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

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

15 August 2018

Abstract
This tutorial describes the definitional package for datatypes and codatatypes, and for primitively recursive and corecursive functions, in Isabelle/HOL. The following commands are provided: datatypen, datatype_compat, primrec, codatatype, primcorec, primcorecursive, bnf, lift_bnf, copy_bnf, bnf_axiomatization, print_bnf, 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
### Contents

2.4.2 Functorial Theorems ........................................ 16
2.4.3 Inductive Theorems ........................................ 21
2.5 Proof Method .................................................... 22
  2.5.1 countable_datatype ....................................... 22
2.6 Compatibility Issues .......................................... 22

3 Defining Primitively Recursive Functions ...................... 23
  3.1 Introductory Examples ........................................ 23
    3.1.1 Nonrecursive Types ..................................... 23
    3.1.2 Simple Recursion ....................................... 24
    3.1.3 Mutual Recursion ..................................... 25
    3.1.4 Nested Recursion ..................................... 26
    3.1.5 Nested-as-Mutual Recursion ............................ 27
  3.2 Command Syntax ................................................ 28
    3.2.1 primrec ................................................... 28
  3.3 Generated Theorems .......................................... 29
  3.4 Recursive Default Values for Selectors ...................... 29
  3.5 Compatibility Issues ........................................ 30

4 Defining Codatatypes ............................................. 31
  4.1 Introductory Examples ........................................ 31
    4.1.1 Simple Corecursion .................................... 31
    4.1.2 Mutual Corecursion ................................... 32
    4.1.3 Nested Corecursion ................................... 32
  4.2 Command Syntax ................................................ 32
    4.2.1 codatatype ................................................ 32
  4.3 Generated Constants .......................................... 33
  4.4 Generated Theorems .......................................... 33
    4.4.1 Coinductive Theorems .................................. 33

5 Defining Primitively Corecursive Functions ................... 35
  5.1 Introductory Examples ........................................ 36
    5.1.1 Simple Corecursion .................................... 36
    5.1.2 Mutual Corecursion ................................... 37
    5.1.3 Nested Corecursion ................................... 38
    5.1.4 Nested-as-Mutual Corecursion .......................... 39
    5.1.5 Constructor View ....................................... 40
    5.1.6 Destructor View ....................................... 41
  5.2 Command Syntax ................................................ 42
    5.2.1 primcorec and primcorecursive ........................ 42
  5.3 Generated Theorems .......................................... 44
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:

\begin{verbatim}
datatype 'a treefs = Nodefs (lblfs: 'a) (subfs: "'a treefs fset")
\end{verbatim}

Another strong point is the support for local definitions:

\begin{verbatim}
context linorder
begin
datatype flag = Less | Eq | Greater
end
\end{verbatim}
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:

\[
\text{codatatype } \alpha \text{llist} = \text{LNil} \mid \text{LCons } \alpha \text{llist}
\]

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

\[
\begin{align*}
\text{datatype } \alpha \text{treelf} & = \text{Nodeff } \alpha \text{ treelf llist} \\
\text{datatype } \alpha \text{treelfi} & = \text{Nodeffi } \alpha \text{ treelfi llist} \\
\text{codatatype } \alpha \text{treelfi} & = \text{Nodeifi } \alpha \text{ treelfi llist} \\
\text{codatatype } \alpha \text{treelli} & = \text{Nodeilli } \alpha \text{ treelli llist}
\end{align*}
\]

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 \texttt{Main}. Additional functionality is provided by the theory \texttt{-/-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.\footnote{However, some of the internal constructions and most of the internal proof obligations are omitted if the \texttt{quick\_and\_dirty} option is enabled.} The package is described in a number of scientific papers [2, 4, 9, 11]. The central notion is that of a \textit{bounded natural functor} (BNF)—a well-behaved type constructor for which nested (co)recursion is supported.

This tutorial is organized as follows:

- Section 2, “Defining Datatypes,” describes how to specify datatypes using the \texttt{datatype} command.
- Section 3, “Defining Primitively Recursive Functions,” describes how to specify functions using \texttt{primrec}. (A separate tutorial [6] describes the more powerful \texttt{fun} and \texttt{function} commands.)
- Section 4, “Defining Codatatypes,” describes how to specify codatatypes using the \texttt{codatatype} command.
Defining Datatypes

Datatypes can be specified using the \texttt{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 \texttt{~/src/HOL/Datatype_Examples}.

2.1.1 Nonrecursive Types

Datatypes are introduced by specifying the desired names and argument types for their constructors. \textit{Enumeration} types are the simplest form of \texttt{datatype}. All their constructors are nullary:

\begin{verbatim}
  datatype trool = Truue | Faalse | Perhaaps
\end{verbatim}

\texttt{Truue}, \texttt{Faalse}, and \texttt{Perhaaps} have the type \texttt{trool}. 

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 \texttt{cl-isabelle-users} mailing list.
Polymorphic types are possible, such as the following option type, modeled after its homologue from the HOL Option theory:

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

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

The next example has three type parameters:

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

The constructor is `Triple :: 'a ⇒ 'b ⇒ 'c ⇒ ('a, 'b, 'c) triple`. Unlike in Standard ML, curried constructors are supported. The uncurried variant is also possible:

```
datatype ('a, 'b, 'c) triple_u = 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:

```
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:

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

```
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:

```
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 option type, follows:

\[
\text{datatype } \mathsf{a} \text{ btree } = \\
\quad \text{BNode } \mathsf{a} \text{ "a btree option" "a btree option"
}\]

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

\[
\text{datatype } \mathsf{a} \text{ wrong } = W1 | W2 "\mathsf{a} \text{ wrong } \Rightarrow \mathsf{a}"
\]

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

\[
\text{datatype } (\mathsf{a}, \mathsf{b}) \text{ fun_copy } = \text{Fun } "\mathsf{a} \Rightarrow \mathsf{b}"
\]

\[
\text{datatype } \mathsf{a} \text{ also_wrong } = W1 | W2 "(\mathsf{a} \text{ also_wrong, } \mathsf{a}) \text{ fun_copy}"
\]

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

\[
\text{datatype } \mathsf{a} \text{ ftree } = \text{FTLeaf } \mathsf{a} | \text{FTNode } "\mathsf{a} \Rightarrow \mathsf{a} \text{ ftree}"
\]

And so is the definition of hereditarily finite sets:

\[
\text{datatype } \text{hfset } = \text{HFSet } "\text{hfset fset}"
\]

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

\[
\text{datatype } \mathsf{a} \text{ pow_list } = \text{PNil } \mathsf{a} | \text{PCons } "(\mathsf{a} * \mathsf{a}) \text{ pow_list}"
\]

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

2.1.5 Auxiliary Constants

The \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:

```plaintext
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"
```

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

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

Discriminator: 
- null :: 'a list ⇒ bool

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

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

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

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

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

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

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

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

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

\[
\text{datatype } (\cdot a, \cdot b) \text{ prod (infixr \"\ast\" 20)} = \text{Pair } \cdot a \cdot b
\]

\[
\text{datatype } (\text{set: } \cdot a) \text{ list } =
\]
  \[
  \text{null: Nil (\"\[]\")}
  \]
  \[
  \mid \text{Cons (hd: } \cdot a) \text{ (tl: \"\cdot a list\") (infixr \"\#\" 65)}
  \]

\[
\text{for}
\]
  \[
  \text{map: map}
  \]
  \[
  \text{rel: list_all2}
  \]
  \[
  \text{pred: list_all}
  \]

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

\[
\text{syntax \"_list\" :: \"args ⇒ \cdot a list\" (\"[\_]\")}
\]

\[
\text{translations}
\]
  \[
  \text{\"[x, xs]\"} == \text{\"x \# [xs]\"}
  \]
  \[
  \text{\"[x]\"} == \text{\"x \# []\"}
  \]

2.2 Command Syntax

2.2.1 datatype

\[
\text{datatype : local_theory \rightarrow local_theory}
\]
The \texttt{datatype} command introduces a set of mutually recursive datatypes specified by their constructors.

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

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

- The \texttt{plugins} option indicates which plugins should be enabled (\textit{only}) or disabled (\textit{del}). By default, all plugins are enabled.
- The \texttt{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 \texttt{where} clause specifies default values for selectors. Each proposition must be an equation of the form \texttt{un\_D (C ...)} = \texttt{...}, where \texttt{C} is a constructor and \texttt{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:

\[
\text{dt-name}
\]

\[
\text{tyargs}
\]

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

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

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

\[ \text{dt-ctor-arg} \]

\[
\begin{array}{c}
\text{name} \\
\text{type}
\end{array}
\]

The syntactic entity \textit{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 \textit{un}\_\textit{C}j\_\textit{i}.

