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

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

25 October 2022

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

Contents
1 Introduction
2 Defining Datatypes
   2.1 Introductory Examples
   2.1.1 Nonrecursive Types
   2.1.2 Simple Recursion
   2.1.3 Mutual Recursion
   2.1.4 Nested Recursion
   2.1.5 Auxiliary Constants
   2.2 Command Syntax
   2.2.1 datatype
   2.2.2 datatype_compat
   2.3 Generated Constants
   2.4 Generated Theorems
      2.4.1 Free Constructor Theorems
2.4.2 Functorial Theorems ........................................... 16
2.4.3 Inductive Theorems ........................................... 21
2.5 Proof Method ..................................................... 22
  2.5.1 countable_datatype ......................................... 22
2.6 Antiquotation ................................................... 22
  2.6.1 datatype ....................................................... 22
2.7 Compatibility Issues ........................................... 22

3 Defining Primitively Recursive Functions ................................ 24
  3.1 Introductory Examples .......................................... 24
    3.1.1 Nonrecursive Types ........................................ 24
    3.1.2 Simple Recursion ......................................... 25
    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 ......................... 30
  3.5 Compatibility Issues .......................................... 31

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 ...................................... 33
  4.2 Command Syntax ................................................ 33
    4.2.1 codatatype .................................................. 33
  4.3 Generated Constants ........................................... 33
  4.4 Generated Theorems ............................................ 33
    4.4.1 Coinductive Theorems ................................. 34
  4.5 Antiquotation .................................................. 35
    4.5.1 codatatype .................................................. 35

5 Defining Primitively Corecursive Functions ............................ 36
  5.1 Introductory Examples .......................................... 37
    5.1.1 Simple Corecursion ...................................... 37
    5.1.2 Mutual Corecursion ...................................... 38
    5.1.3 Nested Corecursion ...................................... 39
    5.1.4 Nested-as-Mutual Corecursion ......................... 40
    5.1.5 Constructor View ........................................ 41
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 tree = Node {lbl: 'a} (subfs: "'a tree fset")
\end{verbatim}

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

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

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

In addition to inductive datatypes, the package supports coinductive datatypes, or codatatypes, which allow infinite values. For example, the following command introduces the type of lazy lists, which comprises both finite and infinite values:

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

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

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

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

The package is part of Main. Additional functionality is provided by the theory ~/.src/HOL/Library/BNF_Axiomatization.thy.

The package, like its predecessor, fully adheres to the LCF philosophy [5]: The characteristic theorems associated with the specified (co)datatypes are derived rather than introduced axiomatically.1 The package is described in a number of scientific papers [2, 4, 9, 11]. The central notion is that of a bounded natural functor (BNF)—a well-behaved type constructor for which nested (co)recursion is supported.

This tutorial is organized as follows:

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

1However, some of the internal constructions and most of the internal proof obligations are omitted if the quick_and_dirty option is enabled.
• Section 4, “Defining Codatatypes,” describes how to specify codatatypes using the codatatype command.

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

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

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

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

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

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

2 Defining Datatypes

Datatypes can be specified using the datatype command.

2.1 Introductory Examples

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

2.1.1 Nonrecursive Types

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

    datatype trool = True | False | 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 HOL.

```
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 DEFINING DATATYPES

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:

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

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

```
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 ⇒:

```
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:

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

And so is the definition of hereditarily finite sets:

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

In general, type constructors (′a1, . . . , ′am) t allow recursion on a subset of their type arguments ′a1, . . . , ′am. 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:

```
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:

```isar
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:

\[
\text{null } xs \iff xs = \text{Nil} \quad \neg \text{null } xs \iff \text{Cons } (\text{hd } xs) \ (\text{tl } xs) = xs
\]

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

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 \(\lambda xs. xs = \text{Nil}\) 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

```plaintext
datatype ('a, 'b) prod (infixr "\,*\) 20) = Pair 'a 'b

datatype (set: 'a) list =
  null: Nil ("[]")
| Cons (hd: 'a) (tl: "'a list") (infixr "\#\) 65)
for
  map: map
  rel: list_all2
  pred: list_all
```

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

```plaintext
syntax "._list" :: "args \Rightarrow 'a list" ("[(_)]")
translations
"[x, xs]\) \Rightarrow "x \# [xs]"
"[x]\) \Rightarrow "x \# []"
```

2.2 Command Syntax

2.2.1 datatype

```plaintext
datatype : local_theory \rightarrow local_theory
```

```
datatype
    target
    dt-options
```

dt-options

```
( plugins
    discs_sels
, )
```

plugins

```
plugins
    only
    del
    : name
```

The **datatype** command introduces a set of mutually recursive datatypes specified by their constructors.

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

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

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

The optional **where** clause specifies default values for selectors. Each proposition must be an equation of the form `un_D (C . . .) = . . .`, where *C* is a constructor and `un_D` is a selector.
The left-hand sides of the datatype equations specify the name of the type to define, its type parameters, and additional information:

\[
\text{dt-name}
\]

\[
\text{tyargs}
\]

\[
\text{mixfix}
\]

\[
\text{typefree}
\]

\[
\text{name}
\]

\[
\text{dead}
\]

\[
\text{name}
\]

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

The optional names preceding the type variables allow to override the default names of the set functions (\(\text{set}_1t, \ldots, \text{set}_mt\)). Type arguments can be marked as dead by entering \texttt{dead} in front of the type variable (e.g., \(\text{dead} \text{\`a}\)); 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.

\[
\text{dt-ctor}
\]

\[
\text{name}
\]

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

\[
\text{mixfix}
\]
The main constituents of a constructor specification are the name of the constructor and the list of its argument types. An optional discriminator name can be supplied at the front. If discriminators are enabled (cf. the discs_sels option) but no name is supplied, the default is $\lambda x. x = C_j$ for nullary constructors and $t.is_C_j$ otherwise.

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

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

2.2.2 datatype_compat

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

