Defining Nonprimitively (Co)recursive Functions in Isabelle/HOL

Jasmin Christian Blanchette, Aymeric Bouzy,
Andreas Lochbihler, Andrei Popescu, and
Dmitriy Traytel

23 May 2024

Abstract

This tutorial describes the definitional package for nonprimitively corecursive functions in Isabelle/HOL. The following commands are provided: **corec**, **corecursive**, **friend_of_corec**, and **coinduction_upto**. They supplement **codatatype**, **primcorec**, and **primcorecursive**, which define codatatypes and primitively corecursive functions.

Contents

1 Introduction

Isabelle's (co)datatype package [1] offers a convenient syntax for introducing codatatypes. For example, the type of (infinite) streams can be defined as follows (cf. `~~/src/HOL/Library/Stream.thy`):

codatatype *'a stream* =
SCons (*shd*: *'a*) (*stl*: "*'a stream*")

The (co)datatype package also provides two commands, **primcorec** and **primcorecursive**, for defining primitively corecursive functions.

This tutorial presents a definitional package for functions beyond primitive corecursion. It describes **corec** and related commands: **corecursive**, **friend_of_corec**, and **coinduction_upto**. It also covers the *corec_unique* proof method. The package is not part of *Main*; it is located in `~~/src/HOL/Library/BNF_Corec.thy`.

The **corec** command generalizes **primcorec** in three main respects. First, it allows multiple constructors around corecursive calls, where **primcorec** expects exactly one. For example:

corec *oneTwos* :: "*nat stream*" **where**
"*oneTwos* = *SCons* 1 (*SCons* 2 *oneTwos*)"

Second, **corec** allows other functions than constructors to appear in the corecursive call context (i.e., around any self-calls on the right-hand side of the equation). The requirement on these functions is that they must be *friendly*. Intuitively, a function is friendly if it needs to destruct at most one constructor of input to produce one constructor of output. We can register functions as friendly using the **friend_of_corec** command, or by passing

the *friend* option to **corec**. The friendliness check relies on an internal syntactic check in combination with a parametricity subgoal, which must be discharged manually (typically using *transfer_prover* or *transfer_prover_eq*).

Third, **corec** allows self-calls that are not guarded by a constructor, as long as these calls occur in a friendly context (a context consisting exclusively of friendly functions) and can be shown to be terminating (well founded). The mixture of recursive and corecursive calls in a single function can be quite useful in practice.

Internally, the package synthesizes corecursors that take into account the possible call contexts. The corecursor is accompanined by a corresponding, equally general coinduction principle. The corecursor and the coinduction principle grow in expressiveness as we interact with it. In process algebra terminology, corecursion and coinduction take place *up to* friendly contexts.

The package fully adheres to the LCF philosophy [5]: The characteristic theorems associated with the specified corecursive functions are derived rather than introduced axiomatically. (Exceptionally, most of the internal proof obligations are omitted if the *quick_and_dirty* option is enabled.) The package is described in a pair of scientific papers [2,3]. Some of the text and examples below originate from there.

This tutorial is organized as follows:

- Section 2, "Introductory Examples," describes how to specify corecursive functions and to reason about them.
- Section 3, "Command Syntax," describes the syntax of the commands offered by the package.
- Section 4, "Generated Theorems," lists the theorems produced by the package's commands.
- Section 5, "Proof Methods," briefly describes the *corec_unique* and *transfer_prover_eq* proof methods.
- Section 6, "Attribute," briefly describes the *friend_of_corec_simps* attribute, which can be used to strengthen the tactics underlying the **friend_of_corec** and **corec** (*friend*) commands.
- Section 7, "Known Bugs and Limitations," concludes with known open issues.

Although it is more powerful than **primcorec** in many respects, **corec** suffers from a number of limitations. Most notably, it does not support mutually corecursive codatatypes, and it is less efficient than **primcorec** because it needs to dynamically synthesize corecursors and corresponding coinduction principles to accommodate the friends.

Comments and bug reports concerning either the package or this tutorial should be directed to the first author at `jasmin.blanchette@gmail.com` or to the `cl-isabelle-users` mailing list.

2 Introductory Examples

The package is illustrated through concrete examples featuring different flavors of corecursion. More examples can be found in the directory `~~/src/HOL/Corec_Examples`.

