Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL

Jasmin Christian Blanchette, Martin Desharnais, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel

25 May 2015

Abstract

This tutorial describes the definitional package for datatypes and codatatypes, and for primitively recursive and corecursive functions, in Isabelle/HOL. The package provides these commands: \texttt{datatype}, \texttt{datatype_compat}, \texttt{primrec}, \texttt{codatatype}, \texttt{primcorec}, \texttt{primcorecursive}, \texttt{bnf}, \texttt{bnf_axiomatization}, \texttt{print_bnf}, and \texttt{free_constructors}.

Contents

1 Introduction 3

2 Defining Datatypes 5
  2.1 Introductory Examples ........................................... 5
     2.1.1 Nonrecursive Types ......................................... 5
     2.1.2 Simple Recursion ........................................... 6
     2.1.3 Mutual Recursion ........................................... 6
     2.1.4 Nested Recursion ........................................... 6
     2.1.5 Auxiliary Constants ...................................... 7
  2.2 Command Syntax .................................................. 9
     2.2.1 \texttt{datatype} ............................................. 9
     2.2.2 \texttt{datatype_compat} ..................................... 12
  2.3 Generated Constants ............................................. 13
  2.4 Generated Theorems ............................................. 13
     2.4.1 Free Constructor Theorems ................................ 14
     2.4.2 Functorial Theorems ....................................... 16
     2.4.3 Inductive Theorems ....................................... 19
2.5 Compatibility Issues

3 Defining Primitively Recursive Functions
  3.1 Introductory Examples
    3.1.1 Nonrecursive Types
    3.1.2 Simple Recursion
    3.1.3 Mutual Recursion
    3.1.4 Nested Recursion
    3.1.5 Nested-as-Mutual Recursion
  3.2 Command Syntax
  3.3 Generated Theorems
  3.4 Recursive Default Values for Selectors
  3.5 Compatibility Issues

4 Defining Codatatypes
  4.1 Introductory Examples
    4.1.1 Simple Corecursion
    4.1.2 Mutual Corecursion
    4.1.3 Nested Corecursion
  4.2 Command Syntax
  4.3 Generated Constants
  4.4 Generated Theorems
  4.4.1 Coinductive Theorems

5 Defining Primitively Corecursive Functions
  5.1 Introductory Examples
    5.1.1 Simple Corecursion
    5.1.2 Mutual Corecursion
    5.1.3 Nested Corecursion
    5.1.4 Nested-as-Mutual Corecursion
    5.1.5 Constructor View
    5.1.6 Destructor View
  5.2 Command Syntax
    5.2.1 primcorec and primcorecursive
  5.3 Generated Theorems

6 Registering Bounded Natural Functors
  6.1 Bounded Natural Functors
  6.2 Introductory Examples
1 Introduction

The 2013 edition of Isabelle introduced a definitional package for freely generated datatypes and codatatypes. This package replaces the earlier implementation due to Berghofer and Wenzel [1]. Perhaps the main advantage of the new package is that it supports recursion through a large class of non-datatypes, such as finite sets:

\[
\text{datatype } 'a \text{ tree}_{fs} = \text{Node}_{fs} (\text{lbl}_{fs}: 'a) (\text{sub}_{fs}: "'a tree_{fs} fset")
\]

Another strong point is the support for local definitions:

\[
\text{context linorder}
\begin{align*}
\text{begin} \\
\text{datatype flag} &= \text{Less} | \text{Eq} | \text{Greater}
\end{align*}
\]

Furthermore, the package provides a lot of convenience, including automatically generated discriminators, selectors, and relators as well as a wealth of properties about them.

In addition to inductive datatypes, the package supports coinductive datatypes, or codatatypes, which allow infinite values. For example, the following command introduces the type of lazy lists, which comprises both finite and infinite values:
Mixed inductive–coinductive recursion is possible via nesting. Compare the following four Rose tree examples:

datatype 'a treelff = Nodeff 'a ""'a treelff list"
datatype 'a treefi = Nodefi 'a ""'a treefi list"
codataatatype 'a treeff = Nodeeff 'a ""'a treelff list"
codataatatype 'a treeifi = Nodeifi 'a ""'a treeifi list"

The first two tree types allow only paths of finite length, whereas the last two allow infinite paths. Orthogonally, the nodes in the first and third types have finitely many direct subtrees, whereas those of the second and fourth may have infinite branching.

The package is part of Main. Additional functionality is provided by the theory BNF_Axiomatization, located in the directory ~/src/HOL/Library.

The package, like its predecessor, fully adheres to the LCF philosophy [4]: The characteristic theorems associated with the specified (co)datatypes are derived rather than introduced axiomatically.\footnote{However, some of the internal constructions and most of the internal proof obligations are omitted if the quick_and_dirty option is enabled.} The package is described in a number of papers [2, 3, 7, 8]. The central notion is that of a \textit{bounded natural functor} (BNF)—a well-behaved type constructor for which nested (co)recursion is supported.

This tutorial is organized as follows:

- Section 2, “Defining Datatypes,” describes how to specify datatypes using the \texttt{datatype} command.
- Section 3, “Defining Primitively Recursive Functions,” describes how to specify functions using \texttt{primrec}. (A separate tutorial [5] describes the more general \texttt{fun} and \texttt{function} commands.)
- Section 4, “Defining Codatatypes,” describes how to specify codatatypes using the \texttt{codatatype} command.
- Section 5, “Defining Primitively Corecursive Functions,” describes how to specify functions using the \texttt{primcorec} and \texttt{primcorecursive} commands.
- Section 6, “Registering Bounded Natural Functors,” explains how to use the \texttt{bnf} command to register arbitrary type constructors as BNFs.
- Section 7, “Deriving Destructors and Theorems for Free Constructors,” explains how to use the command \texttt{free_constructors} to derive destructor constants and theorems for freely generated types, as performed internally by \texttt{datatype} and \texttt{codatatype}.
• Section 8, “Selecting Plugins,” is concerned with the package’s interoperability with other Isabelle packages and tools, such as the code generator, Transfer, Lifting, and Quickcheck.

• Section 9, “Known Bugs and Limitations,” concludes with known open issues at the time of writing.

Comments and bug reports concerning either the package or this tutorial should be directed to the authors at blanchette@in.tum.de, desharna@in.tum.de, lorenz.panny@in.tum.de, popescua@in.tum.de, and traytel@in.tum.de.

2 Defining Datatypes

Datatypes can be specified using the `datatype` command.

2.1 Introductory Examples

Datatypes are illustrated through concrete examples featuring different flavors of recursion. More examples can be found in the directory `~/src/HOL/Datatype_Examples`.

2.1.1 Nonrecursive Types

Datatypes are introduced by specifying the desired names and argument types for their constructors. `Enumeration` types are the simplest form of datatype. All their constructors are nullary:

```
datatype trool = True | False | Perhaps
```

`True`, `False`, and `Perhaps` have the type `trool`.

Polymorphic types are possible, such as the following option type, modeled after its homologue from the `Option` theory:

```
datatype 'a option = None | Some 'a
```

The constructors are `None :: 'a option` and `Some :: 'a ⇒ 'a option`.

The next example has three type parameters:

```
datatype ('a, 'b, 'c) triple = Triple 'a 'b 'c
```

The constructor is `Triple :: 'a ⇒ 'b ⇒ 'c ⇒ ('a, 'b, 'c) triple`. Unlike in Standard ML, curried constructors are supported. The uncurried variant is also possible:
datatype 
\( (\textit{a}', \textit{b}', \textit{c}') \) triple \( _u = \text{Triple}_u \text{"a}' \text{'}b \text{'}c" \)

Occurrences of nonatomic types on the right-hand side of the equal sign must be enclosed in double quotes, as is customary in Isabelle.

### 2.1.2 Simple Recursion

Natural numbers are the simplest example of a recursive type:

**datatype** \( \textit{nat} = \text{Zero} \mid \text{Succ} \textit{nat} \)

Lists were shown in the introduction. Terminated lists are a variant that stores a value of type \( \textit{b} \) at the very end:

**datatype** \( (\textit{a}', \textit{b}') \textit{tlist} = \text{TNil}'\textit{b} \mid \text{TCons}'(\textit{a}', \textit{b}') \textit{tlist}" \)

### 2.1.3 Mutual Recursion

Mutually recursive types are introduced simultaneously and may refer to each other. The example below introduces a pair of types for even and odd natural numbers:

**datatype** \( \textit{even\_nat} = \text{Even\_Zero} \mid \text{Even\_Succ} \textit{odd\_nat} \)

and \( \textit{odd\_nat} = \text{Odd\_Succ} \textit{even\_nat} \)

Arithmetic expressions are defined via terms, terms via factors, and factors via expressions:

**datatype** \( (\textit{a}', \textit{b}') \textit{exp} = \text{Term} \text{"'(a}', \textit{b}') \textit{trm}" \mid \text{Sum} \text{"'(a}', \textit{b}') \textit{trm}" \text{"'(a}', \textit{b}') \textit{exp}" \)

and \( (\textit{a}', \textit{b}') \textit{trm} = \text{Factor} \text{"'(a}', \textit{b}') \textit{fct}" \mid \text{Prod} \text{"'(a}', \textit{b}') \textit{fct}" \text{"'(a}', \textit{b}') \textit{trm}" \)

and \( (\textit{a}', \textit{b}') \textit{fct} = \text{Const}'\textit{a} \mid \text{Var}'\textit{b} \mid \text{Expr} \text{"'(a}', \textit{b}') \textit{exp}" \)

### 2.1.4 Nested Recursion

Nested recursion occurs when recursive occurrences of a type appear under a type constructor. The introduction showed some examples of trees with nesting through lists. A more complex example, that reuses our \( \textit{option} \) type, follows:

**datatype** \( \textit{a} \textit{btree} = \text{BNode} \text{"'(a} \textit{btree option}" \text{"'(a} \textit{btree option}" \)

Not all nestings are admissible. For example, this command will fail:

**datatype** \( \textit{a} \textit{wrong} = \text{W}1 \mid \text{W}2 \text{"'(a} \textit{wrong} \Rightarrow \text{'}a"} \)
The issue is that the function arrow \( \Rightarrow \) allows recursion only through its right-hand side. This issue is inherited by polymorphic datatypes defined in terms of \( \Rightarrow \):

```ml
datatype ('a, 'b) fun_copy = Fun "'a => 'b"
datatype 'a also_wrong = W1 | W2 "('a also_wrong, 'a) fun_copy"
```

The following definition of \('a\)-branching trees is legal:

```ml
datatype 'a ftree = FTLeaf 'a | FTNode "'a => 'a ftree"
```

And so is the definition of hereditarily finite sets:

```ml
datatype hfset = HFSet "hfset fset"
```

In general, type constructors \((\forall a_1, \ldots, a_m) t\) allow recursion on a subset of their type arguments \(a_1, \ldots, a_m\). These type arguments are called live; the remaining type arguments are called dead. In \(t \Rightarrow b\) and \((a, b) fun_copy\), the type variable \(a\) is dead and \(b\) is live.

Type constructors must be registered as BNFs to have live arguments. This is done automatically for datatypes and codatatypes introduced by the `datatype` and `codatatype` commands. Section 6 explains how to register arbitrary type constructors as BNFs.

Here is another example that fails:

```ml
datatype 'a pow_list = PNil 'a | PCons "('a * 'a) pow_list"
```

This attempted definition features a different flavor of nesting, where the recursive call in the type specification occurs around (rather than inside) another type constructor.

### 2.1.5 Auxiliary Constants

The `datatype` command introduces various constants in addition to the constructors. With each datatype are associated set functions, a map function, a relator, discriminators, and selectors, all of which can be given custom names. In the example below, the familiar names `null`, `hd`, `tl`, `set`, `map`, and `list_all2` override the default names `is.Nil`, `un.Cons1`, `un.Cons2`, `set.list`, `map.list`, and `rel.list`:

```ml
datatype (set: 'a) list =
  null: Nil
| Cons (hd: 'a) (tl: "'a list")
for
  map: map
  rel: list_all2
where
```
"tl Nil = Nil"

The types of the constants that appear in the specification are listed below.

Constructors:  
   Nil :: 'a list  
   Cons :: 'a ⇒ 'a list ⇒ 'a list

Discriminator:  
   null :: 'a list ⇒ bool

Selectors:  
   hd :: 'a list ⇒ 'a  
   tl :: 'a list ⇒ 'a list

Set function:  
   set :: 'a list ⇒ 'a set

Map function:  
   map :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list

Relator:  
   list_all2 :: ('a ⇒ 'b ⇒ bool) ⇒ 'a list ⇒ 'b list ⇒ bool

The discriminator null and the selectors hd and tl are characterized by

the following conditional equations:

null xs ⇒ xs = Nil  ¬ null xs ⇒ Cons (hd xs) (tl xs) = xs

For two-.constructor datatypes, a single discriminator constant is sufficient.
The discriminator associated with Cons is simply λxs. ¬ null xs.