```
datatype_compat even_nat odd_nat
```

```
ML (Old_Datatype_Data.get_info theory type_name (even_nat))
```

The syntactic entity name denotes an identifier [12].

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

A few remarks concern nested recursive datatypes:
• The old-style, nested-as-mutual induction rule and recursor theorems are generated under their usual names but with “compat_” prefixed (e.g., compat_tree.induct, compat_tree.inducts, and compat_tree.rec). These theorems should be identical to the ones generated by the old datatype package, up to the order of the premises—meaning that the subgoals generated by the induct or induction method may be in a different order than before.

• All types through which recursion takes place must be new-style datatypes or the function type.

2.3 Generated Constants

Given a datatype $(\alpha_1, \ldots, \alpha_m)$ $t$ with $m$ live type variables and $n$ constructors $t.C_1, \ldots, t.C_n$, the following auxiliary constants are introduced:

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

The discriminators and selectors are generated only if the discs_sels option is enabled or if names are specified for discriminators or selectors. The set functions, map function, predicator, and relator are generated only if $m > 0$.

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

2.4 Generated Theorems

The characteristic theorems generated by 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

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

### 2.4.1 Free Constructor Theorems

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

- **t.inject** [iff, induct_simp]:
  
  \[(x_1 \# x_2 = y_1 \# y_2) = (x_1 = y_1 \land x_2 = y_2)\]

- **t.distinct** [simp, induct_simp]:
  
  \[\begin{align*}
  & \emptyset \neq x_1 \# x_2 \\
  & x_1 \# x_2 \neq \emptyset
  \end{align*}\]

- **t.exhaust** [cases t, case_names C₁ ... Cₙ]:
  
  \[\begin{align*}
  & [y = \emptyset \implies P; \land x_21 \ x_22 \ y = x_1 \# x_22 \implies P] \implies P
  \end{align*}\]

- **t.nchotomy:**
  
  \[\forall \text{list}. \ \text{list} = \emptyset \lor (\exists x_21 \ x_22. \ \text{list} = x_1 \# x_22)\]

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

- **t.distinct** [THEN notE, elim!]:
  
  \[\begin{align*}
  & \emptyset = x_1 \# x_22 \implies R \\
  & x_1 \# x_22 = \emptyset \implies R
  \end{align*}\]

The next subgroup is concerned with the case combinator:

- **t.case** [simp, code]:
  
  \[\begin{align*}
  & (\text{case } [\ ] \ \text{of } [\ ] \Rightarrow f_1 \mid x \# x a \Rightarrow f_2 \ x x a) = f_1 \\
  & (\text{case } x_21 \# x_22 \ \text{of } [\ ] \Rightarrow f_1 \mid x \# x a \Rightarrow f_2 \ x x a) = f_2 \ x_21 \ x_22
  \end{align*}\]

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

\[ \text{case\_cong [fundef\_cong]:} \]
\[ \begin{align*}
\text{null list \Rightarrow} & f1 = g1; \\land x21 \ x22. \text{list'} = x21 \ # x22 \Rightarrow \\
& f2 \ x21 \ x22 = g2 \ x21 \ x22 \Rightarrow \text{(case list of [] \Rightarrow} f1 \ | \ x21 \ # x22 \Rightarrow \\
& f2 \ x21 \ x22) = \text{(case list' of [] \Rightarrow} g1 \ | \ x21 \ # x22 \Rightarrow g2 \ x21 \ x22) \\
\end{align*} \]

\[ \text{case\_cong\_weak [cong]:} \]
\[ \begin{align*}
\text{null list \Rightarrow} & (\text{case list of [] \Rightarrow} f1 \ | \ x \ # xa \Rightarrow f2 \ x xa) = (\text{case list' of [] \Rightarrow} f1 \ | \ x \ # xa \Rightarrow f2 \ x xa) \\
\end{align*} \]

\[ \text{case\_distrib:} \]
\[ h (\text{case list of [] \Rightarrow} f1 \ | \ x \ # xa \Rightarrow f2 \ x xa) = (\text{case list of [] \Rightarrow} h \\
& f1 \ | \ x1 \ # x2 \Rightarrow h (f2 \ x1 \ x2)) \]

\[ \text{split:} \]
\[ P (\text{case list of [] \Rightarrow} f1 \ | \ x \ # xa \Rightarrow f2 \ x xa) = (\text{(list = [] \Rightarrow} P f1) \\
& \land (\forall x21 \ x22. \text{list = x21 \ # x22 \Rightarrow} P (f2 \ x21 \ x22))) \]

\[ \text{split\_asm:} \]
\[ P (\text{case list of [] \Rightarrow} f1 \ | \ x \ # xa \Rightarrow f2 \ x xa) = (\neg (\text{list = [] \land} \neg P \\
& f1 \lor (\exists x21 \ x22. \text{list = x21 \ # x22 \land} \neg P (f2 \ x21 \ x22)))) \]

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

The third subgroup revolves around discriminators and selectors:

\[ \text{disc [simp]:} \]
\[ \text{null [] \Rightarrow} \neg \text{null (x21 \ # x22)} \]

\[ \text{discI:} \]
\[ \text{null list \Rightarrow} \text{null list} \]
\[ \text{null list \Rightarrow} \neg \text{null list} \]

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

\[ \text{collapse [simp]:} \]
\[ \text{null list \Rightarrow} \text{list = []} \]
\[ \neg \text{null list \Rightarrow} \text{hd list \ # tl list = list} \]
The [simp] attribute is exceptionally omitted for datatypes equipped with a single nullary constructor, because a property of the form \( x = C \) is not suitable as a simplification rule.

\[ \text{distinct\_disc [dest]:} \]
These properties are missing for 'a list because there is only one
proper discriminator. If the datatype had been introduced with a second discriminator called nonnull, they would have read as follows:

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

t.exhaust_disc [case_names C_1 \ldots C_n]:
\[ [\text{null list} \implies P; \neg \text{nonnull list} \implies P] \implies P \]

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

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

t.split_sel:
\[ P (\text{case list of } [] \Rightarrow f_1 | x \# x a \Rightarrow f_2 x xa) = ((\text{list} = [] \implies P f_1) \\
\land (\text{list} = \text{hd list} \# \text{tl list} \implies P (f_2 (\text{hd list}) (\text{tl list})))) \]

t.split_sel_asm:
\[ P (\text{case list of } [] \Rightarrow f_1 | x \# x a \Rightarrow f_2 x xa) = (\neg (\text{list} = []) \land \neg P \\
f_1 \lor \text{list} = \text{hd list} \# \text{tl list} \land \neg P (f_2 (\text{hd list}) (\text{tl list}))) \]

t.split_sels = split_sel split_sel_asm

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

t.disc_eq_case:
\[ \text{null list} = (\text{case list of } [] \Rightarrow \text{True} | u_\_ \# uua_\_ \Rightarrow \text{False}) \\
(\neg \text{null list}) = (\text{case list of } [] \Rightarrow \text{False} | u_\_ \# uua_\_ \Rightarrow \text{True}) \]

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:

\[ t.\text{case_transfer} [\text{transfer_rule}]: \]
\[ \text{rel}_\text{fun} S (\text{rel}_\text{fun} (\text{rel}_\text{fun} R (\text{rel}_\text{fun} (\text{list all2} R) S)) (\text{rel}_\text{fun} (\text{list all2} R) S)) \text{ case list case list} \]

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

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

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

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

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

\[ t.\text{set\_cases} \ [\text{consumes 1, cases set: } set\_t]\ : \]
\[ \begin{align*}
  \forall e \in \text{set a}; \forall z2. a = e \ # z2 \Rightarrow \text{thesis} &; \forall z1 z2. [a = z1 \ # z2; e \in \text{set z2}] \Rightarrow \text{thesis} \\
\end{align*} \]

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

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

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

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

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

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

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

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

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

\[\text{t.rel\_cases [consumes 1, case_names t\_1 \ldots t\_m, cases pred]:}\]
\[
\text{list\_all2 } R \ a \ b \implies \text{thesis}; \ \land x1 \ x2 \ y1 \ y2. \ [a = x1 \ # x2; \ b = y1 \ # y2; \ R \ x1 \ y1; \ \text{list\_all2 } R \ x2 \ y2] \implies \text{thesis} \implies \text{thesis}
\]

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

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

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

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

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

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

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

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

t.map_id0:
map id = id 

t.map_id:
map id t = t 

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

t.map_ident_strong:
(\land z. z \in \text{set} t \implies f z = z) \implies \text{map} f t = t 

t.map_transfer [transfer_rule]:
rel_fun (rel_fun Rb Sd) (rel_fun (list_all2 Rb) (list_all2 Sd)) map 
map 
The [transfer_rule] attribute is set by the transfer plugin (Section 8.3) for type constructors with no dead type arguments. 

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

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

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

t.pred_mono [mono]:
P \leq Pa \implies \text{list_all} P \leq \text{list_all} Pa 

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

t.pred_rel:
\text{list_all} P x = \text{list_all2} (eq_onp P) x x 

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

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

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

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

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

t.rel_conversep:
   \texttt{list_all2 R} ^ = (\texttt{list_all2 R}) ^

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

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

t.rel_flip:
   \texttt{list_all2 R} ^ a b = \texttt{list_all2 R} b a

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

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

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

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

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

t.rel_refl:
   \[(\forall x. Ra x x) \implies \texttt{list_all2 R} x x\]
2 DEFINING DATATYPES

\[ t.\text{rel}_\text{refl\_strong}: \]
\[ (\forall z. z \in \text{set} \ x \implies Ra z z) \implies \text{list}_\text{all}2 Ra x x \]

\[ t.\text{rel}_\text{refl\_p}: \]
\[ \text{reflp} \ R \implies \text{reflp} (\text{list}_\text{all}2 R) \]

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

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

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

2.4.3 Inductive Theorems

The inductive theorems are as follows:

\[ t.\text{induct} [\text{case\_names} \ C_1 \ldots C_n, \text{induct} t]: \]
\[ [P []; \\forall x1 x2. P x2 \implies P (x1 \# x2)] \implies P \text{list} \]

\[ t.\text{rel\_induct} [\text{case\_names} \ C_1 \ldots C_n, \text{induct pred}]: \]
\[ [\text{list}_\text{all}2 R x y; Q []; \\forall a21 a22 b21 b22. [R a21 b21; Q a22 b22] \implies Q (a21 \# a22) (b21 \# b22)] \implies Q x y \]

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

\[ t.\text{rec} [\text{simp, code}]: \]
\[ \text{rec\_list} f1 f2 [] = f1 \]
\[ \text{rec\_list} f1 f2 (x21 \# x22) = f2 x21 x22 (\text{rec\_list} f1 f2 x22) \]
The [code] attribute is set by the code plugin (Section 8.1).

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

\[ t.\text{rec\_transfer} [\text{transfer\_rule}]: \]
\[ \text{rel}_\text{fun} S (\text{rel}_\text{fun} (\text{rel}_\text{fun} R (\text{rel}_\text{fun} (\text{list}_\text{all}2 R) (\text{rel}_\text{fun} S S)))) (\text{rel}_\text{fun} (\text{list}_\text{all}2 R) S)) \text{rec\_list} \text{rec\_list} \]
The [transfer\_rule] attribute is set by the transfer plugin (Section 8.3) for type constructors with no dead type arguments.
For convenience, `datatype` also provides the following collection:

\[
\begin{align*}
t . \text{simps} &= t . \text{inject} \ t . \text{distinct} \ t . \text{case} \ t . \text{rec} \ t . \text{map} \ t . \text{rel}\_\text{inject} \\
&\quad \ t . \text{rel}\_\text{distinct} \ t . \text{set}
\end{align*}
\]

### 2.5 Proof Method

#### 2.5.1 `countable_datatype`

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

