# Defining (Co)datatypes in Isabelle/HOL

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

5 December 2013

**Abstract**

This tutorial describes how to use the new package for defining datatypes and codatatypes in Isabelle/HOL. The package provides five main commands: **datatype_new**, **codatatype**, **primrec_new**, **primcorecursive**, and **primcorec**. The commands suffixed by **_new** are intended to subsume, and eventually replace, the corresponding commands from the old datatype package.

# 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 [8]; 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** $'a\ tree_{fs} = Node_{fs}\ (lbl_{fs}\colon 'a)\ (sub_{fs}\colon \text{“}'a\ tree_{fs}\ fset\text{”})$

Another strong point is the support for local definitions:

**context** *linorder*
**begin**
**datatype_new** $flag = Less \mid Eq \mid Greater$
**end**

The package also provides some convenience, notably automatically generated discriminators and selectors.

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

**codatatype** $'a\ llist = LNil \mid LCons\ 'a\ \text{“}'a\ llist\text{”}$

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

**datatype_new** $'a\ tree_{ff} = Node_{ff}\ 'a\ \text{“}'a\ tree_{ff}\ list\text{”}$
**datatype_new** $'a\ tree_{fi} = Node_{fi}\ 'a\ \text{“}'a\ tree_{fi}\ llist\text{”}$
**codatatype** $'a\ tree_{if} = Node_{if}\ 'a\ \text{“}'a\ tree_{if}\ list\text{”}$
**codatatype** $'a\ tree_{ii} = Node_{ii}\ 'a\ \text{“}'a\ tree_{ii}\ llist\text{”}$

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

To use the package, it is necessary to import the *BNF* theory, which can be precompiled into the `HOL-BNF` image. The following commands show how to launch jEdit/PIDE with the image loaded and how to build the image without launching jEdit:

```
isabelle jedit -l HOL-BNF
isabelle build -b HOL-BNF
```

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's metatheory is described in a pair of papers [3, 7]. 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_new**, **fun**, and **function**.
- 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, "Registering Bounded Natural Functors," explains how to use the **bnf** command to register arbitrary type constructors as BNFs.
- Section 7, "Deriving Destructors and Theorems for Free Constructors," explains how to use the command **wrap_free_constructors** to derive destructor constants and theorems for freely generated types, as performed internally by **datatype_new** and **codatatype**.

The commands **datatype_new** and **primrec_new** are expected to replace **datatype** and **primrec** 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.

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

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

**Warning:** This tutorial and the package it describes are under construction. Please forgive their appearance. Should you have suggestions or comments regarding either, please let the authors know.

# 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** *trool* = *Truue* | *Faalse* | *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** $'a$ *option* = *None* | *Some* $'a$

The constructors are *None* :: $'a$ *option* and *Some* :: $'a \Rightarrow 'a$ *option*.

The next example has three type parameters:

**datatype_new** $('a, 'b, 'c)$ *triple* = *Triple* $'a$ $'b$ $'c$

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

**datatype_new** $('a, 'b, 'c)$ $triple_u$ = $Triple_u$ "$'a * 'b * 'c$"

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_new** *nat* = *Zero* | *Suc nat*

Lists were shown in the introduction. Terminated lists are a variant:

**datatype_new** (*'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_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 *Datatypes.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* = *Wrong* "*'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*) *fn* = *Fn* "*'a* ⇒ *'b*"
**datatype_new** *'a also_wrong* = *Also_Wrong* "(*'a also_wrong*, *'a*) *fn*"

This is legal:

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

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 $('a, 'b)$ *fn*, 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.

### 2.1.5 Custom Names and Syntaxes

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 traditional names *set*, *map*, *list_all2*, *null*, *hd*, and *tl* override the default names *list_set*, *list_map*, *list_rel*, *is_Nil*, *un_Cons1*, and *un_Cons2*:

**datatype_new** (*set*: *'a*) *list* (*map*: *map* *rel*: *list_all2*) =
  *null*: *Nil* (*defaults* *tl*: *Nil*)
| *Cons* (*hd*: *'a*) (*tl*: "*'a list*")

The command introduces a discriminator *null* and a pair of selectors *hd* and *tl* characterized as follows:

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

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

The *defaults* clause following the *Nil* constructor specifies a default value for selectors associated with other constructors. 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 specified by entering "=" instead of the identifier *null*. Although this may look appealing, 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* (*map*: *map* *rel*: *list_all2*) =

*null*: *Nil* (“[]”)
| *Cons* (*hd*: ′*a*) (*tl*: “′*a list*”) (**infixr** “#” 65)

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

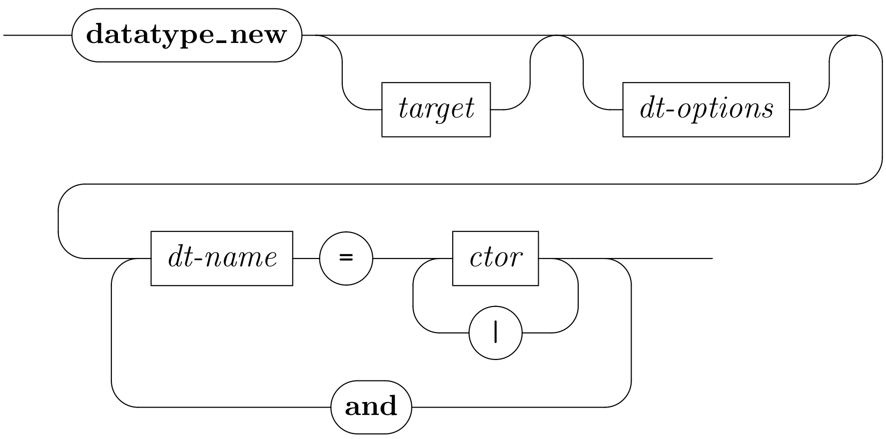
**syntax** “_*list*” :: “*args* ⇒ ′*a list*” (“[(_)]”)