2.1 Simple Corecursion

The case studies by Rutten [7] and Hinze [6] on stream calculi serve as our starting point. The following definition of pointwise sum can be performed with either **primcorec** or **corec**:

primcorec *ssum* :: "('*a* :: *plus*) *stream* ⇒ '*a stream* ⇒ '*a stream*" **where**
"*ssum xs ys* = *SCons* (*shd xs* + *shd ys*) (*ssum* (*stl xs*) (*stl ys*))"

Pointwise sum meets the friendliness criterion. We register it as a friend using the **friend_of_corec** command. The command requires us to give a specification of *ssum* where a constructor (*SCons*) occurs at the outermost position on the right-hand side. Here, we can simply reuse the **primcorec** specification above:

friend_of_corec *ssum* :: "('*a* :: *plus*) *stream* ⇒ '*a stream* ⇒ '*a stream*" **where**
"*ssum xs ys* = *SCons* (*shd xs* + *shd ys*) (*ssum* (*stl xs*) (*stl ys*))"
apply (*rule ssum.code*)
by *transfer_prover*

The command emits two subgoals. The first subgoal corresponds to the equation we specified and is trivial to discharge. The second subgoal is a parametricity property that captures the the requirement that the function may destruct at most one constructor of input to produce one constructor of output. This subgoal can usually be discharged using the *transfer_prover* or *transfer_prover_eq* proof method (Section 5.2). The latter replaces equality relations by their relator terms according to the *relator_eq* theorem collection before it invokes *transfer_prover*.

After registering *ssum* as a friend, we can use it in the corecursive call context, either inside or outside the constructor guard:

corec *fibA* :: "*nat stream*" **where**
"*fibA* = *SCons* 0 (*ssum* (*SCons* 1 *fibA*) *fibA*)"

corec *fibB* :: "*nat stream*" **where**
"*fibB* = *ssum* (*SCons* 0 (*SCons* 1 *fibB*)) (*SCons* 0 *fibB*)"

Using the *friend* option, we can simultaneously define a function and register it as a friend:

corec (*friend*)
sprod :: "('*a* :: {*plus*,*times*}) *stream* ⇒ '*a stream* ⇒ '*a stream*"
where
"*sprod xs ys* =
SCons (*shd xs* ∗ *shd ys*) (*ssum* (*sprod xs* (*stl ys*)) (*sprod* (*stl xs*) *ys*))"

corec (*friend*) *sexp* :: "*nat stream* ⇒ *nat stream*" **where**
"*sexp xs* = *SCons* (2 ^^ *shd xs*) (*sprod* (*stl xs*) (*sexp xs*))"

The parametricity subgoal is given to *transfer_prover_eq* (Section 5.2).

The *sprod* and *sexp* functions provide shuffle product and exponentiation on streams. We can use them to define the stream of factorial numbers in two different ways:

corec *factA* :: "*nat stream*" **where**
"*factA* = (*let zs* = *SCons* 1 *factA in sprod zs zs*)"

corec *factB* :: "*nat stream*" **where**
"*factB* = *sexp* (*SCons* 0 *factB*)"

The arguments of friendly functions can be of complex types involving the target codatatype. The following example defines the supremum of a finite set of streams by primitive corecursion and registers it as friendly:

corec (*friend*) *sfsup* :: "*nat stream fset* ⇒ *nat stream*" **where**
"*sfsup X* = *SCons* (*Sup* (*fset* (*fimage shd X*))) (*sfsup* (*fimage stl X*))"

In general, the arguments may be any bounded natural functor (BNF) [1], with the restriction that the target codatatype (*nat stream*) may occur only in a *live* position of the BNF. For this reason, the following function, on unbounded sets, cannot be registered as a friend:

primcorec *ssup* :: "*nat stream set* ⇒ *nat stream*" **where**
"*ssup X* = *SCons* (*Sup* (*image shd X*)) (*ssup* (*image stl X*))"

2.2 Nested Corecursion

The package generally supports arbitrary codatatypes with multiple constructors and nesting through other type constructors (BNFs). Consider the following type of finitely branching Rose trees of potentially infinite depth:

codatatype '*a tree* =

```
  Node (lab: 'a) (sub: "'a tree list")
```

We first define the pointwise sum of two trees analogously to *ssum*:

```
corec (friend) tsum :: "('a :: plus) tree ⇒ 'a tree ⇒ 'a tree" where
  "tsum t u =
   Node (lab t + lab u) (map (λ(t', u'). tsum t' u') (zip (sub t) (sub u)))"
```

Here, *map* is the standard map function on lists, and *zip* converts two parallel lists into a list of pairs. The *tsum* function is primitively corecursive. Instead of **corec** (*friend*), we could also have used **primcorec** and **friend_of_corec**, as we did for *ssum*.