```isar
instance list :: (countable) countable 
by countable_datatype
```

### 2.6 Antiquotation

#### 2.6.1 `datatype`

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

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

### 2.7 Compatibility Issues

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

- **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 \texttt{primrec}. Recursion through functions was handled specially. In the new package, nested recursion (for functions and non-functions) is handled in a more modular fashion. The old-style recursor can be generated on demand using \texttt{primrec} if the recursion is via new-style datatypes, as explained in Section 3.1.5, or using \texttt{datatype_compat}.

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

- \textbf{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 \texttt{size} in terms of the generic function \texttt{t.size_t}. Moreover, the new function considers nested occurrences of a value, in the nested recursive case. The old behavior can be obtained by disabling the \texttt{size} plugin (Section 8) and instantiating the \texttt{size} type class manually.

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

- \textbf{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_ti} 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}.

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

- \textbf{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.
3 Defining Primitively Recursive Functions

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

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

3.1 Introductory Examples

Primitive recursion is illustrated through concrete examples based on the datatypes defined in Section 2.1. More examples can be found in the directory \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
  "bool_of_trool False ←→ False"
| "bool_of_trool Truue ←→ True"

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

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

primrec mirrror :: "('a, 'b, 'c) triple ⇒ ('c, 'b, 'a) triple" where
  "mirrror (Triple a b c) = Triple c b a"
\end{verbatim}

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:

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

```ml
primrec (nonexhaustive) at :: "'a list ⇒ nat ⇒ 'a" where
  "at (x # xs) j = (case j of
    Zero ⇒ x
  | Succ j' ⇒ at xs j')"
```

```ml
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`.

```ml
simps_of_case at_simps_alt: at.simps
```

This generates the lemma collection `at.simps_alt`:

```ml
at (x # xs) Zero = x      at (xa # xs) (Succ x) = at xs x
```