**translations**
“[*x*, *xs*]” == “*x* # [*xs*]”
“[*x*]” == “*x* # []”

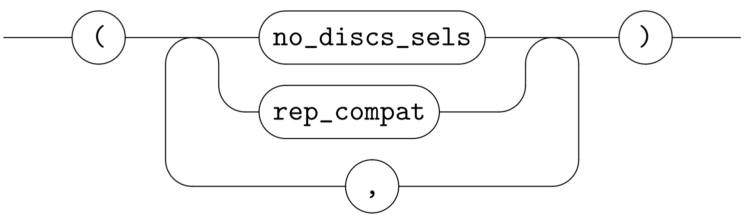
## 2.2 Command Syntax

### 2.2.1 datatype_new

**datatype_new** : *local_theory* → *local_theory*




*dt-options*

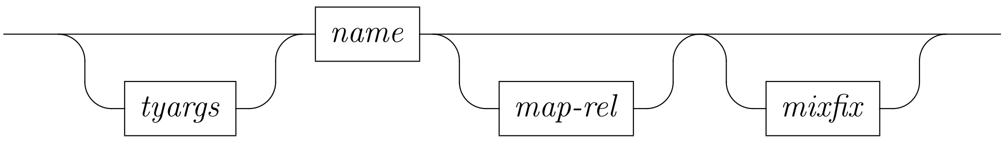



The syntactic entity *target* can be used to specify a local context—e.g., (*in linorder*). It is documented in the Isar reference manual [8]. The optional target is optionally followed by datatype-specific options:

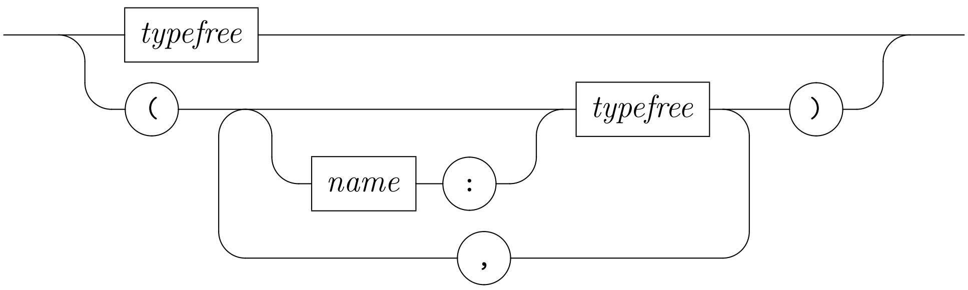
- The *no_discs_sels* option indicates that no discriminators or selectors should be generated.
- The *rep_compat* option indicates that the generated names should contain optional (and normally not displayed) "*new*." components to prevent clashes with a later call to **rep_datatype**. See Section 2.5 for details.

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

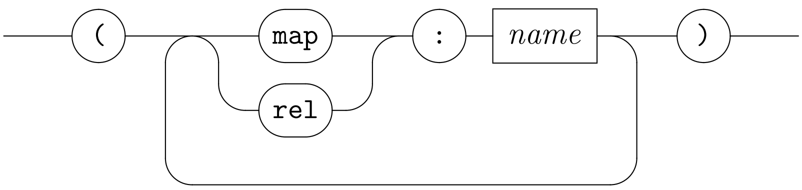
*dt-name*




*tyargs*



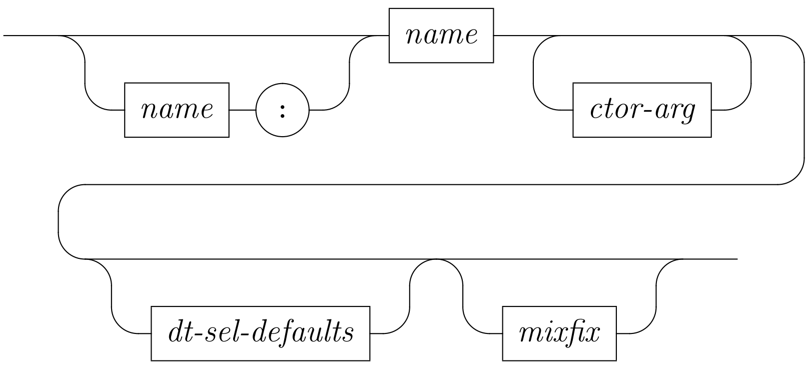

*map-rel*




The syntactic entity *name* denotes an identifier, *typefree* denotes fixed type variable (*'a*, *'b*, ...), and *mixfix* denotes the usual parenthesized mixfix notation. They are documented in the Isar reference manual [8].

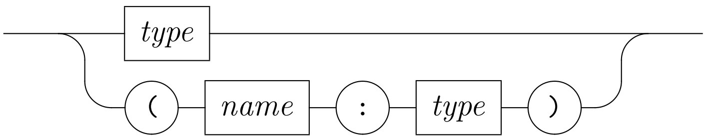
The optional names preceding the type variables allow to override the default names of the set functions ($t\_set1$, ..., $t\_setM$). Inside a mutually recursive specification, all defined datatypes must mention exactly the same type variables in the same order.

*ctor*



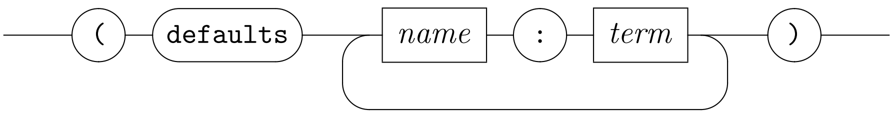

The main constituents of a constructor specification are the name of the constructor and the list of its argument types. An optional discriminator name can be supplied at the front to override the default name ($t.is\_C_j$).

*ctor-arg*




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.

*dt-sel-defaults*




Given a constructor $C :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_p \Rightarrow \sigma$, default values can be specified for any selector $un\_D :: \sigma \Rightarrow \tau$ associated with other constructors. The specified default value must be of type $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_p \Rightarrow \tau$ (i.e., it may depend on $C$'s arguments).

### 2.2.2 datatype_new_compat

**datatype_new_compat** : *local_theory* → *local_theory*


The old datatype package provides some functionality that is not yet replicated in the new package:

- It is integrated with **fun** and **function** [5], Nitpick [2], Quickcheck, and other packages.
- It is extended by various add-ons, notably to produce instances of the *size* function.

New-style datatypes can in most cases be registered as old-style datatypes using **datatype_new_compat**. The *names* argument is a space-separated list of type names that are mutually recursive. For example:

**datatype_new_compat** *even_nat odd_nat*

**thm** *even_nat_odd_nat.size*

**ML** ⟪ *Datatype_Data.get_info* @{*theory*} @{*type_name even_nat*} ⟫

A few remarks concern nested recursive datatypes only:

- The old-style, nested-as-mutual induction rule, iterator theorems, 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 the command does not support this yet (and there is currently no way to register old-style datatypes as new-style datatypes).

