *k* of None \Rightarrow None*
*| Some *v* \Rightarrow (case rbt-lookup t2 *k* of None \Rightarrow Some *v* | - \Rightarrow None))*

using *assms*(1,2)

unfolding *map-of-entries*[*OF assms*(3), *symmetric*] *RBT-Impl.fold-def xs-def*[*symmetric*]

by (*induction xs arbitrary: t1 rule: rev-induct*)

(*auto simp: rbt-delete rbt-sorted-delete rbt-lookup-delete split!: option.splits*)

qed

lemma *rbtreeify-filter-minus*:

assumes *rbt-sorted t1*

shows *rbt-sorted (rbtreeify (filter-minus t1 t2)) \wedge*
*rbt-lookup (rbtreeify (filter-minus t1 t2)) *k* =*
*(case rbt-lookup t1 *k* of None \Rightarrow None*
*| Some *v* \Rightarrow (case rbt-lookup t2 *k* of None \Rightarrow Some *v* | - \Rightarrow None))*

proof –

have *map-of-filter: map-of (filter ($\lambda(k, -).$ rbt-lookup t2 *k* = None) xs) *k* =*
*(case map-of xs *k* of None \Rightarrow None*
*| Some *v* \Rightarrow (case rbt-lookup t2 *k* of None \Rightarrow Some *v* | Some *x* \Rightarrow Map.empty*
x))

for *xs* :: (*'a* \times *'b*) list

by (*induction xs*) (*auto split: option.splits*)

have *map-fst-filter-minus: map fst (filter-minus t1 t2) =*

*filter ($\lambda k.$ rbt-lookup t2 *k* = None) (map fst (RBT-Impl.entries t1))*

by (*auto simp: filter-minus-def filter-map comp-def case-prod-unfold*)

have *sorted (map fst (filter-minus t1 t2)) distinct (map fst (filter-minus t1 t2))*

using *distinct-filter distinct-entries*[*OF assms*]

sorted-filter[*of id*] *rbt-sorted-entries*[*OF assms*]

by (*auto simp: map-fst-filter-minus intro!: rbt-sorted-rbtreeify*)

then show *?thesis*

by (*auto simp: rbt-lookup-rbtreeify filter-minus-def map-of-filter map-of-entries*[*OF assms*]

intro!: rbt-sorted-rbtreeify)

qed


```

lemma rbt-lookup-minus-rec: inv-12 t1  $\impl$  rbt-sorted t1  $\impl$  rbt-sorted t2  $\impl$ 
  rbt-sorted (rbt-minus-rec t1 t2)  $\wedge$  rbt-lookup (rbt-minus-rec t1 t2) k =
  (case rbt-lookup t1 k of None  $\impl$  None
  | Some v  $\impl$  (case rbt-lookup t2 k of None  $\impl$  Some v | -  $\impl$  None))
proof(induction t1 t2 arbitrary: k rule: rbt-minus-rec.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t2)
    case Empty
    show ?thesis
    using rbtreeify-filter-minus[OF 1(4)] 1(4)
    by (auto simp: rbt-minus-rec.simps[of t1] Empty split: option.splits)
  next
  case (Branch c l2 a b r2)
  {
    assume not-small:  $\neg$ small-rbt t2  $\neg$ small-rbt t1
    obtain l1  $\beta$  r1 where rbt-split-t1: rbt-split t1 a = (l1,  $\beta$ , r1)
    by (cases rbt-split t1 a) auto
    note rbt-split-t1-props = rbt-split-props[OF rbt-split-t1 1(4)]
    have minus-l1-l2: rbt-sorted (rbt-minus-rec l1 l2)
      rbt-lookup (rbt-minus-rec l1 l2) k =
      (case rbt-lookup l1 k of None  $\impl$  None
      | Some v  $\impl$  (case rbt-lookup l2 k of None  $\impl$  Some v | Some x  $\impl$  None))
    for k
      using 1(1)[OF not-small Branch rbt-split-t1[symmetric] refl] 1(5) rbt-split-t1-props
        rbt-split[OF rbt-split-t1 1(3)]
      by (auto simp: Branch)
      have minus-r1-r2: rbt-sorted (rbt-minus-rec r1 r2)
        rbt-lookup (rbt-minus-rec r1 r2) k =
        (case rbt-lookup r1 k of None  $\impl$  None
        | Some v  $\impl$  (case rbt-lookup r2 k of None  $\impl$  Some v | Some x  $\impl$  None))
      for k
        using 1(2)[OF not-small Branch rbt-split-t1[symmetric] refl] 1(5) rbt-split-t1-props
          rbt-split[OF rbt-split-t1 1(3)]
        by (auto simp: Branch)
        have minus-l1-l2-keys: set (RBT-Impl.keys (rbt-minus-rec l1 l2)) =
          set (RBT-Impl.keys l1) - set (RBT-Impl.keys l2)
        using minus-l1-l2(1) 1(5) rbt-lookup-iff-keys(3) rbt-split-t1-props
          by (auto simp: Branch rbt-lookup-iff-keys(1) minus-l1-l2(2) split: option.splits)
        have minus-r1-r2-keys: set (RBT-Impl.keys (rbt-minus-rec r1 r2)) =
          set (RBT-Impl.keys r1) - set (RBT-Impl.keys r2)
        using minus-r1-r2(1) 1(5) rbt-lookup-iff-keys(3) rbt-split-t1-props
          by (auto simp: Branch rbt-lookup-iff-keys(1) minus-r1-r2(2) split: option.splits)
        have rbt-lookup-join2: rbt-lookup (rbt-join2 (rbt-minus-rec l1 l2) (rbt-minus-rec
          r1 r2)) k =
          (case rbt-lookup (rbt-minus-rec l1 l2) k of None  $\impl$  rbt-lookup (rbt-minus-rec

```

```

r1 r2) k
  | Some v  $\Rightarrow$  Some v) for k
using rbt-lookup-rbt-join2[OF minus-l1-l2(1) minus-r1-r2(1)] rbt-split-t1-props
  by (fastforce simp: minus-l1-l2-keys minus-r1-r2-keys)
have lookup-l1-r1-a: rbt-lookup l1 a = None rbt-lookup r1 a = None
  using rbt-split-t1-props
  by (auto simp: rbt-lookup-iff-keys(2))
have rbt-lookup (rbt-minus-rec t1 t2) k =
  (case rbt-lookup t1 k of None  $\Rightarrow$  None
  | Some v  $\Rightarrow$  (case rbt-lookup t2 k of None  $\Rightarrow$  Some v | -  $\Rightarrow$  None))
  using not-small rbt-lookup-iff-keys(2)[of l1] rbt-lookup-iff-keys(3)[of l1]
  rbt-lookup-iff-keys(3)[of r1] rbt-split-t1-props
  using [[simp-depth-limit = 2]]
  by (auto simp: rbt-minus-rec.simps[of t1] Branch rbt-split-t1 rbt-lookup-join2
      minus-l1-l2(2) minus-r1-r2(2) rbt-split-lookup[OF rbt-split-t1 1(4)]
      lookup-l1-r1-a
      split: option.splits)
  moreover have rbt-sorted (rbt-minus-rec t1 t2)
  using not-small minus-l1-l2(1) minus-r1-r2(1) rbt-split-t1-props rbt-sorted-rbt-join2
  by (fastforce simp: rbt-minus-rec.simps[of t1] Branch rbt-split-t1 minus-l1-l2-keys
      minus-r1-r2-keys)
  ultimately have ?thesis
  by (auto split: if-splits split: option.splits)
}
then show ?thesis
  using fold-rbt-delete[OF 1(3,4,5)] rbtreeify-filter-minus[OF 1(4)]
  by (auto simp: rbt-minus-rec.simps[of t1])
qed
qed
end

context ord begin

definition rbt-union-with-key :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) rbt  $\Rightarrow$  ('a, 'b) rbt
 $\Rightarrow$  ('a, 'b) rbt
where
  rbt-union-with-key f t1 t2 = paint B (rbt-union-swap-rec f False t1 t2)

definition rbt-union-with where
  rbt-union-with f = rbt-union-with-key ( $\lambda$ -. f)

definition rbt-union where
  rbt-union = rbt-union-with-key (%- - rv. rv)

definition rbt-inter-with-key :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) rbt  $\Rightarrow$  ('a, 'b) rbt
 $\Rightarrow$  ('a, 'b) rbt
where
  rbt-inter-with-key f t1 t2 = paint B (rbt-inter-swap-rec f False t1 t2)

```

definition *rbt-inter-with* **where**

rbt-inter-with $f = \text{rbt-inter-with-key } (\lambda\cdot. f)$

definition *rbt-inter* **where**

rbt-inter $= \text{rbt-inter-with-key } (\lambda\cdot - \text{rv. rv})$

definition *rbt-minus* **where**

rbt-minus $t1\ t2 = \text{paint } B\ (\text{rbt-minus-rec } t1\ t2)$

end

context *linorder* **begin**

lemma *is-rbt-rbt-unionwk* [*simp*]:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-union-with-key } f\ t1\ t2)$

using *rbt-union-rec rbt-lookup-union-rec*

by (*fastforce simp: rbt-union-with-key-def rbt-union-swap-rec is-rbt-def inv-12-def*)

lemma *rbt-lookup-rbt-unionwk*:

$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$

$\implies \text{rbt-lookup } (\text{rbt-union-with-key } f\ t1\ t2)\ k =$

$(\text{case } \text{rbt-lookup } t1\ k \text{ of } \text{None} \Rightarrow \text{rbt-lookup } t2\ k$

$\mid \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2\ k \text{ of } \text{None} \Rightarrow \text{Some } v$

$\mid \text{Some } w \Rightarrow \text{Some } (f\ k\ v\ w))$

using *rbt-lookup-union-rec*

by (*auto simp: rbt-union-with-key-def rbt-union-swap-rec*)

lemma *rbt-unionw-is-rbt*: $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \implies \text{is-rbt } (\text{rbt-union-with } f\ lt\ rt)$

by(*simp add: rbt-union-with-def*)

lemma *rbt-union-is-rbt*: $\llbracket \text{is-rbt } lt; \text{is-rbt } rt \rrbracket \implies \text{is-rbt } (\text{rbt-union } lt\ rt)$

by(*simp add: rbt-union-def*)

lemma *rbt-lookup-rbt-union*:

$\llbracket \text{rbt-sorted } s; \text{rbt-sorted } t \rrbracket \implies$

$\text{rbt-lookup } (\text{rbt-union } s\ t) = \text{rbt-lookup } s\ ++\ \text{rbt-lookup } t$

by(*rule ext*)(*simp add: rbt-lookup-rbt-unionwk rbt-union-def map-add-def split: option.split*)

lemma *rbt-interwk-is-rbt* [*simp*]:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter-with-key } f\ t1\ t2)$

using *rbt-inter-rec rbt-lookup-inter-rec*

by (*fastforce simp: rbt-inter-with-key-def rbt-inter-swap-rec is-rbt-def inv-12-def rbt-sorted-paint*)

lemma *rbt-interw-is-rbt*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter-with } f\ t1\ t2)$

by(*simp add: rbt-inter-with-def*)

lemma *rbt-inter-is-rbt*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-inter } t1 \ t2)$
by(*simp add: rbt-inter-def*)

lemma *rbt-lookup-rbt-interwk*:

$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$
 $\implies \text{rbt-lookup } (\text{rbt-inter-with-key } f \ t1 \ t2) \ k =$
 $(\text{case } \text{rbt-lookup } t1 \ k \text{ of } \text{None} \Rightarrow \text{None}$
 $\quad | \text{Some } v \Rightarrow \text{case } \text{rbt-lookup } t2 \ k \text{ of } \text{None} \Rightarrow \text{None}$
 $\quad \quad | \text{Some } w \Rightarrow \text{Some } (f \ k \ v \ w))$
using *rbt-lookup-inter-rec*
by (*auto simp: rbt-inter-with-key-def rbt-inter-swap-rec*)

lemma *rbt-lookup-rbt-inter*:

$\llbracket \text{rbt-sorted } t1; \text{rbt-sorted } t2 \rrbracket$
 $\implies \text{rbt-lookup } (\text{rbt-inter } t1 \ t2) = \text{rbt-lookup } t2 \mid' \text{dom } (\text{rbt-lookup } t1)$
by(*auto simp add: rbt-inter-def rbt-lookup-rbt-interwk restrict-map-def split: option.split*)

lemma *rbt-minus-is-rbt*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket \implies \text{is-rbt } (\text{rbt-minus } t1 \ t2)$
using *rbt-minus-rec[of t1 t2] rbt-lookup-minus-rec[of t1 t2]*
by (*auto simp: rbt-minus-def is-rbt-def inv-12-def*)

lemma *rbt-lookup-rbt-minus*:

$\llbracket \text{is-rbt } t1; \text{is-rbt } t2 \rrbracket$
 $\implies \text{rbt-lookup } (\text{rbt-minus } t1 \ t2) = \text{rbt-lookup } t1 \mid' (- \text{dom } (\text{rbt-lookup } t2))$
by (*rule ext*)
(auto simp: rbt-minus-def is-rbt-def inv-12-def restrict-map-def rbt-lookup-minus-rec split: option.splits)

end

129.11 Code generator setup

lemmas [*code*] =

ord.rbt-less-prop
ord.rbt-greater-prop
ord.rbt-sorted.simps
ord.rbt-lookup.simps
ord.is-rbt-def
ord.rbt-ins.simps
ord.rbt-insert-with-key-def
ord.rbt-insertw-def
ord.rbt-insert-def
ord.rbt-del-from-left.simps
ord.rbt-del-from-right.simps
ord.rbt-del.simps

```

ord.rbt-delete-def
ord.rbt-split.simps
ord.rbt-union-swap-rec.simps
ord.map-filter-inter-def
ord.rbt-inter-swap-rec.simps
ord.filter-minus-def
ord.rbt-minus-rec.simps
ord.rbt-union-with-key-def
ord.rbt-union-with-def
ord.rbt-union-def
ord.rbt-inter-with-key-def
ord.rbt-inter-with-def
ord.rbt-inter-def
ord.rbt-minus-def
ord.rbt-map-entry.simps
ord.rbt-bulkload-def

```

More efficient implementations for *entries* and *keys*

definition *gen-entries* ::

$((a \times b) \times (a, b) \text{ rbt}) \text{ list} \Rightarrow (a, b) \text{ rbt} \Rightarrow (a \times b) \text{ list}$

where

gen-entries kvs t = *entries t @ concat (map ($\lambda(kv, t). kv \# \text{entries } t$) kvs)*

lemma *gen-entries-simps* [*simp*, *code*]:

gen-entries [] Empty = *[]*

gen-entries ((kv, t) # kvs) Empty = *kv # gen-entries kvs t*

gen-entries kvs (Branch c l k v r) = *gen-entries (((k, v), r) # kvs) l*

by(*simp-all add: gen-entries-def*)

lemma *entries-code* [*code*]:

entries = *gen-entries []*

by(*simp add: gen-entries-def fun-eq-iff*)

definition *gen-keys* :: $(a \times (a, b) \text{ rbt}) \text{ list} \Rightarrow (a, b) \text{ rbt} \Rightarrow a \text{ list}$

where *gen-keys kts t* = *RBT-Impl.keys t @ concat (List.map ($\lambda(k, t). k \# \text{keys } t$) kts)*

lemma *gen-keys-simps* [*simp*, *code*]:

gen-keys [] Empty = *[]*

gen-keys ((k, t) # kts) Empty = *k # gen-keys kts t*

gen-keys kts (Branch c l k v r) = *gen-keys ((k, r) # kts) l*

by(*simp-all add: gen-keys-def*)

lemma *keys-code* [*code*]:

keys = *gen-keys []*

by(*simp add: gen-keys-def fun-eq-iff*)

Restore original type constraints for constants

setup <

```

fold Sign.add-const-constraint
  [(const-name <rbt-less>, SOME typ <('a :: order) ⇒ ('a, 'b) rbt ⇒ bool>),
   (const-name <rbt-greater>, SOME typ <('a :: order) ⇒ ('a, 'b) rbt ⇒ bool>),
   (const-name <rbt-sorted>, SOME typ <('a :: linorder, 'b) rbt ⇒ bool>),
   (const-name <rbt-lookup>, SOME typ <('a :: linorder, 'b) rbt ⇒ 'a → 'b>),
   (const-name <is-rbt>, SOME typ <('a :: linorder, 'b) rbt ⇒ bool>),
   (const-name <rbt-ins>, SOME typ <('a::linorder ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ 'a ⇒ 'b
⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-insert-with-key>, SOME typ <('a::linorder ⇒ 'b ⇒ 'b ⇒ 'b)
⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-insert-with>, SOME typ <('b ⇒ 'b ⇒ 'b) ⇒ ('a :: linorder)
⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-insert>, SOME typ <('a :: linorder) ⇒ 'b ⇒ ('a,'b) rbt ⇒
('a,'b) rbt>),
   (const-name <rbt-del-from-left>, SOME typ <('a::linorder) ⇒ ('a,'b) rbt ⇒ 'a
⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-del-from-right>, SOME typ <('a::linorder) ⇒ ('a,'b) rbt ⇒
'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-del>, SOME typ <('a::linorder) ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-delete>, SOME typ <('a::linorder) ⇒ ('a,'b) rbt ⇒ ('a,'b)
rbt>),
   (const-name <rbt-union-with-key>, SOME typ <('a::linorder ⇒ 'b ⇒ 'b ⇒ 'b)
⇒ ('a,'b) rbt ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-union-with>, SOME typ <('b ⇒ 'b ⇒ 'b) ⇒ ('a::linorder,'b)
rbt ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-union>, SOME typ <('a::linorder,'b) rbt ⇒ ('a,'b) rbt ⇒
('a,'b) rbt>),
   (const-name <rbt-map-entry>, SOME typ <'a::linorder ⇒ ('b ⇒ 'b) ⇒ ('a,'b)
rbt ⇒ ('a,'b) rbt>),
   (const-name <rbt-bulkload>, SOME typ <('a × 'b) list ⇒ ('a::linorder,'b) rbt>)]
,

```

hide-const (open) MR MB R B Empty entries keys fold gen-keys gen-entries

end

130 Abstract type of RBT trees

```

theory RBT
imports Main RBT-Impl
begin

```

130.1 Type definition

```

typedef (overloaded) ('a, 'b) rbt = {t :: ('a::linorder, 'b) RBT-Impl.rbt. is-rbt
t}
morphisms impl-of RBT
proof –
  have RBT-Impl.Empty ∈ ?rbt by simp

```

then show *?thesis* ..
qed

lemma *rbt-eq-iff*:
 $t1 = t2 \iff \text{impl-of } t1 = \text{impl-of } t2$
by (*simp add: impl-of-inject*)

lemma *rbt-eqI*:
 $\text{impl-of } t1 = \text{impl-of } t2 \implies t1 = t2$
by (*simp add: rbt-eq-iff*)

lemma *is-rbt-impl-of* [*simp, intro*]:
 $\text{is-rbt } (\text{impl-of } t)$
using *impl-of [of t]* **by** *simp*

lemma *RBT-impl-of* [*simp, code abstype*]:
 $\text{RBT } (\text{impl-of } t) = t$
by (*simp add: impl-of-inverse*)

130.2 Primitive operations

setup-lifting *type-definition-rbt*

lift-definition *lookup* :: $('a::\text{linorder}, 'b) \text{ rbt} \Rightarrow 'a \rightarrow 'b$ **is** *rbt-lookup* .

lift-definition *empty* :: $('a::\text{linorder}, 'b) \text{ rbt}$ **is** *RBT-Impl.Empty*
by (*simp add: empty-def*)

lift-definition *insert* :: $'a::\text{linorder} \Rightarrow 'b \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **is** *rbt-insert*
by *simp*

lift-definition *delete* :: $'a::\text{linorder} \Rightarrow ('a, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$ **is** *rbt-delete*
by *simp*

lift-definition *entries* :: $('a::\text{linorder}, 'b) \text{ rbt} \Rightarrow ('a \times 'b) \text{ list}$ **is** *RBT-Impl.entries* .

lift-definition *keys* :: $('a::\text{linorder}, 'b) \text{ rbt} \Rightarrow 'a \text{ list}$ **is** *RBT-Impl.keys* .

lift-definition *bulkload* :: $('a::\text{linorder} \times 'b) \text{ list} \Rightarrow ('a, 'b) \text{ rbt}$ **is** *rbt-bulkload* ..

lift-definition *map-entry* :: $'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a::\text{linorder}, 'b) \text{ rbt} \Rightarrow ('a, 'b) \text{ rbt}$
is *rbt-map-entry*
by *simp*

lift-definition *map* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a::\text{linorder}, 'b) \text{ rbt} \Rightarrow ('a, 'c) \text{ rbt}$ **is** *RBT-Impl.map*
by *simp*

lift-definition *fold* :: ($'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c$) \Rightarrow ($'a::\text{linorder}, 'b$) $\text{rbt} \Rightarrow 'c \Rightarrow 'c$ **is** *RBT-Impl.fold* .

lift-definition *union* :: ($'a::\text{linorder}, 'b$) $\text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **is** *rbt-union*
by (*simp add: rbt-union-is-rbt*)

lift-definition *foldi* :: ($'c \Rightarrow \text{bool}$) \Rightarrow ($'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c$) \Rightarrow ($'a::\text{linorder}, 'b$) $\text{rbt} \Rightarrow 'c \Rightarrow 'c$
is *RBT-Impl.foldi* .

lift-definition *combine-with-key* :: ($'a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b$) \Rightarrow ($'a::\text{linorder}, 'b$) $\text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$
is *RBT-Impl.rbt-union-with-key* **by** (*rule is-rbt-rbt-unionwk*)

lift-definition *combine* :: ($'b \Rightarrow 'b \Rightarrow 'b$) \Rightarrow ($'a::\text{linorder}, 'b$) $\text{rbt} \Rightarrow ('a, 'b) \text{rbt} \Rightarrow ('a, 'b) \text{rbt}$
is *RBT-Impl.rbt-union-with* **by** (*rule rbt-unionw-is-rbt*)

130.3 Derived operations

definition *is-empty* :: ($'a::\text{linorder}, 'b$) $\text{rbt} \Rightarrow \text{bool}$ **where**
 $[\text{code}]: \text{is-empty } t = (\text{case impl-of } t \text{ of } \text{RBT-Impl.Empty} \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

definition *filter* :: ($'a \Rightarrow 'b \Rightarrow \text{bool}$) \Rightarrow ($'a::\text{linorder}, 'b$) $\text{rbt} \Rightarrow ('a, 'b) \text{rbt}$ **where**
 $[\text{code}]: \text{filter } P \ t = \text{fold } (\lambda k \ v \ t. \text{if } P \ k \ v \text{ then insert } k \ v \ t \text{ else } t) \ t \ \text{empty}$

130.4 Abstract lookup properties

lemma *lookup-RBT*:
 $\text{is-rbt } t \Longrightarrow \text{lookup } (\text{RBT } t) = \text{rbt-lookup } t$
by (*simp add: lookup-def RBT-inverse*)

lemma *lookup-impl-of*:
 $\text{rbt-lookup } (\text{impl-of } t) = \text{lookup } t$
by *transfer (rule refl)*

lemma *entries-impl-of*:
 $\text{RBT-Impl.entries } (\text{impl-of } t) = \text{entries } t$
by *transfer (rule refl)*

lemma *keys-impl-of*:
 $\text{RBT-Impl.keys } (\text{impl-of } t) = \text{keys } t$
by *transfer (rule refl)*

lemma *lookup-keys*:
 $\text{dom } (\text{lookup } t) = \text{set } (\text{keys } t)$
by *transfer (simp add: rbt-lookup-keys)*

lemma *lookup-empty* [simp]:
 $\text{lookup empty} = \text{Map.empty}$
by (simp add: empty-def lookup-RBT fun-eq-iff)

lemma *lookup-insert* [simp]:
 $\text{lookup (insert } k \ v \ t) = (\text{lookup } t)(k \mapsto v)$
by transfer (rule rbt-lookup-rbt-insert)

lemma *lookup-delete* [simp]:
 $\text{lookup (delete } k \ t) = (\text{lookup } t)(k := \text{None})$
by transfer (simp add: rbt-lookup-rbt-delete restrict-complement-singleton-eq)

lemma *map-of-entries* [simp]:
 $\text{map-of (entries } t) = \text{lookup } t$
by transfer (simp add: map-of-entries)

lemma *entries-lookup*:
 $\text{entries } t1 = \text{entries } t2 \longleftrightarrow \text{lookup } t1 = \text{lookup } t2$
by transfer (simp add: entries-rbt-lookup)

lemma *lookup-bulkload* [simp]:
 $\text{lookup (bulkload } xs) = \text{map-of } xs$
by transfer (rule rbt-lookup-rbt-bulkload)

lemma *lookup-map-entry* [simp]:
 $\text{lookup (map-entry } k \ f \ t) = (\text{lookup } t)(k := \text{map-option } f \ (\text{lookup } t \ k))$
by transfer (rule rbt-lookup-rbt-map-entry)

lemma *lookup-map* [simp]:
 $\text{lookup (map } f \ t) \ k = \text{map-option } (f \ k) \ (\text{lookup } t \ k)$
by transfer (rule rbt-lookup-map)

lemma *lookup-combine-with-key* [simp]:
 $\text{lookup (combine-with-key } f \ t1 \ t2) \ k = \text{combine-options } (f \ k) \ (\text{lookup } t1 \ k) \ (\text{lookup } t2 \ k)$
by transfer (simp-all add: combine-options-def rbt-lookup-rbt-unionwk)

lemma *combine-altdef*: $\text{combine } f \ t1 \ t2 = \text{combine-with-key } (\lambda \cdot. f) \ t1 \ t2$
by transfer (simp add: rbt-union-with-def)

lemma *lookup-combine* [simp]:
 $\text{lookup (combine } f \ t1 \ t2) \ k = \text{combine-options } f \ (\text{lookup } t1 \ k) \ (\text{lookup } t2 \ k)$
by (simp add: combine-altdef)

lemma *fold-fold*:
 $\text{fold } f \ t = \text{List.fold (case-prod } f) \ (\text{entries } t)$
by transfer (rule RBT-Impl.fold-def)

lemma *impl-of-empty*:

impl-of empty = RBT-Impl.Empty

by *transfer (rule refl)*

lemma *is-empty-empty [simp]*:

is-empty t \longleftrightarrow t = empty

unfolding *is-empty-def* **by** *transfer (simp split: rbt.split)*

lemma *RBT-lookup-empty [simp]*:

rbt-lookup t = Map.empty \longleftrightarrow t = RBT-Impl.Empty

by *(cases t) (auto simp add: fun-eq-iff)*

lemma *lookup-empty-empty [simp]*:

lookup t = Map.empty \longleftrightarrow t = empty

by *transfer (rule RBT-lookup-empty)*

lemma *keys-empty-eq [simp]*:

⟨keys empty = []⟩

by *transfer simp*

lemma *sorted-keys [iff]*:

sorted (keys t)

by *transfer (simp add: RBT-Impl.keys-def rbt-sorted-entries)*

lemma *distinct-keys [iff]*:

distinct (keys t)

by *transfer (simp add: RBT-Impl.keys-def distinct-entries)*

lemma *finite-dom-lookup [simp, intro!]*: *finite (dom (lookup t))*

by *transfer simp*

lemma *lookup-union*: *lookup (union s t) = lookup s ++ lookup t*

by *transfer (simp add: rbt-lookup-rbt-union)*

lemma *lookup-in-tree*: *(lookup t k = Some v) = ((k, v) \in set (entries t))*

by *transfer (simp add: rbt-lookup-in-tree)*

lemma *keys-entries*: *(k \in set (keys t)) = ($\exists v. (k, v) \in$ set (entries t))*

by *transfer (simp add: keys-entries)*

lemma *fold-def-alt*:

fold f t = List.fold (case-prod f) (entries t)

by *transfer (auto simp: RBT-Impl.fold-def)*

lemma *distinct-entries*: *distinct (List.map fst (entries t))*

by *transfer (simp add: distinct-entries)*

lemma *sorted-entries*: *sorted (List.map fst (entries t))*

by *(transfer) (simp add: rbt-sorted-entries)*

lemma *non-empty-keys*: $t \neq \text{empty} \implies \text{keys } t \neq []$
by *transfer* (*simp add: non-empty-rbt-keys*)

lemma *keys-def-alt*:
 $\text{keys } t = \text{List.map fst } (\text{entries } t)$
by *transfer* (*simp add: RBT-Impl.keys-def*)

context
begin

private lemma *lookup-filter-aux*:
assumes *distinct* ($\text{List.map fst } xs$)
shows $\text{lookup } (\text{List.fold } (\lambda(k, v) t. \text{if } P \ k \ v \ \text{then } \text{insert } k \ v \ t \ \text{else } t) \ xs \ t) \ k =$
 $(\text{case map-of } xs \ k \ \text{of}$
 $\quad \text{None} \Rightarrow \text{lookup } t \ k$
 $\quad | \text{Some } v \Rightarrow \text{if } P \ k \ v \ \text{then } \text{Some } v \ \text{else } \text{lookup } t \ k)$
using *assms* **by** (*induction xs arbitrary: t*) (*force split: option.splits*)+

lemma *lookup-filter*:
 $\text{lookup } (\text{filter } P \ t) \ k =$
 $(\text{case lookup } t \ k \ \text{of } \text{None} \Rightarrow \text{None} \mid \text{Some } v \Rightarrow \text{if } P \ k \ v \ \text{then } \text{Some } v \ \text{else } \text{None})$
unfolding *filter-def* **using** *lookup-filter-aux* [*of entries t P empty k*]
by (*simp add: fold-fold distinct-entries split: option.splits*)

end

130.5 Quickcheck generators

quickcheck-generator *rbt predicate: is-rbt constructors: empty, insert*

130.6 Hide implementation details

lifting-update *rbt.lifting*
lifting-forget *rbt.lifting*

hide-const (**open**) *impl-of empty lookup keys entries bulkload delete map fold*
union insert map-entry foldi
is-empty filter
hide-fact (**open**) *empty-def lookup-def keys-def entries-def bulkload-def delete-def*
map-def fold-def
union-def insert-def map-entry-def foldi-def is-empty-def filter-def

end

131 Implementation of mappings with Red-Black Trees

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *HOL-Library.RBT-Impl*.

131.1 Data type and invariant

The type $(\text{'}k, \text{'}v) \text{ RBT-Impl.rbt}$ denotes red-black trees with keys of type $\text{'}k$ and values of type $\text{'}v$. To function properly, the key type must belong to the *linorder* class.

A value t of this type is a valid red-black tree if it satisfies the invariant *is-rbt* t . The abstract type $(\text{'}k, \text{'}v) \text{ RBT.rbt}$ always obeys this invariant, and for this reason you should only use this in our application. Going back to $(\text{'}k, \text{'}v) \text{ RBT-Impl.rbt}$ may be necessary in proofs if not yet proven properties about the operations must be established.

The interpretation function *RBT.lookup* returns the partial map represented by a red-black tree:

RBT.lookup

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

131.2 Operations

Currently, the following operations are supported:

RBT.empty

Returns the empty tree. $O(1)$

RBT.insert

Updates the map at a given position. $O(\log n)$

RBT.delete

Deletes a map entry at a given position. $O(\log n)$

RBT.entries

Return a corresponding key-value list for a tree.

RBT.bulkload

Builds a tree from a key-value list.

RBT.map-entry

Maps a single entry in a tree.

RBT.map

Maps all values in a tree. $O(n)$

RBT.fold

Folds over all entries in a tree. $O(n)$

131.3 Invariant preservation

<i>is-rbt</i> <i>rbt.Empty</i>	(<i>Empty-is-rbt</i>)
<i>is-rbt</i> ? <i>t</i> \implies <i>is-rbt</i> (<i>rbt-insert</i> ? <i>k</i> ? <i>v</i> ? <i>t</i>)	(<i>rbt-insert-is-rbt</i>)
<i>is-rbt</i> ? <i>t</i> \implies <i>is-rbt</i> (<i>rbt-delete</i> ? <i>k</i> ? <i>t</i>)	(<i>delete-is-rbt</i>)
<i>is-rbt</i> (<i>rbt-bulkload</i> ? <i>xs</i>)	(<i>bulkload-is-rbt</i>)
<i>is-rbt</i> (<i>rbt-map-entry</i> ? <i>k</i> ? <i>f</i> ? <i>t</i>) = <i>is-rbt</i> ? <i>t</i>	(<i>map-entry-is-rbt</i>)
<i>is-rbt</i> (<i>RBT-Impl.map</i> ? <i>f</i> ? <i>t</i>) = <i>is-rbt</i> ? <i>t</i>	(<i>map-is-rbt</i>)
$\llbracket \text{is-rbt } ?lt; \text{is-rbt } ?rt \rrbracket \implies \text{is-rbt } (\text{rbt-union } ?lt \text{ } ?rt)$	(<i>union-is-rbt</i>)

131.4 Map Semantics

lookup-empty

Mapping.lookup Mapping.empty ?*k* = *None*

lookup-insert

RBT.lookup (*RBT.insert* ?*k* ?*v* ?*t*) = (*RBT.lookup* ?*t*)(?*k* \mapsto ?*v*)

lookup-delete

Mapping.lookup (*Mapping.delete* ?*k* ?*m*) ?*k* = *None*

lookup-bulkload

RBT.lookup (*RBT.bulkload* ?*xs*) = *map-of* ?*xs*

lookup-map

RBT.lookup (*RBT.map* ?*f* ?*t*) ?*k* = *map-option* (?*f* ?*k*) (*RBT.lookup* ?*t* ?*k*)

end

132 Implementation of sets using RBT trees

```
theory RBT-Set
imports RBT Product-Lexorder
begin
```

133 Definition of code datatype constructors

```
definition Set :: ('a::linorder, unit) rbt  $\Rightarrow$  'a set
  where Set t = {x . RBT.lookup t x = Some ()}
```

```
definition Coset :: ('a::linorder, unit) rbt  $\Rightarrow$  'a set
  where [simp]: Coset t = - Set t
```

134 Lemmas

134.1 Auxiliary lemmas

```
lemma [simp]: x  $\neq$  Some ()  $\longleftrightarrow$  x = None
by (auto simp: not-Some-eq[THEN iffD1])
```

```
lemma Set-set-keys: Set x = dom (RBT.lookup x)
by (auto simp: Set-def)
```

```
lemma finite-Set [simp, intro!]: finite (Set x)
by (simp add: Set-set-keys)
```

```
lemma set-keys: Set t = set(RBT.keys t)
by (simp add: Set-set-keys lookup-keys)
```

134.2 fold and filter

```
lemma finite-fold-rbt-fold-eq:
  assumes comp-fun-commute f
  shows Finite-Set.fold f A (set (RBT.entries t)) = RBT.fold (curry f) t A
proof -
  interpret comp-fun-commute: comp-fun-commute f
  by (fact assms)
  have *: remdups (RBT.entries t) = RBT.entries t
  using distinct-entries distinct-map by (auto intro: distinct-remdups-id)
  show ?thesis using assms by (auto simp: fold-def-alt comp-fun-commute.fold-set-fold-remdups
*)
qed
```

```
definition fold-keys :: ('a :: linorder  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, -) rbt  $\Rightarrow$  'b  $\Rightarrow$  'b
  where [code-unfold]: fold-keys f t A = RBT.fold ( $\lambda$ k - t. f k t) t A
```

```
lemma fold-keys-def-alt:
  fold-keys f t s = List.fold f (RBT.keys t) s
```

by (*auto simp: fold-map o-def split-def fold-def-alt keys-def-alt fold-keys-def*)

lemma *finite-fold-fold-keys*:

assumes *comp-fun-commute f*

shows *Finite-Set.fold f A (Set t) = fold-keys f t A*

using *assms*

proof –

interpret *comp-fun-commute f by fact*

have *set (RBT.keys t) = fst ‘(set (RBT.entries t))* **by** (*auto simp: fst-eq-Domain keys-entries*)

moreover have *inj-on fst (set (RBT.entries t))* **using** *distinct-entries distinct-map* **by** *auto*

ultimately show *?thesis*

by (*auto simp add: set-keys fold-keys-def curry-def fold-image finite-fold-rbt-fold-eq*

comp-comp-fun-commute)

qed

definition *rbt-filter* :: *(‘a :: linorder \Rightarrow bool) \Rightarrow (‘a, ‘b) rbt \Rightarrow ‘a set* **where**

rbt-filter P t = RBT.fold (λk - A'. if P k then Set.insert k A' else A') t {}

lemma *Set-filter-rbt-filter*:

Set.filter P (Set t) = rbt-filter P t

by (*subst Set-filter-fold*)

(*simp-all add: fold-keys-def rbt-filter-def finite-fold-fold-keys [OF comp-fun-commute-filter-fold]*)

134.3 foldi and Ball

lemma *Ball-False: RBT-Impl.fold (λk v s. s \wedge P k) t False = False*

by (*induction t*) *auto*

lemma *rbt-foldi-fold-conj*:

RBT-Impl.foldi (λs . s = True) (λk v s. s \wedge P k) t val = RBT-Impl.fold (λk v s. s \wedge P k) t val

proof (*induction t arbitrary: val*)

case (*Branch c t1*) **then show** *?case*

by (*cases RBT-Impl.fold (λk v s. s \wedge P k) t1 True*) (*simp-all add: Ball-False*)

qed *simp*

lemma *foldi-fold-conj: RBT.foldi (λs . s = True) (λk v s. s \wedge P k) t val = fold-keys (λk s. s \wedge P k) t val*

unfolding *fold-keys-def* **including** *rbt.lifting* **by** *transfer (rule rbt-foldi-fold-conj)*

134.4 foldi and Bex

lemma *Bex-True: RBT-Impl.fold (λk v s. s \vee P k) t True = True*

by (*induction t*) *auto*

lemma *rbt-foldi-fold-disj*:

$RBT-Impl.foldi (\lambda s. s = False) (\lambda k v s. s \vee P k) t val = RBT-Impl.fold (\lambda k v s. s \vee P k) t val$

proof (*induction t arbitrary: val*)

case (*Branch c t1*) **then show** *?case*

by (*cases RBT-Impl.fold (\lambda k v s. s \vee P k) t1 False*) (*simp-all add: Bex-True*)

qed *simp*

lemma *foldi-fold-disj*: $RBT.foldi (\lambda s. s = False) (\lambda k v s. s \vee P k) t val = fold-keys (\lambda k s. s \vee P k) t val$

unfolding *fold-keys-def* **including** *rbt.lifting* **by** *transfer* (*rule rbt-foldi-fold-disj*)