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

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

3.1.3 Mutual Recursion

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

```ml
primrec
  nat_of_even_nat :: "even_nat ⇒ nat" and
  nat_of_odd_nat :: "odd_nat ⇒ nat"
where
  "nat_of_even_nat Even_Zero = Zero"
| "nat_of_even_nat (Even_Succ n) = Succ (nat_of_odd_nat n)"
| "nat_of_odd_nat Zero = Succ Zero"
| "nat_of_odd_nat (Succ n) = nat_of_even_nat n"
```
3. DEFINING PRIMITIVELY RECURSIVE FUNCTIONS

| “nat_of_odd_n (Odd_Succ n) = Succ (nat_of_even_n 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:

primrec atff :: "'a treeff ⇒ nat list ⇒ 'a" where
“atff (Nodeff a ts) js =
(case js of
  [] ⇒ a
  | j # js’ ⇒ atff (map atff ts) js’)”

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:

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 \((\Rightarrow)\) is simply composition \((\circ)\):

\[
\text{primrec relabel_ft} :: \left\langle \left\langle (\Rightarrow a) \Rightarrow (\Rightarrow a ftree) \Rightarrow (\Rightarrow a ftree) \right\rangle \right\rangle \quad \text{where} \\
\text{relabel_ft} f \ (\text{FTLeaf} x) = \text{FTLeaf} \ (f \ x) \\
| \text{relabel_ft} f \ (\text{FTNode} g) = \text{FTNode} \ (\text{relabel_ft} f \circ g)
\]

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

\[
\text{primrec relabel_ft} :: \left\langle \left\langle (\Rightarrow a) \Rightarrow (\Rightarrow a ftree) \Rightarrow (\Rightarrow a ftree) \right\rangle \right\rangle \quad \text{where} \\
\text{relabel_ft} f \ (\text{FTLeaf} x) = \text{FTLeaf} \ (f \ x) \\
| \text{relabel_ft} f \ (\text{FTNode} g) = \text{FTNode} \ \left(\lambda x. \text{relabel_ft} f \ (g \ x)\right)
\]

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

\[
\text{datatype } \Rightarrow a ftree2 = \text{FTLeaf2} \Rightarrow a \mid \text{FTNode2} \Rightarrow (\Rightarrow a ftree2) \Rightarrow (\Rightarrow a ftree2) \\
\text{primrec relabel_ft2} :: \left\langle \left\langle (\Rightarrow a) \Rightarrow (\Rightarrow a ftree2) \Rightarrow (\Rightarrow a ftree2) \right\rangle \right\rangle \quad \text{where} \\
\text{relabel_ft2} f \ (\text{FTLeaf2} x) = \text{FTLeaf2} \ (f \ x) \\
| \text{relabel_ft2} f \ (\text{FTNode2} g) = \text{FTNode2} \ ((\circ) \ ((\circ) \ (\text{relabel_ft2} f)) g)
\]

\[
\text{primrec relabel_ft2} :: \left\langle \left\langle (\Rightarrow a) \Rightarrow (\Rightarrow a ftree2) \Rightarrow (\Rightarrow a ftree2) \right\rangle \right\rangle \quad \text{where} \\
\text{relabel_ft2} f \ (\text{FTLeaf2} x) = \text{FTLeaf2} \ (f \ x) \\
| \text{relabel_ft2} f \ (\text{FTNode2} g) = \text{FTNode2} \ (\lambda x y. \text{relabel_ft2} f \ (g \ x \ y))
\]

\[
\text{primrec (nonexhaustive) subtree_ft2} :: \left\langle (\Rightarrow a) \Rightarrow (\Rightarrow a ftree2) \Rightarrow (\Rightarrow a ftree2) \right\rangle \quad \text{where} \\
\text{subtree_ft2} \ x \ y \ (\text{FTNode2} g) = g \ x \ y
\]

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

\[
\text{primrec increasing_tree} :: \left\langle \text{int} \Rightarrow \text{int treeff} \Rightarrow \text{bool} \right\rangle \quad \text{where} \\
\text{increasing_tree} m \ (\text{Nodeff} n \ ts) \iff n \geq m \land \text{list_all} \ (\text{increasing_tree} \ (n + 1)) \ ts
\]

### 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 \( \text{(nonexhaustive)} \)
\[ \begin{align*}
\text{at} f f & \colon \text{"a tree} f f \Rightarrow \text{nat list} \Rightarrow \text{'}a" \text{ and} \\
\text{ats} f f & \colon \text{"a tree} f f \text{ list} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{'}a" \\
\end{align*} \]

where
\[ \begin{align*}
\text{"at} f f (\text{Node} f f a ts) js & = \\
(\text{case } js \text{ of} & ) \\
\text{[]} & \Rightarrow a \\
| j \neq js' & \Rightarrow \text{ats} f f ts j js' \\
\end{align*} \]
\[ \begin{align*}
\text{"ats} f f (t \# ts) j & = \\
(\text{case } j \text{ of} & ) \\
\text{Zero} & \Rightarrow \text{at} f f t \\
| \text{Succ } j' & \Rightarrow \text{ats} f f ts j' \\
\end{align*} \]

Appropriate induction rules are generated as \text{at}_f f \text{.induct}, \text{ats}_f f \text{.induct}, and \text{at}_f f \_\text{ats}_f f \text{.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
\[ \begin{align*}
\text{sum_btree} & \colon \text{"(a::\{\text{zero, plus}\}) btree} \Rightarrow \text{'}a" \text{ and} \\
\text{sum_btree_option} & \colon \text{"a btree option} \Rightarrow \text{'}a" \\
\end{align*} \]

where
\[ \begin{align*}
\text{"sum_btree} (\text{BNode } a \text{ lt rt}) & = \\
\text{a} + \text{sum_btree_option } \text{lt} + \text{sum_btree_option } \text{rt} \\
\end{align*} \]
\[ \begin{align*}
\text{"sum_btree_option } \text{None} & = 0 \\
\text{"sum_btree_option } (\text{Some } t) & = \text{sum_btree } t \\
\end{align*} \]

3.2 Command Syntax

3.2.1 primrec

\text{primrec} : \text{local_theory} \rightarrow \text{local_theory}
The `primrec` command introduces a set of mutually recursive functions over datatypes.

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

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

- The `plugins` option indicates which plugins should be enabled (only) or disabled (del). By default, all plugins are enabled.
- The `nonexhaustive` option indicates that the functions are not necessarily specified for all constructors. It can be used to suppress the warning that is normally emitted when some constructors are missing.
- The `transfer` option indicates that an unconditional transfer rule should be generated and proved by `transfer_prover`. The `[transfer_rule]` attribute is set on the generated theorem.

