Defining (Co)datatypes in Isabelle/HOL

Jasmin Christian Blanchette, Martin Desharnais,
Lorenz Panny, Andrei Popescu, and Dmitriy Traytel
Fakultät für Informatik, Technische Universität München

27 August 2014

Abstract

This tutorial describes the new package for defining datatypes and codatatypes in Isabelle/HOL. The package provides four main commands: **datatype_new**, **codatatype**, **primrec**, and **primcorec**. The first command is expected to eventually replace the old **datatype** command.

Contents

1 Introduction

The 2013 edition of Isabelle introduced a new definitional package for freely generated datatypes and codatatypes. The datatype support is similar to that provided by the earlier package due to Berghofer and Wenzel [1], documented in the Isar reference manual [9]; indeed, replacing the keyword **datatype** by **datatype_new** is usually all that is needed to port existing theories to use the new package.

Perhaps the main advantage of the new package is that it supports recursion through a large class of non-datatypes, such as finite sets:

datatype_new *'a tree*$_{fs}$ = *Node*$_{fs}$ (*lbl*$_{fs}$: *'a*) (*sub*$_{fs}$: "*'a tree*$_{fs}$ *fset*")

Another strong point is the support for local definitions:

context *linorder*
begin
datatype_new *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 new 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_new *'a tree*$_{ff}$ = *Node*$_{ff}$ *'a* "*'a tree*$_{ff}$ *list*"
datatype_new *'a tree*$_{fi}$ = *Node*$_{fi}$ *'a* "*'a tree*$_{fi}$ *llist*"
codatatype *'a tree*$_{if}$ = *Node*$_{if}$ *'a* "*'a tree*$_{if}$ *list*"

codatatype $'a\ tree_{ii} = Node_{ii}\ 'a$ "$'a\ tree_{ii}\ llist$"

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 *BNF_Axiomatization*, located in the directory `~~/src/HOL/Library`.

The package, like its predecessor, fully adheres to the LCF philosophy [4]: The characteristic theorems associated with the specified (co)datatypes are derived rather than introduced axiomatically.[1] The package is described in a number of papers [2, 3, 7, 8]. 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_new** command.
- Section 3, "Defining Recursive Functions," describes how to specify recursive functions using **primrec**.
- Section 4, "Defining Codatatypes," describes how to specify codatatypes using the **codatatype** command.
- Section 5, "Defining Corecursive Functions," describes how to specify corecursive functions using the **primcorec** and **primcorecursive** commands.
- Section 6, "Introducing Bounded Natural Functors," explains how to use the **bnf** command to register arbitrary type constructors as BNFs.
- Section 7, "Deriving Destructors and Theorems for Free Constructors," explains how to use the command **free_constructors** to derive destructor constants and theorems for freely generated types, as performed internally by **datatype_new** and **codatatype**.

The command **datatype_new** is expected to replace **datatype** in a future release. Authors of new theories are encouraged to use the new commands, and maintainers of older theories may want to consider upgrading.

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

[1]However, some of the internal constructions and most of the internal proof obligations are skipped if the *quick_and_dirty* option is enabled.

2 Defining Datatypes

Datatypes can be specified using the **datatype_new** 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/BNF/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_new $\mathit{trool} = \mathit{Truue} \mid \mathit{Faalse} \mid \mathit{Perhaaps}$

Here, *Truue*, *Faalse*, and *Perhaaps* have the type *trool*.

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

datatype_new $'a\ \mathit{option} = \mathit{None} \mid \mathit{Some}\ 'a$

The constructors are $\mathit{None} :: 'a\ \mathit{option}$ and $\mathit{Some} :: 'a \Rightarrow 'a\ \mathit{option}$.

The next example has three type parameters:

datatype_new $('a, 'b, 'c)\ \mathit{triple} = \mathit{Triple}\ 'a\ 'b\ 'c$

The constructor is $\mathit{Triple} :: 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow ('a, 'b, 'c)\ \mathit{triple}$. Unlike in Standard ML, curried constructors are supported. The uncurried variant is also possible:

datatype_new $('a, 'b, 'c)\ \mathit{triple}_u = \mathit{Triple}_u$ “$'a * 'b * 'c$”

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_new $\mathit{nat} = \mathit{Zero} \mid \mathit{Suc}\ \mathit{nat}$

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

datatype_new $('a, 'b)\ \mathit{tlist} = \mathit{TNil}\ 'b \mid \mathit{TCons}\ 'a$ “$('a, 'b)\ \mathit{tlist}$”

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_new even_nat = Even_Zero | Even_Suc odd_nat
and odd_nat = Odd_Suc even_nat
```

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

```
datatype_new ('a, 'b) exp =
  Term "('a, 'b) trm" | Sum "('a, 'b) trm" "('a, 'b) exp"
and ('a, 'b) trm =
  Factor "('a, 'b) fct" | Prod "('a, 'b) fct" "('a, 'b) trm"
and ('a, 'b) fct =
  Const 'a | Var 'b | Expr "('a, 'b) exp"
```

2.1.4 Nested Recursion

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

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

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

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

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

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

And so is the definition of hereditarily finite sets:

```
datatype_new hfset = HFSet "hfset fset"
```

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

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

Here is another example that fails:

datatype_new *'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 and Properties

The **datatype_new** command introduces various constants in addition to the constructors. With each datatype are associated set functions, a map function, a relator, discriminators, and selectors, all of which can be given custom names. In the example below, the familiar names *null*, *hd*, *tl*, *set*, *map*, and *list_all2*, override the default names *is_Nil*, *un_Cons1*, *un_Cons2*, *set_list*, *map_list*, and *rel_list*:

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

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

Constructors:	$Nil :: 'a\ list$
	$Cons :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
Discriminator:	$null :: 'a\ list \Rightarrow bool$
Selectors:	$hd :: 'a\ list \Rightarrow 'a$
	$tl :: 'a\ list \Rightarrow 'a\ list$
Set function:	$set :: 'a\ list \Rightarrow 'a\ set$
Map function:	$map :: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$
Relator:	$list\_all2 :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow bool$

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

$$\mathit{null}\ \mathit{xs} \Longrightarrow \mathit{xs} = \mathit{Nil} \qquad \neg\ \mathit{null}\ \mathit{xs} \Longrightarrow \mathit{Cons}\ (\mathit{hd}\ \mathit{xs})\ (\mathit{tl}\ \mathit{xs}) = \mathit{xs}$$

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

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. Here, 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 $\lambda \mathit{xs}.\ \mathit{xs} = \mathit{Nil}$ 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:

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

datatype_new (*set*: *'a*) *list* =
  null: *Nil* ("[]")
| *Cons* (*hd*: *'a*) (*tl*: "*'a list*") (**infixr** "#" 65)
for
  map: *map*
  rel: *list_all2*

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

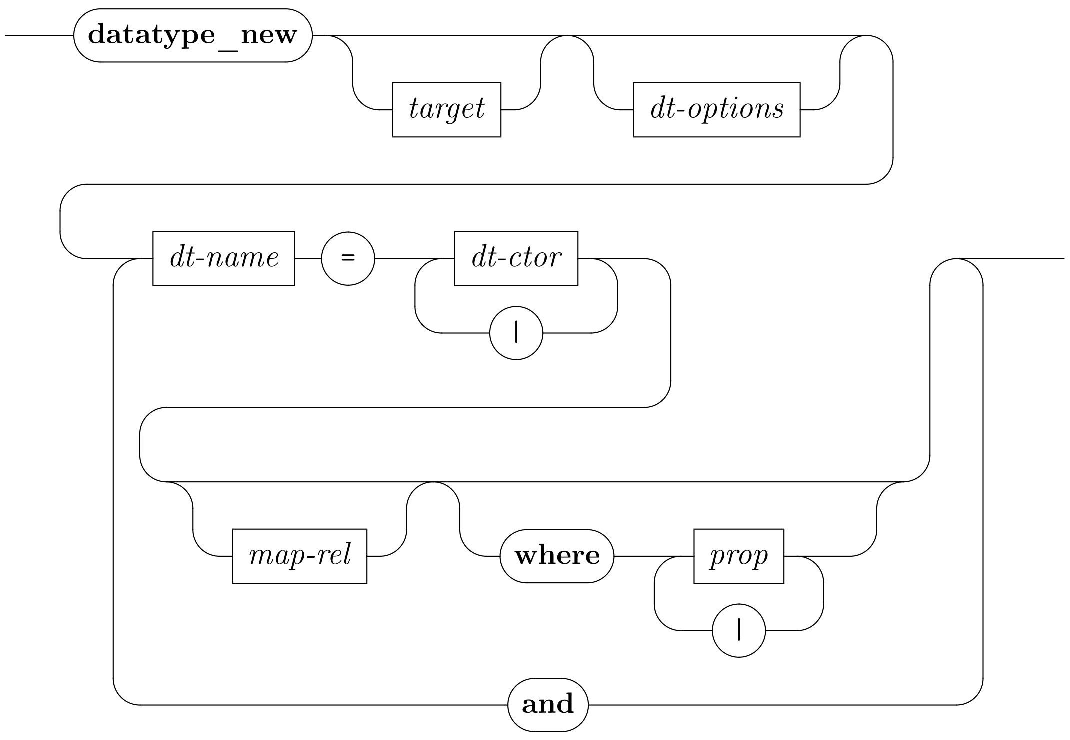
syntax "*_list*" :: "*args* ⇒ *'a list*" ("[(_)]")

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

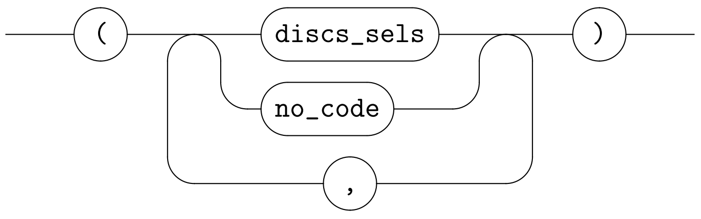
2.2 Command Syntax

2.2.1 datatype_new

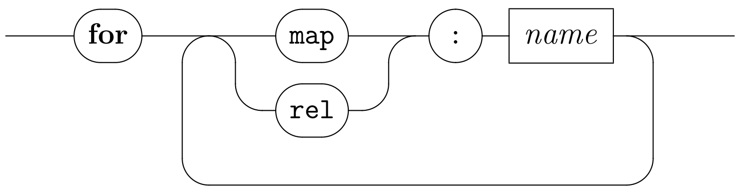
datatype_new : *local_theory* → *local_theory*



dt-options



map-rel



The **datatype_new** 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*) [9]), and *prop* denotes a HOL proposition.

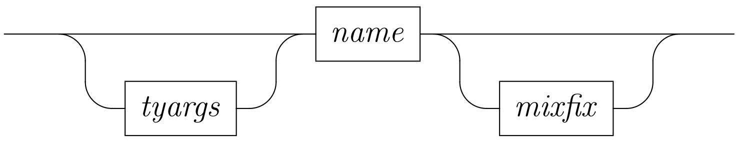
The optional target is optionally followed by one or both of the following options:

- 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 *no_code* option indicates that the datatype should not be registered for code generation.

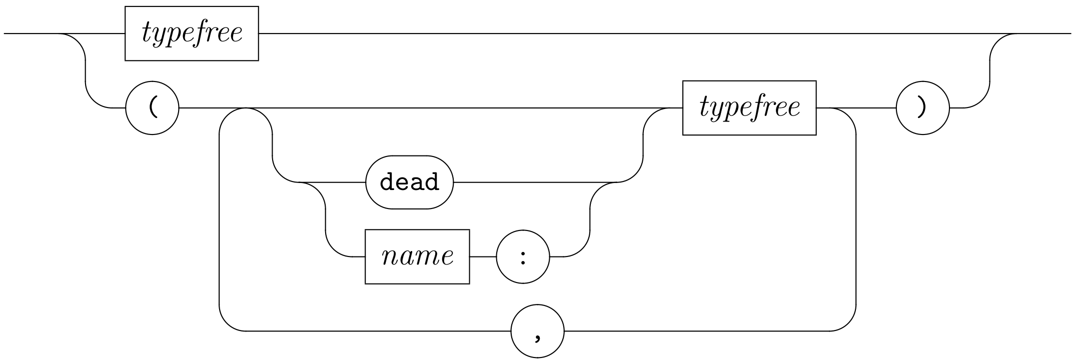
The optional **where** clause specifies default values for selectors. Each proposition must be an equation of the form $un\_D\ (C\ \ldots) = \ldots$, where C is a constructor and un_D is a selector.

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

dt-name



tyargs



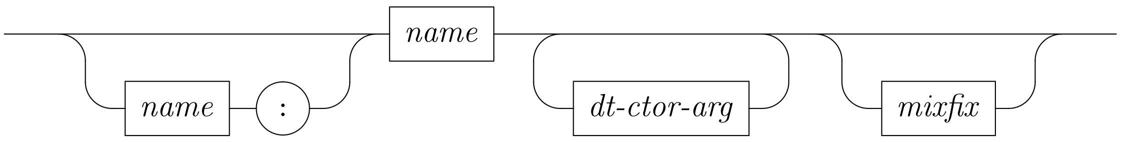
The syntactic entity *name* denotes an identifier, *mixfix* denotes the usual parenthesized mixfix notation, and *typefree* denotes fixed type variable (*'a*, *'b*, . . .) [9].

The optional names preceding the type variables allow to override the default names of the set functions ($set1\_t, \ldots, setM\_t$). Type arguments can be marked as dead by entering "*dead*" in front of the type variable (e.g., "(*dead* *'a*)"); otherwise, they are live or dead (and a set function is generated or not) depending on where they occur in the right-hand sides of the definition.

Declaring a type argument as dead can speed up the type definition but will prevent any later (co)recursion through that type argument.

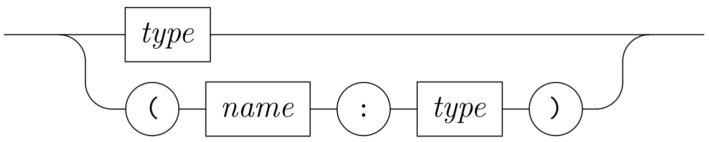
Inside a mutually recursive specification, all defined datatypes must mention exactly the same type variables in the same order.

dt-ctor



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 to override the default, which is $\lambda x.\ x = C_j$ for nullary constructors and $t.is\_C_j$ otherwise.

dt-ctor-arg

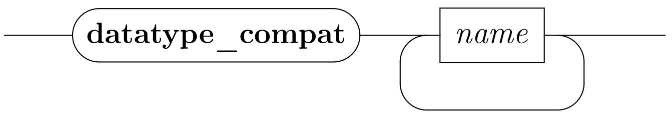


The syntactic entity *type* denotes a HOL type [9].

In addition to the type of a constructor argument, it is possible to specify a name for the corresponding selector to override the default name ($un\_C_j i$). The same selector names can be reused for several constructors as long as they share the same type.

2.2.2 datatype_compat

$$\textbf{datatype\_compat} \ : \ local\_theory \rightarrow local\_theory$$



The **datatype_compat** command registers new-style datatypes as old-style datatypes. For example:

datatype_compat *even_nat odd_nat*

ML $\{*$ *Datatype_Data.get_info* @$\{$*theory*$\}$ @$\{$*type_name even_nat*$\}$ $*\}$

The syntactic entity *name* denotes an identifier [9].

The command is interesting because the old datatype package provides some functionality that is not yet replicated in the new package, such as the integration with Quickcheck.

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*).
- All types through which recursion takes place must be new-style datatypes or the function type. In principle, it should be possible to support old-style datatypes as well, but this has not been implemented yet (and there is currently no way to register old-style datatypes as new-style datatypes).
- The recursor produced for types that recurse through functions has a different signature than with the old package. This might affect the behavior of some third-party extensions.

An alternative to **datatype_compat** is to use the old package's **rep_ datatype** command. The associated proof obligations must then be discharged manually.

2.3 Generated Constants

Given a datatype $('a_1, \ldots, 'a_m)\ t$ with $m > 0$ 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:	$t.set1\_t, \ldots, t.setm\_t$
Map function:	$t.map\_t$
Relator:	$t.rel\_t$
Recursor:	$t.rec\_t$
Size function:	$t.size\_t$

The case combinator, discriminators, and selectors are collectively called *destructors*. The prefix "*t.*" is an optional component of the names and is normally hidden.

In addition to the above, the *size* class is instantiated to overload the *size* function based on *t.size_t*.

2.4 Generated Theorems

The characteristic theorems generated by **datatype_new** 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 as usual by entering the command **print_theorems** immediately after the datatype definition. This list normally excludes low-level theorems that reveal internal constructions. To make these accessible, add the line

declare [[*bnf_note_all*]]

to the top of the theory file.

2.4.1 Free Constructor Theorems

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

*t.***inject** [*iff*, *induct_simp*]:
$(x21 \mathbin{\#} x22 = y21 \mathbin{\#} y22) = (x21 = y21 \wedge x22 = y22)$

*t.***distinct** [*simp*, *induct_simp*]:
$[] \neq x21 \mathbin{\#} x22$
$x21 \mathbin{\#} x22 \neq []$

*t.***exhaust** [*cases t*, *case_names* $C_1 \ldots C_n$]:
$[\![y = [] \Longrightarrow P; \bigwedge x21\ x22.\ y = x21 \mathbin{\#} x22 \Longrightarrow P]\!] \Longrightarrow P$

$t.$***nchotomy***:
: $\forall\, list.\ list = [] \lor (\exists\, x21\ x22.\ list = x21\ \#\ x22)$

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

$t.$***distinct*** **[*THEN notE*,** *elim!***]**:
: $[] = x21\ \#\ x22 \Longrightarrow R$
: $x21\ \#\ x22 = [] \Longrightarrow R$

The next subgroup is concerned with the case combinator:

$t.$***case*** [*simp*, *code*]:
: $(case\ []\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = f1$
: $(case\ x21\ \#\ x22\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = f2\ x21\ x22$

$t.$***case_cong*** [*fundef_cong*]:
: $[\![list = list';\ list' = [] \Longrightarrow f1 = g1;\ \bigwedge x21\ x22.\ list' = x21\ \#\ x22 \Longrightarrow f2\ x21\ x22 = g2\ x21\ x22]\!] \Longrightarrow (case\ list\ of\ [] \Rightarrow f1 \mid x21\ \#\ x22 \Rightarrow f2\ x21\ x22) = (case\ list'\ of\ [] \Rightarrow g1 \mid x21\ \#\ x22 \Rightarrow g2\ x21\ x22)$

$t.$***weak_case_cong*** [*cong*]:
: $list = list' \Longrightarrow (case\ list\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = (case\ list'\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa)$

$t.$***split***:
: $P\ (case\ list\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = ((list = [] \longrightarrow P\ f1) \land (\forall\, x21\ x22.\ list = x21\ \#\ x22 \longrightarrow P\ (f2\ x21\ x22)))$

