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

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

12 December 2021

Abstract
This tutorial describes the definitional package for datatypes and co-
datatypes, and for primitively recursive and corecursive functions,
in Isabelle/HOL. The following commands are provided: datatype,
datatype_compat, primrec, codatatype, primcorec, primcorecursive,
bnf, lift_bnf, copy_bnf, bnf_axiomatization,
print_bnsfs, and 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 .......................... 7
    2.1.5 Auxiliary Constants ....................... 7
  2.2 Command Syntax ............................... 9
    2.2.1 datatype ................................... 9
    2.2.2 datatype_compat ......................... 12
  2.3 Generated Constants ........................... 13
  2.4 Generated Theorems ............................ 13
    2.4.1 Free Constructor Theorems ............... 14
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:

\[
\text{datatype } 'a \text{ tree } = \text{Node}_f (\text{lbl}_s : 'a) (\text{sub}_s : "'a \text{ tree } fset")
\]

Another strong point is the support for local definitions:
1 INTRODUCTION

context linorder
begin
datatype flag = Less | Eq | Greater
end

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:

codatatype 'a llist = LNil | LCons 'a "'a llist"

Mixed inductive–coinductive recursion is possible via nesting. Compare the following four Rose tree examples:

datatype 'a treeff = Nodeff 'a "'a treeff list"
datatype 'a treefi = Nodefi 'a "'a treefi list"
codatatype 'a treeif = Nodeif 'a "'a treeif list"
codatatype 'a treeii = Nodeii 'a "'a treeii llist"

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 ~/src/HOL/Library/BNF_Axiomatization.thy.

The package, like its predecessor, fully adheres to the LCF philosophy [5]: The characteristic theorems associated with the specified (co)datatypes are derived rather than introduced axiomatically. The package is described in a number of scientific papers [2, 4, 9, 11]. The central notion is that of a 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 datatype command.
- Section 3, “Defining Primitively Recursive Functions,” describes how to specify functions using primrec. (A separate tutorial [6] describes the more powerful fun and function commands.)

\footnote{However, some of the internal constructions and most of the internal proof obligations are omitted if the quick_and_dirty option is enabled.}
• Section 4, “Defining Codatatypes,” describes how to specify codatatypes using the codatatype command.

• Section 5, “Defining Primitively Corecursive Functions,” describes how to specify functions using the primcorec and primcorecursive commands. (A separate tutorial [3] describes the more powerful corec and corecursive commands.)

• Section 6, “Registering Bounded Natural Functors,” explains how to use the bnf command to register arbitrary type constructors as BNFs.

• Section 7, “Deriving Destructors and Constructor Theorems,” explains how to use the command free_constructors to derive destructor constants and theorems for freely generated types, as performed internally by datatype and 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.

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

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 = Truue | Faalse | Perhaaps
True, False, and Possibly have the type 'bool.

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

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

The constructors are None :: 'a option and Some :: 'a => 'a option.

The next example has three type parameters:

```plaintext
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:

```plaintext
datatype ('a, 'b, 'c) triple = Triple u "'a * 'b * '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:

```plaintext
datatype nat = Zero | Succ nat
```

Lists were shown in the introduction. Terminated lists are a variant that stores a value of type 'b at the very end:

```plaintext
datatype ('a, 'b) tlist = TNil 'b | TCons ('a, 'b) 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:

```plaintext
datatype even_nat = Even_Zero | Even_Succ odd_nat
    and odd_nat = Odd_Succ even_nat
```

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

```plaintext
datatype ('a, 'b) exp =
    Term "('a, 'b) trm" | Sum "('a, 'b) trm" "('a, 'b) exp"
    and ('a, 'b) trm =
    Factor "('a, 'b) fct" | Prod "('a, 'b) fct" "('a, 'b) trm"
    and ('a, 'b) fct =
    Const 'a | Var 'b | Expr "('a, 'b) 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 \texttt{option} type, follows:

\begin{verbatim}
datatype 'a btree =
  BNode 'a "a btree option" "a btree option"
\end{verbatim}

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

\begin{verbatim}
datatype 'a wrong = W1 | W2 "a wrong ⇒ 'a"
\end{verbatim}

The issue is that the function arrow \(⇒\) allows recursion only through its right-hand side. This issue is inherited by polymorphic datatypes defined in terms of \(⇒\):

\begin{verbatim}
datatype ('a, 'b) fun_copy = Fun "a ⇒ 'b"
datatype 'a also_wrong = W1 | W2 "(a also_wrong, 'a) fun_copy"
\end{verbatim}

The following definition of \(\mathit{a}\)-branching trees is legal:

\begin{verbatim}
datatype 'a ftree =
  FTLeaf 'a |
  FTreeNode "a ⇒ 'a ftree"
\end{verbatim}

And so is the definition of hereditarily finite sets:

\begin{verbatim}
datatype hfset = HFSet "hfset fset"
\end{verbatim}

In general, type constructors \((\mathit{a}_1, \ldots, \mathit{a}_m)\) allow recursion on a subset of their type arguments \(\mathit{a}_1, \ldots, \mathit{a}_m\). These type arguments are called \textit{live}; the remaining type arguments are called \textit{dead}. In \(\mathit{a} ⇒ \mathit{b}\) and \((\mathit{a}, \mathit{b}) \mathit{fun_copy}\), the type variable \(\mathit{a}\) is dead and \(\mathit{b}\) is live.

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

Here is another example that fails:

\begin{verbatim}
datatype 'a pow_list = PNil 'a | PCons "(a * a) pow_list"
\end{verbatim}

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 \texttt{datatype} command introduces various constants in addition to the constructors. With each datatype are associated set functions, a map function, a
predicator, 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:

\begin{verbatim}
datatype (set: 'a) list =
  null: Nil
| Cons (hd: 'a) (tl: "a list")
for
  map: map
rel: list_all2
pred: list_all
where
  "tl Nil = Nil"
\end{verbatim}

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

Constructors:  \hspace{1cm} \begin{align*}
  Nil :: 'a list \\
  Cons :: 'a \Rightarrow 'a list \Rightarrow 'a list
\end{align*}

Discriminator: \hspace{1cm} \begin{align*}
  null :: 'a list \Rightarrow bool
\end{align*}

Selectors: \hspace{1cm} \begin{align*}
  hd :: 'a list \Rightarrow 'a \\
  tl :: 'a list \Rightarrow 'a list
\end{align*}

Set function: \hspace{1cm} \begin{align*}
  set :: 'a list \Rightarrow 'a set
\end{align*}

Map function: \hspace{1cm} \begin{align*}
  map :: ('a \Rightarrow 'b) \Rightarrow 'a list \Rightarrow 'b list
\end{align*}

Relator: \hspace{1cm} \begin{align*}
  list_all2 :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'b list \Rightarrow bool
\end{align*}

The discriminator null and the selectors hd and tl are characterized by
the following conditional equations:

$$
null \; xs \implies xs = Nil \quad \neg null \; xs \implies Cons \; (hd \; xs) \; (tl \; xs) = xs
$$

For two-constructor datatypes, a single discriminator constant is sufficient.
The discriminator associated with Cons is simply \( \lambda xs. \neg null \; xs \).

The \texttt{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 \( \lambda xs. \; xs = Nil \) as a
discriminator. 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:
### 2.2 Command Syntax

#### 2.2.1 datatype

```plaintext
datatype : local_theory → local_theory
```

**dt-options**

```plaintext
( plugins discs_sels )
```

**plugins**

```plaintext
plugins only : del name
```
The `datatype` command introduces a set of mutually recursive datatypes specified by their constructors.

The syntactic entity `target` can be used to specify a local context (e.g., `(in linorder) [12]`), and `prop` denotes a HOL proposition.

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 `discs_sels` option indicates that discriminators and selectors should be generated. The option is implicitly enabled if names are specified for discriminators or selectors.

The optional `where` clause specifies default values for selectors. Each proposition must be an equation of the form `un_D (C . . .) = . . .`, where `C` is a constructor and `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:

\[ \text{dt-name} \]

\[ \text{tyargs} \]

\[ \text{mixfix} \]

The syntactic entity \textit{name} denotes an identifier, \textit{mixfix} denotes the usual parenthesized mixfix notation, and \textit{typefree} denotes fixed type variable (\textit{′a}, \textit{′b}, \ldots) [12].

The optional names preceding the type variables allow to override the default names of the set functions (\textit{set}_1\textit{t}, \ldots, \textit{set}_m\textit{t}). Type arguments can be marked as dead by entering \textit{dead} in front of the type variable (e.g., \textit{(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.
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 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.

The syntactic entity `type` denotes a HOL type [12].

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 `un_C_{ij}`.

### 2.2.2 datatype_compat

The `datatype_compat` command registers new-style datatypes as old-style datatypes and invokes the old-style plugins. For example:

```ml
datatype_compat even_nat odd_nat
```

The syntactic entity `name` denotes an identifier [12]. 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). These theorems should be identical to the ones generated by the old datatype package, up to the order of the premises—meaning that the subgoals generated by the induct or induction method may be in a different order than before.

All types through which recursion takes place must be new-style datatypes 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 \( t.\cdot C_1, \ldots, t.\cdot C_n \), the following auxiliary constants are introduced:

- **Case combinator:** \( t.\cdot \text{case}_t \) (rendered using the familiar case–of syntax)
- **Discriminators:** \( t.\cdot \text{is}_C_1, \ldots, t.\cdot \text{is}_C_n \)
- **Selectors:** \( t.\cdot \text{un}_C_1^1, \ldots, t.\cdot \text{un}_C_1^{k_1} \)
  
  \[ \vdots \]
  
  \( t.\cdot \text{un}_C_n^1, \ldots, t.\cdot \text{un}_C_n^{k_n} \)
- **Set functions:** \( t.\cdot \text{set}_1^1_t, \ldots, t.\cdot \text{set}_m^m_t \)
- **Map function:** \( t.\cdot \text{map}_t \)
- **Relator:** \( t.\cdot \text{rel}_t \)
- **Recursor:** \( t.\cdot \text{rec}_t \)

The discriminators and selectors are generated only if the discs_sels option is enabled or if names are specified for discriminators or selectors. The set functions, map function, predicator, 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 “\( t.\cdot \)” is an optional component of the names and is normally hidden.

