

Isabelle/CTT — Constructive Type Theory with extensional equality and without universes

Larry Paulson

January 18, 2026

Contents

1	Constructive Type Theory: axiomatic basis	1
1.1	Tactics and derived rules for Constructive Type Theory . . .	6
1.2	Tactics for type checking	7
1.3	Simplification	7
1.4	The elimination rules for fst/snd	8
2	The two-element type (booleans and conditionals)	8
2.1	Derivation of rules for the type <i>Bool</i>	8
3	Elementary arithmetic	9
3.1	Arithmetic operators and their definitions	9
3.2	Proofs about elementary arithmetic: addition, multiplication, etc.	9
3.2.1	Addition	9
3.2.2	Multiplication	10
3.2.3	Difference	10
3.3	Simplification	10
3.4	Addition	11
3.5	Multiplication	11
3.6	Difference	12
3.7	Absolute difference	12
3.8	Remainder and Quotient	13
4	Easy examples: type checking and type deduction	14
4.1	Single-step proofs: verifying that a type is well-formed	14
4.2	Multi-step proofs: Type inference	14
5	Examples with elimination rules	15
6	Equality reasoning by rewriting	18

```

theory CTT
imports Pure
begin

```

1 Constructive Type Theory: axiomatic basis

$\langle ML \rangle$

```

typeddecl i
typeddecl t
typeddecl o

```

consts

— Judgments

```

Type    :: t ⇒ prop      (⟨⟨notation=⟨postfix Type⟩- type⟩ [10] 5)
Eqtype  :: [t,t]⇒prop    (⟨⟨notation=⟨infix Eqtype⟩- =/ -⟩ [10,10] 5)
Elem     :: [i, t]⇒prop   (⟨⟨notation=⟨infix Elem⟩- /: -⟩ [10,10] 5)
Egelem   :: [i,i,t]⇒prop  (⟨⟨notation=⟨mixfix Egelem⟩- =/ - :/ -⟩ [10,10,10]
5)

```

```

Reduce   :: [i,i]⇒prop    (⟨Reduce[-,-]⟩)

```

— Types for truth values

```

F        :: t
T        :: t           — F is empty, T contains one element
contr    :: i⇒i
tt       :: i

```

— Natural numbers

```

N        :: t
Zero     :: i           (⟨0⟩)
succ     :: i⇒i
rec      :: [i, i, [i,i]⇒i] ⇒ i

```

— Binary sum

```

Plus     :: [t,t]⇒t      (infixr ⟨+⟩ 40)
inl      :: i⇒i
inr      :: i⇒i
when     :: [i, i⇒i, i⇒i]⇒i

```

— General sum and binary product

```

Sum       :: [t, i⇒t]⇒t
pair      :: [i,i]⇒i      (⟨⟨indent=1 notation=⟨mixfix pair⟩- <-,->⟩)
fst       :: i⇒i
snd       :: i⇒i
split     :: [i, [i,i]⇒i] ⇒ i

```

— General product and function space

```

Prod      :: [t, i⇒t]⇒t
lambda    :: (i ⇒ i) ⇒ i  (binder ⟨λ⟩ 10)
app       :: [i,i]⇒i      (infixl ⟨'⟩ 60)

```

— Equality type

$Eq \quad :: [t, i, i] \Rightarrow t$
 $eq \quad :: i$

Some inexplicable syntactic dependencies; in particular, "0" must be introduced after the judgment forms.

syntax

$-PROD \quad :: [idt, t, t] \Rightarrow t \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \prod \rangle \prod \text{ :-./ -} \rangle 10)$
 $-SUM \quad :: [idt, t, t] \Rightarrow t \quad (\langle (\langle indent=3 \text{ notation}=\langle binder \sum \rangle \sum \text{ :-./ -} \rangle 10)$

syntax-consts

$-PROD \Leftarrow Prod$ **and**
 $-SUM \Leftarrow Sum$

translations

$\prod x:A. B \Leftarrow CONST Prod(A, \lambda x. B)$
 $\sum x:A. B \Leftarrow CONST Sum(A, \lambda x. B)$

abbreviation $Arrow \quad :: [t, t] \Rightarrow t$ (**infixr** $\langle \longrightarrow \rangle$ 30)
where $A \longrightarrow B \equiv \prod \text{:-}A. B$

abbreviation $Times \quad :: [t, t] \Rightarrow t$ (**infixr** $\langle \times \rangle$ 50)
where $A \times B \equiv \sum \text{:-}A. B$

Reduction: a weaker notion than equality; a hack for simplification. *Reduce* $[a, b]$ means either that $a = b : A$ for some A or else that a and b are textually identical.

Does not verify $a:A!$ Sound because only *trans-red* uses a *Reduce* premise. No new theorems can be proved about the standard judgments.

axiomatization

where

$refl\text{-}red: \bigwedge a. Reduce[a, a]$ **and**
 $red\text{-}if\text{-}equal: \bigwedge a \ b \ A. a = b : A \Longrightarrow Reduce[a, b]$ **and**
 $trans\text{-}red: \bigwedge a \ b \ c \ A. \llbracket a = b : A; Reduce[b, c] \rrbracket \Longrightarrow a = c : A$ **and**

— Reflexivity

$refl\text{-}type: \bigwedge A. A \text{ type} \Longrightarrow A = A$ **and**
 $refl\text{-}elem: \bigwedge a \ A. a : A \Longrightarrow a = a : A$ **and**

— Symmetry

$sym\text{-}type: \bigwedge A \ B. A = B \Longrightarrow B = A$ **and**
 $sym\text{-}elem: \bigwedge a \ b \ A. a = b : A \Longrightarrow b = a : A$ **and**

— Transitivity

$trans\text{-}type: \bigwedge A \ B \ C. \llbracket A = B; B = C \rrbracket \Longrightarrow A = C$ **and**
 $trans\text{-}elem: \bigwedge a \ b \ c \ A. \llbracket a = b : A; b = c : A \rrbracket \Longrightarrow a = c : A$ **and**

$equal\text{-}types: \bigwedge a \ A \ B. \llbracket a : A; A = B \rrbracket \Longrightarrow a : B$ **and**

equal-typesL: $\bigwedge a\ b\ A\ B. \llbracket a = b : A; A = B \rrbracket \Longrightarrow a = b : B$ **and**

