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

12 December 2016

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: \texttt{datatype}, \texttt{datatype_compat}, \texttt{primrec}, \texttt{codatatype}, \texttt{primcorec}, \texttt{primcorecursive}, \texttt{bnf}, \texttt{lift_bnf}, \texttt{copy_bnf}, \texttt{bnf_axiomatization}, \texttt{print_bns}, and \texttt{free_constructors}.

Contents

1 Introduction 3

2 Defining Datatypes 5
  2.1 Introductory Examples ............................... 5
    2.1.1 Nonrecursive Types ........................... 5
    2.1.2 Simple Recursion ......................... 6
    2.1.3 Mutual Recursion ............................ 6
    2.1.4 Nested Recursion ......................... 7
    2.1.5 Auxiliary Constants ....................... 7
  2.2 Command Syntax ................................ 9
    2.2.1 \texttt{datatype} ............................ 9
    2.2.2 \texttt{datatype_compat} .................... 12
  2.3 Generated Constants ............................. 13
  2.4 Generated Theorems .............................. 13
    2.4.1 Free Constructor Theorems ................. 14
2.4.2 Functorial Theorems ........................................ 16
2.4.3 Inductive Theorems ....................................... 21
2.5 Proof Method ................................................. 21
  2.5.1 countable_datatype ........................................ 21
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 = Node_fs (lbl_fs: 'a) (sub_fs: "'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:

\[ \text{codatatype } 'a \text{llist} = L\text{Nil} | L\text{Cons } 'a\text{ }'a \text{llist} \]

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

\[ \text{datatype } 'a \text{tree}_f = \text{Node}_f 'a\text{ }'a \text{tree}_f \text{ list} \]
\[ \text{datatype } 'a \text{tree}_i = \text{Node}_i 'a\text{ }'a \text{tree}_i \text{ llist} \]
\[ \text{codatatype } 'a \text{tree}_f = \text{Node}_f 'a\text{ }'a \text{tree}_f \text{ list} \]
\[ \text{codatatype } 'a \text{tree}_i = \text{Node}_i 'a\text{ }'a \text{tree}_i \text{ llist} \]

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

The package is part of \textit{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 \cite{LCF2005}: The characteristic theorems associated with the specified (co)datatypes are derived rather than introduced axiomatically.\footnote{However, some of the internal constructions and most of the internal proof obligations are omitted if the quick\_and\_dirty option is enabled.} The package is described in a number of scientific papers \cite{BNF2011,Coinductive2014,Inductive2015,Lazy2016}. 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:

\begin{itemize}
  \item Section 2, “Defining Datatypes,” describes how to specify datatypes using the \textbf{datatype} command.
  \item Section 3, “Defining Primitively Recursive Functions,” describes how to specify functions using \textbf{primrec}. (A separate tutorial \cite{BNFFunctions2018} describes the more powerful \textbf{fun} and \textbf{function} commands.)
  \item Section 4, “Defining Codatatypes,” describes how to specify codatatypes using the \textbf{codatatype} command.
\end{itemize}
• Section 5, “Defining Primitively Corecursive Functions,” describes how to specify functions using the `primcorec` and `primcorecursive` commands. (A separate tutorial [3] describes the more powerful `corec` and `corecursive` commands.)

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

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

• Section 8, “Selecting Plugins,” is concerned with the package’s interoperability with other Isabelle packages and tools, such as the code generator, Transfer, Lifting, and Quickcheck.

• Section 9, “Known Bugs and Limitations,” concludes with known open issues.

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

2 Defining Datatypes

Datatypes can be specified using the `datatype` command.

2.1 Introductory Examples

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

2.1.1 Nonrecursive Types

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

```
datatype trool = Truue | Faalse | Perhaaps
```

`Truue`, `Faalse`, and `Perhaaps` have the type `trool`. 
Polymorphic types are possible, such as the following option type, modeled after its homologue from the Option theory:

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

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

The next example has three type parameters:

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

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

```
datatype ('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:

```plaintext
datatype 'a btree =
    BNode 'a "a btree option" "a btree option"
```

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

```plaintext
datatype 'a wrong = W1 | W2 "a wrong ⇒ 'a"
```

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

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

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

```plaintext
datatype 'a ftree = FTLeaf 'a |
    FTNode "'a ⇒ 'a ftree"
```

And so is the definition of hereditarily finite sets:

```plaintext
datatype hfset = HFSet "hfset fset"
```

In general, type constructors ('a_1, ..., 'a_m) t allow recursion on a subset of their type arguments 'a_1, ..., 'a_m. These type arguments are called live; the remaining type arguments are called dead. In 'a ⇒ 'b and ('a, 'b) fun_copy, the type variable 'a is dead and 'b is live.

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

Here is another example that fails:

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

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

2.1.5 Auxiliary Constants

The `datatype` command introduces various constants in addition to the constructors. With each datatype are associated set functions, a map function, a
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:

- `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

\textbf{datatype} (\texttt{'}\texttt{a}, \texttt{'}\texttt{b}) \textit{prod} (\textbf{infixr} “∗” 20) = \textit{Pair} \texttt{'}\texttt{a} \texttt{'}\texttt{b}

\textbf{datatype} \textit{set}: \texttt{'}\texttt{a} \textit{list} =
\begin{itemize}
\item \textit{null}: \textit{Nil} (“[]”)
\item \textit{Cons} (\texttt{hd}: \texttt{'}\texttt{a}) (\texttt{tl}: “\texttt{'}\texttt{a list}”) (\textbf{infixr} “#” 65)
\end{itemize}
\textbf{for}
\begin{itemize}
\item \textit{map}: \textit{map}
\item \textit{rel}: \textit{list\_all2}
\item \textit{pred}: \textit{list\_all}
\end{itemize}

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

\begin{itemize}
\item \textbf{syntax} “\_\textit{list}” :: “\textit{args} ⇒ \texttt{'}\texttt{a list}” (“[\_\_]”) \textbf{translations}
\begin{itemize}
\item “[\texttt{x, zs}]” == “\texttt{x \# [zs]}”
\item “[\texttt{x}]” == “\texttt{x \# [\_]}”
\end{itemize}
\end{itemize}

2.2 Command Syntax

2.2.1 \textbf{datatype}

\textbf{datatype} : \textit{local\_theory} → \textit{local\_theory}

\begin{itemize}
\item \textit{datatypes}
\item \textit{target}
\item \textit{dt-options}
\item \textit{dt-spec}
\end{itemize}

\textit{dt-options}

\begin{itemize}
\item \textit{plugins}
\item \textit{discs\_sels}
\item \textit{only}
\item \textit{del}
\end{itemize}

\textit{plugins}

\begin{itemize}
\item \textit{name}
\end{itemize}
The `datatype` command introduces a set of mutually recursive datatypes specified by their constructors.

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

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

- The `plugins` option indicates which plugins should be enabled (`only`) or disabled (`del`). By default, all plugins are enabled.
- The `discs_sels` option indicates that discriminators and selectors should be generated. The option is implicitly enabled if names are specified for discriminators or selectors.

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

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

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

The optional names preceding the type variables allow to override the default names of the set functions \((\text{set}_1\_t, \ldots, \text{set}_m\_t)\). Type arguments can be marked as dead by entering \textit{dead} in front of the type variable (e.g., \((\text{dead} \ 'a)\)); 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.
discs_sels option) but no name is supplied, the default is \( \lambda x. x = C_j \) for nullary constructors and \( t.is\_C_j \) otherwise.

\[ dt-ctor-arg \]

The syntactic entity \( type \) denotes a HOL type [12]. In addition to the type of a constructor argument, it is possible to specify a name for the corresponding selector. The same selector name can be reused for arguments to several constructors as long as the arguments share the same type. If selectors are enabled (cf. the discs_sels option) but no name is supplied, the default name is \( un\_C_j \).

### 2.2.2 datatype_compat

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

\begin{verbatim}
datatype_compat even_nat odd_nat
ML (Old_Datatype_Data.get_info @{theory} @{type_name even_nat})
\end{verbatim}

The syntactic entity \( name \) denotes an identifier [12]. The command is sometimes useful when migrating from the old datatype package to the new one.

A few remarks concern nested recursive datatypes:

- The old-style, nested-as-mutual induction rule and recursor theorems are generated under their usual names but with “compat_” prefixed (e.g., \texttt{compat\_tree.induct}, \texttt{compat\_tree.inducts}, and \texttt{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 data-types or the function type.

### 2.3 Generated Constants

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

- **Case combinator:** \( t\text{.case}_t \) (rendered using the familiar *case-of* syntax)
- **Discriminators:** \( t\text{.is}_C_1, \ldots, t\text{.is}_C_n \)
- **Selectors:** \( t\text{.un}_C_1k_1, \ldots, t\text{.un}_Cnk_n \)
- **Set functions:** \( t\text{.set}_1t, \ldots, t\text{.set}_mt \)
- **Map function:** \( t\text{.map}_t \)
- **Relator:** \( t\text{.rel}_t \)
- **Recursor:** \( t\text{.rec}_t \)

The discriminators and selectors are generated only if the *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 *datatype* are grouped in three broad categories:

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

The full list of named theorems can be obtained by issuing the command `print_theorems` immediately after the datatype definition. This list includes theorems produced by plugins (Section 8), but normally excludes low-level theorems that reveal internal constructions. To make these accessible, add the line

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

### 2.4.1 Free Constructor Theorems

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

- **t.inj** [iff, induct_simp]:
  \[(x21 ≠ x22 = y21 ≠ y22) = (x21 = y21 ∧ x22 = y22)\]

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

- **t.exhaust** [cases t, case_names C1 ... Cn]:
  \[y = \[] ⇒ P; \∧ x21. y = x21 ≠ x22 ⇒ P] ⇒ P\]

- **t.nchotomy**:\[∀ list. list = \[] ∨ (∃ x21 x22. list = x21 ≠ x22)\]

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

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

The next subgroup is concerned with the case combinator:

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

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

- **t.case_cong** [fundef_cong]:
  \[\[] = list'; list' = \[] ⇒ f1 = g1; \∧ x21 x22. list' = x21 ≠ x22 ⇒ f2 x21 x22 = g2 x21 x22\] ⇒ (case list of \[] ⇒ f1 | x21 ≠ x22 ⇒ f2 x21 x22) = (case list' of \[] ⇒ g1 | x21 ≠ x22 ⇒ g2 x21 x22)
The third subgroup revolves around discriminators and selectors:

\[ t.\text{case\_cong\_weak} \quad \text{[cong]}:\]
\[
\begin{align*}
\text{list} &= \text{list}' \quad \Rightarrow \quad (\text{case list of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 \ x \ xa) \quad = \quad (\text{case list of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 \ x \ xa)
\end{align*}
\]

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

\[ t.\text{split}:\]
\[
\begin{align*}
P \ (\text{case list of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 \ x \ xa) &= \quad ((\text{list} = [] \quad \Rightarrow \ P \ f1) \ \wedge \ \ (\forall \ x21 \ x22. \text{list} = x21 \# x22 \quad \Rightarrow \ P \ (f2 \ x21 \ x22)))
\end{align*}
\]

\[ t.\text{split\_asm}:\]
\[
\begin{align*}
P \ (\text{case list of } [] \Rightarrow f1 \mid x \# xa \Rightarrow f2 \ x \ xa) &= \quad (\neg \ (\text{list} = [] \quad \wedge \ \neg \ P \ f1 \ \lor \ (\exists \ x21 \ x22. \text{list} = x21 \# x22 \quad \wedge \ \neg \ P \ (f2 \ x21 \ x22))))
\end{align*}
\]

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

\[ t.\text{disc} \quad \text{[simp]}:\]
\[
\begin{align*}
\text{null} \ [] &= \quad \neg \text{null} \ (x21 \# x22)
\end{align*}
\]

\[ t.\text{discI}:\]
\[
\begin{align*}
\text{list} &= \ [] \quad \Rightarrow \quad \text{null \ list}
\text{list} &= \ x21 \# x22 \quad \Rightarrow \quad \neg \text{null \ list}
\end{align*}
\]

\[ t.\text{sel} \quad \text{[simp, code]}:\]
\[
\begin{align*}
\text{hd} \ (x21 \# x22) &= \ x21 \\
\text{tl} \ (x21 \# x22) &= \ x22
\end{align*}
\]

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

\[ t.\text{collapse} \quad \text{[simp]}:\]
\[
\begin{align*}
\text{null \ list} \quad \Rightarrow \quad \text{list} &= \ []
\neg \text{null \ list} \quad \Rightarrow \quad \text{hd \ list} \# \text{tl \ list} &= \ \text{list}
\end{align*}
\]

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.

\[ t.\text{distinct\_disc} \quad \text{[dest]}:\]

These properties are missing for 'a list because there is only one proper discriminator. If the datatype had been introduced with a second discriminator called nonnull, they would have read as follows:
\[
\begin{align*}
\text{null \ list} \quad \Rightarrow \quad \neg \text{nonnull \ list}
\text{nonnull \ list} \quad \Rightarrow \quad \neg \text{null \ list}
\end{align*}
\]
t.exhaust_disc [case_names C_1 \ldots C_n]:
\[ \text{null list } \Rightarrow P; \neg \text{null list } \Rightarrow P \] \Rightarrow P

t.exhaust_sel [case_names C_1 \ldots C_n]:
\[ \text{list } = \[] \Rightarrow P; \text{list } = \text{hd list } \# \text{tl list } \Rightarrow P \] \Rightarrow P

t.expand:
\[
\begin{align*}
\text{null list } &= \text{null list}'; \neg \text{null list}; \neg \text{null list}' & \Rightarrow \text{hd list } &= \text{hd list}' \\
\wedge \text{tl list } &= \text{tl list}' & \Rightarrow \text{list } &= \text{list}'
\end{align*}
\]

t.split_sel:
\[
\begin{align*}
P \ (\text{case list of } \[] \Rightarrow f_1 \mid x \# xa \Rightarrow f_2 (x xa)) &= ((\text{list } = \[] \rightarrow P f_1) \wedge (\text{list } = \text{hd list } \# \text{tl list } \rightarrow P (f_2 (\text{hd list }) (\text{tl list}))))
\end{align*}
\]

t.split_sel_asm:
\[
\begin{align*}
P \ (\text{case list of } \[] \Rightarrow f_1 \mid x \# xa \Rightarrow f_2 (x xa)) &= (\neg (\text{list } = \[] \wedge \neg P \\
f_1 \lor \text{list } &= \text{hd list } \# \text{tl list } \Rightarrow \neg P (f_2 (\text{hd list }) (\text{tl list}))))
\end{align*}
\]

t.split_sels = split_sel split_sel_asm

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

t.disc_eq_case:
\[
\begin{align*}
\text{null list } &= (\text{case list of } \[] \Rightarrow \text{True} \mid \text{uu} \# \text{uua} \Rightarrow \text{False}) \\
\neg \text{null list } &= (\text{case list of } \[] \Rightarrow \text{False} \mid \text{uu} \# \text{uua} \Rightarrow \text{True})
\end{align*}
\]

In addition, equational versions of t.disc are registered with the [code] attribute. The [code] attribute is set by the 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., '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:

t.case_transfer [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 transfer plugin (Section 8.3).

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).
2 Defining Datatypes

\texttt{t.ctr\_transfer} \texttt{[transfer\_rule]}:
\begin{itemize}
\item \texttt{list\_all2} \texttt{R} \texttt{[]} 
\item \texttt{rel\_fun} \texttt{R} (\texttt{rel\_fun} (\texttt{list\_all2} \texttt{R}) (\texttt{list\_all2} \texttt{R}) \texttt{op} \# \texttt{op} 
\end{itemize}

The \texttt{[transfer\_rule]} attribute is set by the \texttt{transfer} plugin (Section 8.3).

\texttt{t.disc\_transfer} \texttt{[transfer\_rule]}:
\begin{itemize}
\item \texttt{rel\_fun} \texttt{(list\_all2} \texttt{R}) \texttt{op} = \texttt{null} \texttt{null}
\item \texttt{rel\_fun} \texttt{(list\_all2} \texttt{R}) \texttt{op} = (\lambda x. \neg \texttt{null} \texttt{list}) (\lambda y. \neg \texttt{null} \texttt{list})
\end{itemize}

The \texttt{[transfer\_rule]} attribute is set by the \texttt{transfer} plugin (Section 8.3).

\texttt{t.set} \texttt{[simp, code]}:
\begin{itemize}
\item \texttt{set} \texttt{[]} \texttt{=} \texttt{[]} 
\item \texttt{set} \texttt{(x21 \# x22)} \texttt{=} \texttt{insert} \texttt{x21} \texttt{(set} \texttt{x22)}