134.5 folding over non empty trees and selecting the minimal and maximal element

134.5.1 concrete

The concrete part is here because it’s probably not general enough to be moved to *RBT-Impl*

definition *rbt-fold1-keys* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a::linorder, 'b) RBT-Impl.rbt \Rightarrow 'a$

where *rbt-fold1-keys* *f* *t* = *List.fold f (tl(RBT-Impl.keys t)) (hd(RBT-Impl.keys t))*

minimum definition *rbt-min* :: $('a::linorder, unit) RBT-Impl.rbt \Rightarrow 'a$

where *rbt-min* *t* = *rbt-fold1-keys min t*

lemma *key-le-right*: $rbt-sorted (Branch c lt k v rt) \Longrightarrow (\bigwedge x. x \in set (RBT-Impl.keys rt) \Longrightarrow k \leq x)$

by (*auto simp: rbt-greater-prop less-imp-le*)

lemma *left-le-key*: $rbt-sorted (Branch c lt k v rt) \Longrightarrow (\bigwedge x. x \in set (RBT-Impl.keys lt) \Longrightarrow x \leq k)$

by (*auto simp: rbt-less-prop less-imp-le*)

lemma *fold-min-triv*:

fixes *k* :: $- :: linorder$

shows $(\forall x \in set xs. k \leq x) \Longrightarrow List.fold min xs k = k$

by (*induct xs*) (*auto simp add: min-def*)

lemma *rbt-min-simps*:

is-rbt (*Branch c RBT-Impl.Empty k v rt*) $\Longrightarrow rbt-min (Branch c RBT-Impl.Empty k v rt) = k$

by (*auto intro: fold-min-triv dest: key-le-right is-rbt-rbt-sorted simp: rbt-fold1-keys-def rbt-min-def*)

fun *rbt-min-opt* **where**

rbt-min-opt (*Branch c RBT-Impl.Empty k v rt*) = *k* |

rbt-min-opt (*Branch c (Branch lc llc lk lv lrt) k v rt*) = *rbt-min-opt* (*Branch lc llc lk lv lrt*)

lemma *rbt-min-opt-Branch*:

$t1 \neq \text{rbt.Empty} \implies \text{rbt-min-opt } (\text{Branch } c \ t1 \ k \ () \ t2) = \text{rbt-min-opt } t1$
by (*cases t1*) *auto*

lemma *rbt-min-opt-induct* [*case-names empty left-empty left-non-empty*]:

fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes $P \ \text{rbt.Empty}$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t1 = \text{rbt.Empty} \implies P \ (\text{Branch } \text{color } t1 \ a \ b \ t2)$
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. P \ t1 \implies P \ t2 \implies t1 \neq \text{rbt.Empty} \implies P \ (\text{Branch } \text{color } t1 \ a \ b \ t2)$
shows $P \ t$
using *assms*
proof (*induct t*)
case *Empty*
then show *?case* **by** *simp*
next
case (*Branch x1 t1 x3 x4 t2*)
then show *?case* **by** (*cases t1 = rbt.Empty*) *simp-all*
qed

lemma *rbt-min-opt-in-set*:

fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes $t \neq \text{rbt.Empty}$
shows $\text{rbt-min-opt } t \in \text{set } (\text{RBT-Impl.keys } t)$
using *assms* **by** (*induction t rule: rbt-min-opt.induct*) (*auto*)

lemma *rbt-min-opt-is-min*:

fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{ RBT-Impl.rbt}$
assumes *rbt-sorted t*
assumes $t \neq \text{rbt.Empty}$
shows $\bigwedge y. y \in \text{set } (\text{RBT-Impl.keys } t) \implies y \geq \text{rbt-min-opt } t$
using *assms*
proof (*induction t rule: rbt-min-opt-induct*)
case *empty*
then show *?case* **by** *simp*
next
case *left-empty*
then show *?case* **by** (*auto intro: key-le-right simp del: rbt-sorted.simps*)
next
case (*left-non-empty c t1 k v t2 y*)
then consider $y = k \mid y \in \text{set } (\text{RBT-Impl.keys } t1) \mid y \in \text{set } (\text{RBT-Impl.keys } t2)$
by *auto*
then show *?case*
proof *cases*
case 1
with *left-non-empty* **show** *?thesis*

```

    by (auto simp add: rbt-min-opt-Branch intro: left-le-key rbt-min-opt-in-set)
  next
    case 2
    with left-non-empty show ?thesis
      by (auto simp add: rbt-min-opt-Branch)
  next
    case y: 3
    have rbt-min-opt t1 ≤ k
      using left-non-empty by (simp add: left-le-key rbt-min-opt-in-set)
    moreover have k ≤ y
      using left-non-empty y by (simp add: key-le-right)
    ultimately show ?thesis
      using left-non-empty y by (simp add: rbt-min-opt-Branch)
  qed
qed

lemma rbt-min-eq-rbt-min-opt:
  assumes t ≠ RBT-Impl.Empty
  assumes is-rbt t
  shows rbt-min t = rbt-min-opt t
proof -
  from assms have hd (RBT-Impl.keys t) ≠ tl (RBT-Impl.keys t) = RBT-Impl.keys
  t by (cases t) simp-all
  with assms show ?thesis
    by (simp add: rbt-min-def rbt-fold1-keys-def rbt-min-opt-is-min
      Min.set-eq-fold [symmetric] Min-eqI rbt-min-opt-in-set)
qed

maximum definition rbt-max :: ('a::linorder, unit) RBT-Impl.rbt ⇒ 'a
  where rbt-max t = rbt-fold1-keys max t

lemma fold-max-triv:
  fixes k :: - :: linorder
  shows (∀ x∈set xs. x ≤ k) ⇒ List.fold max xs k = k
by (induct xs) (auto simp add: max-def)

lemma fold-max-rev-eq:
  fixes xs :: ('a :: linorder) list
  assumes xs ≠ []
  shows List.fold max (tl xs) (hd xs) = List.fold max (tl (rev xs)) (hd (rev xs))
  using assms by (simp add: Max.set-eq-fold [symmetric])

lemma rbt-max-simps:
  assumes is-rbt (Branch c lt k v RBT-Impl.Empty)
  shows rbt-max (Branch c lt k v RBT-Impl.Empty) = k
proof -
  have List.fold max (tl (rev(RBT-Impl.keys lt @ [k]))) (hd (rev(RBT-Impl.keys
  lt @ [k]))) = k
    using assms by (auto intro!: fold-max-triv dest!: left-le-key is-rbt-rbt-sorted)

```

then show *?thesis* **by** (*auto simp add: rbt-max-def rbt-fold1-keys-def fold-max-rev-eq*)
qed

fun *rbt-max-opt* **where**

rbt-max-opt (*Branch c lt k v RBT-Impl.Empty*) = *k* |
rbt-max-opt (*Branch c lt k v (Branch rc rlc rk rv rrt)*) = *rbt-max-opt* (*Branch rc*
rlc rk rv rrt)

lemma *rbt-max-opt-Branch*:

t2 \neq *rbt.Empty* \implies *rbt-max-opt* (*Branch c t1 k () t2*) = *rbt-max-opt t2*
by (*cases t2*) *auto*

lemma *rbt-max-opt-induct* [*case-names empty right-empty right-non-empty*]:

fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *P rbt.Empty*
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. \ P \ t1 \implies P \ t2 \implies t2 = \text{rbt.Empty} \implies P \ (\text{Branch}$
color t1 a b t2)
assumes $\bigwedge \text{color } t1 \ a \ b \ t2. \ P \ t1 \implies P \ t2 \implies t2 \neq \text{rbt.Empty} \implies P \ (\text{Branch}$
color t1 a b t2)
shows *P t*
using *assms*
proof (*induct t*)
case *Empty*
then show *?case* **by** *simp*
next
case (*Branch x1 t1 x3 x4 t2*)
then show *?case* **by** (*cases t2 = rbt.Empty*) *simp-all*
qed

lemma *rbt-max-opt-in-set*:

fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *t* \neq *rbt.Empty*
shows *rbt-max-opt t* \in *set (RBT-Impl.keys t)*
using *assms* **by** (*induction t rule: rbt-max-opt.induct*) (*auto*)

lemma *rbt-max-opt-is-max*:

fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *rbt-sorted t*
assumes *t* \neq *rbt.Empty*
shows $\bigwedge y. \ y \in \text{set } (\text{RBT-Impl.keys } t) \implies y \leq \text{rbt-max-opt } t$
using *assms*
proof (*induction t rule: rbt-max-opt-induct*)
case *empty*
then show *?case* **by** *simp*
next
case *right-empty*
then show *?case* **by** (*auto intro: left-le-key simp del: rbt-sorted.simps*)
next
case (*right-non-empty c t1 k v t2 y*)

```

then consider  $y = k \mid y \in \text{set } (RBT\text{-}Impl.\text{keys } t2) \mid y \in \text{set } (RBT\text{-}Impl.\text{keys } t1)$ 
  by auto
then show ?case
proof cases
  case 1
    with right-non-empty show ?thesis
    by (auto simp add: rbt-max-opt-Branch intro: key-le-right rbt-max-opt-in-set)
  next
    case 2
    with right-non-empty show ?thesis
    by (auto simp add: rbt-max-opt-Branch)
  next
    case  $y: 3$ 
    have  $rbt\text{-}max\text{-}opt\ t2 \geq k$ 
    using right-non-empty by (simp add: key-le-right rbt-max-opt-in-set)
    moreover have  $y \leq k$ 
    using right-non-empty y by (simp add: left-le-key)
    ultimately show ?thesis
    using right-non-empty by (simp add: rbt-max-opt-Branch)
qed
qed

```

```

lemma rbt-max-eq-rbt-max-opt:
  assumes  $t \neq RBT\text{-}Impl.Empty$ 
  assumes is-rbt t
  shows  $rbt\text{-}max\ t = rbt\text{-}max\text{-}opt\ t$ 
proof –
  from assms have  $hd\ (RBT\text{-}Impl.\text{keys } t) \neq tl\ (RBT\text{-}Impl.\text{keys } t) = RBT\text{-}Impl.\text{keys } t$ 
  by (cases t) simp-all
  with assms show ?thesis
  by (simp add: rbt-max-def rbt-fold1-keys-def rbt-max-opt-is-max
    Max.set-eq-fold [symmetric] Max-eqI rbt-max-opt-in-set)
qed

```

134.5.2 abstract

context includes *rbt.lifting* **begin**

lift-definition *fold1-keys* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a::linorder, 'b) \text{ rbt} \Rightarrow 'a$
is *rbt-fold1-keys* .

lemma *fold1-keys-def-alt*:

```

 $fold1\text{-}keys\ f\ t = List.fold\ f\ (tl\ (RBT.\text{keys } t))\ (hd\ (RBT.\text{keys } t))$ 
by transfer (simp add: rbt-fold1-keys-def)

```

lemma *finite-fold1-fold1-keys*:

```

assumes semilattice f
assumes  $\neg RBT.is\text{-}empty\ t$ 
shows  $semilattice\text{-}set.F\ f\ (Set\ t) = fold1\text{-}keys\ f\ t$ 

```

proof –

from $\langle \text{semilattice } f \rangle$ **interpret** *semilattice-set* *f* **by** (rule *semilattice-set.intro*)
show ?thesis **using** *assms*
by (auto simp: fold1-keys-def-alt set-keys fold-def-alt non-empty-keys set-eq-fold
[*symmetric*])
qed

minimum lift-definition *r-min* :: (*'a* :: *linorder*, *unit*) *rbt* \Rightarrow *'a* **is** *rbt-min* .

lift-definition *r-min-opt* :: (*'a* :: *linorder*, *unit*) *rbt* \Rightarrow *'a* **is** *rbt-min-opt* .

lemma *r-min-alt-def*: *r-min* *t* = *fold1-keys min* *t*
by *transfer* (simp add: *rbt-min-def*)

lemma *r-min-eq-r-min-opt*:
assumes \neg (*RBT.is-empty* *t*)
shows *r-min* *t* = *r-min-opt* *t*
using *assms* **unfolding** *is-empty-empty* **by** *transfer* (auto intro: *rbt-min-eq-rbt-min-opt*)

lemma *fold-keys-min-top-eq*:
fixes *t* :: (*'a*::{*linorder*,*bounded-lattice-top*}, *unit*) *rbt*
assumes \neg (*RBT.is-empty* *t*)
shows *fold-keys min* *t* *top* = *fold1-keys min* *t*

proof –

have *: $\bigwedge t. \text{RBT-Impl.keys } t \neq [] \implies \text{List.fold min (RBT-Impl.keys } t) \text{ top} =$
 $\text{List.fold min (hd (RBT-Impl.keys } t) \# \text{tl (RBT-Impl.keys } t)) \text{ top}$
by (simp add: *hd-Cons-tl*[*symmetric*])
have **: $\text{List.fold min (x \# xs) top} = \text{List.fold min xs } x \text{ for } x :: 'a \text{ and } xs$
by (simp add: *inf-min*[*symmetric*])
show ?thesis
using *assms*
unfolding *fold-keys-def-alt fold1-keys-def-alt is-empty-empty*
apply *transfer*
apply (*case-tac* *t*)
apply *simp*
apply (*subst* *)
apply *simp*
apply (*subst* **)
apply *simp*
done

qed

maximum lift-definition *r-max* :: (*'a* :: *linorder*, *unit*) *rbt* \Rightarrow *'a* **is** *rbt-max* .

lift-definition *r-max-opt* :: (*'a* :: *linorder*, *unit*) *rbt* \Rightarrow *'a* **is** *rbt-max-opt* .