$t.$***split_asm***:
: $P\ (case\ list\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = (\neg\ (list = [] \land \neg\ P\ f1 \lor (\exists\, x21\ x22.\ list = x21\ \#\ x22 \land \neg\ P\ (f2\ x21\ x22))))$

$t.$***splits*** = *split split_asm*

The third subgroup revolves around discriminators and selectors:

$t.$***disc*** [*simp*]:
: $null\ []$
: $\neg\ null\ (x21\ \#\ x22)$

$t.$***discI***:
: $list = [] \Longrightarrow null\ list$
: $list = x21\ \#\ x22 \Longrightarrow \neg\ null\ list$

$t.$***sel*** [*simp*, *code*]:
: $hd\ (x21\ \#\ x22) = x21$
: $tl\ (x21\ \#\ x22) = x22$

$t.$***collapse*** [*simp*]:
: $null\ list \Longrightarrow list = []$
: $\neg\ null\ list \Longrightarrow hd\ list\ \#\ tl\ list = list$

$t.$***disc_exclude*** [*dest*]:
: These properties are missing for *'a list* because there is only one proper discriminator. Had the datatype been introduced with a second discriminator called *nonnull*, they would have read thusly:
: $null\ list \Longrightarrow \neg\ nonnull\ list$
: $nonnull\ list \Longrightarrow \neg\ null\ list$

$t.$***disc_exhaust*** [*case_names* $C_1 \ldots C_n$]:
: $[\![null\ list \Longrightarrow P; \neg\ null\ list \Longrightarrow P]\!] \Longrightarrow P$

$t.$***sel_exhaust*** [*case_names* $C_1 \ldots C_n$]:
: $[\![list = [] \Longrightarrow P; list = hd\ list\ \#\ tl\ list \Longrightarrow P]\!] \Longrightarrow P$

$t.$***expand***:
: $[\![null\ list = null\ list'; [\![\neg\ null\ list; \neg\ null\ list']\!] \Longrightarrow hd\ list = hd\ list' \wedge tl\ list = tl\ list']\!] \Longrightarrow list = list'$

$t.$***sel_split***:
: $P\ (case\ list\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = ((list = [] \longrightarrow P\ f1) \wedge (list = hd\ list\ \#\ tl\ list \longrightarrow P\ (f2\ (hd\ list)\ (tl\ list))))$

$t.$***sel_split_asm***:
: $P\ (case\ list\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = (\neg\ (list = [] \wedge \neg\ P\ f1 \vee list = hd\ list\ \#\ tl\ list \wedge \neg\ P\ (f2\ (hd\ list)\ (tl\ list))))$

$t.$***case_eq_if***:
: $(case\ list\ of\ [] \Rightarrow f1 \mid x\ \#\ xa \Rightarrow f2\ x\ xa) = (if\ null\ list\ then\ f1\ else\ f2\ (hd\ list)\ (tl\ list))$

In addition, equational versions of $t.disc$ are registered with the [*code*] attribute.

2.4.2 Functorial Theorems

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

$t.$***set*** [*simp*, *code*]:
: $set\ [] = \{\}$
: $set\ (x21a\ \#\ x22) = \{x21a\} \cup set\ x22$

t.***set_empty***:
  $null\ list \Longrightarrow set\ list = \{\}$

t.***sel_set***:
  $\neg\ null\ a \Longrightarrow hd\ a \in set\ a$
  $[\![\neg\ null\ a;\ x \in set\ (tl\ a)]\!] \Longrightarrow x \in set\ a$

t.***map*** [*simp*, *code*]:
  $map\ fi\ [] = []$
  $map\ fi\ (x21a\ \#\ x22) = fi\ x21a\ \#\ map\ fi\ x22$

t.***disc_map_iff*** [*simp*]:
  $null\ (map\ f\ a) = null\ a$

t.***sel_map***:
  $\neg\ null\ a \Longrightarrow hd\ (map\ f\ a) = f\ (hd\ a)$
  $\neg\ null\ a \Longrightarrow tl\ (map\ f\ a) = map\ f\ (tl\ a)$

t.***rel_inject*** [*simp*]:
  $list\_all2\ R\ []\ []$
  $list\_all2\ R\ (x21a\ \#\ x22)\ (y21\ \#\ y22) = (R\ x21a\ y21 \wedge list\_all2\ R\ x22\ y22)$

t.***rel_distinct*** [*simp*]:
  $\neg\ list\_all2\ R\ []\ (y21\ \#\ y22)$
  $\neg\ list\_all2\ R\ (y21\ \#\ y22)\ []$

t.***rel_intros***:
  $list\_all2\ R\ []\ []$
  $[\![R\ x21a\ y21;\ list\_all2\ R\ x22\ y22]\!] \Longrightarrow list\_all2\ R\ (x21a\ \#\ x22)\ (y21\ \#\ y22)$

t.***rel_sel***:
  $list\_all2\ R\ a\ b = (null\ a = null\ b \wedge (\neg\ null\ a \longrightarrow \neg\ null\ b \longrightarrow R\ (hd\ a)\ (hd\ b) \wedge list\_all2\ R\ (tl\ a)\ (tl\ b)))$

In addition, equational versions of t.*rel_inject* and *rel_distinct* are registered with the [*code*] attribute.

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

t.***set_map***:
  $set\ (map\ f\ v) = f\ `\ set\ v$

t.***map_comp***:
  $(\bigwedge z.\ z \in set\ x \Longrightarrow f\ z = g\ z) \Longrightarrow map\ f\ x = map\ g\ x$

t.***map_ cong*** [*fundef_ cong*]:
$[\![x = y; \bigwedge z.\ z \in set\ y \Longrightarrow f\ z = g\ z]\!] \Longrightarrow map\ f\ x = map\ g\ y$

t.***map_ id***:
$map\ id\ t = t$

t.***map_ id0***:
$map\ id = id$

t.***map_ ident***:
$map\ (\lambda x.\ x)\ t = t$

t.***rel_ compp***:
$list\_all2\ (R\ OO\ S) = list\_all2\ R\ OO\ list\_all2\ S$

t.***rel_ conversep***:
$list\_all2\ R^{--} = (list\_all2\ R)^{--}$

t.***rel_ eq***:
$list\_all2\ op = = op =$

t.***rel_ flip***:
$list\_all2\ R^{--}\ a\ b = list\_all2\ R\ b\ a$

t.***rel_ mono***:
$R \leq Ra \Longrightarrow list\_all2\ R \leq list\_all2\ Ra$

2.4.3 Inductive Theorems

The inductive theorems are as follows:

t.***induct*** [*case_ names* $C_1 \ldots C_n$, *induct* t]:
$[\![P\ []; \bigwedge x1\ x2.\ P\ x2 \Longrightarrow P\ (x1\ \#\ x2)]\!] \Longrightarrow P\ list$

$t_1\_\ldots\_t_m$.***induct*** [*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.***rel_ induct*** [*case_ names* $C_1 \ldots C_n$, *induct pred*]:
$[\![list\_all2\ R\ x\ y; Q\ []\ []; \bigwedge a21\ a22\ b21\ b22.\ [\![R\ a21\ b21; Q\ a22\ b22]\!] \Longrightarrow Q\ (a21\ \#\ a22)\ (b21\ \#\ b22)]\!] \Longrightarrow Q\ x\ y$

$t_1\_\ldots\_t_m$.***rel_ induct*** [*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.***rec*** [*simp*, *code*]:
$rec\_list\ f1\ f2\ [] = f1$
$rec\_list\ f1\ f2\ (x21\ \#\ x22) = f2\ x21\ x22\ (rec\_list\ f1\ f2\ x22)$

$t.$***rec_o_map***:
  $rec\_list\ h\ ha \circ map\ g = rec\_list\ h\ (\lambda x\ xa.\ ha\ (g\ x)\ (map\ g\ xa))$

$t.$***size*** $[simp,\ code]$:
  $size\_list\ x\ [] = 0$
  $size\_list\ x\ (x21\ \#\ x22) = x\ x21 + size\_list\ x\ x22 + Nat.Suc\ 0$
  $size\ [] = 0$
  $size\ (x21\ \#\ x22) = size\ x22 + Nat.Suc\ 0$

$t.$***size_o_map***:
  $size\_list\ f \circ map\ g = size\_list\ (f \circ g)$

For convenience, **datatype_new** also provides the following collection:

$t.$***simps*** $=$ *t.inject t.distinct t.case t.rec t.map t.rel_inject*
  t.rel_distinct t.set

2.5 Compatibility Issues

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

- *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. This concerns Quickcheck in particular. Whenever possible, it is recommended to use **datatype_compat** to register new-style datatypes as old-style datatypes.
- *The constants t_case, t_rec, and t_size are now called case_t, rec_t, and size_t.*
- *The recursor rec_t has a different signature for nested recursive datatypes.* In the old package, nested recursion through non-functions was internally reduced to mutual recursion. This reduction was visible in the type of the recursor, used by **primrec**. Recursion through functions was handled specially. In the new package, nested recursion (for functions and non-functions) is handled in a more modular fashion. The old-style recursor can be generated on demand using **primrec** if the recursion is via new-style datatypes, as explained in Section 3.1.5.
- *Accordingly, the induction rule is different for nested recursive datatypes.* Again, the old-style induction rule can be generated on demand using **primrec** if the recursion is via new-style datatypes, as explained in Section 3.1.5.

- *The internal constructions are completely different.* Proof texts that unfold the definition of constants introduced by **datatype** will be difficult to port.
- *Some theorems have different names.* For non-mutually recursive datatypes, the alias *t.inducts* for *t.induct* is no longer generated. For $m > 1$ mutually recursive datatypes, $t_1\_\ldots\_t_m$*.inducts*(i) has been renamed t_i*.induct* for each $i \in \{1, \ldots, t\}$, and similarly the collection $t_1\_\ldots\_t_m$*.size* has been divided into t_1*.size*, …, t_m*.size*.
- *The t.simps collection has been extended.* Previously available theorems are available at the same index.
- *Variables in generated properties have different names.* This is rarely an issue, except in proof texts that refer to variable names in the [*where* …] attribute. The solution is to use the more robust [*of* …] syntax.

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

3 Defining Recursive Functions

Recursive functions over datatypes can be specified using the **primrec** command, which supports primitive recursion, or using the more general **fun** and **function** commands. Here, the focus is on **primrec**; the other two commands are described in a separate tutorial [5].

3.1 Introductory Examples

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

3.1.1 Nonrecursive Types

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

primrec *bool_of_trool* :: "*trool* ⇒ *bool*" **where**
  "*bool_of_trool Faalse* ⟷ *False*" |

