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

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

23 May 2024

Abstract

This tutorial describes the definitional package for datatypes and codatatypes, and for primitively recursive and corecursive functions, in Isabelle/HOL. The following commands are provided: **datatype**, **datatype_compat**, **primrec**, **codatatype**, **primcorec**, **primcorecursive**, **bnf**, **lift_bnf**, **copy_bnf**, **bnf_axiomatization**, **print_bnfs**, and **free_constructors**.

Contents

1 Introduction

The 2013 edition of Isabelle introduced a definitional package for freely generated datatypes and codatatypes. This package replaces the earlier implementation due to Berghofer and Wenzel [1]. Perhaps the main advantage of the new package is that it supports recursion through a large class of non-datatypes, such as finite sets:

datatype $'a\ tree_{fs} = Node_{fs}\ (lbl_{fs}\colon 'a)\ (sub_{fs}\colon$ "$'a\ tree_{fs}\ fset$"$)$

Another strong point is the support for local definitions:

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

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

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

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

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

datatype $'a\ tree_{ff} = Node_{ff}\ 'a$ "$'a\ tree_{ff}\ list$"
datatype $'a\ tree_{fi} = Node_{fi}\ 'a$ "$'a\ tree_{fi}\ llist$"
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 `~~/src/HOL/Library/BNF_Axiomatization.thy`.

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

This tutorial is organized as follows:

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

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

- Section 4, "Defining Codatatypes," describes how to specify codatatypes using the **codatatype** command.
- Section 5, "Defining Primitively Corecursive Functions," describes how to specify functions using the **primcorec** and **primcorecursive** commands. (A separate tutorial [3] describes the more powerful **corec** and **corecursive** commands.)
- Section 6, "Registering Bounded Natural Functors," explains how to use the **bnf** command to register arbitrary type constructors as BNFs.
- Section 7, "Deriving Destructors and Constructor Theorems," explains how to use the command **free_constructors** to derive destructor constants and theorems for freely generated types, as performed internally by **datatype** and **codatatype**.
- Section 8, "Selecting Plugins," is concerned with the package's interoperability with other Isabelle packages and tools, such as the code generator, Transfer, Lifting, and Quickcheck.
- Section 9, "Known Bugs and Limitations," concludes with known open issues.

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

2 Defining Datatypes

Datatypes can be specified using the **datatype** command.

2.1 Introductory Examples

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

2.1.1 Nonrecursive Types

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

datatype *trool* = *Truue* | *Faalse* | *Perhaaps*

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

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

datatype $'a$ *option* = *None* | *Some* $'a$

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

The next example has three type parameters:

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

Occurrences of nonatomic types on the right-hand side of the equal sign must be enclosed in double quotes, as is customary in Isabelle.

2.1.2 Simple Recursion

Natural numbers are the simplest example of a recursive type:

datatype *nat* = *Zero* | *Succ nat*

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

datatype $('a, 'b)$ *tlist* = *TNil* $'b$ | *TCons* $'a$ "$('a, 'b)$ *tlist*"

2.1.3 Mutual Recursion

Mutually recursive types are introduced simultaneously and may refer to each other. The example below introduces a pair of types for even and odd natural numbers:

datatype *even_nat* = *Even_Zero* | *Even_Succ odd_nat*
and *odd_nat* = *Odd_Succ even_nat*

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

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

2.1.4 Nested Recursion

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

datatype $'a$ *btree* =
  BNode $'a$ "$'a$ *btree option*" "$'a$ *btree option*"

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

datatype $'a$ *wrong* = *W1* | *W2* "$'a$ *wrong* $\Rightarrow$ $'a$"

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

datatype ($'a$, $'b$) *fun_copy* = *Fun* "$'a \Rightarrow 'b$"
datatype $'a$ *also_wrong* = *W1* | *W2* "($'a$ *also_wrong*, $'a$) *fun_copy*"

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

datatype $'a$ *ftree* = *FTLeaf* $'a$ | *FTNode* "$'a \Rightarrow 'a$ *ftree*"

And so is the definition of hereditarily finite sets:

datatype *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** and **codatatype** commands. Section 6 explains how to register arbitrary type constructors as BNFs.

Here is another example that fails:

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

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

2.1.5 Auxiliary Constants

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

datatype (*set*: *'a*) *list* =
  null: *Nil*
| *Cons* (*hd*: *'a*) (*tl*: "*'a list*")
for
  map: *map*
  rel: *list_all2*
  pred: *list_all*
where
  "*tl Nil* = *Nil*"

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

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

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. In the example, it is used to ensure that the tail of the empty list is itself (instead of being left unspecified).

Because *Nil* is nullary, it is also possible to use $\lambda \mathit{xs}.\ \mathit{xs} = \mathit{Nil}$ 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 ('*a*, '*b*) *prod* (**infixr** "∗" 20) = *Pair* '*a* '*b*

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

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

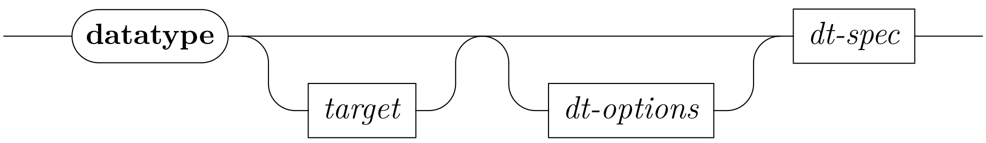
syntax "*_list*" :: "*args* ⇒ '*a list*" ("[(_)]")

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

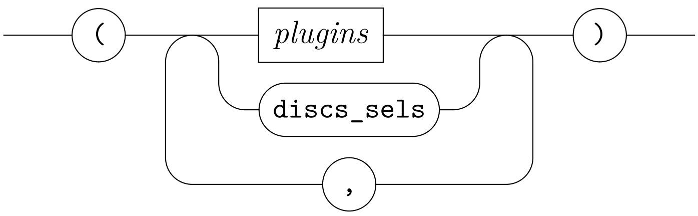
2.2 Command Syntax

2.2.1 datatype

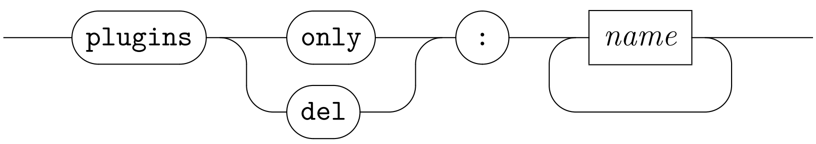
datatype : *local_theory* → *local_theory*



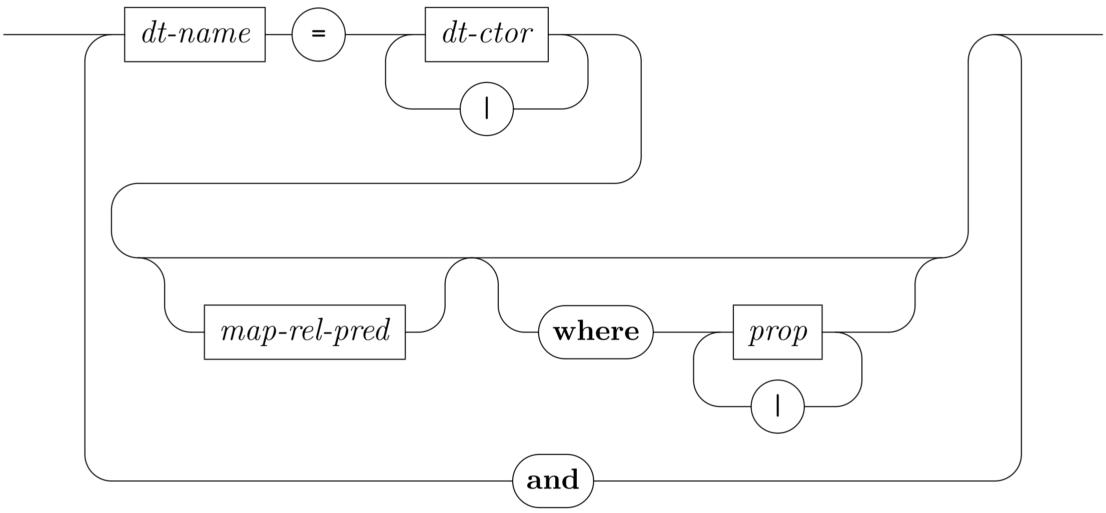
dt-options



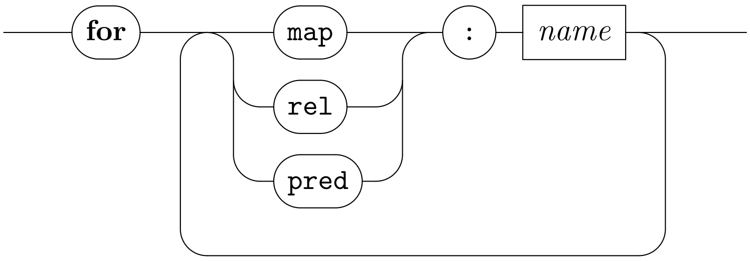
plugins



dt-spec



map-rel-pred



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

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

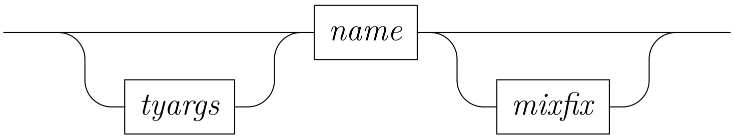
The optional target is optionally followed by a combination of the following options:

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

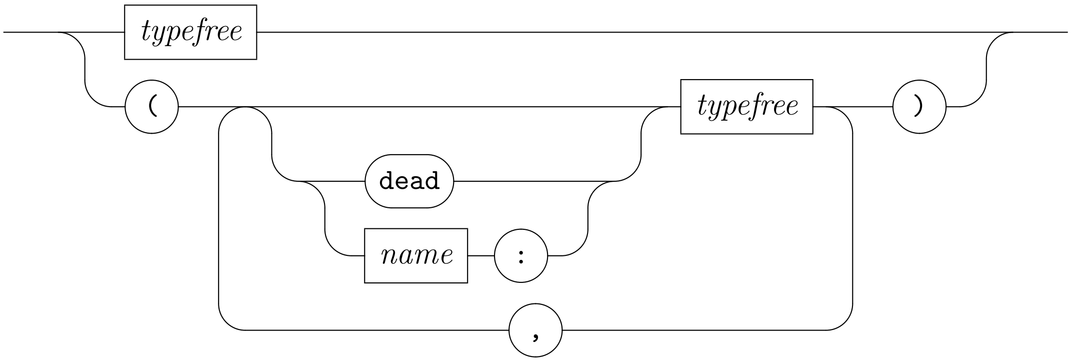
The optional **where** clause specifies default values for selectors. Each proposition must be an equation of the form $un\_D\ (C\ \ldots) = \ldots$, where C is a constructor and un_D is a selector.