lemma *r-max-alt-def*: *r-max* *t* = *fold1-keys max* *t*
by *transfer* (simp add: *rbt-max-def*)

lemma *r-max-eq-r-max-opt*:
 assumes $\neg (RBT.is-empty\ t)$
 shows $r-max\ t = r-max-opt\ t$
using *assms unfolding is-empty-empty by transfer (auto intro: rbt-max-eq-rbt-max-opt)*

lemma *fold-keys-max-bot-eq*:
 fixes $t :: ('a::\{linorder, bounded-lattice-bot\}, unit)\ rbt$
 assumes $\neg (RBT.is-empty\ t)$
 shows $fold-keys\ max\ t\ bot = fold1-keys\ max\ t$
proof –
 have *: $\bigwedge t. RBT-Impl.keys\ t \neq [] \implies List.fold\ max\ (RBT-Impl.keys\ t)\ bot =$
 $List.fold\ max\ (hd(RBT-Impl.keys\ t) \# tl(RBT-Impl.keys\ t))\ bot$
 by (*simp add: hd-Cons-tl[symmetric]*)
 have **: $List.fold\ max\ (x \# xs)\ bot = List.fold\ max\ xs\ x$ **for** $x :: 'a$ **and** xs
 by (*simp add: sup-max[symmetric]*)
 show ?thesis
 using *assms*
 unfolding *fold-keys-def-alt fold1-keys-def-alt is-empty-empty*
 apply *transfer*
 apply (*case-tac t*)
 apply *simp*
 apply (*subst **)
 apply *simp*
 apply (*subst ***)
 apply *simp*
 done
qed
end

135 Code equations

code-datatype *Set Coset*

declare *list.set[code]*

lemma *empty-Set [code]*:
 $Set.empty = Set\ RBT.empty$
by (*auto simp: Set-def*)

lemma *UNIV-Coset [code]*:
 $UNIV = Coset\ RBT.empty$
by (*auto simp: Set-def*)

lemma *is-empty-Set [code]*:
 $Set.is-empty\ (Set\ t) = RBT.is-empty\ t$
using *non-empty-keys [of t] by (auto simp add: set-keys)*

lemma *compl-code [code]*:

– $Set\ xs = Coset\ xs$
 – $Coset\ xs = Set\ xs$
by (*simp-all add: Set-def*)

lemma *member-code* [*code*]:
 $x \in (Set\ t) = (RBT.lookup\ t\ x = Some\ ())$
 $x \in (Coset\ t) = (RBT.lookup\ t\ x = None)$
by (*simp-all add: Set-def*)

lemma *insert-code* [*code*]:
 $Set.insert\ x\ (Set\ t) = Set\ (RBT.insert\ x\ ()\ t)$
 $Set.insert\ x\ (Coset\ t) = Coset\ (RBT.delete\ x\ t)$
by (*auto simp: Set-def*)

lemma *remove-code* [*code*]:
 $Set.remove\ x\ (Set\ t) = Set\ (RBT.delete\ x\ t)$
 $Set.remove\ x\ (Coset\ t) = Coset\ (RBT.insert\ x\ ()\ t)$
by (*auto simp: Set-def*)

lemma *inter-Set* [*code*]:
 $A \cap Set\ t = rbt-filter\ (\lambda k. k \in A)\ t$
by (*simp flip: Set-filter-rbt-filter add: inter-Set-filter*)

lemma *union-Set-Set* [*code*]:
 $Set\ t1 \cup Set\ t2 = Set\ (RBT.union\ t1\ t2)$
by (*auto simp add: lookup-union map-add-Some-iff Set-def*)

lemma *union-Set* [*code*]:
 $Set\ t \cup A = fold-keys\ Set.insert\ t\ A$
proof –
interpret *comp-fun-idem* *Set.insert*
by (*fact comp-fun-idem-insert*)
from *finite-fold-fold-keys[OF comp-fun-commute-axioms]*
show *?thesis* **by** (*auto simp add: union-fold-insert*)
qed

lemma *minus-Set* [*code*]:
 $A - Set\ t = fold-keys\ Set.remove\ t\ A$
proof –
interpret *comp-fun-idem* *Set.remove*
by (*fact comp-fun-idem-remove*)
from *finite-fold-fold-keys[OF comp-fun-commute-axioms]*
show *?thesis* **by** (*auto simp add: minus-fold-remove*)
qed

lemma *inter-Coset-Coset* [*code*]:
 $Coset\ t1 \cap Coset\ t2 = Coset\ (RBT.union\ t1\ t2)$
by (*auto simp add: lookup-union map-add-Some-iff Set-def*)

lemma *inter-Coset* [code]:

$A \cap \text{Coset } t = \text{fold-keys } \text{Set.remove } t \ A$

by (*simp add: Diff-eq [symmetric] minus-Set*)

lemma *union-Coset* [code]:

$\text{Coset } t \cup A = - \text{rbt-filter } (\lambda k. k \notin A) \ t$

proof –

have *: $\bigwedge A \ B. (-A \cup B) = -(-B \cap A)$ **by** *blast*

show ?thesis **by** (*simp del: boolean-algebra-class.compl-inf add: * inter-Set*)

qed

lemma *minus-Coset* [code]:

$A - \text{Coset } t = \text{rbt-filter } (\lambda k. k \in A) \ t$

by (*simp add: inter-Set[simplified Int-commute]*)

lemma *filter-Set* [code]:

$\text{Set.filter } P \ (\text{Set } t) = \text{rbt-filter } P \ t$

by (*fact Set-filter-rbt-filter*)

lemma *image-Set* [code]:

$\text{image } f \ (\text{Set } t) = \text{fold-keys } (\lambda k \ A. \text{Set.insert } (f \ k) \ A) \ t \ \{\}$

proof –

have *comp-fun-commute* $(\lambda k. \text{Set.insert } (f \ k))$

by *standard auto*

then show ?thesis

by (*auto simp add: image-fold-insert intro!: finite-fold-fold-keys*)

qed

lemma *Ball-Set* [code]:

$\text{Ball } (\text{Set } t) \ P \longleftrightarrow \text{RBT.foldi } (\lambda s. s = \text{True}) \ (\lambda k \ v \ s. s \wedge P \ k) \ t \ \text{True}$

proof –

have *comp-fun-commute* $(\lambda k \ s. s \wedge P \ k)$

by *standard auto*

then show ?thesis

by (*simp add: foldi-fold-conj[symmetric] Ball-fold finite-fold-fold-keys*)

qed

lemma *Bex-Set* [code]:

$\text{Bex } (\text{Set } t) \ P \longleftrightarrow \text{RBT.foldi } (\lambda s. s = \text{False}) \ (\lambda k \ v \ s. s \vee P \ k) \ t \ \text{False}$

proof –

have *comp-fun-commute* $(\lambda k \ s. s \vee P \ k)$

by *standard auto*

then show ?thesis

by (*simp add: foldi-fold-disj[symmetric] Bex-fold finite-fold-fold-keys*)

qed

lemma *subset-code* [code]:

$\text{Set } t \leq B \longleftrightarrow (\forall x \in \text{Set } t. x \in B)$

$A \leq \text{Coset } t \longleftrightarrow (\forall y \in \text{Set } t. y \notin A)$

by *auto*

lemma *subset-Coset-empty-Set-empty* [code]:

$\text{Coset } t1 \leq \text{Set } t2 \iff (\text{case } (\text{RBT.impl-of } t1, \text{RBT.impl-of } t2) \text{ of}$
 $(\text{rbt.Empty}, \text{rbt.Empty}) \Rightarrow \text{False} \mid$
 $(-, -) \Rightarrow \text{Code.abort } (\text{STR "non-empty-trees"}) (\lambda-. \text{Coset } t1 \leq \text{Set } t2))$

proof –

have *: $\bigwedge t. \text{RBT.impl-of } t = \text{rbt.Empty} \implies t = \text{RBT rbt.Empty}$
by (*subst(asm) RBT-inverse[symmetric]*) (*auto simp: impl-of-inject*)
have **: *eq-onp is-rbt rbt.Empty rbt.Empty unfolding eq-onp-def by simp*
show ?thesis
by (*auto simp: Set-def lookup.abs-eq[OF **] dest!: * split: rbt.split*)

qed

A frequent case – avoid intermediate sets

lemma [code-unfold]:

$\text{Set } t1 \subseteq \text{Set } t2 \iff \text{RBT.foldi } (\lambda s. s = \text{True}) (\lambda k v s. s \wedge k \in \text{Set } t2) t1 \text{ True}$
by (*simp add: subset-code Ball-Set*)

lemma *card-Set* [code]:

$\text{card } (\text{Set } t) = \text{fold-keys } (\lambda-. n. n + 1) t 0$
by (*auto simp add: card.eq-fold intro: finite-fold-fold-keys comp-fun-commute-const*)

lemma *sum-Set* [code]:

$\text{sum } f (\text{Set } xs) = \text{fold-keys } (\text{plus} \circ f) xs 0$

proof –

have *comp-fun-commute* $(\lambda x. (+) (f x))$
by *standard (auto simp: ac-simps)*
then show ?thesis
by (*auto simp add: sum.eq-fold finite-fold-fold-keys o-def*)

qed

lemma *prod-Set* [code]:

$\text{prod } f (\text{Set } xs) = \text{fold-keys } (\text{times} \circ f) xs 1$

proof –

have *comp-fun-commute* $(\lambda x. (*) (f x))$
by *standard (auto simp: ac-simps)*
then show ?thesis
by (*auto simp add: prod.eq-fold finite-fold-fold-keys o-def*)

qed

lemma *the-elem-set* [code]:

fixes $t :: ('a :: \text{linorder}, \text{unit}) \text{rbt}$
shows *the-elem* $(\text{Set } t) = (\text{case } \text{RBT.impl-of } t \text{ of}$
 $(\text{Branch } \text{RBT-Impl.B } \text{RBT-Impl.Empty } x \ () \ \text{RBT-Impl.Empty}) \Rightarrow x$
 $\mid - \Rightarrow \text{Code.abort } (\text{STR "not-a-singleton-tree"}) (\lambda-. \text{the-elem } (\text{Set } t)))$

proof –

{
fix $x :: 'a :: \text{linorder}$

```

let ?t = Branch RBT-Impl.B RBT-Impl.Empty x () RBT-Impl.Empty
have *: ?t ∈ {t. is-rbt t} unfolding is-rbt-def by auto
then have **:eq-onp is-rbt ?t ?t unfolding eq-onp-def by auto

have RBT.impl-of t = ?t  $\implies$  the-elem (Set t) = x
  by (subst(asm) RBT-inverse[symmetric, OF *])
    (auto simp: Set-def the-elem-def lookup.abs-eq[OF **] impl-of-inject)
}
then show ?thesis
  by(auto split: rbt.split unit.split color.split)
qed

```

```

lemma Pow-Set [code]: Pow (Set t) = fold-keys (λx A. A ∪ Set.insert x ‘ A) t
  {}
by (simp add: Pow-fold finite-fold-fold-keys[OF comp-fun-commute-Pow-fold])

```

```

lemma product-Set [code]:
  Product-Type.product (Set t1) (Set t2) =
    fold-keys (λx A. fold-keys (λy. Set.insert (x, y)) t2 A) t1 {}
proof –
  have *: comp-fun-commute (λy. Set.insert (x, y)) for x
    by standard auto
  show ?thesis using finite-fold-fold-keys[OF comp-fun-commute-product-fold, of
    Set t2 {} t1]
  by (simp add: product-fold Product-Type.product-def finite-fold-fold-keys[OF *])
qed

```

```

lemma Id-on-Set [code]: Id-on (Set t) = fold-keys (λx. Set.insert (x, x)) t {}
proof –
  have comp-fun-commute (λx. Set.insert (x, x))
    by standard auto
  then show ?thesis
    by (auto simp add: Id-on-fold intro!: finite-fold-fold-keys)
qed

```

```

lemma Image-Set [code]:
  (Set t) “ S = fold-keys (λ(x,y) A. if x ∈ S then Set.insert y A else A) t {}
by (auto simp add: Image-fold finite-fold-fold-keys[OF comp-fun-commute-Image-fold])

```

```

lemma trancl-set-ntrancl [code]:
  trancl (Set t) = ntrancl (card (Set t) – 1) (Set t)
by (simp add: finite-trancl-ntrancl)

```

```

lemma relcomp-Set[code]:
  (Set t1) O (Set t2) = fold-keys
    (λ(x,y) A. fold-keys (λ(w,z) A'. if y = w then Set.insert (x,z) A' else A') t2 A)
  t1 {}
proof –
  interpret comp-fun-idem Set.insert

```

```

    by (fact comp-fun-idem-insert)
    have *:  $\bigwedge x y. \text{comp-fun-commute } (\lambda(w, z) A'. \text{if } y = w \text{ then } \text{Set.insert } (x, z) A' \text{ else } A')$ 
    by standard (auto simp add: fun-eq-iff)
    show ?thesis
    using finite-fold-fold-keys[OF comp-fun-commute-relcomp-fold, of Set t2 {} t1]
    by (simp add: relcomp-fold finite-fold-fold-keys[OF *])
qed

```

```

lemma wf-set: wf (Set t) = acyclic (Set t)
  by (simp add: wf-iff-acyclic-if-finite)

```

```

lemma wf-code-set[code]: wf-code (Set t) = acyclic (Set t)
  unfolding wf-code-def using wf-set .