— Substitution

subst-type: $\bigwedge a\ A\ B. \llbracket a : A; \bigwedge z. z:A \Longrightarrow B(z) \text{ type} \rrbracket \Longrightarrow B(a) \text{ type}$ **and**
subst-typeL: $\bigwedge a\ c\ A\ B\ D. \llbracket a = c : A; \bigwedge z. z:A \Longrightarrow B(z) = D(z) \rrbracket \Longrightarrow B(a) = D(c)$ **and**

subst-elim: $\bigwedge a\ b\ A\ B. \llbracket a : A; \bigwedge z. z:A \Longrightarrow b(z):B(z) \rrbracket \Longrightarrow b(a):B(a)$ **and**
subst-elimL:
 $\bigwedge a\ b\ c\ d\ A\ B. \llbracket a = c : A; \bigwedge z. z:A \Longrightarrow b(z)=d(z) : B(z) \rrbracket \Longrightarrow b(a)=d(c) : B(a)$ **and**

— The type N – natural numbers

NF: N type **and**

NI0: $0 : N$ **and**

NI-succ: $\bigwedge a. a : N \Longrightarrow \text{succ}(a) : N$ **and**

NI-succL: $\bigwedge a\ b. a = b : N \Longrightarrow \text{succ}(a) = \text{succ}(b) : N$ **and**

NE:

$\bigwedge p\ a\ b\ C. \llbracket p : N; a : C(0); \bigwedge u\ v. \llbracket u : N; v : C(u) \rrbracket \Longrightarrow b(u,v) : C(\text{succ}(u)) \rrbracket$
 $\Longrightarrow \text{rec}(p, a, \lambda u\ v. b(u,v)) : C(p)$ **and**

NEL:

$\bigwedge p\ q\ a\ b\ c\ d\ C. \llbracket p = q : N; a = c : C(0);$
 $\bigwedge u\ v. \llbracket u : N; v : C(u) \rrbracket \Longrightarrow b(u,v) = d(u,v) : C(\text{succ}(u)) \rrbracket$
 $\Longrightarrow \text{rec}(p, a, \lambda u\ v. b(u,v)) = \text{rec}(q, c, d) : C(p)$ **and**

NC0:

$\bigwedge a\ b\ C. \llbracket a : C(0); \bigwedge u\ v. \llbracket u : N; v : C(u) \rrbracket \Longrightarrow b(u,v) : C(\text{succ}(u)) \rrbracket$
 $\Longrightarrow \text{rec}(0, a, \lambda u\ v. b(u,v)) = a : C(0)$ **and**

NC-succ:

$\bigwedge p\ a\ b\ C. \llbracket p : N; a : C(0); \bigwedge u\ v. \llbracket u : N; v : C(u) \rrbracket \Longrightarrow b(u,v) : C(\text{succ}(u)) \rrbracket \Longrightarrow$
 $\text{rec}(\text{succ}(p), a, \lambda u\ v. b(u,v)) = b(p, \text{rec}(p, a, \lambda u\ v. b(u,v))) : C(\text{succ}(p))$ **and**

— The fourth Peano axiom. See page 91 of Martin-Löf's book.

zero-ne-succ: $\bigwedge a. \llbracket a : N; 0 = \text{succ}(a) : N \rrbracket \Longrightarrow 0 : F$ **and**

— The Product of a family of types

ProdF: $\bigwedge A\ B. \llbracket A \text{ type}; \bigwedge x. x:A \Longrightarrow B(x) \text{ type} \rrbracket \Longrightarrow \prod x:A. B(x) \text{ type}$ **and**

ProdFL:

$\bigwedge A\ B\ C\ D. \llbracket A = C; \bigwedge x. x:A \Longrightarrow B(x) = D(x) \rrbracket \Longrightarrow \prod x:A. B(x) = \prod x:C. D(x)$ **and**

ProdI:

$\bigwedge b \ A \ B. \llbracket A \text{ type}; \bigwedge x. x:A \implies b(x):B(x) \rrbracket \implies \lambda x. b(x) : \prod x:A. B(x) \text{ and}$

ProdIL: $\bigwedge b \ c \ A \ B. \llbracket A \text{ type}; \bigwedge x. x:A \implies b(x) = c(x) : B(x) \rrbracket \implies$

$\lambda x. b(x) = \lambda x. c(x) : \prod x:A. B(x) \text{ and}$

ProdE: $\bigwedge p \ a \ A \ B. \llbracket p : \prod x:A. B(x); a : A \rrbracket \implies p'a : B(a) \text{ and}$

ProdEL: $\bigwedge p \ q \ a \ b \ A \ B. \llbracket p = q : \prod x:A. B(x); a = b : A \rrbracket \implies p'a = q'b : B(a)$
and

ProdC: $\bigwedge a \ b \ A \ B. \llbracket a : A; \bigwedge x. x:A \implies b(x) : B(x) \rrbracket \implies (\lambda x. b(x)) ' a = b(a) : B(a) \text{ and}$

ProdC2: $\bigwedge p \ A \ B. p : \prod x:A. B(x) \implies (\lambda x. p'x) = p : \prod x:A. B(x) \text{ and}$

— The Sum of a family of types

SumF: $\bigwedge A \ B. \llbracket A \text{ type}; \bigwedge x. x:A \implies B(x) \text{ type} \rrbracket \implies \sum x:A. B(x) \text{ type and}$

SumFL: $\bigwedge A \ B \ C \ D. \llbracket A = C; \bigwedge x. x:A \implies B(x) = D(x) \rrbracket \implies \sum x:A. B(x) = \sum x:C. D(x) \text{ and}$

SumI: $\bigwedge a \ b \ A \ B. \llbracket a : A; b : B(a) \rrbracket \implies \langle a, b \rangle : \sum x:A. B(x) \text{ and}$

SumIL: $\bigwedge a \ b \ c \ d \ A \ B. \llbracket a = c : A; b = d : B(a) \rrbracket \implies \langle a, b \rangle = \langle c, d \rangle : \sum x:A. B(x) \text{ and}$