\end{itemize}

The \texttt{[code]} attribute is set by the \texttt{code} plugin (Section 8.1).

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

\texttt{t.set\_intros}:
\begin{itemize}
\item \texttt{x21 \in set} \texttt{(x21 \# x22)}
\item \texttt{y \in set} \texttt{x22} \implies \texttt{y \in set} \texttt{(x21 \# x22)}
\end{itemize}

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

\texttt{t.map} \texttt{[simp, code]}:
\begin{itemize}
\item \texttt{map} \texttt{f} \texttt{[]} \texttt{=} \texttt{[]} 
\item \texttt{map} \texttt{f} \texttt{(x21 \# x22)} \texttt{=} \texttt{f} \texttt{x21} \texttt{\# map} \texttt{f} \texttt{x22}
\end{itemize}

The \texttt{[code]} attribute is set by the \texttt{code} plugin (Section 8.1).

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

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

\texttt{t.pred\_inject} \texttt{[simp]}:
\begin{itemize}
\item \texttt{list\_all} \texttt{P} \texttt{[]} 
\item \texttt{list\_all} \texttt{P (a \# aa)} \texttt{=} \texttt{(P a \land list\_all} \texttt{P aa)}
\end{itemize}

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

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

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

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

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

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

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

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

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

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

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

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

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

\[ t.\text{map\_id0}: \]
\[ \text{map} \ id = id \]
2  Defining Datatypes

\[ t.\text{map}_\text{id} \]
\[ \text{map } \text{id} \ t = t \]

\[ t.\text{map}_\text{ident} \]
\[ \text{map} (\lambda x. \ x) \ t = t \]

\[ t.\text{map}_\text{transfer} \ [\text{transfer}_\text{rule}] \]
\[ \text{rel}_\text{fun} (\text{rel}_\text{fun} \ Rb \ Sd) (\text{rel}_\text{fun} (\text{list}_\text{all}2 \ Rb) (\text{list}_\text{all}2 \ Sd)) \ \text{map} \]
\[ \text{map} \]

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

\[ t.\text{pred}_\text{cong} \ [\text{fundef}_\text{cong}] \]
\[ [x = ya; \ \land \ z. \ z \in \text{set} \ ya \implies P z = Pa z] \implies \text{list}_\text{all} \ P x = \text{list}_\text{all} \ Pa \ ya \]

\[ t.\text{pred}_\text{cong}_\text{simp} \]
\[ [x = ya; \ \land \ z. \ z \in \text{set} \ ya = \text{simp=} > P z = Pa z] \implies \text{list}_\text{all} \ P x = \text{list}_\text{all} \ Pa \ ya \]

\[ t.\text{pred}_\text{map} \]
\[ \text{list}_\text{all} \ Q (\text{map} \ f \ x) = \text{list}_\text{all} (Q \circ f) \ x \]

\[ t.\text{pred}_\text{mono}_\text{strong} \]
\[ [\text{list}_\text{all} \ P \ x; \ \land \ z. \ z \in \text{set} \ x; \ P z \implies Pa z] \implies \text{list}_\text{all} \ Pa \ x \]

\[ t.\text{pred}_\text{rel} \]
\[ \text{list}_\text{all} \ P x = \text{list}_\text{all}2 (\text{eq}_\text{onp} \ P) \ x \]

\[ t.\text{pred}_\text{set} \]
\[ \text{list}_\text{all} \ P = (\lambda x. \ \text{Ball} (\text{set} \ x) \ P) \]

\[ t.\text{pred}_\text{transfer} \ [\text{transfer}_\text{rule}] \]
\[ \text{rel}_\text{fun} (\text{rel}_\text{fun} \ R \ \text{op} =) (\text{rel}_\text{fun} (\text{list}_\text{all}2 \ R) \ \text{op} =) \ \text{list}_\text{all} \ \text{list}_\text{all} \]
\[ \text{The } \text{[transfer}_\text{rule}] \text{ attribute is set by the } \text{transfer} \text{ plugin (Section 8.3)} \]
\[ \text{for type constructors with no dead type arguments}. \]

\[ t.\text{pred}_\text{True} \]
\[ \text{list}_\text{all} (\lambda _. \ \text{True}) = (\lambda _. \ \text{True}) \]

\[ t.\text{set}_\text{map} \]
\[ \text{set} (\text{map} \ f \ v) = f \mapsto \text{set} \ v \]

\[ t.\text{set}_\text{transfer} \ [\text{transfer}_\text{rule}] \]
\[ \text{rel}_\text{fun} (\text{list}_\text{all}2 \ R) (\text{rel}_\text{set} \ R) \ \text{set} \]
\[ \text{The } \text{[transfer}_\text{rule}] \text{ attribute is set by the } \text{transfer} \text{ plugin (Section 8.3)} \]
\[ \text{for type constructors with no dead type arguments}. \]

\[ t.\text{rel}_\text{compp} \ [\text{relator}_\text{distr}] \]
\[ \text{list}_\text{all}2 (R \ \text{OO} \ S) = \text{list}_\text{all}2 R \ \text{OO} \ \text{list}_\text{all}2 \ S \]
\[ \text{The } \text{[relator}_\text{distr}] \text{ attribute is set by the } \text{lifting} \text{ plugin (Section 8.4)}. \]
### Defining Datatypes

**t.rel_conversep:**
\[
\text{list}_\text{all}^2 R^- = (\text{list}_\text{all}^2 R)^-\]

**t.rel_eq:**
\[
\text{list}_\text{all}^2 \text{op} = \text{op} =
\]

**t.rel_eq_onp:**
\[
\text{list}_\text{all}^2 (\text{eq}_\text{onp} P) = \text{eq}_\text{onp} (\text{list}_\text{all} P)
\]

**t.rel_flip:**
\[
\text{list}_\text{all}^2 R^- a b = \text{list}_\text{all}^2 R^- b a
\]

**t.rel_map:**
\[
\text{list}_\text{all}^2 Sb (\text{map } i x) y = \text{list}_\text{all}^2 (\lambda x. Sb (i x)) x y
\]
\[
\text{list}_\text{all}^2 Sa x (\text{map } g y) = \text{list}_\text{all}^2 (\lambda x y. Sa x (g y)) x y
\]

**t.rel_mono [mono, relator_mono]:**
\[
R \leq Ra \implies \text{list}_\text{all}^2 R \leq \text{list}_\text{all}^2 Ra
\]

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

**t.rel_mono_strong [list_all2 R x y; \forall z yb. [z \in \text{set } x; yb \in \text{set } y; R z yb] \implies Ra z yb]**
\[
\implies \text{list}_\text{all}^2 R a x y
\]

**t.rel_cong [fundef_cong]:**
\[
[x = ya; y = xa; \forall z yb. [z \in \text{set } ya; yb \in \text{set } xa] \implies R z yb = Ra z yb]
\]
\[
\implies \text{list}_\text{all}^2 R x y = \text{list}_\text{all}^2 Ra ya xa
\]

**t.rel_cong_simp:**
\[
[x = ya; y = xa; \forall z yb. [z \in \text{set } ya; yb \in \text{set } xa] = \text{simp} = \implies yb \in \text{set } xa = \text{simp} =
\]
\[
R z yb = Ra z yb \implies \text{list}_\text{all}^2 R x y = \text{list}_\text{all}^2 Ra ya xa
\]

**t.rel_refl:**
\[
(\forall x. Ra x x) \implies \text{list}_\text{all}^2 R a x
\]

**t.rel_refl_strong:**
\[
(\forall z. z \in \text{set } x \implies Ra z z) \implies \text{list}_\text{all}^2 R a x x
\]

**t.rel_refl:**
\[
\text{reflp } R \implies \text{reflp} (\text{list}_\text{all}^2 R)
\]

**t.rel_symp:**
\[
\text{symp } R \implies \text{symp} (\text{list}_\text{all}^2 R)
\]

**t.rel_transp:**
\[
\text{transp } R \implies \text{transp} (\text{list}_\text{all}^2 R)
\]

**t.rel_transfer [transfer_rule]:**
\[
\text{rel_fun} (\text{rel_fun } Sa (\text{rel_fun } Sc \text{ op } =)) (\text{rel_fun} (\text{list}_\text{all}^2 Sa) (\text{rel_fun}
\]
\[
(\text{list}_\text{all}^2 Sc) \text{ op } =)) \text{ list}_\text{all}^2 \text{ list}_\text{all}^2
\]
The \textit{transfer_rule} attribute is set by the \textit{transfer} plugin (Section 8.3) for type constructors with no dead type arguments.

### 2.4.3 Inductive Theorems

The inductive theorems are as follows:

- \texttt{t.induct} [case_names \emph{C}_1 \ldots \emph{C}_n, induct \emph{t}]:

  \[
  [P [] ; \forall x_1 x_2. P x_2 \implies P (x_1 \# x_2)] \implies P \text{ list}
  \]

- \texttt{t.rel_induct} [case_names \emph{C}_1 \ldots \emph{C}_n, induct pred]:

  \[
  [\text{list_all} \emph{R} \emph{x y}; Q [] [] ; \forall a_21 a_22 b_21 b_22. [R a_21 b_21; Q a_22 b_22] \implies Q (a_21 \# a_22) (b_21 \# b_22)] \implies Q \emph{x y}
  \]

- \texttt{t1,..,tm.induct} [case_names \emph{C}_1 \ldots \emph{C}_n]:

- \texttt{t1,..,tm.rel_induct} [case_names \emph{C}_1 \ldots \emph{C}_n]:

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

- \texttt{t.rec} [simp, code]:

  \[
  \text{rec_list} \emph{f} \emph{f2} [] = \emph{f1}
  \]

  \[
  \text{rec_list} \emph{f} \emph{f2} (\emph{x21} \# \emph{x22}) = \emph{f2} \emph{x21} \emph{x22} (\text{rec_list} \emph{f} \emph{f2} \emph{x22})
  \]

  The \textit{code} attribute is set by the \textit{code} plugin (Section 8.1).

- \texttt{t.rec_o_map}:

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

- \texttt{t.rec_transfer} [transfer_rule]:

  \[
  \text{rel_fun} \emph{S} (\text{rel_fun} (\text{rel_fun} \emph{R} (\text{rel_fun} (\text{list_all} \emph{R}) \emph{S})) (\text{rel_fun} (\text{list_all} \emph{R}) \emph{S})) \text{ rec_list rec_list}
  \]

  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{datatype} also provides the following collection:

\[
\begin{align*}
\texttt{t.simps} &= \texttt{t.inject t.distinct t.case t.rec t.map t.rel_inject} \\
& \quad \texttt{t.rel_distinct t.set}
\end{align*}
\]

### 2.5 Proof Method

#### 2.5.1 \texttt{countable_datatype}

The theory \texttt{-/-src/HOL/Library/Countable.thy} provides a proof method called \texttt{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 \texttt{t.inducts} for \texttt{t.induct} is no longer generated. For \( m > 1 \) mutually recursive datatypes, \texttt{rec_t1,...,tm_i} has been renamed \texttt{rec_t_i} for each \( i \in \{1, \ldots, m\} \), \texttt{t1,...,tm.inducts(i)} has been renamed \texttt{t_i.induct} for each \( i \in \{1, \ldots, m\} \), and the collection \texttt{t1,...,tm.size} (generated by the \texttt{size} plugin, Section 8.2) has been divided into \texttt{t1.size}, \ldots, \texttt{tm.size}.

• The \texttt{t.simps} collection has been extended. Previously available theorems are available at the same index as before.

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

The old command is still available as \texttt{old_datatype} in theory \texttt{~~/src/HOL/Library/Old_Datatype.thy}. However, there is no general way to register old-style datatypes as new-style datatypes. If the objective is to define new-style datatypes with nested recursion through old-style datatypes, the old-style datatypes can be registered as a BNF (Section 6). If the objective is to derive discriminators and selectors, this can be achieved using \texttt{free_constructors} (Section 7).

3 Defining Primitively Recursive Functions

Recursive functions over datatypes can be specified using the \texttt{primrec} command, which supports primitive recursion, or using the more general \texttt{fun}, \texttt{function}, and \texttt{partial_function} commands. In this tutorial, the focus is on \texttt{primrec}; \texttt{fun} and \texttt{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 \texttt{~~/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:

\begin{verbatim}
primrec (nonexhaustive) bool_of_trool :: “trool ⇒ bool” where
\end{verbatim}
“bool_of_trool False ↔ False”
| “bool_of_trool True ↔ True”

primrec the_list :: “a option ⇒ ’a list” where
“the_list None = []”
| “the_list (Some a) = [a]”

primrec the_default :: “’a ⇒ ’a option ⇒ ’a” where
“the_default d None = d”
| “the_default _ (Some a) = a”

primrec mirror :: “(’a, ’b, ’c) triple ⇒ (’c, ’b, ’a) triple” where
“mirror (Triple a b c) = Triple c b a”

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

3.1.2 Simple Recursion

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

primrec replicate :: “nat ⇒ ’a ⇒ ’a list” where
“replicate Zero _ = []”
| “replicate (Succ n) x = x # replicate n x”

primrec (nonexhaustive) at :: “’a list ⇒ nat ⇒ ’a” where
“at (x # xs) j =
   (case j of
    Zero ⇒ x
    | Succ j’ ⇒ at xs j)”

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

Pattern matching is only available for the argument on which the recursion takes place. Fortunately, it is easy to generate pattern-matching equations using the simps_of_case command provided by the theory ~/src/HOL/Library/Simps_Case_Conv.thy.

simps_of_case at_simps_alt: at.simps

This generates the lemma collection at_simps_alt:

at (x # xs) Zero = x  at (xa # xs) (Succ x) = at xs x
The next example is defined using `fun` to escape the syntactic restrictions imposed on primitively recursive functions:

```
fun at_least_two :: “nat ⇒ bool” where
  “at_least_two (Succ (Succ _))” → True
| “at_least_two _” → False
```

### 3.1.3 Mutual Recursion

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

```
primrec
  nat_of_even_nat :: “even_nat ⇒ nat” and
  nat_of_odd_nat :: “odd_nat ⇒ nat”
where
  “nat_of_even_nat Even_Zero = Zero”
| “nat_of_even_nat (Even_Succ n) = Succ (nat_of_odd_nat n)”
| “nat_of_odd_nat (Odd_Succ n) = Succ (nat_of_even_nat n)”
primrec
  eval_e :: “(′a ⇒ int) ⇒ (′b ⇒ int) ⇒ (′a, ′b) exp ⇒ int” and
  eval_t :: “(′a ⇒ int) ⇒ (′b ⇒ int) ⇒ (′a, ′b) trm ⇒ int” and
  eval_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`:

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

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

```ml
primrec atff :: "'a treeff ⇒ nat list ⇒ 'a" where
  "atff (Nodeff a ts) js =
    (case js of
      [] ⇒ a
    | j # js' ⇒ at (map (λt. atff t js') ts) j)"
```

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

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

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

```ml
primrec relabel_ft :: "('a⇒'a) ⇒ 'a ftree ⇒ 'a ftree" where
  "relabel_ft f (FTLeaf x) = FTLeaf (f x)"
  | "relabel_ft f (FTNode g) = FTNode (op ◦ (relabel_ft f) ◦ g)"
```

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

```ml
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))"

primrec (nonexhaustive) subtree_ft :: "'a ⇒ 'a ftree ⇒ 'a ftree" where
  "subtree_ft x (FTNode g) = g x"
```

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

```ml
datatype 'a ftree2 = FTLeaf2 'a | FTreeNode2 "'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) = FTreeNode2 (op ◦ (op ◦ (relabel_ft2 f)) g)"

primrec relabel_ft2 :: "('a ⇒ 'a) ⇒ 'a ftree2 ⇒ 'a ftree2" where
```

---

**3 Defining Primitively Recursive Functions**

26
3 Defining Primitively Recursive Functions

“relabel\_ft f (FTLeaf x) = FTLeaf f(x)”
| “relabel\_ft f (FTNode g) = FTNode (\(xy.\) relabel\_ft f (g x y))”

primrec (nonexhaustive) subtree\_ft :: “'a ⇒ 'a ftree ⇒ 'a ftree” where

“subtree\_ft x y (FTNode g) = g x y”