```
  "bool_of_trool Truue ⟷ True"

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

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

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

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

3.1.2 Simple Recursion

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

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

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

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

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

```
simps_of_case at_simps: at.simps
```

This generates the lemma collection *at_simps*:

$$at\ (x\ \#\ xs)\ Zero = x \qquad at\ (xa\ \#\ xs)\ (nat.Suc\ x) = at\ xs\ x$$

The next example is defined using **fun** to escape the syntactic restrictions imposed on primitively recursive functions. The **datatype_compat** command is needed to register new-style datatypes for use with **fun** and **function** (Section 2.2.2):

```
datatype_compat nat

fun at_least_two :: "nat ⇒ bool" where
  "at_least_two (Suc (Suc _)) ⟷ True" |
  "at_least_two _ ⟷ False"
```

3.1.3 Mutual Recursion

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

```
primrec
  nat_of_even_nat :: "even_nat ⇒ nat" and
  nat_of_odd_nat :: "odd_nat ⇒ nat"
where
  "nat_of_even_nat Even_Zero = Zero" |
  "nat_of_even_nat (Even_Suc n) = Suc (nat_of_odd_nat n)" |
  "nat_of_odd_nat (Odd_Suc n) = Suc (nat_of_even_nat n)"

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

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

```
fun
  even :: "nat ⇒ bool" and
  odd :: "nat ⇒ bool"
where
  "even Zero = True" |
  "even (Suc n) = odd n" |
  "odd Zero = False" |
  "odd (Suc 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 at_ff :: "'a tree_ff ⇒ nat list ⇒ 'a" where
  "at_ff (Node_ff a ts) js =
     (case js of
        [] ⇒ a
     | j # js' ⇒ at (map (λt. at_ff t js') ts) j)"
```

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

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

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

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

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

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

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

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