An alternative to **datatype_new_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* (rendered using the familiar *case–of* syntax)
- Discriminators: $t.is\_C_1,\ \ldots,\ t.is\_C_n$
- Selectors: $t.un\_C_1 1, \ldots, t.un\_C_1 k_1,$
  $\vdots$
  $t.un\_C_n 1,\ \ldots,\ t.un\_C_n k_n.$
- Set functions (or natural transformations): *t_set*1, ..., *t_setm*
- Map function (or functorial action): *t_map*
- Relator: *t_rel*
- Iterator: *t_fold*
- Recursor: *t_rec*

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

## 2.4 Generated Theorems

The characteristic theorems generated by **datatype_new** are grouped in three broad categories:

- The *free constructor theorems* are properties about the constructors and destructors that can be derived for any freely generated type. Internally, the derivation is performed by **wrap_free_constructors**.
- The *functorial theorems* are properties of datatypes related to their BNF nature.
- The *inductive theorems* 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 first subgroup of properties is concerned with the constructors. They are listed below for $'a\ list$:

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

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

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

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

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

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

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

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

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

The third and last subgroup revolves around discriminators and selectors:

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

$t$.***discI***:
*list* = [] ⟹ *null list*
*list* = $x21$ # $x22$ ⟹ ¬ *null list*

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

$t$.***collapse*** [*simp*]:
*null list* ⟹ *list* = []
¬ *null list* ⟹ *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* ⟹ ¬ *nonnull list*
*nonnull list* ⟹ ¬ *null list*

$t$.***disc_exhaust*** [*case_names* $C_1 \ldots C_n$]:
⟦*null list* ⟹ $P$; ¬ *null list* ⟹ $P$⟧ ⟹ $P$

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

$t$.***expand***:
⟦*null list* = *null list'*; ⟦¬ *null list*; ¬ *null list'*⟧ ⟹ *hd list* = *hd list'* ∧ *tl list* = *tl list'*⟧ ⟹ *list* = *list'*

$t$.***sel_split***:
$P$ (*case list of* [] ⇒ $f1$ | $x$ # $xa$ ⇒ $f2$ $x$ $xa$) = ((*list* = [] ⟶ $P$ $f1$) ∧ (*list* = *hd list* # *tl list* ⟶ $P$ ($f2$ (*hd list*) (*tl list*))))

$t$.***sel_split_asm***:
$P$ (*case list of* [] ⇒ $f1$ | $x$ # $xa$ ⇒ $f2$ $x$ $xa$) = (¬ (*list* = [] ∧ ¬ $P$ $f1$ ∨ *list* = *hd list* # *tl list* ∧ ¬ $P$ ($f2$ (*hd list*) (*tl list*))))

$t$.***case_conv_if***:
(*case list of* [] ⇒ $f1$ | $x$ # $xa$ ⇒ $f2$ $x$ $xa$) = (*if null list then* $f1$ *else* $f2$ (*hd list*) (*tl list*))

### 2.4.2 Functorial Theorems

The BNF-related theorem are as follows:

$t.$***set*** [*simp*, *code*]:
 *set* [] = {}
 *set* ($x21a$ # $x22a$) = {$x21a$} ∪ *set* $x22a$

$t.$***map*** [*simp*, *code*]:
 *map fi* [] = []
 *map fi* ($x21a$ # $x22a$) = *fi* $x21a$ # *map fi* $x22a$

$t.$***rel_inject*** [*simp*, *code*]:
 *list_all2* $R$ [] []
 *list_all2* $R$ ($x21a$ # $x22a$) ($y21$ # $y22a$) = ($R$ $x21a$ $y21$ ∧ *list_all2* $R$ $x22a$ $y22a$)

$t.$***rel_distinct*** [*simp*, *code*]:
 ¬ *list_all2* $R$ [] ($y21$ # $y22a$)
 ¬ *list_all2* $R$ ($y21$ # $y22a$) []

### 2.4.3 Inductive Theorems

The inductive theorems are as follows:

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

$t_1$_…_$t_m$.***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.

$t.$***fold*** [*simp*, *code*]:
 *list_fold* $f1$ $f2$ [] = $f1$
 *list_fold* $f1$ $f2$ ($x21$ # $x22$) = $f2$ $x21$ (*list_fold* $f1$ $f2$ $x22$)

$t.$***rec*** [*simp*, *code*]:
 *list_rec* $f1$ $f2$ [] = $f1$
 *list_rec* $f1$ $f2$ ($x21$ # $x22$) = $f2$ $x21$ $x22$ (*list_rec* $f1$ $f2$ $x22$)

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

$t.$***simps*** = $t.$*inject* $t.$*distinct* $t.$*case* $t.$*rec* $t.$*fold* $t.$*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. Little has been done so far in this direction. Whenever possible, it is recommended to use **datatype_new_compat** or **rep_datatype** to register new-style datatypes as old-style datatypes.
- *The recursor t_rec has a different signature for nested recursive datatypes.* In the old package, nested recursion was internally reduced to mutual recursion. This reduction was visible in the type of the recursor, used by **primrec**. In the new package, nested recursion is handled in a more modular fashion. The old-style recursor can be generated on demand using **primrec_new**, as explained in Section 3.1.5, if the recursion is via new-style datatypes.
- *Accordingly, the induction principle is different for nested recursive datatypes.* Again, the old-style induction principle can be generated on demand using **primrec_new**, as explained in Section 3.1.5, if the recursion is via new-style datatypes.
- *The internal constructions are completely different.* Proof texts that unfold the definition of constants introduced by **datatype** will be difficult to port.
- *A few theorems have different names.* The properties $t.cases$ and $t.recs$ have been renamed $t.case$ and $t.rec$. For non-mutually recursive datatypes, $t.inducts$ is available as $t.induct$. For $m > 1$ mutually recursive datatypes, $t_1\_\ldots\_t_m.inducts(i)$ has been renamed $t_i.induct$.
- *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 **wrap_free_constructors** (Section 7).