Once *tsum* is registered as friendly, we can use it in the corecursive call context of another function:

```
corec (friend) ttimes :: "('a :: {plus,times}) tree ⇒ 'a tree ⇒ 'a tree" where
  "ttimes t u = Node (lab t * lab u)
     (map (λ(t', u'). tsum (ttimes t u') (ttimes t' u)) (zip (sub t) (sub u)))"
```

All the syntactic convenience provided by **primcorec** is also supported by **corec**, **corecursive**, and **friend_of_corec**. In particular, nesting through the function type can be expressed using λ-abstractions and function applications rather than through composition ((∘), the map function for ⇒). For example:

```
codatatype 'a language =
  Lang (𝔬: bool) (𝔡: "'a ⇒ 'a language")

corec (friend) Plus :: "'a language ⇒ 'a language ⇒ 'a language" where
  "Plus r s = Lang (𝔬 r ∨ 𝔬 s) (λa. Plus (𝔡 r a) (𝔡 s a))"

corec (friend) Times :: "'a language ⇒ 'a language ⇒ 'a language" where
  "Times r s = Lang (𝔬 r ∧ 𝔬 s)
     (λa. if 𝔬 r then Plus (Times (𝔡 r a) s) (𝔡 s a) else Times (𝔡 r a) s)"

corec (friend) Star :: "'a language ⇒ 'a language" where
  "Star r = Lang True (λa. Times (𝔡 r a) (Star r))"

corec (friend) Inter :: "'a language ⇒ 'a language ⇒ 'a language" where
  "Inter r s = Lang (𝔬 r ∧ 𝔬 s) (λa. Inter (𝔡 r a) (𝔡 s a))"

corec (friend) PLUS :: "'a language list ⇒ 'a language" where
  "PLUS xs = Lang (∃x ∈ set xs. 𝔬 x) (λa. PLUS (map (λr. 𝔡 r a) xs))"
```

2.3 Mixed Recursion–Corecursion

It is often convenient to let a corecursive function perform some finite computation before producing a constructor. With mixed recursion–corecursion,

a finite number of unguarded recursive calls perform this calculation before reaching a guarded corecursive call. Intuitively, the unguarded recursive call can be unfolded to arbitrary finite depth, ultimately yielding a purely corecursive definition. An example is the *primes* function from Di Gianantonio and Miculan [4]:

```
corecursive primes :: "nat ⇒ nat ⇒ nat stream" where
  "primes m n =
   (if (m = 0 ∧ n > 1) ∨ coprime m n then
      SCons n (primes (m * n) (n + 1))
    else
      primes m (n + 1))"
  apply (relation "measure (λ(m, n).
    if n = 0 then 1 else if coprime m n then 0 else m - n mod m)")
  apply (auto simp: mod_Suc diff_less_mono2 intro: Suc_lessI elim!: not_coprimeE)
  apply (metis dvd_1_iff_1 dvd_eq_mod_eq_0 mod_0 mod_Suc mod_Suc_eq
mod_mod_cancel)
  done
```

The **corecursive** command is a variant of **corec** that allows us to specify a termination argument for any unguarded self-call.

When called with $m = 1$ and $n = 2$, the *primes* function computes the stream of prime numbers. The unguarded call in the *else* branch increments n until it is coprime to the first argument m (i.e., the greatest common divisor of m and n is 1).

For any positive integers m and n, the numbers m and $m * n + 1$ are coprime, yielding an upper bound on the number of times n is increased. Hence, the function will take the *else* branch at most finitely often before taking the then branch and producing one constructor. There is a slight complication when $m = 0 \wedge n > 1$: Without the first disjunct in the *if* condition, the function could stall. (This corner case was overlooked in the original example [4].)

In the following examples, termination is discharged automatically by **corec** by invoking *lexicographic_order*:

```
corec catalan :: "nat ⇒ nat stream" where
  "catalan n =
   (if n > 0 then ssum (catalan (n - 1)) (SCons 0 (catalan (n + 1)))
    else SCons 1 (catalan 1))"

corec collatz :: "nat ⇒ nat stream" where
  "collatz n = (if even n ∧ n > 0 then collatz (n div 2)
     else SCons n (collatz (3 * n + 1)))"
```

A more elaborate case study, revolving around the filter function on lazy

lists, is presented in `~~/src/HOL/Corec_Examples/LFilter.thy`.

2.4 Self-Friendship

The package allows us to simultaneously define a function and use it as its own friend, as in the following definition of a "skewed product":

corec (*friend*)
  sskew :: "('*a* :: {*plus*,*times*}) *stream* ⇒ '*a stream* ⇒ '*a stream*"
where
  "*sskew xs ys* =
   SCons (*shd xs* ∗ *shd ys*) (*sskew* (*sskew xs* (*stl ys*)) (*sskew* (*stl xs*) *ys*))"

Such definitions, with nested self-calls on the right-hand side, cannot be separated into a **corec** part and a **friend_of_corec** part.