The where clause at the end of the command specifies a default value
for selectors applied to constructors on which they are not a priori specified.
In the example, it is used to ensure that the tail of the empty list is itself
(instead of being left unspecified).

Because Nil is nullary, it is also possible to use λxs. xs = Nil as a dis-

criminator. This is the default behavior if we omit the identifier null and
the associated colon. Some users argue against this, because the mixture
of constructors and selectors in the characteristic theorems can lead Isabelle’s
automation to switch between the constructor and the destructor view in
surprising ways.

The usual mixfix syntax annotations are available for both types and
constructors. For example:

datatype ('a,'b) prod (infixr "∗" 20) = Pair 'a 'b

datatype (set: 'a) list =
   null: Nil ("[]")
   | Cons (hd: 'a) (tl: "'a list") (infixr "#" 65)

Incidentally, this is how the traditional syntax can be set up:

syntax "_list" :: "args ⇒ 'a list" ("[(_)]")

translations
"[x, xs]" == "x \# [xs]"
"[x]" == "x \# []"

2.2 Command Syntax
2.2.1 datatype

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

\[
\text{datatype}
\quad \text{target}
\quad \text{dt-options}
\quad \text{dt-spec}
\]

\text{dt-options}

\[
\text{plugins}
\quad \text{discs\_sels}
\]

\text{plugins}

\[
\text{plugins}
\quad \text{only}
\quad \text{del}
\quad \text{name}
\]
The \textbf{datatype} command introduces a set of mutually recursive datatypes specified by their constructors.

The syntactic entity \textit{target} can be used to specify a local context (e.g., (in \textit{linorder}) \cite{9}), and \textit{prop} denotes a HOL proposition.

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

- The \textit{plugins} option indicates which plugins should be enabled (\textit{only}) or disabled (\textit{del}). By default, all plugins are enabled.
- The \textit{discs\_sels} option indicates that discriminators and selector should be generated. The option is implicitly enabled if names are specified for discriminators or selectors.

The optional \textbf{where} clause specifies default values for selectors. Each proposition must be an equation of the form \textit{un\_D} \((C \ldots) = \ldots\), where \(C\) is a constructor and \textit{un\_D} is a selector.

The left-hand sides of the datatype equations specify the name of the type to define, its type parameters, and additional information:
2  Defining Datatypes

\textit{dt-name}

\begin{center}
\begin{tikzpicture}
    \node [circle, fill=black, inner sep=1pt] (n1) at (0,0) {name};
    \node [circle, fill=black, inner sep=1pt] (n2) at (-1.5,-1.5) {tyargs};
    \node [circle, fill=black, inner sep=1pt] (n3) at (1.5,-1.5) {mixfix};

    \draw [->] (n1) -- (n2);
    \draw [->] (n1) -- (n3);
\end{tikzpicture}
\end{center}

\textit{tyargs}

\begin{center}
\begin{tikzpicture}
    \node [circle, fill=black, inner sep=1pt] (n1) at (0,0) {typefree};
    \node [circle, fill=black, inner sep=1pt] (n2) at (-1.5,-1.5) {();};
    \node [circle, fill=black, inner sep=1pt] (n3) at (1.5,-1.5) {typefree};

    \draw [->] (n1) -- (n2);
    \draw [->] (n1) -- (n3);
    \draw [->] (n2) -- (n3);
\end{tikzpicture}
\end{center}

The syntactic entity \textit{name} denotes an identifier, \textit{mixfix} denotes the usual parenthesized mixfix notation, and \textit{typefree} denotes fixed type variable (\texttt{'a}, \texttt{'b}, \ldots) [9].

The optional names preceding the type variables allow to override the default names of the set functions (\texttt{set}_1, \ldots, \texttt{set}_m). Type arguments can be marked as dead by entering \texttt{dead} in front of the type variable (e.g., \texttt{(dead 'a)}); otherwise, they are live or dead (and a set function is generated or not) depending on where they occur in the right-hand sides of the definition. Declaring a type argument as dead can speed up the type definition but will prevent any later (co)recursion through that type argument.

Inside a mutually recursive specification, all defined datatypes must mention exactly the same type variables in the same order.

\textit{dt-ctor}

\begin{center}
\begin{tikzpicture}
    \node [circle, fill=black, inner sep=1pt] (n1) at (0,0) {name};
    \node [circle, fill=black, inner sep=1pt] (n2) at (-1.5,-1.5) {name};
    \node [circle, fill=black, inner sep=1pt] (n3) at (1.5,-1.5) {mixfix};

    \draw [->] (n1) -- (n2);
    \draw [->] (n1) -- (n3);
    \draw [->] (n2) -- (n3);
\end{tikzpicture}
\end{center}

The main constituents of a constructor specification are the name of the constructor and the list of its argument types. An optional discriminator name can be supplied at the front. If discriminators are enabled (cf. the
2 Defining Datatypes

discs_sels option) but no name is supplied, the default is $\lambda x. \ x = C_j$ for
nullary constructors and $t.is\ C_j$ otherwise.

dt-ctor-arg

The syntactic entity type denotes a HOL type [9].

In addition to the type of a constructor argument, it is possible to specify a
name for the corresponding selector. The same selector name can be reused
for arguments to several constructors as long as the arguments share the
same type. If selectors are enabled (cf. the discs_sels option) but no name
is supplied, the default name is \textit{un}_C_ji.

2.2.2 datatype_compat

\textbf{datatype_compat} : \textit{local\_theory} \rightarrow \textit{local\_theory}

The \textbf{datatype_compat} command registers new-style datatypes as old-style
datatypes and invokes the old-style plugins. For example:

\texttt{datatype_compat even_nat odd_nat辖}

\texttt{ML \{/* Old_Datatype_Data.get_info \theory \{\theory_name even_nat\} */\}}

The syntactic entity name denotes an identifier [9].

The command is sometimes useful when migrating from the old datatype
package to the new one.

A few remarks concern nested recursive datatypes:

- The old-style, nested-as-mutual induction rule and recursor theorems
  are generated under their usual names but with “compat_” prefixed
  (e.g., compat_tree.induct, compat_tree.inducts, and compat_tree.rec).
- All types through which recursion takes place must be new-style data-
types or the function type.
2.3 Generated Constants

Given a datatype \( (a_1, \ldots, a_m) \ t \) with \( m \) live type variables and \( n \) constructors \( t.C_1, \ldots, t.C_n \), the following auxiliary constants are introduced:

- **Case combinator**: \( t.\text{case\_t} \) (rendered using the familiar \texttt{case-of} syntax)
- **Discriminators**: \( t.\text{is\_C}_1, \ldots, t.\text{is\_C}_n \)
- **Selectors**: \( t.\text{un\_C}_1 k_1, \ldots, t.\text{un\_C}_n k_n \)
- **Set functions**: \( t.\text{set}_1 t, \ldots, t.\text{set}_m t \)
- **Map function**: \( t.\text{map}_t \)
- **Relator**: \( t.\text{rel}_t \)
- **Recursor**: \( t.\text{rec}_t \)

The discriminators and selectors are generated only if the \texttt{discs\_sels} option is enabled or if names are specified for discriminators or selectors. The set functions, map function, and relator are generated only if \( m > 0 \).

In addition, some of the plugins introduce their own constants (Section 8). The case combinator, discriminators, and selectors are collectively called *destructors*. The prefix “\( t. \)” is an optional component of the names and is normally hidden.

2.4 Generated Theorems

The characteristic theorems generated by \texttt{datatype} are grouped in three broad categories:

- **The free constructor theorems** (Section 2.4.1) are properties of the constructors and destructors that can be derived for any freely generated type. Internally, the derivation is performed by \texttt{free\_constructors}.
- **The functorial theorems** (Section 2.4.2) are properties of datatypes related to their BNF nature.
- **The inductive theorems** (Section 2.4.3) are properties of datatypes related to their inductive nature.

The full list of named theorems can be obtained as usual by entering the command \texttt{print\_theorems} immediately after the datatype definition. This list includes theorems produced by plugins (Section 8), but normally excludes low-level theorems that reveal internal constructions. To make these accessible, add the line

\[ \text{declare } [[\text{bnf\_note\_all}]] \]
to the top of the theory file.

### 2.4.1 Free Constructor Theorems

The free constructor theorems are partitioned in three subgroups. The first subgroup of properties is concerned with the constructors. They are listed below for 'a list:

**t.inject** [iff, induct_simp]:

\[(x21 \# x22 = y21 \# y22) = (x21 = y21 \land x22 = y22)\]

**t.distinct** [simp, induct_simp]:

\[
\begin{align*}
\& [\neq x21 \# x22 \\
\& x21 \# x22 \neq [] \\
\end{align*}
\]

**t.exhaust** [cases t, case_names C1 . . . Cn]:

\[
[y = []; \land x21 x22. y = x21 \# x22 \rightarrow P] \rightarrow P
\]

**t.nchotony**:

\[
\forall \text{list}. \text{list} = [] \lor (\exists \; x21 x22. \text{list} = x21 \# x22)
\]

In addition, these nameless theorems are registered as safe elimination rules:

**t.distinct** [THEN notE, elim!]:

\[
\begin{align*}
\& [] = x21 \# x22 \rightarrow R \\
\& x21 \# x22 = [] \rightarrow R
\end{align*}
\]

The next subgroup is concerned with the case combinator:

**t.case** [simp, code]:

\[
\begin{align*}
&(\text{case } [] \text{ of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 x xa) = f1 \\
&(\text{case } x21 \# x22 \text{ of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 x xa) = f2 x21 x22
\end{align*}
\]

The [code] attribute is set by the code plugin (Section 8.1).

**t.case_cong** [fundef_cong]:

\[
\begin{align*}
&(\text{case list of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 x xa) = f1 \\
&(\text{case list of } x21 x22 \Rightarrow g1 \mid x21 x22 \Rightarrow g2 x21 x22) = (\text{case list' of } [] \Rightarrow g1 \mid x21 x22 \Rightarrow g2 x21 x22)
\end{align*}
\]

**t.case_cong_weak** [cong]:

\[
\begin{align*}
&(\text{case list' of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 x xa) = (\text{case list' of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 x xa)
\end{align*}
\]

**t.case_distrib**:  

\[
\begin{align*}
&h (\text{case list of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 x xa) = (\text{case list of } [] \Rightarrow h \\
&f1 \mid x1 \# x2 \Rightarrow h (f2 x1 x2))
\end{align*}
\]


2 Defining Datatypes

`t.split`:

\[ P \left( \text{case list of } \emptyset \Rightarrow f_1 \mid x \neq xa \Rightarrow f_2 x xa \right) = ((\text{list} = \emptyset \rightarrow P f_1) \wedge (\forall x_1 \ x_2. \ \text{list} = x_1 \neq x_2 \rightarrow P (f_2 x_1 \ x_2))) \]

`t.split_asm`:

\[ P \left( \text{case list of } \emptyset \Rightarrow f_1 \mid x \neq xa \Rightarrow f_2 x xa \right) = (\neg (\text{list} = \emptyset \wedge \neg P f_1) \vee (\exists x_1 \ x_2. \ \text{list} = x_1 \neq x_2 \wedge \neg P (f_2 x_1 \ x_2))) \]

`t.splits = split split_asm`

The third subgroup revolves around discriminators and selectors:

`t.disc [simp]`:

null \emptyset

\(- \text{null} (x_1 \neq x_2)\)

`t.discI`:

\(\text{list} = \emptyset \implies \text{null list}\)

\(\text{list} = x_1 \neq x_2 \implies \neg \text{null list}\)

`t.sel [simp, code]`:

hd \(x_1 \neq x_2) = x_1\)

\(tl (x_1 \neq x_2) = x_2\)

The [code] attribute is set by the code plugin (Section 8.1).

`t.collapse [simp]`:

\(\text{null list} \implies \text{list} = \emptyset\)

\(\neg \text{null list} \implies \text{hd list} \neq \text{tl list} = \text{list}\)

The [simp] attribute is exceptionally omitted for datatypes equipped with a single nullary constructor, because a property of the form \(x = C\) is not suitable as a simplification rule.