SumE: $\bigwedge p \ c \ A \ B \ C. \llbracket p : \sum x:A. B(x); \bigwedge x \ y. \llbracket x:A; y:B(x) \rrbracket \implies c(x,y) : C(\langle x, y \rangle) \rrbracket \implies \text{split}(p, \lambda x \ y. c(x,y)) : C(p) \text{ and}$

SumEL: $\bigwedge p \ q \ c \ d \ A \ B \ C. \llbracket p = q : \sum x:A. B(x);$

$\bigwedge x \ y. \llbracket x:A; y:B(x) \rrbracket \implies c(x,y) = d(x,y) : C(\langle x, y \rangle) \rrbracket$

$\implies \text{split}(p, \lambda x \ y. c(x,y)) = \text{split}(q, \lambda x \ y. d(x,y)) : C(p) \text{ and}$

SumC: $\bigwedge a \ b \ c \ A \ B \ C. \llbracket a : A; b : B(a); \bigwedge x \ y. \llbracket x:A; y:B(x) \rrbracket \implies c(x,y) : C(\langle x, y \rangle) \rrbracket$

$\implies \text{split}(\langle a, b \rangle, \lambda x \ y. c(x,y)) = c(a,b) : C(\langle a, b \rangle) \text{ and}$

fst-def: $\bigwedge a. \text{fst}(a) \equiv \text{split}(a, \lambda x \ y. x) \text{ and}$

snd-def: $\bigwedge a. \text{snd}(a) \equiv \text{split}(a, \lambda x \ y. y) \text{ and}$

— The sum of two types

PlusF: $\bigwedge A \ B. \llbracket A \text{ type}; B \text{ type} \rrbracket \implies A+B \text{ type and}$

PlusFL: $\bigwedge A \ B \ C \ D. \llbracket A = C; B = D \rrbracket \implies A+B = C+D \text{ and}$

PlusI-inl: $\bigwedge a \ A \ B. \llbracket a : A; B \text{ type} \rrbracket \implies \text{inl}(a) : A+B \text{ and}$

PlusI-inlL: $\bigwedge a \ c \ A \ B. \llbracket a = c : A; B \text{ type} \rrbracket \implies \text{inl}(a) = \text{inl}(c) : A+B \text{ and}$

PlusI-inr: $\bigwedge b A B. \llbracket A \text{ type}; b : B \rrbracket \implies \text{inr}(b) : A+B$ **and**
PlusI-inrL: $\bigwedge b d A B. \llbracket A \text{ type}; b = d : B \rrbracket \implies \text{inr}(b) = \text{inr}(d) : A+B$ **and**

PlusE:

$\bigwedge p c d A B C. \llbracket p : A+B;$
 $\bigwedge x. x:A \implies c(x) : C(\text{inl}(x));$
 $\bigwedge y. y:B \implies d(y) : C(\text{inr}(y)) \rrbracket \implies \text{when}(p, \lambda x. c(x), \lambda y. d(y)) : C(p)$ **and**

PlusEL:

$\bigwedge p q c d e f A B C. \llbracket p = q : A+B;$
 $\bigwedge x. x: A \implies c(x) = e(x) : C(\text{inl}(x));$
 $\bigwedge y. y: B \implies d(y) = f(y) : C(\text{inr}(y)) \rrbracket$
 $\implies \text{when}(p, \lambda x. c(x), \lambda y. d(y)) = \text{when}(q, \lambda x. e(x), \lambda y. f(y)) : C(p)$ **and**

PlusC-inl:

$\bigwedge a c d A B C. \llbracket a : A;$
 $\bigwedge x. x:A \implies c(x) : C(\text{inl}(x));$
 $\bigwedge y. y:B \implies d(y) : C(\text{inr}(y)) \rrbracket$
 $\implies \text{when}(\text{inl}(a), \lambda x. c(x), \lambda y. d(y)) = c(a) : C(\text{inl}(a))$ **and**

PlusC-inr:

$\bigwedge b c d A B C. \llbracket b : B;$
 $\bigwedge x. x:A \implies c(x) : C(\text{inl}(x));$
 $\bigwedge y. y:B \implies d(y) : C(\text{inr}(y)) \rrbracket$
 $\implies \text{when}(\text{inr}(b), \lambda x. c(x), \lambda y. d(y)) = d(b) : C(\text{inr}(b))$ **and**

— The type *Eq*

EqF: $\bigwedge a b A. \llbracket A \text{ type}; a : A; b : A \rrbracket \implies \text{Eq}(A,a,b) \text{ type}$ **and**

EqFL: $\bigwedge a b c d A B. \llbracket A = B; a = c : A; b = d : A \rrbracket \implies \text{Eq}(A,a,b) = \text{Eq}(B,c,d)$ **and**

EqI: $\bigwedge a b A. a = b : A \implies \text{eq} : \text{Eq}(A,a,b)$ **and**

EqE: $\bigwedge p a b A. p : \text{Eq}(A,a,b) \implies a = b : A$ **and**

— By equality of types, can prove $C(p)$ from $C(\text{eq})$, an elimination rule

EqC: $\bigwedge p a b A. p : \text{Eq}(A,a,b) \implies p = \text{eq} : \text{Eq}(A,a,b)$ **and**

— The type *F*

FF: $F \text{ type}$ **and**

FE: $\bigwedge p C. \llbracket p : F; C \text{ type} \rrbracket \implies \text{contr}(p) : C$ **and**

FEL: $\bigwedge p q C. \llbracket p = q : F; C \text{ type} \rrbracket \implies \text{contr}(p) = \text{contr}(q) : C$ **and**

— The type *T*

— Martin-Löf's book (page 68) discusses elimination and computation. Elimination can be derived by computation and equality of types, but with an extra

premise $C(x)$ type $x:T$. Also computation can be derived from elimination.