2.5 Coinduction

Once a corecursive specification has been accepted, we normally want to reason about it. The *codatatype* command generates a structural coinduction principle that matches primitively corecursive functions. For nonprimitive specifications, our package provides the more advanced proof principle of *coinduction up to congruence*—or simply *coinduction up-to*.

The structural coinduction principle for '*a stream*, called *stream.coinduct*, is as follows:

⟦*R stream stream*′; ⋀*stream stream*′. *R stream stream*′ ⟹ *shd stream* = *shd stream*′ ∧ *R* (*stl stream*) (*stl stream*′)⟧ ⟹ *stream* = *stream*′

Coinduction allows us to prove an equality $l = r$ on streams by providing a relation R that relates l and r (first premise) and that constitutes a bisimulation (second premise). Streams that are related by a bisimulation cannot be distinguished by taking observations (via the selectors *shd* and *stl*); hence they must be equal.

The coinduction up-to principle after registering *sskew* as friendly is available as *sskew.coinduct* and as one of the components of the theorem collection *stream.coinduct_upto*:

⟦*R stream stream*′; ⋀*stream stream*′. *R stream stream*′ ⟹ *shd stream* = *shd stream*′ ∧ *stream.v5.congclp R* (*stl stream*) (*stl stream*′)⟧ ⟹ *stream* = *stream*′

This rule is almost identical to structural coinduction, except that the corecursive application of R is generalized to *stream.v5.congclp R*.

The *stream.v5.congclp* predicate is equipped with the following introduction rules:

sskew.cong_base:
$P\ x\ y \Longrightarrow$ *stream.v5.congclp* $P\ x\ y$

sskew.cong_refl:
$x = y \Longrightarrow$ *stream.v5.congclp* $R\ x\ y$

sskew.cong_sym:
stream.v5.congclp $R\ x\ y \Longrightarrow$ *stream.v5.congclp* $R\ y\ x$

sskew.cong_trans:
$[\![$*stream.v5.congclp* $R\ x\ y$; *stream.v5.congclp* $R\ y\ z]\!] \Longrightarrow$ *stream.v5.congclp* $R\ x\ z$

sskew.cong_SCons:
$[\![x1 = y1$; *stream.v5.congclp* $R\ x2\ y2]\!] \Longrightarrow$ *stream.v5.congclp* R (*SCons* $x1\ x2$) (*SCons* $y1\ y2$)

sskew.cong_ssum:
$[\![$*stream.v5.congclp* $R\ x1\ y1$; *stream.v5.congclp* $R\ x2\ y2]\!] \Longrightarrow$ *stream.v5.congclp* R (*ssum* $x1\ x2$) (*ssum* $y1\ y2$)

sskew.cong_sprod:
$[\![$*stream.v5.congclp* $R\ x1\ y1$; *stream.v5.congclp* $R\ x2\ y2]\!] \Longrightarrow$ *stream.v5.congclp* R (*sprod* $x1\ x2$) (*sprod* $y1\ y2$)

sskew.cong_sskew:
$[\![$*stream.v5.congclp* $R\ x1\ y1$; *stream.v5.congclp* $R\ x2\ y2]\!] \Longrightarrow$ *stream.v5.congclp* R (*sskew* $x1\ x2$) (*sskew* $y1\ y2$)

The introduction rules are also available as *sskew.cong_intros*.

Notice that there is no introduction rule corresponding to *sexp*, because *sexp* has a more restrictive result type than *sskew* (*nat stream* vs. *'a stream*.

The version numbers, here *v5*, distinguish the different congruence closures generated for a given codatatype as more friends are registered. As much as possible, it is recommended to avoid referring to them in proof documents.

Since the package maintains a set of incomparable corecursors, there is also a set of associated coinduction principles and a set of sets of introduction rules. A technically subtle point is to make Isabelle choose the right rules in most situations. For this purpose, the package maintains the collection *stream.coinduct_upto* of coinduction principles ordered by increasing generality, which works well with Isabelle's philosophy of applying the first rule that matches. For example, after registering *ssum* as a friend, proving

the equality $l = r$ on *nat stream* might require coinduction principle for *nat stream*, which is up to *ssum*.