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

primrec increasing\_tree :: “\(int ⇒ int ftree ⇒ bool\)” where

“increasing\_tree m (Node 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 through new-style datatypes. For example:

primrec (nonexhaustive)

at\_ff :: “'a ftree ff ⇒ nat list ⇒ 'a” and
ats\_ff :: “'a ftree ff list ⇒ nat ⇒ nat list ⇒ 'a”

where

“at\_ff (Node a ts) js =
\(\text{case js of}
\[
\text{[] ⇒ a}
\mid j \# js' ⇒ ats\_ff ts j js'\]”
| “ats\_ff (t \# ts) j =
\(\text{case j of}
\[
\text{Zero ⇒ at\_ff t}
\mid \text{Succ j' ⇒ ats\_ff ts j'}\]”

Appropriate induction rules are generated as at\_ff.induct, ats\_ff.induct, and at\_ff.ats\_ff.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 :: “\(\{zero, plus\} \text{ btree ⇒ 'a}\)” and
sum\_btree\_option :: “\(\text{btree option ⇒ 'a}\)”

where

“sum\_btree (BNode a lt rt) =
\(a + \text{sum\_btree\_option lt + sum\_btree\_option rt}\)”
| “sum\_btree\_option None = 0”
| “sum\_btree\_option (Some t) = sum\_btree t”
3.2 Command Syntax

3.2.1 primrec

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

The \textbf{primrec} command introduces a set of mutually recursive functions over datatypes.

The syntactic entity \textit{target} can be used to specify a local context, \textit{fixes} denotes a list of names with optional type signatures, \textit{thmdecl} denotes an optional name for the formula that follows, and \textit{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 nonehaustive option indicates that the functions are not necessarily specified for all constructors. It can be used to suppress the warning that is normally emitted when some constructors are missing.

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

3.3 Generated Theorems

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

\[ f \text{. simps [simp, code]}; \]
\[ tfold uu (TNil y) = y \]
\[ tfold f (TCons x xs) = f x (tfold f xs) \]

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

\[ f \text{. transfer [transfer_rule]}; \]
\[ rel\_fun (rel\_fun R2 (rel\_fun R1 R1)) (rel\_fun (rel\_tlist R2 R1) R1) tfold tfold \]

This theorem is generated by the transfer plugin (Section 8.3) for functions declared with the transfer option enabled.

\[ f \text{. induct [case_names C}_1 \ldots C_n]; \]

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

\[ f_{1\ldots m} \text{. induct [case_names C}_1 \ldots C_n]; \]

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

3.4 Recursive Default Values for Selectors

A datatype selector un_D can have a default value for each constructor on which it is not otherwise specified. Occasionally, it is useful to have the default value be defined recursively. This leads to a chicken-and-egg situation, because the datatype is not introduced yet at the moment when the selectors are introduced. Of course, we can always define the selectors
3 Defining Primitively Recursive Functions

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 \( \text{un}_D 0 \) using \texttt{consts}.
2. Define the datatype, specifying \( \text{un}_D 0 \) as the selector’s default value.
3. Define the behavior of \( \text{un}_D 0 \) on values of the newly introduced datatype using the \texttt{overloading} command.
4. Derive the desired equation on \( \text{un}_D \) from the characteristic equations for \( \text{un}_D 0 \).

The following example illustrates this procedure:

\[
\text{consts} \quad \text{termi}_0 :: 'a
\]
\[
\text{datatype} \quad ('a, 'b) \text{tlist} =
\]
\[
\quad \text{TNil} (\text{termi}; 'b)
\|
\quad \text{TCons} (\text{thd}; 'a) (\text{ttl}; "('a, 'b) tlist")
\]
\[
\text{where}
\]
\[
\quad "\text{ttl} (\text{TNil} y) = \text{TNil} y"
\|
\quad "\text{termi} (\text{TCons \_ xs}) = \text{termi}_0 xs"
\]
\[
\text{overloading}
\]
\[
\quad \text{termi}_0 \equiv "\text{termi}_0 :: ('a, 'b) \text{tlist} \Rightarrow 'b"
\]
\[
\text{begin}
\]
\[
\quad \text{primrec} \quad \text{termi}_0 :: "('a, 'b) \text{tlist} \Rightarrow 'b" \text{ where}
\]
\[
\quad "\text{termi}_0 (\text{TNil} y) = y"
\|
\quad "\text{termi}_0 (\text{TCons x xs}) = \text{termi}_0 xs"
\]
\[
\text{end}
\]
\[
\text{lemma} \quad \text{termi}_\text{tCons}[\text{simp}]; "\text{termi} (\text{TCons x xs}) = \text{termi} xs"
\]
\[
\text{by} \quad \text{(cases xs)} \quad \text{auto}
\]

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, \( f_1 \ldots f_m \text{.simps} \) has been broken down into separate sub-collections \( f_i \text{.simps} \).
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) llist = lnull: LNil | LCons (lhd: 'a) (ltl: "'a llist")
for map: lmap
rel: llist_all2
pred: llist_all
where
"l tl 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) 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 (ESucc (...)))`, that represents \( \infty \). In addition, it has finite values of the form `ESucc (ESucc (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:

```plaintext
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:

```plaintext
codatatype ’a tree isi = Node isi (lbl isi: ’a) (sub isi: “’a tree isi llist”)
codatatype ’a tree isi sisi = Node isi sisi (lbl isi sisi: ’a) (sub isi sisi: “’a tree isi sisi fset”)  
codatatype ’a sm = SM (accept: bool) (trans: “’a ⇒ ’a sm”)  
```

4.2 Command Syntax

4.2.1 codatatype

```plaintext
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 Defining 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:

Corecursor: \( t.corec_t \)

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 \( 'a \ llist \):

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

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

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

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

### \( t_1 \ldots t_m \).coinduct

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

### \( t_1 \ldots t_m \).coinduct_strong

\[
\begin{align*}
t_1 \ldots t_m &.\text{coinduct_strong} \quad \text{[case_names } t_1 \ldots t_m, \text{ case_conclusion } D_1 \ldots D_n]\end{align*}
\]

### \( t_1 \ldots t_m \).rel_coinduct

\[
\begin{align*}
t_1 \ldots t_m &.\text{rel_coinduct} \quad \text{[case_names } t_1 \ldots t_m, \text{ case_conclusion } D_1 \ldots D_n]\end{align*}
\]

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

### \( t_1 \ldots t_m \).set_induct

\[
\begin{align*}
t_1 \ldots t_m &.\text{set_induct} \quad \text{[case_names } C_1 \ldots C_n, \text{ induct set: } \text{set}_j t_1, \ldots, \text{ induct set: } \text{set}_j t_m]\end{align*}
\]

\[
\begin{align*}
[x \in \text{lset } a; \land z_1 z_2. \ P z_1 (LCons z_1 z_2); \land z_1 z_2 xa. \ [xa \in \text{lset } z_2; \ P xa z_2] = P x a] \Rightarrow P x a
\end{align*}
\]

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

### \( t.\text{corec} \)

\[
\begin{align*}
p a & \Rightarrow \text{corec}_\text{llist} p g21 q22 g221 g222 a = LNil \\
\neg p a & \Rightarrow \text{corec}_\text{llist} p g21 q22 g221 g222 a = LCons (g21 a) \quad \text{(if } q22 a \text{ then } g221 a \text{ else } \text{corec}_\text{llist} p g21 q22 g221 g222 (g222 a)\text{)}
\end{align*}
\]

### \( t.\text{corec}_\text{code} \)

\[
\begin{align*}
\text{corec}_\text{llist} p g21 q22 g221 g222 a = (\text{if } p a \text{ then } \text{LNil} \text{ else } \text{LCons} (g21 a) \text{ (if } q22 a \text{ then } g221 a \text{ else } \text{corec}_\text{llist} p g21 q22 g221 g222 (g222 a)))
\end{align*}
\]

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

### \( t.\text{corec}_\text{disc} \)

\[
\begin{align*}
p a & \Rightarrow \text{lnull} \ (\text{corec}_\text{llist} p g21 q22 g221 g222 a) \\
\neg p a & \Rightarrow \neg \text{lnull} \ (\text{corec}_\text{llist} p g21 q22 g221 g222 a)
\end{align*}
\]