TF: T type **and**
TI: $tt : T$ **and**
TE: $\bigwedge p \ c \ C. \llbracket p : T; c : C(tt) \rrbracket \implies c : C(p)$ **and**
TEL: $\bigwedge p \ q \ c \ d \ C. \llbracket p = q : T; c = d : C(tt) \rrbracket \implies c = d : C(p)$ **and**
TC: $\bigwedge p. p : T \implies p = tt : T$

1.1 Tactics and derived rules for Constructive Type Theory

Formation rules.

lemmas *form-rls* = *NF ProdF SumF PlusF EqF FF TF*
and *formL-rls* = *ProdFL SumFL PlusFL EqFL*

Introduction rules. OMITTED:

- *EqI*, because its premise is an *equelem*, not an *elem*.

lemmas *intr-rls* = *NI0 NI-succ ProdI SumI PlusI-inl PlusI-inr TI*
and *intrL-rls* = *NI-succL ProdIL SumIL PlusI-inlL PlusI-inrL*

Elimination rules. OMITTED:

- *EqE*, because its conclusion is an *equelem*, not an *elem*
- *TE*, because it does not involve a constructor.

lemmas *elim-rls* = *NE ProdE SumE PlusE FE*
and *elimL-rls* = *NEL ProdEL SumEL PlusEL FEL*

OMITTED: *eqC* are *TC* because they make rewriting loop: $p = un = un = \dots$

lemmas *comp-rls* = *NC0 NC-succ ProdC SumC PlusC-inl PlusC-inr*

Rules with conclusion $a:A$, an *elem* judgment.

lemmas *element-rls* = *intr-rls elim-rls*

Definitions are (meta)equality axioms.

lemmas *basic-defs* = *fst-def snd-def*

Compare with standard version: B is applied to UNSIMPLIFIED expression!

lemma *SumIL2*: $\llbracket c = a : A; d = b : B(a) \rrbracket \implies \langle c, d \rangle = \langle a, b \rangle : \text{Sum}(A, B)$
 $\langle \text{proof} \rangle$

lemmas *intrL2-rls* = *NI-succL ProdIL SumIL2 PlusI-inlL PlusI-inrL*

Exploit $p:Prod(A,B)$ to create the assumption $z:B(a)$. A more natural form of product elimination.

lemma *subst-prodE*:
assumes $p: Prod(A,B)$
and $a: A$
and $\bigwedge z. z: B(a) \implies c(z): C(z)$
shows $c(p'a): C(p'a)$
 $\langle proof \rangle$

1.2 Tactics for type checking

$\langle ML \rangle$

For simplification: type formation and checking, but no equalities between terms.

lemmas *routine-rls* = *form-rls formL-rls refl-type element-rls*

$\langle ML \rangle$

1.3 Simplification

To simplify the type in a goal.

lemma *replace-type*: $\llbracket B = A; a : A \rrbracket \implies a : B$
 $\langle proof \rangle$

Simplify the parameter of a unary type operator.

lemma *subst-eqtyparg*:
assumes $1: a=c : A$
and $2: \bigwedge z. z:A \implies B(z) \text{ type}$
shows $B(a) = B(c)$
 $\langle proof \rangle$

Simplification rules for Constructive Type Theory.

lemmas *reduction-rls* = *comp-rls [THEN trans-elem]*

$\langle ML \rangle$

1.4 The elimination rules for fst/snd

lemma *SumE-fst*: $p : Sum(A,B) \implies fst(p) : A$
 $\langle proof \rangle$

The first premise must be $p:Sum(A,B)!!$.

lemma *SumE-snd*:
assumes *major*: $p: Sum(A,B)$
and $A \text{ type}$
and $\bigwedge x. x:A \implies B(x) \text{ type}$
shows $snd(p) : B(fst(p))$
 $\langle proof \rangle$

2 The two-element type (booleans and conditionals)

definition $Bool :: t$
where $Bool \equiv T + T$

definition $true :: i$
where $true \equiv \text{inl}(tt)$

definition $false :: i$
where $false \equiv \text{inr}(tt)$

definition $cond :: [i, i, i] \Rightarrow i$
where $cond(a, b, c) \equiv \text{when}(a, \lambda -. b, \lambda -. c)$

lemmas $\text{bool-defs} = \text{Bool-def true-def false-def cond-def}$

2.1 Derivation of rules for the type $Bool$

Formation rule.

lemma $\text{boolF}: Bool \text{ type}$
 $\langle \text{proof} \rangle$

Introduction rules for $true, false$.

lemma $\text{boolI-true}: true : Bool$
 $\langle \text{proof} \rangle$

lemma $\text{boolI-false}: false : Bool$
 $\langle \text{proof} \rangle$

Elimination rule: typing of $cond$.

lemma $\text{boolE}: \llbracket p : Bool; a : C(true); b : C(false) \rrbracket \Longrightarrow cond(p, a, b) : C(p)$
 $\langle \text{proof} \rangle$

lemma $\text{boolEL}: \llbracket p = q : Bool; a = c : C(true); b = d : C(false) \rrbracket$
 $\Longrightarrow cond(p, a, b) = cond(q, c, d) : C(p)$
 $\langle \text{proof} \rangle$

Computation rules for $true, false$.

lemma $\text{boolC-true}: \llbracket a : C(true); b : C(false) \rrbracket \Longrightarrow cond(true, a, b) = a : C(true)$
 $\langle \text{proof} \rangle$

lemma $\text{boolC-false}: \llbracket a : C(true); b : C(false) \rrbracket \Longrightarrow cond(false, a, b) = b : C(false)$
 $\langle \text{proof} \rangle$

3 Elementary arithmetic

3.1 Arithmetic operators and their definitions

definition $add :: [i, i] \Rightarrow i$ (**infixr** $\langle \# + \rangle$ 65)
where $a \# + b \equiv rec(a, b, \lambda u v. succ(v))$

definition $diff :: [i, i] \Rightarrow i$ (**infixr** $\langle - \rangle$ 65)
where $a - b \equiv rec(b, a, \lambda u v. rec(v, 0, \lambda x y. x))$

definition $absdiff :: [i, i] \Rightarrow i$ (**infixr** $\langle |-| \rangle$ 65)
where $a |-| b \equiv (a - b) \# + (b - a)$

definition $mult :: [i, i] \Rightarrow i$ (**infixr** $\langle \# * \rangle$ 70)
where $a \# * b \equiv rec(a, 0, \lambda u v. b \# + v)$

definition $mod :: [i, i] \Rightarrow i$ (**infixr** $\langle mod \rangle$ 70)
where $a mod b \equiv rec(a, 0, \lambda u v. rec(succ(v) |-| b, 0, \lambda x y. succ(v)))$

definition $div :: [i, i] \Rightarrow i$ (**infixr** $\langle div \rangle$ 70)
where $a div b \equiv rec(a, 0, \lambda u v. rec(succ(u) mod b, succ(v), \lambda x y. v))$

lemmas $arith-defs = add-def\ diff-def\ absdiff-def\ mult-def\ mod-def\ div-def$

3.2 Proofs about elementary arithmetic: addition, multiplication, etc.

3.2.1 Addition

Typing of add : short and long versions.