The collection *stream.coinduct_upto* is guaranteed to be complete and up to date with respect to the type instances of definitions considered so far, but occasionally it may be necessary to take the union of two incomparable coinduction principles. This can be done using the **coinduction_upto** command. Consider the following definitions:

```
codatatype ('a, 'b) tllist =
  TNil (terminal: 'b)
| TCons (thd: 'a) (ttl: "('a, 'b) tllist")

corec (friend) square_elems :: "(nat, 'b) tllist ⇒ (nat, 'b) tllist" where
  "square_elems xs =
   (case xs of
     TNil z ⇒ TNil z
   | TCons y ys ⇒ TCons (y ^^ 2) (square_elems ys))"

corec (friend) square_terminal :: "('a, int) tllist ⇒ ('a, int) tllist" where
  "square_terminal xs =
   (case xs of
     TNil z ⇒ TNil (z ^^ 2)
   | TCons y ys ⇒ TCons y (square_terminal ys))"
```

At this point, *tllist.coinduct_upto* contains three variants of the coinduction principles:

- ('*a*, *int*) *tllist* up to *TNil*, *TCons*, and *square_terminal*;
- (*nat*, '*b*) *tllist* up to *TNil*, *TCons*, and *square_elems*;
- ('*a*, '*b*) *tllist* up to *TNil* and *TCons*.

The following variant is missing:

- (*nat*, *int*) *tllist* up to *TNil*, *TCons*, *square_elems*, and *square_terminal*.

To generate it without having to define a new function with **corec**, we can use the following command:

```
coinduction_upto nat_int_tllist: "(nat, int) tllist"
```

This produces the theorems

```
nat_int_tllist.coinduct_upto
nat_int_tllist.cong_intros
```

(as well as the individually named introduction rules) and extends the dynamic collections *tllist.coinduct_upto* and *tllist.cong_intros*.

2.6 Uniqueness Reasoning

It is sometimes possible to achieve better automation by using a more specialized proof method than coinduction. Uniqueness principles maintain a good balance between expressiveness and automation. They exploit the property that a corecursive definition is the unique solution to a fixpoint equation.

The **corec**, **corecursive**, and **friend_of_corec** commands generate a property *f.unique* about the function of interest *f* that can be used to prove that any function that satisfies *f*'s corecursive specification must be equal to *f*. For example:

$$f = (\lambda xs\ ys.\ SCons\ (shd\ xs + shd\ ys)\ (f\ (stl\ xs)\ (stl\ ys))) \Longrightarrow f = ssum$$

The uniqueness principles are not restricted to functions defined using **corec** or **corecursive** or registered with **friend_of_corec**. Suppose $t\ x$ is an arbitrary term depending on x. The *corec_unique* proof method, provided by our tool, transforms subgoals of the form

$$\forall x.\ f\ x = H\ x\ f \Longrightarrow f\ x = t\ x$$

into

$$\forall x.\ t\ x = H\ x\ t$$

The higher-order functional H must be such that $f\,x = H\,x\,f$ would be a valid **corec** specification, but without nested self-calls or unguarded (recursive) calls. Thus, *corec_unique* proves uniqueness of t with respect to the given corecursive equation regardless of how t was defined. For example:

```
lemma
  fixes f :: "nat stream ⇒ nat stream ⇒ nat stream"
  assumes "∀ xs ys. f xs ys =
    SCons (shd ys * shd xs) (ssum (f xs (stl ys)) (f (stl xs) ys))"
  shows "f = sprod"
    using assms
  proof corec_unique
    show "sprod = (λxs ys :: nat stream.
        SCons (shd ys * shd xs) (ssum (sprod xs (stl ys)) (sprod (stl xs) ys)))"
      apply (rule ext)+
      apply (subst sprod.code)
      by simp
  qed
```

The proof method relies on some theorems generated by the package. If no function over a given codatatype has been defined using **corec** or

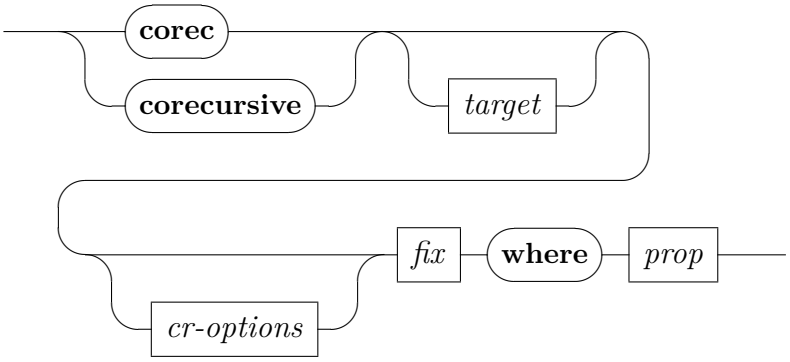
corecursive or registered as friendly using **friend_of_corec**, the theorems will not be available yet. In such cases, the theorems can be explicitly generated using the command

coinduction_upto *stream*: "*'a stream*"

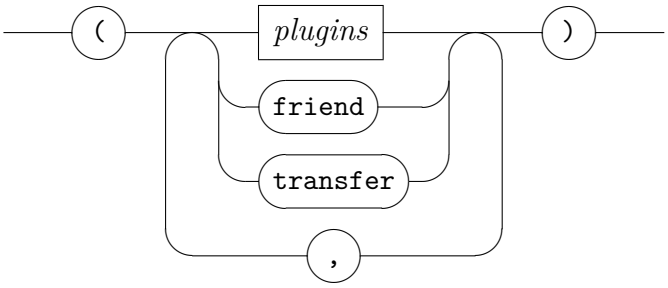
3 Command Syntax

3.1 corec and corecursive

corec : *local_theory* → *local_theory*
corecursive : *local_theory* → *proof*(*prove*)



cr-options



The **corec** and **corecursive** commands introduce a corecursive function over a codatatype.