`t.distinct_disc [dest]`:

These properties are missing for `a list` because there is only one proper discriminator. If the datatype had been introduced with a second discriminator called `nonnull`, they would have read thusly:

\(\text{null list} \implies \neg \text{nonnull list}\)

\(\neg \text{nonnull list} \implies \neg \text{null list}\)

`t.exhaust_disc [case_names C_1 \ldots C_n]`:

\([\text{null list} \implies P; \neg \text{null list} \implies P] \implies P\)

`t.exhaust_sel [case_names C_1 \ldots C_n]`:

\([\text{list} = \emptyset \implies P; \text{list} = \text{hd list} \neq \text{tl list} \implies P] \implies P\)

`t.expand`:

\([\text{null list} = \text{null list}'; \neg \neg \text{null list}; \neg \neg \text{null list}'] \implies \text{hd list} = \text{hd list}'\)

\(\neg \text{null list} \implies \text{tl list} = \text{tl list}'\)

\(\implies \text{list} = \text{list}'\)
2 Defining Datatypes

\textit{t.split\_sel}:
\[ \begin{align*}
P \left( \text{case list of } [] \Rightarrow f1 \mid x \neq xa \Rightarrow f2 \, x \, xa \right) &= ((list = [] \rightarrow P \, f1) \wedge (list = \text{hd list} \neq \text{tl list} \rightarrow P \, (f2 \, (\text{hd list}) \, (\text{tl list})))) \end{align*} \]

\textit{t.split\_sel\_asm}:
\[ \begin{align*}
P \left( \text{case list of } [] \Rightarrow f1 \mid x \neq xa \Rightarrow f2 \, x \, xa \right) &= (\neg (list = [] \wedge \neg P \, f1 \lor list = \text{hd list} \neq \text{tl list} \wedge \neg P \, (f2 \, (\text{hd list}) \, (\text{tl list})))) \end{align*} \]

\textit{t.split\_sels} = \text{split\_sel split\_sel\_asm}

\textit{t.case\_eq\_if}:
\[ \begin{align*}
(\text{case list of } [] \Rightarrow f1 \mid x \neq xa \Rightarrow f2 \, x \, xa) &= (\text{if null list then } f1 \text{ else } f2 \, (\text{hd list}) \, (\text{tl list}))
\end{align*} \]

\textit{t.disc\_eq\_case}:
\[ \begin{align*}
\text{null list} &= (\text{case list of } [] \Rightarrow True \mid uu\_ \neq wua_ \Rightarrow False) \\
(\neg \text{null list}) &= (\text{case list of } [] \Rightarrow False \mid uu\_ \neq wua_ \Rightarrow True)
\end{align*} \]

In addition, equational versions of \textit{t.disc} are registered with the [\texttt{code}] attribute. The [\texttt{code}] attribute is set by the \texttt{code} plugin (Section 8.1).

### 2.4.2 Functorial Theorems

The functorial theorems are partitioned in two subgroups. The first subgroup consists of properties involving the constructors or the destructors and either a set function, the map function, or the relator:

\textit{t.case\_transfer} [\texttt{transfer\_rule}]:
\[ \begin{align*}
\text{rel\_fun } S \ (\text{rel\_fun } (\text{rel\_fun } R \ (\text{rel\_fun } (\text{list\_all2 } R) \ S)) \ (\text{rel\_fun } (\text{list\_all2 } R) \ S)) \text{ case\_list case\_list}
\end{align*} \]

The [\texttt{transfer\_rule}] attribute is set by the \texttt{transfer} plugin (Section 8.3) for type constructors with no dead type arguments.

\textit{t.sel\_transfer} [\texttt{transfer\_rule}]:
This property is missing for \texttt{'a list} because there is no common selector to all constructors.

The [\texttt{transfer\_rule}] attribute is set by the \texttt{transfer} plugin (Section 8.3) for type constructors with no dead type arguments.

\textit{t.ctr\_transfer} [\texttt{transfer\_rule}]:
\[ \begin{align*}
\text{list\_all2 } R \ [\ []
\end{align*} \]

\[ \begin{align*}
\text{rel\_fun } R \ (\text{rel\_fun } (\text{list\_all2 } R) \ (\text{list\_all2 } R)) \ \text{op} \ # \ \text{op} \ #
\end{align*} \]

The [\texttt{transfer\_rule}] attribute is set by the \texttt{transfer} plugin (Section 8.3) for type constructors with no dead type arguments.
2 Defining Datatypes

\begin{itemize}
  \item \texttt{t.disc_transfer [transfer_rule]}:
    \[
      \text{rel\_fun (list\_all2 R \ op = null null} \quad \text{rel\_fun (list\_all2 R \ op = (\lambda \text{list.} \neg \ \text{null list}) \ (\lambda \text{list.} \neg \ \text{null list})} \quad \\
    \text{The [transfer\_rule] attribute is set by the transfer plugin (Section 8.3) for type constructors with no dead type arguments.}
  \end{itemize}

\begin{itemize}
  \item \texttt{t.set [simp, code]}:
    \[
      \text{set \[] = \{\}} \quad \text{set (x21 \# x22) = insert x21 (set x22)} \quad \\
    \text{The [code] attribute is set by the code plugin (Section 8.1).}
  \end{itemize}

\begin{itemize}
  \item \texttt{t.set\_cases [consumes 1, cases set: set\_t]}:
    \[
      \text{[} e \in \text{set a}; \bigwedge \text{z2}. a = e \# z2 \implies \text{thesis}; \bigwedge \text{z1 z2}. [a = z1 \# z2; e} \in \text{set z2] \implies \text{thesis] \implies \text{thesis}}
    \end{itemize}

\begin{itemize}
  \item \texttt{t.set\_intros:}
    \[
      a1 \in \text{set (a1 \# a2)} \quad x \in \text{set a2} \implies x \in \text{set (a1 \# a2)}
    \end{itemize}

\begin{itemize}
  \item \texttt{t.set\_sel:}
    \[
      \neg \text{null a} \implies \text{hd a} \in \text{set a} \quad \\
    \neg \text{null a; x} \in \text{set (tl a)}] \implies x \in \text{set a}
    \end{itemize}

\begin{itemize}
  \item \texttt{t.map [simp, code]}:
    \[
      \text{map f \[] = \[]} \quad \text{map f (x21 \# x22) = f x21 \# \text{map f x22}} \quad \\
    \text{The [code] attribute is set by the code plugin (Section 8.1).}
  \end{itemize}

\begin{itemize}
  \item \texttt{t.map\_disc\_iff [simp]}:
    \[
      \text{null (map f a) = null a}
    \end{itemize}

\begin{itemize}
  \item \texttt{t.map\_sel:}
    \[
      \neg \text{null a} \implies \text{hd (map f a) = f (hd a)} \quad \\
    \neg \text{null a} \implies \text{tl (map f a) = map f (tl a)}
    \end{itemize}

\begin{itemize}
  \item \texttt{t.rel\_inject [simp]}:
    \[
      \text{list\_all2 R \[] \[]} \quad \\
    \text{list\_all2 R (x21 \# x22) (y21 \# y22) = (R x21 y21 \land \text{list\_all2 R x22 y22})}
    \end{itemize}

\begin{itemize}
  \item \texttt{t.rel\_distinct [simp]}:
    \[
      \neg \text{list\_all2 R \[] (y21 \# y22)} \quad \\
    \neg \text{list\_all2 R (y21 \# y22) \[]}
    \end{itemize}

\begin{itemize}
  \item \texttt{t.rel\_intros:}
    \[
      \text{list\_all2 R \[] \[]} \quad \\
    [R x21 y21; \text{list\_all2 R x22 y22}] \implies \text{list\_all2 R (x21 \# x22) (y21 \# y22)}
    \end{itemize}
t.rel_cases [consumes 1, case_names t_1 ... t_m, cases pred]:
\[
\begin{align*}
\text{list_all2} R \ a \ b; \ [a = []; \ b = []] \Rightarrow \text{thesis}; \\
\land x_1 x_2 y_1 y_2; \ [a = x_1 \ # \ x_2; \ b = y_1 \ # \ y_2; \ R \ x_1 y_1; \ \text{list_all2} R \ x_2 y_2] \Rightarrow \text{thesis} & \Rightarrow \text{thesis} \\
\end{align*}
\]

In addition, equational versions of \textit{t.rel_inject} and \textit{rel_distinct} are registered with the \texttt{[code]} attribute. The \texttt{[code]} attribute is set by the \texttt{code} plugin (Section 8.1).

The second subgroup consists of more abstract properties of the set functions, the map function, and the relator:

\textit{t.inj_map}:
\[
\text{inj} \ f \Rightarrow \text{inj} \ (\text{map} \ f)
\]

\textit{t.inj_map_strong}:
\[
\forall z. \ za. \ [z \in \text{set} \ x; \ za \in \text{set} \ xa; \ f \ z = fa \ za] \Rightarrow z = za; \ \text{map} \ f \ x = map \ fa \ xa \Rightarrow x = xa
\]

\textit{t.set_map}:
\[
\text{set} \ (\text{map} \ f \ v) = f \cdot \text{set} \ v
\]

\textit{t.set_transfer} [transfer_rule]:
\[
\text{rel_fun} \ (\text{list_all2} R) \ (\text{rel_set} R) \ \text{set} \ \text{set}
\]
The \texttt{[transfer_rule]} attribute is set by the \texttt{transfer} plugin (Section 8.3) for type constructors with no dead type arguments.

\textit{t.map_cong0}:
\[
(\forall \ z. \ z \in \text{set} \ x \Rightarrow f \ z = g \ z) \Rightarrow \text{map} \ f \ x = \text{map} \ g \ x
\]

\textit{t.map_cong} [fundef_cong]:
\[
[x = y; \ \forall \ z. \ z \in \text{set} \ y \Rightarrow f \ z = g \ z] \Rightarrow \text{map} \ f \ x = \text{map} \ g \ y
\]

\textit{t.map_cong_simp}:
\[
[x = y; \ \forall \ z. \ z \in \text{set} \ y = \text{simp}\Rightarrow f \ z = g \ z] \Rightarrow \text{map} \ f \ x = \text{map} \ g \ y
\]

\textit{t.map_id0}:
\[
\text{map} \ id = id
\]

\textit{t.map_id}:
\[
\text{map} \ id \ t = t
\]

\textit{t.map_ident}:
\[
\text{map} \ (\lambda x. \ x) \ t = t
\]
2. Defining Datatypes

\textbf{2.4.3 Inductive Theorems}

The inductive theorems are as follows:

\textbf{t.induct} [\textit{case_names} \(C_1 \ldots C_n\), \textit{induct} \(t\)]:
\[ [P :] \quad \forall x_1 x_2. \; P \; x_2 \implies P \; (x_1 \# x_2) \implies P \; \text{list} \]

\textbf{t.rel_induct} [\textit{case_names} \(C_1 \ldots C_n\), \textit{induct} \(\text{pred}\)]:
\[
\begin{align*}
\text{list}_\text{all}2 \; R \; x \; y & \implies Q \; [] \; \quad \forall a_{21} a_{22} b_{21} b_{22}. \; [R \; a_{21} \; b_{21}; \; Q \; a_{22} \; b_{22}] \\
& \implies Q \; (a_{21} \# a_{22}) \; (b_{21} \# b_{22}) \implies Q \; x \; y
\end{align*}
\]
2 Defining Datatypes

\[ t_1 \ldots \_t_m \text{\_induct} \ [\text{case\_names } C_1 \ldots C_n] : \]
\[ t_1 \ldots \_t_m \text{\_rel\_induct} \ [\text{case\_names } C_1 \ldots C_n] : \]
Given \( m > 1 \) mutually recursive datatypes, this induction rule can be used to prove \( m \) properties simultaneously.

\[ t\text{\_rec} \ [\text{simp, code}] : \]
\[ \text{rec\_list } f_1 f_2 [] = f_1 \]
\[ \text{rec\_list } f_1 f_2 (x_{21} \# x_{22}) = f_2 x_{21} x_{22} (\text{rec\_list } f_1 f_2 x_{22}) \]
The [code] attribute is set by the code plugin (Section 8.1).

\[ t\text{\_rec\_o\_map} : \]
\[ \text{rec\_list } g ga \circ \text{map } f = \text{rec\_list } g (\lambda x xa. ga (f x) (\text{map } f xa)) \]

\[ t\text{\_rec\_transfer} \ [\text{transfer\_rule}] : \]
\[ \text{rel\_fun } S (\text{rel\_fun } (\text{rel\_fun } R (\text{rel\_fun } (\text{list\_all2 } R) (\text{rel\_fun } S S)))) (\text{rel\_fun } (\text{list\_all2 } R) S)) \text{\_rec\_list \text{\_rec\_list} \]
The [transfer\_rule] attribute is set by the transfer plugin (Section 8.3) for type constructors with no dead type arguments.