### 2.2.2 datatype_compat

\[ \text{datatype_compat} : \; \text{local theory} \rightarrow \text{local theory} \]

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

\[ \text{datatype_compat even\_nat odd\_nat} \]

\[ \text{ML \langle} \text{Old\_Datatype\_Data.get\_info @\{theory\} @\{type\_name even\_nat\}} \rangle \]

The syntactic entity \textit{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 \(t.C_1, \ldots, t.C_n\), the following auxiliary constants are introduced:

- **Case combinator**: \(t.case_t\) (rendered using the familiar case–of syntax)
- **Discriminators**: \(t.is_{C_1}, \ldots, t.is_{C_n}\)
- **Selectors**: 
  \[
  t.un_{C_11}, \ldots, t.un_{C_1k_1} \\
  \vdots \\
  t.un_{C_n1}, \ldots, t.un_{C_nk_n}
  \]
- **Set functions**: \(t.set_{1\_t}, \ldots, t.set_{m\_t}\)
- **Map function**: \(t.map_t\)
- **Relator**: \(t.rel_t\)
- **Recursor**: \(t.rec_t\)

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 “\(t\)” is an optional component of the names and is normally hidden.

### 2.4 Generated Theorems

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

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

• The inductive theorems (Section 2.4.3) are properties of datatypes related to their inductive nature.