### 3.3 Generated Theorems

The `primrec` command generates the following properties (listed for `tfold`):
3 DEFINING PRIMITIVELY RECURSIVE FUNCTIONS

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

\[ f \text{.} \text{function} \{ \text{transfer}_\text{rule} \} : \]
\[ \text{rel\_fun} \ (\text{rel\_fun } R2 \ (\text{rel\_fun } R1 R1)) \ (\text{rel\_fun} \ (\text{rel\_tlist } R2 R1) \ R1) \ \text{tfold} \ \text{tfold} \]
This theorem is generated by the \text{transfer} plugin (Section 8.3) for functions declared with the \text{transfer} option enabled.

\[ f \text{.} \text{induct} \ [\text{case\_names } C_1 \ldots C_n] : \]
This induction rule is generated for nested-as-mutual recursive functions (Section 3.1.5).

\[ f_1 \ldots f_m \text{.} \text{induct} \ [\text{case\_names } C_1 \ldots C_n] : \]
This induction rule is generated for nested-as-mutual recursive functions (Section 3.1.5). Given \( m > 1 \) mutually recursive functions, this rule can be used to prove \( m \) properties simultaneously.

3.4 Recursive Default Values for Selectors

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

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

1. Introduce a fully unspecified constant \( \text{un}_D_0 :: \texttt{'}a \) 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:

\texttt{consts} \( \text{termi}_0 :: \texttt{'}a \)
4 DEFINING CODATATYPES

```ml
datatype ('a, 'b) tlist =
  TNil (termi: 'b)
| TCons (thd: 'a) (ttl: ('a, 'b) tlist)
where
  "ttl (TNil y) = TNil y"
| "termi (TCons _ xs) = termi0 xs"
overloading
  termi0 ≡ "termi0 :: ('a, 'b) tlist ⇒ 'b"
begin
primrec termi0 :: ('a, 'b) tlist ⇒ 'b where
  "termi0 (TNil y) = y"
| "termi0 (TCons x xs) = termi0 xs"
end
lemma termi_TCons[simp]: "termi (TCons x xs) = termi xs"
  by (cases xs) auto
```

3.5 Compatibility Issues

The command `primrec`'s behavior on new-style datatypes has been designed
to be highly compatible with that for old, pre-Isabelle2015 datatypes, to ease
migration. There is nonetheless at least one incompatibility that may arise
when porting to the new package:

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

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
4 DEFINING CODATATYPES

syntax as recursive datatypes, except for the command name. For example, here is the definition of lazy lists:

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

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

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

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

```ml
codatatype enat = EZero | ESucc enat
```

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

Here is an example with many constructors:

```ml
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:

```ml
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:

\begin{verbatim}
  codatatype 'a tree_i.i = Node_i.i (lbl_i:i: 'a) (sub_i.i: "'a tree_i.i list")
  codatatype 'a tree_i.s = Node_i.s (lbl_i.s: 'a) (sub_i.s: "'a tree_i.s fset")
  codatatype 'a sm = SM (accept: bool) (trans: "'a ⇒ 'a sm")
\end{verbatim}

4.2 Command Syntax

4.2.1 codatatype

\begin{verbatim}
codatatype : local_theory → local_theory
\end{verbatim}

Definitions of codatatypes have almost exactly the same syntax as for data-
types (Section 2.2). The discrs_sels option is superfluous because discrimi-
nators and selectors are always generated for codatatypes.

4.3 Generated Constants

Given a codatatype ('a_1, ..., 'a_m) t with m > 0 live type variables and n
constructors t.C_1, ..., 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 codatatype are grouped in three
broad categories:

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

The coinductive theorems (Section 4.4.1) are properties of datatypes related to their coinductive nature.

The first two categories are exactly as for datatypes.

4.4.1 Coinductive Theorems

The coinductive theorems are listed below for 'a list:

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

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

\[
\begin{align*}
t.\text{rel\_coinduct} & \quad [\text{consumes } m, \text{ case_names } t_1 \ldots t_m, \\
& \quad \text{case_conclusion } D_1 \ldots D_n, \text{ coinduct pred}] : \\
& \quad [P \ x y; \land \text{llist llist'} \implies \text{lnull llist} = \text{lnull llist'} \land \\
& \quad (\neg \text{lnull llist} \implies \neg \text{lnull llist'}) \implies \text{lhd llist} = \text{lhd llist'} \land (\text{lll llist'})] \implies \text{llist\_all2 R x y} \\
\end{align*}
\]

\[
\begin{align*}
t_{1\ldots m}.\text{coinduct} & \quad [\text{case_names } t_1 \ldots t_m, \text{ case_conclusion } D_1 \ldots D_n] \\
\end{align*}
\]

\[
\begin{align*}
t_{1\ldots m}.\text{coinduct\_strong} & \quad [\text{case_names } t_1 \ldots t_m, \\
& \quad \text{case_conclusion } D_1 \ldots D_n] : \\
\end{align*}
\]

\[
\begin{align*}
t_{1\ldots m}.\text{rel\_coinduct} & \quad [\text{case_names } t_1 \ldots t_m, \\
& \quad \text{case_conclusion } D_1 \ldots D_n] : \\
\end{align*}
\]

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

\[
\begin{align*}
t_{1\ldots m}.\text{set\_induct} & \quad [\text{case_names } C_1 \ldots C_n, \\
& \quad \text{induct set: } \text{set}_{j\ldots t_1}, \ldots, \text{induct set: } \text{set}_{j\ldots t_m}] : \\
& \quad [x \in \text{lset } a; \land z_1 z_2. P \ z_1 (\text{LCons } z_1 z_2); \land z_1 z_2 xa. [xa \in \text{lset } z2; \\
& \quad P \ xa z2] \implies P \ xa (\text{LCons } z1 z2)] \implies P \ x a \\
\end{align*}
\]

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

4.5 Antiquotation

4.5.1 codatatype

The codatatype antiquotation, written `\<\<\texttt{codatatype}\>\{t\}` or `@{codatatype \{t\}}`, where `t` is a type name, expands to \LaTeX code for the definition of
the codatatype, with each constructor listed with its argument types. For example, if $t$ is $l\text{list}$:

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

5 Defining Primitively Corecursive Functions

Corecursive functions can be specified using the \texttt{primcorec} and \texttt{primcorecursive} commands, which support primitive corecursion. Other approaches include the more general \texttt{partial\_function} command, the \texttt{corec} and \texttt{corecursive} commands, and techniques based on domains and topologies [8]. In this tutorial, the focus is on \texttt{primcorec} and \texttt{primcorecursive}; \texttt{corec} and \texttt{corecursive} are described in a separate tutorial [3]. More examples can be found in the directories \texttt{~/src/HOL/Datatype\_Examples} and \texttt{~/src/HOL/Corec\_Examples}.