lemma $add\text{-}typing: \llbracket a:N; b:N \rrbracket \Longrightarrow a \# + b : N$
 $\langle proof \rangle$

lemma $add\text{-}typingL: \llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \# + b = c \# + d : N$
 $\langle proof \rangle$

Computation for add : 0 and successor cases.

lemma $addC0: b:N \Longrightarrow 0 \# + b = b : N$
 $\langle proof \rangle$

lemma $addC\text{-}succ: \llbracket a:N; b:N \rrbracket \Longrightarrow succ(a) \# + b = succ(a \# + b) : N$
 $\langle proof \rangle$

3.2.2 Multiplication

Typing of $mult$: short and long versions.

lemma $mult\text{-}typing: \llbracket a:N; b:N \rrbracket \Longrightarrow a \# * b : N$
 $\langle proof \rangle$

lemma *mult-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \#* b = c \#* d : N$
 $\langle proof \rangle$

Computation for *mult*: 0 and successor cases.

lemma *multC0*: $b:N \Longrightarrow 0 \#* b = 0 : N$
 $\langle proof \rangle$

lemma *multC-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow succ(a) \#* b = b \#+ (a \#* b) : N$
 $\langle proof \rangle$

3.2.3 Difference

Typing of difference.

lemma *diff-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a - b : N$
 $\langle proof \rangle$

lemma *diff-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a - b = c - d : N$
 $\langle proof \rangle$

Computation for difference: 0 and successor cases.

lemma *diffC0*: $a:N \Longrightarrow a - 0 = a : N$
 $\langle proof \rangle$

Note: $rec(a, 0, \lambda z w.z)$ is $pred(a)$.

lemma *diff-0-eq-0*: $b:N \Longrightarrow 0 - b = 0 : N$
 $\langle proof \rangle$

Essential to simplify FIRST!! (Else we get a critical pair) $succ(a) - succ(b)$ rewrites to $pred(succ(a) - b)$.

lemma *diff-succ-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow succ(a) - succ(b) = a - b : N$
 $\langle proof \rangle$

3.3 Simplification

lemmas *arith-typing-rls* = *add-typing mult-typing diff-typing*
and *arith-congr-rls* = *add-typingL mult-typingL diff-typingL*

lemmas *congr-rls* = *arith-congr-rls intrL2-rls elimL-rls*

lemmas *arithC-rls* =
addC0 addC-succ
multC0 multC-succ
diffC0 diff-0-eq-0 diff-succ-succ

$\langle ML \rangle$

3.4 Addition

Associative law for addition.

lemma *add-assoc*: $\llbracket a:N; b:N; c:N \rrbracket \implies (a \# + b) \# + c = a \# + (b \# + c) : N$
<proof>

Commutative law for addition. Can be proved using three inductions. Must simplify after first induction! Orientation of rewrites is delicate.

lemma *add-commute*: $\llbracket a:N; b:N \rrbracket \implies a \# + b = b \# + a : N$
<proof>

3.5 Multiplication

Right annihilation in product.

lemma *mult-0-right*: $a:N \implies a \# * 0 = 0 : N$
<proof>

Right successor law for multiplication.

lemma *mult-succ-right*: $\llbracket a:N; b:N \rrbracket \implies a \# * \text{succ}(b) = a \# + (a \# * b) : N$
<proof>

Commutative law for multiplication.

lemma *mult-commute*: $\llbracket a:N; b:N \rrbracket \implies a \# * b = b \# * a : N$
<proof>

Addition distributes over multiplication.

lemma *add-mult-distrib*: $\llbracket a:N; b:N; c:N \rrbracket \implies (a \# + b) \# * c = (a \# * c) \# + (b \# * c) : N$
<proof>

Associative law for multiplication.

lemma *mult-assoc*: $\llbracket a:N; b:N; c:N \rrbracket \implies (a \# * b) \# * c = a \# * (b \# * c) : N$
<proof>

3.6 Difference

Difference on natural numbers, without negative numbers

- $a - b = 0$ iff $a \leq b$
- $a - b = \text{succ}(c)$ iff $a > b$

lemma *diff-self-eq-0*: $a:N \implies a - a = 0 : N$
<proof>

lemma *add-0-right*: $\llbracket c : N; 0 : N; c : N \rrbracket \Longrightarrow c \# + 0 = c : N$
 $\langle proof \rangle$

Addition is the inverse of subtraction: if $b \leq x$ then $b \# + (x - b) = x$. An example of induction over a quantified formula (a product). Uses rewriting with a quantified, implicative inductive hypothesis.

schematic-goal *add-diff-inverse-lemma*:
 $b:N \Longrightarrow ?a : \prod x:N. Eq(N, b-x, 0) \longrightarrow Eq(N, b \# + (x-b), x)$
 $\langle proof \rangle$

Version of above with premise $b - a = 0$ i.e. $a \geq b$. Using *ProdE* does not work – for $?B(?a)$ is ambiguous. Instead, *add-diff-inverse-lemma* states the desired induction scheme; the use of *THEN* below instantiates Vars in *ProdE* automatically.

lemma *add-diff-inverse*: $\llbracket a:N; b:N; b - a = 0 : N \rrbracket \Longrightarrow b \# + (a-b) = a : N$
 $\langle proof \rangle$

3.7 Absolute difference

Typing of absolute difference: short and long versions.

lemma *absdiff-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \mid - \mid b : N$
 $\langle proof \rangle$

lemma *absdiff-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \mid - \mid b = c \mid - \mid d : N$
 $\langle proof \rangle$

lemma *absdiff-self-eq-0*: $a:N \Longrightarrow a \mid - \mid a = 0 : N$
 $\langle proof \rangle$

lemma *absdiffC0*: $a:N \Longrightarrow 0 \mid - \mid a = a : N$
 $\langle proof \rangle$

lemma *absdiff-succ-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow succ(a) \mid - \mid succ(b) = a \mid - \mid b : N$
 $\langle proof \rangle$