2.4 Generated Theorems

The characteristic theorems generated by **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 **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 by issuing the command `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

```
declare [[bnf_internals]]
```

### 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 ∧ x22 = y22)

 t.distinct  [simp, induct_simp]:
  [] ≠ x21 ≠ x22
  x21 ≠ x22 ≠ []

t.exhaust  [cases t, case_names C1 ... C_n]:
  [y = [] ⇒ P; \(\land\)x21 x22. y = x21 ≠ x22 ⇒ P] ⇒ P

t.nchotomy:
  \(\forall\) list. list = [] ∨ (∃x21 x22. list = x21 ≠ x22)
```

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

```
t.distinct  [THEN notE, elim!]:
  [] = x21 ≠ x22 ⇒ R
  x21 ≠ x22 = [] ⇒ R
```

The next subgroup is concerned with the case combinator:

```
t.case  [simp, code]:
  (case [] of [] ⇒ f1 | x ≠ xa ⇒ f2 x xa) = f1
  (case x21 ≠ x22 of [] ⇒ f1 | x ≠ xa ⇒ f2 x xa) = f2 x21 x22
```

The [code] attribute is set by the code plugin (Section 8.1).
The third subgroup revolves around discriminators and selectors:

\[ t.\text{disc} \begin{cases} \text{null} \, \emptyset \Rightarrow \text{null} \, \emptyset \\ \neg \text{null} \, (x_{21} \# x_{22}) \end{cases} \]

\[ t.\text{disc} \begin{cases} \text{list} = \emptyset \Rightarrow \text{null list} \\ \text{list} = x_{21} \# x_{22} \Rightarrow \neg \text{null list} \end{cases} \]

\[ t.\text{sel} \begin{cases} \text{hd} \, (x_{21} \# x_{22}) = x_{21} \\ \text{tl} \, (x_{21} \# x_{22}) = x_{22} \end{cases} \]

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

\[ t.\text{collapse} \begin{cases} \text{null list} \Rightarrow \text{list} = \emptyset \\ \neg \text{null list} \Rightarrow \text{hd list} \# \text{tl list} = \text{list} \end{cases} \]

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.

\[ t.\text{distinct_disc} \begin{cases} \text{These properties are missing for } 'a \text{ list} \text{ because there is only one} \end{cases} \]
proper discriminator. If the datatype had been introduced with a
second discriminator called nonnull, they would have read as follows:

\[ \text{null list} \implies \neg \text{nonnull list} \]
\[ \text{nonnull list} \implies \neg \text{null list} \]

\textit{t.exhaust\_disc} \texttt{[case\_names C}_{1}\ldots C}_{n}]:
\[ \text{null list} \implies P; \neg \text{null list} \implies P \implies P \]

\textit{t.exhaust\_sel} \texttt{[case\_names C}_{1}\ldots C}_{n}]:
\[ \text{list} = \emptyset \implies P; \text{list} = \text{hd list} \# \text{tl list} \implies P \implies P \]

\textit{t.expand}:
\[ \text{null list} = \text{null list'}; \neg \text{null list}; \neg \text{null list'} \implies \text{hd list} = \text{hd list'} \]
\[ \land \text{tl list} = \text{tl list'} \implies \text{list} = \text{list'} \]

\textit{t.split\_sel}:
\[ P \text{ (case list of } \emptyset \implies f_{1} \mid x \# xa \implies f_{2} x xa) = \left((\text{list} = \emptyset \implies P f_{1}\right) \land \left(\text{list} = \text{hd list} \# \text{tl list} \implies P f_{2} (\text{hd list} \text{ (tl list)))}\right)\]

\textit{t.split\_sel\_asm}:
\[ P \text{ (case list of } \emptyset \implies f_{1} \mid x \# xa \implies f_{2} x xa) = \left(\neg \left(\text{list} = \emptyset \land \neg P \right)
\land \left(\text{list} = \text{hd list} \# \text{tl list} \land \neg P f_{2} (\text{hd list} \text{ (tl list)))}\right)\]

\[ \textit{t.split\_sels} = \textit{split\_sel split\_sel\_asm} \]

\textit{t.case\_eq\_if}:
\[ (\text{case list of } \emptyset \implies f_{1} \mid x \# xa \implies f_{2} x xa) = \left(\text{if null list then } f_{1} \text{ else } f_{2} (\text{hd list} \text{ (tl list)))}\right)\]

\textit{t.disc\_eq\_case}:
\[ \text{null list} = (\text{case list of } \emptyset \implies \text{True} \mid wu\#wua\implies \text{False}) \]
\[ \neg \text{null list} = (\text{case list of } \emptyset \implies \text{False} \mid wu\#wua\implies \text{True}) \]

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 generated for type constructors with at least
one live type argument (e.g., '\text{'a list}'). They 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, the predicator, or
the relator:

\textit{t.case\_transfer} \texttt{[transfer\_rule]}:
\[ \text{rel\_fun } S \left(\text{rel\_fun } \left(\text{rel\_fun } R \left(\text{rel\_fun } (\text{list\_all}2 R) S\right) \text{ (rel\_fun } (\text{list\_all}2 R) S)\right) \right) \text{ case\_list case\_list} \]

This property is generated by the \texttt{transfer} plugin (Section 8.3).
2 DEFINING DATATYPES

\textit{t.sel\_transfer} [transfer\_rule]:
This property is missing for 'a list because there is no common se-
lector to all constructors.
The [transfer\_rule] attribute is set by the \textit{transfer} plugin (Section 8.3).

\textit{t.ctr\_transfer} [transfer\_rule]:
\texttt{list\_all2} R [] []
\texttt{rel\_fun} R (\texttt{rel\_fun} (\texttt{list\_all2} R) (\texttt{list\_all2} R)) (#) (#)
The [transfer\_rule] attribute is set by the \textit{transfer} plugin (Section 8.3).

\textit{t.disc\_transfer} [transfer\_rule]:
\texttt{rel\_fun} (\texttt{list\_all2} R) (=) \texttt{null} \texttt{null}
\texttt{rel\_fun} (\texttt{list\_all2} R) (=) (\texttt{\lambda} \texttt{list}. \neg \texttt{null} \texttt{list}) (\texttt{\lambda} \texttt{list}. \neg \texttt{null} \texttt{list})
The [transfer\_rule] attribute is set by the \textit{transfer} plugin (Section 8.3).

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

\textit{t.set\_cases} [consumes 1, cases set: set\_t]:
\[ e \in \text{set} a; \land z2. a = e \# z2 \implies \text{thesis}; \land z1 z2. [a = z1 \# z2; e \in \text{set} z2] \implies \text{thesis} \implies \text{thesis} \]

\textit{t.set\_intros}:
\[ x21 \in \text{set} (x21 \# x22) \]
\[ y \in \text{set} x22 \implies y \in \text{set} (x21 \# x22) \]

\textit{t.set\_sel}:
\[ \neg \text{null} a \implies \text{hd} a \in \text{set} a \]
\[ [\neg \text{null} a; x \in \text{set} (\text{tl} a)] \implies x \in \text{set} a \]

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

\textit{t.map\_disc\_iff} [simp]:
\[ \text{null} (\text{map} f a) = \text{null} a \]

\textit{t.map\_sel}:
\[ \neg \text{null} a \implies \text{hd} (\text{map} f a) = f (\text{hd} a) \]
\[ \neg \text{null} a \implies \text{tl} (\text{map} f a) = \text{map} f (\text{tl} a) \]

\textit{t.pred\_inject} [simp]:
\[ \text{list\_all} P [] \]
\[ \text{list\_all} P (a \# aa) = (P a \land \text{list\_all} P aa) \]
2  DEFINING DATATYPES

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

\[ t.\text{rel}\_distinct \ [\text{simp}]: \]
\[ \neg \text{list}\_all2 R \ [] \ (y21 \ # y22) \]
\[ \neg \text{list}\_all2 R (y21 \ # y22) \ [] \]

\[ t.\text{rel}\_intros: \]
\[ \text{list}\_all2 R \ [] \ [] \]
\[ [R x21 y21; \text{list}\_all2 R x22 y22] \implies \text{list}\_all2 R (x21 \ # x22) (y21 \ # y22) \]

\[ t.\text{rel}\_cases [\text{consumes 1, case_names t1 \ldots t_m, cases pred}]: \]
\[ [\text{list}\_all2 R a b; [a = []; b = []]] \implies \text{thesis}; \land x1 x2 y1 y2. [a = x1 \ # x2; b = y1 \ # y2; R x1 y1; \text{list}\_all2 R x2 y2] \implies \text{thesis}] \implies \text{thesis} \]

