Abstract

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

Contents

1 Introduction 2

2 Introductory Examples 4
  2.1 Simple Corecursion ............................................. 4
  2.2 Nested Corecursion ........................................... 5
  2.3 Mixed Recursion–Corecursion ............................... 6
  2.4 Self-Friendship .................................................. 8
  2.5 Coinduction ...................................................... 8
  2.6 Uniqueness Reasoning .......................................... 11

3 Command Syntax 12
  3.1 \texttt{corec} and \texttt{corecursive} ......................... 12
  3.2 \texttt{friend_of_corec} .......................................... 13
  3.3 \texttt{coinduction_upto} ...................................... 14
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 primitively 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 accompanied 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`:

```plaintext
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:

```plaintext
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 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:

```plaintext
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 \texttt{ssum}:

\begin{verbatim}
corec (friend) tsum :: "(a :: plus) tree ⇒ 'a tree ⇒ 'a tree" where
  \texttt{tsum t u = Node (lab t + lab u) (map (\lambda (t', u'). \texttt{tsum t' u'}) (\texttt{zip (sub t) (sub u)}))}"
\end{verbatim}

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

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

\begin{verbatim}
corec (friend) ttimes :: "(a :: {plus,times}) tree ⇒ 'a tree ⇒ 'a tree" where
  \texttt{ttimes t u = Node (lab t * lab u) (map (\lambda (t', u'). \texttt{ttimes t' u'}) (\texttt{zip (sub t) (sub u)}))}"
\end{verbatim}

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

\begin{verbatim}
codatatype 'a language =
  Lang (o: bool) (d: "a ⇒ 'a language")

corec (friend) Plus :: "a language ⇒ 'a language ⇒ 'a language" where
  \texttt{Plus r s = Lang (o r ∨ o s) (\lambda. Plus (d r a) (d s a))}"

corec (friend) Times :: "a language ⇒ 'a language ⇒ 'a language" where
  \texttt{Times r s = Lang (o r ∧ o s) (\lambda a. 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
  \texttt{Star r = Lang True (\lambda a. Times (d r a) (Star r))}"

corec (friend) Inter :: "a language ⇒ 'a language ⇒ 'a language" where
  \texttt{Inter r s = Lang (o r ∧ o s) (\lambda. Inter (d r a) (d s a))}"

corec (friend) PLUS :: "a language list ⇒ 'a language" where
  \texttt{PLUS xs = Lang (\exists x \in set xs. o x) (\lambda a. PLUS (map (\lambda r. d r a) xs))}"
\end{verbatim}

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]:

```ocaml
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_mono 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 \( m = 0 \land n > 1 \): 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:

```ocaml
corec catalan :: “nat ⇒ nat stream” where
  “catalan n =
  (if n > 0 then ssu"m (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”:

\[
\begin{align*}
\textbf{corec (friend)} \\
\text{sskew :: } &(\text{'a }::\text{ }\{\text{plus,times}\}) \text{ stream } \Rightarrow \text{'a stream } \Rightarrow \text{'a stream} \\
\text{where} \\
\text{"sskew xs ys } = \\
\text{SCons (shd xs } \ast \text{ shd ys) (sskew (sskew xs (stl ys)) (sskew (stl xs) ys))"}
\end{align*}
\]

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:

\[
\begin{align*}
[R \text{ stream stream'}; \land \text{stream stream'} & \Rightarrow \text{shd stream } = \\
\text{shd stream'} \land R \text{ (stl stream) (stl stream')} & \Rightarrow \text{stream } = \text{stream'}
\end{align*}
\]

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 \( \text{sskew} \) as friendly is available as \( \text{sskew.coinduct} \) and as one of the components of the theorem collection stream.coinduct_upto:

\[
\begin{align*}
[R \text{ stream stream'}; \land \text{stream stream'} & \Rightarrow \text{shd stream } = \\
\text{shd stream'} \land \text{stream.v5.congclp R (stl stream) (stl stream')} & \Rightarrow \text{stream } = \text{stream'}
\end{align*}
\]

This rule is almost identical to structural coinduction, except that the corecursive application of \( R \) is generalized to \( \text{stream.v5.congclp R} \).
The \textit{stream.v5.congclp} predicate is equipped with the following introduction rules:

\begin{align*}
\text{sskew.cong\_base:} & \quad P \ x \ y \implies \text{stream.v5.congclp} \ P \ x \ y \\
\text{sskew.cong\_refl:} & \quad x = y \implies \text{stream.v5.congclp} \ R \ x \ y \\
\text{sskew.cong\_sym:} & \quad \text{stream.v5.congclp} \ R \ x \ y \implies \text{stream.v5.congclp} \ R \ y \ x \\
\text{sskew.cong\_trans:} & \quad \left[ \text{stream.v5.congclp} \ R \ x \ y; \text{stream.v5.congclp} \ R \ y \ z \right] \implies \text{stream.v5.congclp} \ R \ x \ z
\end{align*}

\begin{align*}
\text{sskew.cong\_SCons:} & \quad \left[ x1 = y1; \text{stream.v5.congclp} \ R \ x2 \ y2 \right] \implies \text{stream.v5.congclp} \ R \ (\text{SCons} \ x1 \ x2) (\text{SCons} \ y1 \ y2) \\
\text{sskew.cong\_ssum:} & \quad \left[ \text{stream.v5.congclp} \ R \ x1 \ y1; \text{stream.v5.congclp} \ R \ x2 \ y2 \right] \implies \text{stream.v5.congclp} \ R \ (\text{ssum} \ x1 \ x2) (\text{ssum} \ y1 \ y2) \\
\text{sskew.cong\_sprod:} & \quad \left[ \text{stream.v5.congclp} \ R \ x1 \ y1; \text{stream.v5.congclp} \ R \ x2 \ y2 \right] \implies \text{stream.v5.congclp} \ R \ (\text{sprod} \ x1 \ x2) (\text{sprod} \ y1 \ y2) \\
\text{sskew.cong\_sskew:} & \quad \left[ \text{stream.v5.congclp} \ R \ x1 \ y1; \text{stream.v5.congclp} \ R \ x2 \ y2 \right] \implies \text{stream.v5.congclp} \ R \ (\text{sskew} \ x1 \ x2) (\text{sskew} \ y1 \ y2)
\end{align*}

The introduction rules are also available as \textit{sskew.cong\_intros}.