```

```

lemma Min-fin-set-fold [code]:
  Min (Set t) =
    (if RBT.is-empty t
     then Code.abort (STR "not-non-empty-tree") ( $\lambda-. \text{Min } (Set t)$ )
     else r-min-opt t)
proof -
  have *: semilattice (min :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a) ..
  with finite-fold1-fold1-keys [OF *, folded Min-def]
  show ?thesis
  by (simp add: r-min-alt-def r-min-eq-r-min-opt [symmetric])
qed

```

```

lemma Inf-fin-set-fold [code]:
  Inf-fin (Set t) = Min (Set t)
by (simp add: inf-min Inf-fin-def Min-def)

```

```

lemma Inf-Set-fold:
  fixes t :: ('a :: {linorder, complete-lattice}, unit) rbt
  shows Inf (Set t) = (if RBT.is-empty t then top else r-min-opt t)
proof -
  have comp-fun-commute (min :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a)
  by standard (simp add: fun-eq-iff ac-simps)
  then have t  $\neq$  RBT.empty  $\implies$  Finite-Set.fold min top (Set t) = fold1-keys min t
  by (simp add: finite-fold-fold-keys fold-keys-min-top-eq)
  then show ?thesis
  by (auto simp add: Inf-fold-inf inf-min empty-Set[symmetric]
    r-min-eq-r-min-opt[symmetric] r-min-alt-def)
qed

```

```

lemma Max-fin-set-fold [code]:
  Max (Set t) =
    (if RBT.is-empty t
     then Code.abort (STR "not-non-empty-tree") ( $\lambda-. \text{Max } (Set t)$ )

```

```

    else r-max-opt t)
proof –
  have *: semilattice (max :: 'a ⇒ 'a ⇒ 'a) ..
  with finite-fold1-fold1-keys [OF *, folded Max-def]
  show ?thesis
    by (simp add: r-max-alt-def r-max-eq-r-max-opt [symmetric])
qed

lemma Sup-fin-set-fold [code]:
  Sup-fin (Set t) = Max (Set t)
by (simp add: sup-max Sup-fin-def Max-def)

lemma Sup-Set-fold:
  fixes t :: ('a :: {linorder, complete-lattice}, unit) rbt
  shows Sup (Set t) = (if RBT.is-empty t then bot else r-max-opt t)
proof –
  have comp-fun-commute (max :: 'a ⇒ 'a ⇒ 'a)
    by standard (simp add: fun-eq-iff ac-simps)
  then have t ≠ RBT.empty ⇒ Finite-Set.fold max bot (Set t) = fold1-keys max
  t
    by (simp add: finite-fold-fold-keys fold-keys-max-bot-eq)
  then show ?thesis
    by (auto simp add: Sup-fold-sup sup-max empty-Set[symmetric]
      r-max-eq-r-max-opt[symmetric] r-max-alt-def)
qed

context
begin

qualified definition Inf' :: 'a :: {linorder, complete-lattice} set ⇒ 'a
  where [code-abbrev]: Inf' = Inf

lemma Inf'-Set-fold [code]:
  Inf' (Set t) = (if RBT.is-empty t then top else r-min-opt t)
  by (simp add: Inf'-def Inf-Set-fold)

qualified definition Sup' :: 'a :: {linorder, complete-lattice} set ⇒ 'a
  where [code-abbrev]: Sup' = Sup

lemma Sup'-Set-fold [code]:
  Sup' (Set t) = (if RBT.is-empty t then bot else r-max-opt t)
  by (simp add: Sup'-def Sup-Set-fold)

end

lemma [code]:
  Gcdfin (Set t) = fold-keys gcd t (0 :: 'a :: {semiring-gcd, linorder})
proof –
  have comp-fun-commute (gcd :: 'a ⇒ -)

```

```

    by standard (simp add: fun-eq-iff ac-simps)
  with finite-fold-fold-keys [of - 0 t]
  have Finite-Set.fold gcd 0 (Set t) = fold-keys gcd t 0
    by blast
  then show ?thesis
    by (simp add: Gcd-fin.eq-fold)
qed

```

```

lemma [code]:
  Gcd (Set t) = (Gcdfin (Set t) :: nat)
  by simp

```

```

lemma [code]:
  Gcd (Set t) = (Gcdfin (Set t) :: int)
  by simp

```

```

lemma [code]:
  Lcmfin (Set t) = fold-keys lcm t (1::'a::{semiring-gcd, linorder})
proof -
  have comp-fun-commute (lcm :: 'a ⇒ -)
    by standard (simp add: fun-eq-iff ac-simps)
  with finite-fold-fold-keys [of - 1 t]
  have Finite-Set.fold lcm 1 (Set t) = fold-keys lcm t 1
    by blast
  then show ?thesis
    by (simp add: Lcm-fin.eq-fold)
qed

```

```

lemma [code]:
  Lcm (Set t) = (Lcmfin (Set t) :: nat)
  by simp

```

```

lemma [code]:
  Lcm (Set t) = (Lcmfin (Set t) :: int)
  by simp

```

```

lemma sorted-list-set [code]: sorted-list-of-set (Set t) = RBT.keys t
  by (auto simp add: set-keys intro: sorted-distinct-set-unique)

```

```

lemma Least-code [code]:
  ⟨Lattices-Big.Least (Set t) = (if RBT.is-empty t then Lattices-Big.Least-abort {}
  else Min (Set t))⟩
  apply (auto simp add: Lattices-Big.Least-abort-def simp flip: empty-Set)
  apply (subst Least-Min)
  using is-empty-Set
  apply auto
  done

```

```

lemma Greatest-code [code]:

```

```

  <Lattices-Big.Greatest (Set t) = (if RBT.is-empty t then Lattices-Big.Greatest-abort
  {} else Max (Set t))>
  apply (auto simp add: Lattices-Big.Greatest-abort-def simp flip: empty-Set)
  apply (subst Greatest-Max)
  using is-empty-Set
  apply auto
  done

```

```

lemma [code]:
  <Option.these A = the ‘ Set.filter (Not ∘ Option.is-none) A>
  by (fact Option.these-eq)

```

```

lemma [code]:
  <Option.image-filter f A = Option.these (image f A)>
  by (fact Option.image-filter-eq)

```

```

lemma [code]:
  <Set.can-select P A = is-singleton (Set.filter P A)>
  by (fact Set.can-select-iff-is-singleton)

```

```

declare [[code drop:
  <Inf :: - ⇒ 'a set>
  <Sup :: - ⇒ 'a set>
  <Inf :: - ⇒ 'a Predicate.pred>
  <Sup :: - ⇒ 'a Predicate.pred>
  pred-of-set
  Wellfounded.acc
]]

```

```

hide-const (open) RBT-Set.Set RBT-Set.Coset

```

```

end

```

```

theory Time-Manual
imports HOL-Library.Time-Commands
begin

```

136 Introduction

This manual describes the framework for the automatic definition of step-counting ‘running-time’ functions from HOL functions. The principles of the translation are described in Section 1.5, Running Time, of the book Functional Data Structures and Algorithms. A Proof Assistant Approach. <https://fdsa-book.net> To load the framework import *HOL-Library.Time-Commands* The framework was implemented by Jonas Stahl.

As a first simple example consider *len*, which we define here returning an *int* (to distinguish it from the time functions returning *nat*):

```

fun len :: 'a list  $\Rightarrow$  int where
  len [] = 0 |
  len (x#xs) = 1 + len xs

```

```

time-fun len

```

Command *time-fun* defines a new function *T-len* of type $'a \text{ list} \Rightarrow \text{nat}$, the time function for *len* that counts the number of computation steps. The definition is printed by *time-fun*: *fun T-len :: 'a list \Rightarrow nat where T-len [] = 1 | T-len (x # xs) = T-len xs + 1* The details of this translation are described in the book referenced above. This manual is about the use of the time framework.

Command *time-fun f* retrieves the definition of *f* and defines a corresponding step-counting running-time function *T-f*. For all auxiliary functions used by *f* (excluding constructors and predefined functions (see below)), running time functions must already have been defined. Example:

```

fun aux :: 'a  $\Rightarrow$  'a where
  aux x = x

```

```

time-fun aux

```

```

fun main :: bool  $\Rightarrow$  bool where
  main x = aux x

```

```

time-fun main

```

For functions defined by *definition*, there is a corresponding *time-definition* command. Example:

```

definition gdef :: 'a  $\Rightarrow$  'a where gdef x = x

```

```

time-definition gdef

```

```

thm T-gdef.simps

```

Note that *T-gdef* is defined via *fun*, which means that the defining equation is not named *T-gdef-def* but *T-gdef.simps* and is a simp-rule.

The time functions for many standard functions (in particular on lists) are already defined in theory *HOL-Library.Time-Functions* and basic upper bounds are proved.

137 Termination

If the definition of a recursive function requires a manual termination proof, use *time-function* accompanied by a *termination* command.

```

function sum-to :: int  $\Rightarrow$  int  $\Rightarrow$  int where
  sum-to i j = (if j  $\leq$  i then 0 else i + sum-to (i+1) j)

```

```

by pat-completeness auto
termination
  by (relation measure ( $\lambda(i,j). \text{nat}(j - i)$ )) auto

time-function sum-to
termination
  by (relation measure ( $\lambda(i,j). \text{nat}(j - i)$ )) auto

```

138 Partial Functions

Partial functions can also be ‘timed’.

```

partial-function (tailrec) positive :: int  $\Rightarrow$  bool where
positive i = (if i = 1 then True else positive (i-1))

```

```

time-partial-function positive

```

The difference is that *T-positive* has return type *nat option* because *positive* may not terminate.

Timing a function defined with *partial-function* (*option*) is trickier and we do not go into it here.

139 Higher-Order Functions

A large subclass of higher-order functions are supported, covering *map*, *filter* and other standard functions. For example,

```

time-fun map

```

defines a time function *T-map* :: (*a* \Rightarrow *nat*) \Rightarrow '*a list* \Rightarrow *nat*. The first argument (called *T-f* below) is the time function for the first argument *f* of *map*. We ignore the definition of *T-map* because the output of *time-fun map* suggests that you should add these lemmas

```

lemma T-map-simps [simp,code]:
  T-map T-f [] = 1
  T-map T-f (x # xs) = T-f x + T-map T-f xs + 1
by (simp-all add: T-map-def)

```

which are what you would expect as defining equations. You can click on the suggestion to have it copied into your theory. Afterwards, you can work with *T-map* as if it were defined via those equations.

In general, things are a bit more complicated, which is why *T-map* is defined the way it is. Consider

```

fun foldl :: (b  $\Rightarrow$  'a  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  'a list  $\Rightarrow$  'b where
foldl f a [] = a |
foldl f a (x # xs) = foldl f (f a x) xs

```


time-fun *foldl*

This definition is generated:

```
fun T-foldl :: ('b ⇒ 'a ⇒ 'b) × ('b ⇒ 'a ⇒ nat) ⇒ 'b ⇒ 'a list ⇒ nat
where T-foldl (f, T-f) a [] = 1 | T-foldl (f, T-f) a (x # xs) = T-f a x +
T-foldl f (f a x) xs + 1
```

The meaning of the pair $(f, T-f)$ is obvious. The difference to $T-map$ is that $T-foldl$ needs not just $T-f$ (like $T-map$) but also f . Function $T-map$ does not need f : in the recursion equation $map\ f\ (x\ \# \ xs) = f\ x\ \# \ map\ f\ xs$ the result of subterm $f\ x$ is irrelevant for the computation of $T-map$ because the running time of $(\#)$ is constant. This is in contrast to $foldl$, whose running time may depend on its second argument.

All higher-order functions are translated like $foldl$, but if the first element in $(f, T-f)$ is unused, a simplified definition is derived. This is the case for $T-map$.

In case you wonder how it is ensured that $T-foldl$ is always passed a corresponding pair of a function and its timing function: this is the responsibility of the time framework when translating functions that use $foldl$. Example:

definition *inc* :: *int* ⇒ *'a* ⇒ *int* **where** *inc i x* = *i+1*

definition *len2 xs* = *foldl inc 0 xs*

time-definition *inc*

time-definition *len2*

In the defining equation $T-len2\ xs = T-foldl\ (inc, T-inc)\ 0\ xs$ we find the correct pair $(inc, T-inc)$.

139.1 Limitations

Partial application and lambda-abstraction are currently not supported. They need to be replaced by additional function definitions, if possible. For example,

definition *fHO* :: *bool list* ⇒ *bool list* **where** $\langle fHO = map\ (\lambda x. x \wedge x) \rangle$

is not acceptable (i.e. *time-definition fHO* fails), but can be replaced with

definition *double* :: *int* ⇒ *int* **where** $\langle double\ i = 2 * i \rangle$

definition *fHO'* :: *int list* ⇒ *int list* **where** $\langle fHO'\ xs = map\ double\ xs \rangle$

time-definition *double*

time-definition *fHO'*

That is why in the definition of *len2* above we could not just write $foldl\ (\lambda i\ x. i+1)\ 0\ xs$.

140 Predefined Functions

The time framework requires executable functions. However, many basic types and functions are not defined via *datatype* and *fun* but in an abstract mathematical fashion and are not executable, i.e. the time framework does not apply (or gives the ‘wrong’ result).