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

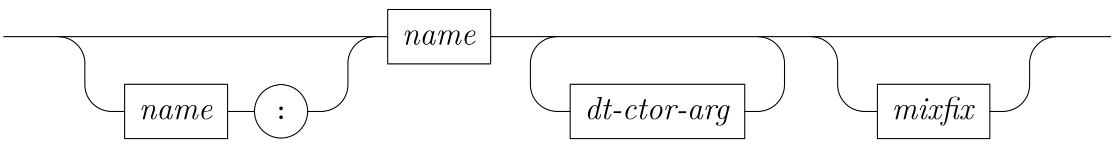


The syntactic entity *name* denotes an identifier, *mixfix* denotes the usual parenthesized mixfix notation, and *typefree* denotes fixed type variable ($'a$, $'b$, . . .) [12].

The optional names preceding the type variables allow to override the default names of the set functions (set_1_t, . . . , set_m_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.

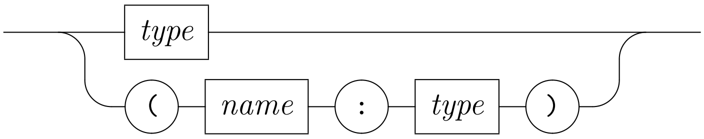
Inside a mutually recursive specification, all defined datatypes must mention exactly the same type variables in the same order.

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. If discriminators are enabled (cf. the *discs_sels* option) but no name is supplied, the default is $\lambda x.\ x = C_j$ for nullary constructors and $t.is\_C_j$ otherwise.

dt-ctor-arg

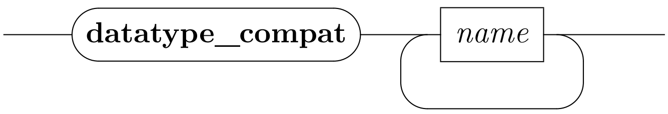


The syntactic entity *type* denotes a HOL type [12].

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

2.2.2 datatype_compat

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



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

datatype_compat *even_nat odd_nat*

ML ‹*Old_Datatype_Data.get_info* ***theory*** ***type_name*** ‹*even_nat*››

The syntactic entity *name* denotes an identifier [12].

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

A few remarks concern nested recursive datatypes:

- The old-style, nested-as-mutual induction rule and recursor theorems are generated under their usual names but with "*compat_*" prefixed (e.g., *compat_tree.induct*, *compat_tree.inducts*, and *compat_tree.rec*). These theorems should be identical to the ones generated by the old datatype package, *up to the order of the premises*—meaning that the subgoals generated by the *induct* or *induction* method may be in a different order than before.
- All types through which recursion takes place must be new-style datatypes or the function type.

2.3 Generated Constants

Given a datatype $('a_1, \ldots, 'a_m)\ t$ with m live type variables and n constructors $t.C_1, \ldots, t.C_n$, the following auxiliary constants are introduced:

Case combinator:	$t.case\_t$ (rendered using the familiar *case–of* syntax)
Discriminators:	$t.is\_C_1, \ldots, t.is\_C_n$
Selectors:	$t.un\_C_1 1, \ldots, t.un\_C_1 k_1$
	$\vdots$
	$t.un\_C_n 1, \ldots, t.un\_C_n k_n$
Set functions:	$t.set_1\_t, \ldots, t.set_m\_t$
Map function:	$t.map\_t$
Relator:	$t.rel\_t$
Recursor:	$t.rec\_t$

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

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

2.4 Generated Theorems

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

- The *free constructor theorems* (Section 2.4.1) are properties of the constructors and destructors that can be derived for any freely generated type. Internally, the derivation is performed by **free_constructors**.

- The *functorial theorems* (Section 2.4.2) are properties of datatypes related to their BNF nature.
- The *inductive theorems* (Section 2.4.3) are properties of datatypes related to their inductive nature.

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

declare [[*bnf_internals*]]

2.4.1 Free Constructor Theorems

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

t.***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* $C_1 \ldots C_n$]:
$[\![y = [] \Longrightarrow P; \bigwedge x21\ x22.\ y = x21 \# x22 \Longrightarrow P]\!] \Longrightarrow P$

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

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

t.***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*]:
$(case\ []\ of\ [] \Rightarrow f1 \mid x \# xa \Rightarrow f2\ x\ xa) = f1$
$(case\ x21 \# x22\ of\ [] \Rightarrow f1 \mid x \# xa \Rightarrow f2\ x\ xa) = f2\ x21\ x22$
The [*code*] attribute is set by the *code* plugin (Section 8.1).

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

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

$t.$***case_distrib***:
$h\ (case\ list\ of\ [] \Rightarrow f1 \mid x \# xa \Rightarrow f2\ x\ xa) = (case\ list\ of\ [] \Rightarrow h\ f1 \mid x1 \# x2 \Rightarrow h\ (f2\ x1\ x2))$

$t.$***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 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*]:
$hd\ (x21 \# x22) = x21$
$tl\ (x21 \# x22) = x22$
The [*code*] attribute is set by the *code* plugin (Section 8.1).

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

$t.$***distinct_disc*** [*dest*]:
These properties are missing for *'a list* because there is only one

proper discriminator. If the datatype had been introduced with a second discriminator called *nonnull*, they would have read as follows:

$null\ list \Longrightarrow \neg\ nonnull\ list$
$nonnull\ list \Longrightarrow \neg\ null\ list$

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

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

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

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

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

$t.$***split_sels*** **=** *split_sel split_sel_asm*

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

$t.$***disc_eq_case***:
$null\ list = (case\ list\ of\ [] \Rightarrow True \mid uu\_\ \#\ uua\_ \Rightarrow False)$
$(\neg\ null\ list) = (case\ list\ of\ [] \Rightarrow False \mid uu\_\ \#\ uua\_ \Rightarrow True)$

In addition, equational versions of $t.disc$ are registered with the [*code*] attribute. The [*code*] attribute is set by the *code* plugin (Section 8.1).

2.4.2 Functorial Theorems

The functorial theorems are generated for type constructors with at least one live type argument (e.g., *'a list*). They are partitioned in two subgroups. The first subgroup consists of properties involving the constructors or the destructors and either a set function, the map function, the predicator, or the relator:

$t.$***case_transfer*** [*transfer_rule*]:
rel_fun S (rel_fun (rel_fun R (rel_fun (list_all2 R) S)) (rel_fun (list_all2 R) S)) case_list case_list
This property is generated by the *transfer* plugin (Section 8.3).

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

$t.$***ctr_transfer*** [*transfer_rule*]:
list_all2 R [] []
rel_fun R (*rel_fun* (*list_all2 R*) (*list_all2 R*)) (#) (#)
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3)
.

$t.$***disc_transfer*** [*transfer_rule*]:
rel_fun (*list_all2 R*) (=) *null null*
rel_fun (*list_all2 R*) (=) (λ*list*. ¬ *null list*) (λ*list*. ¬ *null list*)
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3).

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

$t.$***set_cases*** [*consumes* 1, *cases set*: set_i_t]:
$[\![e \in set\ a; \bigwedge z2.\ a = e \ \#\ z2 \Longrightarrow thesis; \bigwedge z1\ z2.\ [\![a = z1 \ \#\ z2;\ e \in set\ z2]\!] \Longrightarrow thesis]\!] \Longrightarrow thesis$

$t.$***set_intros***:
$x21 \in set\ (x21\ \#\ x22)$
$y \in set\ x22 \Longrightarrow y \in set\ (x21\ \#\ x22)$

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

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

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

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

$t.$***pred_inject*** [*simp*]:
list_all P []
list_all P (*a* # *aa*) = (*P a* ∧ *list_all P aa*)

$t.$***rel_inject*** [*simp*]:
$list\_all2\ R\ []\ []$
$list\_all2\ R\ (x21\ \#\ x22)\ (y21\ \#\ y22) = (R\ x21\ y21 \wedge list\_all2\ R\ x22\ y22)$

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

$t.$***rel_intros***:
$list\_all2\ R\ []\ []$
$[\![R\ x21\ y21;\ list\_all2\ R\ x22\ y22]\!] \Longrightarrow list\_all2\ R\ (x21\ \#\ x22)\ (y21\ \#\ y22)$

$t.$***rel_cases*** [*consumes* 1, *case_names* $t_1\ \ldots\ t_m$, *cases pred*]:
$[\![list\_all2\ R\ a\ b;\ [\![a = [];\ b = []]\!] \Longrightarrow thesis;\ \bigwedge x1\ x2\ y1\ y2.\ [\![a = x1\ \#\ x2;\ b = y1\ \#\ y2;\ R\ x1\ y1;\ list\_all2\ R\ x2\ y2]\!] \Longrightarrow thesis]\!] \Longrightarrow thesis$

$t.$***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 [*code*] attribute is set by the *code* plugin (Section 8.1).

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

$t.$***inj_map***:
$inj\ f \Longrightarrow inj\ (map\ f)$

$t.$***inj_map_strong***:
$[\![\bigwedge z\ za.\ [\![z \in set\ x;\ za \in set\ xa;\ f\ z = fa\ za]\!] \Longrightarrow z = za;\ map\ f\ x = map\ fa\ xa]\!] \Longrightarrow x = xa$

$t.$***map_comp***:
$map\ g\ (map\ f\ v) = map\ (g \circ f)\ v$

$t.$***map_cong0***:
$(\bigwedge z.\ z \in set\ x \Longrightarrow f\ z = g\ z) \Longrightarrow map\ f\ x = map\ g\ x$

$t.$***map_cong*** [*fundef_cong*]:
$[\![x = ya;\ \bigwedge z.\ z \in set\ ya \Longrightarrow f\ z = g\ z]\!] \Longrightarrow map\ f\ x = map\ g\ ya$

$t.$***map_cong_pred***:
$[\![x = ya;\ list\_all\ (\lambda z.\ f\ z = g\ z)\ ya]\!] \Longrightarrow map\ f\ x = map\ g\ ya$

$t.$***map_cong_simp***:
$[\![x = ya; \bigwedge z.\ z \in set\ ya =simp=> f\ z = g\ z]\!] \Longrightarrow map\ f\ x = map\ g\ ya$

$t.$***map_id0***:
$map\ id = id$

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

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

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

$t.$***map_transfer*** [*transfer_rule*]:
$rel\_fun\ (rel\_fun\ Rb\ Sd)\ (rel\_fun\ (list\_all2\ Rb)\ (list\_all2\ Sd))\ map\ map$
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3) for type constructors with no dead type arguments.

$t.$***pred_cong*** [*fundef_cong*]:
$[\![x = ya; \bigwedge z.\ z \in set\ ya \Longrightarrow P\ z = Pa\ z]\!] \Longrightarrow list\_all\ P\ x = list\_all\ Pa\ ya$

$t.$***pred_cong_simp***:
$[\![x = ya; \bigwedge z.\ z \in set\ ya =simp=> P\ z = Pa\ z]\!] \Longrightarrow list\_all\ P\ x = list\_all\ Pa\ ya$

$t.$***pred_map***:
$list\_all\ Q\ (map\ f\ x) = list\_all\ (Q \circ f)\ x$

$t.$***pred_mono*** [*mono*]:
$P \le Pa \Longrightarrow list\_all\ P \le list\_all\ Pa$

$t.$***pred_mono_strong***:
$[\![list\_all\ P\ x; \bigwedge z.\ [\![z \in set\ x; P\ z]\!] \Longrightarrow Pa\ z]\!] \Longrightarrow list\_all\ Pa\ x$

$t.$***pred_rel***:
$list\_all\ P\ x = list\_all2\ (eq\_onp\ P)\ x\ x$

$t.$***pred_set***:
$list\_all\ P = (\lambda x.\ Ball\ (set\ x)\ P)$

$t.$***pred_transfer*** [*transfer_rule*]:
$rel\_fun\ (rel\_fun\ R\ (=))\ (rel\_fun\ (list\_all2\ R)\ (=))\ list\_all\ list\_all$
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3) for type constructors with no dead type arguments.