For convenience, datatype also provides the following collection:

\[ t\text{\_simps} = t\text{\_inject} t\text{\_distinct} t\text{\_case} t\text{\_rec} t\text{\_map} t\text{\_rel\_inject} t\text{\_rel\_distinct} t\text{\_set} \]

2.5 Compatibility Issues
The command datatype has been designed to be highly compatible with the old command (which is now called old\_datatype), to ease migration. There are nonetheless a few incompatibilities that may arise when porting:

- **The Standard ML interfaces are different.** Tools and extensions written to call the old ML interfaces will need to be adapted to the new interfaces. The BNF\_LFP\_Compat structure provides convenience functions that simulate the old interfaces in terms of the new ones.

- **The recursor rec\_t has a different signature for nested recursive datatypes.** In the old package, nested recursion through non-functions was internally reduced to mutual recursion. This reduction was visible in the type of the recursor, used by primrec. Recursion through functions was handled specially. In the new package, nested recursion (for functions and non-functions) is handled in a more modular fashion. The old-style recursor can be generated on demand using primrec if the recursion is via new-style datatypes, as explained in Section 3.1.5.
Accordingly, the induction rule is different for nested recursive data-typess. Again, the old-style induction rule can be generated on demand using `primrec` if the recursion is via new-style datatypes, as explained in Section 3.1.5. For recursion through functions, the old-style induction rule can be obtained by applying the `[unfolded all_mem_range]` attribute on `t.induct`.

The size function has a slightly different definition. The new function returns 1 instead of 0 for some nonrecursive constructors. This departure from the old behavior made it possible to implement size in terms of the generic function `t.size_t`. Moreover, the new function considers nested occurrences of a value, in the nested recursive case. The old behavior can be obtained by disabling the size plugin (Section 8) and instantiating the size type class manually.

The internal constructions are completely different. Proof texts that unfold the definition of constants introduced by `old_datatype` will be difficult to port.

Some constants and theorems have different names. For non-mutually recursive datatypes, the alias `t.inducts` for `t.induct` is no longer generated. For `m > 1` mutually recursive datatypes, `rec_t1_..._tm_i` has been renamed `rec_t_i` for each `i ∈ {1, ... , t}`, `t1_..._tm.inducts(i)` has been renamed `t_i.induct` for each `i ∈ {1, ... , t}`, and the collection `t1_..._tm.size` (generated by the size plugin, Section 8.2) has been divided into `t1.size`, ..., `tm.size`.

The `t.simps` collection has been extended. Previously available theorems are available at the same index as before.

Variables in generated properties have different names. This is rarely an issue, except in proof texts that refer to variable names in the `[where ...]` attribute. The solution is to use the more robust `[of ...]` syntax.

In the other direction, there is currently no way to register old-style datatypes as new-style datatypes. If the goal is to define new-style datatypes with nested recursion through old-style datatypes, the old-style datatypes can be registered as a BNF (Section 6). If the goal is to derive discriminators and selectors, this can be achieved using `free_constructors` (Section 7).

### 3 Defining Primitively Recursive Functions

Recursive functions over datatypes can be specified using the `primrec` command, which supports primitive recursion, or using the more general `fun`,
3 Defining Primitively Recursive Functions

function, and partial_function commands. In this tutorial, the focus is
on primrec; fun and function are described in a separate tutorial [5].

3.1 Introductory Examples

Primitive recursion is illustrated through concrete examples based on the
datatypes defined in Section 2.1. More examples can be found in the directory
~/src/HOL/Datatype_Examples.

3.1.1 Nonrecursive Types

Primitive recursion removes one layer of constructors on the left-hand side
in each equation. For example:

```plaintext
primrec (nonexhaustive) bool_of_trool :: "trool ⇒ bool" where
  "bool_of_trool False" = "False"
  "bool_of_trool True" = "True"

primrec the_list :: "'a option ⇒ 'a list" where
  "the_list None" = "[]"
  "the_list (Some a)" = "[a]"

primrec the_default :: "'a ⇒ 'a option ⇒ 'a" where
  "the_default d None" = "d"
  "the_default _ (Some a)" = "a"

primrec mirror :: "('a, 'b, 'c) triple ⇒ ('c, 'b, 'a) triple" where
  "mirror (Triple a b c)" = "Triple c b a"
```

The equations can be specified in any order, and it is acceptable to leave out
some cases, which are then unspecified. Pattern matching on the left-hand
side is restricted to a single datatype, which must correspond to the same
argument in all equations.

3.1.2 Simple Recursion

For simple recursive types, recursive calls on a constructor argument are
allowed on the right-hand side:

```plaintext
primrec replicate :: "nat ⇒ 'a ⇒ 'a list" where
  "replicate Zero _" = "[]"
  "replicate (Succ n) x" = "x # replicate n x"

primrec (nonexhaustive) at :: "'a list ⇒ nat ⇒ 'a" where
  "at (x # xs) j" =
```
Pattern matching is only available for the argument on which the recursion takes place. Fortunately, it is easy to generate pattern-matching equations using the `simps_of_case` command provided by the theory `~/src/HOL/Library/Simps_Case_Conv`.

This generates the lemma collection `at_simps`:

\[
\begin{align*}
\text{at} (x \# xs) \text{ Zero} &= x \\
\text{at} (xa \# xs) (\text{ Succ} x) &= at \text{ xs} x
\end{align*}
\]

The next example is defined using `fun` to escape the syntactic restrictions imposed on primitively recursive functions:

\[
\begin{align*}
\text{fun} \text{ at_least_two} :: \text{ "nat } \Rightarrow \text{ bool" where} \\
\text{at_least_two} (\text{ Succ} (\text{ Succ } _)) \longleftrightarrow \text{ True} \\
\text{at_least_two} _ \longleftrightarrow \text{ False}
\end{align*}
\]

### 3.1.3 Mutual Recursion

The syntax for mutually recursive functions over mutually recursive datatypes is straightforward:

\[
\begin{align*}
\text{primrec} & \\
\text{nat_of_even_nat} :: \text{ "even_nat } \Rightarrow \text{ nat" and} \\
\text{nat_of_odd_nat} :: \text{ "odd_nat } \Rightarrow \text{ nat"}
\end{align*}
\]

\[
\begin{align*}
\text{where} & \\
\text{nat_of_even_nat } \text{ Even_Zero} &= \text{ Zero} \\
\text{nat_of_even_nat } (\text{ Even_Succ } n) &= \text{ Succ } (\text{ nat_of_odd_nat } n) \\
\text{nat_of_odd_nat } (\text{ Odd_Succ } n) &= \text{ Succ } (\text{ nat_of_even_nat } n)
\end{align*}
\]

\[
\begin{align*}
\text{primrec} & \\
\text{eval}_e :: \text{ "(a } \Rightarrow \text{ int} ) \Rightarrow (b \Rightarrow \text{ int}) \Rightarrow (a, b) \text{ exp } \Rightarrow \text{ int" and} \\
\text{eval}_t :: \text{ "(a } \Rightarrow \text{ int} ) \Rightarrow (b \Rightarrow \text{ int}) \Rightarrow (a, b) \text{ trm } \Rightarrow \text{ int" and} \\
\text{eval}_f :: \text{ "(a } \Rightarrow \text{ int} ) \Rightarrow (b \Rightarrow \text{ int}) \Rightarrow (a, b) \text{ fct } \Rightarrow \text{ int"}
\end{align*}
\]

\[
\begin{align*}
\text{where} & \\
\text{eval}_e \gamma \xi (\text{ Term } t) &= \text{ eval}_t \gamma \xi t \\
\text{eval}_e \gamma \xi (\text{ Sum } t e) &= \text{ eval}_t \gamma \xi t + \text{ eval}_e \gamma \xi e \\
\text{eval}_t \gamma \xi (\text{ Factor } f) &= \text{ eval}_f \gamma \xi f
\end{align*}
\]
"evalₜ γ ξ (Prod f t) = evalₜ γ ξ f + evalₜ γ ξ t" |
"eval₂ γ _ (Const a) = γ a" |
"evalₙ ξ (Var b) = ξ b" |
"eval₂ γ ξ (Expr e) = evalₑ γ ξ e”

Mutual recursion is possible within a single type, using `fun`:

```plaintext
fun
  even :: “nat ⇒ bool” and
  odd :: “nat ⇒ bool”
where
  “even Zero = True” |
  “even (Succ n) = odd n” |
  “odd Zero = False” |
  “odd (Succ n) = even n”
```

### 3.1.4 Nested Recursion

In a departure from the old datatype package, nested recursion is normally handled via the map functions of the nesting type constructors. For example, recursive calls are lifted to lists using `map`:

```plaintext
primrec atff :: "'a treeff ⇒ nat list ⇒ 'a" where
  "atff (Nodeff a ts) js =
  (case js of
   [] ⇒ a
   | j # js' ⇒ atff (map_option atff js) ts) j"
```

The next example features recursion through the `option` type. Although `option` is not a new-style datatype, it is registered as a BNF with the map function `map_option`:

```plaintext
primrec sum_btree :: "'(a::{zero,plus}) btree ⇒ 'a" where
  "sum_btree (BNode a lt rt) =
   a + the_default 0 (map_option sum_btree lt) +
   the_default 0 (map_option sum_btree rt)"
```

The same principle applies for arbitrary type constructors through which recursion is possible. Notably, the map function for the function type (`⇒`) is simply composition (`op o`):

```plaintext
primrec relabel_ft :: "('a ⇒ 'a) ⇒ 'a ftree ⇒ 'a ftree" where
  "relabel_ft f (FTLeaf x) = FTLeaf (f x)" |
  "relabel_ft f (FTNode g) = FTNode (relabel_ft f o g)"
```

For convenience, recursion through functions can also be expressed using `λ`-abstractions and function application rather than through composition. For example:
3 Defining Primitively Recursive Functions

\textbf{primrec} \texttt{relabel\_ft} :: "'(a ⇒ 'a) ⇒ 'a ftree ⇒ 'a ftree" where
\hspace*{1em}"relabel\_ft f (FTLeaf x) = FTLeaf (f x)" |
\hspace*{1em}"relabel\_ft f (FTNode g) = FTNode (λx. relabel\_ft f (g x))"

\textbf{primrec} (nonexhaustive) \texttt{subtree\_ft} :: "'(a ⇒ 'a) ⇒ 'a ftree ⇒ 'a ftree" where
\hspace*{1em}"subtree\_ft x (FTNode g) = g x"

For recursion through curried \(n\)-ary functions, \(n\) applications of \(op \circ\) are necessary. The examples below illustrate the case where \(n = 2\):

\texttt{datatype} 'a ftree2 = FTLeaf2 'a | FTNode2 "'(a ⇒ 'a) ⇒ 'a⇒ 'a ftree2"'

\textbf{primrec} \texttt{relabel\_ft2} :: "'(a ⇒ 'a) ⇒ 'a ftree2 ⇒ 'a ftree2" where
\hspace*{1em}"relabel\_ft2 f (FTLeaf2 x) = FTLeaf2 (f x)" |
\hspace*{1em}"relabel\_ft2 f (FTNode2 g) = FTNode2 (op \circ (op \circ (relabel\_ft2 f)) g)"

\textbf{primrec} \texttt{relabel\_ft2} :: "'(a ⇒ 'a) ⇒ 'a ftree2 ⇒ 'a ftree2" where
\hspace*{1em}"relabel\_ft2 f (FTLeaf2 x) = FTLeaf2 (f x)" |
\hspace*{1em}"relabel\_ft2 f (FTNode2 g) = FTNode2 (λx y. relabel\_ft2 f (g x y))"

\textbf{primrec} (nonexhaustive) \texttt{subtree\_ft2} :: "'(a ⇒ 'a) ⇒ 'a ftree2 ⇒ 'a ftree2" where
\hspace*{1em}"subtree\_ft2 x y (FTNode2 g) = g x y"

3.1.5 Nested-as-Mutual Recursion

For compatibility with the old package, but also because it is sometimes convenient in its own right, it is possible to treat nested recursive datatypes as mutually recursive ones if the recursion takes place through new-style datatypes. For example:

\textbf{primrec} (nonexhaustive)
\hspace*{1em} \texttt{at\_ff} :: "'a tree\_ff ⇒ nat list ⇒ 'a" and
\hspace*{1em} \texttt{ats\_ff} :: "'a tree\_ff list ⇒ nat ⇒ nat list ⇒ 'a"

where
\hspace*{1em}"at\_ff (Node\_ff a ts) js =
\hspace*{2em}(case js of
\hspace*{3em}[] ⇒ a
\hspace*{3em}| j # js' ⇒ ats\_ff ts j js')" |
\hspace*{1em}"ats\_ff (t # ts) j =
\hspace*{2em}(case j of
\hspace*{3em}Zero ⇒ at\_ff t
\hspace*{3em}| Succ' ⇒ ats\_ff ts j')"

Appropriate induction rules are generated as \texttt{at\_ff.induct}, \texttt{ats\_ff.induct}, and \texttt{at\_ff_ats\_ff.induct}. The induction rules and the underlying recursors are generated on a per-need basis and are kept in a cache to speed up subsequent definitions.
Here is a second example:

```ml
primrec
sum_btree :: "('a::{zero,plus}) btree ⇒ 'a" and
sum_btree_option :: "'a btree option ⇒ 'a"
where
"sum_btree (BNode a lt rt) =
a + sum_btree_option lt + sum_btree_option rt" |
"sum_btree_option None = 0" |
"sum_btree_option (Some t) = sum_btree t"
```

3.2 Command Syntax

3.2.1 primrec

```
primrec : local_theory → local_theory
```

```
primrec
    target
    pr-options

where
    pr-equation

pr-options

( ( plugins
  nonexhaustive
  transfer ) )
```
The `primrec` command introduces a set of mutually recursive functions over datatypes.