The full list of named theorems can be obtained 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

```plaintext
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]:**
  \[(x21 \ # \ x22 = y21 \ # \ y22) = (x21 = y21 \land x22 = y22)\]

- **t.distinct [simp, induct_simp]:**
  \[[\neq x21 \ # \ x22\]
  \[x21 \ # \ x22 \neq []\]

- **t.exhaust [cases t, case_names C₁ ... Cₙ]:**
  \[
  \begin{align*}
  &y = [] \implies P; \forall x21 \ x22. \ y = x21 \ # \ x22 \implies P] \implies P
  
  &\forall list. \ list = [] \lor (\exists x21 \ x22. \ list = x21 \ # \ x22)
  \end{align*}
  \]

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

- **t.distinct [THEN notE, elim!]:**
  \[
  \begin{align*}
  &[] = x21 \ # \ x22 \implies R
  
  &x21 \ # \ x22 = [] \implies R
  \end{align*}
  \]

The next subgroup is concerned with the case combinator:

- **t.case [simp, code]:**
  \[
  \begin{align*}
  &\text{(case } [] \text{ of } [] \Rightarrow f1 \mid x \ # \ xa \Rightarrow f2 \ x \ xa) = f1
  
  &\text{(case } x21 \ # \ x22 \text{ of } [] \Rightarrow f1 \mid x \ # \ xa \Rightarrow f2 \ x \ xa) = f2 \ x21 \ x22
  \end{align*}
  \]

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

\[t\text{-}case\_cong\text{ [fundef\_cong]}:\]
\[
\begin{align*}
\text{null } [] & \implies f1 = g1; \land x21 \ x22. \text{null}' = x21 \neq x22 \implies \\
\text{null}' = x21 \neq x22 \implies (\text{case list of } [] \implies f1 | x21 \neq x22 \implies \\
f2 \ x21 \ x22) &= (\text{case list of } [] \implies g1 | x21 \neq x22 \implies g2 \ x21 \ x22)
\end{align*}
\]

\[t\text{-}case\_cong\_weak\text{ [cong]}:\]
\[
\text{null } [] \implies (\text{case list of } [] \implies f1 | x \neq xa \implies f2 \ x \ xa)
\]

\[t\text{-}case\_distrib:\]
\[
h. (\text{case list of } [] \implies f1 | x \neq xa \implies f2 \ x \ xa) = (\text{case list of } [] \implies h \ f1 \ x \ xa)
\]

\[t\text{-}split:\]
\[
\text{null } [] \implies f1 \ x \ neq xa \implies f2 \ x \ xa)
\]

\[t\text{-}split\_asm:\]
\[
\text{null } [] \implies (\text{case list of } [] \implies f1 | x \neq xa \implies f2 \ x \ xa)
\]

\[t\text{-}splits = \text{split split\_asm}\]

The third subgroup revolves around discriminators and selectors:

\[t\text{-}disc\text{ [simp]}:\]
\[
\text{null } [] \implies f1 \ x \ neq xa \implies f2 \ x \ xa)
\]

\[t\text{-}discI:\]
\[
\text{null } [] \implies f1 \ x \ neq xa \implies f2 \ x \ xa)
\]

\[t\text{-}sel\text{ [simp, code]}:\]
\[
\text{null } [] \implies f1 \ x \ neq xa \implies f2 \ x \ xa)
\]

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

\[t\text{-}collapse\text{ [simp]}:\]
\[
\text{null } [] \implies f1 \ x \ neq xa \implies f2 \ x \ xa)
\]

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.

\[t\text{-}distinct\_disc\text{ [dest]}:\]
\[
\text{null } [] \implies f1 \ x \ neq xa \implies f2 \ x \ xa)
\]

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

\[
\begin{align*}
null\ list &\implies \neg\ nonnull\ list \\
nonnull\ list &\implies \neg\ null\ list
\end{align*}
\]

\textit{t.exhaust\_disc} \ {\text{[case\_names}\ C_1 \ldots \ C_n]}: \\
\[
null\ list \implies P; \neg\ null\ list \implies P \implies P
\]

\textit{t.exhaust\_sel} \ {\text{[case\_names}\ C_1 \ldots \ C_n]}: \\
\[
\text{[list} = \text{]} \implies P; \text{list} = \text{hd\ list\ #}\ \text{tl\ list} \implies P \implies P
\]

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

\textit{t.split\_sel}: \\
\[
P (\text{case\ list\ of}\ [\text{]} \Rightarrow f_1 | x \#\ xa \Rightarrow f_2 \ x\ xa) = ((\text{list} = \text{]} \implies P\ f_1) \\
\wedge\ (\text{list} = \text{hd\ list\ #}\ \text{tl\ list} \implies P (f_2 (\text{hd\ list}) (\text{tl\ list})))
\]

\textit{t.split\_sel\_asm}: \\
\[
P (\text{case\ list\ of}\ [\text{]} \Rightarrow f_1 | x \#\ xa \Rightarrow f_2 \ x\ xa) = (\neg\ (\text{list} = \text{]} \wedge \neg\ P\ f_1 \vee \text{list} = \text{hd\ list\ #}\ \text{tl\ list} \wedge \neg\ P (f_2 (\text{hd\ list}) (\text{tl\ list})))
\]

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

\textit{t.case\_eq\_if}: \\
\[
(\text{case\ list\ of}\ [\text{]} \Rightarrow f_1 | x \#\ xa \Rightarrow f_2 \ x\ xa) = (\text{if\ null\ list\ then\ f_1\ else} \\
f_2 (\text{hd\ list}) (\text{tl\ list}))
\]

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

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., \texttt{\'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} \ {\text{[transfer\_rule]}]: \\
\[
\text{rel\_fun}\ S (\text{rel\_fun}\ (\text{rel\_fun}\ R (\text{rel\_fun}\ (\text{list\_all2}\ R)\ S)\ (\text{rel\_fun}\ (\text{list\_all2}\ R)\ S)))\ \text{case\_list}\ \text{case\_list}
\]

This property is generated by the \textit{transfer} plugin (Section 8.3).
### Defining Datatypes

#### t.sel_transfer [transfer_rule]:
This property is missing for 'a list because there is no common selector to all constructors.
The [transfer_rule] attribute is set by the transfer plugin (Section 8.3).

#### t.ctr_transfer [transfer_rule]:
\[\text{list\_all2} R \, \rightsquigarrow \, \text{transfer}\_rule\]
\[\text{rel\_fun} R (\text{rel\_fun} (\text{list\_all2} R) (\text{list\_all2} R)) (\#) (\#)\]
The [transfer_rule] attribute is set by the transfer plugin (Section 8.3).

#### t.disc_transfer [transfer_rule]:
\[\text{rel\_fun} (\text{list\_all2} R) (=) \null \null\]
\[\text{rel\_fun} (\text{list\_all2} R) (=) (\lambda \text{list}. \neg \text{null list}) (\lambda \text{list}. \neg \text{null list})\]
The [transfer_rule] attribute is set by the transfer plugin (Section 8.3).

#### t.set [simp, code]:
\[\text{set} [] = \{\}\]
\[\text{set} (x_21 \# x_22) = \text{insert} x_21 (\text{set} x_22)\]
The [code] attribute is set by the code plugin (Section 8.1).

#### t.set_cases [consumes 1, cases set: set\_t]:
\[e \in \text{set} a; \forall z_2. a = e \# z_2 \implies \text{thesis}; \forall z_1 \, z_2. [a = z_1 \# z_2; e \in \text{set} z_2] \implies \text{thesis}] \implies \text{thesis}\]

#### t.set_intros:
\[x_21 \in \text{set} (x_21 \# x_22)\]
\[y \in \text{set} x_22 \implies y \in \text{set} (x_21 \# x_22)\]

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

#### t.map [simp, code]:
\[\text{map} f [] = []\]
\[\text{map} f (x_21 \# x_22) = f \, x_21 \# \text{map} \, f \, x_22\]
The [code] attribute is set by the code plugin (Section 8.1).

#### t.map_disc_iff [simp]:
\[\text{null} (\text{map} \, f \, a) = \text{null} \, a\]

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

#### t.pred_inject [simp]:
\[\text{list\_all} P []\]
\[\text{list\_all} P \, (a \# a) = (P \, a \land \text{list\_all} P \, a)\]
t.rel_inject [simp]:
\[ \text{list_all}2 R \[
\text{list_all}2 R (x21 \not\equiv x22) (y21 \not\equiv y22) = (R x21 y21 \land \text{list_all}2 R x22 y22) \]

t.rel_distinct [simp]:
\[ \neg \text{list_all}2 R \[
\neg \text{list_all}2 R (y21 \not\equiv y22) \]

t.rel_intros:
\[ \text{list_all}2 R \[
[R x21 y21; \text{list_all}2 R x22 y22] \implies \text{list_all}2 R (x21 \not\equiv x22) (y21 \not\equiv y22) \]

t.rel_cases [consumes 1, case_names t_1 ... t_m, cases pred]:
\[ [\text{list_all}2 R a b; [a = \[]; b = \[]] \implies \text{thesis}; \land x1 x2 y1 y2. [a = x1 \not\equiv x2; b = y1 \not\equiv y2; R x1 y1; \text{list_all}2 R x2 y2] \implies \text{thesis}] \implies \text{thesis} \]

t.rel_sel:
\[ \text{list_all}2 R a b = (\text{null} a = \text{null} b \land (\neg \text{null} a \implies \neg \text{null} b \implies R (\text{hd} a) (\text{hd} b) \land \text{list_all}2 R (\text{tl} a) (\text{tl} b))) \]

In addition, equational versions of \text{t.rel_inject} and \text{rel_distinct} are registered with the [code] attribute. The [code] attribute is set by the 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.inj_map:
\[ \text{inj} f \implies \text{inj} (\text{map} f) \]

t.inj_map_strong:
\[ [\forall z. z \in \text{set} x; z a \in \text{set} x a; f z = f a z a] \implies z = z a; \text{map} f x = \text{map} f a x a] \implies x = x a \]

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

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

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

t.map_cong_pred:
\[ [x = ya; \text{list_all} (\lambda z. f z = g z) ya] \implies \text{map} f x = \text{map} g ya \]
t.map_cong_simp:
\[
[x = ya; \forall z. z \in \text{set } ya = \text{simp} \Rightarrow f z = g z] \implies \text{map } f x = \text{map } g ya
\]

t.map_id0:
map id = id

t.map_id:
map id t = t

t.map_ident:
map (\lambda x. x) t = t

t.map_transfer [\text{transfer_rule}]:
rel_fun (rel_fun Rb Sd) (rel_fun (list_all2 Rb) (list_all2 Sd)) map map

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

t.pred_cong [\text{fundef_cong}]:
\[
[x = ya; \forall z. z \in \text{set } ya \Rightarrow P z = Pa z] \implies \text{list_all } P x = \text{list_all } Pa ya
\]

t.pred_cong_simp:
\[
[x = ya; \forall z. z \in \text{set } ya = \text{simp} \Rightarrow P z = Pa z] \implies \text{list_all } P x = \text{list_all } Pa y
\]

t.pred_map:
\[
\text{list_all } Q (\text{map } f x) = \text{list_all } (Q \circ f) x
\]

t.pred_mono_strong:
\[
[\text{list_all } P x; \forall z. [z \in \text{set } x; P z] \Rightarrow Pa z] \implies \text{list_all } Pa x
\]

t.pred_rel:
\[
\text{list_all } P x = \text{list_all2 } (eq\_onp P) x x
\]

t.pred_set:
\[
\text{list_all } P = (\lambda x. \text{Ball (set } x) P)
\]

t.pred_transfer [\text{transfer_rule}]:
rel_fun (rel_fun R (\_)) (rel_fun (list_all2 R) (\_)) list_all list_all

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

t.pred_True:
\[
\text{list_all } (\lambda _\_. \text{True}) = (\lambda _\_. \text{True})
\]

t.set_map:
\[
\text{set } (\text{map } f v) = f \uparrow \text{set } v
\]
The [transfer_rule] attribute is set by the transfer plugin (Section 8.3) for type constructors with no dead type arguments.

The [relator_distr] attribute is set by the lifting plugin (Section 8.4).

The [rel_flip] operation takes a relation and reverses the arguments: $R \rightarrow a b = R \rightarrow b a$.

The [rel_map] operation maps a relation over a list: $\text{list_all}(Sb \ (\text{map} \ i \ x) \ y) = \text{list_all}(\lambda x. \ Sb \ (i \ x)) \ x \ y$.

The [rel_cong] operation respects equality and congruence: $[x = ya; \ y = xa; [z \ yb. \ z \in \text{set} \ ya; \ yb \in \text{set} \ xa] \Longrightarrow R z yb = Ra z yb]$.

The [rel_cong_simp] operation simplifies the equality and congruence: $R z yb = Ra z yb$.

The [rel_refl] operation respects reflexivity: $(\lambda x. \ Ra \ x \ x)$.

The [rel_refl_strong] operation respects strong reflexivity: $(\lambda z. \ z \in \text{set} \ x \Longrightarrow Ra z z)$.

The [rel_reflp] operation respects strong reflexivity: $\text{reflp} R \Longrightarrow \text{reflp} (\text{list_all} \ R)$. 

The [rel_eq] operation respects equality: $\text{list_all} (\lambda x. \ (=) \ x) = (=)$.

The [rel_eq_onp] operation respects equality over a relation: $\text{list_all}(\text{eq_onp} \ P) = \text{eq_onp}(\text{list_all} \ P)$.
2.4.3 Inductive Theorems

The inductive theorems are as follows:

\[ t.\text{induct} \ [ \text{case_names} \ C_1 \ldots \ C_n, \ \text{induct} \ t] : \]
\[ \begin{align*}
& [P \ []; \ \& x_1 x_2. \ P x_2 \implies P (x_1 \ # \ x_2)] \implies P \ \text{list} \\
\end{align*} \]

\[ t.\text{rel_induct} \ [ \text{case_names} \ C_1 \ldots \ C_n, \ \text{induct pred}] : \]
\[ \begin{align*}
& [\text{list_all2} \ R \ x \ y; \ Q \ []; \ \& a_1 a_2 b_1 b_2. \ [R a_1 b_1; \ Q a_2 b_2] \\
& \implies Q (a_1 \ # \ a_2) (b_1 \ # \ b_2)] \implies Q \ x \ y \\
\end{align*} \]

\[ t_1, \ldots, t_m.\text{rel_induct} \ [ \text{case_names} \ C_1 \ldots \ C_n]: \]

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

\[ t.\text{rec} \ [ \text{simp, code}] : \]
\[ \begin{align*}
& \text{rec_list} \ f_1 f_2 \ [] = f_1 \\
& \text{rec_list} \ f_1 f_2 (x_1 \ # \ x_2) = f_2 \ x_1 \ x_2 \ (\text{rec_list} \ f_1 f_2 x_2) \\
\end{align*} \]

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

\[ t.\text{rec_o_map} : \]
\[ \text{rec_list} \ g \ ga \circ \text{map} \ f = \text{rec_list} \ g (\lambda x \ xa. \ ga (f \ x) \ (\text{map} \ f \ xa)) \]

\[ t.\text{rec_transfer} \ [ \text{transfer_rule}] : \]
\[ \begin{align*}
& \text{rel_fun} \ S (\text{rel_fun} (\text{rel_fun} R (\text{rel_fun} \ (\text{list_all2} \ R) (\text{rel_fun} S S)))) \\
& (\text{rel_fun} (\text{list_all2} \ R) S)) \ \text{rec_list} \ \text{rec_list} \\
\end{align*} \]

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

For convenience, \text{datatype} also provides the following collection:

\[ t.\text{simps} = t.\text{inject} \ t.\text{distinct} \ t.\text{case} \ t.\text{rec} \ t.\text{map} \ t.\text{rel_inject} \ t.\text{rel_distinct} \ t.\text{set} \]
2.5 Proof Method

2.5.1 countable_datatype

The theory 
provides a proof method called countable_datatype that can be used to prove the countability of many datatypes, building on the countability of the types appearing in their definitions and of any type arguments. For example:

instance list :: (countable) countable
  by countable_datatype

2.6 Compatibility Issues

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

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

- The recursor rec_t has a different signature for nested recursive datatypes. In the old package, nested recursion through non-functions was internally reduced to mutual recursion. This reduction was visible in the type of the recursor, used by primrec. Recursion through functions was handled specially. In the new package, nested recursion (for functions and non-functions) is handled in a more modular fashion. The old-style recursor can be generated on demand using primrec if the recursion is via new-style datatypes, as explained in Section 3.1.5, or using datatype_compat.

- Accordingly, the induction rule is different for nested recursive datatypes. Again, the old-style induction rule can be generated on demand using primrec if the recursion is via new-style datatypes, as explained in Section 3.1.5, or using datatype_compat. For recursion through functions, the old-style induction rule can be obtained by applying the [unfolded all_mem_range] attribute on t.induct.

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

- **The internal constructions are completely different.** Proof texts that unfold the definition of constants introduced by the old command will be difficult to port.

- **Some constants and theorems have different names.** For non-mutually recursive datatypes, the alias $t.inducts$ for $t.induct$ is no longer generated. For $m > 1$ mutually recursive datatypes, $rec\_t_1\ldots\_t_m\_i$ has been renamed $rec\_t_i$ for each $i \in \{1, \ldots, m\}$, $t_1\ldots\_t_m.inducts(i)$ has been renamed $t_i.induct$ for each $i \in \{1, \ldots, m\}$, and the collection $t_1\ldots\_t_m.size$ (generated by the size plugin, Section 8.2) has been divided into $t_1.size$, $\ldots$, $t_m.size$.

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

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

## 3 Defining Primitively Recursive Functions

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

### 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:

```
primrec (nonexhaustive) bool_of_trool :: "trool ⇒ bool" where
 "bool_of_trool False ←→ False"
```
3 Defining Primitively Recursive Functions

| “bool_of_trool Truue $\iff$ True”

primrec the_list :: "'a option $\Rightarrow$ 'a list" where
“the_list None = []”
| “the_list (Some a) = [a]”

primrec the_default :: "'a $\Rightarrow$ 'a option $\Rightarrow$ 'a" where
“the_default d None = d”
| “the_default _ (Some a) = a”

primrec mirror :: "('a, 'b, 'c) triple $\Rightarrow$ ('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:

primrec replicate :: "nat $\Rightarrow$ 'a $\Rightarrow$ 'a list" where
“replicate Zero _ = []”
| “replicate (Succ n) x = x # replicate n x”

primrec (nonexhaustive) at :: "'a list $\Rightarrow$ nat $\Rightarrow$ 'a" where
“at (x # xs) j =
(case j of
  Zero $\Rightarrow$ x
| Succ j $\Rightarrow$ at xs j)”

primrec tfold :: "('a $\Rightarrow$ 'b $\Rightarrow$ 'b) $\Rightarrow$ ('a, 'b) tlist $\Rightarrow$ '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-maching 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:

```ml
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:

```ml
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_f :: "('a ⇒ int) ⇒ ('b ⇒ int) ⇒ ('a, 'b) fct ⇒ int”
where
  "eval_e γ ξ (Term t) = eval_t γ ξ t”
| “eval_e γ ξ (Sum t e) = eval_t γ ξ t + eval_e γ ξ e”
| “eval_t γ ξ (Factor f) = eval_f γ ξ f”
| “eval_t γ ξ (Prod f t) = eval_f γ ξ f + eval_t γ ξ t”
| “eval_f γ _ (Const a) = γ a”
| “eval_f _ ξ (Var b) = ξ b”
| “eval_f γ ξ (Expr e) = eval_e γ ξ e”
```

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

```ml
fun
  even :: "nat ⇒ bool” and
  odd :: "nat ⇒ bool”
where
  "even Zero = True”
| "even (Succ n) = odd n”
| "odd Zero = False”
| "odd (Succ n) = even n”
```
3.1.4 Nested Recursion

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

\begin{verbatim}
primrec at_ff :: "'a treeff ⇒ nat list ⇒ 'a" where
  "at_ff (Node_ff a ts) js =
  (case js of
    [] ⇒ a
  | j # js' ⇒ at_ff (map (λt. at_ff t js') ts) j)"
\end{verbatim}

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

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

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

\begin{verbatim}
primrec relabel_ft :: "('a⇒'a)⇒'a ftree⇒'a ftree" where
  "relabel_ft f (FTLeaf x) = FTLeaf (f x)"
  | "relabel_ft f (FTNode g) = FTNode (relabel_ft f o g)"
\end{verbatim}

For convenience, recursion through functions can also be expressed using λ-abstractions and function application rather than through composition. For example:

\begin{verbatim}
primrec relabel_ft :: "('a⇒'a)⇒'a ftree⇒'a ftree" where
  "relabel_ft f (FTLeaf x) = FTLeaf (f x)"
  | "relabel_ft f (FTNode g) = FTNode (λx. relabel_ft f (g x))"
\end{verbatim}

\begin{verbatim}
primrec (nonexhaustive) subtree_ft :: "'a⇒'a ftree⇒'a ftree" where
  "subtree_ft x (FTNode g) = g x"
\end{verbatim}

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

\begin{verbatim}
datatype 'a ftree2 = FTLeaf2 'a | FTNode2 "'a⇒'a⇒'a ftree2"
primrec relabel_ft2 :: "('a⇒'a)⇒'a ftree2⇒'a ftree2" where
  "relabel_ft2 f (FTLeaf2 x) = FTLeaf2 (f x)"
  | "relabel_ft2 f (FTNode2 g) = FTNode2 ((o) ((o) (relabel_ft2 f)) g)"
\end{verbatim}

\begin{verbatim}
primrec relabel_ft2 :: "('a⇒'a)⇒'a ftree2⇒'a ftree2" where
\end{verbatim}
"relabel\_ft2 f (FTLeaf2 x) = FTLeaf2 (f x)"
| "relabel\_ft2 f (FTNode2 g) = FTVNode2 (\(x\ y\). relabel\_ft2 f (g x y))"

**primrec** (nonexhaustive) **subtree\_ft2** :: "'a ⇒ 'a ftree2 ⇒ 'a ftree2" where
"subtree\_ft2 x y (FTNode2 g) = g x y"

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

**primrec** increasing\_tree :: "int ⇒ int ftree ⇒ bool" where
"increasing\_tree m (Nodeff n ts) ↔
  n ≥ m ∧ 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:

**primrec** (nonexhaustive)
"atff :: "'a ftree ⇒ nat list ⇒ 'a" and
atsff :: "'a ftree list ⇒ nat ⇒ nat list ⇒ 'a"

where
"atff (Nodeff a ts) js =
  (case js of
    [] ⇒ a
  | j # js' ⇒ atsff ts j js')"
| "atsff (t # ts) j =
  (case j of
    Zero ⇒ atff t
  | Succ j' ⇒ atsff ts j')"

Appropriate induction rules are generated as `atff.induct`, `atsff.induct`, and `atff_atsff.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:

**primrec**
"sum\_btree :: "'(a::{zero,plus}) btree ⇒ 'a" and
sum\_btree\_option :: "a btree option ⇒ 'a"

where
"sum\_btree (BNode a lt rt) =
  a + sum\_btree\_option lt + sum\_btree\_option rt"
| "sum\_btree\_option None = 0"
| "sum\_btree\_option (Some t) = sum\_btree t"
3.2 Command Syntax

3.2.1 primrec

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

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

3.3 Generated Theorems
The \texttt{primrec} command generates the following properties (listed for \texttt{tfold}):

\begin{verbatim}
\texttt{f.simps [simp, code]:}
\hspace{1em}tfold \texttt{uu (TNil y) = y}
\hspace{1em}tfold \texttt{f (TCons x xs) = f x (tfold f xs)}
\hspace{1em}The \texttt{[code]} attribute is set by the \texttt{code} plugin (Section 8.1).