$t.$***pred_True***:
$list\_all\ (\lambda\_.\ True) = (\lambda\_.\ True)$

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

$t.$***set_transfer*** [*transfer_rule*]:
$rel\_fun\ (list\_all2\ R)\ (rel\_set\ R)\ set\ set$
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3) for type constructors with no dead type arguments.

$t.$***rel_compp*** [*relator_distr*]:
$list\_all2\ (R\ OO\ S) = list\_all2\ R\ OO\ list\_all2\ S$
The [*relator_distr*] attribute is set by the *lifting* plugin (Section 8.4).

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

$t.$***rel_eq***:
$list\_all2\ (=) = (=)$

$t.$***rel_eq_onp***:
$list\_all2\ (eq\_onp\ P) = eq\_onp\ (list\_all\ P)$

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

$t.$***rel_map***:
$list\_all2\ Sb\ (map\ i\ x)\ y = list\_all2\ (\lambda x.\ Sb\ (i\ x))\ x\ y$
$list\_all2\ Sa\ x\ (map\ g\ y) = list\_all2\ (\lambda x\ y.\ Sa\ x\ (g\ y))\ x\ y$

$t.$***rel_mono*** [*mono*, *relator_mono*]:
$R \le Ra \Longrightarrow list\_all2\ R \le list\_all2\ Ra$
The [*relator_mono*] attribute is set by the *lifting* plugin (Section 8.4).

$t.$***rel_mono_strong***:
$[\![list\_all2\ R\ x\ y;\ \bigwedge z\ yb.\ [\![z \in set\ x;\ yb \in set\ y;\ R\ z\ yb]\!] \Longrightarrow Ra\ z\ yb]\!] \Longrightarrow list\_all2\ Ra\ x\ y$

$t.$***rel_cong*** [*fundef_cong*]:
$[\![x = ya;\ y = xa;\ \bigwedge z\ yb.\ [\![z \in set\ ya;\ yb \in set\ xa]\!] \Longrightarrow R\ z\ yb = Ra\ z\ yb]\!] \Longrightarrow list\_all2\ R\ x\ y = list\_all2\ Ra\ ya\ xa$

$t.$***rel_cong_simp***:
$[\![x = ya;\ y = xa;\ \bigwedge z\ yb.\ z \in set\ ya =simp=> yb \in set\ xa =simp=> R\ z\ yb = Ra\ z\ yb]\!] \Longrightarrow list\_all2\ R\ x\ y = list\_all2\ Ra\ ya\ xa$

$t.$***rel_refl***:
$(\bigwedge x.\ Ra\ x\ x) \Longrightarrow list\_all2\ Ra\ x\ x$

t.***rel_refl_strong***:
$(\bigwedge z.\ z \in set\ x \Longrightarrow Ra\ z\ z) \Longrightarrow list\_all2\ Ra\ x\ x$

t.***rel_reflp***:
$reflp\ R \Longrightarrow reflp\ (list\_all2\ R)$

t.***rel_symp***:
$symp\ R \Longrightarrow symp\ (list\_all2\ R)$

t.***rel_transp***:
$transp\ R \Longrightarrow transp\ (list\_all2\ R)$

t.***rel_transfer*** [*transfer_rule*]:
$rel\_fun\ (rel\_fun\ Sa\ (rel\_fun\ Sc\ (=)))\ (rel\_fun\ (list\_all2\ Sa)\ (rel\_fun\ (list\_all2\ Sc)\ (=)))\ list\_all2\ list\_all2$
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3) for type constructors with no dead type arguments.

2.4.3 Inductive Theorems

The inductive theorems are as follows:

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

t.***rel_induct*** [*case_names* $C_1 \ldots C_n$, *induct pred*]:
$[\![list\_all2\ R\ x\ y;\ Q\ []\ [];\ \bigwedge a21\ a22\ b21\ b22.\ [\![R\ a21\ b21;\ Q\ a22\ b22]\!] \Longrightarrow Q\ (a21\ \#\ a22)\ (b21\ \#\ b22)]\!] \Longrightarrow Q\ x\ y$

$t_1\_\ldots\_t_m$.***induct*** [*case_names* $C_1 \ldots C_n$]:
$t_1\_\ldots\_t_m$.***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*]:
$rec\_list\ f1\ f2\ [] = f1$
$rec\_list\ f1\ f2\ (x21\ \#\ x22) = f2\ x21\ x22\ (rec\_list\ f1\ f2\ x22)$
The [*code*] attribute is set by the *code* plugin (Section 8.1).

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

t.***rec_transfer*** [*transfer_rule*]:
$rel\_fun\ S\ (rel\_fun\ (rel\_fun\ R\ (rel\_fun\ (list\_all2\ R)\ (rel\_fun\ S\ S)))\ (rel\_fun\ (list\_all2\ R)\ S))\ rec\_list\ rec\_list$
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3) for type constructors with no dead type arguments.

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

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

2.5 Proof Method

2.5.1 *countable_datatype*

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

instance *list* :: (*countable*) *countable*
  by *countable_datatype*

2.6 Antiquotation

2.6.1 *datatype*

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

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

2.7 Compatibility Issues

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

- *The Standard ML interfaces are different.* Tools and extensions written to call the old ML interfaces will need to be adapted to the new interfaces. The *BNF_LFP_Compat* structure provides convenience functions that simulate the old interfaces in terms of the new ones.
- *The recursor rec_t has a different signature for nested recursive datatypes.* In the old package, nested recursion through non-functions was internally reduced to mutual recursion. This reduction was visible in

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

- *Accordingly, the induction rule is different for nested recursive datatypes.* Again, the old-style induction rule can be generated on demand using **primrec** if the recursion is via new-style datatypes, as explained in Section 3.1.5, or using **datatype_compat**. For recursion through functions, the old-style induction rule can be obtained by applying the [*unfolded all_mem_range*] attribute on *t.induct*.
- *The size function has a slightly different definition.* The new function returns 1 instead of 0 for some nonrecursive constructors. This departure from the old behavior made it possible to implement *size* in terms of the generic function *t.size_t*. Moreover, the new function considers nested occurrences of a value, in the nested recursive case. The old behavior can be obtained by disabling the *size* plugin (Section 8) and instantiating the *size* type class manually.
- *The internal constructions are completely different.* Proof texts that unfold the definition of constants introduced by the old command will be difficult to port.
- *Some constants and theorems have different names.* For non-mutually recursive datatypes, the alias *t.inducts* for *t.induct* is no longer generated. For $m > 1$ mutually recursive datatypes, $rec\_t_1\_\ldots\_t_m\_i$ has been renamed rec_t_i for each $i \in \{1, \ldots, m\}$, $t_1\_\ldots\_t_m.inducts(i)$ has been renamed $t_i.induct$ for each $i \in \{1, \ldots, m\}$, and the collection $t_1\_\ldots\_t_m.size$ (generated by the *size* plugin, Section 8.2) has been divided into $t_1.size, \ldots, t_m.size$.
- *The t.simps collection has been extended.* Previously available theorems are available at the same index as before.
- *Variables in generated properties have different names.* This is rarely an issue, except in proof texts that refer to variable names in the [*where* ...] attribute. The solution is to use the more robust [*of* ...] syntax.

3 Defining Primitively Recursive Functions

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

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

3.1 Introductory Examples

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

3.1.1 Nonrecursive Types

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

primrec (*nonexhaustive*) *bool_of_trool* :: "*trool* ⇒ *bool*" **where**
"*bool_of_trool 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 (Succ n) x = x # replicate n x*"

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

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

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

simps_of_case *at_simps_alt*: *at.simps*

This generates the lemma collection *at_simps_alt*:

$$at\ (x\ \#\ xs)\ Zero = x \qquad at\ (xa\ \#\ xs)\ (Succ\ x) = at\ xs\ x$$

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

fun *at_least_two* :: "*nat* ⇒ *bool*" **where**
  "*at_least_two (Succ (Succ _)) ⟷ True*"
| "*at_least_two _ ⟷ False*"

3.1.3 Mutual Recursion

The syntax for mutually recursive functions over mutually recursive 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_Succ n) = Succ (nat_of_odd_nat n)*"

| "$nat\_of\_odd\_nat$ (Odd_Succ n) = $Succ$ ($nat\_of\_even\_nat$ n)"

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

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

fun
even :: "*nat* ⇒ *bool*" **and**
odd :: "*nat* ⇒ *bool*"
where
"*even Zero* = *True*"
| "*even* (*Succ n*) = *odd n*"
| "*odd Zero* = *False*"
| "*odd* (*Succ n*) = *even n*"

3.1.4 Nested Recursion

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

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 ($\lambda t.$ 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 ($\Rightarrow$) is simply composition (($\circ$)):

primrec *relabel_ft* :: "$('a \Rightarrow 'a) \Rightarrow 'a\ ftree \Rightarrow 'a\ ftree$" **where**
"$relabel\_ft\ f\ (FTLeaf\ x) = FTLeaf\ (f\ x)$"
| "$relabel\_ft\ f\ (FTNode\ g) = FTNode\ (relabel\_ft\ f \circ g)$"

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

primrec *relabel_ft* :: "$('a \Rightarrow 'a) \Rightarrow 'a\ ftree \Rightarrow 'a\ ftree$" **where**
"$relabel\_ft\ f\ (FTLeaf\ x) = FTLeaf\ (f\ x)$"
| "$relabel\_ft\ f\ (FTNode\ g) = FTNode\ (\lambda x.\ relabel\_ft\ f\ (g\ x))$"

primrec (*nonexhaustive*) *subtree_ft* :: "$'a \Rightarrow 'a\ ftree \Rightarrow 'a\ ftree$" **where**
"$subtree\_ft\ x\ (FTNode\ g) = g\ x$"

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

datatype $'a\ ftree2 = FTLeaf2\ 'a \mid FTNode2$ "$'a \Rightarrow 'a \Rightarrow 'a\ ftree2$"

primrec *relabel_ft2* :: "$('a \Rightarrow 'a) \Rightarrow 'a\ ftree2 \Rightarrow 'a\ ftree2$" **where**
"$relabel\_ft2\ f\ (FTLeaf2\ x) = FTLeaf2\ (f\ x)$"
| "$relabel\_ft2\ f\ (FTNode2\ g) = FTNode2\ ((\circ)\ ((\circ)\ (relabel\_ft2\ f))\ g)$"

primrec *relabel_ft2* :: "$('a \Rightarrow 'a) \Rightarrow 'a\ ftree2 \Rightarrow 'a\ ftree2$" **where**
"$relabel\_ft2\ f\ (FTLeaf2\ x) = FTLeaf2\ (f\ x)$"
| "$relabel\_ft2\ f\ (FTNode2\ g) = FTNode2\ (\lambda x\ y.\ relabel\_ft2\ f\ (g\ x\ y))$"

primrec (*nonexhaustive*) *subtree_ft2* :: "$'a \Rightarrow 'a \Rightarrow 'a\ ftree2 \Rightarrow 'a\ ftree2$" **where**
"$subtree\_ft2\ x\ y\ (FTNode2\ g) = g\ x\ y$"

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

primrec *increasing_tree* :: "$int \Rightarrow int\ tree_{ff} \Rightarrow bool$" **where**
"$increasing\_tree\ m\ (Node_{ff}\ n\ ts) \longleftrightarrow$
$n \geq m \wedge list\_all\ (increasing\_tree\ (n + 1))\ ts$"

3.1.5 Nested-as-Mutual Recursion

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

primrec (*nonexhaustive*)
  at_{ff} :: "$'a\ tree_{ff} \Rightarrow nat\ list \Rightarrow 'a$" **and**
  ats_{ff} :: "$'a\ tree_{ff}\ list \Rightarrow nat \Rightarrow nat\ list \Rightarrow 'a$"
where
  "$at_{ff}\ (Node_{ff}\ a\ ts)\ js =$
    $(case\ js\ of$
      $[] \Rightarrow a$
    $|\ j\ \#\ js' \Rightarrow ats_{ff}\ ts\ j\ js')$"
| "$ats_{ff}\ (t\ \#\ ts)\ j =$
    $(case\ j\ of$
      $Zero \Rightarrow at_{ff}\ t$
    $|\ Succ\ j' \Rightarrow ats_{ff}\ ts\ j')$"

Appropriate induction rules are generated as $at_{ff}.induct$, $ats_{ff}.induct$, and $at_{ff}\_ats_{ff}.induct$. The induction rules and the underlying recursors are generated dynamically and are kept in a cache to speed up subsequent definitions.