Notice that there is no introduction rule corresponding to \textit{sexp}, because \textit{sexp} has a more restrictive result type than \textit{sskew} (\textit{nat stream} vs. "a stream.

The version numbers, here \textit{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 \textit{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 \textit{ssum} as a friend, proving
the equality $l = r$ on $\texttt{nat stream}$ might require coinduction principle for $\texttt{nat stream}$, which is up to $\texttt{ssum}$.

The collection $\texttt{stream.coinduct}_{\texttt{upto}}$ 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 $\texttt{coinduction}_{\texttt{upto}}$ 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, $\texttt{tllist.coinduct}_{\texttt{upto}}$ contains three variants of the coinduction principles:

- $\texttt{(a, int) tllist}$ up to $\texttt{TNil}$, $\texttt{TCons}$, and $\texttt{square-terminal}$;
- $\texttt{(nat, 'b) tllist}$ up to $\texttt{TNil}$, $\texttt{TCons}$, and $\texttt{square.elems}$;
- $\texttt{(a, 'b) tllist}$ up to $\texttt{TNil}$ and $\texttt{TCons}$.

The following variant is missing:

- $\texttt{(nat, int) tllist}$ up to $\texttt{TNil}$, $\texttt{TCons}$, $\texttt{square.elems}$, and $\texttt{square-terminal}$.

To generate it without having to define a new function with $\texttt{corec}$, 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 $\texttt{tllist.coinduct}_{\texttt{upto}}$ and $\texttt{tllist.cong_intros}$.)
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 \( f\text{.unique} \) 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:

\[
\begin{align*}
  f &= (\lambda xs ys. SCons (shd xs + shd ys) (f (stl xs) (stl ys))) \implies f = \text{ssum} \\
  f &= (\lambda xs ys. SCons (shd xs + shd ys) (f (stl xs) (stl ys))) \implies f = \text{ssum}
\end{align*}
\]

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

\[
\forall x. f x = H x f \implies f x = t x
\]

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 "\forall 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

\texttt{coinduction_upto stream: "a stream"}

3 Command Syntax

3.1 corec and corecursive

\begin{verbatim}
corec : local_theory → local_theory
corecursive : local_theory → proof(prove)
\end{verbatim}

The \texttt{corec} and \texttt{corecursive} commands introduce a corecursive function over a codatatype.

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

The optional target is optionally followed by a combination of the following options:
The \textit{plugins} option indicates which plugins should be enabled (\emph{only}) or disabled (\emph{del}). By default, all plugins are enabled.

- The \textit{friend} option indicates that the defined function should be registered as a friend. This gives rise to additional proof obligations.

- The \textit{transfer} option indicates that an unconditional transfer rule should be generated and proved by \texttt{transfer_prover}. The [\texttt{transfer_rule}] attribute is set on the generated theorem.

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

### 3.2 friend_of_corec

\begin{verbatim}
friend_of_corec : local_theory → proof(prove)
\end{verbatim}