Note how easy using commutative laws can be? ...not always...

lemma *absdiff-commute*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \mid - \mid b = b \mid - \mid a : N$
 $\langle proof \rangle$

If $a + b = 0$ then $a = 0$. Surprisingly tedious.

schematic-goal *add-eq0-lemma*: $\llbracket a:N; b:N \rrbracket \Longrightarrow ?c : Eq(N, a \# + b, 0) \longrightarrow Eq(N, a, 0)$
 $\langle proof \rangle$

Version of above with the premise $a + b = 0$. Again, resolution instantiates variables in *ProdE*.

lemma *add-eq0*: $\llbracket a:N; b:N; a \# + b = 0 : N \rrbracket \Longrightarrow a = 0 : N$
 $\langle proof \rangle$

Here is a lemma to infer $a - b = 0$ and $b - a = 0$ from $a \mid\mid b = 0$, below.

schematic-goal *absdiff-eq0-lem*:

$$\llbracket a:N; b:N; a \mid\mid b = 0 : N \rrbracket \Longrightarrow ?a : Eq(N, a-b, 0) \times Eq(N, b-a, 0)$$

<proof>

If $a \mid\mid b = 0$ then $a = b$ proof: $a - b = 0$ and $b - a = 0$, so $b = a + (b - a) = a + 0 = a$.

lemma *absdiff-eq0*: $\llbracket a \mid\mid b = 0 : N; a:N; b:N \rrbracket \Longrightarrow a = b : N$

<proof>

3.8 Remainder and Quotient

Typing of remainder: short and long versions.

lemma *mod-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \text{ mod } b : N$

<proof>

lemma *mod-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \text{ mod } b = c \text{ mod } d : N$

<proof>

Computation for *mod*: 0 and successor cases.

lemma *modC0*: $b:N \Longrightarrow 0 \text{ mod } b = 0 : N$

<proof>

lemma *modC-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow$
 $\text{succ}(a) \text{ mod } b = \text{rec}(\text{succ}(a \text{ mod } b) \mid\mid b, 0, \lambda x y. \text{succ}(a \text{ mod } b)) : N$

<proof>

Typing of quotient: short and long versions.

lemma *div-typing*: $\llbracket a:N; b:N \rrbracket \Longrightarrow a \text{ div } b : N$

<proof>

lemma *div-typingL*: $\llbracket a = c:N; b = d:N \rrbracket \Longrightarrow a \text{ div } b = c \text{ div } d : N$

<proof>

lemmas *div-typing-rls* = *mod-typing div-typing absdiff-typing*

Computation for quotient: 0 and successor cases.

lemma *divC0*: $b:N \Longrightarrow 0 \text{ div } b = 0 : N$

<proof>

lemma *divC-succ*: $\llbracket a:N; b:N \rrbracket \Longrightarrow$
 $\text{succ}(a) \text{ div } b = \text{rec}(\text{succ}(a) \text{ mod } b, \text{succ}(a \text{ div } b), \lambda x y. a \text{ div } b) : N$

<proof>

Version of above with same condition as the *mod* one.

lemma *divC-succ2*: $\llbracket a:N; b:N \rrbracket \Longrightarrow$
 $\text{succ}(a) \text{ div } b = \text{rec}(\text{succ}(a \text{ mod } b) \mid\mid b, \text{succ}(a \text{ div } b), \lambda x y. a \text{ div } b) : N$

$\langle proof \rangle$

For case analysis on whether a number is 0 or a successor.

lemma *iszero-decidable*: $a:N \implies rec(a, inl(eq), \lambda ka kb. inr(<ka, eq>)) :$
 $Eq(N, a, 0) + (\sum x:N. Eq(N, a, succ(x)))$
 $\langle proof \rangle$

Main Result. Holds when b is 0 since $a \bmod 0 = a$ and $a \div 0 = 0$.

lemma *mod-div-equality*: $\llbracket a:N; b:N \rrbracket \implies a \bmod b \# + (a \div b) \# * b = a : N$
 $\langle proof \rangle$

end

4 Easy examples: type checking and type deduction

theory *Typechecking*
imports *../CTT*
begin

4.1 Single-step proofs: verifying that a type is well-formed

schematic-goal $?A \text{ type}$
 $\langle proof \rangle$

schematic-goal $?A \text{ type}$
 $\langle proof \rangle$

schematic-goal $\prod z: ?A . N + ?B(z) \text{ type}$
 $\langle proof \rangle$

4.2 Multi-step proofs: Type inference

lemma $\prod w:N . N + N \text{ type}$
 $\langle proof \rangle$

schematic-goal $<0, succ(0)> : ?A$
 $\langle proof \rangle$

schematic-goal $\prod w:N . Eq(?A, w, w) \text{ type}$
 $\langle proof \rangle$

schematic-goal $\prod x:N . \prod y:N . Eq(?A, x, y) \text{ type}$
 $\langle proof \rangle$

typechecking an application of fst

schematic-goal $(\lambda u. split(u, \lambda v w. v)) ' <0, succ(0)> : ?A$
 $\langle proof \rangle$

typechecking the predecessor function

schematic-goal $\lambda n. \text{rec}(n, 0, \lambda x y. x) : ?A$
 $\langle \text{proof} \rangle$

typechecking the addition function

schematic-goal $\lambda n. \lambda m. \text{rec}(n, m, \lambda x y. \text{succ}(y)) : ?A$
 $\langle \text{proof} \rangle$

Proofs involving arbitrary types. For concreteness, every type variable left over is forced to be N

$\langle ML \rangle$

schematic-goal $\lambda w. \langle w, w \rangle : ?A$
 $\langle \text{proof} \rangle$

schematic-goal $\lambda x. \lambda y. x : ?A$
 $\langle \text{proof} \rangle$

typechecking fst (as a function object)

schematic-goal $\lambda i. \text{split}(i, \lambda j k. j) : ?A$
 $\langle \text{proof} \rangle$

end

5 Examples with elimination rules

theory *Elimination*
imports *../CTT*
begin

This finds the functions fst and snd!