\texttt{f.transfer [transfer_rule]:}
\hspace{1em}rel\_fun (rel\_fun R2 (rel\_fun R1 R1)) (rel\_fun (rel\_tlist R2 R1) R1) tfold tfold
\hspace{1em}This theorem is generated by the \texttt{transfer} plugin (Section 8.3) for functions declared with the \texttt{transfer} option enabled.

\texttt{f.induct [case_names C1 \ldots C_n]:}
\hspace{1em}This induction rule is generated for nested-as-mutual recursive functions (Section 3.1.5).

\texttt{f1_\ldots_fm.induct [case_names C1 \ldots C_n]:}
\hspace{1em}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.
\end{verbatim}

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

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

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

The following example illustrates this procedure:

```isar
cconsts termi_0 :: 'a
datatype ('a, 'b) tlist =
  TNil (termi: 'b)
| TCons (thd: 'a) (ttl: "('a, 'b) tlist")
where
  "ttl (TNil y) = TNil y"
| "termi (TCons _ xs) = termi_0 xs"
overloading
  termi_0 ≡ "termi_0 :: ('a, 'b) tlist ⇒ 'b"
begin
primrec termi_0 :: "('a, 'b) tlist ⇒ 'b" where
  "termi_0 (TNil y) = y"
| "termi_0 (TCons x xs) = termi_0 xs"
end
lemma termi_TCons[simp]: "termi (TCons x xs) = termi xs"
  by (cases xs) auto
```

### 3.5 Compatibility Issues

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

- \textit{Some theorems have different names.} For $m > 1$ mutually recursive
  functions, $f_1 \ldots f_m$.\texttt{simps} has been broken down into separate sub-
collections $f_i$.\texttt{simps}. 
4 Defining Codatatypes

Codatatypes can be specified using the codatatype command. The command is first illustrated through concrete examples featuring different flavors of corecursion. More examples can be found in the directory ~/src/HOL/Datatype_Examples. The Archive of Formal Proofs also includes some useful codatatypes, notably for lazy lists [7].

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 ('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 (...)) and LCons 0 (LCons 1 (LCons 2 (...))). Here is a related type, that of infinite streams:

```plaintext
codatatype ('a) stream =
  SCons (shd: 'a) (stl: "'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 (...))). That represents ∞. In addition, it has finite values of the form ESucc (... (ESucc EZero)....). 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:

```
codatatype even_enat = Even_EZero | Even_ESucc odd_enat
and odd_enat = Odd_ESucc even_enat
```

4.1.3 Nested Corecursion

The next examples feature nested corecursion:

```
codatatype ’a tree_i = Node_i_i (lbl_i: ’a) (sub_i_i: “’a tree_i llist”)
codatatype ’a tree_s = Node_s (lbl_s: ’a) (sub_s: “’a tree_s fset”)
codatatype ’a sm = SM (accept: bool) (trans: “’a ⇒ ’a sm”)
```

4.2 Command Syntax

4.2.1 `codatatype`

```
codatatype : local_theory → local_theory
```

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

Given a codatatype \( (a_1, \ldots, a_m) \ t \) 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:

Corecisor:  \( t.corec_t \)

4.4 Generated Theorems

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

- The \textit{free constructor theorems} (Section 2.4.1) are properties of the constructors and destructors that can be derived for any freely generated type.
- The \textit{functorial theorems} (Section 2.4.2) are properties of datatypes related to their BNF nature.
- The \textit{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{verbatim}
t.coinduct [consumes m, case_names t1 ... tm, case_conclusion D1 ... Dn, coinduct t]:
\[ R llist llist'; \land llist llist' \Rightarrow lnull llist = lnull llist' \land (\neg lnull llist \rightarrow \neg lnull llist' \rightarrow lhd llist = lhd llist' \land R (lhd llist)) \land \neg lnull llist') \Rightarrow llist = llist'
\]
t.coinduct_strong [consumes m, case_names t1 ... tm, case_conclusion D1 ... Dn]:
\[ R llist llist'; \land llist llist' \Rightarrow lnull llist = lnull llist' \land (\neg lnull llist \rightarrow \neg lnull llist' \rightarrow lhd llist = lhd llist' \land (lhd llist') \land (ltl llist = tlt llist')) \Rightarrow llist = llist'
\]
t.rel_coinduct [consumes m, case_names t1 ... tm, case_conclusion D1 ... Dn, coinduct pred]:
\[ P x y; llist llist'; P llist llist' \Rightarrow lnull llist = lnull llist' \land (\neg lnull llist \rightarrow \neg lnull llist' \rightarrow R (lhd llist) (lhd llist') \land P (lhd llist) (lhd llist')) \Rightarrow llist_all2 R x y
\]
\end{verbatim}
4 Defining Codatatypes

$t_1 \ldots t_m.\text{coinduct} \ [\text{case_names } t_1 \ldots t_m, \text{ case_conclusion } D_1 \ldots D_n]$

$t_1 \ldots t_m.\text{coinduct\_strong} \ [\text{case_names } t_1 \ldots t_m,\
\text{ case_conclusion } D_1 \ldots D_n]:$

$t_1 \ldots t_m.\text{rel\_coinduct} \ [\text{case_names } t_1 \ldots t_m,\
\text{ case_conclusion } D_1 \ldots D_n]:$

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

$t_1 \ldots t_m.\text{set\_induct} \ [\text{case_names } C_1 \ldots C_n,\
\text{ induct set: set}_{j \cdot t_1}, \ldots, \text{ induct set: set}_{j \cdot t_m}]:$

$[x \in \text{ lset } a; \land z_1 z_2. P z_1 (LCons z_1 z_2); \land z_1 z_2 xa. [xa \in \text{ lset } z_2; P xa z_2] \implies P xa (LCons z_1 z_2)] \implies P x a$

If $m = 1$, the attribute [consumes 1] is generated as well.

t.corec:
$p a \implies \text{corec\_llist } p g_21 q_22 g_222 a = \text{LNil}$
$\neg p a \implies \text{corec\_llist } p g_21 q_22 g_221 g_222 a = \text{LCons } (g_21 a) (\text{if } q_22 a \text{ then } g_221 a \text{ else } \text{corec\_llist } p g_21 q_22 g_221 g_222 (g_222 a))$

t.corec\_code [code]:
$\text{corec\_llist } p g_21 q_22 g_221 g_222 a = (\text{if } p a \text{ then } \text{LNil} \text{ else } \text{LCons } (g_21 a) (\text{if } q_22 a \text{ then } g_221 a \text{ else } \text{corec\_llist } p g_21 q_22 g_221 g_222 (g_222 a)))$

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

t.corec\_disc:
$p a \implies \text{lnull } (\text{corec\_llist } p g_21 q_22 g_221 g_222 a)$
$\neg p a \implies \neg \text{lnull } (\text{corec\_llist } p g_21 q_22 g_221 g_222 a)$

t.corec\_disc\_iff [simp]:
$\text{lnull } (\text{corec\_llist } p g_21 q_22 g_221 g_222 a) = p a$
$(\neg \text{lnull } (\text{corec\_llist } p g_21 q_22 g_221 g_222 a)) = (\neg p a)$

t.corec\_sel [simp]:
$\neg p a \implies \text{lhd } (\text{corec\_llist } p g_21 q_22 g_221 g_222 a) = g_21 a$
$\neg p a \implies \text{ltl } (\text{corec\_llist } p g_21 q_22 g_221 g_222 a) = (\text{if } q_22 a \text{ then } g_221 a \text{ else } \text{corec\_llist } p g_21 q_22 g_221 g_222 (g_222 a))$

t.map\_o\_corec:
$\text{lmap } f \circ \text{corec\_llist } g \text{ ga gb gc gd } = \text{corec\_llist } g (f \circ ga) \text{ gb } (\text{lmap } f \circ gc) \text{ gd}$

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

For convenience, \texttt{codatatype} also provides the following collection:

\begin{align*}
  \textit{t.simps} &= \textit{t.inject} \ \textit{t.distinct} \ \textit{t.case} \ \textit{t.corec\_disc\_iff} \ \textit{t.corec\_sel} \\
  &\quad \textit{t.map} \ \textit{t.rel\_inject} \ \textit{t.rel\_distinct} \ \textit{t.set}
\end{align*}

5 Defining Primitively Corecursive Functions

Corecursive functions can be specified using the \texttt{primcorec} and \texttt{primcorecursive} commands, which support primitive corecursion. Other approaches include the more general \texttt{partial\_function} command, the \texttt{corec} and \texttt{corecursive} commands, and techniques based on domains and topologies \cite{8}. In this tutorial, the focus is on \texttt{primcorec} and \texttt{primcorecursive}; \texttt{corec} and \texttt{corecursive} are described in a separate tutorial \cite{3}. More examples can be found in the directories \texttt{~/src/HOL/Datatype\_Examples} and \texttt{~/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 \textit{destructor view} specifies $f$ by implications of the form
  \[
  \ldots \implies \textit{is\_C}_j (f \ x_1 \ldots \ x_n)
  \]
  and equations of the form
  \[
  \textit{un\_C}_j i (f \ x_1 \ldots \ x_n) = \ldots
  \]
  This style is popular in the coalgebraic literature.

- The \textit{constructor view} specifies $f$ by equations of the form
  \[
  \ldots \implies f \ x_1 \ldots \ x_n = C_j \ldots
  \]
  This style is often more concise than the previous one.

- The \textit{code view} specifies $f$ by a single equation of the form
  \[
  f \ x_1 \ldots \ x_n = \ldots
  \]
  with restrictions on the format of the right-hand side. Lazy functional programming languages such as Haskell support a generalized version of this style.
All three styles are available as input syntax. Whichever syntax is chosen, characteristic theorems for all three styles are generated.

5.1 Introductory Examples

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

5.1.1 Simple Corecursion

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