\[ t.\text{rel}\_sel: \]
\[ \text{list}\_all2 R a b = (\text{null} a = \text{null} b \land (\neg \text{null} a \implies \neg \text{null} b \implies R (\text{hd} a) (\text{hd} b) \land \text{list}\_all2 R (\text{tl} a) (\text{tl} b))) \]

In addition, equational versions of \( t.\text{rel}\_inject \) and \( \text{rel}\_distinct \) are registered with the \([\text{code}]\) attribute. The \([\text{code}]\) 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, the predicator, and the relator:

\[ t.\text{inj}\_map: \]
\[ \text{inj} f \implies \text{inj} (\text{map} f) \]

\[ t.\text{inj}\_map\_strong: \]
\[ [\land z za. [z \in \text{set} x; za \in \text{set} xa; f z = fa za] \implies z = za; \text{map} f x = \text{map} fa xa] \implies x = xa \]

\[ t.\text{map}\_comp: \]
\[ \text{map} g (\text{map} f v) = \text{map} (g \circ f) v \]

\[ t.\text{map}\_cong0: \]
\[ (\land z. z \in \text{set} x \implies f z = g z) \implies \text{map} f x = \text{map} g x \]

\[ t.\text{map}\_cong \ [\text{fundef_cong}]: \]
\[ [x = ya; \land z. z \in \text{set} ya \implies f z = g z] \implies \text{map} f x = \text{map} g ya \]

\[ t.\text{map}\_cong\_pred: \]
\[ [x = ya; \text{list}\_all (\lambda z. f z = g z) ya] \implies \text{map} f x = \text{map} g ya \]
2 DEFINING DATATYPES

\textit{t.map\_cong\_simp}:
\[
x = ya; \big\\land z. z \in \text{set} \ ya = \text{simp} \Rightarrow f z = g z \implies \text{map} f x = \text{map} g ya
\]

\textit{t.map\_id0}:
\[
\text{map id} = \text{id}
\]

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

\textit{t.map\_ident}:
\[
\text{map} (\lambda x. \ x) \ t = t
\]

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

\textit{t.pred\_cong} \ [\text{fundef\_cong}]:
\[
x = ya; \big\\land z. z \in \text{set} \ ya = \text{simp} \Rightarrow P z = Pa z \implies \text{list}\_all P x = \text{list}\_all Pa ya
\]

\textit{t.pred\_cong\_simp}:
\[
x = ya; \big\\land z. z \in \text{set} \ ya = \text{simp} \Rightarrow P z = Pa z \implies \text{list}\_all P x = \text{list}\_all Pa ya
\]

\textit{t.pred\_map}:
\[
\text{list}\_all Q (\text{map} f x) = \text{list}\_all (Q \circ f) \ x
\]

\textit{t.pred\_mono} \ [\text{mono}]:
\[
P \leq Pa \implies \text{list}\_all P \leq \text{list}\_all Pa
\]

\textit{t.pred\_mono\_strong}:
\[
\text{list}\_all P x; \big\\land z. [z \in \text{set} \ x; P z] \implies Pa z \implies \text{list}\_all Pa x
\]

\textit{t.pred\_rel}:
\[
\text{list}\_all P x = \text{list}\_all2 (eq\_onp P) \ x \ x
\]

\textit{t.pred\_set}:
\[
\text{list}\_all P = (\lambda x. \text{Ball} (\text{set} \ x) \ P)
\]

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

\textit{t.pred\_True}:
\[
\text{list}\_all (\lambda _. \text{True}) = (\lambda _. \text{True})
2 DEFINING DATATYPES

\texttt{t.set_map:}
\texttt{set (map f v) = f ↦ set v}

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

\texttt{t.rel_compp [relator_distr]:}
\texttt{list_all2 (R OO S) = list_all2 R OO list_all2 S}
The [relator_distr] attribute is set by the \texttt{lifting} plugin (Section 8.4).

\texttt{t.rel_conversep:}
\texttt{list_all2 R'' = (list_all2 R)''}

\texttt{t.rel_eq:}
\texttt{list_all2 (=) = (=)}

\texttt{t.rel_eq_onp:}
\texttt{list_all2 (eq_onp P) = eq_onp (list_all P)}

\texttt{t.rel_flip:}
\texttt{list_all2 R'' a b = list_all2 R b a}

\texttt{t.rel_map:}
\texttt{list_all2 Sb (map i x) y = list_all2 (\lambda x. Sb (i x)) x y}
\texttt{list_all2 Sa x (map g y) = list_all2 (\lambda x y. Sa x (g y)) x y}

\texttt{t.rel_mono [mono, relator_mono]:}
\texttt{R \leq Ra \implies list_all2 R \leq list_all2 Ra}
The [relator_mono] attribute is set by the \texttt{lifting} plugin (Section 8.4).

\texttt{t.rel_mono_strong:}
\texttt{[\ [ \ [ x \in set x; y \in set y; R z yb] \implies Ra z yb] \implies list_all2 R x y]}
\texttt{[\ [ \ [ z \in set x; yb \in set xa] \implies R z yb = Ra z yb] \implies list_all2 R x y = list_all2 Ra ya xa]}

\texttt{t.rel_cong [fundef_cong]:}
\texttt{[\ [ x = ya; y = xa; \ \ \ \ \ \ [ [ z \in set xa \implies R z yb = R z yb] \implies list_all2 R x y = list_all2 Ra ya xa]}
\texttt{t.rel_cong_simp:}
\texttt{[\ [ x = ya; y = xa; \ \ \ \ \ \ [ [ z \in set ya \implies R z yb = R z yb] \implies list_all2 R x y = list_all2 Ra ya xa]}

\texttt{t.rel_refl:}
\texttt{(\ \ [ x. Ra x x) \implies list_all2 Ra x x}
\texttt{t.rel_refl_strong:}
\texttt{(\ \ [ z. z \in set x \implies Ra z z) \implies list_all2 Ra x x}
2 DEFINING DATATYPES

\[ t.rel_{\text{reflp}} : \]
\[ \text{reflp } R \Rightarrow \text{reflp } (\text{list}_\text{all}^2 R) \]

\[ t.rel_{\text{symp}} : \]
\[ \text{symp } R \Rightarrow \text{symp } (\text{list}_\text{all}^2 R) \]

\[ t.rel_{\text{transp}} : \]
\[ \text{transp } R \Rightarrow \text{transp } (\text{list}_\text{all}^2 R) \]

\[ t.rel_{\text{transfer}} \text{ [transfer_rule]} : \]
\[ \text{rel}_\text{fun} (\text{rel}_\text{fun} \text{Sa} (\text{rel}_\text{fun} \text{Sc} (=))) (\text{rel}_\text{fun} \text{list}_\text{all}^2 \text{Sa}) (\text{rel}_\text{fun} \text{list}_\text{all}^2 \text{Sc} (=))) \text{ list}_\text{all}^2 \text{ list}_\text{all}^2 \]

The [transfer_rule] attribute is set by the transfer plugin (Section 8.3) for type constructors with no dead type arguments.

2.4.3 Inductive Theorems

The inductive theorems are as follows:

\[ t.induct \text{ [case_names C}_1 \ldots \text{ C}_n, \text{ induct } t] : \]
\[ [P []; \backslash x_1 x_2. P x2 \Rightarrow P (x_1 \# x_2)] \Rightarrow P \text{ list} \]

\[ t.rel\_induct \text{ [case_names C}_1 \ldots \text{ C}_n, \text{ induct } \text{pred]} : \]
\[ [\text{list}_\text{all}^2 R x y; Q [] []; \backslash a_{21} a_{22} b_{21} b_{22}. [R a_{21} b_{21}; Q a_{22} b_{22}] \Rightarrow Q (a_{21} \# a_{22}) (b_{21} \# b_{22})] \Rightarrow Q x y \]

\[ t_{1, \ldots, t_m}.\text{induct} \text{ [case_names C}_1 \ldots \text{ C}_n] : \]
\[ t_{1, \ldots, t_m}.\text{rel\_induct} \text{ [case_names C}_1 \ldots \text{ C}_n] : \]

Given \( m > 1 \) mutually recursive datatypes, this induction rule can be used to prove \( m \) properties simultaneously.

\[ t.rec \text{ [simp, code]} : \]
\[ \text{rec}_\text{list} f_1 f_2 [] = f_1 \]
\[ \text{rec}_\text{list} f_1 f_2 (x_{21} \# x_{22}) = f_2 x_{21} x_{22} \text{ (rec}_\text{list} f_1 f_2 x_{22}) \]

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

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

\[ t.rec\_\text{transfer} \text{ [transfer_rule]} : \]
\[ \text{rel}_\text{fun} S (\text{rel}_\text{fun} \text{rel}_\text{fun} R (\text{rel}_\text{fun} \text{list}_\text{all}^2 R) (\text{rel}_\text{fun} S S))) \text{ (rel}_\text{fun} \text{list}_\text{all}^2 R) S) \text{ rec}_\text{list} \text{ rec}_\text{list} \]

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:
2 DEFINING DATATYPES

\[ t.\text{simp} = t.\text{inject} \quad t.\text{distinct} \quad t.\text{case} \quad t.\text{rec} \quad t.\text{map} \quad t.\text{rel\_inject} \quad t.\text{rel\_distinct} \quad t.\text{set} \]

2.5 Proof Method

2.5.1 countable_datatype

The theory 