### \( t.\text{corec}_\text{disc}_\text{iff} \)

\[
\begin{align*}
\text{lnull} \ (\text{corec}_\text{llist} p g21 q22 g221 g222 a) = p a \\
\neg \text{lnull} \ (\text{corec}_\text{llist} p g21 q22 g221 g222 a)) = (\neg p a)
\end{align*}
\]

### \( t.\text{corec}_\text{sel} \)

\[
\begin{align*}
\neg p a & \Rightarrow \text{lhd} \ (\text{corec}_\text{llist} p g21 q22 g221 g222 a) = g21 a \\
\neg p a & \Rightarrow \text{lll} \ (\text{corec}_\text{llist} p g21 q22 g221 g222 a) = (\text{if } q22 a \text{ then } g221 a \text{ else } \text{corec}_\text{llist} p g21 q22 g221 g222 (g222 a))
\end{align*}
\]

### \( t.\text{map}_\circ \text{corec} \)

\[
\begin{align*}
\text{lmap } f \circ \text{corec}_\text{llist} g ga gb gc gd & = \text{corec}_\text{llist} g (f \circ ga) gb (\text{lmap } f \circ gc) gd
\end{align*}
\]

### \( t.\text{corec}_\text{transfer} \)

\[
\begin{align*}
\text{rel}_\text{fun} \ (\text{rel}_\text{fun } S \ op =) & \quad \text{(rel}_\text{fun } (\text{rel}_\text{fun } S \ R) \ (\text{rel}_\text{fun } (\text{rel}_\text{fun } S \ op =) \ (\text{rel}_\text{fun } (\text{rel}_\text{fun } S \ (\text{llist_all} R)) \ (\text{rel}_\text{fun } (\text{rel}_\text{fun } S \ S) \ (\text{rel}_\text{fun } S \ (\text{llist_all} R)))))) \text{ corec}_\text{llist} \text{ corec}_\text{llist}
\end{align*}
\]
The \texttt{transfer_rule} attribute is set by the \texttt{transfer} plugin (Section 8.3) for type constructors with no dead type arguments.

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

\begin{itemize}
\item \texttt{t.simps = t.inject t.distinct t.case t.corec_disc_iff t.corec_sel t.map t.rel_inject t.rel_distinct t.set}
\end{itemize}

\section{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 $f$:

\begin{itemize}
\item The \textit{destructor view} specifies $f$ by implications of the form
  \[ \ldots \implies \text{is}_i C_j \left( f \ x_1 \ldots \ x_n \right) \]
  and equations of the form
  \[ \text{un}_i C_j \left( f \ x_1 \ldots \ x_n \right) = \ldots \]
  This style is popular in the coalgebraic literature.
\item 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.
\item 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.
\end{itemize}
5 Defining Primitively Corecursive Functions

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))"

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 \([x, g x, g (g x), \ldots]\). Productivity guarantees that prefixes \([x, g x, g (g x), \ldots, (g \ldots^k) x]\) of arbitrary finite length \(k\) can be computed by unfolding the code equation a finite number of times.

Corecursive functions construct codatatype values, but nothing prevents them from also consuming such values. The following function drops every second element in a stream:

```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 

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

This generates the lemma collection lapp_simps:

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

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 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))))"
```

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
  even_infty :: even_enat and
  odd_infty :: odd_enat
where
```

5. Defining Primitively Corecursive Functions
\begin{quote}
"even_infty = Even_ESucc odd_infty"
| "odd_infty = Odd_ESucc even_infty"
\end{quote}

### 5.1.3 Nested Corecursion

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

**primcorec iterate**_{\textit{i}, i} :: "(\textit{\prime}a \Rightarrow \textit{\prime}a \textit{llist}) \Rightarrow \textit{\prime}a \Rightarrow \textit{\prime}a \textit{tree}_{\textit{i}, i}" where
\begin{align*}
\text{iterate}_{\textit{i}, i} g x &= \text{Node}_{\textit{i}, i} x (\text{lmap} (\text{iterate}_{\textit{i}, i} g) (g x))
\end{align*}

**primcorec iterate**_{\textit{i}, s} :: "(\textit{\prime}a \Rightarrow \textit{\prime}a \textit{fset}) \Rightarrow \textit{\prime}a \Rightarrow \textit{\prime}a \textit{tree}_{\textit{i}, s}" where
\begin{align*}
\text{iterate}_{\textit{i}, s} g x &= \text{Node}_{\textit{i}, s} x (\text{fimage} (\text{iterate}_{\textit{i}, s} g) (g x))
\end{align*}

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

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

**primcorec tree**_{\textit{i}, i} of stream :: "\textit{\prime}a \textit{stream} \Rightarrow \textit{\prime}a \textit{tree}_{\textit{i}, i}" where
\begin{align*}
\text{tree}_{\textit{i}, i} \text{ of stream } s &= \\
&= \text{Node}_{\textit{i}, i} (\text{shd} s) (\text{lmap} \text{tree}_{\textit{i}, i} \text{ of stream} (\text{LCons} (\text{stl} s) \text{LNil})))
\end{align*}

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

**primcorec tree**_{\textit{i}, i} of stream :: "\textit{\prime}a \textit{stream} \Rightarrow \textit{\prime}a \textit{tree}_{\textit{i}, i}" where
\begin{align*}
\text{tree}_{\textit{i}, i} \text{ of stream } s &= \\
&= \text{Node}_{\textit{i}, i} (\text{shd} s) (\text{LCons} (\text{tree}_{\textit{i}, i} \text{ of stream} (\text{stl} s) \text{LNil})))
\end{align*}

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

**primcorec sm of dfa** :: "(\textit{\prime}q \Rightarrow \textit{\prime}a \Rightarrow \textit{\prime}q \Rightarrow \textit{\prime}q \Rightarrow \textit{\prime}a \textit{sm})\ where
\begin{align*}
\text{sm of dfa} \; \delta \; F \; q &= \text{SM} \; (q \in F) \; (\text{sm of dfa} \; \delta \; F \circ \delta \; q)
\end{align*}

The map function for the function type \((\Rightarrow)\) is composition \((\text{op } \circ)\). For convenience, corecursion through functions can also be expressed using \(\lambda\)-abstractions and function application rather than through composition. For example:
5 Defining Primitively Corecursive Functions

```haskell
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))"
```

```haskell
primcorec empty_sm :: "'a sm" where
  "empty_sm = SM False (λ_. empty_sm)"
```

```haskell
primcorec not_sm :: "'a sm ⇒ 'a sm" where
  "not_sm M = SM (¬ accept M) (λa. not_sm (trans M a))"
```

```haskell
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 \( op \circ \) are necessary. The examples below illustrate the case where \( n = 2 \):

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

```haskell
primcorec
  sm2_of_dfa :: "'(q ⇒ 'a ⇒ 'b ⇒ 'q) ⇒ 'q set ⇒ 'q ⇒ ('a, 'b) sm2" where
  "sm2_of_dfa δ F q = SM2 (q ∈ F) (op ∘ (op ∘ (sm2_of_dfa δ F))) (δ q)"
```

Coinduction rules are generated as `iterate_i_i`, `iterates_i_i`, `coinduct`, and analogously for `coinduct_strong`. These rules and the underlying corecursors are generated dynamically and are kept in a cache to speed up subsequent definitions.