The syntactic entity `target` can be used to specify a local context, `fixes` denotes a list of names with optional type signatures, `thmdecl` denotes an optional name for the formula that follows, and `prop` denotes a HOL proposition [9].

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 `noneexhaustive` option indicates that the functions are not necessarily specified for all constructors. It can be used to suppress the warning that is normally emitted when some constructors are missing.
- 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 Generated Theorems

The `primrec` command generates the following properties (listed for `tfold`):

#### `f.simps [simp, code]`:
- `tfold uu (TNil y) = y`
- `tfold f (TCons x xs) = f x (tfold f xs)`
  The `[code]` attribute is set by the `code` plugin (Section 8.1).

#### `f.transfer [transfer_rule]`:
- `rel_fun (rel_fun R2 (rel_fun R1 R1)) (rel_fun (rel_tlist R2 R1) R1) tfold tfold`
  This theorem is generated by the `transfer` plugin (Section 8.3) for functions declared with the `transfer` option enabled.
\( f \text{\_induct} [\text{case\_names } C_1 \ldots C_n] \):

This induction rule is generated for nested-as-mutual recursive functions (Section 3.1.5).

\( f_1 \ldots f_m \text{\_induct} [\text{case\_names } C_1 \ldots C_n] \):

This induction rule is generated for nested-as-mutual recursive functions (Section 3.1.5). Given \( m > 1 \) mutually recursive functions, this rule can be used to prove \( m \) properties simultaneously.

### 3.4 Recursive Default Values for Selectors

A datatype selector \( un\_D \) can have a default value for each constructor on which it is not otherwise specified. Occasionally, it is useful to have the default value be defined recursively. This leads to a chicken-and-egg situation, because the datatype is not introduced yet at the moment when the selectors are introduced. Of course, we can always define the selectors manually afterward, but we then have to state and prove all the characteristic theorems ourselves instead of letting the package do it.

Fortunately, there is a workaround that relies on overloading to relieve us from the tedium of manual derivations:

1. Introduce a fully unspecified constant \( un\_D_0 :: 'a \) using \texttt{consts}.
2. Define the datatype, specifying \( un\_D_0 \) as the selector’s default value.
3. Define the behavior of \( un\_D_0 \) on values of the newly introduced datatype using the \texttt{overloading} command.
4. Derive the desired equation on \( un\_D \) from the characteristic equations for \( un\_D_0 \).

The following example illustrates this procedure:

\texttt{consts termi0 :: 'a}

\texttt{datatype ('a, 'b) tlist =}
\texttt{TNil (termi: 'b)}
\texttt{| TCons (thd: 'a) (ttl: "('a, 'b) tlist")}
\texttt{where}
\texttt{"ttl (TNil y) = TNil y"}
\texttt{| "termi (TCons _ xs) = termi0 xs"}

\texttt{overloading}
\texttt{termi0 \equiv "termi0 :: ('a, 'b) tlist \Rightarrow 'b"}

\texttt{begin}
4 Defining Codatatypes

primrec termi₀ :: "('a, 'b) tlist ⇒ 'b" where
  "termi₀ (TNil y) = y" |
  "termi₀ (TCons x xs) = termi₀ xs"
end

lemma termi_TCons[simp]: "termi (TCons x xs) = termi xs"
  by (cases xs) auto

3.5 Compatibility Issues

The command primrec’s behavior on new-style datatypes has been designed
to be highly compatible with that for old-style datatypes, to ease migration.
There is nonetheless at least one incompatibility that may arise when porting
to the new package:

- Some theorems have different names. For \( m > 1 \) mutually recursive
  functions, \( f_1 \ldots f_m \).simp has been broken down into separate sub-
collections \( f_i .simp \).

4 Defining Codatatypes

Codatatypes can be specified using the codatatype command. The command
is first illustrated through concrete examples featuring different flavors
of corecursion. More examples can be found in the directory ~~/src/HOL/
Datatype_Examples. The Archive of Formal Proofs also includes some useful
codatatypes, notably for lazy lists [6].

4.1 Introductory Examples

4.1.1 Simple Corecursion

Non-corecursive codatatypes coincide with the corresponding datatypes, so
they are rarely used in practice. Corecursive codatatypes have the same
syntax as recursive datatypes, except for the command name. For example,
here is the definition of lazy lists:

codatatype (lset: 'a) llist =
  lnull: LNil
| LCons (lhd: 'a) (ltl: "'a llist")
for
  map: lmap
  rel: llist_all2
Lazy lists can be infinite, such as \( \text{LCons} \, 0 \, (\text{LCons} \, 0 \, (\ldots)) \) and \( \text{LCons} \, 0 \, (\text{LCons} \, 1 \, (\text{LCons} \, 2 \, (\ldots))) \). Here is a related type, that of infinite streams:

\[
\text{codatatype} \ (\text{sset} : \prime a) \ \text{stream} = \ S\text{Cons} \ (\text{shd} : \prime a) \ (\text{stl} : "\prime a \ stream")
\]

for

\[
\begin{align*}
\text{map} : & \ \text{smap} \\
\text{rel} : & \ \text{stream\_all2}
\end{align*}
\]

Another interesting type that can be defined as a codatatype is that of the extended natural numbers:

\[
\text{codatatype} \ \text{enat} = \text{EZero} \mid \text{ESucc}\ \text{enat}
\]

This type has exactly one infinite element, \( \text{ESucc} \ (\text{ESucc} \ (\text{ESucc} \ (\ldots))) \), that represents \( \infty \). In addition, it has finite values of the form \( \text{ESucc} \ (\ldots \ (\text{ESucc} \ \text{EZero}) \ldots) \).

Here is an example with many constructors:

\[
\text{codatatype} \ \prime a \ \text{process} = \ Fail \\
| \ \text{Skip} \ (\text{cont} : "\prime a \ \text{process}") \\
| \ \text{Action} \ (\text{prefix} : \prime a) \ (\text{cont} : "\prime a \ \text{process}") \\
| \ \text{Choice} \ (\text{left} : "\prime a \ \text{process}") \ (\text{right} : "\prime a \ \text{process}")
\]

Notice that the \text{cont} selector is associated with both \text{Skip} and \text{Action}.

### 4.1.2 Mutual Corecursion

The example below introduces a pair of \textit{mutually corecursive} types:

\[
\text{codatatype} \ \text{even\_enat} = \text{Even\_EZero} \mid \text{Even\_ESucc}\ \text{odd\_enat} \\
\text{and} \ \text{odd\_enat} = \text{Odd\_ESucc}\ \text{even\_enat}
\]

### 4.1.3 Nested Corecursion

The next examples feature \textit{nested corecursion}:

\[
\text{codatatype} \ \prime a \ \text{tree}_{1i} = \text{Node}_{1i} \ (\text{lbl}_{1i} : \prime a) \ (\text{sub}_{1i} : "\prime a \ \text{tree}_{1i}\ \text{llist}")
\]

\[
\text{codatatype} \ \prime a \ \text{tree}_{1s} = \text{Node}_{1s} \ (\text{lbl}_{1s} : \prime a) \ (\text{sub}_{1s} : "\prime a \ \text{tree}_{1s}\ \text{fset}")
\]

\[
\text{codatatype} \ \prime a \ \text{sm} = \text{SM} \ (\text{accept} : \text{bool}) \ (\text{trans} : "\prime a \ \Rightarrow \prime a \ \text{sm}")
\]
4.2 Command Syntax

4.2.1 codatatype

codatatype : local_theory → local_theory

Definitions of codatatypes have almost exactly the same syntax as for datatypes (Section 2.2). The discs_sels option is superfluous because discriminators and selectors are always generated for codatatypes.

4.3 Generated Constants

Given a codatatype \((a_1, \ldots, a_m)\) with \(m > 0\) live type variables and \(n\) constructors \(t.C_1, \ldots, t.C_n\), the same auxiliary constants are generated as for datatypes (Section 2.3), except that the recursor is replaced by a dual concept:

Corecursion: \(t.corec_t\)

4.4 Generated Theorems

The characteristic theorems generated by codatatype are grouped in three broad categories:

- The free constructor theorems (Section 2.4.1) are properties of the constructors and destructors that can be derived for any freely generated type.
- The functorial theorems (Section 2.4.2) are properties of datatypes related to their BNF nature.
- The coinductive theorems (Section 4.4.1) are properties of datatypes related to their coinductive nature.

The first two categories are exactly as for datatypes.
4.4.1 Coinductive Theorems

The coinductive theorems are listed below for 'a llist:

\[\text{t.coinduct} \left[\text{cons} m, \text{case} \_na m t_1 \ldots t_m, \text{case} \_con D_1 \ldots D_n; \text{coind} t\right]:\]

\[R \text{ llist}'; \bigwedge \text{llist'}. R \text{ llist llist'} \implies \text{lnull llist} = \text{lnull llist'} \land (\neg lnull llist \rightarrow \neg lnull llist') \implies \text{lhd llist} = \text{lhd llist'} \land (R \text{ lllist}) (\text{lll llist'})) \implies \text{llist} = \text{llist'}\]

\[\text{t.coinduct} \_\text{strong} \left[\text{cons} m, \text{case} \_na m t_1 \ldots t_m, \text{case} \_con D_1 \ldots D_n; \text{coind} t\right]:\]

\[R \text{ llist}'; \bigwedge \text{llist'}. R \text{ llist llist'} \implies \text{lnull llist} = \text{lnull llist'} \land (\neg lnull llist \rightarrow \neg lnull llist') \implies \text{lhd llist} = \text{lhd llist'} \land (R \text{ lllist}) (\text{lll llist'})) \implies \text{llist} = \text{llist'}\]

\[\text{t.rel} \_\text{coinduct} \left[\text{cons} m, \text{case} \_na m t_1 \ldots t_m, \text{case} \_con D_1 \ldots D_n; \text{coind} t\right]:\]

\[P x y; \bigwedge \text{llist'}. P \text{l list llist'} \implies \text{lnull list} = \text{lnull list'} \land (\neg lnull llist \rightarrow \neg lnull llist') \implies R (\text{lhd list}) (\text{lhd list'}) \land P (R \text{ lllist}) (R \text{ lllist'}) \implies \text{llist} = \text{llist'}\]

\[t_1 \ldots t_m.\text{coinduct} \left[\text{case} \_na m t_1 \ldots t_m, \text{case} \_con D_1 \ldots D_n\right]\]

\[t_1 \ldots t_m.\text{coinduct} \_\text{strong} \left[\text{case} \_na m t_1 \ldots t_m, \text{case} \_con D_1 \ldots D_n\right]:\]

\[t_1 \ldots t_m.\text{rel} \_\text{coinduct} \left[\text{case} \_na m t_1 \ldots t_m, \text{case} \_con D_1 \ldots D_n\right]:\]

Given \( m > 1 \) mutually corecursive codatatypes, these coinduction rules can be used to prove \( m \) properties simultaneously.

\[t_1 \ldots t_m.\text{set} \_\text{induct} \left[\text{case} \_na C_1 \ldots C_n, \text{ind} \_\text{set}: \text{setj} \_\text{t} _1, \ldots, \text{ind} \_\text{set}: \text{setj} \_\text{t} _m\right]:\]

\[x \in \text{lset a}; \bigwedge z1 z2. P z1 (\text{LCons} z1 z2); \bigwedge z1 z2 xa. [xa \in \text{lset z2}; P xa \text{LCons} z1 z2]] \implies P x a\]

If \( m = 1 \), the attribute [consumes 1] is generated as well.

\[t.\text{corec}:\]

\[p a \implies \text{corec llist p g21 g22 g222 a} = \text{LNil}\]

\[- p a \implies \text{corec llist p g21 g22 g222 a} = \text{LCons (g21 a)} (\text{if g22 a then g221 a else corec llist p g21 g22 g222 (g222 a)})\]

\[t.\text{corec} \_\text{code} \left[\text{code}\right]:\]

\[\text{corec llist p g21 g22 g222 a} = (\text{if p a then LNil else LCons (g21 a)} (\text{if g22 a then g221 a else corec llist p g21 g22 g222 g222 a}))\]

The [code] attribute is set by the code plugin (Section 8.1).
5 Defining Primitively Corecursive Functions

Corecursive functions can be specified using the `primcorec` and `primcorecursive` commands, which support primitive corecursion, or using the more general `partial_function` command. In this tutorial, the focus is on the first two. More examples can be found in the directory `~/src/HOL/Datatype_Examples`.