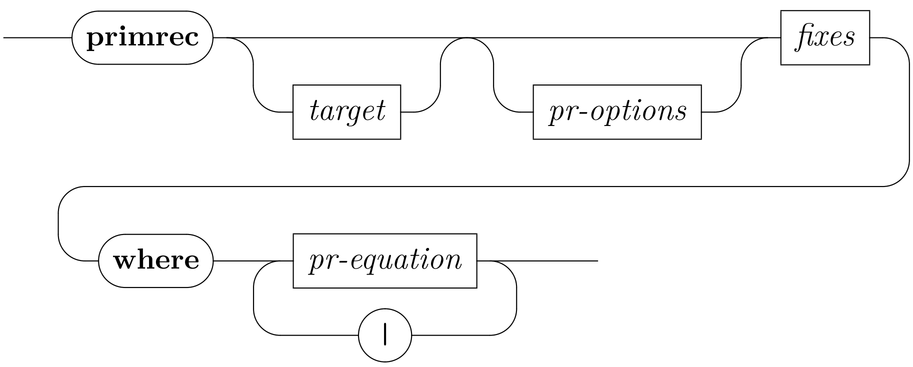
Here is a second example:

primrec
  sum_btree :: "$('a{::}\{zero,plus\})\ btree \Rightarrow 'a$" **and**
  sum_btree_option :: "$'a\ btree\ option \Rightarrow 'a$"
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*"

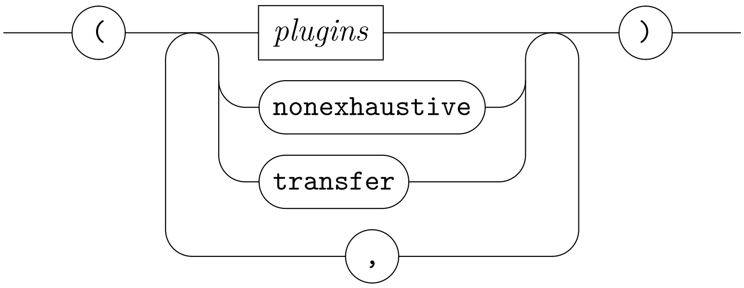
3.2 Command Syntax

3.2.1 primrec

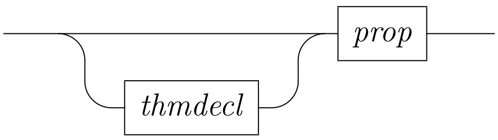
$$\textbf{primrec} \ : \ local\_theory \rightarrow local\_theory$$



pr-options



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 [12].

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

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

3.3 Generated Theorems

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

f.***simps*** [*simp*, *code*]:
: *tfold uu* (*TNil y*) = *y*
: *tfold f* (*TCons x xs*) = *f x* (*tfold f xs*)
: The [*code*] attribute is set by the *code* plugin (Section 8.1).

f.***transfer*** [*transfer_rule*]:
: *rel_fun* (*rel_fun R*2 (*rel_fun R*1 *R*1)) (*rel_fun* (*rel_tlist R*2 *R*1) *R*1) *tfold tfold*
: This theorem is generated by the *transfer* plugin (Section 8.3) for functions declared with the *transfer* option enabled.

f.***induct*** [*case_names* $C_1 \ldots C_n$]:
: This induction rule is generated for nested-as-mutual recursive functions (Section 3.1.5).

f_1_..._f_m.***induct*** [*case_names* $C_1 \ldots C_n$]:
: This induction rule is generated for nested-as-mutual recursive functions (Section 3.1.5). Given $m > 1$ mutually recursive functions, this rule can be used to prove m properties simultaneously.

3.4 Recursive Default Values for Selectors

A datatype selector *un_D* can have a default value for each constructor on which it is not otherwise specified. Occasionally, it is useful to have the default value be defined recursively. This leads to a chicken-and-egg situation, because the datatype is not introduced yet at the moment when the selectors are introduced. Of course, we can always define the selectors 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 $termi_0$ :: *'a*

datatype $('a, 'b)$ *tlist* =
  TNil (*termi*: *'b*)
| *TCons* (*thd*: *'a*) (*ttl*: "(*'a*, *'b*) *tlist*")
where
  "*ttl* (*TNil y*) = *TNil y*"
| "*termi* (*TCons* _ *xs*) = $termi_0$ *xs*"

overloading
  $termi_0$ ≡ "$termi_0$ :: (*'a*, *'b*) *tlist* ⇒ *'b*"
begin
primrec $termi_0$ :: "(*'a*, *'b*) *tlist* ⇒ *'b*" **where**
  "$termi_0$ (*TNil y*) = *y*"
| "$termi_0$ (*TCons x xs*) = $termi_0$ *xs*"
end

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

3.5 Compatibility Issues

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

- *Some theorems have different names.* For $m > 1$ mutually recursive functions, f_1_..._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/Datatype_Examples`. The *Archive of Formal Proofs* also includes some useful codatatypes, notably for lazy lists [7].

4.1 Introductory Examples

4.1.1 Simple Corecursion

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

codatatype (*lset*: $'a$) *llist* =
  lnull: *LNil*
| *LCons* (*lhd*: $'a$) (*ltl*: "$'a$ *llist*")
for
  map: *lmap*
  rel: *llist_all*2
  pred: *llist_all*
where
  "*ltl LNil* = *LNil*"

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

codatatype (*sset*: $'a$) *stream* =
  SCons (*shd*: $'a$) (*stl*: "$'a$ *stream*")
for
  map: *smap*
  rel: *stream_all*2

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

codatatype *enat* = *EZero* | *ESucc enat*

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

Here is an example with many constructors:

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

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

4.1.2 Mutual Corecursion

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

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

4.1.3 Nested Corecursion

The next examples feature *nested corecursion*:

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

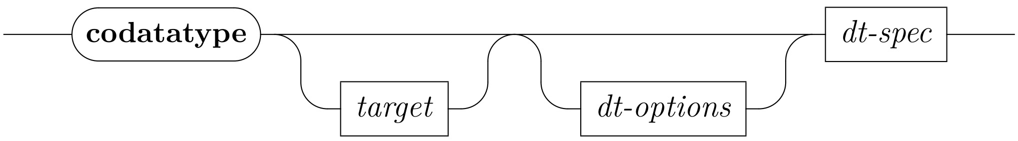
codatatype $'a\ tree_{is} = Node_{is}\ (lbl_{is}\colon\ 'a)\ (sub_{is}\colon$ "$'a\ tree_{is}\ fset$")

codatatype $'a\ sm = SM\ (accept\colon bool)\ (trans\colon$ "$'a \Rightarrow\ 'a\ sm$")

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:

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

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

t_1_…_t_m.***coinduct*** [*case_names* $t_1 \ldots t_m$, *case_conclusion* $D_1 \ldots D_n$]
t_1_…_t_m.***coinduct_strong*** [*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_1_…_t_m.***set_induct*** [*case_names* $C_1 \ldots C_n$,
induct set: $set_j\_t_1, \ldots,$ *induct set*: $set_j\_t_m$]:
$[\![x \in lset\ a;\ \bigwedge z1\ z2.\ P\ z1\ (LCons\ z1\ z2);\ \bigwedge z1\ z2\ xa.\ [\![xa \in lset\ z2;\ P\ xa\ z2]\!] \Longrightarrow P\ xa\ (LCons\ z1\ z2)]\!] \Longrightarrow P\ x\ a$
If $m = 1$, the attribute [*consumes* 1] is generated as well.

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

$p\ a \Longrightarrow corec\_llist\ p\ g21\ q22\ g221\ g222\ a = LNil$
$\neg\ p\ a \Longrightarrow corec\_llist\ p\ g21\ q22\ g221\ g222\ a = LCons\ (g21\ a)\ (if\ q22\ a\ then\ g221\ a\ else\ corec\_llist\ p\ g21\ q22\ g221\ g222\ (g222\ a))$

$t.$***corec_code*** [*code*]:

$corec\_llist\ p\ g21\ q22\ g221\ g222\ a = (if\ p\ a\ then\ LNil\ else\ LCons\ (g21\ a)\ (if\ q22\ a\ then\ g221\ a\ else\ corec\_llist\ p\ g21\ q22\ g221\ g222\ (g222\ a)))$
The [*code*] attribute is set by the *code* plugin (Section 8.1).

$t.$***corec_disc***:

$p\ a \Longrightarrow lnull\ (corec\_llist\ p\ g21\ q22\ g221\ g222\ a)$
$\neg\ p\ a \Longrightarrow \neg\ lnull\ (corec\_llist\ p\ g21\ q22\ g221\ g222\ a)$

$t.$***corec_disc_iff*** [*simp*]:

$lnull\ (corec\_llist\ p\ g21\ q22\ g221\ g222\ a) = p\ a$
$(\neg\ lnull\ (corec\_llist\ p\ g21\ q22\ g221\ g222\ a)) = (\neg\ p\ a)$

$t.$***corec_sel*** [*simp*]:

$\neg\ p\ a \Longrightarrow lhd\ (corec\_llist\ p\ g21\ q22\ g221\ g222\ a) = g21\ a$
$\neg\ p\ a \Longrightarrow ltl\ (corec\_llist\ p\ g21\ q22\ g221\ g222\ a) = (if\ q22\ a\ then\ g221\ a\ else\ corec\_llist\ p\ g21\ q22\ g221\ g222\ (g222\ a))$

$t.$***map_o_corec***:

$lmap\ f \circ corec\_llist\ g\ ga\ gb\ gc\ gd = corec\_llist\ g\ (f \circ ga)\ gb\ (lmap\ f \circ gc)\ gd$

$t.$***corec_transfer*** [*transfer_rule*]:

$rel\_fun\ (rel\_fun\ S\ (=))\ (rel\_fun\ (rel\_fun\ S\ R)\ (rel\_fun\ (rel\_fun\ S\ (=))\ (rel\_fun\ (rel\_fun\ S\ (llist\_all2\ R))\ (rel\_fun\ (rel\_fun\ S\ S)\ (rel\_fun\ S\ (llist\_all2\ R))))))\ corec\_llist\ corec\_llist$
The [*transfer_rule*] attribute is set by the *transfer* plugin (Section 8.3) for type constructors with no dead type arguments.