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

- The \textit{destructor view} specifies $f$ by implications of the form
  \[
  \ldots \implies \text{is}_C j (f \, x_1 \ldots \, x_n)
  \]
  and equations of the form
  \[
  \text{un}_C j i (f \, x_1 \ldots \, x_n) = \ldots
  \]
  This style is popular in the coalgebraic literature.

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

- The \textit{code view} specifies $f$ by a single equation of the form
  \[
  f \, x_1 \ldots \, x_n = \ldots
  \]
  with restrictions on the format of the right-hand side. Lazy functional programming languages such as Haskell support a generalized version of this style.

All three styles are available as input syntax. Whichever syntax is chosen, characteristic theorems for all three styles are generated.
5 DEFINING PRIMITIVELY CORECURSIVE FUNCTIONS

5.1 Introductory Examples

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

5.1.1 Simple Corecursion

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

\[
\text{primcorec literate :: } '(a ⇒ 'a) ⇒ 'a ⇒ 'a llist'' \text{ where}
\]

\[
\text{literate } g \ x = \text{LCons } x \ (\text{literate } g \ (g \ x))
\]

\[
\text{primcorec siterate :: } '(a ⇒ 'a) ⇒ 'a ⇒ 'a stream'' \text{ where}
\]

\[
\text{siterate } g \ x = \text{SCons } x \ (\text{siterate } g \ (g \ x))
\]

The constructor ensures that progress is made—i.e., the function is productive. The above functions compute the infinite lazy list or stream \([x, g x, g (g x), \ldots]\). Productivity guarantees that prefixes \([x, g x, g (g x), \ldots, (g \ldots k) x]\) of arbitrary finite length \(k\) can be computed by unfolding the code equation a finite number of times.

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

\[
\text{primcorec every_snd :: } 'a stream ⇒ 'a stream'' \text{ where}
\]

\[
\text{every snd } s = \text{SCons } (\text{shd } s) \ (\text{stl } (\text{stl } s))
\]

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

\[
\text{primcorec lapp :: } 'a llist ⇒ 'a llist ⇒ 'a llist'' \text{ where}
\]

\[
\text{lapp } xs \ ys =
\]

\[
\text{case } xs \ of
\]

\[
\text{LNil ⇒ } ys
\]

\[
\text{| LCons } x \ xs' ⇒ \text{LCons } x \ (\text{lapp } xs' \ ys)
\]

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

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

**simps_of_case lapp_simps: lapp.code**

This generates the lemma collection `lapp_simps`:

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

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 ∘ f)\ (f\ n))
    (random_process\ every_snd\ (stl\ s)\ (f ∘ 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
"even_infty = Even_ESucc odd_infty"
| "odd_infty = Odd_ESucc even_infty"
```
5.1.3 Nested Corecursion

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

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

primcorec iterate_is: "'(a ⇒ 'a fset) ⇒ 'a ⇒ 'a tree_is" where
  "iterate_is g x = Node_is x (fimage (iterate_is g) (g x))"
```

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

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

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

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

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

The next example illustrates corecursion through functions, which is a bit special. Deterministic finite automata (DFAs) are traditionally defined as 5-tuples `(Q, Σ, δ, q_0, F)`, where `Q` is a finite set of states, `Σ` is a finite alphabet, `δ` 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:

```plaintext
primcorec sm_of_dfa :: "'(q ⇒ 'a ⇒ 'q) ⇒ 'q set ⇒ 'q ⇒ 'a sm" where
  "sm_of_dfa δ F q = SM (q ∈ F) (sm_of_dfa δ F o δ q)"
```

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

```plaintext
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))"
```

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

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

primcorec or_sm :: "a sm ⇒ 'a sm ⇒ 'a sm" where
  "or_sm M N = SM (accept M ∨ accept N) (λa. or_sm (trans M a) (trans N a))"

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

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

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

Primcorec
  sm2_of_dfa :: "('q ⇒ 'a ⇒ 'b ⇒ 'q) ⇒ 'q set ⇒ 'q ⇒ ('a, 'b) sm2"
  where
  "sm2_of_dfa δ F q = SM2 (q ∈ F) (λa b. sm2_of_dfa δ F (δ q a b))"

5.1.4 Nested-as-Mutual Corecursion

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

primcorec
  iterate :i :i :: "('a ⇒ 'a llist) ⇒ 'a ⇒ 'a tree"i" and
  iterates :i :i :: "('a ⇒ 'a llist) ⇒ 'a llist ⇒ 'a tree"i llist"
  where
  "iterate g x = Node :i i x (iterates :i :i g (g x))"
  | "iterates :i :i g xs =
        (case xs of
          LNil ⇒ LNil
        | LCons x xs' ⇒ LCons (iterate :i :i g x) (iterates :i :i g xs'))"

Coinduction rules are generated as iteratei_i.coinduct, iteratesi_i.coinduct, and iteratei_i.iteratesi_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 DEFINING PRIMITIVELY CORECURSIVE FUNCTIONS

5.1.5 Constructor View

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

Here is an example where there is a difference:

\[
\text{primcorec } \text{lapp} :: \text{"\'}a \text{ list} \Rightarrow \text{\'}a \text{ list} \Rightarrow \text{\'}a \text{ list}\n\]

| “lnull xs \implies lnull ys \implies lapp xs ys = LNil” |
| “\_ \implies 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:

\[
\text{primcorec } \text{random_process} :: \text{"\'}a \text{ stream} \Rightarrow (\text{int} \Rightarrow \text{int}) \Rightarrow \text{int} \Rightarrow \text{\'}a \text{ process}\n\]

| “n \mod 4 = 0 \implies \text{random_process } s \ f \ n = \text{Fail}” |
| “n \mod 4 = 1 \implies \text{random_process } s \ f \ n = \text{Skip (random_process } s \ f \ (f \ n))” |
| “n \mod 4 = 2 \implies \text{random_process } s \ f \ n = \text{Action (shd } s) (\text{random_process } (\text{stl } s) \ f \ (f \ n))” |
| “n \mod 4 = 3 \implies \text{random_process } s \ f \ n = \text{Choice (random_process } (\text{every snd } s) \ f \ (f \ n)) |
| (\text{random_process } (\text{every snd } (\text{stl } s)) \ f \ (f \ n))” |

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

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

```plaintext
primcorec literate :: "'(a ⇒ 'a) ⇒ 'a ⇒ 'a llist" where
   "⇒ lnull (literate _ x)"
| "lhd (literate _ x) = x"
| "ltl (literate g x) = literate g (g x)"

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

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

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:

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