```plaintext
t::(countable) countable
```

by countable_datatype

2.6 Antiquotation

2.6.1 datatype

The \textit{datatype} antiquotation, written \texttt{\textbackslash<\textbackslash datatype\texttt{\{}t\texttt{\}}} or \texttt{@\{datatype \texttt{\{}t\texttt{\}}}, where \textit{t} is a type name, expands to \LaTeX{} code for the definition of the datatype, with each constructor listed with its argument types. For example, if \textit{t} is \textit{option}:

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

2.7 Compatibility Issues

The command \texttt{datatype} has been designed to be highly compatible with the old, pre-Isabelle2015 command, to ease migration. There are nonetheless a few incompatibilities that may arise when porting:

- \textit{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 \texttt{BNF_LFP_Compat} structure provides convenience functions that simulate the old interfaces in terms of the new ones.

- \textit{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 \texttt{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 \texttt{primrec} if the recursion is via new-style datatypes, as explained in Section 3.1.5, or using \texttt{datatypeCompat}.

- Accordingly, the induction rule is different for nested recursive datatypes. Again, the old-style induction rule can be generated on demand using \texttt{primrec} if the recursion is via new-style datatypes, as explained in Section 3.1.5, or using \texttt{datatypeCompat}. For recursion through functions, the old-style induction rule can be obtained by applying the \texttt{[unfolded all_mem_range]} attribute on \texttt{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 \texttt{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 the old command will be difficult to port.

- Some constants and theorems have different names. For non-mutually recursive datatypes, the alias \texttt{t.inducts} for \texttt{t.induct} is no longer generated. For \( m > 1 \) mutually recursive datatypes, \texttt{rec_t1.\ldots tm_i} has been renamed \texttt{rec_t_i} for each \( i \in \{1, \ldots, m\} \), \texttt{t1.\ldots tm.inducts(i)} has been renamed \texttt{t_i.induct} for each \( i \in \{1, \ldots, m\} \), and the collection \texttt{t1.\ldots tm.size} (generated by the size plugin, Section 8.2) has been divided into \texttt{t1.size, \ldots, tm.size}.

- The \texttt{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 \texttt{[where \ldots]} attribute. The solution is to use the more robust \texttt{[of \ldots]} syntax.

3 Defining Primitively Recursive Functions

Recursive functions over datatypes can be specified using the \texttt{primrec} command, which supports primitive recursion, or using the \texttt{fun}, \texttt{function}, and
partial_function commands. In this tutorial, the focus is on primrec; fun and function are described in a separate tutorial [6].

Because it is restricted to primitive recursion, primrec is less powerful than fun and function. However, there are primitively recursive specifications (e.g., based on infinitely branching or mutually recursive datatypes) for which fun’s termination check fails. It is also good style to use the simpler primrec mechanism when it works, both as an optimization and as documentation.

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 Faalse ←→ False"
| "bool_of_trool Truue ←→ 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:
3 DEFINING PRIMITIVELY RECURSIVE FUNCTIONS

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 = (case j of
    Zero ⇒ x
  | Succ j′ ⇒ at xs j′)"

primrec tfold :: "('a ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) tlist ⇒ 'b" where
  "tfold _ (TNil y) = y"
| "tfold f (TCons x xs) = f x (tfold f xs)"

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.thy.

simps_of_case at_simps_alt: at_simps

This generates the lemma collection at_simps_alt:

at (x ≠ xs) Zero = x   at (xa ≠ xs) (Succ x) = at xs x

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

fun at_least_two :: "nat ⇒ bool" where
  "at_least_two (Succ (Succ _)) ←→ True"
| "at_least_two _ ←→ False"

3.1.3 Mutual Recursion

The syntax for mutually recursive functions over mutually recursive data-
types is straightforward:

primrec
  nat_of_even_nat :: "even_nat ⇒ nat" and
  nat_of_odd_nat :: "odd_nat ⇒ nat"
where
  "nat_of_even_nat Even_Zero = Zero"
| "nat_of_even_nat (Even_Succ n) = Succ (nat_of_odd_nat n)"
| "nat_of_odd_nat (Odd_Succ n) = Succ (nat_of_even_nat n)"

primrec
eval_e :: "('a ⇒ int) ⇒ ('b ⇒ int) ⇒ ('a, 'b) exp ⇒ int" and
eval_t :: "('a ⇒ int) ⇒ ('b ⇒ int) ⇒ ('a, 'b) trm ⇒ int" and
eval \text{f} :: \text{"(}a \rightarrow \text{int}\text{) \Rightarrow (}b \rightarrow \text{int}\text{) \Rightarrow (}a, b\text{) fct \Rightarrow \text{int}}"

where

\text{eval}_{\gamma \xi} (\text{Term} \ t) = \text{eval}_t (\gamma \xi t)

| \text{eval}_{\gamma \xi} (\text{Sum} \ t \ e) = \text{eval}_t (\gamma \xi t + \text{eval}_e (\gamma \xi e))
| \text{eval}_{\gamma \xi} (\text{Factor} \ f) = \text{eval}_f (\gamma \xi f)
| \text{eval}_{\gamma \xi} (\text{Prod} \ f \ t) = \text{eval}_f (\gamma \xi f + \text{eval}_t (\gamma \xi t))
| \text{eval}_{\gamma \xi} (\text{Const} \ a) = \gamma \ a
| \text{eval}_{\gamma \xi} (\text{Var} \ b) = \xi \ b
| \text{eval}_{\gamma \xi} (\text{Expr} \ e) = \text{eval}_e (\gamma \xi e)

Mutual recursion is possible within a single type, using \textit{fun}:

fun
even :: “nat ⇒ bool” and
odd :: “nat ⇒ bool”

where

\text{even} \ Zero = True
| \text{even} \ (\text{Succ} \ n) = \text{odd} \ n
| \text{odd} \ Zero = False
| \text{odd} \ (\text{Succ} \ n) = \text{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 \textit{map}:

\begin{verbatim}
primrec \textit{at}\_ff :: “\textit{a} \textit{tree}\_ff \Rightarrow \textit{nat} \textit{list} \Rightarrow \textit{a}” where
\textit{at}\_ff (\textit{Node}\_ff \ a \ \textit{ts} \ js) =
\hspace{1em} (\text{case} \ js \ of
\hspace{1em} | [] ⇒ a
\hspace{1em} | j # js' ⇒ \text{at} (\text{map} \ (λ \ t. \ \textit{at}\_ff \ t \ js') \ \textit{ts}) \ j)
\end{verbatim}

The next example features recursion through the \textit{option} type. Although \textit{option} is not a new-style datatype, it is registered as a BNF with the map function \textit{map\_option}:

\begin{verbatim}
primrec \textit{sum}\_btree :: “\textit{a}::\{\text{zero},\text{plus}\} \textit{btree} \Rightarrow \textit{a}” where
\textit{sum}\_btree (\textit{BNode} \ a \ \textit{lt} \ \textit{rt}) =
\hspace{1em} a + \text{the\_default} \ 0 \ (\text{map\_option} \ \textit{sum}\_btree \ \textit{lt}) +
\hspace{1em} \text{the\_default} \ 0 \ (\text{map\_option} \ \textit{sum}\_btree \ \textit{rt})
\end{verbatim}

The same principle applies for arbitrary type constructors through which recursion is possible. Notably, the map function for the function type \textit{(⇒)} is simply composition \((\circ)):

\begin{verbatim}
primrec \textit{relabel}\_ft :: “\textit{a} ⇒ \textit{a} \textit{ftree} ⇒ \textit{a} \textit{ftree}” where
\end{verbatim}
"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:

\[
\text{primrec } \text{relabel}_\text{ft} :: \forall \alpha. (\alpha \Rightarrow \alpha \Rightarrow \alpha \text{ ftree}) \Rightarrow \alpha \text{ ftree}
\]

where

"relabel_ft f (FTLeaf x) = FTLeaf (f x)"
| "relabel_ft f (FTNode g) = FTNode (\lambda x. relabel_ft f (g x))"

\[
\text{primrec (nonexhaustive) } \text{subtree}_\text{ft} :: \forall \alpha. (\alpha \Rightarrow \alpha \Rightarrow \alpha \text{ ftree}) \Rightarrow \alpha \text{ ftree}
\]

where

"subtree_ft x (FTNode g) = g x"

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

\[
\text{datatype } \forall \alpha. \text{ ftree}_2 = \text{FTLeaf}_2 \alpha | \text{FTNode}_2 \alpha \Rightarrow \alpha \Rightarrow \alpha \text{ ftree}_2
\]

\[
\text{primrec } \text{relabel}_\text{ft}_2 :: \forall \alpha. (\alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \text{ ftree}_2) \Rightarrow \alpha \text{ ftree}_2
\]

where

"relabel_ft2 f (FTLeaf2 x) = FTLeaf2 (f x)"
| "relabel_ft2 f (FTNode2 g) = FTNode2 ((\alpha \Rightarrow (\alpha \Rightarrow (\alpha \Rightarrow \alpha \text{ ftree}_2) (g x))) x)"

\[
\text{primrec } \text{relabel}_\text{ft}_2 :: \forall \alpha. (\alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \text{ ftree}_2) \Rightarrow \alpha \text{ ftree}_2
\]

where

"relabel_ft2 f (FTLeaf2 x) = FTLeaf2 (f x)"
| "relabel_ft2 f (FTNode2 g) = FTNode2 (\alpha x y. relabel_ft2 f (g x y))"

\[
\text{primrec (nonexhaustive) } \text{subtree}_\text{ft}_2 :: \forall \alpha. (\alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \text{ ftree}_2) \Rightarrow \alpha \text{ ftree}_2
\]

where

"subtree_ft2 x y (FTNode2 g) = g x y"

For any datatype featuring nesting, the predicator can be used instead of the map function, typically when defining predicates. For example:

\[
\text{primrec } \text{increasing}_\text{tree} :: \forall m. \text{ int } \Rightarrow \text{ int tree}_f \Rightarrow \text{ bool}
\]

where

"increasing_tree m (Node_f n ts) \iff n \geq m \land \text{ list_all (increasing_tree (n + 1)) ts}"

### 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 though new-style datatypes. For example:

\[
\text{primrec (nonexhaustive)}
\]

\[
\text{at}_f :: \forall \alpha. \text{ tree}_f \Rightarrow \text{ nat list} \Rightarrow \alpha \land
\]

\[
\text{ats}_f :: \forall \alpha. \text{ tree}_f \text{ list} \Rightarrow \text{ nat} \Rightarrow \text{ nat list} \Rightarrow \alpha
\]

where

"at_f (Node_f a ts) js =
3 DEFINING PRIMITIVELY RECURSIVE FUNCTIONS

\[
(\text{case } js \text{ of }
\begin{align*}
&[] \Rightarrow a \\
&j \neq js' \Rightarrow at_{ff} (ts \ j \ js')
\end{align*}
\]
\[
| \text{“} at_{ff} (t \ # \ ts) \ j = \\
(\text{case } j \text{ of }
\begin{align*}
&\text{Zero} \Rightarrow at_{ff} t \\
&\text{Succ } j' \Rightarrow at_{ff} (ts \ j')
\end{align*}
\]
\]

Appropriate induction rules are generated as \texttt{at}_{ff}.\texttt{induct}, \texttt{ats}_{ff}.\texttt{induct}, and \texttt{at}_{ff\_ats}_{ff}.\texttt{induct}. The induction rules and the underlying recursors are generated dynamically and are kept in a cache to speed up subsequent definitions.

Here is a second example:

\texttt{primrec}

\texttt{sum\_btree} :: “’a::{\texttt{zero},\texttt{plus}} \ btree \Rightarrow ‘a” and

\texttt{sum\_btree\_option} :: “’a \ btree\_option \Rightarrow ‘a”

\texttt{where}

“\texttt{sum\_btree} (BNode a lt rt) = \\
a + \texttt{sum\_btree\_option} lt + \texttt{sum\_btree\_option} rt”

| “\texttt{sum\_btree\_option} \texttt{None} = 0”

| “\texttt{sum\_btree\_option} \texttt{(Some} t \texttt{)} = \texttt{sum\_btree} \texttt{t}”

3.2 Command Syntax

3.2.1 primrec

\texttt{primrec} : \texttt{local\_theory} \to \texttt{local\_theory}
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 [12].

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 `nonexhaustive` 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`):
3 DEFINING PRIMITIVELY RECURSIVE FUNCTIONS

\[ f.\text{simp} \text{[simp, code]:} \]
\[
\text{tfold } uu \ (TNil \ y) = y \\
\text{tfold } f \ (TCons \ x \ xs) = f \ x \ (\text{tfold } f \ xs)
\]
The \[\text{code}\] attribute is set by the \[\text{code}\] plugin (Section 8.1).

\[ f.\text{transfer} \text{[transfer_rule]:} \]
\[
\text{rel_fun} \ (\text{rel_fun } R2 \ (\text{rel_fun } R1 \ R1)) \ (\text{rel_fun} \ (\text{rel_tlist } R2 \ R1) \ R1) \ \text{tfold} \ \text{tfold}
\]
This theorem is generated by the \[\text{transfer}\] plugin (Section 8.3) for functions declared with the \[\text{transfer}\] 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 \text{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 \text{overloading} command.
4. Derive the desired equation on \(un.D\) from the characteristic equations for \(un.D_0\).

The following example illustrates this procedure:

\text{consts} \ \text{termi}_0 :: 'a
4 DEFining Codatatypes

**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:

```plaintext
codatatype (lset: 'a) llist =
  lnull: LNil
| LCons (lhd: 'a) (ltl: "'a llist")
for
  map: lmap
  rel: llist_all2
  pred: llist_all
where
  "ltl LNil = LNil"
```

Lazy lists can be infinite, such as \( LCons 0 (LCons 0 (\ldots)) \) and \( LCons 0 (LCons 1 (LCons 2 (\ldots))) \). Here is a related type, that of infinite streams:

```plaintext
codatatype (sset: 'a) stream =
  SCons (shd: 'a) (sdl: "'a stream")
for
  map: smap
  rel: stream_all2
```

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

```plaintext
codatatype enat = EZero | ESucc enat
```

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

Here is an example with many constructors:

```plaintext
codatatype 'a process =
  Fail
| Skip (cont: "'a process")
| Action (prefix: 'a) (cont: "'a process")
| Choice (left: "'a process") (right: "'a process")
```

Notice that the `cont` selector is associated with both `Skip` and `Action`.

### 4.1.2 Mutual Corecursion

The example below introduces a pair of mutually corecursive types:

```plaintext
codatatype even_enat = Even_EZero | Even_ESucc odd_enat
and odd_enat = Odd_ESucc even_enat
```
4 DEFINING CODATATYPES

4.1.3 Nested Corecursion

The next examples feature nested corecursion:

\[
\text{codatatype } 'a\ \text{tree}_i = \text{Node}_i (\text{lbl}_i; 'a) (\text{sub}_i; "'a\ \text{tree}_i\ \text{llist}")
\]

\[
\text{codatatype } 'a\ \text{tree}_s = \text{Node}_s (\text{lbl}_s; 'a) (\text{sub}_s; "'a\ \text{tree}_s\ \text{fset}")
\]

\[
\text{codatatype } 'a\ \text{sm} = \text{SM} (\text{accept}; \text{bool}) (\text{trans}; "'a \Rightarrow 'a\ \text{sm}")
\]

4.2 Command Syntax

4.2.1 codatatype

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

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

4.3 Generated Constants

Given a codatatype \(('a_1, \ldots, 'a_m)\ t\) 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:

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

4.4 Generated Theorems

The characteristic theorems generated by \text{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':

\[
\begin{align*}
t.coinduct \ [ & \text{consumes } m, \text{ case_names } t_1 \ldots t_m, \\
& \text{case_conclusion } D_1 \ldots D_n, \text{ coinduct } t]: \\
& [R \ llist llist' \land llist llist' \implies \ lnull llist = \ lnull llist' \land \\
& (\neg \ lnull llist \implies \neg \ lnull llist') \implies \ lhd llist = \ lhd llist' \land (R \ ltl llist) \\
& (ltl llist'))] \implies llist = llist'
\end{align*}
\]

\[
\begin{align*}
t.coinduct\_strong \ [ & \text{consumes } m, \text{ case_names } t_1 \ldots t_m, \\
& \text{case_conclusion } D_1 \ldots D_n]: \\
& [R \ llist llist' \land llist llist' \implies \ lnull llist = \ lnull llist' \land \\
& (\neg \ lnull llist \implies \neg \ lnull llist') \implies \ lhd llist = \ lhd llist' \land (R \ ltl llist) \\
& (ltl llist')) \lor (ltl llist = ltl llist'))] \implies llist = llist'
\end{align*}
\]

\[
\begin{align*}
t.rel\_coinduct \ [ & \text{consumes } m, \text{ case_names } t_1 \ldots t_m, \\
& \text{case_conclusion } D_1 \ldots D_n, \text{ coinduct pred}]: \\
& [P \ x \ y; \ llist llist' \implies \ lnull llist = \ lnull llist' \land (\neg \ lnull llist \implies \neg \ lnull llist') \implies R \ (lhd llist) \ (lhd llist') \land P \ (ltl llist) \ (ltl llist'))] \implies llist-all2 \ R \ x \ y
\end{align*}
\]

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

\[
\begin{align*}
t1\ldots t_m.coinduct \ [ & \text{case_names } t_1 \ldots t_m, \text{ case_conclusion } D_1 \ldots D_n] \\
t1\ldots t_m.coinduct\_strong \ [ & \text{case_names } t_1 \ldots t_m, \\
& \text{case_conclusion } D_1 \ldots D_n]:
\end{align*}
\]

\[
\begin{align*}
t1\ldots t_m.rel\_coinduct \ [ & \text{case_names } t_1 \ldots t_m, \\
& \text{case_conclusion } D_1 \ldots D_n]:
\end{align*}
\]

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

4.5 Antiquotation

4.5.1 codatatype

The codatatype antiquotation, written \<\<\texttt{codatatype} \(t\)>\ or \@{\texttt{codatatype} \(t\)}\, where \(t\) is a type name, expands to \LaTeX\ code for the definition of
5 Defining Primitively Corecursive Functions

Corecursive functions can be specified using the `primcorec` and `primcorecursive` commands, which support primitive corecursion. Other approaches include the more general `partial_function` command, the `corec` and `corecursive` commands, and techniques based on domains and topologies [8]. In this tutorial, the focus is on `primcorec` and `primcorecursive`; `corec` and `corecursive` are described in a separate tutorial [3]. More examples can be found in the directories `~/src/HOL/Datatype_Examples` and `~/src/HOL/Corec_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
  \[ \ldots \implies \text{is	extunderscore C}_j \ (f \ x_1 \ldots \ x_n) \]
  and equations of the form
  \[ \text{un	extunderscore C}_j i \ (f \ x_1 \ldots \ x_n) = \ldots \]
  This style is popular in the coalgebraic literature.

- The constructor view specifies $f$ by equations of the form
  \[ \ldots \implies f \ x_1 \ldots \ x_n = \text{C}_j \ldots \]
  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 DEFINING PRIMITIVELY CORECURSIVE FUNCTIONS

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:

\[
\text{primcorec literate :: "'(a ⇒ 'a) ⇒ 'a ⇒ 'a llist" where}
\quad \text{"literate g x = LCons x (literate g (g x))"}
\]

\[
\text{primcorec siterate :: "'(a ⇒ 'a) ⇒ 'a ⇒ 'a stream" where}
\quad \text{"siterate g x = SCons x (siterate g (g x))"}
\]

The constructor ensures that progress is made—i.e., the function is produc-
tive. The above functions compute the infinite lazy list or stream \([x, g x, g (g x), \ldots]\). Productivity guarantees that prefixes \([x, g x, g (g x), \ldots, (g \isasymwedge k) x]\) 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:

\[
\text{primcorec every_snd :: "'a stream ⇒ 'a stream" where}
\quad \text{"every_snd s = SCons (shd s) (stl (stl s))"}
\]

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

\[
\text{primcorec lapp :: "'a llist ⇒ 'a llist ⇒ 'a llist" where}
\quad \text{"lapp xs ys =}
\quad \quad \text{(case xs of}
\quad \quad \quad \text{LNil ⇒ ys}
\quad \quad \quad \text{| LCons x xs' ⇒ LCons x (lapp xs' ys))"}
\]

For technical reasons, case−of is only supported for case distinctions on
(co)datatypes that provide discriminators and selectors.

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

\textbf{primcorec} infty :: enat \textbf{where}
  \texttt{"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 \((n)\):

\textbf{primcorec}
  random\_process :: "\texttt{\'a stream} ⇒ \texttt{(int ⇒ int)} ⇒ \texttt{int} ⇒ \texttt{\'a process}"
  \textbf{where}
  \texttt{"random\_process \textit{s f n} =}
  \begin{align*}
  & (\textit{if n mod 4 = 0 then} \\
  & \quad \text{Fail} \\
  & \text{else if n mod 4 = 1 then} \\
  & \quad \text{Skip (random\_process \textit{s f (f n)})} \\
  & \text{else if n mod 4 = 2 then} \\
  & \quad \text{Action (shd s) (random\_process (stl s) f (f n))} \\
  & \text{else} \\
  & \quad \text{Choice (random\_process (every\_snd s) (f o f) (f n))} \\
  & \quad (\text{random\_process (every\_snd (stl s)) (f o f) (f (f n)))")}
\end{align*}

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:

\textbf{primcorec}
  even\_infty :: even\_enat \textbf{and}
  odd\_infty :: odd\_enat
  \textbf{where}
  \texttt{"even\_infty = Even\_ESucc odd\_infty”}
  | \texttt{"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:

\[
\begin{align*}
\text{primcorec iterate}_{i i} :: & 
\text{"'(a ⇒ 'a llist) ⇒ 'a ⇒ 'a tree}_{i i}" \text{ where} \\
& \text{"iterate}_{i i} \ g \ x = \text{Node}_{i i} \ x \ \text{(lmap (iterate}_{i i} \ g \ (g \ x))}"
\end{align*}
\]

\[
\begin{align*}
\text{primcorec iterate}_{i s} :: & 
\text{"'(a ⇒ 'a fset) ⇒ 'a ⇒ 'a tree}_{i s}" \text{ where} \\
& \text{"iterate}_{i s} \ g \ x = \text{Node}_{i s} \ x \ \text{(fimage (iterate}_{i s} \ g \ (g \ x))}"
\end{align*}
\]

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:

\[
\begin{align*}
\text{primcorec tree}_{i i} \_\text{of_stream} :: & 
\text{"a stream ⇒ 'a tree}_{i i}" \text{ where} \\