# 3 Defining Recursive Functions

Recursive functions over datatypes can be specified using **primrec_new**, which supports primitive recursion, or using the more general **fun** and **function** commands. Here, the focus is on **primrec_new**; 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_new** *bool_of_trool* :: "*trool* $\Rightarrow$ *bool*" **where**
"*bool_of_trool Faalse* $\longleftrightarrow$ *False*" |
"*bool_of_trool Truue* $\longleftrightarrow$ *True*"

**primrec_new** *the_list* :: "$'a$ *option* $\Rightarrow$ $'a$ *list*" **where**
"*the_list None* = []" |
"*the_list* (*Some a*) = [*a*]"

**primrec_new** *the_default* :: "$'a \Rightarrow 'a$ *option* $\Rightarrow 'a$" **where**
"*the_default d None* = *d*" |
"*the_default* _ (*Some a*) = *a*"

**primrec_new** *mirrror* :: "$('a, 'b, 'c)$ *triple* $\Rightarrow$ $('c, 'b, 'a)$ *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_new** *replicate* :: "*nat* $\Rightarrow 'a \Rightarrow 'a$ *list*" **where**
"*replicate Zero* _ = []" |

"*replicate* (*Suc n*) *x* = *x* # *replicate n x*"

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

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

The next example is not primitive recursive, but it can be defined easily using **fun**. The **datatype_new_compat** command is needed to register new-style datatypes for use with **fun** and **function** (Section 2.2.2):

**datatype_new_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_new**
*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_new**
$eval_e$ :: "('*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**
"$eval_e\ \gamma\ \xi$ (*Term t*) = $eval_t\ \gamma\ \xi\ t$" |
"$eval_e\ \gamma\ \xi$ (*Sum t e*) = $eval_t\ \gamma\ \xi\ t + eval_e\ \gamma\ \xi\ e$" |
"$eval_t\ \gamma\ \xi$ (*Factor f*) = $eval_f\ \gamma\ \xi\ f$" |
"$eval_t\ \gamma\ \xi$ (*Prod f t*) = $eval_f\ \gamma\ \xi\ f + eval_t\ \gamma\ \xi\ t$" |
"$eval_f\ \gamma$ _ (*Const a*) = $\gamma\ a$" |
"$eval_f$ _ $\xi$ (*Var b*) = $\xi\ b$" |
"$eval_f\ \gamma\ \xi$ (*Expr e*) = $eval_e\ \gamma\ \xi\ e$"

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_new 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 *option_map*:

```
primrec_new sum_btree :: "('a::{zero,plus}) btree ⇒ 'a" where
  "sum_btree (BNode a lt rt) =
     a + the_default 0 (option_map sum_btree lt) +
      the_default 0 (option_map 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_new ftree_map :: "('a ⇒ 'a) ⇒ 'a ftree ⇒ 'a ftree" where
  "ftree_map f (FTLeaf x) = FTLeaf (f x)" |
  "ftree_map f (FTNode g) = FTNode (ftree_map f ∘ g)"
```

(No such map function is defined by the package because the type variable *'a* is dead in *'a ftree*.)

Using **fun** or **function**, recursion through functions can be expressed using λ-expressions and function application rather than through composition. For example:

```
datatype_new_compat ftree
```

**function** *ftree_map* :: "('a ⇒ 'a) ⇒ 'a ftree ⇒ 'a ftree" **where**
"*ftree_map f* (*FTLeaf x*) = *FTLeaf* (*f x*)" |
"*ftree_map f* (*FTNode g*) = *FTNode* (λ*x*. *ftree_map f* (*g x*))"
**by** *auto* (*metis ftree.exhaust*)

### 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_new**
$at_{ff}$ :: "'a $tree_{ff}$ ⇒ nat list ⇒ 'a" **and**
$ats_{ff}$ :: "'a $tree_{ff}$ list ⇒ nat ⇒ nat list ⇒ 'a"
**where**
"$at_{ff}$ ($Node_{ff}$ a ts) js =
(case js of
[] ⇒ a
| j # js' ⇒ $ats_{ff}$ ts j js')" |
"$ats_{ff}$ (t # ts) j =
(case j of
Zero ⇒ $at_{ff}$ t
| Suc j' ⇒ $ats_{ff}$ ts j')"

Appropriate induction principles are generated under the names $at_{ff}$*.induct*, $ats_{ff}$*.induct*, and $at_{ff}$_$ats_{ff}$*.induct*.

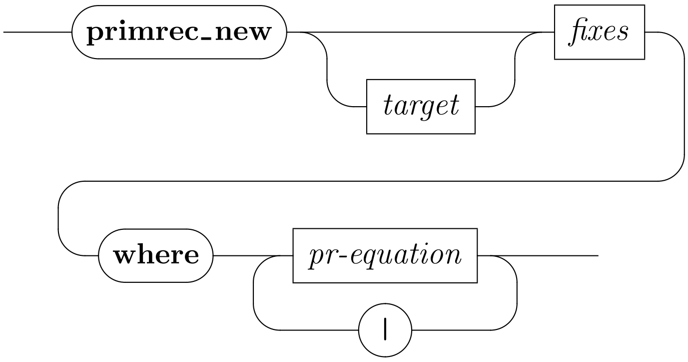
Here is a second example:

**primrec_new**
*sum_btree* :: "('a::{zero,plus}) btree ⇒ 'a" **and**
*sum_btree_option* :: "'a btree option ⇒ 'a"
**where**
"*sum_btree* (*BNode a lt rt*) =
a + *sum_btree_option lt* + *sum_btree_option rt*" |
"*sum_btree_option None* = 0" |
"*sum_btree_option* (*Some t*) = *sum_btree t*"

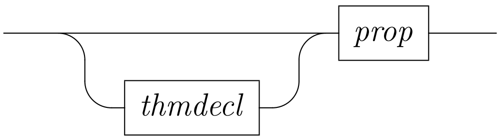
## 3.2 Command Syntax

### 3.2.1 primrec_new

**primrec_new** : *local_theory* → *local_theory*




*pr-equation*




## 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 produces a chicken-and-egg situation that may seem unsolvable, 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 fairly elegant workaround that relies on overloading and that avoids 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** $termi_0 :: 'a$