The syntactic entity *target* can be used to specify a local context, *fix* denotes name with an optional type signature, and *prop* denotes a HOL proposition [8].

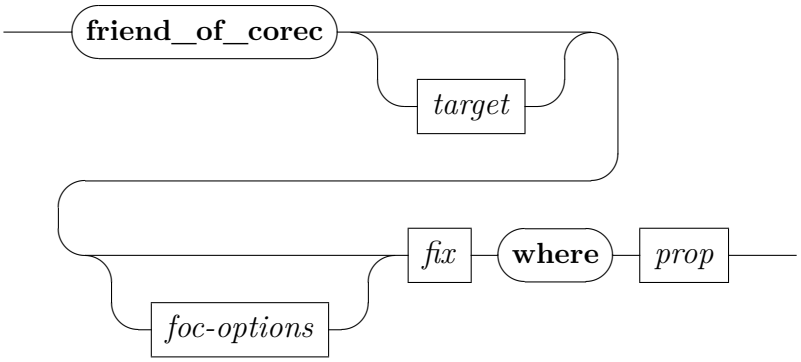
The optional target is optionally followed by a combination of the following options:

- The *plugins* option indicates which plugins should be enabled (*only*) or disabled (*del*). By default, all plugins are enabled.
- The *friend* option indicates that the defined function should be registered as a friend. This gives rise to additional proof obligations.
- The *transfer* option indicates that an unconditional transfer rule should be generated and proved *by transfer_prover*. The [*transfer_rule*] attribute is set on the generated theorem.

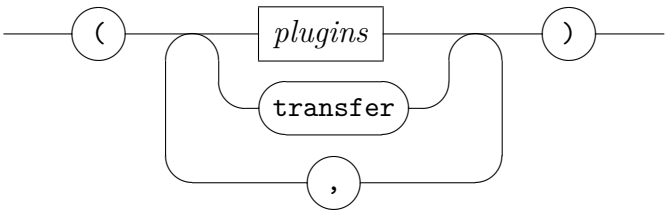
The **corec** command is an abbreviation for **corecursive** with appropriate applications of *transfer_prover_eq* (Section 5.2) and *lexicographic_order* to discharge any emerging proof obligations.

3.2 friend_of_corec

friend_of_corec : *local_theory* → *proof*(*prove*)



foc-options



The **friend_of_corec** command registers a corecursive function as friendly.

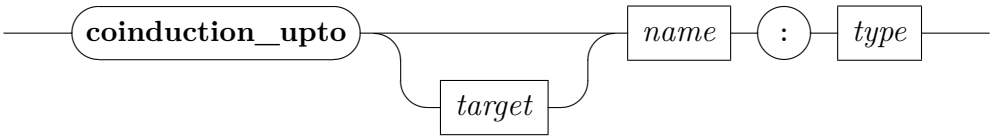
The syntactic entity *target* can be used to specify a local context, *fix* denotes name with an optional type signature, and *prop* denotes a HOL proposition [8].

The optional target is optionally followed by a combination of the following options:

- The *plugins* option indicates which plugins should be enabled (*only*) or disabled (*del*). By default, all plugins are enabled.
- The *transfer* option indicates that an unconditional transfer rule should be generated and proved *by transfer_prover*. The [*transfer_rule*] attribute is set on the generated theorem.

3.3 coinduction_upto

coinduction_upto : *local_theory* → *local_theory*



The **coinduction_upto** generates a coinduction up-to rule for a given instance of a (possibly polymorphic) codatatype and notes the result with the specified prefix.

The syntactic entity *name* denotes an identifier and *type* denotes a type [8].

4 Generated Theorems

The full list of named theorems generated by the package can be obtained by issuing the command **print_theorems** immediately after the datatype definition. This list excludes low-level theorems that reveal internal constructions. To make these accessible, add the line

declare [[*bnf_internals*]]

In addition to the theorem listed below for each command provided by the package, all commands update the dynamic theorem collections

t.***coinduct_upto***

t.***cong_intros***

for the corresponding codatatype t so that they always contain the most powerful coinduction up-to principles derived so far.