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

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

4.5 Antiquotation

4.5.1 *codatatype*

The *codatatype* antiquotation, written `\<^codatatype>`‹*t*› or `@{codatatype` *t*`}`, where *t* is a type name, expands to LaTeX code for the definition of the codatatype, with each constructor listed with its argument types. For example, if t is *llist*:

codatatype *'a llist* = *LNil* | *LCons 'a ('a llist)*

5 Defining Primitively Corecursive Functions

Corecursive functions can be specified using the **primcorec** and **primcorecursive** commands, which support primitive corecursion. Other approaches include the more general **partial_function** command, the **corec** and **corecursive** commands, and techniques based on domains and topologies [8]. In this tutorial, the focus is on **primcorec** and **primcorecursive**; **corec** and **corecursive** are described in a separate tutorial [3]. More examples can be found in the directories `~~/src/HOL/Datatype_Examples` and `~~/src/HOL/Corec_Examples`.

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

- The *destructor view* specifies f by implications of the form

  $$\ldots \Longrightarrow is\_C_j\ (f\ x_1\ \ldots\ x_n)$$

  and equations of the form

  $$un\_C_ji\ (f\ x_1\ \ldots\ x_n) = \ldots$$

  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/Datatype_Examples`. The code view is favored in the examples below. Sections 5.1.5 and 5.1.6 present the same examples expressed using the constructor and destructor views.

5.1.1 Simple Corecursion

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

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

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

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

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

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

simps_of_case *lapp_simps*: *lapp.code*

This generates the lemma collection *lapp_simps*:

$$lapp\ LNil\ ys = ys$$
$$lapp\ (LCons\ xa\ x)\ ys = LCons\ xa\ (lapp\ x\ ys)$$

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

primcorec *infty* :: *enat* **where**
"*infty* = *ESucc infty*"

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

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

The main disadvantage of the code view is that the conditions are tested sequentially. This is visible in the generated theorems. The constructor and destructor views offer nonsequential alternatives.

5.1.2 Mutual Corecursion

The syntax for mutually corecursive functions over mutually corecursive datatypes is unsurprising:

```
primcorec
  even_infty :: even_enat and
  odd_infty :: odd_enat
where
  "even_infty = Even_ESucc odd_infty"
| "odd_infty = Odd_ESucc even_infty"
```

5.1.3 Nested Corecursion

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

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

primcorec $iterate_{is}$ :: "$('a \Rightarrow 'a\ fset) \Rightarrow 'a \Rightarrow 'a\ tree_{is}$" **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 :: "$'a\ stream \Rightarrow 'a\ tree_{ii}$" **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 :: "$'a\ stream \Rightarrow 'a\ tree_{ii}$" **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 $(Q, \Sigma, \delta, q_0, F)$, 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* :: "$('q \Rightarrow 'a \Rightarrow 'q) \Rightarrow 'q\ set \Rightarrow 'q \Rightarrow 'a\ sm$" **where**
"*sm_of_dfa* $\delta\ F\ q = SM\ (q \in F)$ (*sm_of_dfa* $\delta\ F \circ \delta\ q$)"

The map function for the function type ($\Rightarrow$) is composition ($(\circ)$). 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* :: "$('q \Rightarrow 'a \Rightarrow 'q) \Rightarrow 'q\ set \Rightarrow 'q \Rightarrow 'a\ sm$" **where**
"*sm_of_dfa* $\delta\ F\ q = SM\ (q \in F)$ ($\lambda a.$ *sm_of_dfa* $\delta\ F\ (\delta\ q\ a)$)"

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

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

primcorec *not_sm* :: "'*a sm* $\Rightarrow$ '*a sm*" **where**
"*not_sm M* = *SM* ($\neg$ *accept M*) (λ*a*. *not_sm* (*trans M a*))"

primcorec *or_sm* :: "'*a sm* $\Rightarrow$ '*a sm* $\Rightarrow$ '*a sm*" **where**
"*or_sm M N* =
SM (*accept M* $\vee$ *accept N*) (λ*a*. *or_sm* (*trans M a*) (*trans N a*))"

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

codatatype ('*a*, '*b*) *sm2* =
SM2 (*accept2*: *bool*) (*trans2*: "'*a* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *sm2*")

primcorec
sm2_of_dfa :: "('*q* $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ '*q*) $\Rightarrow$ '*q set* $\Rightarrow$ '*q* $\Rightarrow$ ('*a*, '*b*) *sm2*"
where
"*sm2_of_dfa* δ *F q* = *SM2* ($q \in F$) (($\circ$) (($\circ$) (*sm2_of_dfa* δ *F*)) (δ *q*))"

primcorec
sm2_of_dfa :: "('*q* $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ '*q*) $\Rightarrow$ '*q set* $\Rightarrow$ '*q* $\Rightarrow$ ('*a*, '*b*) *sm2*"
where
"*sm2_of_dfa* δ *F q* = *SM2* ($q \in F$) (λ*a 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* $\Rightarrow$ '*a llist*) $\Rightarrow$ '*a* $\Rightarrow$ '*a* $tree_{ii}$" **and**
$iterates_{ii}$ :: "('*a* $\Rightarrow$ '*a llist*) $\Rightarrow$ '*a llist* $\Rightarrow$ '*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 $\Rightarrow$ *LNil*
| *LCons x xs'* $\Rightarrow$ *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 *coinduct_strong*. These rules and the underlying corecursors are generated dynamically and are kept in a cache to speed up subsequent definitions.

5.1.5 Constructor View

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

Here is an example where there is a difference:

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

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

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

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

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

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

5.1.6 Destructor View

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

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

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

"_ $\Longrightarrow$ $\neg$ *lnull* (*lapp xs ys*)"

to the specification. The generated selector theorems are conditional.

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

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 ⟹ *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:

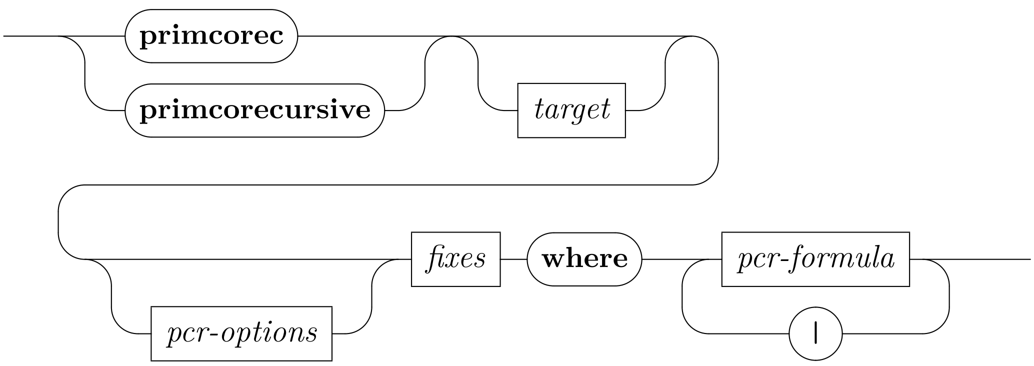
primcorec
even_infty :: *even_enat* **and**
odd_infty :: *odd_enat*
where
“*even_infty* ≠ *Even_EZero*”
| “*un_Even_ESucc even_infty* = *odd_infty*”
| “*un_Odd_ESucc odd_infty* = *even_infty*”

primcorec $iterate_{ii}$ :: “('a ⇒ 'a *llist*) ⇒ 'a ⇒ 'a $tree_{ii}$” **where**
“lbl_{ii} ($iterate_{ii}$ *g x*) = *x*”
| “sub_{ii} ($iterate_{ii}$ *g x*) = *lmap* ($iterate_{ii}$ *g*) (*g x*)”

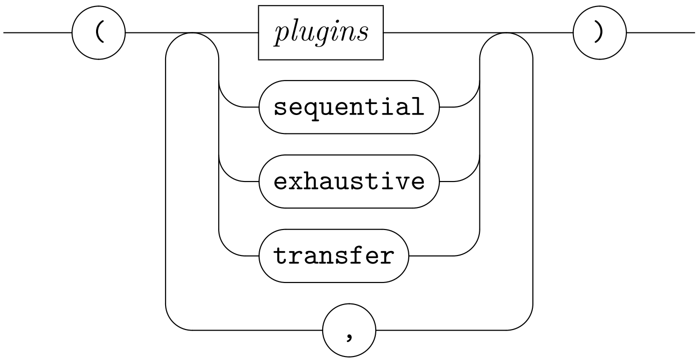
5.2 Command Syntax

5.2.1 primcorec and primcorecursive

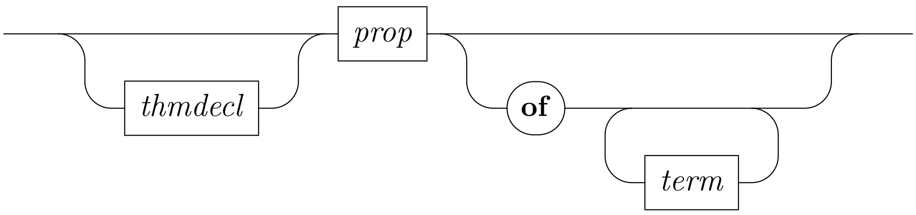
$$\begin{array}{rcl} \textbf{primcorec} & : & \mathit{local\_theory} \to \mathit{local\_theory} \\ \textbf{primcorecursive} & : & \mathit{local\_theory} \to \mathit{proof}(\mathit{prove}) \end{array}$$



pcr-options



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 [12].

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

- The *plugins* option indicates which plugins should be enabled (*only*) or disabled (*del*). By default, all plugins are enabled.
- The *sequential* option indicates that the conditions in specifications expressed using the constructor or destructor view are to be interpreted sequentially.
- The *exhaustive* option indicates that the conditions in specifications expressed using the constructor or destructor view cover all possible cases. This generally gives rise to an additional proof obligation.
- The *transfer* option indicates that an unconditional transfer rule should be generated and proved *by transfer_prover*. The [*transfer_rule*] attribute is set on the generated theorem.

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

5.3 Generated Theorems

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

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

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

f.***disc*** [*simp*, *code*]:
: ¬ *lnull (literate g x)*
  The [*code*] attribute is set by the *code* plugin (Section 8.1). The [*simp*] attribute is set only for functions for which *f.disc_iff* is not available.

f.***disc_iff*** [*simp*]:
: ¬ *lnull (literate g x)*
  This property is generated only for functions declared with the *exhaustive* option or whose conditions are trivially exhaustive.

f.***sel*** [*simp*, *code*]:
: ¬ *lnull (literate g x)*
  The [*code*] attribute is set by the *code* plugin (Section 8.1).