**datatype_new** $('a, 'b)$ *tlist* =
  *TNil* (*termi*: $'b$) (*defaults ttl*: *TNil*)
| *TCons* (*thd*: $'a$) (*ttl* : "$('a, 'b)$ *tlist*") (*defaults termi*: "$\lambda$_ *xs*. $termi_0$ *xs*")

**overloading**
  $termi_0 \equiv$ "$termi_0$ :: $('a, 'b)$ *tlist* $\Rightarrow$ $'b$"
**begin**
**primrec_new** $termi_0$ :: "$('a, 'b)$ *tlist* $\Rightarrow$ $'b$" **where**
  "$termi_0$ (*TNil y*) = *y*" |
  "$termi_0$ (*TCons x xs*) = $termi_0$ *xs*"
**end**

**lemma** *terminal_TCons*[*simp*]: "*termi* (*TCons x xs*) = *termi xs*"
**by** (*cases xs*) *auto*

## 3.4 Compatibility Issues

The command **primrec_new** has been designed to be highly compatible with the old **primrec**, to ease migration. There is nonetheless at least one incompatibility that may arise when porting to the new package:

- *Theorems sometimes have different names.* For $m > 1$ mutually recursive functions, $f_1$_…_$f_m$.*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

Noncorecursive codatatypes coincide with the corresponding datatypes, so they are useless 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** $(lset\colon 'a)$ $llist$ $(map\colon lmap$ $rel\colon llist\_all2) =$
  $lnull\colon LNil$ $(defaults$ $ltl\colon LNil)$
| $LCons$ $(lhd\colon 'a)$ $(ltl\colon$ "$'a$ $llist$"$)$

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:

**codatatype** $(sset\colon 'a)$ $stream$ $(map\colon smap$ $rel\colon stream\_all2) =$
  $SCons$ $(shd\colon 'a)$ $(stl\colon$ "$'a$ $stream$"$)$

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

**codatatype** $enat = EZero \mid ESuc$ $enat$

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

Here is an example with many constructors:

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

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

### 4.1.2 Mutual Corecursion

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

**codatatype** $even\_enat = Even\_EZero \mid Even\_ESuc$ $odd\_enat$
**and** $odd\_enat = Odd\_ESuc$ $even\_enat$

### 4.1.3 Nested Corecursion

The next examples feature *nested corecursion*:

**codatatype** $'a$ $tree_{ii} = Node_{ii}$ $(lbl_{ii}\colon 'a)$ $(sub_{ii}\colon$ "$'a$ $tree_{ii}$ $llist$"$)$

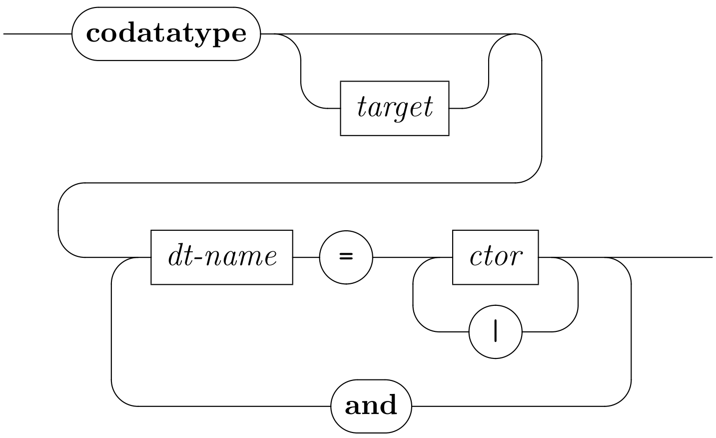
**codatatype** $'a$ $tree_{is} = Node_{is}$ $(lbl_{is}\colon 'a)$ $(sub_{is}\colon$ "$'a$ $tree_{is}$ $fset$"$)$

**codatatype** $'a$ $state\_machine$ $=$
  $State\_Machine$ $(accept\colon bool)$ $(trans\colon$ "$'a \Rightarrow 'a$ $state\_machine$"$)$

## 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 *no_discs_sels* option is not available, because destructors are a crucial notion 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 iterator and the recursor are replaced by dual concepts:

- Coiterator: *t_unfold*
- Corecursor: *t_corec*

## 4.4 Generated Theorems

The characteristic theorems generated by **codatatype** are grouped in three broad categories:

- The *free constructor theorems* are properties about the constructors and destructors that can be derived for any freely generated type.
- The *functorial theorems* are properties of datatypes related to their BNF nature.
- The *coinductive theorems* are properties of datatypes related to their coinductive nature.

The first two categories are exactly as for datatypes and are described in Sections 2.4.1 and 2.4.2.

### 4.4.1 Coinductive Theorems

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

*t*.***coinduct*** [*coinduct t*, *consumes m*, *case_names* $t_1 \ldots t_m$, *case_conclusion* $D_1 \ldots D_n$]:

$[\![R\ llist\ llist'; \bigwedge llist\ llist'.\ R\ llist\ llist' \Longrightarrow lnull\ llist = lnull\ llist' \wedge (\neg\ lnull\ llist \longrightarrow \neg\ lnull\ llist' \longrightarrow lhd\ llist = lhd\ llist' \wedge R\ (ltl\ llist)\ (ltl\ llist'))]\!] \Longrightarrow llist = llist'$

*t*.***strong_coinduct*** [*consumes m*, *case_names* $t_1 \ldots t_m$, *case_conclusion* $D_1 \ldots D_n$]:

$[\![R\ llist\ llist'; \bigwedge llist\ llist'.\ R\ llist\ llist' \Longrightarrow lnull\ llist = lnull\ llist' \wedge (\neg\ lnull\ llist \longrightarrow \neg\ lnull\ llist' \longrightarrow lhd\ llist = lhd\ llist' \wedge (R\ (ltl\ llist)\ (ltl\ llist') \vee ltl\ llist = ltl\ llist'))]\!] \Longrightarrow llist = llist'$

$t_1$_…_$t_m$.***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* $D_1 \ldots D_n$]:

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

*t*.***unfold***:

$p\ a \Longrightarrow llist\_unfold\ p\ g21\ g22\ a = LNil$
$\neg\ p\ a \Longrightarrow llist\_unfold\ p\ g21\ g22\ a = LCons\ (g21\ a)\ (llist\_unfold\ p\ g21\ g22\ (g22\ a))$

*t*.***corec***:

$p\ a \Longrightarrow llist\_corec\ p\ g21\ q22\ g22\ h22\ a = LNil$
$\neg\ p\ a \Longrightarrow llist\_corec\ p\ g21\ q22\ g22\ h22\ a = LCons\ (g21\ a)\ (\mathit{if}\ q22\ a\ \mathit{then}\ g22\ a\ \mathit{else}\ llist\_corec\ p\ g21\ q22\ g22\ h22\ (h22\ a))$

*t*.***disc_unfold***:

$p\ a \Longrightarrow lnull\ (llist\_unfold\ p\ g21\ g22\ a)$
$\neg\ p\ a \Longrightarrow \neg\ lnull\ (llist\_unfold\ p\ g21\ g22\ a)$

$t.$***disc_corec***:
$p\ a \Longrightarrow$ *lnull* (*llist_corec* $p\ g21\ q22\ g22\ h22\ a$)
$\neg\ p\ a \Longrightarrow \neg$ *lnull* (*llist_corec* $p\ g21\ q22\ g22\ h22\ a$)

$t.$***disc_unfold_iff*** [*simp*]:
*lnull* (*llist_unfold* $p\ g21\ g22\ a$) $= p\ a$
($\neg$ *lnull* (*llist_unfold* $p\ g21\ g22\ a$)) $= (\neg\ p\ a)$

$t.$***disc_corec_iff*** [*simp*]:
*lnull* (*llist_corec* $p\ g21\ q22\ g22\ h22\ a$) $= p\ a$
($\neg$ *lnull* (*llist_corec* $p\ g21\ q22\ g22\ h22\ a$)) $= (\neg\ p\ a)$

$t.$***sel_unfold*** [*simp*]:
$\neg\ p\ a \Longrightarrow$ *lhd* (*llist_unfold* $p\ g21\ g22\ a$) $= g21\ a$
$\neg\ p\ a \Longrightarrow$ *ltl* (*llist_unfold* $p\ g21\ g22\ a$) $=$ *llist_unfold* $p\ g21\ g22\ (g22\ a)$

$t.$***sel_corec*** [*simp*]:
$\neg\ p\ a \Longrightarrow$ *lhd* (*llist_corec* $p\ g21\ q22\ g22\ h22\ a$) $= g21\ a$
$\neg\ p\ a \Longrightarrow$ *ltl* (*llist_corec* $p\ g21\ q22\ g22\ h22\ a$) $=$ (*if* $q22\ a$ *then* $g22\ a$ *else* *llist_corec* $p\ g21\ q22\ g22\ h22\ (h22\ a)$)

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

$t.$***simps*** $=$ *t.inject t.distinct t.case t.disc_corec t.disc_corec_iff t.sel_corec t.disc_unfold t.disc_unfold_iff t.sel_unfold t.map t.rel_inject t.rel_distinct t.set*

# 5 Defining Corecursive Functions

Corecursive functions can be specified using **primcorec** and **primcorecursive**, which support primitive corecursion, or using the more general **partial_function** command. Here, the focus is on the former 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

$$un\_C_j i \ (f\, x_1 \ \ldots \ x_n) = \ldots$$

This style is popular in the coalgebraic literature.

- The *constructor view* specifies $f$ by equations of the form

  $$\ldots \Longrightarrow f\, x_1 \ \ldots \ x_n = C \ \ldots$$

  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.

> **Warning:** The **primcorec** and **primcorecursive** commands are under development. Some of the functionality described here is vaporware. An alternative is to define corecursive functions directly using the generated *t_unfold* or *t_corec* combinators.

## 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* :: "$('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ llist$" **where**
  "$literate\ f\ x = LCons\ x\ (literate\ f\ (f\ x))$"

**primcorec** *siterate* :: "$('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ stream$" **where**
"$siterate\ f\ x = SCons\ x\ (siterate\ f\ (f\ x))$"

The constructor ensures that progress is made—i.e., the function is *productive*. The above functions compute the infinite lazy list or stream $[x, f\,x, f\ (f\ x), \ldots]$. Productivity guarantees that prefixes $[x, f\,x, f\ (f\,x), \ldots, (f\ \hat{}\hat{}\ k)\ x]$ 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* :: "$'a\ stream \Rightarrow 'a\ stream$" **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_notyet** *lappend* :: "$'a\ llist \Rightarrow 'a\ llist \Rightarrow 'a\ llist$" **where**
"$lappend\ xs\ ys =$
$(case\ xs\ of$
$LNil \Rightarrow ys$
$\mid LCons\ x\ xs' \Rightarrow LCons\ x\ (lappend\ xs'\ ys))$"

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_notyet**
*random_process* :: "$'a\ stream \Rightarrow (int \Rightarrow int) \Rightarrow int \Rightarrow 'a\ process$"
**where**
"$random\_process\ s\ f\ n =$
$(if\ n\ mod\ 4 = 0\ then$
$Fail$
$else\ if\ n\ mod\ 4 = 1\ then$
$Skip\ (random\_process\ s\ f\ (f\ n))$
$else\ if\ n\ mod\ 4 = 2\ then$
$Action\ (shd\ s)\ (random\_process\ (stl\ s)\ f\ (f\ n))$
$else$
$Choice\ (random\_process\ (every\_snd\ s)\ (f \circ f)\ (f\ n))$
$(random\_process\ (every\_snd\ (stl\ s))\ (f \circ f)\ (f\ (f\ n))))$"

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 ($tree_{is}$):

**primcorec** $iterate_{ii}$ :: "('a ⇒ 'a *llist*) ⇒ 'a ⇒ 'a $tree_{ii}$" **where**
  "$iterate_{ii}$ *f x* = $Node_{ii}$ *x* (*lmap* ($iterate_{ii}$ *f*) (*f x*))"

**primcorec** $iterate_{is}$ :: "('a ⇒ 'a *fset*) ⇒ 'a ⇒ 'a $tree_{is}$" **where**
  "$iterate_{is}$ *f x* = $Node_{is}$ *x* (*fimage* ($iterate_{is}$ *f*) (*f x*))"

Deterministic finite automata (DFAs) are traditionally defined as 5-tuples $(Q,\ \Sigma,\ \delta,\ q_0,\ F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet, $\delta$ is a transition function, $q_0$ is an initial state, and $F$ is a set of final states. The following function translates a DFA into a *state_machine*:

**primcorec** *sm_of_dfa* :: "('q ⇒ 'a ⇒ 'q) ⇒ 'q *set* ⇒ 'q ⇒ 'a *state_machine*"
**where**
  "*sm_of_dfa* δ *F q* = *State_Machine* (*q* ∈ *F*) (*sm_of_dfa* δ *F o* δ *q*)"

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

**primcorec**
  *sm_of_dfa* :: "('q ⇒ 'a ⇒ 'q) ⇒ 'q *set* ⇒ 'q ⇒ 'a *state_machine*"
**where**
  "*sm_of_dfa* δ *F q* = *State_Machine* (*q* ∈ *F*) (*sm_of_dfa* δ *F o* δ *q*)"

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

"*empty_sm* = *State_Machine False* (λ_. *empty_sm*)"

**primcorec** *not_sm* :: "'*a state_machine* ⇒ '*a state_machine*" **where**
"*not_sm M* = *State_Machine* (¬ *accept M*) (λ*a*. *not_sm* (*trans M a*))"

**primcorec**
*or_sm* :: "'*a state_machine* ⇒ '*a state_machine* ⇒ '*a state_machine*"
**where**
"*or_sm M N* =
*State_Machine* (*accept M* ∨ *accept N*)
(λ*a*. *or_sm* (*trans M a*) (*trans N a*))"

### 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_notyet**
$iterate_{ii}$ :: "('*a* ⇒ '*a llist*) ⇒ '*a* ⇒ '*a* $tree_{ii}$" **and**
$iterates_{ii}$ :: "('*a* ⇒ '*a llist*) ⇒ '*a llist* ⇒ '*a* $tree_{ii}$ *llist*"
**where**
"$iterate_{ii}$ *f x* = $Node_{ii}$ *x* ($iterates_{ii}$ *f* (*f x*))" |
"$iterates_{ii}$ *f xs* =
(*case xs of*
*LNil* ⇒ *LNil*
| *LCons x xs'* ⇒ *LCons* ($iterate_{ii}$ *f x*) ($iterates_{ii}$ *f xs'*))"

### 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* $\Rightarrow$ (*int* $\Rightarrow$ *int*) $\Rightarrow$ *int* $\Rightarrow$ $'a$ *process*"
**where**
  "$n$ *mod* $4 = 0 \Longrightarrow$ *random_process* $s\ f\ n$ = *Fail*" |
  "$n$ *mod* $4 = 1 \Longrightarrow$
    *random_process* $s\ f\ n$ = *Skip* (*random_process* $s\ f$ ($f\ n$))" |
  "$n$ *mod* $4 = 2 \Longrightarrow$
    *random_process* $s\ f\ n$ = *Action* (*shd* $s$) (*random_process* (*stl* $s$) $f$ ($f\ n$))" |
  "$n$ *mod* $4 = 3 \Longrightarrow$
    *random_process* $s\ f\ n$ = *Choice* (*random_process* (*every_snd* $s$) $f$ ($f\ n$))
      (*random_process* (*every_snd* (*stl* $s$)) $f$ ($f\ n$))"

Since there is no sequentiality, we can apply the equation for *Choice* without having first to discharge $n$ *mod* $4 \neq 0$, $n$ *mod* $4 \neq 1$, and $n$ *mod* $4 \neq 2$. The price to pay for this elegance is that we must discharge exclusivity 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:

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

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

**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 *literate* specification indicates which constructor to choose. For *siterate* and *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* ⇒ '*a llist* ⇒ '*a llist*" **where**
"*lnull xs* ⟹ *lnull ys* ⟹ *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

"_ ⟹ ¬ *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* ⇒ (*int* ⇒ *int*) ⇒ *int* ⇒ '*a process*"
**where**
"*n mod* 4 = 0 ⟹ *is_Fail* (*random_process s f n*)" |
"*n mod* 4 = 1 ⟹ *is_Skip* (*random_process s f n*)" |
"*n mod* 4 = 2 ⟹ *is_Action* (*random_process s f n*)" |
"*n mod* 4 = 3 ⟹ *is_Choice* (*random_process s f n*)" |
"*cont* (*random_process s f n*) = *random_process s f* (*f n*)" *of Skip* |
"*prefix* (*random_process s f n*) = *shd s*" |
"*cont* (*random_process s f n*) = *random_process* (*stl s*) *f* (*f n*)" *of Action* |
"*left* (*random_process s f n*) = *random_process* (*every_snd s*) *f* (*f n*)" |
"*right* (*random_process s f n*) = *random_process* (*every_snd* (*stl s*)) *f* (*f n*)"

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

Here are more examples to conclude:

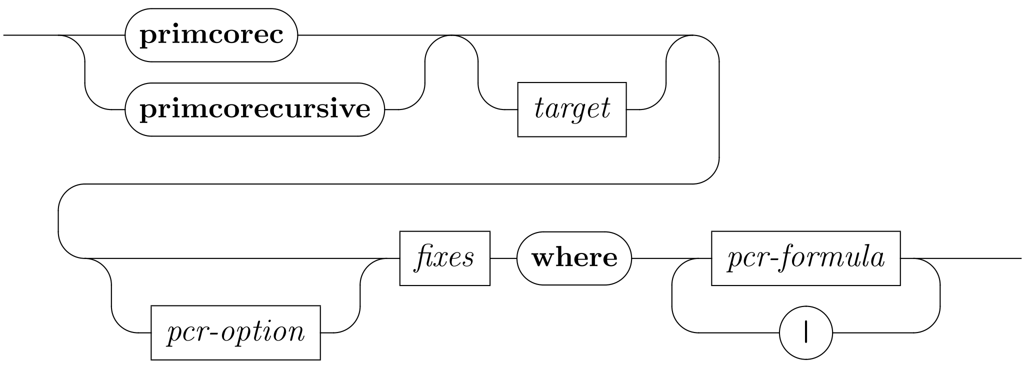
**primcorec**
*even_infty* :: *even_enat* **and**
*odd_infty* :: *odd_enat*
**where**
"¬ *is_Even_EZero even_infty*" |
"*un_Even_ESuc even_infty* = *odd_infty*" |
"*un_Odd_ESuc odd_infty* = *even_infty*"