```plaintext
primcorec literate :: "'(a ⇒ 'a) ⇒ 'a ⇒ 'a llist" where
  "literate g x = LCons x (literate g (g x))"
```

```plaintext
primcorec siterate :: "'(a ⇒ 'a) ⇒ 'a ⇒ 'a stream" where
  "siterate g x = SCons x (siterate g (g x))"
```

The constructor ensures that progress is made—i.e., the function is productive. The above functions compute the infinite lazy list or stream `[x, g x, g (g x), ...]`. Productivity guarantees that prefixes `[x, g x, g (g x), ..., (g ^^ 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:

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

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

```plaintext
primcorec lapp :: "'a llist ⇒ 'a llist ⇒ 'a llist" where
  "lapp xs ys =
    (case xs of
     LNil ⇒ ys
    | 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 \(-/-\text{src/HOL/Library/Simps_Case_Conv.thy}\).

**simps_of_case** lapp_simps: lapp.code

This generates the lemma collection **lapp_simps**:

\begin{align*}
\text{lapp} \text{ LNil } \text{ys} &= \text{ys} \\
\text{lapp} \left( \text{LCons } \text{xa } x \right) \text{ys} &= \text{LCons} \text{ xa } \left( \text{lapp } x \text{ ys} \right)
\end{align*}

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

**primcorec** infty :: enat where

“infty = ESucc infty”

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

**primcorec**

\begin{verbatim}
random_process :: "a stream ⇒ (int ⇒ int) ⇒ int ⇒ 'a process"
where
  "random_process s f n =
  (if n mod 4 = 0 then
   Fail
  else if n mod 4 = 1 then
   Skip (random_process s f (f n))
  else if n mod 4 = 2 then
   Action (shd s) (random_process (stl s) f (f n))
  else
   Choice (random_process (every_snd s) (f o f) (f n))
          (random_process (every_snd (stl s)) (f o f) (f (f n))))"
\end{verbatim}

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 data-types is unsurprising:

**primcorec**

\begin{verbatim}
even_infty :: even_enat and
odd_infty :: odd_enat
where
\end{verbatim}
5.1.3 Nested Corecursion

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

\[
\text{primcorec iterate}_{ii} :: \left(\nu a . a \Rightarrow a \text{ llist} \right) \Rightarrow \nu a . a \Rightarrow \nu a . a \text{ tree}_{ii}\]

\[
\text{iterate}_{ii} g x = \text{Node}_{ii} x \left(\text{lmap} \text{iterate}_{ii} g \left(\text{g x}\right)\right)
\]

\[
\text{primcorec iterate}_{is} :: \left(\nu a . a \Rightarrow a \text{ fset} \right) \Rightarrow \nu a . a \Rightarrow \nu a . a \text{ tree}_{is}\]

\[
\text{iterate}_{is} g x = \text{Node}_{is} x \left(\text{fimage} \text{iterate}_{is} g \left(\text{g x}\right)\right)
\]

Both examples follow the usual format for constructor arguments associated with nested recursive occurrences of the datatype. Consider \(\text{iterate}_{ii}\). The term \(g x\) constructs an \(\nu a . a \text{ llist}\) value, which is turned into an \(\nu a . a \text{ tree}_{ii} \text{ llist}\) value using \(\text{lmap}\).

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

\[
\text{primcorec tree}_{ii} \text{ of stream} :: \left(\nu a . a \Rightarrow a \text{ stream} \Rightarrow a \Rightarrow a \text{ tree}_{ii}\right) \Rightarrow \nu a . a \Rightarrow \nu a . a \text{ tree}_{ii} \text{ of stream} \Rightarrow \nu a . a \text{ stream} \Rightarrow \nu a . a \Rightarrow a \text{ sm}
\]

\[
\text{tree}_{ii} \text{ of stream} s = \text{Node}_{ii} \left(\text{shd} s\right) \left(\text{lmap} \text{tree}_{ii} \text{ of stream} \left(\text{LCons} \left(\text{stl} s\right) \text{LNil}\right)\right)
\]

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

\[
\text{primcorec tree}_{ii} \text{ of stream} :: \left(\nu a . a \Rightarrow a \text{ stream} \Rightarrow a \Rightarrow a \text{ tree}_{ii}\right) \Rightarrow \nu a . a \Rightarrow \nu a . a \text{ tree}_{ii} \text{ of stream} \Rightarrow \nu a . a \text{ stream} \Rightarrow \nu a . a \Rightarrow a \text{ sm}
\]

\[
\text{tree}_{ii} \text{ of stream} s = \text{Node}_{ii} \left(\text{shd} s\right) \left(\text{LCons} \left(\text{tree}_{ii} \text{ of stream} \left(\text{stl} s\right)\right) \text{LNil}\right)
\]

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

\[
\text{primcorec sm of dfa} :: \left(\nu q . q \Rightarrow q \Rightarrow q \Rightarrow q \Rightarrow a \text{ sm}\right) \Rightarrow \nu q . q \Rightarrow \nu q . q \Rightarrow a \text{ sm}
\]

\[
\text{sm of dfa} \delta F q = \text{SM} \left(\left(q \in F\right) \left(\text{sm of dfa} \delta F \circ \delta q\right)\right)
\]

The map function for the function type \((\Rightarrow)\) is composition \((\circ)\). For convenience, corecursion through functions can also be expressed using \(\lambda\)-abstractions and function application rather than through composition. For example:
```
primcorec `sm_of_dfa` :: "`(q ⇒ 'a ⇒ 'q) ⇒ 'q set ⇒ 'q ⇒ 'a sm" where
  "sm_of_dfa δ F q = SM (q ∈ F) (λa. sm_of_dfa δ F (δ q a))"

primcorec `empty_sm` :: "'a sm" where
  "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 \(\circ\) are necessary. The examples below illustrate the case where \(n = 2\):

```
codatatype ('a, 'b) sm2 =
  SM2 (accept2 : bool) (trans2: "'a ⇒ 'a llist")

primcorec
  sm2_of_dfa :: "'(q ⇒ 'a ⇒ 'b ⇒ 'q) ⇒ 'q set ⇒ 'q ⇒ ('a, 'b) sm2" where
  "sm2_of_dfa δ F q = SM2 (q ∈ F) ((o) ((o) (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
  iteratei : "'(a ⇒ 'a llist) ⇒ 'a ⇒ 'a treeii" and
  iteratesii : "'(a ⇒ 'a llist) ⇒ 'a llist ⇒ 'a treeii llist" where
  "iteratei g x = Nodeii x (iteratesii g (g x))"
  | "iteratesii g xs =
      (case xs of
       LNil ⇒ LNil
       | LCons x xs' ⇒ LCons (iteratei 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 corecurors are generated dynamically and are kept in a cache to speed up subsequent definitions.
```
5.1.5 Constructor View

The constructor view is similar to the code view, but there is one separate conditional equation per constructor rather than a single unconditional equation. Examples that rely on a single constructor, such as `literate` and `siterate`, are identical in both styles.

Here is an example where there is a difference:

```plaintext
primcorec lapp :: "'a llist ⇒ 'a llist ⇒ 'a llist" where
  "lnull xs ⇒ lnull ys ⇒ lapp xs ys = LNil"
| "_ ⇒ lapp xs ys = LCons (lhd (if lnull xs then ys else xs))
  (if xs = LNil then ltl ys else lapp (ltl xs) ys)"
```

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

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

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

```plaintext
primcorec
random_process :: "a stream ⇒ (int ⇒ int) ⇒ int ⇒ 'a process"
where
  "n mod 4 = 0 ⇒ random_process s f n = Fail"
| "n mod 4 = 1 ⇒
  random_process s f n = Skip (random_process s f (f n))"
| "n mod 4 = 2 ⇒
  random_process s f n = Action (shd s) (random_process (stl s) f (f n))"
| "n mod 4 = 3 ⇒
  random_process s f n = Choice (random_process (every_snd s) f (f n))
  (random_process (every_snd (stl s)) f (f n))"
```

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

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

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

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

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

The first formula in the \texttt{local.literate} specification indicates which constructor to choose. For \texttt{local.siterate} and \texttt{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 :: "'a llist ⇒ 'a llist ⇒ 'a llist" where
  "lnull xs ⇒ lnull ys ⇒ lnull (lapp xs ys)"
  | "lhhd (lapp xs ys) = lhhd (if lnull xs then ys else xs)"
  | "ltl (lapp xs ys) = (if xs = LNil then ltl ys else lapp (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}
"_ ⇒ ¬ 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 :: "'a stream ⇒ (int ⇒ int) ⇒ int ⇒ 'a process" where
  "n mod 4 = 0 ⇒ random_process s f n = Fail"
\end{verbatim}
Defining Primitively Corecursive Functions

| “n mod 4 = 1 \implies \text{is\_Skip} (\text{random\_process} s f n)” |
| “n mod 4 = 2 \implies \text{is\_Action} (\text{random\_process} s f n)” |
| “n mod 4 = 3 \implies \text{is\_Choice} (\text{random\_process} s f n)” |
| “\text{cont} (\text{random\_process} s f n) = \text{random\_process} s f (f n)” of Skip |
| “\text{prefix} (\text{random\_process} s f n) = \text{shd} s” |
| “\text{cont} (\text{random\_process} s f n) = \text{random\_process} (\text{stl} s) f (f n)” of Action |
| “\text{left} (\text{random\_process} s f n) = \text{random\_process} (\text{every\_snd} s) f (f n)” |
| “\text{right} (\text{random\_process} s f n) = \text{random\_process} (\text{every\_snd} (\text{stl} s)) f (f n)” |

Using the \textit{of} keyword, different equations are specified for \texttt{cont} depending on which constructor is selected.

Here are more examples to conclude:

\begin{verbatim}
primcorec
  even_infty :: even_enat and
  odd_infty :: odd_enat
where
  “even_infty \neq \text{Even\_EZero}”
| “\text{un\_Even\_ESucc} even_infty = odd_infty”
| “\text{un\_Odd\_ESucc} odd_infty = even_infty”
primcorec iterate_{ii} :: (“\text{a} \Rightarrow \text{a list}” \Rightarrow \text{a} \Rightarrow \text{a tree_{ii}” where
  “\text{lbl}_{ii} (\text{iterate}_{ii} g x) = x”
| “\text{sub}_{ii} (\text{iterate}_{ii} g x) = \text{lmap} (\text{iterate}_{ii} g) (g x)”
\end{verbatim}

5.2 Command Syntax

5.2.1 primcorec and primcorecursive

\texttt{primcorec} : \texttt{local\_theory} \rightarrow \texttt{local\_theory}
\texttt{primcorecursive} : \texttt{local\_theory} \rightarrow \texttt{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{primcore} command is an abbreviation for \texttt{primcorecursive} with \textit{by auto}? to discharge any emerging proof obligations.

5.3 Generated Theorems

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

\begin{itemize}
    \item \texttt{f.code [code]}:
        \begin{verbatim}
literate g x = LCons x (literate g (g x))
        \end{verbatim}
        The \texttt{[code]} attribute is set by the \texttt{code} plugin (Section 8.1).
    \item \texttt{f.ctr}:
        \begin{verbatim}
literate g x = LCons x (literate g (g x))
        \end{verbatim}
    \item \texttt{f.disc [simp, code]}:
        \begin{verbatim}
\neg lnull (literate g x)
        \end{verbatim}
        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.
    \item \texttt{f.disc_iff [simp]}:
        \begin{verbatim}
\neg lnull (literate g x)
        \end{verbatim}
        This property is generated only for functions declared with the \texttt{exhaustive} option or whose conditions are trivially exhaustive.
    \item \texttt{f.sel [simp, code]}:
        \begin{verbatim}
\neg lnull (literate g x)
        \end{verbatim}
        The \texttt{[code]} attribute is set by the \texttt{code} plugin (Section 8.1).
    \item \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.
    \item \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.
    \item \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).
\end{itemize}
6 Registering Bounded Natural Functors

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

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

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

For convenience, primcorec and primcorecursive also provide the follow-
ing collection:

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

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 require-
ments 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)re-
cursion 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 (func-
torial action), $n$ set functions (natural transformations), and an infinite 
cardinal bound that satisfy certain properties. For example, $'a \text{llist}$ is a unary 
BNF. Its predicator $\text{llist\_all} :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{llist} \Rightarrow \text{bool}$ extends unary 
predicates over elements to unary predicates over lazy lists. Similarly, its re-
lator $\text{llist\_all}^2 :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{llist} \Rightarrow 'b \text{llist} \Rightarrow \text{bool}$ extends binary 
predicates over elements to binary predicates over parallel lazy lists. The
cardinal bound limits the number of elements returned by the set function; it may not depend on the cardinality of \( a \).

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

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

### 6.2 Introductory Examples

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

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

```isar
typedef \( (d, a) \) fn = \texttt{UNIV :: (d \Rightarrow a) set} 
  by simp

setup_lifting type_definition_fn

lift_definition map_fn :: \((a \Rightarrow b) \Rightarrow (d, a) \Rightarrow (d, b) \) fn\ is \((\circ)\).

lift_definition set_fn :: \((d, a) \Rightarrow a \) set\ is range.

lift_definition pred_fn :: \((a \Rightarrow bool) \Rightarrow (d, a) \Rightarrow bool\) is \texttt{pred_fun (\_ \_ True)}.

lift_definition rel_fn :: \((a \Rightarrow b \Rightarrow bool) \Rightarrow (d, a) \Rightarrow (d, b) \Rightarrow bool\) is \texttt{rel_fun (\=}.

bnf "\( (d, a) \) fn"
  map: map_fn
  sets: set_fn
  bd: "\texttt{natLeq + c \mid UNIV :: d set}\"
  rel: rel_fn
  pred: pred_fn