& \text{"tree}_{i i} \_\text{of_stream} \ s = \\
& \text{Node}_{i i} \ (\text{shd} \ s) \ \text{(lmap tree}_{i i} \_\text{of_stream} \ (\text{LCons} \ (\text{stl} \ s) \ \text{LNil}))}"
\end{align*}
\]

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

\[
\begin{align*}
\text{primcorec tree}_{i i} \_\text{of_stream} :: & 
\text{"a stream ⇒ 'a tree}_{i i}" \text{ where} \\
& \text{"tree}_{i i} \_\text{of_stream} \ s = \\
& \text{Node}_{i i} \ (\text{shd} \ s) \ \text{(LCons (tree}_{i i} \_\text{of_stream} \ (\text{stl} \ s) \ \text{LNil})}"
\end{align*}
\]

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:

\[
\begin{align*}
\text{primcorec sm_of_dfa :: "'(q ⇒ 'a ⇒ 'q) ⇒ 'q set ⇒ 'q ⇒ 'a sm" where} \\
& \text{"sm_of_dfa} \ \delta \ F \ q = \text{SM} \ (q \in F) \ (\text{sm_of_dfa} \ \delta \ F \circ \delta \ q)"
\end{align*}
\]

The map function for the function type \((⇒)\) is composition \((\circ)\). For convenience, corecursion through functions can also be expressed using \(\lambda\)-abstractions and function application rather than through composition. For example:

\[
\begin{align*}
\text{primcorec sm_of_dfa :: "'(q ⇒ 'a ⇒ 'q) ⇒ 'q set ⇒ 'q ⇒ 'a sm" where} \\
& \text{"sm_of_dfa} \ \delta \ F \ q = \text{SM} \ (q \in F) \ (\lambda a. \text{sm_of_dfa} \ \delta \ F \ (\delta \ q \ a))"
\end{align*}
\]

\[
\text{primcorec empty_sm :: "'a sm" where}
\]

5 DEFINING PRIMITIVELY CORECURSIVE FUNCTIONS

“empty_sm = SM False (λ_. empty_sm)”

primcorec not_sm :: “a sm ⇒ ’a sm” where
“not_sm M = SM (¬ accept M) (λa. not_sm (trans M a))”

primcorec or_sm :: “a sm ⇒ ’a sm ⇒ ’a sm” where
“or_sm M N = SM (accept M ∨ accept N) (λa. or_sm (trans M a) (trans N a))”

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

codatatype (’a, ’b) sm2 =
SM2 (accept2: bool) (trans2: “’a ⇒ ’b ⇒ (’a, ’b) sm2”)

primcorec
sm2_of_dfa :: “(’q ⇒ ’a ⇒ ’b ⇒ ’q set ⇒ ’q ⇒ (’a, ’b) sm2)” where
“sm2_of_dfa δ F q = SM2 (q ∈ F) ((◦) ((◦) (sm2_of_dfa δ F)) (δ q))”

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:

primcorec
iterateii : “(’a ⇒ ’a llist) ⇒ ’a ⇒ ’a treeii” and
iteratesii : “(’a ⇒ ’a llist) ⇒ ’a llist ⇒ ’a treeii llist” where
“iterateii g x = Nodeii x (iteratesii g (g x))”
| “iteratesii g xs =
  (case xs of
    LNil ⇒ LNil
  | LCons x xs’ ⇒ LCons (iterateii g x) (iteratesii g xs’))”

Coinduction rules are generated as iterateii.coinduct, iteratesii.coinduct, and iterateii.iteratesii.coinduct and analogously for coinduct_strong. These rules and the underlying corecursors are generated dynamically and are kept in a cache to speed up subsequent definitions.
5 DEFINING PRIMITIVELY CORECURSIVE FUNCTIONS

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 \( \text{literate} \) and \( \text{siterate} \), are identical in both styles.

Here is an example where there is a difference:

\[
\text{primcorec} \ lapp ::= \"'a llist } \Rightarrow \ 'a llist ) \Rightarrow \ 'a llist } \" \text{ where} \\
\text{\"lnull } xs } \Rightarrow \ \text{lnull } ys } \Rightarrow \ \text{lapp } xs } \ y} = \text{LNil} \" \\
| \"'_ } \Rightarrow \ \text{lapp } xs } \ y} = \text{LCons } (\text{hd } (\text{if lnull } xs } \text{then } ys } \text{else } xs ) \\
\text{\{(if } xs } = \text{LNil then } \text{ltl } ys } \text{else } \text{lapp } (\text{ltl } xs} } \ y)\" \\
\]

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 (\( \text{ys} \)). 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:

\[
\text{primcorec} \\
\text{random_process} ::= \"'a stream } \Rightarrow (\text{int } \Rightarrow \text{int} ) \Rightarrow \text{int } \Rightarrow \ 'a process } \" \\
\text{ where} \\
\text{\"n mod } 4 = 0 \Rightarrow \text{random_process } s } f } n = \text{Fail} \" \\
| \"n mod } 4 = 1 \Rightarrow \text{random_process } s } f } n = \text{Skip } (\text{random_process } s } f } (f } n))\" \\
| \"n mod } 4 = 2 \Rightarrow \text{random_process } s } f } n = \text{Action } (\text{shd } s} } ) (\text{random_process } (\text{stl } s} ) f} (f } n))\" \\
| \"n mod } 4 = 3 \Rightarrow \text{random_process } s } f } n = \text{Choice } (\text{random_process } (\text{every_snd } s} ) f} (f } n)) \\
\text{(random_process } (\text{every_snd } (\text{stl } s}) f} (f } n))\" \\
\]

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

\begin{verbatim}
primcorec literate :: "(\textquotesingle a \Rightarrow \textquotesingle a) \Rightarrow \textquotesingle a \Rightarrow \textquotesingle a llist" where
  "\textit{lnull} (literate _ x)"
| "\textit{lhd} (literate _ x) = x"
| "\textit{ltl} (literate g x) = literate g (g x)"

primcorec siterate :: "(\textquotesingle a \Rightarrow \textquotesingle a) \Rightarrow \textquotesingle a \Rightarrow \textquotesingle a stream" where
  "\textit{shd} (siterate _ x) = x"
| "\textit{stl} (siterate g x) = siterate g (g x)"

primcorec every_snd :: "\textquotesingle a stream \Rightarrow \textquotesingle a stream" where
  "\textit{shd} (every_snd s) = shd s"
| "\textit{stl} (every_snd s) = stl (stl s)"
\end{verbatim}

The first formula in the \textit{local.literate} specification indicates which constructor to choose. For \textit{local.siterate} and \textit{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 lapp :: "\textquotesingle a llist \Rightarrow \textquotesingle a llist \Rightarrow \textquotesingle a llist" where
  "\textit{lnull} xs \Rightarrow \textit{lnull} ys \Rightarrow \textit{lnull} (lapp xs ys)"
| "\textit{lhd} (lapp xs ys) = lhd (if \textit{lnull} xs then ys else xs)"
| "\textit{ltl} (lapp xs ys) = (if xs = LNil then \textit{ltl} ys else lapp (\textit{ltl} xs) ys)"
\end{verbatim}

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

\begin{verbatim}
  "\textit{lnull} xs \Rightarrow \neg \textit{lnull} (lapp xs ys)"
\end{verbatim}

to the specification. The generated selector theorems are conditional.

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

\begin{verbatim}
primcorec
  random_process :: "\textquotesingle a stream \Rightarrow (\text{int} \Rightarrow \text{int}) \Rightarrow \text{int} \Rightarrow \text{\textquotesingle a process}"
where
  "n \text{ mod} 4 = 0 \Rightarrow \text{random_process s f n = Fail}"
\end{verbatim}
5 DEFINING PRIMITIVELY CORECURSIVE FUNCTIONS

| “n mod 4 = 1 ⇒ is_Skip (random_process s f n)” |
| “n mod 4 = 2 ⇒ is_Action (random_process s f n)” |
| “n mod 4 = 3 ⇒ 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 ≠ Even_EZero”
| “un_Even_ESucc even_infty = odd_infty”
| “un_Odd_ESucc odd_infty = even_infty”

primcorec iterate :: (′a ⇒ ′a list) ⇒ ′a ⇒ ′a tree
where
  “lbl iter :: (′a ⇒ ′a list) ⇒ ′a ⇒ ′a tree”
| “sub iter :: (′a ⇒ ′a list) ⇒ ′a ⇒ ′a tree”

5.2 Command Syntax

5.2.1 primcorec and primcorecursive

primcorec : local_theory → local_theory
primcorecursive : local_theory → 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 [12].

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 \texttt{primcorec} command is an abbreviation for \texttt{primcorecursive} with \texttt{by auto} to discharge any emerging proof obligations.

### 5.3 Generated Theorems

The \texttt{primcorec} and \texttt{primcorecursive} commands generate the following properties (listed for \texttt{literate}):

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

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

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

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

- \texttt{f.sel [simp, code]}:
  \[
  \neg \text{lnull } (\text{literate } g \; x)
  \]
  The [\texttt{code}] attribute is set by the \texttt{code} plugin (Section 8.1).

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

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

- \texttt{f.coinduct [consumes m, case_names t_1 \ldots t_m, case_conclusion D_1 \ldots D_n]}:
  This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4).
6 REGISTERING BOUNDED NATURAL FUNCTORS

\texttt{f.coinduct\_strong} [\texttt{consumes \( m \)}, \texttt{case\_names \( t_1 \ldots t_m \)},
\texttt{case\_conclusion \( D_1 \ldots D_n \)}]:

This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4).

\texttt{f_1\ldots f_m.coinduct} [\texttt{case\_names \( t_1 \ldots t_m \)},