**primcorec** $\mathit{iterate}_{ii}$ :: "($'a \Rightarrow 'a$ *llist*) $\Rightarrow 'a \Rightarrow 'a$ $\mathit{tree}_{ii}$" **where**
"$\mathit{lbl}_{ii}$ ($\mathit{iterate}_{ii}$ $f$ $x$) $= x$" |
"$\mathit{sub}_{ii}$ ($\mathit{iterate}_{ii}$ $f$ $x$) $=$ *lmap* ($\mathit{iterate}_{ii}$ $f$) ($f$ $x$)"

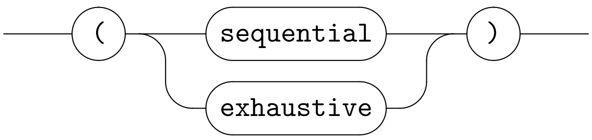
## 5.2 Command Syntax

### 5.2.1 primcorec and primcorecursive

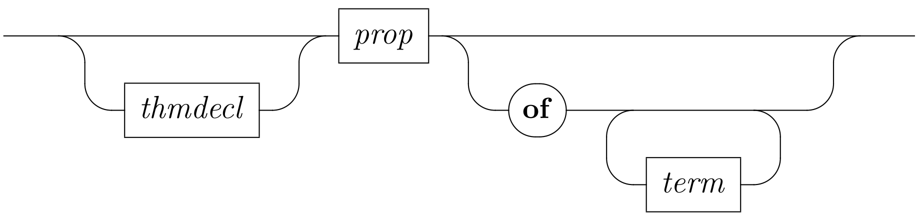
$$\begin{array}{rcl} \textbf{primcorec} & : & \mathit{local\_theory} \rightarrow \mathit{local\_theory} \\ \textbf{primcorecursive} & : & \mathit{local\_theory} \rightarrow \mathit{proof}(\mathit{prove}) \end{array}$$




*pcr-option*




*pcr-formula*




The optional target is optionally followed by a corecursion-specific option:

- 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.

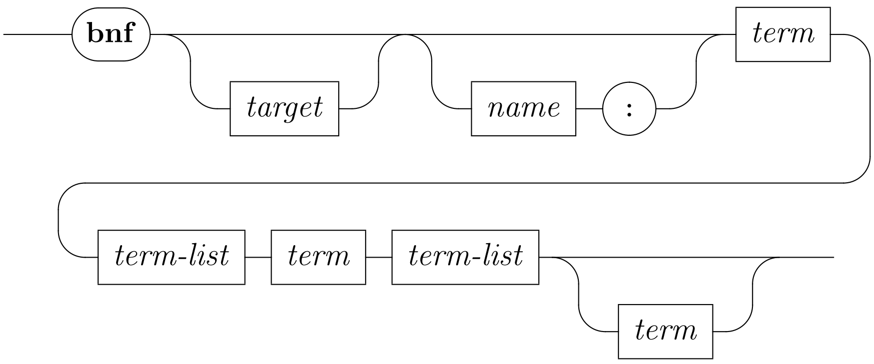
# 6 Registering Bounded Natural Functors

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.

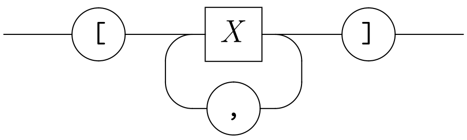
## 6.1 Command Syntax

### 6.1.1 bnf

$$\textbf{bnf} \;:\; \mathit{local\_theory} \rightarrow \mathit{proof}(\mathit{prove})$$




*X-list*



#### 6.1.2 print_bnfs

**print_bnfs** : *local_theory* →

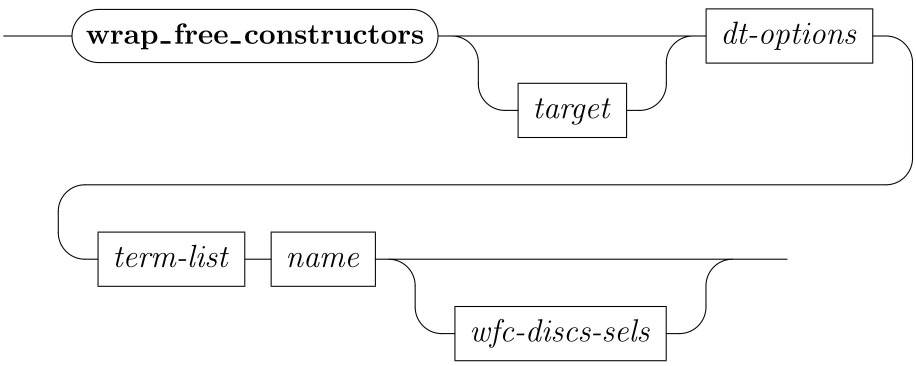

# 7 Deriving Destructors and Theorems for Free Constructors

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 **wrap_free_constructors**.

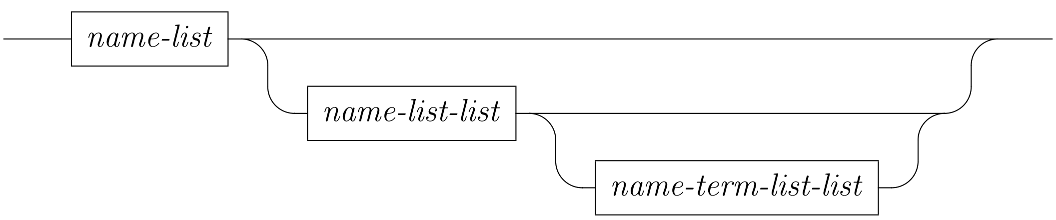
## 7.1 Command Syntax

### 7.1.1 wrap_free_constructors

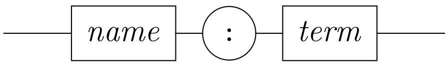
**wrap_free_constructors** : *local_theory* → *proof*(*prove*)




*wfc-discs-sels*



*name-term*




Section 2.4 lists the generated theorems.

# 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 for the coinductive part. Brian Huffman suggested major simplifications to the internal constructions, much of which has 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 to eliminate one of the BNF assumptions. 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. *Picking Nits: A User's Guide to Nitpick for Isabelle/ HOL*. http://isabelle.in.tum.de/doc/nitpick.pdf.

[3] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. http://www21.in.tum.de/~traytel/papers/witness/wit.pdf, 2013.

[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. Coinduction. 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] 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.

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