```
datatype_new 'a ftree2 = FTLeaf2 'a | FTNode2 "'a ⇒ 'a ⇒ 'a ftree2"
```

```
primrec relabel_ft2 :: "('a ⇒ 'a) ⇒ 'a ftree2 ⇒ 'a ftree2" where
  "relabel_ft2 f (FTLeaf2 x) = FTLeaf2 (f x)" |
  "relabel_ft2 f (FTNode2 g) = FTNode2 (op ∘ (op ∘ (relabel_ft2 f)) g)"
```

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

“$\mathit{relabel\_ft2}\ f\ (\mathit{FTLeaf2}\ x) = \mathit{FTLeaf2}\ (f\ x)$” |
“$\mathit{relabel\_ft2}\ f\ (\mathit{FTNode2}\ g) = \mathit{FTNode2}\ (\lambda x\ y.\ \mathit{relabel\_ft2}\ f\ (g\ x\ y))$”

primrec $\mathit{subtree\_ft2}$ :: “$'a \Rightarrow 'a \Rightarrow 'a\ \mathit{ftree2} \Rightarrow 'a\ \mathit{ftree2}$” **where**
“$\mathit{subtree\_ft2}\ x\ y\ (\mathit{FTNode2}\ g) = g\ x\ y$”

3.1.5 Nested-as-Mutual Recursion

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

primrec
at_{ff} :: “$'a\ \mathit{tree}_{ff} \Rightarrow \mathit{nat}\ \mathit{list} \Rightarrow 'a$” **and**
ats_{ff} :: “$'a\ \mathit{tree}_{ff}\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}\ \mathit{list} \Rightarrow 'a$”
where
“$\mathit{at}_{ff}\ (\mathit{Node}_{ff}\ a\ \mathit{ts})\ \mathit{js} =$
$(\mathit{case}\ \mathit{js}\ \mathit{of}$
$[] \Rightarrow a$
$|\ j\ \#\ \mathit{js}' \Rightarrow \mathit{ats}_{ff}\ \mathit{ts}\ j\ \mathit{js}')$” |
“$\mathit{ats}_{ff}\ (t\ \#\ \mathit{ts})\ j =$
$(\mathit{case}\ j\ \mathit{of}$
$\mathit{Zero} \Rightarrow \mathit{at}_{ff}\ t$
$|\ \mathit{Suc}\ j' \Rightarrow \mathit{ats}_{ff}\ \mathit{ts}\ j')$”

Appropriate induction rules are generated as $\mathit{at}_{ff}.\mathit{induct}$, $\mathit{ats}_{ff}.\mathit{induct}$, and $\mathit{at}_{ff}\_\mathit{ats}_{ff}.\mathit{induct}$. The induction rules and the underlying recursors are generated on a per-need basis and are kept in a cache to speed up subsequent definitions.

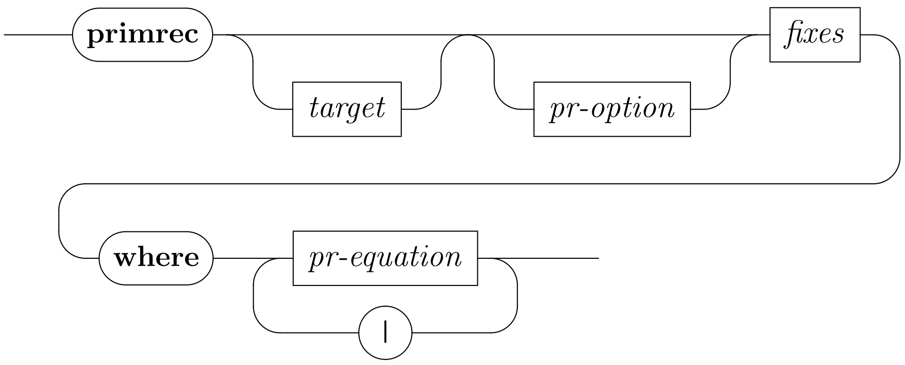
Here is a second example:

primrec
$\mathit{sum\_btree}$ :: “$('a::\{\mathit{zero},\mathit{plus}\})\ \mathit{btree} \Rightarrow 'a$” **and**
$\mathit{sum\_btree\_option}$ :: “$'a\ \mathit{btree}\ \mathit{option} \Rightarrow 'a$”
where
“$\mathit{sum\_btree}\ (\mathit{BNode}\ a\ \mathit{lt}\ \mathit{rt}) =$
$a + \mathit{sum\_btree\_option}\ \mathit{lt} + \mathit{sum\_btree\_option}\ \mathit{rt}$” |
“$\mathit{sum\_btree\_option}\ \mathit{None} = 0$” |
“$\mathit{sum\_btree\_option}\ (\mathit{Some}\ t) = \mathit{sum\_btree}\ t$”

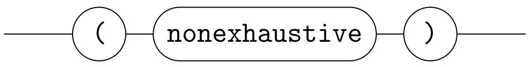
3.2 Command Syntax

3.2.1 primrec

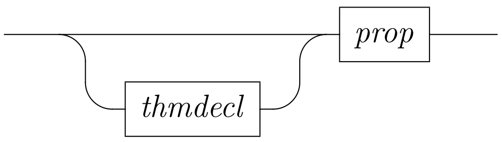
$$\textbf{primrec} \ : \ \mathit{local\_theory} \to \mathit{local\_theory}$$



pr-option



pr-equation



The **primrec** command introduces a set of mutually recursive functions over datatypes.

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

The optional target is optionally followed by the following option:

- 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.

3.3 Recursive Default Values for Selectors

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

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

1. Introduce a fully unspecified constant $un\_D_0 :: 'a$ using **consts**.
2. Define the datatype, specifying un_D_0 as the selector's default value.
3. Define the behavior of un_D_0 on values of the newly introduced datatype using the **overloading** command.
4. Derive the desired equation on un_D from the characteristic equations for un_D_0.

The following example illustrates this procedure:

```
consts termi₀ :: 'a

datatype_new ('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
```

3.4 Compatibility Issues

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

- *Some theorems have different names.* For $m > 1$ mutually recursive functions, $f_1\_\ldots\_f_m$*.simps* has been broken down into separate subcollections 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/BNF/Examples`. The *Archive of Formal Proofs* also includes some useful codatatypes, notably for lazy lists [6].

4.1 Introductory Examples

4.1.1 Simple Corecursion

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

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

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

```
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:

```
codatatype enat = EZero | ESuc enat
```

This type has exactly one infinite element, *ESuc* (*ESuc* (*ESuc* (…))), that represents ∞. In addition, it has finite values of the form *ESuc* (… (*ESuc* *EZero*)…).

Here is an example with many constructors:

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

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

4.1.2 Mutual Corecursion

The example below introduces a pair of *mutually corecursive* types:

codatatype *even_enat* = *Even_EZero* | *Even_ESuc odd_enat*
and *odd_enat* = *Odd_ESuc even_enat*

4.1.3 Nested Corecursion

The next examples feature *nested corecursion*:

codatatype $'a\ tree_{ii} = Node_{ii}\ (lbl_{ii}\colon 'a)\ (sub_{ii}\colon \text{“}'a\ tree_{ii}\ llist\text{”})$

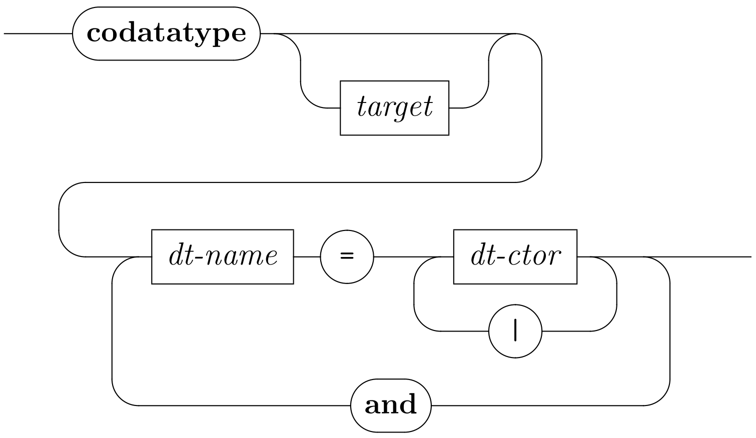
codatatype $'a\ tree_{is} = Node_{is}\ (lbl_{is}\colon 'a)\ (sub_{is}\colon \text{“}'a\ tree_{is}\ fset\text{”})$

codatatype *'a sm* = *SM* (*accept*: *bool*) (*trans*: "*'a* ⇒ *'a sm*")

4.2 Command Syntax

4.2.1 codatatype

codatatype : *local_theory* → *local_theory*



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

4.3 Generated Constants

Given a codatatype $('a_1, \ldots, 'a_m)\ t$ with $m > 0$ live type variables and n constructors $t.C_1, \ldots, t.C_n$, the same auxiliary constants are generated as for datatypes (Section 2.3), except that the recursor is replaced by a dual concept and no size function is produced:

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

The first two categories are exactly as for datatypes.

4.4.1 Coinductive Theorems

The coinductive theorems are listed below for *'a llist*:

t.***coinduct*** [*consumes* m, *case_names* $t_1 \ldots t_m$,
case_conclusion $D_1 \ldots D_n$, *coinduct* t]:
$[\![R\ \mathit{llist}\ \mathit{llist}';\ \bigwedge \mathit{llist}\ \mathit{llist}'.\ R\ \mathit{llist}\ \mathit{llist}' \Longrightarrow \mathit{lnull}\ \mathit{llist} = \mathit{lnull}\ \mathit{llist}' \land (\neg\ \mathit{lnull}\ \mathit{llist} \longrightarrow \neg\ \mathit{lnull}\ \mathit{llist}' \longrightarrow \mathit{lhd}\ \mathit{llist} = \mathit{lhd}\ \mathit{llist}' \land R\ (\mathit{ltl}\ \mathit{llist})\ (\mathit{ltl}\ \mathit{llist}'))]\!] \Longrightarrow \mathit{llist} = \mathit{llist}'$