\texttt{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.

\texttt{f_1\ldots f_m.coinduct\_strong} [\texttt{case\_names \( t_1 \ldots t_m \)},
\texttt{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:

\[ \texttt{f.simps} = \texttt{f.disc iff} \text{ (or \texttt{f.disc}) \ t.sel} \]

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 [4,11].

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, \( \forall a \ \text{llist} \) is a unary BNF. Its predicator \( \text{llist\_all} :: (\forall a \Rightarrow \text{bool}) \Rightarrow \forall a \text{llist} \Rightarrow \text{bool} \) extends unary predicates over elements to unary predicates over lazy lists. Similarly, its relator \( \text{llist\_all2} :: (\forall a \Rightarrow b \Rightarrow \text{bool}) \Rightarrow \forall a \text{llist} \Rightarrow b \text{llist} \Rightarrow \text{bool} \) extends binary predicates over elements to binary predicates over parallel lazy lists. The
6 REGISTERING BOUNDED NATURAL FUNCTORS

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 datatype and 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 bnf command. Some of the proof obligations are best viewed with the bundle 'cardinal_syntax' included.

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

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 “(_)” .
lift_definition set_fn :: “('d, 'a) fn ⇒ 'a set” is range .
lift_definition pred_fn :: “('a ⇒ bool) ⇒ ('d, 'a) fn ⇒ bool”
is “pred_fun (λ_. True)” .
lift_definition rel_fn :: “('a ⇒ 'b ⇒ bool) ⇒ ('d, 'a) fn ⇒ ('d, 'b) fn ⇒ bool”
is “rel_fun (=)” .
bnf “('d, 'a) fn”
map: map_fn
sets: set_fn
bd: “natLeq + c | UNIV :: 'd set|”
rel: rel_fn
pred: pred_fn
proof –
show “map_fn id = id”
by \textit{transfer auto}

next
fix \( f :: "a \Rightarrow b" \) and \( g :: "b \Rightarrow c" \)
show "\text{map}_\text{fn} (g \circ f) = \text{map}_\text{fn} g \circ \text{map}_\text{fn} f"
by \textit{transfer (auto simp add: \textit{comp_def})}

next
fix \( F :: "(d, a) \mathsf{fn}" \) and \( f g :: "a \Rightarrow b" \)
assume "\( \forall x. x \in \text{set}_\text{fn} F \implies f x = g x \)"
then show "\text{map}_\text{fn} f F = \text{map}_\text{fn} g F"
by \textit{transfer auto}

next
fix \( f :: "a \Rightarrow b" \)
show "\text{set}_\text{fn} \circ \text{map}_\text{fn} f = (\cdot) \circ \text{set}_\text{fn}"
by \textit{transfer (auto simp add: \textit{comp_def})}

next
show "\text{card}_\text{order} (\text{natLeq} + c \mid \text{UNIV} :: 'd \ set\ )"
apply \textit{(rule card_order_csum)}
apply \textit{(rule natLeq_card_order)}
by \textit{(rule card_of_card_order_on)}

next
show "\text{cinfinite} (\text{natLeq} + c \mid \text{UNIV} :: 'd \ set\ )"
apply \textit{(rule cinfinite_csum)}
apply \textit{(rule \textit{disjI1})}
by \textit{(rule natLeq_cinfinite)}

next
fix \( F :: "(d, a) \mathsf{fn}" \)
have "\( |\text{set}_\text{fn} F| \leq o \mid \text{UNIV} :: 'd \ set\ | (\text{is } _\leq o ?U)\)"
by \textit{transfer (rule card_of_image)}
also have "\( ?U \leq o \text{natLeq} + c ?U \)"
by \textit{(rule ordLeq_csum2) (rule card_of_card_order)}
finally show "\( |\text{set}_\text{fn} F| \leq o \text{natLeq} + c \mid \text{UNIV} :: 'd \ set\ | \)".

next
fix \( R :: "a \Rightarrow b \Rightarrow \text{bool}" \) and \( S :: "b \Rightarrow c \Rightarrow \text{bool}" \)
show "\text{rel}_\text{fn} R \ \text{OO} \ \text{rel}_\text{fn} S \leq \text{rel}_\text{fn} (R \ \text{OO} \ S)"
by \textit{(rule, transfer) (auto simp add: \textit{rel_fun_def})}

next
fix \( R :: "a \Rightarrow b \Rightarrow \text{bool}" \)
show "\text{rel}_\text{fn} R = (\lambda x y. \exists z. \text{set}_\text{fn} z \subseteq \{(x, y). R x y\} \land \text{map}_\text{fn} \text{fst} z = x \land \text{map}_\text{fn} \text{snd} z = y\)"
\textit{unfolding \textit{fun_eq_iff relcompp.simps conversep.simps}}
by \textit{transfer (force simp: \textit{rel_fun_def subset_iff})}

next
fix \( P :: "a \Rightarrow \text{bool}" \)
show "pred_fn P = (λx. Ball (set_fn x) P)"
unfolding fun_eq_iff by transfer simp
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/Countable_Set_Type.thy, ~/src/HOL/Library/FSet.thy, and ~/src/HOL/Library/Multiset.thy for further examples of BNF registration, some of which feature custom witnesses.

For many typedefs and quotient types, lifting the BNF structure from the raw typ to the abstract type can be done uniformly. This is the task of the lift_bnf command. Using lift_bnf, the above registration of ('d, 'a) fn as a BNF becomes much shorter:

lift_bnf ('d, 'a) fn
by force+

For type copies (typedefs with UNIV as the representing set), the proof obligations are so simple that they can be discharged automatically, yielding another command, copy_bnf, which does not emit any proof obligations:

copy_bnf ('d, 'a) fn

Since record schemas are type copies, copy_bnf can be used to register them as BNFs:

record 'a point =
  xval :: 'a
  yval :: 'a

copy_bnf ('a, 'z) point_ext

In the general case, the proof obligations generated by lift_bnf are simpler than the actual BNF properties. In particular, no cardinality reasoning is required. Consider the following type of nonempty lists:

typedef 'a nonempty_list = "{xs :: 'a list. xs ≠ []}" by auto

The lift_bnf command requires us to prove that the set of nonempty lists is closed under the map function and the zip function. The latter only occurs implicitly in the goal, in form of the variable zs.

lift_bnf 'a nonempty_list
proof –
fix f and xs :: "'a list"
assume "xs ∈ {xs. xs ≠ []}"
then show "map f xs ∈ {xs. xs ≠ []}"
  by (cases xs) auto
next
fix zs :: "('a × 'b) list"
assume "map fst zs ∈ {xs. xs ≠ []}" "map snd zs ∈ {xs. xs ≠ []}"
then show "∃ zs'∈{xs. xs ≠ []}. set zs' ⊆ set zs ∧
  map fst zs' = map fst zs ∧
  map snd zs' = map snd zs"
  by (cases zs) (auto intro: exI[of _ zs])
qed

The lift_bnf command also supports quotient types. Here is an example
that defines the option type as a quotient of the sum type. The proof obli-
gations generated by lift_bnf for quotients are different from the ones for
typedefs. You can find additional examples of usages of lift_bnf for both
quotients and subtypes in the session HOL-Datatype_Examples.

inductive ignore_Inl :: "'a + 'a ⇒ 'a + 'a ⇒ bool" where
  "ignore_Inl (Inl x) (Inl y)"
| "ignore_Inl (Inr x) (Inr x)"

lemma ignore_Inl_equivp:
  "ignore_Inl x x"
| "ignore_Inl x y ⇒ ignore_Inl y x"
| "ignore_Inl x y ⇒ ignore_Inl y z ⇒ ignore_Inl x z"
  by (cases x; cases y; cases z; auto)+

quotient_type 'a myoption = "'a + 'a" / ignore_Inl
unfolding equivp_reflp_symp_transp reflp_def symp_def transp_def
by (blast intro: ignore_Inl_equivp)

lift_bnf 'a myoption
proof –
fix P :: "'a ⇒ 'b ⇒ bool" and Q :: "'b ⇒ 'c ⇒ bool"
assume "P OO Q ≠ bot"
then show "rel_sum P P OO ignore_Inl OO rel_sum Q Q 
  ≤ ignore_Inl OO rel_sum (P OO Q) (P OO Q) OO ignore_Inl"
  by (fastforce)
next
fix \( S \) :: "\( 'a \text{ set set} \)"
let \(?eq = \{(x, x'). \text{ignore}_\text{Inl} x x\}\)"
let \(?in = "\( \lambda A. \{x. \text{Basic_BNFs.setl}_x \cup \text{Basic_BNFs.setr}_x \subseteq A\}\)"
assume "\( S \neq \{\} \)" "\( \bigcap S \neq \{\} \)"
show "\( (\bigcap A \in S. ?eq \ "?in A) \subseteq ?eq \ (\bigcap S) \)"
proof (intro subsetI)
  fix \( x \)
  assume "\( x \in (\bigcap A \in S. ?eq \ "?in A) \)"
  with \( \bigcap S \neq \{\}\) show "\( x \in ?eq \ "?in (\bigcap S) \)"
  by (cases \( x \)) (fastforce)+
qed

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

\textbf{bnf_axiomatization} (setA: '\( 'a \), setB: '\( 'b \), setC: '\( 'c \) F
[wits: "\( 'a \Rightarrow (\ 'a, 'b, 'c) F\)"

\textbf{print_theorems}
\textbf{print_bnfs}

6.3 Command Syntax
6.3.1 bnf

\begin{verbatim}
bnf : local_theory \rightarrow proof(prove)
\end{verbatim}
The \texttt{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, predicators, relators, and nonemptiness witnesses can be specified; otherwise, default versions are used.

The syntactic entity \texttt{target} can be used to specify a local context, \texttt{type} denotes a HOL type, and \texttt{term} denotes a HOL term [12].

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

\subsection{lift\_bnf}

\begin{verbatim}