```
friend_of_corec
  target

  fix where prop

  foc-options

() (plugins transfer)
```

The \texttt{friend_of_corec} command registers a corecursive function as friendly.

The syntactic entity \texttt{target} can be used to specify a local context, \texttt{fix} denotes name with an optional type signature, and \texttt{prop} denotes a HOL proposition \cite{8}. 

---

\cite{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

\[
\text{coinduction_upto} : \text{local\_theory} \rightarrow \text{local\_theory}
\]

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):

$f$.\texttt{code} [code]:
\[
sexp \mathit{xs} = S\text{Cons} \, 2^\text{shd} \, \mathit{xs} \, (s\text{prod} \, (\text{stl} \, \mathit{xs}) \, (sexp \, \mathit{xs}))
\]
The [code] attribute is set by the code plugin [1].

$f$.\texttt{coinduct} [consumes 1, case\_names $t$, case\_conclusion $D_1 \ldots D_n$]:
\[
[R \, \text{nat\_stream} \, \text{nat\_stream}'; \land \, \text{nat\_stream} \, \text{nat\_stream}'. \, R \, \text{nat\_stream} \, \text{nat\_stream}'] \Rightarrow \text{shd} \, \text{nat\_stream} = \text{shd} \, \text{nat\_stream}' \land \text{stream.}v3.\text{congclp} \, R \, (\text{stl} \, \text{nat\_stream}) \, (\text{stl} \, \text{nat\_stream}') \Rightarrow \text{nat\_stream} = \text{nat\_stream}'
\]

$f$.\texttt{cong\_intros}:
\[
P \, x \, y \Rightarrow \text{stream.}v3.\text{congclp} \, P \, x \, y
\]
\[
x = y \Rightarrow \text{stream.}v3.\text{congclp} \, R \, x \, y
\]
\[
\text{stream.}v3.\text{congclp} \, R \, x \, y \Rightarrow \text{stream.}v3.\text{congclp} \, R \, y \, x
\]
\[
[\text{stream.}v3.\text{congclp} \, R \, x \, y; \, \text{stream.}v3.\text{congclp} \, R \, y \, z] \Rightarrow \text{stream.}v3.\text{congclp} \, R \, x \, z
\]
\[
[x_1 = y_1; \, \text{stream.}v3.\text{congclp} \, R \, x_2 \, y_2] \Rightarrow \text{stream.}v3.\text{congclp} \, R \, (S\text{Cons} \, x_1 \, x_2) \, (S\text{Cons} \, y_1 \, y_2)
\]
\[
[\text{stream.}v3.\text{congclp} \, R \, x_1 \, y_1; \, \text{stream.}v3.\text{congclp} \, R \, x_2 \, y_2] \Rightarrow \text{stream.}v3.\text{congclp} \, R \, (s\text{sum} \, x_1 \, x_2) \, (s\text{sum} \, y_1 \, y_2)
\]
\[
[\text{stream.}v3.\text{congclp} \, R \, x_1 \, y_1; \, \text{stream.}v3.\text{congclp} \, R \, x_2 \, y_2] \Rightarrow \text{stream.}v3.\text{congclp} \, R \, (s\text{prod} \, x_1 \, x_2) \, (s\text{prod} \, y_1 \, y_2)
\]
\[
\text{stream.}v3.\text{congclp} \, R \, x \, y \Rightarrow \text{stream.}v3.\text{congclp} \, R \, (sexp \, x) \, (sexp \, y)
\]

$f$.\texttt{unique}:
\[
f = (\lambda \mathit{xs}. \, S\text{Cons} \, 2^\text{shd} \, \mathit{xs} \, (s\text{prod} \, (\text{stl} \, \mathit{xs}) \, (f \, \mathit{xs}))) \Rightarrow f = sexp
\]
This property is not generated for mixed recursive–corecursive definitions.

$f$.\texttt{inner\_induct}:

This property is only generated for mixed recursive–corecursive definitions. For primes (Section 2.3, it reads as follows:
\[
(\forall m \, n. \, (\forall x \, y. \, [(x, \, y) = (m, \, n); \, \neg \, (x = 0 \land 1 < y \lor \text{coprime} \, x \, y)] \Rightarrow P \, (x, \, y + 1)) \Rightarrow P \, (m, \, n)) \Rightarrow P \, a0
\]

The individual rules making up $f$.\texttt{cong\_intros} are available as

$f$.\texttt{cong\_base}

$f$.\texttt{cong\_refl}
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`):

\[
\text{t.coinduct_upto} \quad \text{[consumes 1, case_names t,}
\text{ case_conclusion D}_1 \ldots D_n];
\]

\[
[R \text{nat_int_tllist nat_int_tllist'}; \text{nat_int_tllist nat_int_tllist'} \Rightarrow \text{is_TNil nat_int_tllist} = \text{is_TNil nat_int_tllist'} \wedge (\text{is_TNil nat_int_tllist} \Rightarrow \text{is_TNil nat_int_tllist'})
\]

\[
\Rightarrow \text{terminal nat_int_tllist = terminal nat_int_tllist'} \wedge (\neg \text{is_TNil nat_int_tllist} \Rightarrow \neg \text{is_TNil nat_int_tllist'})
\]

\[
\Rightarrow \text{thd nat_int_tllist = thd nat_int_tllist'}
\]

\[
\Rightarrow \text{tllist.v3.congclp R (tll nat_int_tllist) (tll nat_int_tllist')}
\]

\[
\Rightarrow \text{nat_int_tllist = nat_int_tllist'}
\]
The individual rules making up \textit{t.cong_intros} are available separately as \textit{t.cong_base}, \textit{t.cong_refl}, etc. (Section 4.1).

5 Proof Methods

5.1 \textit{corec\_unique}

The \textit{corec\_unique} proof method can be used to prove the uniqueness of a corecursive specification. See Section 2.6 for details.

5.2 \textit{transfer\_prover\_eq}

The \textit{transfer\_prover\_eq} proof method replaces the equality relation (=) with compound relator expressions according to \textit{relator\_eq} before calling \textit{transfer\_prover} on the current subgoal. It tends to work better than plain \textit{transfer\_prover} on the parametricity proof obligations of \textit{corecursive} and \textit{friend\_of\_corec}, because they often contain equality relations on complex types, which \textit{transfer\_prover} cannot reason about.

6 Attribute

6.1 \textit{friend\_of\_corec\_simps}

The \textit{friend\_of\_corec\_simps} attribute declares naturality theorems to be used by \textit{friend\_of\_corec} and \textit{corec (friend)} in deriving the user specification from reduction to primitive corecursion. Internally, these commands derive naturality theorems from the parametricity proof obligations discharged by the user or the \textit{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 \textit{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