proof
  show "map_fn id = id"
```
by transfer auto
next
fix f :: "'a ⇒ 'b" and g :: "'b ⇒ 'c"
show "map_fn (g ∘ f) = map_fn g ∘ map_fn f"
  by transfer (auto simp add: comp_def)
next
fix F :: "'(d, 'a) fn" and f g :: "'a ⇒ 'b"
assume "∀x. x ∈ set_fn F ⟹ f x = g x"
then show "map_fn f F = map_fn g F"
  by transfer auto
next
fix f :: "'a ⇒ 'b"
show "set_fn ∘ map_fn f = (') f ∘ set_fn"
  by transfer (auto simp add: comp_def)
next
show "card_order (natLeq + c |UNIV :: 'd set| )"
  apply (rule card_order_csum)
  apply (rule natLeq_card_order)
  by (rule card_of_card_order_on)
next
show "cinfinite (natLeq + c |UNIV :: 'd set| )"
  apply (rule cinfinite_csum)
  apply (rule disjI1)
  by (rule natLeq_cinfinite)
next
fix F :: "'(d, 'a) fn"
have "|set_fn F| ≤o |UNIV :: 'd set| " (is "<_o ?U")
  by transfer (rule card_of_image)
also have "?U ≤o natLeq + c ?U"
  by (rule ordLeq_csum2) (rule card_of_Card_order)
finally show "|set_fn F| ≤o natLeq + c |UNIV :: 'd set| ".
next
fix R :: "'a ⇒ 'b ⇒ bool" and S :: "'b ⇒ 'c ⇒ bool"
show "rel_fn R OO rel_fn S ≤ rel_fn (R OO S)"
  by (rule, transfer) (auto simp add: rel_fun_def)
next
fix R :: "'a ⇒ 'b ⇒ bool"
show "rel_fn R = (λx y. ∃z. set_fn z ⊆ {(x, y). R x y} ∧ map_fn fst z = x ∧ map_fn snd z = y)"
  unfolding fun_eq_iff relcompp.simps conversep.simps
  by transfer (force simp: rel_fun_def subset_iff)
next
fix P :: "'a ⇒ 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, lifting the BNF structure from the raw type 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 auto

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 "\( xs \in \{xs. \; xs \neq \; []\} \)"
  then show "\( \; map \; f \; xs \in \{xs. \; xs \neq \; []\} \)"
    by (cases \( xs \)) auto

next
  fix \( zs :: ("'a \times 'b") \; list" \)
  assume "\( map \; fst \; zs \in \{xs. \; xs \neq \; []\} \)" "\( map \; snd \; zs \in \{xs. \; xs \neq \; []\} \)"
  then show "\( zs \in \{xs. \; xs \neq \; []\} \)"
    by (cases \( zs \)) auto

qed

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

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

\texttt{print_theorems}
\texttt{print_bnfs}

6.3 Command Syntax

6.3.1 \texttt{bnf}

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

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

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

### 6.3.2 `lift_bnf`

\[
\text{lift\_bnf} : \text{local\_theory} \rightarrow \text{proof}(\text{prove})
\]
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. To achieve this, it lifts the BNF structure on the raw type to the abstract type following a *type_definition* theorem. The theorem is usually inferred from the type, but can also be explicitly supplied by means of the optional *via* clause. 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}

\[
\text{copy\_bnf} : \text{local\_theory} \to \text{local\_theory}
\]

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

### 6.3.4 \texttt{bnf\_axiomatization}

\[
\text{bnf\_axiomatization} : \text{local\_theory} \to \text{local\_theory}
\]
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} →

7 Deriving Destructors and Theorems for Free Constructors

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

7.1 Command Syntax

7.1.1 free_constructors

\texttt{free\_constructors} : \texttt{local\_theory} → \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 simps_of_case command provided by theory -/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,

```plaintext
    simps_of_case lapp_simps: lapp.code
```

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

\[
\begin{align*}
lapp \; \text{LNil} \; ys &= ys \\
lapp \; (\text{LCons} \; xa \; x) \; ys &= \text{LCons} \; xa \; (lapp \; x \; ys)
\end{align*}
\]

### 7.1.3 case_of_simps

The case_of_simps command provided by theory -/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. simps_of_case). For example,

```plaintext
    case_of_simps lapp_case: lapp_simps
```

translates

\[
\begin{align*}
lapp \; \text{LNil} \; ys &= ys \\
lapp \; (\text{LCons} \; xa \; x) \; ys &= \text{LCons} \; xa \; (lapp \; x \; ys)
\end{align*}
\]

into \( lapp \; xb \; xc = (\text{case} \; (xb, \; xc) \; \text{of} \; (\text{LNil}, \; ys) \Rightarrow ys \mid (\text{LCons} \; xa \; x, \; ys) \Rightarrow \text{LCons} \; xa \; (lapp \; x \; ys)) \).
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 \texttt{datatype, primrec, codatatype, primcorec, primcorecursive, bnf, bnf_axiomatization, and free_constructors}. For example:

\begin{verbatim}
datatype (plugins del: code "quickcheck") color = Red | Black
\end{verbatim}

Beyond the standard plugins, the \textit{Archive of Formal Proofs} includes a \texttt{derive} command that derives class instances of datatypes \cite{archive}.

8.1 Code Generator

The \texttt{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:

\begin{verbatim}
t.eq.refl [code nbe]:
equal_class.equal x x \equiv \text{True}
t.eq.simps [code]:
equal_class.equal [] (x21 \# x22) \equiv \text{False}
equal_class.equal (x21 \# x22) [] \equiv \text{False}
equal_class.equal (x21 \# x22) [] \equiv \text{False}
equal_class.equal [] (x21 \# x22) \equiv \text{False}
equal_class.equal (x21 \# x22) (y21 \# y22) \equiv x21 = y21 \land x22 = y22
equal_class.equal [] [] \equiv \text{True}
\end{verbatim}

In addition, the plugin sets the \texttt{[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 \texttt{size} plugin generates a generic size function \( t.size_t \) as well as a specific instance \( size :: t \Rightarrow \text{nat} \) belonging to the \texttt{size type class}. 
The **fun** command relies on **size** to prove termination of recursive functions on datatypes.

The plugin derives the following properties:

**t.size** [simp, code]:
- \( size\_list\ x\ [] = 0 \)
- \( size\_list\ x\ (x\_21\ #\ x\_22) = x\_21 + size\_list\ x\ x\_22 + Suc\ 0 \)
- \( size\ [] = 0 \)
- \( size\ (x\_21\ #\ x\_22) = size\ x\_22 + Suc\ 0 \)

**t.size\_gen**:
- \( size\_list\ x\ [] = 0 \)
- \( size\_list\ x\ (x\_21\ #\ x\_22) = x\_21 + size\_list\ x\ x\_22 + Suc\ 0 \)

**t.size\_gen\_o\_map**:
- \( size\_list\ f\circ\ map\ g = size\_list\ (f\circ\ g) \)

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

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 \( 'a_1, \ldots, 'a_m \), by default \( u \) values are given a size of 0. This can be improved upon by registering a custom size function of type \( ('a_1\Rightarrow\text{nat})\Rightarrow\ldots\Rightarrow('a_m\Rightarrow\text{nat})\Rightarrow u\Rightarrow\text{nat} \) 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:

**t.Domainp\_rel** [relator\_domain]:
- \( \text{Domainp} (\text{list\_all}\_2\ R) = \text{list\_all} (\text{Domainp} R) \)

**t.left\_total\_rel** [transfer\_rule]:
- \( \text{left\_total} R \leftrightarrow \text{left\_total} (\text{list\_all}\_2\ R) \)
8 Selecting Plugins

- \texttt{t.left\_unique\_rel} [\texttt{transfer\_rule}]:
  \[ \text{left\_unique } R \Rightarrow \text{left\_unique } (\text{list\_all}2 \ R) \]

- \texttt{t.right\_total\_rel} [\texttt{transfer\_rule}]:
  \[ \text{right\_total } R \Rightarrow \text{right\_total } (\text{list\_all}2 \ R) \]

- \texttt{t.right\_unique\_rel} [\texttt{transfer\_rule}]:
  \[ \text{right\_unique } R \Rightarrow \text{right\_unique } (\text{list\_all}2 \ R) \]

- \texttt{t.bi\_total\_rel} [\texttt{transfer\_rule}]:
  \[ \text{bi\_total } R \Rightarrow \text{bi\_total } (\text{list\_all}2 \ R) \]

- \texttt{t.bi\_unique\_rel} [\texttt{transfer\_rule}]:
  \[ \text{bi\_unique } R \Rightarrow \text{bi\_unique } (\text{list\_all}2 \ R) \]

For (co)datatypes with at least one live type argument, the plugin sets the [\texttt{transfer\_rule}] attribute on the following (co)datatypes properties: \texttt{t.case\_transfer}, \texttt{t.sel\_transfer}, \texttt{t.ctr\_transfer}, \texttt{t/disc\_transfer}, \texttt{t.rec\_transfer}, and \texttt{t.corec\_transfer}. For (co)datatypes that further have no dead type arguments, the plugin sets [\texttt{transfer\_rule}] on \texttt{t.set\_transfer}, \texttt{t.map\_transfer}, and \texttt{t.rel\_transfer}.

For \texttt{primrec}, \texttt{primcorec}, and \texttt{primcorecursive}, the plugin implements the generation of the \texttt{f.transfer} property, conditioned by the \texttt{transfer} option, and sets the [\texttt{transfer\_rule}] attribute on these.

### 8.4 Lifting

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

The plugin derives the following property:

- \texttt{t.Quotient} [\texttt{quot\_map}]:
  \[ \text{Quotient } R \text{ Abs } \text{ Rep } T \Rightarrow \text{Quotient } (\text{list\_all}2 \ R) (\text{map Abs} \text{ map Rep}) (\text{list\_all}2 \ T) \]

In addition, the plugin sets the [\texttt{relator\_eq}] attribute on a variant of the \texttt{t.rel\_eq\_onp} property, the [\texttt{relator\_mono}] attribute on \texttt{t.rel\_mono}, and the [\texttt{relator\_distr}] attribute on \texttt{t.rel\_compp}.

### 8.5 Quickcheck

The integration of datatypes with Quickcheck is accomplished by the \texttt{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