In order to model actual hardware that executes these predefined functions in constant time, there is a command for axiomatically declaring that some function takes 0 time. (This is how we model constant time, to simplify the resulting time expressions. This does not change the asymptotic running time of user-defined functions using the predefined functions because 1 is added for every user-defined function call.) Theory *HOL–Library.Time-Commands* declares a number of predefined functions as 0-time functions. This includes $(+)$, $-$, $(*)$, $/$, *div*, *min*, *max*, $<$, \leq , \neg , \wedge , \vee and $=$ and can be extended with the command *time-fun-0*. This feature has to be used with care:

- Many of these functions are polymorphic and reside in type classes. The constant-time assumption is justified only for those types where the hardware offers suitable support, e.g. numeric types. The argument size is implicitly bounded, too.
- The constant-time assumption for $(=)$ is justified for recursive data types such as lists and trees as long as the comparison is of the form $t = c$ where c is a constant term, for example $xs = []$.

Users of the time framework need to ensure that 0-time functions are used only within these bounds.

141 Locales

If we want to apply the time framework to a function g defined within a locale, we need to add additional locale parameters $T\text{-}f :: \tau \Rightarrow \text{nat}$ for every locale parameter $f :: \tau \Rightarrow \tau'$ used in the definition of g .

In the following example we do not only parameterize the locale with $T\text{-}f$ but also assume a property of $T\text{-}f$. As a result we can prove a property of $T\text{-}g$ inside the locale:

lemma *T-map-sum*: $T\text{-map } T\text{-}f \text{ } xs = \text{sum-list } (\text{map } T\text{-}f \text{ } xs) + \text{length } xs + 1$
by(*induction xs*) (*auto*)

```

locale LT =
fixes f :: 'a  $\Rightarrow$  'a
and T-f :: 'a  $\Rightarrow$  nat
assumes T-f:  $T\text{-}f \text{ } x \leq 1$ 
begin

```

definition g **where** $g\ xs = \text{map}\ f\ xs$

time-definition g

lemma $\text{sum-list-map-}T\text{-}f\text{-ub}$: $\text{sum-list}\ (\text{map}\ T\text{-}f\ xs) \leq \text{length}\ xs$

proof($\text{induction}\ xs$)

case ($\text{Cons}\ a\ xs$) **thus** $?case$ **using** $T\text{-}f[\text{of}\ a]$ **by** auto

qed simp

lemma $T\text{-}g\text{-ub}$: $T\text{-}g\ xs \leq 2 * \text{length}\ xs + 1$

unfolding $T\text{-}g.\text{sims}\ T\text{-map-sum}$ **using** $\text{sum-list-map-}T\text{-}f\text{-ub}[\text{of}\ xs]$

by linarith

end

Of course now you need to prove $T\text{-}f\ x \leq 1$ for every interpretation of the locale. A more flexible approach is not to constrain $T\text{-}f$ inside the locale. It may then be difficult to derive a generic time bound for $T\text{-}g$ inside the locale (in the above example it would not be difficult). If that is the case, one may also derive a bound for $T\text{-}g$ conditional on some specific bound for $T\text{-}f$. Or one can derive the bound for $T\text{-}g$ after a specific interpretation with a specific $T\text{-}f$. For a larger realistic example of the latter approach see theory *HOL-Data-Structures.Time-Locale-Example*.

142 Fine Points

Time functions for mutually recursive functions f, g, \dots : $\text{time-fun}\ f\ g\ \dots$

If you want to generate time functions not from the defining equations of a function but from lemmas proved as equations, you can provide those lemmas explicitly. Example:

fun $f0 :: \text{nat} \Rightarrow \text{nat}$ **where**

$f0\ 0 = 0$ |

$f0\ (\text{Suc}\ n) = f0\ n$

lemma $f0\text{-eq}$: $f0\ n = 0$

by ($\text{induction}\ n$) auto

time-fun $f0$ **equations** $f0\text{-eq}$

The T - prefix can be changed by modifying the time-prefix attribute. Example:

declare $[[\text{time-prefix} = t]]$

The time framework is not verified (which is why the framework always prints out what it defines). There is no underlying formal model. This remains future work.

end

theory *Suc-Notation*
imports *Main*
begin

Nested *Suc* terms of depth $2 \leq n \leq 9$ are abbreviated with new notations *Sucⁿ*:

abbreviation (*input*) *Suc2* **where** *Suc2* $n \equiv \text{Suc } (\text{Suc } n)$
abbreviation (*input*) *Suc3* **where** *Suc3* $n \equiv \text{Suc } (\text{Suc2 } n)$
abbreviation (*input*) *Suc4* **where** *Suc4* $n \equiv \text{Suc } (\text{Suc3 } n)$
abbreviation (*input*) *Suc5* **where** *Suc5* $n \equiv \text{Suc } (\text{Suc4 } n)$
abbreviation (*input*) *Suc6* **where** *Suc6* $n \equiv \text{Suc } (\text{Suc5 } n)$
abbreviation (*input*) *Suc7* **where** *Suc7* $n \equiv \text{Suc } (\text{Suc6 } n)$
abbreviation (*input*) *Suc8* **where** *Suc8* $n \equiv \text{Suc } (\text{Suc7 } n)$
abbreviation (*input*) *Suc9* **where** *Suc9* $n \equiv \text{Suc } (\text{Suc8 } n)$

notation *Suc2* (*Suc*²)
notation *Suc3* (*Suc*³)
notation *Suc4* (*Suc*⁴)
notation *Suc5* (*Suc*⁵)
notation *Suc6* (*Suc*⁶)
notation *Suc7* (*Suc*⁷)
notation *Suc8* (*Suc*⁸)
notation *Suc9* (*Suc*⁹)

Beyond 9, the syntax *Sucⁿ* kicks in:

syntax
 $\text{-Suc-tower} :: \text{num-token} \Rightarrow \text{nat} \Rightarrow \text{nat} \quad (\text{Suc}^-)$

parse-translation \langle
let
fun *mk-sucs-aux* 0 $t = t$
 \mid *mk-sucs-aux* $n = \text{mk-sucs-aux } (n - 1) \ (\text{const } \langle \text{Suc} \rangle \$ t)$
fun *mk-sucs* $n = \text{Abs}(n, \text{typ } \langle \text{nat} \rangle, \text{mk-sucs-aux } n \ (\text{Bound } 0))$

fun *Suc-tr* [*Free* (*str*, -)] = *mk-sucs* (*the* (*Int.fromString* *str*))

in [(**syntax-const** $\langle \text{-Suc-tower} \rangle$, *K Suc-tr*)] *end*
 \rangle

print-translation \langle
let
val *digit-consts* =
 $[\text{const-syntax } \langle \text{Suc2} \rangle, \text{const-syntax } \langle \text{Suc3} \rangle, \text{const-syntax } \langle \text{Suc4} \rangle, \text{const-syntax } \langle \text{Suc5} \rangle,$
 $\text{ta} \rangle \text{Suc5} \rangle,$

```

    const-syntax ⟨Suc6⟩, const-syntax ⟨Suc7⟩, const-syntax ⟨Suc8⟩, const-syntax
    tax ⟨Suc9⟩]
    val num-token-T = Simple-Syntax.read-typ num-token
    val T = num-token-T --> HOLogic.natT --> HOLogic.natT
    fun mk-num-token n = Free (Int.toString n, num-token-T)
    fun dest-Suc-tower (Const (const-syntax ⟨Suc⟩, -) $ t) acc =
      dest-Suc-tower t (acc + 1)
    | dest-Suc-tower t acc = (t, acc)

    fun Suc-tr' [t] =
      let
        val (t', n) = dest-Suc-tower t 1
      in
        if n > 9 then
          Const (syntax-const ⟨-Suc-tower⟩, T) $ mk-num-token n $ t'
        else if n > 1 then
          Const (List.nth (digit-consts, n - 2), T) $ t'
        else
          raise Match
      end

    in [(const-syntax ⟨Suc⟩, K Suc-tr')]
  end
>

```

end

```

theory Predicate-Compile-Alternative-Defs
imports Main
begin

```

143 Common constants

```

declare HOL.if-bool-eq-disj[code-pred-inline]

```

```

declare bool-diff-def[code-pred-inline]
declare inf-bool-def[abs-def, code-pred-inline]
declare less-bool-def[abs-def, code-pred-inline]
declare le-bool-def[abs-def, code-pred-inline]

```

```

lemma min-bool-eq [code-pred-inline]: (min :: bool => bool => bool) == (∧)
by (rule eq-reflection) (auto simp add: fun-eq-iff min-def)

```

```

lemma [code-pred-inline]:
  ((A::bool) ≠ (B::bool)) = ((A ∧ ¬ B) ∨ (B ∧ ¬ A))
by fast

```

setup $\langle \text{Predicate-Compile-Data.ignore-consts } [\text{const-name } \langle \text{Let} \rangle] \rangle$

144 Pairs

setup $\langle \text{Predicate-Compile-Data.ignore-consts } [\text{const-name } \langle \text{fst} \rangle, \text{const-name } \langle \text{snd} \rangle, \text{const-name } \langle \text{case-prod} \rangle] \rangle$

145 Filters

setup $\langle \text{Predicate-Compile-Data.ignore-consts } [\text{const-name } \langle \text{Abs-filter} \rangle, \text{const-name } \langle \text{Rep-filter} \rangle] \rangle$

146 Bounded quantifiers

declare *Ball-def*[code-pred-inline]
declare *Bex-def*[code-pred-inline]

147 Operations on Predicates

lemma *Diff*[code-pred-inline]:
 $(A - B) = (\%x. A\ x \wedge \neg B\ x)$
by (*simp add: fun-eq-iff*)

lemma *subset-eq*[code-pred-inline]:
 $(P :: 'a \Rightarrow \text{bool}) < (Q :: 'a \Rightarrow \text{bool}) \equiv ((\exists x. Q\ x \wedge (\neg P\ x)) \wedge (\forall x. P\ x \longrightarrow Q\ x))$
by (*rule eq-reflection*) (*auto simp add: less-fun-def le-fun-def*)

lemma *set-equality*[code-pred-inline]:
 $A = B \longleftrightarrow (\forall x. A\ x \longrightarrow B\ x) \wedge (\forall x. B\ x \longrightarrow A\ x)$
by (*auto simp add: fun-eq-iff*)

148 Setup for Numerals

setup $\langle \text{Predicate-Compile-Data.ignore-consts } [\text{const-name } \langle \text{numeral} \rangle] \rangle$
setup $\langle \text{Predicate-Compile-Data.keep-functions } [\text{const-name } \langle \text{numeral} \rangle] \rangle$
setup $\langle \text{Predicate-Compile-Data.ignore-consts } [\text{const-name } \langle \text{Char} \rangle] \rangle$
setup $\langle \text{Predicate-Compile-Data.keep-functions } [\text{const-name } \langle \text{Char} \rangle] \rangle$

setup $\langle \text{Predicate-Compile-Data.ignore-consts } [\text{const-name } \langle \text{divide} \rangle, \text{const-name } \langle \text{modulo} \rangle, \text{const-name } \langle \text{times} \rangle] \rangle$

149 Arithmetic operations

149.1 Arithmetic on naturals and integers

definition *plus-eq-nat* :: *nat* => *nat* => *nat* => *bool*

where

plus-eq-nat *x y z* = (*x* + *y* = *z*)

definition *minus-eq-nat* :: *nat* => *nat* => *nat* => *bool*

where

minus-eq-nat *x y z* = (*x* - *y* = *z*)

definition *plus-eq-int* :: *int* => *int* => *int* => *bool*

where

plus-eq-int *x y z* = (*x* + *y* = *z*)

definition *minus-eq-int* :: *int* => *int* => *int* => *bool*

where

minus-eq-int *x y z* = (*x* - *y* = *z*)

definition *subtract*

where

[code-unfold]: *subtract* *x y* = *y* - *x*

setup <

let

val Fun = *Predicate-Compile-Aux.Fun*

val Input = *Predicate-Compile-Aux.Input*

val Output = *Predicate-Compile-Aux.Output*

val Bool = *Predicate-Compile-Aux.Bool*

val iio = *Fun* (*Input*, *Fun* (*Input*, *Fun* (*Output*, *Bool*)))

val ioi = *Fun* (*Input*, *Fun* (*Output*, *Fun* (*Input*, *Bool*)))

val oii = *Fun* (*Output*, *Fun* (*Input*, *Fun* (*Input*, *Bool*)))

val ooi = *Fun* (*Output*, *Fun* (*Output*, *Fun* (*Input*, *Bool*)))

val plus-nat = *Core-Data.functional-compilation* **const-name** <plus> *iio*

val minus-nat = *Core-Data.functional-compilation* **const-name** <minus> *iio*

fun subtract-nat *compfun* (*-* : *typ*) =

let

val T = *Predicate-Compile-Aux.mk-monadT* *compfun* **typ** <nat>

in

absdummy typ <nat> (*absdummy typ* <nat>

(*Const* (**const-name** <If>, **typ** <bool> --> *T* --> *T* --> *T*) \$

(**term** <(>) :: *nat* => *nat* => *bool*> \$ *Bound* 1 \$ *Bound* 0) \$

Predicate-Compile-Aux.mk-empty *compfun* **typ** <nat> \$

Predicate-Compile-Aux.mk-single *compfun*

(**term** <(-) :: *nat* => *nat* => *nat*> \$ *Bound* 0 \$ *Bound* 1)))

end

fun enumerate-addups-nat *compfun* (*-* : *typ*) =

absdummy typ <nat> (*Predicate-Compile-Aux.mk-iterate-upto* *compfun* **typ** <nat