schematic-goal $[\text{folded basic-defs}] : A \text{ type} \implies ?a : (A \times A) \longrightarrow A$
 $\langle \text{proof} \rangle$

schematic-goal $[\text{folded basic-defs}] : A \text{ type} \implies ?a : (A \times A) \longrightarrow A$
 $\langle \text{proof} \rangle$

Double negation of the Excluded Middle

schematic-goal $A \text{ type} \implies ?a : ((A + (A \longrightarrow F)) \longrightarrow F) \longrightarrow F$
 $\langle \text{proof} \rangle$

Experiment: the proof above in Isar

lemma
 assumes $A \text{ type}$ **shows** $(\lambda f. f \text{ ' } \text{inr}(\lambda y. f \text{ ' } \text{inl}(y))) : ((A + (A \longrightarrow F)) \longrightarrow F) \longrightarrow F$
 $\langle \text{proof} \rangle$

schematic-goal $\llbracket A \text{ type}; B \text{ type} \rrbracket \Longrightarrow ?a : (A \times B) \longrightarrow (B \times A)$
 $\langle \text{proof} \rangle$

Binary sums and products

schematic-goal $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : (A + B \longrightarrow C) \longrightarrow (A \longrightarrow C)$
 $\times (B \longrightarrow C)$
 $\langle \text{proof} \rangle$

schematic-goal $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : A \times (B + C) \longrightarrow (A \times B +$
 $A \times C)$
 $\langle \text{proof} \rangle$

schematic-goal
assumes $A \text{ type}$
and $\bigwedge x. x:A \Longrightarrow B(x) \text{ type}$
and $\bigwedge x. x:A \Longrightarrow C(x) \text{ type}$
shows $?a : (\sum x:A. B(x) + C(x)) \longrightarrow (\sum x:A. B(x)) + (\sum x:A. C(x))$
 $\langle \text{proof} \rangle$

Construction of the currying functional

schematic-goal $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : (A \times B \longrightarrow C) \longrightarrow (A \longrightarrow (B$
 $\longrightarrow C))$
 $\langle \text{proof} \rangle$

schematic-goal
assumes $A \text{ type}$
and $\bigwedge x. x:A \Longrightarrow B(x) \text{ type}$
and $\bigwedge z. z: (\sum x:A. B(x)) \Longrightarrow C(z) \text{ type}$
shows $?a : \prod f: (\prod z: (\sum x:A . B(x)) . C(z)).$
 $(\prod x:A . \prod y:B(x) . C(<x,y>))$
 $\langle \text{proof} \rangle$

Martin-Löf (1984), page 48: axiom of sum-elimination (uncurry)

schematic-goal $\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : (A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \times$
 $B \longrightarrow C)$
 $\langle \text{proof} \rangle$

schematic-goal
assumes $A \text{ type}$
and $\bigwedge x. x:A \Longrightarrow B(x) \text{ type}$
and $\bigwedge z. z: (\sum x:A . B(x)) \Longrightarrow C(z) \text{ type}$
shows $?a : (\prod x:A . \prod y:B(x) . C(<x,y>))$
 $\longrightarrow (\prod z: (\sum x:A . B(x)) . C(z))$
 $\langle \text{proof} \rangle$

Function application

schematic-goal $\llbracket A \text{ type}; B \text{ type} \rrbracket \Longrightarrow ?a : ((A \longrightarrow B) \times A) \longrightarrow B$
 $\langle \text{proof} \rangle$

Basic test of quantifier reasoning

schematic-goal
assumes $A \text{ type}$
and $B \text{ type}$
and $\bigwedge x y. \llbracket x:A; y:B \rrbracket \Longrightarrow C(x,y) \text{ type}$
shows
 $?a : (\sum y:B. \prod x:A. C(x,y))$
 $\longrightarrow (\prod x:A. \sum y:B. C(x,y))$
 $\langle \text{proof} \rangle$

Martin-Löf (1984) pages 36-7: the combinator S

schematic-goal
assumes $A \text{ type}$
and $\bigwedge x. x:A \Longrightarrow B(x) \text{ type}$
and $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \Longrightarrow C(x,y) \text{ type}$
shows $?a : (\prod x:A. \prod y:B(x). C(x,y))$
 $\longrightarrow (\prod f: (\prod x:A. B(x)). \prod x:A. C(x, f'x))$
 $\langle \text{proof} \rangle$

Martin-Löf (1984) page 58: the axiom of disjunction elimination

schematic-goal
assumes $A \text{ type}$
and $B \text{ type}$
and $\bigwedge z. z: A+B \Longrightarrow C(z) \text{ type}$
shows $?a : (\prod x:A. C(\text{inl}(x))) \longrightarrow (\prod y:B. C(\text{inr}(y)))$
 $\longrightarrow (\prod z: A+B. C(z))$
 $\langle \text{proof} \rangle$

schematic-goal $[\text{folded basic-defs}]$:

$\llbracket A \text{ type}; B \text{ type}; C \text{ type} \rrbracket \Longrightarrow ?a : (A \longrightarrow B \times C) \longrightarrow (A \longrightarrow B) \times (A \longrightarrow C)$
 $\langle \text{proof} \rangle$

AXIOM OF CHOICE! Delicate use of elimination rules

schematic-goal
assumes $A \text{ type}$
and $\bigwedge x. x:A \Longrightarrow B(x) \text{ type}$
and $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \Longrightarrow C(x,y) \text{ type}$
shows $?a : (\prod x:A. \sum y:B(x). C(x,y)) \longrightarrow (\sum f: (\prod x:A. B(x)). \prod x:A. C(x, f'x))$
 $\langle \text{proof} \rangle$

A structured proof of AC

lemma *Axiom-of-Choice*:

```

assumes  $A$  type
and  $\bigwedge x. x:A \implies B(x)$  type
and  $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies C(x,y)$  type
shows  $(\lambda f. <\lambda x. fst(f'x), \lambda x. snd(f'x)>)$ 
       $: (\prod x:A. \sum y:B(x). C(x,y)) \longrightarrow (\sum f: (\prod x:A. B(x)). \prod x:A. C(x, f'x))$ 
 $\langle proof \rangle$ 