```plaintext
primcorec
random_process :: "'a stream ⇒ (int ⇒ int) ⇒ int ⇒ 'a process" where
   "⇒ n mod 4 = 0 ⇒ random_process s f n = Fail"
```
“n mod 4 = 1 \implies \text{is\_Skip}\ (\text{random\_process}\ s\ f\ n)”
“n mod 4 = 2 \implies \text{is\_Action}\ (\text{random\_process}\ s\ f\ n)”
“n mod 4 = 3 \implies \text{is\_Choice}\ (\text{random\_process}\ s\ f\ n)”
“cont\ (\text{random\_process}\ s\ f\ n) = \text{random\_process}\ s\ f\ (f\ n)” \text{of Skip}
“prefix\ (\text{random\_process}\ s\ f\ n) = \text{shd}\ s”
“cont\ (\text{random\_process}\ s\ f\ n) = \text{random\_process}\ (\text{stl}\ s)\ f\ (f\ n)” \text{of Action}
“left\ (\text{random\_process}\ s\ f\ n) = \text{random\_process}\ (\text{every\_snd}\ s)\ f\ (f\ n)”
“right\ (\text{random\_process}\ s\ f\ n) = \text{random\_process}\ (\text{every\_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:

\begin{verbatim}
primcorec
even_infty :: even_enat and
odd_infty :: odd_enat
where
“even_infty \neq \text{Even\_EZero}”
| “un\_Even\_ESucc\ even_infty = odd_infty”
| “un\_Odd\_ESucc\ odd_infty = even_infty”
primcorec iterate_{i i} :: (“’a \Rightarrow \text{’a list}) \Rightarrow \text{’a tree}_{i i}” where
| “lbl_{i i} (iterate_{i i} g\ x) = x”
| “sub_{i i} (iterate_{i i} g\ x) = \text{lmap}\ (iterate_{i i} g)\ (g\ x)”
\end{verbatim}

5.2 Command Syntax
5.2.1 primcorec and primcorecursive

\begin{verbatim}
primcorec : local\_theory \rightarrow local\_theory
primcorecursive : local\_theory \rightarrow proof(prove)
\end{verbatim}
The \texttt{primcorec} and \texttt{primcorecursive} commands introduce a set of mutually corecursive functions over codatatypes.

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 \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 \textit{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):

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

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

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

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

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

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

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

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

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.

\[ f_1 \ldots f_m.\text{coinduct\_strong} \text{ [case\_names } t_1 \ldots t_m, \]
\[ \text{case\_conclusion } D_1 \ldots D_n] : \]

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

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

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, \( 'a \text{ llist} \) is a unary BNF. Its predicator \( \text{llist\_all} :: ('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ llist} \Rightarrow \text{bool} \) extends unary predicates over elements to unary predicates over lazy lists. Similarly, its relator \( \text{llist\_all2} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ llist} \Rightarrow 'b \text{ llist} \Rightarrow \text{bool} \) extends binary predicates over elements to binary predicates over parallel lazy lists. The
cardinal bound limits the number of elements returned by the set function; it may not depend on the cardinality of 'a.

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

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

### 6.2 Introductory Examples

The example below shows how to register a type as a BNF using the \texttt{bnf} command. Some of the proof obligations are best viewed with the bundle 'cardinal\_syntax' included.

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

\begin{verbatim}
typedef (\forall d, a) fn = "UNIV :: (\forall d \Rightarrow a) set"
by simp

setup_lifting type_definition_fn

lift_definition map_fn :: "(\forall a \Rightarrow b) \Rightarrow (\forall d, a) fn \Rightarrow (\forall d, b) fn" is "(\_)

lift_definition set_fn :: "(\forall d, a) fn \Rightarrow a set" is range

lift_definition pred_fn :: "(\forall a \Rightarrow bool) \Rightarrow (\forall d, a) fn \Rightarrow bool"
  is "pred_fun (\_ \Rightarrow True)"

lift_definition rel_fn :: "(\forall a \Rightarrow b \Rightarrow bool) \Rightarrow (\forall d, a) fn \Rightarrow (\forall d, b) fn \Rightarrow bool"
  is "rel_fun (=)"

bnf "(\forall d, a) fn"
  map: map_fn
  sets: set_fn
  bd: "natLeq + c card_suc | UNIV :: (\forall d) set"
  rel: rel_fn
  pred: pred_fn
proof
  show "map_fn id = id"
\end{verbatim}
by transfer auto

next
fix f :: "'a ⇒ 'b" and g :: "'b ⇒ 'c"
show "map_fn (g ∘ f) = map_fn g ∘ map_fn f"
  by transfer (auto simp add: comp_def)

next
fix F :: "'(d, 'a) fn" and f g :: "'a ⇒ 'b"
assume "\x. x ∈ set_fn F ==> f x = g x"
then show "map_fn f F = map_fn g F"
  by transfer auto

next
fix f :: "'a ⇒ 'b"
show "set_fn ∘ map_fn f = (\_ x. f x) ∘ set_fn"
  by transfer (auto simp add: comp_def)

next
show "card_order (natLeq + c card_suc |UNIV :: 'd set| )"
  by (rule card_order_bd_fun)

next
show "cinfinite (natLeq + c card_suc |UNIV :: 'd set| )"
  by (rule Cinfinite_bd_fun[THEN conjunct1])

next
show "regularCard (natLeq + c card_suc |UNIV :: 'd set| )"
  by (rule regularCard_bd_fun)

next
fix F :: "'(d, 'a) fn"
have "|set_fn F| ≤ o |UNIV :: 'd set|" (is "\_ ≤ o \_U")
  by transfer (rule card_of_image)
also have "\U < o card_suc \U"
  by (simp add: card_of_card_order_on card_suc_greater)
also have "card_suc \U ≤ o natLeq + c card_suc \U"
  using Card_order_card_suc card_of_card_order_on ordLeq_csum2 by blast
finally show "|set_fn F| < o natLeq + c card_suc |UNIV :: 'd set| .".

next
fix R :: "'a ⇒ 'b ⇒ bool" and S :: "'b ⇒ 'c ⇒ bool"
show "rel_fn R OO rel_fn S ≤ rel_fn (R OO S)"
  by (rule, transfer) (auto simp add: rel_fun_def)

next
fix R :: "'a ⇒ 'b ⇒ bool"
show "rel_fn R = (\λx y. \exists z. set_fn z ⊆ {(x, y). R x y} ∧ map_fn fst z = x ∧ map_fn snd z = y)"
  unfolding fun_eq_iff relcompp simps conversep simps
  by transfer (force simp: rel_fun_def subset_iff)
next
fix $P :: "'a ⇒ bool"
show "pred_fn P = (λx. Ball (set_fn x) P)"
  unfolding fun_eq_iff by transfer simp
qed

print_theorems
print_bnfs

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

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

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

lift_bnf $(d, a)\ fn
  by force+

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

copy_bnf $(d, a)\ fn

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

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

copy_bnf $(a, \ z)\ point_ext

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

typedef $(a)\ nonempty_list = \{xs :: 'a\ list. xs \neq []\} \ by\ auto

The lift_bnf command requires us to prove that the set of nonempty lists is closed under the map function and the zip function. The latter only occurs implicitly in the goal, in form of the variable zs.
lift_bnf 'a nonempty_list
proof -
  fix f and xs :: "'a list"
  assume "xs ∈ {xs. xs ≠ []}"
  then show "map f xs ∈ {xs. xs ≠ []}"
    by (cases xs) auto
next
  fix zs :: "('a × 'b) list"
  assume "map fst zs ∈ {xs. xs ≠ []} " "map snd zs ∈ {xs. xs ≠ []}"
  then show "∃zs'∈{xs. xs ≠ []}. set zs' ⊆ set zs ∧
    map fst zs' = map fst zs ∧
    map snd zs' = map snd zs"
    by (cases zs) (auto intro!: exI[of _ zs])
qed

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

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

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

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

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

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

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

\texttt{print_theorems}
\texttt{print\_bnfs}

6.3 Command Syntax

6.3.1 bnf

\texttt{bnf \ : \ local\_theory \ \rightarrow \ \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.

6.3.2 \texttt{lift_bnf}

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

Nonemptiness witnesses are not lifted from the raw type’s BNF, as this would be incomplete. They must be given as terms (on the raw type) and proved to be witnesses. The command warns about witness types that are present in the raw type’s BNF but not supplied by the user. The warning can be disabled by specifying the `no_warn_wits` option.

### 6.3.3 copy_bnf