5.1.4 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:

```haskell
primcorec
  iterate_i_i :: "'(a ⇒ 'a llist) ⇒ 'a ⇒ 'a tree_i_i" and
  iterates_i_i :: "'(a ⇒ 'a llist) ⇒ 'a llist ⇒ 'a tree_i_i llist" where
  "iterate_i_i g x = Node_i_i x (iterates_i_i g (g x))"
```

Coinduction rules are generated as `iterate_i_i.coinduct`, `iterates_i_i.coinduct`, and `iterate_i_i.iterates_i_i.coinduct` and analogously for `coinduct_strong`. These rules and the underlying corecursors are generated dynamically and are kept in a cache to speed up subsequent definitions.
5.1.5 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:

```
primcorec lapp :: "'a list ⇒ 'a list ⇒ 'a list" 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:

```
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 sequential option. Consider the following examples:

\[
\text{primcorec literate :: } \langle \text{'a} \Rightarrow \text{'a} \rangle \Rightarrow \text{'a} \Rightarrow \text{'a} \text{llist} \rangle \text{ where}
\]
\[
\text{"lnull (literate _ x)"}
\]
\[
\text{"lhd (literate _ x) = x"}
\]
\[
\text{"ltl (literate g x) = literate g (g x)"}
\]

\[
\text{primcorec siterate :: } \langle \text{'a} \Rightarrow \text{'a} \rangle \Rightarrow \text{'a} \Rightarrow \text{'a} \text{ stream} \rangle \text{ where}
\]
\[
\text{"shd (siterate _ x) = x"}
\]
\[
\text{"stl (siterate g x) = siterate g (g x)"}
\]

\[
\text{primcorec every_snd :: } \text{a} \text{ stream} \Rightarrow \text{a} \text{ stream} \rangle \text{ where}
\]
\[
\text{"shd (every_snd s) = shd s"}
\]
\[
\text{"stl (every_snd s) = stl (stl s)"}
\]

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

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

\[
\text{primcorec lapp :: } \langle \text{a} \text{llist} \Rightarrow \text{a} \text{llist} \Rightarrow \text{a} \text{llist} \rangle \text{ where}
\]
\[
\text{"lnull xs} \Rightarrow \text{lnull ys} \Rightarrow \text{lnull (lapp xs ys)}"\]
\[
\text{"lhd (lapp xs ys) = lhd (if lnull xs then ys else xs)"}\]
\[
\text{"ltl (lapp xs ys) = (if xs = LNil then ltl ys else lapp (ltl xs) ys)"}
\]

For a codatatype with \(n\) constructors, it is sufficient to specify \(n - 1\) discriminator formulas. The command will then assume that the remaining constructor should be taken otherwise. This can be made explicit by adding
\[
\text{"} \Rightarrow \neg \text{lnull (lapp xs ys)"}
\]
to the specification. The generated selector theorems are conditional.

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

\[
\text{primcorec random_process :: } \langle \text{a} \text{ stream} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{a} \text{ process} \rangle \text{ where}
\]
\[
\text{"n mod 4 = 0} \Rightarrow \text{random_process s f n = Fail"}
\]
5 Defining Primitively Corecursive Functions

| “\(n \mod 4 = 1 \implies \text{is Skip } (\text{random}\_\text{process } s f n)\)” |
| “\(n \mod 4 = 2 \implies \text{is Action } (\text{random}\_\text{process } s f n)\)” |
| “\(n \mod 4 = 3 \implies \text{is Choice } (\text{random}\_\text{process } s f n)\)” |
| “\(\text{cont } (\text{random}\_\text{process } s f n) = \text{random}\_\text{process } s f (f n)\)” of Skip |
| “\(\text{prefix } (\text{random}\_\text{process } s f n) = \text{shd } s\)” |
| “\(\text{cont } (\text{random}\_\text{process } s f n) = \text{random}\_\text{process } (\text{stl } s) f (f n)\)” of Action |
| “\(\text{left } (\text{random}\_\text{process } s f n) = \text{random}\_\text{process } (\text{every}\_\text{snd } s) f (f n)\)” |
| “\(\text{right } (\text{random}\_\text{process } s f n) = \text{random}\_\text{process } (\text{every}\_\text{snd } (\text{stl } s)) f (f n)\)” |

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

Here are more examples to conclude:

\[
\text{primcorec} \\
\text{even\_infty} :: \text{even\_enat and} \\
\text{odd\_infty} :: \text{odd\_enat} \\
\text{where} \\
\text{“even\_infty} \neq \text{Even\_EZero”} \\
\text{“un\_Even\_ESucc even\_infty} = \text{odd\_infty}” \\
\text{“un\_Odd\_ESucc odd\_infty} = \text{even\_infty}” \\
\text{primcorec iterate}_{ii} :: \text{“(’a } \Rightarrow \text{’a list) } \Rightarrow \text{’a } \Rightarrow \text{’a tree}_{ii” where} \\
\text{“lbl}_{ii} \text{ (iterate}_{ii} g x) = x” \\
\text{“sub}_{ii} \text{ (iterate}_{ii} g x) = \text{lmap } (\text{iterate}_{ii} g) (g x)”} \\
\]

5.2 Command Syntax

5.2.1 primcorec and primcorecursive

\[
\text{primcorec} : \text{local\_theory } \rightarrow \text{local\_theory} \\
\text{primcorecursive} : \text{local\_theory } \rightarrow \text{proof } (\text{prove}) \\
\]

\[
\begin{align*}
\text{primcorec} & \quad \text{primcorecursive} \\
\downarrow & \quad \downarrow \\
\text{prim-corec} & \quad \text{prim-corecursive} \\
\text{fixes} & \quad \text{target} \\
\downarrow & \quad \downarrow \\
\text{prim-corec} & \quad \text{prim-corecursive} \\
\text{per-options} & \quad \text{where} \\
\downarrow & \quad \downarrow \\
\text{fixes} & \quad \text{where} \\
\downarrow & \quad \downarrow \\
\text{per-options} & \quad \text{where} \\
\downarrow & \\
\text{1} & \\
\end{align*}
\]
The \texttt{primcorec} and \texttt{primcorecursive} commands introduce a set of mutually corecursive functions over codatatypes.

The syntactic entity \texttt{target} can be used to specify a local context, \texttt{fixes} denotes a list of names with optional type signatures, \texttt{thmdecl} denotes an optional name for the formula that follows, and \texttt{prop} denotes a HOL proposition [12].

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{sequential} option indicates that the conditions in specifications expressed using the constructor or destructor view are to be interpreted sequentially.
- The \texttt{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 \texttt{transfer} option indicates that an unconditional transfer rule should be generated and proved by \texttt{transfer_prover}. The \texttt{[transfer_rule]} attribute is set on the generated theorem.
The **primcorec** command is an abbreviation for **primcorecursive** with `by auto` to discharge any emerging proof obligations.

### 5.3 Generated Theorems

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

- **f.code [code]:**
  
  ```
  literate g x = LCons x (literate g (g x))
  ```

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

- **f.ctr:****
  
  ```
  literate g x = LCons x (literate g (g x))
  ```

- **f.disc [simp, code]:**
  
  ```
  ¬ lnull (literate g x)
  ```

  The [code] attribute is set by the code plugin (Section 8.1). The [simp] attribute is set only for functions for which `f.disc_iff` is not available.

- **f.disc_iff [simp]:**
  
  ```
  ¬ lnull (literate g x)
  ```

  This property is generated only for functions declared with the exhaustive option or whose conditions are trivially exhaustive.

- **f.sel [simp, code]:**
  
  ```
  ¬ lnull (literate g x)
  ```

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

- **f.exclude:**

  These properties are missing for `literate` because no exclusiveness proof obligations arose. In general, the properties correspond to the discharged proof obligations.

- **f.exhaust:**

  This property is missing for `literate` because no exhaustiveness proof obligation arose. In general, the property correspond to the discharged proof obligation.

- **f.coinduct [consumes m, case_names t_1 ... t_m, case_conclusion D_1 ... D_n]:**

  This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4).
6 Registering Bounded Natural Functors

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

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

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

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

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

6 Registering Bounded Natural Functors

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

6.1 Bounded Natural Functors

Bounded natural functors (BNFs) are a semantic criterion for where (co)recursion may appear on the right-hand side of an equation [4,11].