Whereas recursive functions consume datatypes one constructor at a time, corecursive functions construct codatatypes one constructor at a time. Partly reflecting a lack of agreement among proponents of coalgebraic methods, Isabelle supports three competing syntaxes for specifying a function $f$:

- The *destructor view* specifies $f$ by implications of the form
  \[
  \ldots 
  \Rightarrow \text{is}_C j \ (f \ x_1 \ldots \ x_n)
  \]

For convenience, `codatatype` also provides the following collection:

\[
\begin{align*}
  \text{t.simps} &= \text{t.inject t.distinct t.case t.corec_disc_iff t.corec_sel} \\
  \text{t.map t.rel_inject t.rel_distinct t.set}
\end{align*}
\]
and equations of the form
\[ \text{un}_C \text{C}_j i \ (f \ x_1 \ldots \ x_n) = \ldots \]

This style is popular in the coalgebraic literature.

- The constructor view specifies \( f \) by equations of the form
\[ \ldots \implies f \ x_1 \ldots \ x_n = C \]

This style is often more concise than the previous one.

- The code view specifies \( f \) by a single equation of the form
\[ f \ x_1 \ldots \ x_n = \ldots \]

with restrictions on the format of the right-hand side. Lazy functional programming languages such as Haskell support a generalized version of this style.

All three styles are available as input syntax. Whichever syntax is chosen, characteristic theorems for all three styles are generated.

### 5.1 Introductory Examples

Primitive corecursion is illustrated through concrete examples based on the codatatypes defined in Section 4.1. More examples can be found in the directory `~/src/HOL/Datatype_Examples`. The code view is favored in the examples below. Sections 5.1.5 and 5.1.6 present the same examples expressed using the constructor and destructor views.

#### 5.1.1 Simple Corecursion

Following the code view, corecursive calls are allowed on the right-hand side as long as they occur under a constructor, which itself appears either directly to the right of the equal sign or in a conditional expression:

\[
\text{primcorec literate} :: \"(a \Rightarrow \text{C}_a) \Rightarrow \text{C}_a \Rightarrow \text{C}_a \text{ list}\" \text{ where }
\"\text{literate} \ g \ x = \text{LCons} \ x \ (\text{literate} \ g \ (g \ x))\"
\]

\[
\text{primcorec siterate} :: \"(a \Rightarrow \text{C}_a) \Rightarrow \text{C}_a \Rightarrow \text{C}_a \text{ stream}\" \text{ where }
\"\text{siterate} \ g \ x = \text{SCons} \ x \ (\text{siterate} \ g \ (g \ x))\"
\]

The constructor ensures that progress is made—i.e., the function is productive. The above functions compute the infinite lazy list or stream \([x, g \ x, g \ (g \ x), \ldots]\). Productivity guarantees that prefixes \([x, g \ x, g \ (g \ x), \ldots, (g \ ^\ldots \ k) \ x]\) of arbitrary finite length \(k\) can be computed by unfolding the code equation a finite number of times.
Corecursive functions construct codatatype values, but nothing prevents them from also consuming such values. The following function drops every second element in a stream:

```ml
primcorec every_snd :: "'a stream ⇒ 'a stream" where
  "every_snd s = SCons (shd s) (stl (stl s))"
```

Constructs such as `let-in`, `if-then-else`, and `case-of` may appear around constructors that guard corecursive calls:

```ml
primcorec lappend :: "'a llist ⇒ 'a llist ⇒ 'a llist" where
  "lappend xs ys = (case xs of
    LNil ⇒ ys
    | LCons x xs' ⇒ LCons x (lappend xs' ys))"
```

Pattern matching is not supported by `primcorec`. Fortunately, it is easy to generate pattern-matching equations using the `simps_of_case` command provided by the theory `~/src/HOL/Library/Simps_Case_Conv`.

```ml
simps_of_case lappend_simps: lappend.code
```

This generates the lemma collection `lappend_simps`:

```
lappend LNl y = y
lappend (LCons xa x) ys = LCons xa (lappend x y)
```

Corecursion is useful to specify not only functions but also infinite objects:

```ml
primcorec infty :: enat where
  "infty = ESucc infty"
```

The example below constructs a pseudorandom process value. It takes a stream of actions (`s`), a pseudorandom function generator (`f`), and a pseudorandom seed (`n`):

```ml
primcorec
random_process :: "'a stream ⇒ (int ⇒ int) ⇒ int ⇒ 'a process" where
  "random_process s f n = (if n mod 4 = 0 then
    Fail
  else if n mod 4 = 1 then
    Skip (random_process s f (f n))
  else if n mod 4 = 2 then
    Action (shd s) (random_process (stl s) f (f n))
  else
    Choice (random_process (every_snd s) (f ◦ f) (f n)))
```
The main disadvantage of the code view is that the conditions are tested sequentially. This is visible in the generated theorems. The constructor and destructor views offer nonsequential alternatives.

5.1.2 Mutual Corecursion

The syntax for mutually corecursive functions over mutually corecursive datatypes is unsurprising:

```plaintext
primcorec
  even_infty :: even_enat
  odd_infty :: odd_enat
where
  "even_infty = Even_ESucc odd_infty" |
  "odd_infty = Odd_ESucc even_infty"
```

5.1.3 Nested Corecursion

The next pair of examples generalize the `iterate` and `siterate` functions (Section 5.1.3) to possibly infinite trees in which subnodes are organized either as a lazy list (`tree_i i`) or as a finite set (`tree_i s`). They rely on the map functions of the nesting type constructors to lift the corecursive calls:

```plaintext
primcorec iterate_i i :: "'(a ⇒ 'a llist) ⇒ 'a ⇒ 'a tree_i i" where
  "iterate_i i g x = Node_i i x (lmap (iterate_i i g) (g x))"
primcorec iterate_i s :: "'(a ⇒ 'a fset) ⇒ 'a ⇒ 'a tree_i s" where
  "iterate_i s g x = Node_i s x (fimage (iterate_i s g) (g x))"
```

Both examples follow the usual format for constructor arguments associated with nested recursive occurrences of the datatype. Consider `iterate_i i`. The term `g x` constructs an `'a llist` value, which is turned into an `'a tree_i i llist` value using `lmap`.

This format may sometimes feel artificial. The following function constructs a tree with a single, infinite branch from a stream:

```plaintext
primcorec tree_i i_of_stream :: "'a stream ⇒ 'a tree_i i" where
  "tree_i i_of_stream s = Node_i i (shd s) (lmap tree_i i_of_stream (LCons (stl s) LNil))"
```

A more natural syntax, also supported by Isabelle, is to move corecursive calls under constructors:

```plaintext
primcorec tree_i i_of_stream :: "'a stream ⇒ 'a tree_i i" where
  "tree_i i_of_stream s =
```
The next example illustrates corecursion through functions, which is a bit special. Deterministic finite automata (DFAs) are traditionally defined as 5-tuples \((Q, \Sigma, \delta, q_0, F)\), where \(Q\) is a finite set of states, \(\Sigma\) is a finite alphabet, \(\delta\) is a transition function, \(q_0\) is an initial state, and \(F\) is a set of final states. The following function translates a DFA into a state machine:

\[
\text{primcorec } \text{sm_of_dfa} :: (\lambda q \Rightarrow \lambda a \Rightarrow \lambda q') \Rightarrow \lambda q \Rightarrow \lambda a \Rightarrow \lambda (q, a) \text{sm_2}
\]

\[
\text{sm_of_dfa } \delta F q = \text{SM} (q \in F) (\lambda a . \text{sm_of_dfa } \delta F (\delta a q))
\]

For recursion through curried \(n\)-ary functions, \(n\) applications of \(\circ\) are necessary. The examples below illustrate the case where \(n = 2\):

\[
\text{codatatype } (\lambda a \Rightarrow \lambda b \Rightarrow \lambda (a, b)) \text{sm_2} = \\
\text{SM}_2 (\text{accept}_2 : \text{bool}) (\text{trans}_2 : (\lambda a \Rightarrow \lambda b : (a, b)) \text{sm_2})
\]

\[
\text{primcorec } \text{sm_2_of_dfa} :: (\lambda q \Rightarrow \lambda a \Rightarrow \lambda b \Rightarrow (a, b)) \Rightarrow \lambda q \Rightarrow \lambda q \Rightarrow (a, b) \text{sm_2}
\]

\[
\text{sm_2_of_dfa } \delta F q = \text{SM}_2 (q \in F) (\lambda a b . \text{sm_2_of_dfa } \delta F (\delta q a b))
\]

5.1.4 Nested-as-Mutual Corecursion

Just as it is possible to recurse over nested recursive datatypes as if they were mutually recursive (Section 3.1.5), it is possible to pretend that nested codatatypes are mutually corecursive. For example:
5 Defining Primitively Corecursive Functions

\textbf{primcorec}
\begin{align*}
\text{iterate}_{ii} &:: \"(a \Rightarrow 'a \text{ llist}) \Rightarrow 'a \Rightarrow 'a \text{ tree}_{ii} \" \text{ and } \\
\text{iterates}_{ii} &:: \"(a \Rightarrow 'a \text{ llist}) \Rightarrow 'a \text{ llist} \Rightarrow 'a \text{ tree}_{ii} \text{ llist} \"
\end{align*}
\textbf{where}
\begin{align*}
\text{iterate}_{ii} \ g \ x & = \text{Node}_{ii} \ x \ (\text{iterates}_{ii} \ g \ (g \ x)) \ |
\text{iterates}_{ii} \ g \ xs & = \\
& \quad \text{(case } xs \text{ of} \\
& \quad \quad \text{LNil } \Rightarrow \text{ LNil} \\
& \quad \quad \text{LCons } x \ ys' \Rightarrow \text{ LCons } (\text{iterate}_{ii} \ g \ x) \ (\text{iterates}_{ii} \ g \ xs')"
\end{align*}

Coinduction rules are generated as \text{iterate}_{ii}.\text{coinduct}, \text{iterates}_{ii}.\text{coinduct}, and \text{iterate}_{ii}.\_\text{iterates}_{ii}.\text{coinduct} and analogously for \text{coinduct}_{\text{strong}}. These rules and the underlying corecurors are generated on a per-need basis and are kept in a cache to speed up subsequent definitions.

5.1.5 Constructor View
The constructor view is similar to the code view, but there is one separate conditional equation per constructor rather than a single unconditional equation. Examples that rely on a single constructor, such as \text{iterate} and \text{siterate}, are identical in both styles.

Here is an example where there is a difference:

\textbf{primcorec} \ lappend :: \"'a \text{ llist} \Rightarrow 'a \text{ llist} \Rightarrow 'a \text{ llist} \"
\textbf{where}
\begin{align*}
\text{lnull } xs & \Rightarrow \text{lnull } ys \Rightarrow \text{lappend } xs \ ys = \text{LNil} \ |
\_ & \Rightarrow \text{lappend } xs \ ys = \text{LCons } (\text{ltl } xs \text{ then } ys \text{ else } xs)) \\
& \quad (\text{if } xs = \text{LNil } \text{then } \text{ltl } ys \text{ else } \text{lappend } (\text{ltl } xs) \ ys")
\end{align*}

With the constructor view, we must distinguish between the \text{LNil} and the \text{LCons} case. The condition for \text{LCons} is left implicit, as the negation of that for \text{LNil}.

For this example, the constructor view is slightly more involved than the code equation. Recall the code view version presented in Section 5.1.1. The constructor view requires us to analyze the second argument \((ys)\). The code equation generated from the constructor view also suffers from this.