```

Axiom of choice. Proof without fst, snd. Harder still!

schematic-goal [*folded basic-defs*]:

```

assumes  $A$  type
and  $\bigwedge x. x:A \implies B(x)$  type
and  $\bigwedge x y. \llbracket x:A; y:B(x) \rrbracket \implies C(x,y)$  type
shows  $?a : (\prod x:A. \sum y:B(x). C(x,y)) \longrightarrow (\sum f: (\prod x:A. B(x)). \prod x:A. C(x,$ 
 $f'x))$ 
 $\langle proof \rangle$ 

```

Example of sequent-style deduction

schematic-goal

```

assumes  $A$  type
and  $B$  type
and  $\bigwedge z. z:A \times B \implies C(z)$  type
shows  $?a : (\sum z:A \times B. C(z)) \longrightarrow (\sum u:A. \sum v:B. C(<u,v>))$ 
 $\langle proof \rangle$ 

```

end

6 Equality reasoning by rewriting

theory *Equality*

imports *../CTT*

begin

```

lemma split-eq:  $p : Sum(A,B) \implies split(p,pair) = p : Sum(A,B)$ 
 $\langle proof \rangle$ 

```

```

lemma when-eq:  $\llbracket A \text{ type}; B \text{ type}; p : A+B \rrbracket \implies when(p,inl,inr) = p : A + B$ 
 $\langle proof \rangle$ 

```

in the "rec" formulation of addition, $0 + n = n$

```

lemma  $p:N \implies rec(p,0, \lambda y z. succ(y)) = p : N$ 
 $\langle proof \rangle$ 

```

the harder version, $n + 0 = n$: recursive, uses induction hypothesis

```

lemma  $p:N \implies rec(p,0, \lambda y z. succ(z)) = p : N$ 
 $\langle proof \rangle$ 

```

Associativity of addition

```

lemma  $\llbracket a:N; b:N; c:N \rrbracket$ 

```

$\implies \text{rec}(\text{rec}(a, b, \lambda x y. \text{succ}(y)), c, \lambda x y. \text{succ}(y)) =$
 $\text{rec}(a, \text{rec}(b, c, \lambda x y. \text{succ}(y)), \lambda x y. \text{succ}(y)) : N$
 $\langle \text{proof} \rangle$

Martin-Löf (1984) page 62: pairing is surjective

lemma $p : \text{Sum}(A, B) \implies \langle \text{split}(p, \lambda x y. x), \text{split}(p, \lambda x y. y) \rangle = p : \text{Sum}(A, B)$
 $\langle \text{proof} \rangle$

lemma $\llbracket a : A; b : B \rrbracket \implies (\lambda u. \text{split}(u, \lambda v w. \langle w, v \rangle)) \text{ ` } \langle a, b \rangle = \langle b, a \rangle : \sum x : B.$
 A
 $\langle \text{proof} \rangle$

a contrived, complicated simplication, requires sum-elimination also

lemma $(\lambda f. \lambda x. f'(f'x)) \text{ ` } (\lambda u. \text{split}(u, \lambda v w. \langle w, v \rangle)) =$
 $\lambda x. x : \prod x : (\sum y : N. N). (\sum y : N. N)$
 $\langle \text{proof} \rangle$

end

7 Synthesis examples, using a crude form of narrowing

theory *Synthesis*
imports *../CTT*
begin

discovery of predecessor function

schematic-goal $?a : \sum \text{pred} : ?A. \text{Eq}(N, \text{pred}'0, 0) \times (\prod n : N. \text{Eq}(N, \text{pred} \text{ ` } \text{succ}(n), n))$
 $\langle \text{proof} \rangle$

the function fst as an element of a function type

schematic-goal $[\text{folded basic-defs}] :$
 $A \text{ type} \implies ?a : \sum f : ?B. \prod i : A. \prod j : A. \text{Eq}(A, f \text{ ` } \langle i, j \rangle, i)$
 $\langle \text{proof} \rangle$

An interesting use of the eliminator, when

schematic-goal $?a : \prod i : N. \text{Eq}(?A, ?b(\text{inl}(i)), \langle 0, i \rangle)$
 $\times \text{Eq}(?A, ?b(\text{inr}(i)), \langle \text{succ}(0), i \rangle)$
 $\langle \text{proof} \rangle$

schematic-goal $?a : \prod i : N. \text{Eq}(?A(i), ?b(\text{inl}(i)), \langle 0, i \rangle)$
 $\times \text{Eq}(?A(i), ?b(\text{inr}(i)), \langle \text{succ}(0), i \rangle)$
 $\langle \text{proof} \rangle$

A tricky combination of when and split

schematic-goal [*folded basic-defs*]:

$$\begin{aligned} ?a : \prod i:N. \prod j:N. & Eq(?A, ?b(inl(<i,j>)), i) \\ & \times Eq(?A, ?b(inr(<i,j>)), j) \\ \langle proof \rangle \end{aligned}$$

schematic-goal $?a : \prod i:N. \prod j:N. Eq(?A(i,j), ?b(inl(<i,j>)), i)$
 $\times Eq(?A(i,j), ?b(inr(<i,j>)), j)$
 $\langle proof \rangle$

schematic-goal $?a : \prod i:N. \prod j:N. Eq(N, ?b(inl(<i,j>)), i)$
 $\times Eq(N, ?b(inr(<i,j>)), j)$
 $\langle proof \rangle$

Deriving the addition operator

schematic-goal [*folded arith-defs*]:

$$\begin{aligned} ?c : \prod n:N. & Eq(N, ?f(0,n), n) \\ & \times (\prod m:N. Eq(N, ?f(succ(m), n), succ(?f(m,n)))) \\ \langle proof \rangle \end{aligned}$$

The addition function – using explicit lambdas

schematic-goal [*folded arith-defs*]:

$$\begin{aligned} ?c : \sum plus : ?A . \\ \prod x:N. & Eq(N, plus'0'x, x) \\ & \times (\prod y:N. Eq(N, plus'succ(y)'x, succ(plus'y'x))) \\ \langle proof \rangle \end{aligned}$$

end