An \( n \)-ary BNF is a type constructor equipped with a map function (functorial action), \( n \) set functions (natural transformations), and an infinite cardinal bound that satisfy certain properties. For example, \( 'a \ llist \) is a unary BNF. Its predicator \( llist\_all :: ('a => bool) \Rightarrow 'a llist \Rightarrow bool \) extends unary predicates over elements to unary predicates over lazy lists. Similarly, its relator \( llist\_all2 :: ('a => 'b => bool) \Rightarrow 'a llist \Rightarrow 'b llist \Rightarrow bool \) 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.

```plaintext
typedef \(('d, 'a) fn = \textit{UNIV :: ('d => 'a) set}\) by \texttt{simp}

setup_lifting \texttt{type_definition_fn}

lift_definition \texttt{map_fn :: \( ('a => 'b) => ('d, 'a) fn => ('d, 'b) fn\) is \text{"op \circ\"}}.

lift_definition \texttt{set_fn :: \( ('d, 'a) fn => 'a set\) is \text{\textit{range}}.}

lift_definition \texttt{pred_fn :: \( ('a => bool) => ('d, 'a) fn => bool\) is \text{"\texttt{pred_fun (\_ \_. True)}\".}

lift_definition \texttt{rel_fn :: \( ('a => 'b => bool) => ('d, 'a) fn => ('d, 'b) fn => bool\) is \text{"\texttt{rel_fun (op =)}\".}

\textbf{bnf \( ('d, 'a) fn\)}

map: \texttt{map_fn}

sets: \texttt{set_fn}

bd: \texttt{natLeq + c | UNIV :: 'd set|}

rel: \texttt{rel_fn}

pred: \texttt{pred_fn}

proof –

\texttt{show \text{"map_fn id = id"}}
```


by transfer auto
next
  fix \(f :: "'a \Rightarrow 'b"\) and \(g :: "'b \Rightarrow 'c"\)
  show "map_fn (g \circ f) = map_fn g \circ map_fn f"
    by transfer (auto simp add: comp_def)
next
  fix \(F :: "'(d, 'a) fn"\) and \(f g :: "'a \Rightarrow 'b"\)
  assume "\(\forall x. x \in set_fn F \Longrightarrow f x = g x\)"
  then show "map_fn f F = map_fn g F"
    by transfer auto
next
  fix \(F :: "'(d, 'a) fn"\)
  have "\(|set_fn F| \leq o |UNIV :: 'd set|\)"
    by transfer (rule card_of_image)
  also have "\(?U \leq o natLeq + c ?U\)"
    by (rule ordLeq_csum2) (rule card_of_card_order)
  finally show "\(|set_fn F| \leq o natLeq + c |UNIV :: 'd set|\)".
next
  fix \(R :: "'a \Rightarrow 'b \Rightarrow bool"\) and \(S :: "'b \Rightarrow 'c \Rightarrow bool"\)
  show "rel_fn R OO rel_fn S \leq rel_fn (R OO S)"
    by (rule, transfer) (auto simp add: rel_fun_def)
next
  fix \(R :: "'a \Rightarrow 'b \Rightarrow bool"\)
  show "rel_fn R = (\(\lambda x y. \exists z. set_fn z \subseteq \{(x, y)\}. R x y\) \land map_fn fst z = x \land map_fn snd z = y)"
    unfolding fun_eq_iff relcompp.simps conversep.simps simps
    by transfer (force simp: rel_fun_def subset_iff)
next
  fix \(P :: "'a \Rightarrow bool"\)
Registering Bounded Natural Functors

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 ∈ {xs. xs ≠ []}"
  then show "map f xs ∈ {xs. xs ≠ []}"
    by (cases xs) auto
next
  fix zs :: "('a × 'b) list"
  assume "map fst zs ∈ {xs. xs ≠ []}" "map snd zs ∈ {xs. xs ≠ []}"
  then show "zs ∈ {xs. xs ≠ []}"
    by (cases zs) auto
qed

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

`bnf_axiomatization (setA: 'a, setB: 'b, setC: 'c) F
  [wits: "'a ⇒ ('a, 'b, 'c) F"]`

`print_theorems`
`print_bnfs`

6.3 Command Syntax

6.3.1 bnf

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

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

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

\subsection{lift\_bnf}

\begin{verbatim}
   lift\_bnf : local\_theory → proof(prove)
\end{verbatim}
The `lift_bnf` command registers as a BNF an existing type (the `abstract type`) that was defined as a subtype of a BNF (the `raw type`) using the `typedef` command. 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 `no_warn_wits` option.

### 6.3.3 copy_bnf

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

The \text{copy\_bnf} command performs the same lifting as \text{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 bnf_axiomatization

\[
\text{bnf\_axiomatization} : \text{local\_theory} \rightarrow \text{local\_theory}
\]
The `bnf_axiomatization` command declares a new type and associated constants (map, set, predicator, 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

\[\text{print\_bnfs} : \text{local\_theory} \rightarrow\]

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

\[\text{free\_constructors} : \text{local\_theory} \rightarrow \text{proof(prove)}\]
The `freectors` 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,

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

translates $lapp\;xs\;ys = (\text{case } xs \text{ of } LNil \Rightarrow ys | LCons\;x\;xs' \Rightarrow LCons\;x\;(lapp\;xs'\;ys))$ into

\[
lapp\;LNil\;ys = ys \\
lapp\;(LCons\;xa\;x)\;ys = LCons\;xa\;(lapp\;x\;ys)
\]

### 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 a single equation with a complex case expression on the right-hand side (cf. `simps_of_case`). For example,

```
case_of_simps: lapp_case: lapp_simps
```

translates

\[
lapp\;LNil\;ys = ys \\
lapp\;(LCons\;xa\;x)\;ys = LCons\;xa\;(lapp\;x\;ys)
\]

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

```plaintext
datatype (plugins del: code "quickcheck") color = Red | Black
```

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

8.1 Code Generator

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

For types, the plugin derives the following properties:

- **t.eq.refl [code nbe]:**
  
  ```plaintext
type.equal_class.equal x x ≡ True
```

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

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

8.2 Size

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

The plugin derives the following properties:

\[
\begin{align*}
t.\text{size} \ [\text{simp, code}] : \\
size\_\text{list} \ s \ [] & = 0 \\
size\_\text{list} \ s \ (x \# s_2) & = x \cdot s \cdot s_2 + \text{Suc} \ 0 \\
s\ [] & = 0 \\
s \ (x \# s_2) & = x \cdot s + \text{Suc} \ 0 \\
t.\text{size\_gen} : \\
size\_\text{list} \ s \ [] & = 0 \\
size\_\text{list} \ s \ (x \# s_2) & = x \cdot s \cdot s_2 + \text{Suc} \ 0 \\
t.\text{size\_gen\_o\_map} : \\
size\_\text{list} \ f \circ \text{map} \ g & = size\_\text{list} \ (f \circ g) \\
t.\text{size\_neq} : \\
\text{This property is missing for } 'a \text{ list}. \text{ If the } size \text{ function always evaluates to a non-zero value, this theorem has the form } size \ s \neq 0.
\end{align*}
\]

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 \texttt{BNF\_LFP\_Size.register\_size} or \texttt{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:

\[
\begin{align*}
t.\text{Domainp\_rel} \ [\text{relator\_domain}] : \\
\text{Domainp} \ (\text{list\_all} 2 \ R) & = \text{list\_all} \ (\text{Domainp} \ R) \\
t.\text{left\_total\_rel} \ [\text{transfer\_rule}] : \\
\text{left\_total} \ R & \Rightarrow \text{left\_total} \ (\text{list\_all} 2 \ R)
\end{align*}
\]
8.4 Lifting

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

The plugin derives the following property:

\[ \text{t.Quotient [quot_map]:} \quad \text{Quotient} \; R \; \text{Abs Rep} \; T \implies \text{Quotient} \; (\text{list_all2} \; R) \; \text{(map Abs)} \; \text{(map Rep)} \; (\text{list_all2} \; T) \]

In addition, the plugin sets the [relator_eq] attribute on a variant of the \( t.rel_eq_onp \) property, the [relator_mono] attribute on \( t.rel_mono \), and the [relator_distr] attribute on \( t.rel_compp \).

8.5 Quickcheck

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

- quickcheck_random
- quickcheck_exhaustive
- quickcheck_bounded_forall
- quickcheck_full_exhaustive
- quickcheck_narrowing

8.6 Program Extraction

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

9 Known Bugs and Limitations

This section lists the known bugs and limitations of the (co)datatype package at the time of this writing.

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

2. *Locally fixed types and terms cannot be used in type specifications.* The limitation on types can be circumvented by adding type arguments to the local (co)datatypes to abstract over the locally fixed types.

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

4. *The primcorec command does not allow corecursion under case–of for datatypes that are defined without discriminators and selectors.*

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

6. *There is no way to register the same type as both a datatype and a codatatype.* This affects types such as the extended natural numbers, for which both views would make sense (for a different set of constructors).
7. The names of variables are often suboptimal in the properties generated by the package.

8. The compatibility layer sometimes produces induction principles with a slightly different ordering of the premises than the old package.

Acknowledgment

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

References


REFERENCES