In contrast, the next example is arguably more naturally expressed in the constructor view:

\textbf{primcorec}
\begin{align*}
\text{random\_process} &:: \"'a \text{ stream} \Rightarrow (\text{int} \Rightarrow \text{int}) \Rightarrow \text{int} \Rightarrow 'a \text{ process} \"
\end{align*}
\textbf{where}
\begin{align*}
\text{n mod } 4 = 0 & \Rightarrow \text{random\_process } s \ f \ n = \text{Fail} \ |
\text{n mod } 4 = 1 & \Rightarrow \\
& \quad \text{random\_process } s \ f \ n = \text{Skip } (\text{random\_process } s \ f \ (f \ n))" \\
\end{align*}
5 Defining Primitively Corecursive Functions

“\( n \mod 4 = 2 \implies \) 
\begin{equation}
\text{random\_process } s f n = \text{Action} (\text{shd } s) (\text{random\_process } (\text{stl } s) f (f n))
\end{equation}”

“\( n \mod 4 = 3 \implies \) 
\begin{equation}
\text{random\_process } s f n = \text{Choice} (\text{random\_process } (\text{every\_snd } s) f (f n)) \\
(\text{random\_process } (\text{every\_snd } (\text{stl } s)) f (f n))
\end{equation}”

Since there is no sequentiality, we can apply the equation for Choice without having first to discharge \( n \mod 4 \neq 0 \), \( n \mod 4 \neq 1 \), and \( n \mod 4 \neq 2 \). The price to pay for this elegance is that we must discharge exclusiveness proof obligations, one for each pair of conditions \( (n \mod 4 = i, n \mod 4 = j) \) with \( i < j \). If we prefer not to discharge any obligations, we can enable the sequential option. This pushes the problem to the users of the generated properties.

5.1.6 Destructor View

The destructor view is in many respects dual to the constructor view. Conditions determine which constructor to choose, and these conditions are interpreted sequentially or not depending on the sequential option. Consider the following examples:

\begin{verbatim}
primcorec literate :: "'(a ⇒ 'a) ⇒ 'a ⇒ 'a llist" where
  "¬ lnull (literate _ x)" |
  "lhd (literate _ x) = x" |
  "ltl (literate g x) = literate g (g x)"

primcorec siterate :: "'(a ⇒ 'a) ⇒ 'a ⇒ 'a stream" where
  "shd (siterate _ x) = x" |
  "stl (siterate g x) = siterate g (g x)"

primcorec every snd :: "'a stream ⇒ 'a stream" where
  "shd (every snd s) = shd s" |
  "stl (every snd s) = stl (stl s)"
\end{verbatim}

The first formula in the local.literate specification indicates which constructor to choose. For local.siterate and local.every snd, no such formula is necessary, since the type has only one constructor. The last two formulas are equations specifying the value of the result for the relevant selectors. Corecursive calls appear directly to the right of the equal sign. Their arguments are unrestricted.

The next example shows how to specify functions that rely on more than one constructor:

\begin{verbatim}
primcorec lappend :: "'(a llist ⇒ 'a llist ⇒ 'a llist" where
  "lnull xs ⇒ lnull ys ⇒ lnull (lappend xs ys)" |
\end{verbatim}
Defining Primitively Corecursive Functions

```
"lhd (lappend xs ys) = lhd (if lnull xs then ys else xs)" |
"ltl (lappend xs ys) = (if xs = LNil then ltl ys else lappend (ltl xs) ys)"
```

For a codatatype with \( n \) constructors, it is sufficient to specify \( n - 1 \) discriminator formulas. The command will then assume that the remaining constructor should be taken otherwise. This can be made explicit by adding

```
_ \implies \neg lnull (lappend xs ys)
```
to the specification. The generated selector theorems are conditional.

The next example illustrates how to cope with selectors defined for several constructors:

```
primcorec
random_process :: "'a stream \Rightarrow (int \Rightarrow int) \Rightarrow int \Rightarrow 'a process"
where
"n mod 4 = 0 \implies random_process s f n = Fail" |
"n mod 4 = 1 \implies is_Skip (random_process s f n)" |
"n mod 4 = 2 \implies is_Action (random_process s f n)" |
"n mod 4 = 3 \implies is_Choice (random_process s f n)" |
"cont (random_process s f n) = random_process s f (f n)" of Skip |
"prefix (random_process s f n) = shd s" |
"cont (random_process s f n) = random_process (stl s) f (f n)" of Action |
"left (random_process s f n) = random_process (every_snd s) f (f n)" |
"right (random_process s f n) = random_process (every_snd (stl s)) f (f n)"
```

Using the \( of \) keyword, different equations are specified for \( cont \) depending on which constructor is selected.

Here are more examples to conclude:

```
primcorec
even_infty :: even_enat and
odd_infty :: odd_enat
where
"even_infty \neq Even_EZero" |
"un_Even_ESucc even_infty = odd_infty" |
"un_Odd_ESucc odd_infty = even_infty"
```

```
primcorec iterate :: "('a \Rightarrow 'a llist) \Rightarrow 'a \Rightarrow 'a tree" where
"lbl (iterate g x) = x" |
"sub (iterate g x) = lmap (iterate g) (g x)"
```

5.2 Command Syntax

5.2.1 primcorec and primcorecursive

```
primcorec : local_theory \rightarrow local_theory
primcorecursive : local_theory \rightarrow proof(prove)
```
The `primcorec` and `primcorecursive` commands introduce a set of mutually corecursive functions over codatatypes.

The syntactic entity `target` can be used to specify a local context, `fixes` denotes a list of names with optional type signatures, `thmdecl` denotes an optional name for the formula that follows, and `prop` denotes a HOL proposition [9].

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 sequential option indicates that the conditions in specifications expressed using the constructor or destructor view are to be interpreted sequentially.

• The exhaustive option indicates that the conditions in specifications expressed using the constructor or destructor view cover all possible cases. This generally gives rise to an additional proof obligation.

• 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 primcorec command is an abbreviation for primcorecursive with by auto? to discharge any emerging proof obligations.

5.3 Generated Theorems

The primcorec and primcorecursive commands generate the following properties (listed for literate):

\[ \text{\textit{f.code}} \text{ [code]}:\]
\[ \text{literate } g \ x = \text{LCons } x \ (\text{literate } g \ (g \ x)) \]
The [code] attribute is set by the code plugin (Section 8.1).

\[ \text{\textit{f.ctr}}:\]
\[ \text{literate } g \ x = \text{LCons } x \ (\text{literate } g \ (g \ x)) \]

\[ \text{\textit{f.disc}} \text{ [simp, code]}:\]
\[ \neg \text{lnull } (\text{literate } g \ x) \]
The [code] attribute is set by the code plugin (Section 8.1). The [simp] attribute is set only for functions for which f.disc_iff is not available.

\[ \text{\textit{f.disc_iff}} \text{ [simp]}:\]
\[ \neg \text{lnull } (\text{literate } g \ x) \]
This property is generated only for functions declared with the exhaustive option or whose conditions are trivially exhaustive.

\[ \text{\textit{f.sel}} \text{ [simp, code]}:\]
\[ \neg \text{lnull } (\text{literate } g \ x) \]
The [code] attribute is set by the code plugin (Section 8.1).
6 Registering Bounded Natural Functors

\( f.\text{exclude} \):
These properties are missing for \textit{literate} because no exclusiveness proof obligations arose. In general, the properties correspond to the discharged proof obligations.

\( f.\text{exhaust} \):
This property is missing for \textit{literate} because no exhaustiveness proof obligation arose. In general, the property correspond to the discharged proof obligation.

\( f.\text{coinduct} \ [\text{consumes } m, \text{case_names } t_1 \ldots t_m, \newline \quad \text{case_conclusion } D_1 \ldots D_n] \):
This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4).

\( f.\text{coinduct\_strong} \ [\text{consumes } m, \text{case_names } t_1 \ldots t_m, \newline \quad \text{case_conclusion } D_1 \ldots D_n] \):
This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4).

\( f_1 \ldots f_m.\text{coinduct} \ [\text{case_names } t_1 \ldots t_m, \newline \quad \text{case_conclusion } D_1 \ldots D_n] \):
This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4). Given \( m > 1 \) mutually corecursive functions, this rule can be used to prove \( m \) properties simultaneously.

\( f_1 \ldots f_m.\text{coinduct\_strong} \ [\text{case_names } t_1 \ldots t_m, \newline \quad \text{case_conclusion } D_1 \ldots D_n] \):
This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4). Given \( m > 1 \) mutually corecursive functions, this rule can be used to prove \( m \) properties simultaneously.

For convenience, \texttt{primcorec} and \texttt{primcorecursive} also provide the following collection:

\( f.\text{simps} = f.\text{disc\_iff} \) (or \( f.\text{disc} \)) \( t.\text{sel} \)

6 Registering Bounded Natural Functors

The (co)datatype package can be set up to allow nested recursion through arbitrary type constructors, as long as they adhere to the BNF requirements and are registered as BNFs. It is also possible to declare a BNF abstractly without specifying its internal structure.
6.1 Bounded Natural Functors

Bounded natural functors (BNFs) are a semantic criterion for where (co)recursion may appear on the right-hand side of an equation [3,8].

An \( n \)-ary BNF is a type constructor equipped with a map function (functorial action), \( n \) set functions (natural transformations), and an infinite cardinal bound that satisfy certain properties. For example, \( 'a \ llist \) is a unary BNF. Its relator \( llist\_all2 :: (\mathcal{A} \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \ llist \Rightarrow 'b \ llist \Rightarrow \text{bool} \) extends binary predicates over elements to binary predicates over parallel lazy lists. The cardinal bound limits the number of elements returned by the set function; it may not depend on the cardinality of \( 'a \).

The type constructors introduced by \texttt{datatype} and \texttt{codatatype} are automatically registered as BNFs. In addition, a number of old-style datatypes and non-free types are preregistered.

Given an \( n \)-ary BNF, the \( n \) type variables associated with set functions, and on which the map function acts, are live; any other variables are dead. Nested (co)recursion can only take place through live variables.

6.2 Introductory Examples

The example below shows how to register a type as a BNF using the \texttt{bnf} command. Some of the proof obligations are best viewed with the theory \texttt{Cardinal_Notations}, located in \texttt{~/src/HOL/Library}, imported.

The type is simply a copy of the function space \( 'd \Rightarrow 'a \), where \( 'a \) is live and \( 'd \) is dead. We introduce it together with its map function, set function, and relator.

\begin{verbatim}
typedef ('d, 'a) fn = "UNIV :: ('d => 'a) set" by simp
setup_lifting type_definition_fn

lift_definition map_fn :: "('a => 'b) => ('d, 'a) fn => ('d, 'b) fn" is "op o" .
lift_definition set_fn :: "('d, 'a) fn => 'a set" is range .
lift_definition rel_fn :: "('a => 'b => bool) => ('d, 'a) fn => ('d, 'b) fn => bool" is "rel_fun (op =)" .
bnf "('d, 'a) fn"
  map: map_fn
  sets: set_fn
  bd: "natLeq +c \ UNIV :: 'd set"
\end{verbatim}
rel: rel_fn

proof -
  show "map_fn id = id"
    by transfer auto
next
fix f :: "'a ⇒ 'b" and g :: "'b ⇒ 'c"
show "map_fn (g ∘ f) = map_fn g ∘ map_fn f"
  by transfer (auto simp add: comp_def)
next
fix F :: "('d, 'a) fn" and f g :: "'a ⇒ 'b"
assume "∀x. x ∈ set_fn F ⟹ f x = g x"
thus "map_fn f F = map_fn g F"
  by transfer auto
next
fix f :: "'a ⇒ 'b"
show "set_fn ∘ map_fn f = op 'f ∘ set_fn"
  by transfer (auto simp add: comp_def)
next
show "card_order (natLeq +c |UNIV :: 'd set| )"
  apply (rule card_order_csum)
  apply (rule natLeq_card_order)
  by (rule card_of_card_order_on)
next
show "cinfinite (natLeq +c |UNIV :: 'd set| )"
  apply (rule cinfinite_csum)
  apply (rule disjI1)
  by (rule natLeq_cinfinite)
next
fix F :: "('d, 'a) fn"
have "|set_fn F| ≤o |UNIV :: 'd set|" (is "_ ≤o ?U")
  by transfer (rule card_of_image)
also have "?U ≤o natLeq +c ?U"
  by (rule ordLeq_csum2) (rule card_of_Card_order)
finally show "|set_fn F| ≤o natLeq +c |UNIV :: 'd set|".
next
fix R :: "'a ⇒ 'b ⇒ bool" and S :: "'b ⇒ 'c ⇒ bool"
show "rel_fn R OO rel_fn S ≤ rel_fn (R OO S)"
  by (rule, transfer) (auto simp add: rel_fun_def)
next
fix R :: "'a ⇒ 'b ⇒ bool"
show "rel_fn R =
  (BNF_Def.Grp {x. set_fn x ⊆ Collect (split R)} (map_fn fst))−− OO
  BNF_Def.Grp {x. set_fn x ⊆ Collect (split R)} (map_fn snd)"
unfolding $\text{Grp}_\text{def} \text{ fun_eq_iff relcompp.simps conversep.simps}$
apply transfer
unfolding $\text{rel_fun_def subset_iff image_iff}$
by auto (force, metis pair_collapse)
qed

print_theorems
print_bnfs

Using **print_theorems** and **print_bnfs**, we can contemplate and show the world what we have achieved.