t.***strong_coinduct*** [*consumes* m, *case_names* $t_1 \ldots t_m$,
case_conclusion $D_1 \ldots D_n$]:
$[\![R\ \mathit{llist}\ \mathit{llist}';\ \bigwedge \mathit{llist}\ \mathit{llist}'.\ R\ \mathit{llist}\ \mathit{llist}' \Longrightarrow \mathit{lnull}\ \mathit{llist} = \mathit{lnull}\ \mathit{llist}' \land (\neg\ \mathit{lnull}\ \mathit{llist} \longrightarrow \neg\ \mathit{lnull}\ \mathit{llist}' \longrightarrow \mathit{lhd}\ \mathit{llist} = \mathit{lhd}\ \mathit{llist}' \land (R\ (\mathit{ltl}\ \mathit{llist})\ (\mathit{ltl}\ \mathit{llist}') \lor \mathit{ltl}\ \mathit{llist} = \mathit{ltl}\ \mathit{llist}'))]\!] \Longrightarrow \mathit{llist} = \mathit{llist}'$

t.***rel_coinduct*** [*consumes* m, *case_names* $t_1 \ldots t_m$,
case_conclusion $D_1 \ldots D_n$, *coinduct pred*]:
$[\![P\ x\ y;\ \bigwedge \mathit{llist}\ \mathit{llist}'.\ P\ \mathit{llist}\ \mathit{llist}' \Longrightarrow \mathit{lnull}\ \mathit{llist} = \mathit{lnull}\ \mathit{llist}' \land (\neg\ \mathit{lnull}\ \mathit{llist} \longrightarrow \neg\ \mathit{lnull}\ \mathit{llist}' \longrightarrow R\ (\mathit{lhd}\ \mathit{llist})\ (\mathit{lhd}\ \mathit{llist}') \land P\ (\mathit{ltl}\ \mathit{llist})\ (\mathit{ltl}\ \mathit{llist}'))]\!] \Longrightarrow \mathit{llist\_all2}\ R\ x\ y$

t_1_…_t_m.***coinduct*** [*case_names* $t_1 \ldots t_m$, *case_conclusion* $D_1 \ldots D_n$]
t_1_…_t_m.***strong_coinduct*** [*case_names* $t_1 \ldots t_m$,
case_conclusion $D_1 \ldots D_n$]:
t_1_…_t_m.***rel_coinduct*** [*case_names* $t_1 \ldots t_m$,
case_conclusion $D_1 \ldots D_n$]:

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

t.***corec***:
$p\ a \Longrightarrow \mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a = \mathit{LNil}$
$\neg\ p\ a \Longrightarrow \mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a = \mathit{LCons}\ (g21\ a)\ (\mathit{if}\ q22\ a\ \mathit{then}\ g221\ a\ \mathit{else}\ \mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ (g222\ a))$

t.***corec_code*** [*code*]:
$\mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a = (\mathit{if}\ p\ a\ \mathit{then}\ \mathit{LNil}\ \mathit{else}\ \mathit{LCons}\ (g21\ a)\ (\mathit{if}\ q22\ a\ \mathit{then}\ g221\ a\ \mathit{else}\ \mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ (g222\ a)))$

t.***disc_corec***:
$p\ a \Longrightarrow \mathit{lnull}\ (\mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a)$
$\neg\ p\ a \Longrightarrow \neg\ \mathit{lnull}\ (\mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a)$

t.***disc_corec_iff*** [*simp*]:
$\mathit{lnull}\ (\mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a) = p\ a$
$(\neg\ \mathit{lnull}\ (\mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a)) = (\neg\ p\ a)$

t.***sel_corec*** [*simp*]:
$\neg\ p\ a \Longrightarrow \mathit{lhd}\ (\mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a) = g21\ a$
$\neg\ p\ a \Longrightarrow \mathit{ltl}\ (\mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ a) = (\mathit{if}\ q22\ a\ \mathit{then}\ g221\ a\ \mathit{else}\ \mathit{corec\_llist}\ p\ g21\ q22\ g221\ g222\ (g222\ a))$

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

t.***simps*** = *t.inject t.distinct t.case t.disc_corec_iff t.sel_corec t.map t.rel_inject t.rel_distinct t.set*

5 Defining Corecursive Functions

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

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

- The *destructor view* specifies f by implications of the form
$$\ldots \Longrightarrow \mathit{is\_C_j}\ (f\ x_1\ \ldots\ x_n)$$
and equations of the form
$$\mathit{un\_C_j i}\ (f\ x_1\ \ldots\ x_n) = \ldots$$
This style is popular in the coalgebraic literature.
- The *constructor view* specifies f by equations of the form
$$\ldots \Longrightarrow f\ x_1\ \ldots\ x_n = C_j\ \ldots$$
This style is often more concise than the previous one.
- The *code view* specifies f by a single equation of the form
$$f\ x_1\ \ldots\ x_n = \ldots$$
with restrictions on the format of the right-hand side. Lazy functional programming languages such as Haskell support a generalized version of this style.

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

5.1 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/BNF_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:

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

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

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

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

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

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

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

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

simps_of_case *lappend_simps*: *lappend.code*

This generates the lemma collection *lappend_simps*:

$$\begin{aligned} \mathit{lappend}\ \mathit{LNil}\ \mathit{ys} &= \mathit{ys} \\ \mathit{lappend}\ (\mathit{LCons}\ \mathit{xa}\ x)\ \mathit{ys} &= \mathit{LCons}\ \mathit{xa}\ (\mathit{lappend}\ x\ \mathit{ys}) \end{aligned}$$

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

```
primcorec infty :: enat where
  "infty = ESuc 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 datatypes is unsurprising:

```
primcorec
  even_infty :: even_enat and
  odd_infty :: odd_enat
where
  "even_infty = Even_ESuc odd_infty" |
  "odd_infty = Odd_ESuc 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*$_{ii}$) or as a finite set (*tree*$_{is}$). They rely on the map functions of the nesting type constructors to lift the corecursive calls:

primcorec *iterate*$_{ii}$ :: "('*a* ⇒ '*a llist*) ⇒ '*a* ⇒ '*a tree*$_{ii}$" **where**
  "*iterate*$_{ii}$ *g x* = *Node*$_{ii}$ *x* (*lmap* (*iterate*$_{ii}$ *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*$_{ii}$. The term *g x* constructs an '*a llist* value, which is turned into an '*a tree*$_{ii}$ *llist* value using *lmap*.

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

primcorec *tree*$_{ii}$*_of_stream* :: "'*a stream* ⇒ '*a tree*$_{ii}$" **where**
  "*tree*$_{ii}$*_of_stream s* =
    Node$_{ii}$ (*shd s*) (*lmap tree*$_{ii}$*_of_stream* (*LCons* (*stl s*) *LNil*))"

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

primcorec *tree*$_{ii}$*_of_stream* :: "'*a stream* ⇒ '*a tree*$_{ii}$" **where**
  "*tree*$_{ii}$*_of_stream s* =
    Node$_{ii}$ (*shd s*) (*LCons* (*tree*$_{ii}$*_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:

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* ∘ δ *q*)"

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

primcorec *sm_of_dfa* :: "('*q* ⇒ '*a* ⇒ '*q*) ⇒ '*q set* ⇒ '*q* ⇒ '*a sm*" **where**
  "*sm_of_dfa* δ *F q* = *SM* (*q* ∈ *F*) (λ*a*. *sm_of_dfa* δ *F* (δ *q a*))"

primcorec *empty_sm* :: "'*a sm*" **where**

```
  "empty_sm = SM False (λ_. empty_sm)"

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

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

For recursion through curried *n*-ary functions, *n* applications of *op* ∘ 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) (op ∘ (op ∘ (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_ii :: "('a ⇒ 'a llist) ⇒ 'a ⇒ 'a tree_ii" and
  iterates_ii :: "('a ⇒ 'a llist) ⇒ 'a llist ⇒ 'a tree_ii llist"
where
  "iterate_ii g x = Node_ii x (iterates_ii g (g x))" |
  "iterates_ii g xs =
     (case xs of
        LNil ⇒ LNil
      | LCons x xs' ⇒ LCons (iterate_ii g x) (iterates_ii g xs'))"
```

Coinduction rules are generated as *iterate*$_{ii}$*.coinduct*, *iterates*$_{ii}$*.coinduct*, and *iterate*$_{ii}$*_iterates*$_{ii}$*.coinduct* and analogously for *strong_coinduct*. These rules and the underlying corecursors are generated on a per-need basis and are kept in a cache to speed up subsequent definitions.