lift\_bnf : local\_theory \rightarrow proof(prove)
\end{verbatim}
The `lift_bnf` command registers as a BNF an existing type (the *abstract type*) that was defined as a subtype of a BNF (the *raw type*) using the `typedef` command or as a quotient type of a BNF (also, the *raw type*) using the `quotient_type`. To achieve this, it lifts the BNF structure on the raw type to the abstract type following a `type_definition` or a `Quotient` theorem. The theorem is usually inferred from the type, but can also be explicitly supplied by means of the optional `via` clause. In case of quotients, it is sometimes also necessary to supply a second theorem of the form `reflp eq`, that expresses...
the reflexivity (and thus totality) of the equivalence relation. In addition, custom names for the set functions, the map function, the predicator, and the relator, as well as nonemptiness witnesses can be specified.

Nonemptiness witnesses are not lifted from the raw type’s BNF, as this would be incomplete. They must be given as terms (on the raw type) and proved to be witnesses. The command warns about witness types that are present in the raw type’s BNF but not supplied by the user. The warning can be disabled by specifying the \texttt{no\_warn\_wits} option.

### 6.3.3 \texttt{copy\_bnf}

\texttt{copy\_bnf} : \textit{local\_theory} $\rightarrow$ \textit{local\_theory}

\texttt{cb-options}

The \texttt{copy\_bnf} command performs the same lifting as \texttt{lift\_bnf} for type copies (\texttt{typedef}s with \textit{UNIV} as the representing set), without requiring the user to discharge any proof obligations or provide nonemptiness witnesses.

### 6.3.4 \texttt{bnf\_axiomatization}

\texttt{bnf\_axiomatization} : \textit{local\_theory} $\rightarrow$ \textit{local\_theory}
The `bnf_axiomatization` command declares a new type and associated constants (map, set, predicador, 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`, . . . ), `mixfix` denotes the usual parenthesized mixfix notation, and `types` denotes a space-separated list of types [12].

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 `~/src/HOL/Library/BNF_Axiomatization.thy` theory to use this functionality.
6.3.5 print_bnfs

\texttt{print\_bnfs} : \texttt{local\_theory} \rightarrow

7 Deriving Destructors and Constructor Theorems

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} : \texttt{local\_theory} \rightarrow \texttt{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 [12].

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**.

### 7.1.2 simps_of_case

**simps_of_case** : `local_theory` → `local_theory`
The \texttt{simps\_of\_case} command provided by theory \texttt{-/src/HOL/Library/Simps\_Case\_Conv.thy} converts a single equation with a complex case expression on the right-hand side into a set of pattern-matching equations. For example,

\begin{verbatim}
  simps_of_case lapp_simps: lapp.code
\end{verbatim}

translates \( lapp \, xs \, ys = (\text{case} \, xs \, \text{of} \, \text{LNil} \Rightarrow \, ys \mid \text{LCons} \, x \, xs' \Rightarrow \, \text{LCons} \, x \, (lapp \, xs') \, ys) \) into

\begin{align*}
  lapp \, \text{LNil} \, ys &= ys \\
  lapp \, (\text{LCons} \, x a \, x) \, ys &= \text{LCons} \, x a \, (lapp \, x \, ys)
\end{align*}

\subsection{7.1.3 \texttt{case\_of\_simps}}

\begin{verbatim}
  case_of_simps : local_theory \to local_theory
\end{verbatim}

The \texttt{case\_of\_simps} command provided by theory \texttt{-/src/HOL/Library/Simps\_Case\_Conv.thy} converts a set of pattern-matching equations into single equation with a complex case expression on the right-hand side (cf. \texttt{simps\_of\_case}). For example,

\begin{verbatim}
  case_of_simps lapp_case: lapp_simps
\end{verbatim}

translates

\begin{align*}
  lapp \, \text{LNil} \, ys &= ys \\
  lapp \, (\text{LCons} \, x a \, x) \, ys &= \text{LCons} \, x a \, (lapp \, x \, ys)
\end{align*}

into \( lapp \, x b a \, x 3 a = (\text{case} \, x b a \, \text{of} \, \text{LNil} \Rightarrow x 3 a \mid \text{LCons} \, x 2 b a \, x 1 b a \Rightarrow \text{LCons} \, x 2 b a \, (lapp \, x 1 b a \, x 3 a)) \).
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:

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

Beyond the standard plugins, the Archive of Formal Proofs includes a derive command that derives class instances of datatypes [10].

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) ≡ 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 `t`, the size plugin generates a generic size function `t.size_t` as well as a specific instance `size :: t ⇒ nat` belonging to the size type class.
The **fun** command relies on **size** to prove termination of recursive functions on datatypes.

The plugin derives the following properties:

\[ \text{t.size [simp, code]}: \]
\[
\begin{align*}
\text{size_list x} & = 0 \\
\text{size_list x (x21 \ # \ x22)} & = x \times 21 + \text{size_list x} \times x22 + \text{Suc} \ 0 \\
\text{size} & = 0 \\
\text{size (x21 \ # \ x22)} & = \text{size x22} + \text{Suc} \ 0
\end{align*}
\]

\[ \text{t.size_gen:} \]
\[
\begin{align*}
\text{size_list x} & = 0 \\
\text{size_list x (x21 \ # \ x22)} & = x \times 21 + \text{size_list x} \times x22 + \text{Suc} \ 0
\end{align*}
\]

\[ \text{t.size_gen_o_map:} \]
\[
\text{size_list f} \circ \text{map g} = \text{size_list (f \circ g)}
\]

\[ \text{t.size_neq:} \]
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**.

The **t.size** and **t.size_t** functions generated for datatypes defined by nested recursion through a datatype **u** depend on **u.size_u**.

If the recursion is through a non-datatype **u** with type arguments 'a₁, . . ., 'aₘ, by default **u** values are given a size of 0. This can be improved upon by registering a custom size function of type ('a₁ \Rightarrow \text{nat}) \Rightarrow . . . \Rightarrow ('aₘ \Rightarrow \text{nat}) \Rightarrow \text{u} \Rightarrow \text{nat} using the ML function **BNF_LFP_Size.register_size** or **BNF_LFP_Size.register_size_global**. See theory ~/src/HOL/Library/Multiset.thy for an example.

### 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 at least one live type argument and no dead type arguments, the plugin derives the following properties:

\[ \text{t.Domainp_rel [relator_domain]}: \]
\[
\text{Domainp (list_all2 R) = list_all (Domainp R)}
\]

\[ \text{t.left_total_rel [transfer_rule]}: \]
\[
\text{left_total R \implies left_total (list_all2 R)}
\]
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:

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

In addition, the plugin sets the [relator_eq] attribute on a variant of the t.rel_eq_onp property, 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:

- quickcheck_random
- quickcheck_exhaustive
- quickcheck_bounded_forall
- quickcheck_full_exhaustive
- 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 of the (co)datatype package at the time of this writing.

1. *Defining mutually (co)recursive (co)datatypes can be slow.* Fortunately, it is always possible to recast mutual specifications to nested ones, which are processed more efficiently.

2. *Locally fixed types and terms cannot be used in type specifications.* The limitation on types 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. *The primcorec command does not allow corecursion under case–of for datatypes that are defined without discriminators and selectors.*

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

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

8. The compatibility layer sometimes produces induction principles with a slightly different ordering of the premises than the old 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 Kunčar 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. Gerwin Klein and Lars Noschinski implemented the simps_of_case and case_of.simps commands. Florian Haftmann, Christian Urban, and Makarius Wenzel 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. Mamoun Filali-Amine, Gerwin Klein, Andreas Lochbihler, Tobias Nipkow, and Christian Sternagel suggested many textual improvements to this tutorial.

References


REFERENCES