```plaintext
copy_bnf : local_theory → local_theory
```

The `copy_bnf` command performs the same lifting as `lift_bnf` for type copies (typedefs with `UNIV` as the representing set), without requiring the user to discharge any proof obligations or provide nonemptiness witnesses.

### 6.3.4 bnf_axiomatization

```plaintext
bnf_axiomatization : local_theory → local_theory
```
The \texttt{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 \texttt{target} can be used to specify a local context, \texttt{name} denotes an identifier, \texttt{typefree} denotes fixed type variable (\texttt{'a}, \texttt{'b}, \ldots), \texttt{mixfix} denotes the usual parenthesized mixfix notation, and \texttt{types} denotes a space-separated list of types \cite{12}.

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

Type arguments are live by default; they can be marked as dead by entering \texttt{dead} in front of the type variable (e.g., \texttt{(dead 'a)}) instead of an identifier for the corresponding set function. Witnesses can be specified by their types. Otherwise, the syntax of \texttt{bnf\_axiomatization} is identical to the left-hand side of a \texttt{datatype} or \texttt{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 \texttt{~/src/HOL/Library/BNF\_Axiomatization.thy} theory to use this functionality.
7 Deriving Destructors and Constructor Theorems

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

7.1 Command Syntax

7.1.1 \texttt{free_constructors}

\texttt{free_constructors} : \texttt{local\_theory} \rightarrow \texttt{proof(prove)}
The 

**free_constructors** command generates destructor constants for freely constructed types as well as properties about constructors and destructors. It also registers the constants and theorems in a data structure that is queried by various tools (e.g., function).

The syntactic entity `target` can be used to specify a local context, `name` denotes an identifier, `prop` denotes a HOL proposition, and `term` denotes a HOL term [12].

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

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

### 7.1.2 simps_of_case

```
simps_of_case : local_theory → local_theory
```
The `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 = (case xs of LNil ⇒ ys | LCons x xs' ⇒ 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`

```
    case_of_simps : local_theory → local_theory
```

The `case_of_simps` command provided by theory `~/src/HOL/Library/Simps_Case_Conv.thy` converts a set of pattern-matching equations into single equation with a complex case expression on the right-hand side (cf. `simps_of_case`). For example,

```
    case_of_simps lapp_case: lapp_simps
```

translates

```
lapp LNil ys = ys
lapp (LCons xa x) ys = LCons xa (lapp x ys)
```

into `lapp xba x3a = (case xba of LNil ⇒ x3a | LCons x2ba x1ba ⇒ LCons x2ba (lapp x1ba x3a)).`
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:

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

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:

\begin{verbatim}
t.eq.refl [code nbe]:
  equal_class.equal x x \equiv True

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

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 \( size :: t \Rightarrow \text{nat} \) belonging to the size type class.
The \texttt{fun} command relies on \textit{size} to prove termination of recursive functions on datatypes.

The plugin derives the following properties:

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

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

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

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

The \texttt{t.size} and \texttt{t.size_t} functions generated for datatypes defined by nested recursion through a datatype \textit{u} depend on \textit{u.size_u}.

If the recursion is through a non-datatype \textit{u} with type arguments \texttt{'}a\sb{1}, . . . , \texttt{'}a\sb{m}, by default \textit{u} values are given a size of 0. This can be improved upon by registering a custom size function of type \texttt{('a\sb{1} ⇒ nat) ⇒ . . . ⇒ ('a\sb{m} ⇒ nat) ⇒ u ⇒ nat} using the ML function \texttt{BNF\_LFP\_Size.register_size} or \texttt{BNF\_LFP\_Size.register_size_global}. See theory \texttt{~/src/HOL/Library/Multiset.thy} for an example.

\subsection{Transfer}

For each (co)datatype with live type arguments and each manually registered BNF, the \textbf{transfer} plugin generates a predicator \texttt{t.pred\_t} and properties that guide the Transfer tool.

For types with at least one live type argument and \texttt{no dead type arguments}, the plugin derives the following properties:

- \texttt{t.Domainp\_rel [relator\_domain]}:
  \begin{align*}
  & \text{Domainp } (\text{list\_all2 } R) = \text{list\_all } (\text{Domainp } R)
  \end{align*}

- \texttt{t.left\_total\_rel [transfer\_rule]}:
  \begin{align*}
  & \text{left\_total } R \Rightarrow \text{left\_total } (\text{list\_all2 } R)
  \end{align*}
8 SELECTING PLUGINS

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

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

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

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

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

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

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

8.4 Lifting

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

The plugin derives the following property:

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

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

8.5 Quickcheck

The integration of datatypes with Quickcheck is accomplished by the quick-
check 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