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

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

f.***coinduct*** [*consumes m*, *case_names* t_1 ... t_m, *case_conclusion* D_1 ... D_n]:
: This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4).

f.***coinduct_strong*** [*consumes* m, *case_names* $t_1 \ldots t_m$, *case_conclusion* $D_1 \ldots D_n$]:
This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4).

f_1_..._f_m.***coinduct*** [*case_names* $t_1 \ldots t_m$, *case_conclusion* $D_1 \ldots D_n$]:
This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4). Given $m > 1$ mutually corecursive functions, this rule can be used to prove m properties simultaneously.

f_1_..._f_m.***coinduct_strong*** [*case_names* $t_1 \ldots t_m$, *case_conclusion* $D_1 \ldots D_n$]:
This coinduction rule is generated for nested-as-mutual corecursive functions (Section 5.1.4). Given $m > 1$ mutually corecursive functions, this rule can be used to prove m properties simultaneously.

For convenience, **primcorec** and **primcorecursive** also provide the following collection:

f.***simps*** = f.*disc_iff* (or f.*disc*) t.*sel*

6 Registering Bounded Natural Functors

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

6.1 Bounded Natural Functors

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

An n-ary BNF is a type constructor equipped with a map function (functorial action), n set functions (natural transformations), and an infinite cardinal bound that satisfy certain properties. For example, *'a llist* is a unary BNF. Its predicator *llist_all* :: (*'a* ⇒ *bool*) ⇒ *'a llist* ⇒ *bool* extends unary predicates over elements to unary predicates over lazy lists. Similarly, 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** 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 bundle "cardinal_syntax" included.

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

typedef $('d, 'a)$ *fn* = "*UNIV* :: $('d \Rightarrow 'a)$ *set*"
  by *simp*

setup_lifting *type_definition_fn*

lift_definition *map_fn* :: "$('a \Rightarrow 'b) \Rightarrow ('d, 'a)$ *fn* $\Rightarrow ('d, 'b)$ *fn*" **is** "$(\circ)$" **.**
lift_definition *set_fn* :: "$('d, 'a)$ *fn* $\Rightarrow 'a$ *set*" **is** *range* **.**

lift_definition
  pred_fn :: "$('a \Rightarrow$ *bool*$) \Rightarrow ('d, 'a)$ *fn* $\Rightarrow$ *bool*"
is
  "*pred_fun* $(\lambda\_.$ *True*$)$" **.**

lift_definition
  rel_fn :: "$('a \Rightarrow 'b \Rightarrow$ *bool*$) \Rightarrow ('d, 'a)$ *fn* $\Rightarrow ('d, 'b)$ *fn* $\Rightarrow$ *bool*"
is
  "*rel_fun* $(=)$" **.**

bnf "$('d, 'a)$ *fn*"
  map: *map_fn*
  sets: *set_fn*
  bd: "*natLeq* +c *card_suc* |*UNIV* :: $'d$ *set*|"
  rel: *rel_fn*
  pred: *pred_fn*
proof –
  show "*map_fn id* = *id*"

by *transfer auto*
next
fix f :: "$'a \Rightarrow 'b$" **and** g :: "$'b \Rightarrow 'c$"
show "$map\_fn\ (g \circ f) = map\_fn\ g \circ map\_fn\ f$"
by *transfer* (*auto simp add*: *comp_def*)
next
fix F :: "$('d, 'a)\ fn$" **and** $f\ g$ :: "$'a \Rightarrow 'b$"
assume "$\bigwedge x.\ x \in set\_fn\ F \Longrightarrow f\ x = g\ x$"
then show "$map\_fn\ f\ F = map\_fn\ g\ F$"
by *transfer auto*
next
fix f :: "$'a \Rightarrow 'b$"
show "$set\_fn \circ map\_fn\ f = (`)\ f \circ set\_fn$"
by *transfer* (*auto simp add*: *comp_def*)
next
show "$card\_order\ (natLeq +c\ card\_suc\ |UNIV :: 'd\ set|\ )$"
by (*rule card_order_bd_fun*)
next
show "$cinfinite\ (natLeq +c\ card\_suc\ |UNIV :: 'd\ set|\ )$"
by (*rule Cinfinite_bd_fun*[*THEN conjunct*1])
next
show "$regularCard\ (natLeq +c\ card\_suc\ |UNIV :: 'd\ set|\ )$"
by (*rule regularCard_bd_fun*)
next
fix F :: "$('d, 'a)\ fn$"
have "$|set\_fn\ F| \le o\ |UNIV :: 'd\ set|$" (**is** "$\_ \le o\ ?U$")
by *transfer* (*rule card_of_image*)
also have "$?U <o\ card\_suc\ ?U$"
by (*simp add*: *card_of_card_order_on card_suc_greater*)
also have "$card\_suc\ ?U \le o\ natLeq +c\ card\_suc\ ?U$"
using *Card_order_card_suc card_of_card_order_on ordLeq_csum*2 **by** *blast*
finally show "$|set\_fn\ F| <o\ natLeq +c\ card\_suc\ |UNIV :: 'd\ set|$" **.**
next
fix R :: "$'a \Rightarrow 'b \Rightarrow bool$" **and** S :: "$'b \Rightarrow 'c \Rightarrow bool$"
show "$rel\_fn\ R\ OO\ rel\_fn\ S \le rel\_fn\ (R\ OO\ S)$"
by (*rule*, *transfer*) (*auto simp add*: *rel_fun_def*)
next
fix R :: "$'a \Rightarrow 'b \Rightarrow bool$"
show "$rel\_fn\ R = (\lambda x\ y.\ \exists z.\ set\_fn\ z \subseteq \{(x, y).\ R\ x\ y\} \wedge map\_fn\ fst\ z = x \wedge map\_fn\ snd\ z = y)$"
unfolding *fun_eq_iff relcompp.simps conversep.simps*
by *transfer* (*force simp*: *rel_fun_def subset_iff*)

next
  fix P :: "$'a \Rightarrow bool$"
  show "$pred\_fn\ P = (\lambda x.\ Ball\ (set\_fn\ x)\ P)$"
    unfolding *fun_eq_iff* **by** *transfer simp*
qed

print_theorems
print_bnfs

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

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

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

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

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

copy_bnf $('d, 'a)\ fn$

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

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

copy_bnf $('a, 'z)\ point\_ext$

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

typedef $'a\ nonempty\_list$ = "$\{xs :: 'a\ list.\ xs \neq []\}$" **by** *auto*

The **lift_bnf** command requires us to prove that the set of nonempty lists is closed under the map function and the zip function. The latter only occurs implicitly in the goal, in form of the variable *zs*.

```
lift_bnf 'a nonempty_list
proof -
  fix f and xs :: "'a list"
  assume "xs ∈ {xs. xs ≠ []}"
  then show "map f xs ∈ {xs. xs ≠ []}"
    by (cases xs) auto
next
  fix zs :: "('a × 'b) list"
  assume "map fst zs ∈ {xs. xs ≠ []}" "map snd zs ∈ {xs. xs ≠ []}"
  then show "∃zs'∈{xs. xs ≠ []}.
        set zs' ⊆ set zs ∧
        map fst zs' = map fst zs ∧
        map snd zs' = map snd zs"
    by (cases zs ) (auto intro!: exI[of _ zs])
qed
```