5.1.5 Constructor View

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

Here is an example where there is a difference:

```
primcorec lappend :: "'a llist ⇒ 'a llist ⇒ 'a llist" where
  "lnull xs ⟹ lnull ys ⟹ lappend xs ys = LNil" |
  "_ ⟹ lappend xs ys = LCons (lhd (if lnull xs then ys else xs))
     (if xs = LNil then ltl ys else lappend (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 slighlty more involved than the code equation. Recall the code view version presented in Section 5.1.1. The constructor view requires us to analyze the second argument (*ys*). The code equation generated from the constructor view also suffers from this.

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

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

Since there is no sequentiality, we can apply the equation for *Choice* without having first to discharge $n \mathit{mod}\ 4 \neq 0$, $n \mathit{mod}\ 4 \neq 1$, and $n \mathit{mod}\ 4 \neq 2$. The price to pay for this elegance is that we must discharge exclusivity proof obligations, one for each pair of conditions $(n \mathit{mod}\ 4 = i,\ n \mathit{mod}\ 4 = j)$ 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:

primcorec *literate* :: "$('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ llist$" **where**
  "$\neg\ lnull\ (literate\ \_\ x)$" |
  "$lhd\ (literate\ \_\ x) = x$" |
  "$ltl\ (literate\ g\ x) = literate\ g\ (g\ x)$"

primcorec *siterate* :: "$('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ stream$" **where**
  "$shd\ (siterate\ \_\ x) = x$" |
  "$stl\ (siterate\ g\ x) = siterate\ g\ (g\ x)$"

primcorec *every_snd* :: "$'a\ stream \Rightarrow 'a\ stream$" **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:

primcorec *lappend* :: "$'a\ llist \Rightarrow 'a\ llist \Rightarrow 'a\ llist$" **where**
  "$lnull\ xs \Longrightarrow lnull\ ys \Longrightarrow lnull\ (lappend\ xs\ ys)$" |
  "$lhd\ (lappend\ xs\ ys) = lhd\ (if\ lnull\ xs\ then\ ys\ else\ xs)$" |
  "$ltl\ (lappend\ xs\ ys) = (if\ xs = LNil\ then\ ltl\ ys\ else\ lappend\ (ltl\ xs)\ ys)$"

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

  "$\_ \Longrightarrow \neg\ lnull\ (lappend\ xs\ ys)$"

to the specification. The generated selector theorems are conditional.

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

primcorec
  random_process :: "$'a\ stream \Rightarrow (int \Rightarrow int) \Rightarrow int \Rightarrow 'a\ process$"
where
  "$n\ mod\ 4 = 0 \Longrightarrow random\_process\ s\ f\ n = Fail$" |

"n *mod* $4 = 1 \Longrightarrow$ *is_Skip* (*random_process s f n*)" |
"n *mod* $4 = 2 \Longrightarrow$ *is_Action* (*random_process s f n*)" |
"n *mod* $4 = 3 \Longrightarrow$ *is_Choice* (*random_process s f n*)" |
"*cont* (*random_process s f n*) = *random_process s f* (*f n*)" *of Skip* |
"*prefix* (*random_process s f n*) = *shd s*" |
"*cont* (*random_process s f n*) = *random_process* (*stl s*) *f* (*f n*)" *of Action* |
"*left* (*random_process s f n*) = *random_process* (*every_snd s*) *f* (*f n*)" |
"*right* (*random_process s f n*) = *random_process* (*every_snd* (*stl s*)) *f* (*f n*)"

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

Here are more examples to conclude:

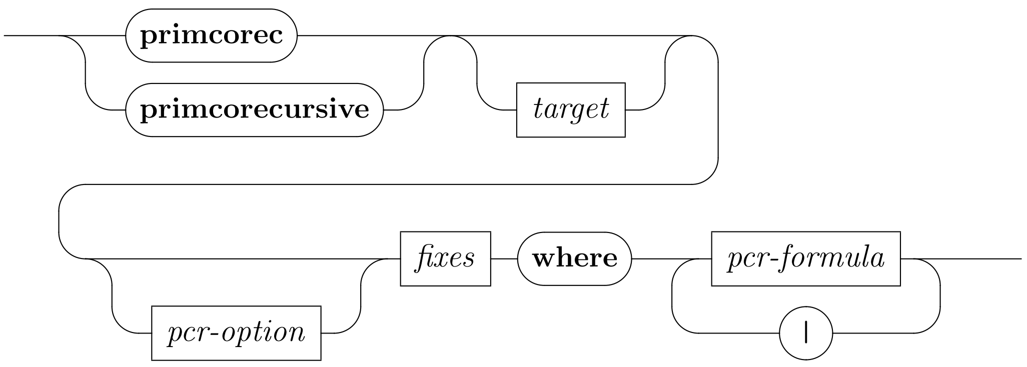
primcorec
even_infty :: *even_enat* **and**
odd_infty :: *odd_enat*
where
"*even_infty* $\neq$ *Even_EZero*" |
"*un_Even_ESuc even_infty* = *odd_infty*" |
"*un_Odd_ESuc odd_infty* = *even_infty*"

primcorec $iterate_{ii}$ :: "('*a* $\Rightarrow$ '*a llist*) $\Rightarrow$ '*a* $\Rightarrow$ '*a* $tree_{ii}$" **where**
"lbl_{ii} ($iterate_{ii}$ *g x*) = *x*" |
"sub_{ii} ($iterate_{ii}$ *g x*) = *lmap* ($iterate_{ii}$ *g*) (*g x*)"

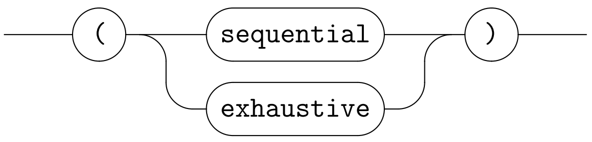
5.2 Command Syntax

5.2.1 primcorec and primcorecursive

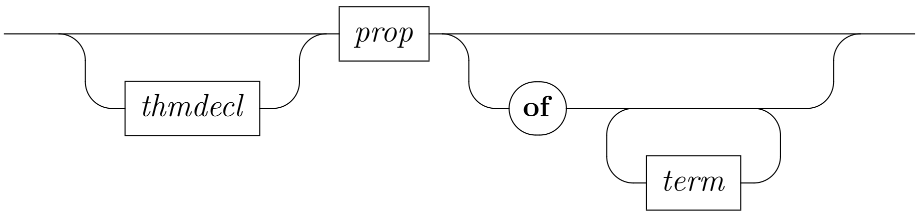
primcorec	:	*local_theory* → *local_theory*
primcorecursive	:	*local_theory* → *proof*(*prove*)



pcr-option



pcr-formula



The **primcorec** and **primcorecursive** commands introduce a set of mutually corecursive functions over codatatypes.

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

The optional target is optionally followed by one or both of the following options:

- The *sequential* option indicates that the conditions in specifications expressed using the constructor or destructor view are to be interpreted sequentially.
- The *exhaustive* option indicates that the conditions in specifications expressed using the constructor or destructor view cover all possible cases.

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

6 Introducing Bounded Natural Functors

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

6.1 Bounded Natural Functors

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

An n-ary BNF is a type constructor equipped with a map function (functorial action), n set functions (natural transformations), and an infinite cardinal bound that satisfy certain properties. For example, *'a llist* is a unary BNF. Its relator *llist_all2* :: *('a ⇒ 'b ⇒ bool) ⇒ 'a llist ⇒ 'b llist ⇒ 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 **datatype_new** and **codatatype** are automatically registered as BNFs. In addition, a number of old-style datatypes and non-free types are preregistered.

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

6.2 Introductory Examples

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

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

```
typedef ('d, 'a) fn = "UNIV :: ('d ⇒ 'a) set"
by simp

setup_lifting type_definition_fn

lift_definition map_fn :: "('a ⇒ 'b) ⇒ ('d, 'a) fn ⇒ ('d, 'b) fn" is "op ∘" .
lift_definition set_fn :: "('d, 'a) fn ⇒ 'a set" is range .

lift_definition
  rel_fn :: "('a ⇒ 'b ⇒ bool) ⇒ ('d, 'a) fn ⇒ ('d, 'b) fn ⇒ bool"
is
  "rel_fun (op =)" .