This particular example does not need any nonemptiness witness, because the one generated by default is good enough, but in general this would be necessary. See `~~/src/HOL/Basic_BNFs.thy`, `~~/src/HOL/Library/FSet.thy`, and `~~/src/HOL/Library/Multiset.thy` for further examples of BNF registration, some of which feature custom witnesses.

The next example declares a BNF axiomatically. This can be convenient for reasoning abstractly about an arbitrary BNF. The **bnf_axiomatization** command below introduces a type $(\alpha, \beta, \gamma)$ $F$, three set constants, a map function, a relator, and a nonemptiness witness that depends only on $\alpha$. The type $\alpha \Rightarrow (\alpha, \beta, \gamma) F$ of the witness can be read as an implication: Given a witness for $\alpha$, we can construct a witness for $(\alpha, \beta, \gamma) F$. The BNF properties are postulated as axioms.

```
bnf_axiomatization (setA: $\alpha$, setB: $\beta$, setC: $\gamma$) $F$
[wits: "$\alpha \Rightarrow (\alpha, \beta, \gamma) F"]
```

print_theorems
print_bnfs

### 6.3 Command Syntax

#### 6.3.1 bnf

```
bnf : local_theory $\rightarrow$ proof(prove)
```
The `bnf` command registers an existing type as a bounded natural functor (BNF). The type must be equipped with an appropriate map function (functorial action). In addition, custom set functions, relators, and nonemptiness witnesses can be specified; otherwise, default versions are used.

The syntactic entity `target` can be used to specify a local context, `type` denotes a HOL type, and `term` denotes a HOL term [9].

The `plugins` option indicates which plugins should be enabled (`only`) or disabled (`del`). By default, all plugins are enabled.

### 6.3.2 `bnf_axiomatization`

\[
\text{bnf\_axiomatization} \quad : \quad \text{local\_theory} \rightarrow \text{local\_theory}
\]
The `bnf_axiomatization` command declares a new type and associated constants (map, set, relator, and cardinal bound) and asserts the BNF properties for these constants as axioms.

The syntactic entity `target` can be used to specify a local context, `name` denotes an identifier, `typefree` denotes fixed type variable (‘a, ‘b, . . .), and `mixfix` denotes the usual parenthesized mixfix notation [9].

The `plugins` option indicates which plugins should be enabled (only) or disabled (del). By default, all plugins are enabled.

Type arguments are live by default; they can be marked as dead by entering `dead` in front of the type variable (e.g., `(dead ’a)`) instead of an identifier for the corresponding set function. Witnesses can be specified by their types. Otherwise, the syntax of `bnf_axiomatization` is identical to the left-hand side of a `datatype` or `codatatype` definition.

The command is useful to reason abstractly about BNFs. The axioms are safe because there exist BNFs of arbitrary large arities. Applications must import the theory `BNF_Axiomatization`, located in the directory `~/src/HOL/Library`, to use this functionality.
6.3.3 print_bnfs

\texttt{print_bnfs : local\_theory → }

7 Deriving Destructors and Theorems for Free Constructors

The derivation of convenience theorems for types equipped with free constructors, as performed internally by \texttt{datatype} and \texttt{codatatype}, is available as a stand-alone command called \texttt{free_constructors}.

7.1 Command Syntax

7.1.1 free_constructors

\texttt{free_constructors : local\_theory → proof(prove)}
The `free_constructors` command generates destructor constants for freely constructed types as well as properties about constructors and destructors. It also registers the constants and theorems in a data structure that is queried by various tools (e.g., `function`).

The syntactic entity `target` can be used to specify a local context, `name` denotes an identifier, `prop` denotes a HOL proposition, and `term` denotes a HOL term [9].

The syntax resembles that of `datatype` and `codatatype` definitions (Sections 2.2 and 4.2). A constructor is specified by an optional name for the discriminator, the constructor itself (as a term), and a list of optional names for the selectors.

Section 2.4 lists the generated theorems. For bootstrapping reasons, the generally useful `[fundef_cong]` attribute is not set on the generated `case_cong` theorem. It can be added manually using `declare`.

8 Selecting Plugins

Plugins extend the (co)datatype package to interoperate with other Isabelle packages and tools, such as the code generator, Transfer, Lifting, and Quickcheck. They can be enabled or disabled individually using the `plugins` option to the commands `datatype`, `primrec`, `codatatype`, `primcorec`, `primcorecursive`, `bnf`, `bnf_axiomatization`, and `free_constructors`. For example:

```
datatype (plugins del: code “quickcheck”) color = Red | Black
```

8.1 Code Generator

The `code` plugin registers freely generated types, including (co)datatypes, and (co)recursive functions for code generation. No distinction is made between datatypes and codatatypes. This means that for target languages with a strict evaluation strategy (e.g., Standard ML), programs that attempt to produce infinite codatatype values will not terminate.

For types, the plugin derives the following properties:
t.eq.refl [code nbe]:
  equal_class.equal x x ≡ True

t.eq.simps [code]:
  equal_class.equal [] (x21 # x22) ≡ False
  equal_class.equal (x21 # x22) [] ≡ False
  equal_class.equal (x21 # x22) [] ≡ False
  equal_class.equal (x21 # x22) (y21 # y22) ≡ x21 = y21 ∧ x22 = y22
  equal_class.equal [] [] ≡ True

In addition, the plugin sets the [code] attribute on a number of properties of freely generated types and of (co)recursive functions, as documented in Sections 2.4, 3.3, 4.4, and 5.3.

8.2 Size

For each datatype, the size plugin generates a generic size function \( t.size_t \) as well as a specific instance \( size :: t \Rightarrow \mathbb{nat} \) belonging to the size type class. The \texttt{fun} command relies on size to prove termination of recursive functions on datatypes.

The plugin derives the following properties:

t.size [simp, code]:
  size_list x [] = 0
  size_list x (x21 # x22) = x x21 + size_list x x22 + Suc 0
  size [] = 0
  size (x21 # x22) = size x22 + Suc 0

t.size_gen:
  size_list x [] = 0
  size_list x (x21 # x22) = x x21 + size_list x x22 + Suc 0

t.size_gen_o_map:
  size_list f \circ map g = size_list (f \circ g)

t.size_neq:
  This property is missing for 'a list. If the size function always evaluates to a non-zero value, this theorem has the form size x \neq 0.
8.3 Transfer

For each (co)datatype with live type arguments and each manually registered BNF, the `transfer` plugin generates a predicator `t.pred_t` and properties that guide the Transfer tool.

For types with no dead type arguments (and at least one live type argument), the plugin derives the following properties:

\[ t.\text{Domainp} \_ \text{rel} [\text{relator\_domain}] : \]
\[ \text{Domainp} \ (\text{list\_all2} \ R) = \text{pred\_list} \ (\text{Domainp} \ R) \]

\[ t.\text{pred\_inject} [\text{simp}] : \]
\[ \text{pred\_list} \ P \ [] \]
\[ \text{pred\_list} \ P \ (a \ # \ aa) = (P \ a \land \text{pred\_list} \ P \ aa) \]
This property is generated only for (co)datatypes.

\[ t.\text{rel\_eq\_onp} : \]
\[ \text{list\_all2} \ (\text{eq\_onp} \ P) = \text{eq\_onp} \ (\text{pred\_list} \ P) \]

\[ t.\text{left\_total\_rel} [\text{transfer\_rule}] : \]
\[ \text{left\_total} \ R \implies \text{left\_total} \ (\text{list\_all2} \ R) \]

\[ t.\text{left\_unique\_rel} [\text{transfer\_rule}] : \]
\[ \text{left\_unique} \ R \implies \text{left\_unique} \ (\text{list\_all2} \ R) \]

\[ t.\text{right\_total\_rel} [\text{transfer\_rule}] : \]
\[ \text{right\_total} \ R \implies \text{right\_total} \ (\text{list\_all2} \ R) \]

\[ t.\text{right\_unique\_rel} [\text{transfer\_rule}] : \]
\[ \text{right\_unique} \ R \implies \text{right\_unique} \ (\text{list\_all2} \ R) \]

\[ t.\text{bi\_total\_rel} [\text{transfer\_rule}] : \]
\[ \text{bi\_total} \ R \implies \text{bi\_total} \ (\text{list\_all2} \ R) \]

\[ t.\text{bi\_unique\_rel} [\text{transfer\_rule}] : \]
\[ \text{bi\_unique} \ R \implies \text{bi\_unique} \ (\text{list\_all2} \ R) \]

In addition, the plugin sets the `[transfer_rule]` attribute on the following (co)datatypes properties: `t.case_transfer`, `t.sel_transfer`, `t.ctr_transfer`, `t.disc_transfer`, `t.set_transfer`, `t.map_transfer`, `t.rel_transfer`, `t.rec_transfer`, and `t.corec_transfer`.

For `primrec`, `primcorec`, and `primcorecursive`, the plugin implements the generation of the `f.transfer` property, conditioned by the `transfer` option, and sets the `[transfer_rule]` attribute on these.
8.4 Lifting

For each (co)datatype and each manually registered BNF with at least one live type argument and no dead type arguments, the lifting plugin generates properties and attributes that guide the Lifting tool.

The plugin derives the following property:

\[
\text{Quotient}\left[quot\_map\right]: \quad \text{Quotient}\left(R\ Abs\ Rep\ T\right) \implies \text{Quotient}\left(\text{list\_all2}\ R\right) (\text{map}\ Abs) (\text{map}\ Rep) (\text{list\_all2}\ T)
\]

In addition, the plugin sets the \[relator\_eq\_onp\] attribute on a variant of the \(t.rel\_eq\_onp\) property generated by the lifting plugin, the \[relator\_mono\] attribute on \(t.rel\_mono\), and the \[relator\_distr\] attribute on \(t.rel\_compp\).

8.5 Quickcheck

The integration of datatypes with Quickcheck is accomplished by the quickcheck plugin. It combines a number of subplugins that instantiate specific type classes. The subplugins can be enabled or disabled individually. They are listed below:

\[
\text{quickcheck\_random}
\quad \text{quickcheck\_exhaustive}
\quad \text{quickcheck\_bounded\_forall}
\quad \text{quickcheck\_full\_exhaustive}
\quad \text{quickcheck\_narrowing}
\]

8.6 Program Extraction

The extraction plugin provides realizers for induction and case analysis, to enable program extraction from proofs involving datatypes. This functionality is only available with full proof objects, i.e., with the HOL-Proofs session.

9 Known Bugs and Limitations

This section lists the known bugs and limitations in the (co)datatype package at the time of this writing. Many of them are expected to be addressed in future releases.
1. Defining mutually (co)recursive (co)datatypes is slow. Fortunately, it is always possible to recast mutual specifications to nested ones, which are processed more efficiently.

2. Locally fixed types cannot be used in (co)datatype specifications. This limitation can be circumvented by adding type arguments to the local (co)datatypes to abstract over the locally fixed types.

3. The primcorec command does not allow user-specified names and attributes next to the entered formulas. The less convenient syntax, using the lemmas command, is available as an alternative.

4. There is no way to use an overloaded constant from a syntactic type class, such as 0, as a constructor.

5. There is no way to register the same type as both a datatype and a codatatype. This affects types such as the extended natural numbers, for which both views would make sense (for a different set of constructors).

6. The names of variables are often suboptimal in the properties generated by the package.

Acknowledgment

Tobias Nipkow and Makarius Wenzel encouraged us to implement the new (co)datatype package. Andreas Lochbihler provided lots of comments on earlier versions of the package, especially on the coinductive part. Brian Huffman suggested major simplifications to the internal constructions. Ondřej Kuncar implemented the transfer and lifting plugins. Christian Sternagel and René Thiemann ported the derive command from the Archive of Formal Proofs to the new datatypes. Florian Haftmann and Christian Urban provided general advice on Isabelle and package writing. Stefan Milius and Lutz Schröder found an elegant proof that eliminated one of the BNF proof obligations. Gerwin Klein, Andreas Lochbihler, Tobias Nipkow, and Christian Sternagel suggested many textual improvements to this tutorial.

References

REFERENCES