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

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


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

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

lift_bnf 'a myoption
proof -
  fix P :: "'a ⇒ 'b ⇒ bool" and Q :: "'b ⇒ 'c ⇒ bool"
  assume "P OO Q ≠ bot"
  then show "rel_sum P P OO ignore_Inl OO rel_sum Q Q
     ≤ ignore_Inl OO rel_sum (P OO Q) (P OO Q) OO ignore_Inl"
    by (fastforce)
```

next
  fix S :: "$'a$ *set set*"
  let $?eq$ = "$\{(x, x').$ *ignore_Inl* x $x'\}$"
  let $?in$ = "$\lambda A.$ $\{x.$ *Basic_BNFs.setl* $x \cup$ *Basic_BNFs.setr* $x \subseteq A\}$"
  assume "$S \neq \{\}$" "$\bigcap$ $S \neq \{\}$"
  show "$(\bigcap A{\in}S.$ $?eq$ `` $?in$ $A) \subseteq$ $?eq$ `` $?in$ $(\bigcap$ $S)$"
  proof (*intro subsetI*)
    fix x
    assume "$x \in (\bigcap A{\in}S.$ $?eq$ `` $?in$ $A)$"
    with ‹$\bigcap$ $S \neq \{\}$› **show** "$x \in$ $?eq$ `` $?in$ $(\bigcap$ $S)$"
      by (*cases* x) (*fastforce*)+
  qed
    qed

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

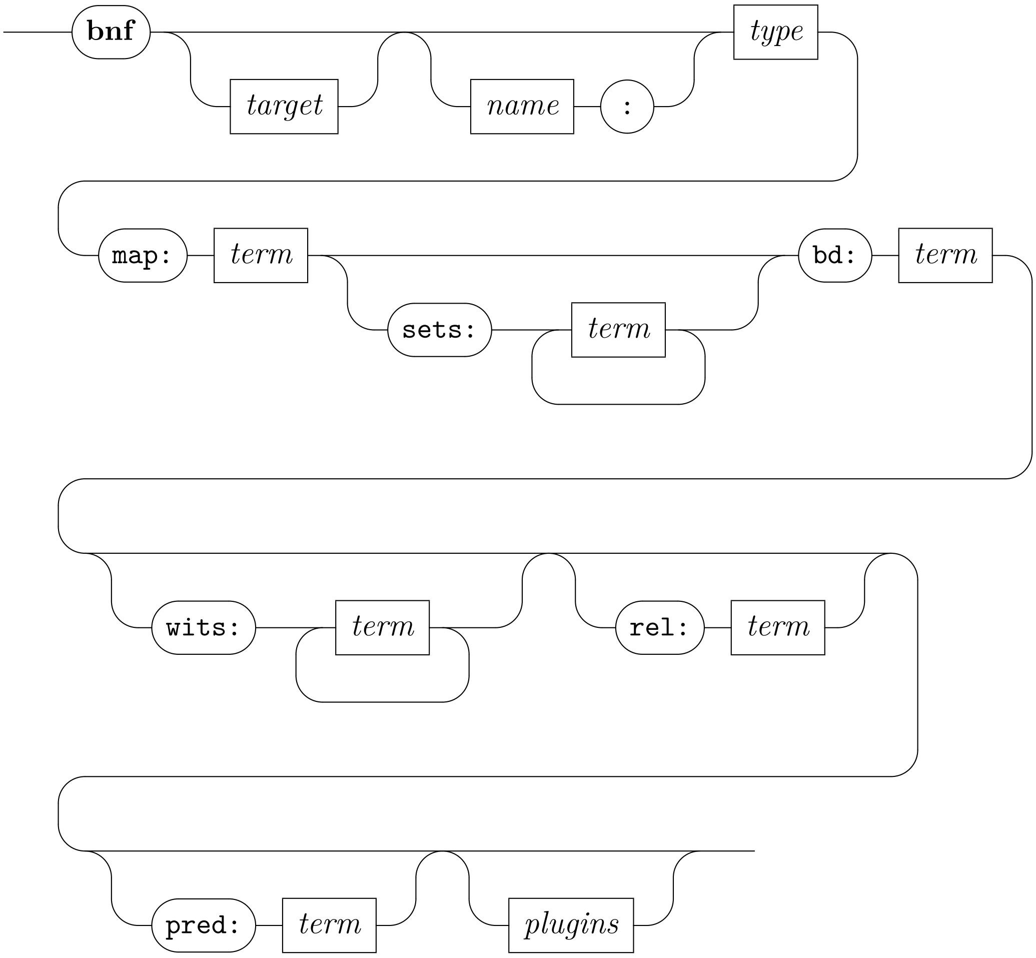
bnf_axiomatization (*setA*: $'a$, *setB*: $'b$, *setC*: $'c$) F
  [*wits*: "$'a \Rightarrow ('a, 'b, 'c)$ F"]

print_theorems
print_bnfs

6.3 Command Syntax

6.3.1 bnf

$$\textbf{bnf} \quad : \quad \textit{local\_theory} \rightarrow \textit{proof}(\textit{prove})$$



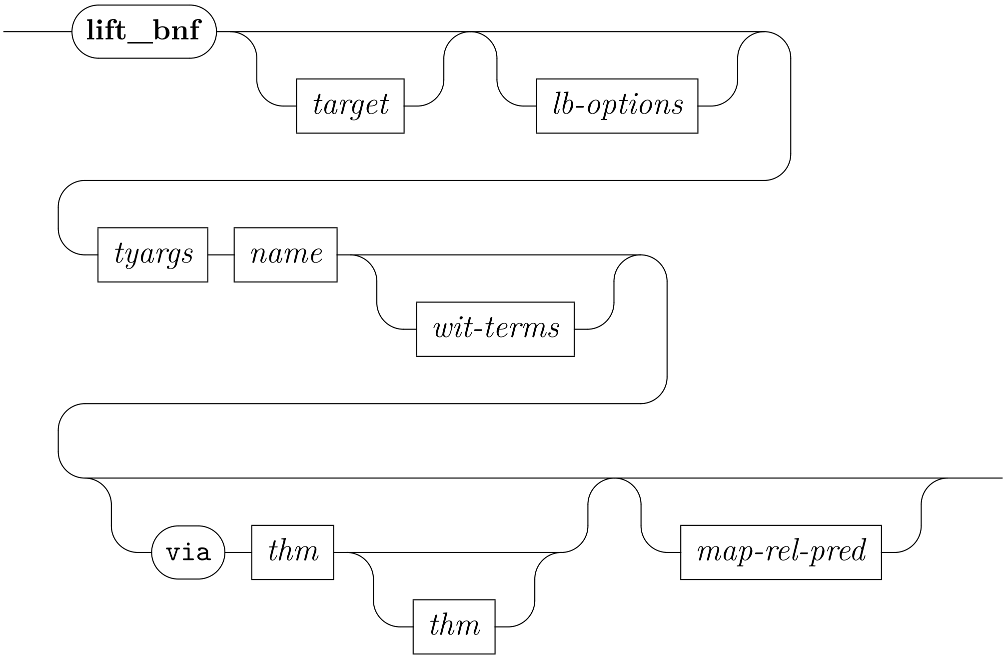
The **bnf** command registers an existing type as a bounded natural functor (BNF). The type must be equipped with an appropriate map function (functorial action). In addition, custom set functions, predicators, relators, and nonemptiness witnesses can be specified; otherwise, default versions are used.

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

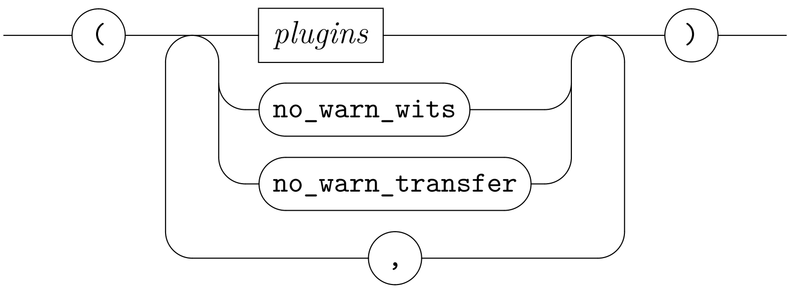
The *plugins* option indicates which plugins should be enabled (*only*) or disabled (*del*). By default, all plugins are enabled.

6.3.2 lift_bnf

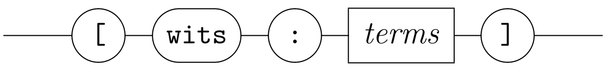
lift_bnf : *local_theory* → *proof*(*prove*)



lb-options



wit-terms

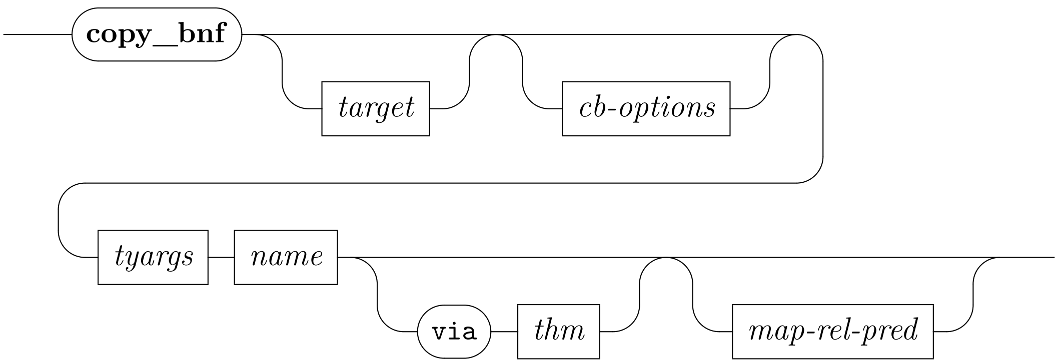


The **lift_bnf** command registers as a BNF an existing type (the *abstract type*) that was defined as a subtype of a BNF (the *raw type*) using the **typedef** command or as a quotient type of a BNF (also, the *raw type*) using the **quotient_type**. To achieve this, it lifts the BNF structure on the raw type to the abstract type following a *type_definition* or a *Quotient* theorem. The theorem is usually inferred from the type, but can also be explicitly supplied by means of the optional *via* clause. In case of quotients, it is sometimes also necessary to supply a second theorem of the form *reflp eq*, that expresses the reflexivity (and thus totality) of the equivalence relation. In addition, custom names for the set functions, the map function, the predicator, and the relator, as well as nonemptiness witnesses can be specified.

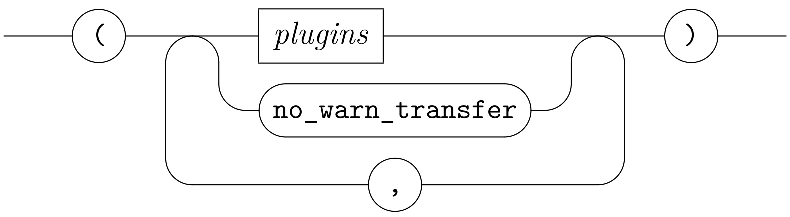
Nonemptiness witnesses are not lifted from the raw type's BNF, as this would be incomplete. They must be given as terms (on the raw type) and proved to be witnesses. The command warns about witness types that are present in the raw type's BNF but not supplied by the user. The warning can be disabled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

6.3.3 copy_bnf

copy_bnf : *local_theory* → *local_theory*



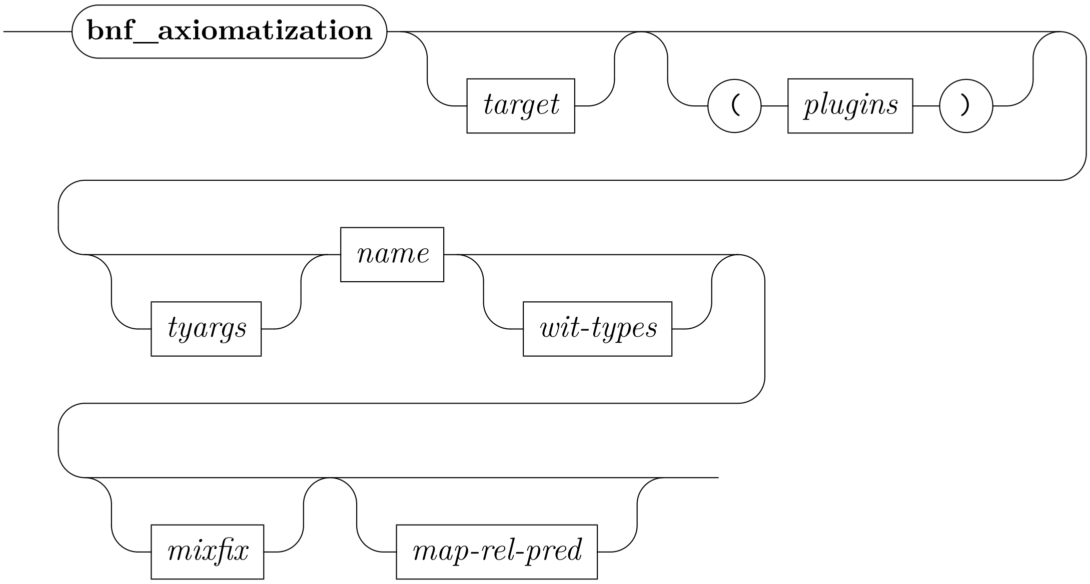
cb-options



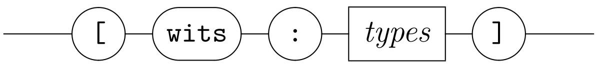
The **copy_bnf** command performs the same lifting as **lift_bnf** for type copies (**typedef**s with *UNIV* as the representing set), without requiring the user to discharge any proof obligations or provide nonemptiness witnesses.

6.3.4 bnf_axiomatization

bnf_axiomatization : *local_theory* → *local_theory*



wit-types



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

The syntactic entity *target* can be used to specify a local context, *name* denotes an identifier, *typefree* denotes fixed type variable (*'a*, *'b*, . . .), *mixfix* denotes the usual parenthesized mixfix notation, and *types* denotes a space-separated list of types [12].

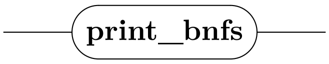
The *plugins* option indicates which plugins should be enabled (*only*) or disabled (*del*). By default, all plugins are enabled.

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

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

6.3.5 print_bnfs

print_bnfs : *local_theory* →



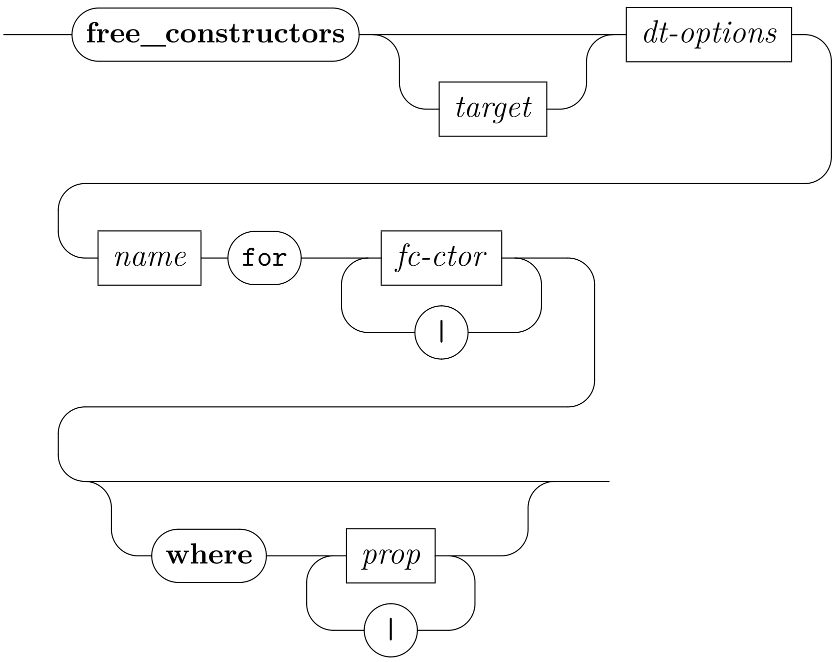
7 Deriving Destructors and Constructor Theorems

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

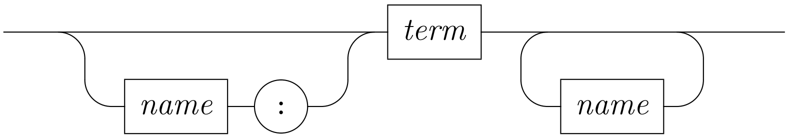
7.1 Command Syntax

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**).

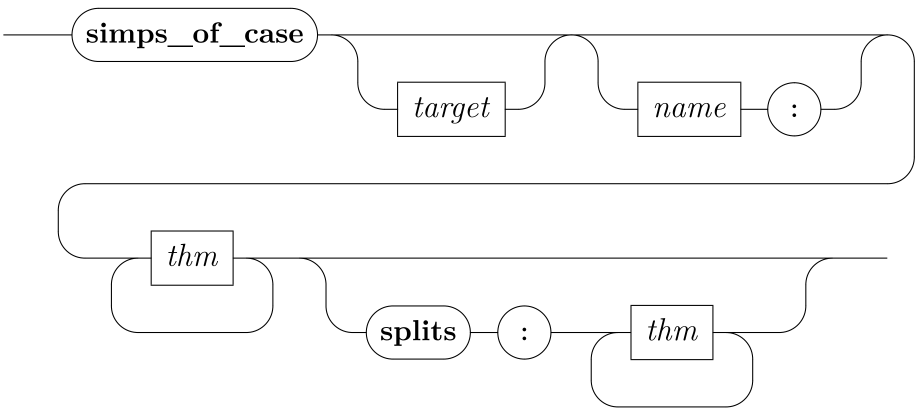
The syntactic entity *target* can be used to specify a local context, *name* denotes an identifier, *prop* denotes a HOL proposition, and *term* denotes a HOL term [12].

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

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

7.1.2 simps_of_case

simps_of_case : *local_theory* → *local_theory*



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

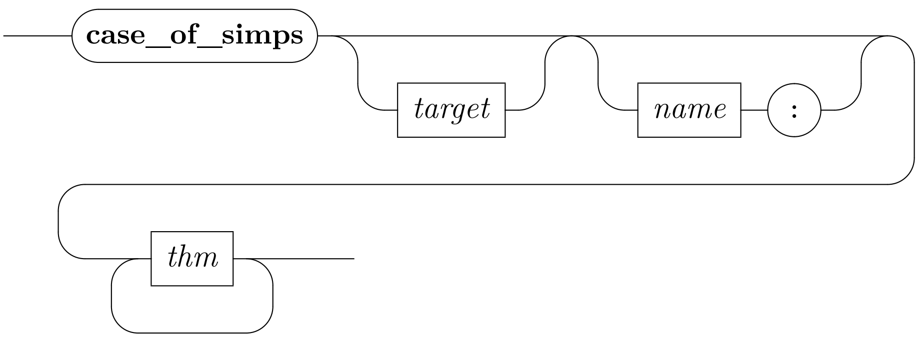
simps_of_case *lapp_simps*: *lapp.code*

translates $lapp\ xs\ ys = (case\ xs\ of\ LNil \Rightarrow ys \mid LCons\ x\ xs' \Rightarrow LCons\ x\ (lapp\ xs'\ ys))$ into

$$lapp\ LNil\ ys = ys$$
$$lapp\ (LCons\ xa\ x)\ ys = LCons\ xa\ (lapp\ x\ ys)$$

7.1.3 case_of_simps

$$\textbf{case\_of\_simps} \ : \ local\_theory \rightarrow local\_theory$$



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

case_of_simps *lapp_case*: *lapp_simps*

translates

$$lapp\ LNil\ ys = ys$$
$$lapp\ (LCons\ xa\ x)\ ys = LCons\ xa\ (lapp\ x\ ys)$$

into $lapp\ xba\ x3a = (case\ xba\ of\ LNil \Rightarrow x3a \mid LCons\ x2ba\ x1ba \Rightarrow LCons\ x2ba\ (lapp\ x1ba\ x3a))$.

8 Selecting Plugins

Plugins extend the (co)datatype package to interoperate with other Isabelle packages and tools, such as the code generator, Transfer, Lifting, and Quickcheck. They can be enabled or disabled individually using the *plugins* option to the commands **datatype**, **primrec**, **codatatype**, **primcorec**, **primcorecursive**, **bnf**, **bnf_axiomatization**, and **free_constructors**. For example:

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

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

8.1 Code Generator

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

For types, the plugin derives the following properties:

t.***eq.refl*** [*code nbe*]:
$equal\_class.equal\ x\ x \equiv True$

t.***eq.simps*** [*code*]:
$equal\_class.equal\ []\ (x21\ \#\ x22) \equiv False$
$equal\_class.equal\ (x21\ \#\ x22)\ [] \equiv False$
$equal\_class.equal\ (x21\ \#\ x22)\ [] \equiv False$
$equal\_class.equal\ []\ (x21\ \#\ x22) \equiv False$
$equal\_class.equal\ (x21\ \#\ x22)\ (y21\ \#\ y22) \equiv x21 = y21 \wedge x22 = y22$
$equal\_class.equal\ []\ [] \equiv True$

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

8.2 Size

For each datatype *t*, the ***size*** plugin generates a generic size function $t.size\_t$ as well as a specific instance $size :: t \Rightarrow nat$ belonging to the *size* type class.

The **fun** command relies on *size* to prove termination of recursive functions on datatypes.

The plugin derives the following properties:

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

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

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

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

The $t.size$ and $t.size\_t$ functions generated for datatypes defined by nested recursion through a datatype u depend on $u.size\_u$.

If the recursion is through a non-datatype u with type arguments $'a_1, \ldots, 'a_m$, by default u values are given a size of 0. This can be improved upon by registering a custom size function of type $('a_1 \Rightarrow nat) \Rightarrow \ldots \Rightarrow ('a_m \Rightarrow nat) \Rightarrow u \Rightarrow nat$ using the ML function `BNF_LFP_Size.register_size` or `BNF_LFP_Size.register_size_global`. See theory `~~/src/HOL/Library/Multiset.thy` for an example.

8.3 Transfer

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

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

$t.$***Domainp_rel*** [*relator_domain*]:
$Domainp\ (list\_all2\ R) = list\_all\ (Domainp\ R)$

$t.$***left_total_rel*** [*transfer_rule*]:
$left\_total\ R \Longrightarrow left\_total\ (list\_all2\ R)$

t.***left_unique_rel*** [*transfer_rule*]:
left_unique R $\Longrightarrow$ *left_unique* (*list_all*2 *R*)

t.***right_total_rel*** [*transfer_rule*]:
right_total R $\Longrightarrow$ *right_total* (*list_all*2 *R*)

t.***right_unique_rel*** [*transfer_rule*]:
right_unique R $\Longrightarrow$ *right_unique* (*list_all*2 *R*)

t.***bi_total_rel*** [*transfer_rule*]:
bi_total R $\Longrightarrow$ *bi_total* (*list_all*2 *R*)

t.***bi_unique_rel*** [*transfer_rule*]:
bi_unique R $\Longrightarrow$ *bi_unique* (*list_all*2 *R*)

For (co)datatypes with at least one live type argument, the plugin sets the [*transfer_rule*] attribute on the following (co)datatypes properties: *t.case_transfer*, *t.sel_transfer*, *t.ctr_transfer*, *t.disc_transfer*, *t.rec_transfer*, and *t.corec_transfer*. For (co)datatypes that further have *no dead type arguments*, the plugin sets [*transfer_rule*] on *t.set_transfer*, *t.map_transfer*, and *t.rel_transfer*.

For **primrec**, **primcorec**, and **primcorecursive**, the plugin implements the generation of the *f.transfer* property, conditioned by the *transfer* option, and sets the [*transfer_rule*] attribute on these.

8.4 Lifting

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

The plugin derives the following property:

t.***Quotient*** [*quot_map*]:
Quotient R Abs Rep T $\Longrightarrow$ *Quotient* (*list_all*2 *R*) (*map Abs*) (*map Rep*) (*list_all*2 *T*)

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

8.5 Quickcheck

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

quickcheck_random
quickcheck_exhaustive
quickcheck_bounded_forall
quickcheck_full_exhaustive
quickcheck_narrowing

8.6 Program Extraction

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

9 Known Bugs and Limitations

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

1. *Defining mutually (co)recursive (co)datatypes can be slow.* Fortunately, it is always possible to recast mutual specifications to nested ones, which are processed more efficiently.
2. *Locally fixed types and terms cannot be used in type specifications.* The limitation on types can be circumvented by adding type arguments to the local (co)datatypes to abstract over the locally fixed types.
3. *The* **primcorec** *command does not allow user-specified names and attributes next to the entered formulas.* The less convenient syntax, using the **lemmas** command, is available as an alternative.
4. *The* **primcorec** *command does not allow corecursion under case–of for datatypes that are defined without discriminators and selectors.*
5. *There is no way to use an overloaded constant from a syntactic type class, such as* 0*, as a constructor.*
6. *There is no way to register the same type as both a datatype and a codatatype.* This affects types such as the extended natural numbers, for which both views would make sense (for a different set of constructors).

7. *The names of variables are often suboptimal in the properties generated by the package.*
8. *The compatibility layer sometimes produces induction principles with a slightly different ordering of the premises than the old package.*

Acknowledgment

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

References

[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, *5th International Conference on Interactive Theorem Proving, ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2014.

[3] J. C. Blanchette, A. Lochbihler, A. Popescu, and D. Traytel. *Defining Nonprimitively Corecursive Functions in Isabelle/HOL*. `https://isabelle.in.tum.de/doc/corec.pdf`.

[4] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In J. Vitek, editor, *24th European Symposium on Programming, ESOP 2015*, volume 9032 of *LNCS*, pages 359–382. Springer, 2015.

[5] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

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

[7] A. Lochbihler. Coinductive. In G. Klein, T. Nipkow, and L. C. Paulson, editors, *The Archive of Formal Proofs*. `https://isa-afp.org/entries/Coinductive.shtml`, Feb. 2010.

[8] A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving — 5th International Conference, ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 341–357. Springer, 2014.

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

[10] C. Sternagel and R. Thiemann. Deriving class instances for datatypes. In G. Klein, T. Nipkow, and L. C. Paulson, editors, *The Archive of Formal Proofs*. `https://isa-afp.org/entries/Deriving.shtml`, March 2015.

[11] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012*, pages 596–605. IEEE, 2012.

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