4.1 corec and corecursive

For a function f over codatatype t, the **corec** and **corecursive** commands generate the following properties (listed for *sexp*, cf. Section 2.1):

f.***code*** [*code*]:
$sexp\ xs = SCons\ 2^{shd\ xs}\ (sprod\ (stl\ xs)\ (sexp\ xs))$
The [*code*] attribute is set by the *code* plugin [1].

f.***coinduct*** [*consumes* 1, *case_names* t, *case_conclusion* $D_1\ \ldots\ D_n$]:
$[\![R\ nat\_stream\ nat\_stream'; \bigwedge nat\_stream\ nat\_stream'.\ R\ nat\_stream\ nat\_stream' \Longrightarrow shd\ nat\_stream = shd\ nat\_stream' \wedge stream.v3.congclp\ R\ (stl\ nat\_stream)\ (stl\ nat\_stream')]\!] \Longrightarrow nat\_stream = nat\_stream'$

f.***cong_intros***:
$P\ x\ y \Longrightarrow stream.v3.congclp\ P\ x\ y$
$x = y \Longrightarrow stream.v3.congclp\ R\ x\ y$
$stream.v3.congclp\ R\ x\ y \Longrightarrow stream.v3.congclp\ R\ y\ x$
$[\![stream.v3.congclp\ R\ x\ y; stream.v3.congclp\ R\ y\ z]\!] \Longrightarrow stream.v3.congclp\ R\ x\ z$
$[\![x1 = y1; stream.v3.congclp\ R\ x2\ y2]\!] \Longrightarrow stream.v3.congclp\ R\ (SCons\ x1\ x2)\ (SCons\ y1\ y2)$
$[\![stream.v3.congclp\ R\ x1\ y1; stream.v3.congclp\ R\ x2\ y2]\!] \Longrightarrow stream.v3.congclp\ R\ (ssum\ x1\ x2)\ (ssum\ y1\ y2)$
$[\![stream.v3.congclp\ R\ x1\ y1; stream.v3.congclp\ R\ x2\ y2]\!] \Longrightarrow stream.v3.congclp\ R\ (sprod\ x1\ x2)\ (sprod\ y1\ y2)$
$stream.v3.congclp\ R\ x\ y \Longrightarrow stream.v3.congclp\ R\ (sexp\ x)\ (sexp\ y)$

f.***unique***:
$f = (\lambda xs.\ SCons\ 2^{shd\ xs}\ (sprod\ (stl\ xs)\ (f\ xs))) \Longrightarrow f = sexp$
This property is not generated for mixed recursive–corecursive definitions.

f.***inner_induct***:
This property is only generated for mixed recursive–corecursive definitions. For *primes* (Section 2.3, it reads as follows:
$(\bigwedge m\ n.\ (\bigwedge x\ y.\ [\![(x,\ y) = (m,\ n); \neg\ (x = 0 \wedge 1 < y \vee coprime\ x\ y)]\!] \Longrightarrow P\ (x,\ y + 1)) \Longrightarrow P\ (m,\ n)) \Longrightarrow P\ a0$

The individual rules making up f*.cong_intros* are available as

f.***cong_base***

f.***cong_refl***

f.***cong_sym***

f.***cong_trans***

f.***cong_C*$_1$**, ..., *f*.***cong_C*$_n$**
where $C_1, \ldots, C_n$ are *t*'s constructors

f.***cong_f*$_1$**, ..., *f*.***cong_f*$_m$**
where $f_1, \ldots, f_m$ are the available friends for *t*

4.2 friend_of_corec

The **friend_of_corec** command generates the same theorems as **corec** and **corecursive**, except that it adds an optional *friend.* component to the names to prevent potential clashes (e.g., *f.friend.code*).

4.3 coinduction_upto

The **coinduction_upto** command generates the following properties (listed for *nat_int_tllist*):

t.***coinduct_upto*** [*consumes* 1, *case_names t*,
case_conclusion D_1 ... D_n]:
$[\![R\ nat\_int\_tllist\ nat\_int\_tllist'; \bigwedge nat\_int\_tllist\ nat\_int\_tllist'.\ R\ nat\_int\_tllist\ nat\_int\_tllist' \Longrightarrow is\_TNil\ nat\_int\_tllist = is\_TNil\ nat\_int\_tllist' \wedge (is\_TNil\ nat\_int\_tllist \longrightarrow is\_TNil\ nat\_int\_tllist' \longrightarrow terminal\ nat\_int\_tllist = terminal\ nat\_int\_tllist') \wedge (\neg\ is\_TNil\ nat\_int\_tllist \longrightarrow \neg\ is\_TNil\ nat\_int\_tllist' \longrightarrow thd\ nat\_int\_tllist = thd\ nat\_int\_tllist' \wedge tllist.v3.congclp\ R\ (ttl\ nat\_int\_tllist)\ (ttl\ nat\_int\_tllist'))]\!] \Longrightarrow nat\_int\_tllist = nat\_int\_tllist'$