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

print_theorems
print_bnfs
```

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

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

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

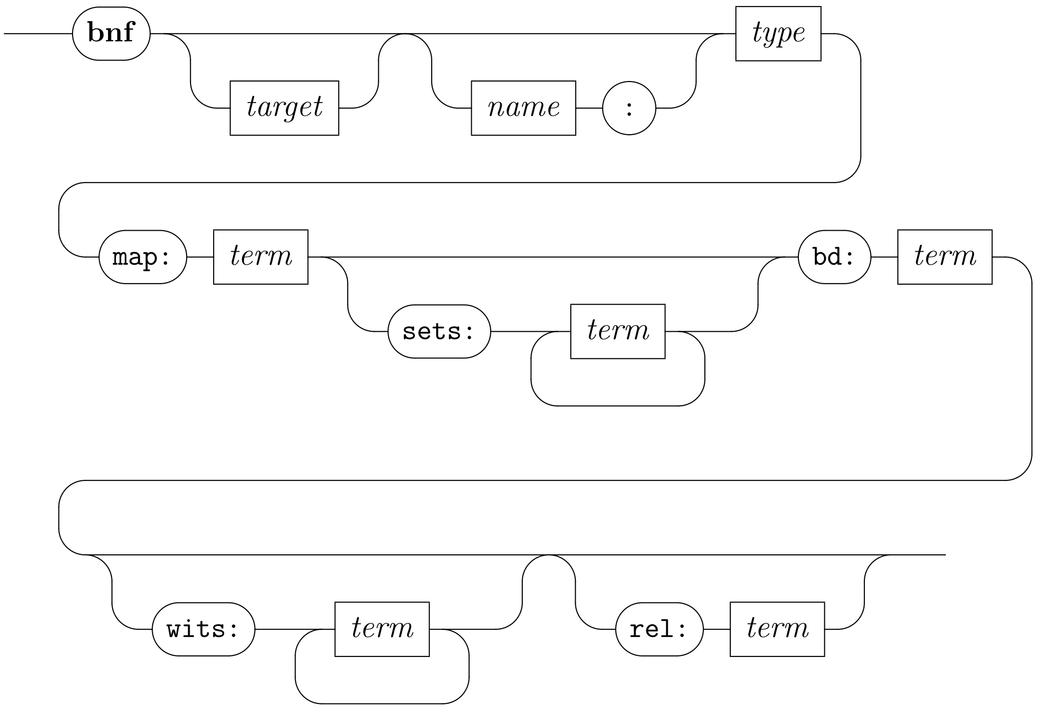
```
bnf_axiomatization (setA: 'a, setB: 'b, setC: 'c) F
  [wits: "'a ⇒ ('a, 'b, 'c) F"]

print_theorems
print_bnfs
```

6.3 Command Syntax

6.3.1 bnf

bnf : *local_theory → proof(prove)*

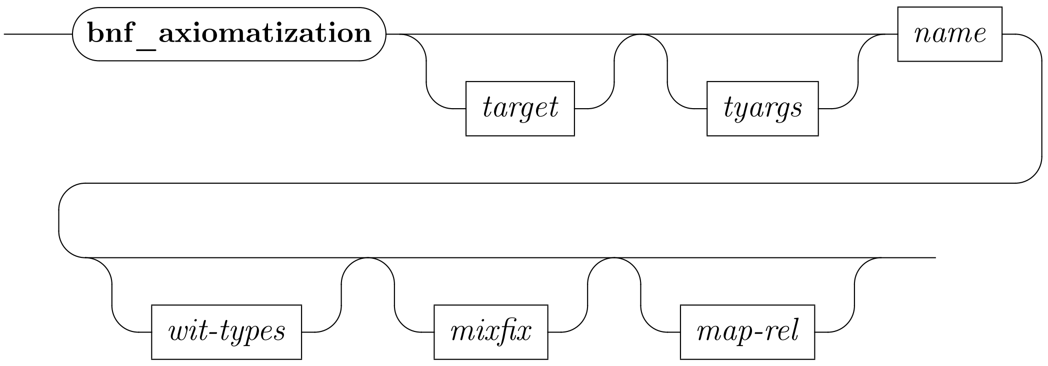


The **bnf** command registers an existing type as a bounded natural functor (BNF). The type must be equipped with an appropriate map function (functorial action). In addition, custom set functions, relators, and nonemptiness witnesses can be specified; otherwise, default versions are used.

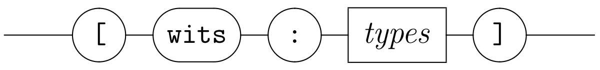
The syntactic entity *target* can be used to specify a local context, *type* denotes a HOL type, and *term* denotes a HOL term [9].

6.3.2 bnf_axiomatization

bnf_axiomatization : *local_theory* → *local_theory*



wit-types



The **bnf_axiomatization** command declares a new type and associated constants (map, set, relator, and cardinal bound) and asserts the BNF properties for these constants as axioms.

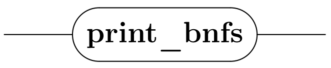
The syntactic entity *target* can be used to specify a local context, *name* denotes an identifier, *typefree* denotes fixed type variable (*'a*, *'b*, ...), and *mixfix* denotes the usual parenthesized mixfix notation [9].

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

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

6.3.3 print_bnfs

print_bnfs : *local_theory* →



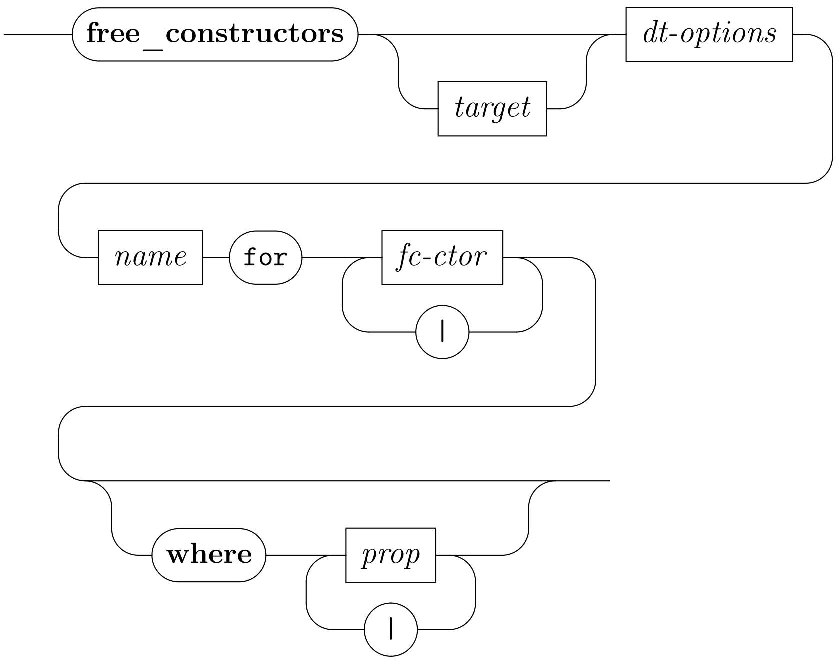
7 Deriving Destructors and Theorems for Free Constructors

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

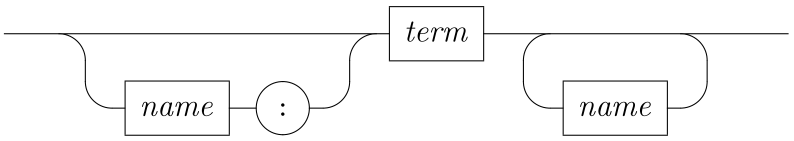
7.1 Command Syntax

7.1.1 free_constructors

free_constructors : *local_theory* → *proof*(*prove*)



fc-ctor



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

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

The syntax resembles that of **datatype_new** 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**.

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, many of which have yet to be implemented. Florian Haftmann and Christian Urban provided general advice on Isabelle and package writing. Stefan Milius and Lutz Schröder found an elegant proof that eliminated one of the BNF proof obligations. Andreas Lochbihler and Christian Sternagel suggested many textual improvements to this tutorial.

References

[1] S. Berghofer and M. Wenzel. Inductive datatypes in HOL — lessons learned in Formal-Logic Engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[2] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

[3] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. `http://www21.in.tum.de/~blanchet/wit.pdf`, 2014.

[4] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS 78. Springer, 1979.

[5] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. `http://isabelle.in.tum.de/doc/functions.pdf`.

[6] A. Lochbihler. Coinductive. In G. Klein, T. Nipkow, and L. C. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sourceforge.net/entries/Coinductive.shtml`, Feb. 2010.

[7] L. Panny, J. C. Blanchette, and D. Traytel. Primitively (co)recursive definitions for Isabelle/HOL. In *Isabelle Workshop 2014*, 2014.

[8] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS 2012*, pages 596–605. IEEE, 2012.

[9] M. Wenzel. *The Isabelle/Isar Reference Manual*. `http://isabelle.in.tum.de/doc/isar-ref.pdf`.