*** nat**>

```

    (absdummy typ ⟨natural⟩ (term ⟨Pair :: nat => nat => nat * nat⟩ $
      (term ⟨nat-of-natural⟩ $ Bound 0) $
      (term ⟨(-) :: nat => nat => nat⟩ $ Bound 1 $ (term ⟨nat-of-natural⟩ $
Bound 0))),
    term ⟨0 :: natural⟩, term ⟨natural-of-nat⟩ $ Bound 0))
  fun enumerate-nats compfuns (- : typ) =
    let
      val (single-const, -) = strip-comb (Predicate-Compile-Aux.mk-single compfuns
term ⟨0 :: nat⟩)
      val T = Predicate-Compile-Aux.mk-monadT compfuns typ ⟨nat⟩
    in
      absdummy typ ⟨nat⟩ (absdummy typ ⟨nat⟩
        (Const (const-name ⟨If⟩, typ ⟨bool⟩ --> T --> T --> T) $
          (term ⟨(=) :: nat => nat => bool⟩ $ Bound 0 $ term ⟨0::nat⟩) $
          (Predicate-Compile-Aux.mk-iterate-upto compfuns typ ⟨nat⟩ (term ⟨nat-of-natural⟩,
            term ⟨0::natural⟩, term ⟨natural-of-nat⟩ $ Bound 1)) $
            (single-const $ (term ⟨(+) :: nat => nat => nat⟩ $ Bound 1 $ Bound
0))))))
      end
    in
      Core-Data.force-modes-and-compilations const-name ⟨plus-eq-nat⟩
        [(iio, (plus-nat, false)), (oii, (subtract-nat, false)), (ioi, (subtract-nat, false)),
          (ooi, (enumerate-addups-nat, false))]
      #> Predicate-Compile-Fun.add-function-predicate-translation
        (term ⟨plus :: nat => nat => nat⟩, term ⟨plus-eq-nat⟩)
      #> Core-Data.force-modes-and-compilations const-name ⟨minus-eq-nat⟩
        [(iio, (minus-nat, false)), (oii, (enumerate-nats, false))]
      #> Predicate-Compile-Fun.add-function-predicate-translation
        (term ⟨minus :: nat => nat => nat⟩, term ⟨minus-eq-nat⟩)
      #> Core-Data.force-modes-and-functions const-name ⟨plus-eq-int⟩
        [(iio, (const-name ⟨plus⟩, false)), (ioi, (const-name ⟨subtract⟩, false)),
          (oii, (const-name ⟨subtract⟩, false))]
      #> Predicate-Compile-Fun.add-function-predicate-translation
        (term ⟨plus :: int => int => int⟩, term ⟨plus-eq-int⟩)
      #> Core-Data.force-modes-and-functions const-name ⟨minus-eq-int⟩
        [(iio, (const-name ⟨minus⟩, false)), (oii, (const-name ⟨plus⟩, false)),
          (ioi, (const-name ⟨minus⟩, false))]
      #> Predicate-Compile-Fun.add-function-predicate-translation
        (term ⟨minus :: int => int => int⟩, term ⟨minus-eq-int⟩)
    end
  >

```

149.2 Inductive definitions for ordering on naturals

inductive *less-nat*

where

less-nat 0 (Suc y)

| *less-nat* x y ==> *less-nat* (Suc x) (Suc y)


```

lemma less-nat[code-pred-inline]:
   $x < y = \text{less-nat } x \ y$ 
apply (rule iffI)
apply (induct x arbitrary: y)
apply (case-tac y) apply (auto intro: less-nat.intros)
apply (case-tac y)
apply (auto intro: less-nat.intros)
apply (induct rule: less-nat.induct)
apply auto
done

```

```

inductive less-eq-nat
where
   $\text{less-eq-nat } 0 \ y$ 
|  $\text{less-eq-nat } x \ y ==> \text{less-eq-nat } (\text{Suc } x) \ (\text{Suc } y)$ 

```

```

lemma [code-pred-inline]:
   $x \leq y = \text{less-eq-nat } x \ y$ 
apply (rule iffI)
apply (induct x arbitrary: y)
apply (auto intro: less-eq-nat.intros)
apply (case-tac y) apply (auto intro: less-eq-nat.intros)
apply (induct rule: less-eq-nat.induct)
apply auto done

```

150 Alternative list definitions

150.1 Alternative rules for *length*

```

definition size-list' :: 'a list => nat
where size-list' = size

```

```

lemma size-list'-simps:
   $\text{size-list'} [] = 0$ 
   $\text{size-list'} (x \# xs) = \text{Suc } (\text{size-list'} xs)$ 
by (auto simp add: size-list'-def)

```

```

declare size-list'-simps[code-pred-def]
declare size-list'-def[symmetric, code-pred-inline]

```

150.2 Alternative rules for *list-all2*

```

lemma list-all2-NilI [code-pred-intro]:  $\text{list-all2 } P \ [] \ []$ 
by auto

```

```

lemma list-all2-ConsI [code-pred-intro]:  $\text{list-all2 } P \ xs \ ys ==> P \ x \ y ==> \text{list-all2}$ 
 $P \ (x \# xs) \ (y \# ys)$ 
by auto

```

```

code-pred [skip-proof] list-all2
proof –
  case list-all2
  from this show thesis
  apply –
    apply (case-tac xb)
    apply (case-tac xc)
    apply auto
    apply (case-tac xc)
    apply auto
  done
qed

```

150.3 Alternative rules for membership in lists

```

lemma in-set-member [code-pred-inline]:
   $x \in \text{set } xs \iff \text{List.member } xs \ x$ 
by simp

```

```

lemma member-intros [code-pred-intro]:
  List.member (x#xs) x
  List.member xs x  $\implies$  List.member (y#xs) x
by simp-all

```

```

code-pred List.member
by(auto simp add: elim: list.set-cases)

```

```

code-identifier constant member-i-i
   $\rightarrow$  (SML) List.member-i-i
  and (OCaml) List.member-i-i
  and (Haskell) List.member-i-i
  and (Scala) List.member-i-i

```

```

code-identifier constant member-i-o
   $\rightarrow$  (SML) List.member-i-o
  and (OCaml) List.member-i-o
  and (Haskell) List.member-i-o
  and (Scala) List.member-i-o

```

151 Setup for String.literal

```

setup  $\langle$  Predicate-Compile-Data.ignore-consts [const-name  $\langle$ String.Literal $\rangle$  $\rangle$ 

```

152 Simplification rules for optimisation

```

lemma [code-pred-simp]:  $\neg \text{False} == \text{True}$ 
by auto

```

```
lemma [code-pred-simp]:  $\neg \text{True} == \text{False}$ 
by auto
```

```
lemma less-nat-k-0 [code-pred-simp]:  $\text{less-nat } k \ 0 == \text{False}$ 
unfolding less-nat[symmetric] by auto
```

```
end
```

153 A Prototype of Quickcheck based on the Predicate Compiler

```
theory Predicate-Compile-Quickcheck
imports Predicate-Compile-Alternative-Defs
begin
```

```
ML-file <../Tools/Predicate-Compile/predicate-compile-quickcheck.ML>
```

```
end
```

154 TFL: recursive function definitions

```
theory Old-Recdef
imports Main
keywords
  recdef :: thy-defn and
  permissive congs hints
begin
```

154.1 Lemmas for TFL

```
lemma tfl-wf-induct:  $\forall R. \text{wf } R \longrightarrow$ 
 $(\forall P. (\forall x. (\forall y. (y,x) \in R \longrightarrow P y) \longrightarrow P x) \longrightarrow (\forall x. P x))$ 
apply clarify
apply (rule-tac  $r = R$  and  $P = P$  and  $a = x$  in wf-induct, assumption, blast)
done
```

```
lemma tfl-cut-def:  $\text{cut } f \ r \ x \equiv (\lambda y. \text{if } (y,x) \in r \text{ then } f y \text{ else undefined})$ 
unfolding cut-def .
```

```
lemma tfl-cut-apply:  $\forall f R. (x,a) \in R \longrightarrow (\text{cut } f \ R \ a)(x) = f(x)$ 
apply clarify
apply (rule cut-apply, assumption)
done
```

```
lemma tfl-wfrec:
 $\forall M R f. (f = \text{wfrec } R \ M) \longrightarrow \text{wf } R \longrightarrow (\forall x. f x = M (\text{cut } f \ R \ x) \ x)$ 
apply clarify
apply (erule wfrec)
```

done

lemma *tfl-eq-True*: $(x = \text{True}) \longrightarrow x$
by *blast*

lemma *tfl-rev-eq-mp*: $(x = y) \longrightarrow y \longrightarrow x$
by *blast*

lemma *tfl-simp-thm*: $(x \longrightarrow y) \longrightarrow (x = x') \longrightarrow (x' \longrightarrow y)$
by *blast*

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = \text{True}$
by *blast*

lemma *tfl-imp-trans*: $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$
by *blast*

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
by *simp*

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$
by *blast*

lemma *tfl-exE*: $\exists x. P\ x \Longrightarrow \forall x. P\ x \longrightarrow Q \Longrightarrow Q$
by *blast*

ML-file $\langle \text{old-recdef.ML} \rangle$

154.2 Rule setup

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong *let-cong* *image-cong* *INF-cong* *SUP-cong* *bex-cong* *ball-cong* *imp-cong*
map-cong *filter-cong* *takeWhile-cong* *dropWhile-cong* *foldl-cong* *foldr-cong*

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-measures
wf-pred-nat

```

wf-same-fst
wf-on-bot

```

```
end
```

155 Program extraction from proofs involving datatypes and inductive predicates

```

theory Realizers
imports Main
begin

```

```

ML-file <~~/src/HOL/Tools/datatype-realizer.ML>
ML-file <~~/src/HOL/Tools/inductive-realizer.ML>

```

```
end
```

156 Refute

```

theory Refute
imports Main
keywords
  refute :: diag and
  refute-params :: thy-decl
begin

```

```
ML-file <refute.ML>
```

```

refute-params
[itself = 1,
 minsize = 1,
 maxsize = 8,
 maxvars = 10000,
 maxtime = 60,
 satsolver = auto,
 no-assms = false]

```

```

(* ----- *)
(* REFUTE                                     *)
(*                                           *)
(* We use a SAT solver to search for a (finite) model that refutes a given *)
(* HOL formula.                             *)
(* ----- *)

(* ----- *)
(* NOTE                                     *)
(*                                           *)
(*                                           *)

```

```

(* I strongly recommend that you install a stand-alone SAT solver if you *)
(* want to use 'refute'. For details see 'HOL/Tools/sat_solver.ML'. If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME' *)
(* in 'etc/settings'. *)
(* ----- *)

(* ----- *)
(* USAGE *)
(* *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples. The supported *)
(* parameters are explained below. *)
(* ----- *)

(* ----- *)
(* CURRENT LIMITATIONS *)
(* *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and *)
(* inductively defined sets. Constants are unfolded automatically, and sort *)
(* axioms are added as well. Other, user-asserted axioms however are *)
(* ignored. Inductive datatypes and recursive functions are supported, but *)
(* may lead to spurious countermodels. *)
(* *)
(* The (space) complexity of the algorithm is non-elementary. *)
(* *)
(* Schematic type variables are not supported. *)
(* ----- *)

(* ----- *)
(* PARAMETERS *)
(* *)
(* The following global parameters are currently supported (and required, *)
(* except for "expect"): *)
(* *)
(* Name          Type      Description *)
(* *)
(* "minsize"      int       Only search for models with size at least *)
(*                  'minsize'. *)
(* "maxsize"      int       If >0, only search for models with size at most *)
(*                  'maxsize'. *)
(* "maxvars"      int       If >0, use at most 'maxvars' boolean variables *)
(*                  when transforming the term into a propositional *)
(*                  formula. *)
(* "maxtime"      int       If >0, terminate after at most 'maxtime' seconds. *)
(*                  This value is ignored under some ML compilers. *)
(* "satsolver"    string    Name of the SAT solver to be used. *)
(* "no_assms"     bool      If "true", assumptions in structured proofs are *)
(*                  not considered. *)

```

```

(* "expect"      string Expected result ("genuine", "potential", "none", or *)
(*              "unknown"). *)
(* *)
(* The size of particular types can be specified in the form type=size *)
(* (where 'type' is a string, and 'size' is an int). Examples: *)
(* "'a'=1 *)
(* "List.list "=2 *)
(* ----- *)

(* ----- *)
(* FILES *)
(* *)
(* HOL/Tools/prop_logic.ML      Propositional logic *)
(* HOL/Tools/sat_solver.ML      SAT solvers *)
(* HOL/Tools/refute.ML          Translation HOL -> propositional logic and *)
(*                               Boolean assignment -> HOL model *)
(* HOL/Refute.thy               This file: loads the ML files, basic setup, *)
(*                               documentation *)
(* HOL/SAT.thy                  Sets default parameters *)
(* HOL/ex/Refute_Examples.thy   Examples *)
(* ----- *)

end

```

References

- [1] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009, 2010.
- [2] D. Leijen. Division and modulus for computer scientists. 2001.
- [3] A. Lochbihler and P. Stoop. Lazy algebraic types in Isabelle/HOL. In *Isabelle Workshop 2018*, 2018.
- [4] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.