t.***cong_intros***:
$P\ x\ y \Longrightarrow tllist.v3.congclp\ P\ x\ y$
$x = y \Longrightarrow tllist.v3.congclp\ R\ x\ y$
$tllist.v3.congclp\ R\ x\ y \Longrightarrow tllist.v3.congclp\ R\ y\ x$
$[\![tllist.v3.congclp\ R\ x\ y; tllist.v3.congclp\ R\ y\ z]\!] \Longrightarrow tllist.v3.congclp\ R\ x\ z$
$x = y \Longrightarrow tllist.v3.congclp\ R\ (TNil\ x)\ (TNil\ y)$
$[\![x1 = y1; tllist.v3.congclp\ R\ x2\ y2]\!] \Longrightarrow tllist.v3.congclp\ R\ (TCons\ x1\ x2)\ (TCons\ y1\ y2)$
$tllist.v3.congclp\ R\ x\ y \Longrightarrow tllist.v3.congclp\ R\ (square\_elems\ x)\ (square\_elems\ y)$

tllist.v3.congclp R x y $\Longrightarrow$ *tllist.v3.congclp R* (*square_terminal x*) (*square_terminal y*)

The individual rules making up *t.cong_intros* are available separately as *t.cong_base*, *t.cong_refl*, etc. (Section 4.1).

5 Proof Methods

5.1 *corec_unique*

The *corec_unique* proof method can be used to prove the uniqueness of a corecursive specification. See Section 2.6 for details.

5.2 *transfer_prover_eq*

The *transfer_prover_eq* proof method replaces the equality relation (=) with compound relator expressions according to *relator_eq* before calling *transfer_prover* on the current subgoal. It tends to work better than plain *transfer_prover* on the parametricity proof obligations of **corecursive** and **friend_of_corec**, because they often contain equality relations on complex types, which *transfer_prover* cannot reason about.

6 Attribute

6.1 *friend_of_corec_simps*

The *friend_of_corec_simps* attribute declares naturality theorems to be used by **friend_of_corec** and **corec** (*friend*) in deriving the user specification from reduction to primitive corecursion. Internally, these commands derive naturality theorems from the parametricity proof obligations dischared by the user or the *transfer_prover_eq* method, but this derivation fails if in the arguments of a higher-order constant a type variable occurs on both sides of the function type constructor. The required naturality theorem can then be declared with *friend_of_corec_simps*. See `~~/src/HOL/Corec_Examples/Tests/Iterate_GPV.thy` for an example.

7 Known Bugs and Limitations

This section lists the known bugs and limitations of the corecursion package at the time of this writing.

1. *Mutually corecursive codatatypes are not supported.*
2. *The signature of friend functions may not depend on type variables beyond those that appear in the codatatype.*
3. *The internal tactics may fail on legal inputs.* In some cases, this limitation can be circumvented using the *friend_of_corec_simps* attribute (Section 6.1).
4. *The transfer option is not implemented yet.*
5. *The constructor and destructor views offered by* **primcorec** *are not supported by* **corec** *and* **corecursive**.
6. *There is no mechanism for registering custom plugins.*
7. *The package does not interact well with locales.*
8. *The undocumented corecUU_transfer theorem is not as polymorphic as it could be.*
9. *All type variables occurring in the arguments of a friendly function must occur as direct arguments of the type constructor of the resulting type.*

References

[1] J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL.* `https://isabelle.in.tum.de/doc/datatypes.pdf`.

[2] J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with benefits: Implementing corecursion in foundational proof assistants. `http://www21.in.tum.de/~blanchet/amico.pdf`, 2016.

[3] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: A proof assistant perspective. In K. Fisher and J. H. Reppy, editors, *20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 192–204. ACM, 2015.

[4] P. Di Gianantonio and M. Miculan. A unifying approach to recursive and co-recursive definitions. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002*, volume 2646 of *LNCS*, pages 148–161. Springer, 2003.

[5] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[6] R. Hinze. Concrete stream calculus—An extended study. *J. Funct. Program.*, 20:463–535, 2010.

[7] J. J. M. M. Rutten. A coinductive calculus of streams. *Math. Struct. Comp. Sci.*, 15(1):93–147, 2005.

[8] M. Wenzel. *The Isabelle/Isar Reference Manual*. `https://isabelle.in.tum.de/doc/isar-ref.pdf`.