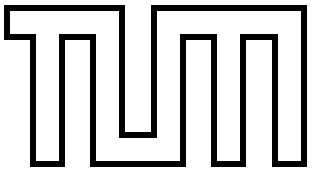


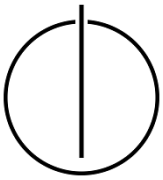
FAKULTÄT FÜR INFORMATIK

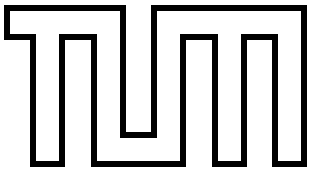
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

Primitively (co)recursive function definitions for Isabelle/HOL

Lorenz Panny





FAKULTÄT FÜR INFORMATIK

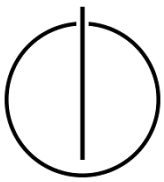
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

Primitively (co)recursive function definitions for Isabelle/HOL

Primitiv-(ko)rekursive Funktionsdefinitionen für Isabelle/HOL

Author:	Lorenz Panny
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Dr. Jasmin Blanchette
Advisor:	Dmitriy Traytel
Date:	August 15, 2014



I assure the single-handed composition of this thesis only supported by declared resources.

Abstract

Isabelle/HOL has lately been extended with a definitional package supporting modular (co)datatypes based on category theoretical constructions. The implementation generates the specified types and associated theorems and constants, notably (co)recursors, but initially, there was no convenient way of specifying functions over these types. This thesis introduces the high-level commands **primrec**, **primcorec** and **primcorecursive** that can be used to define primitively (co)recursive functions over Isabelle's new (co)datatypes using an intuitive syntax. Automating a tedious process, a user specification is internally reduced to a (co)recursor-based definition. Using the (co)recursor theorems, it is then proved and introduced as theorems that the definition does in fact fulfill the specified properties.

Zusammenfassung

Isabelle/HOL wurde kürzlich um ein definitorisches Modul zur Unterstützung modularer Ko-/Datentypen, basierend auf einer kategorientheoretischen Konstruktion, erweitert. Die Implementierung generiert die spezifizierten Typen sowie zugehörige Theoreme und Konstanten (insbesondere Ko-/Rekursoren), aber zunächst stand keine bequeme Methode zum Erzeugen von Funktionen über diesen Typen zur Verfügung. In dieser Arbeit werden die Befehle **primrec**, **primcorec** und **primcorecursive** vorgestellt, mit deren Hilfe unter Benutzung intuitiver Syntax primitiv-(ko)rekursive Funktionen über Isabelles neuen Ko-/Datentypen definiert werden können. Intern automatisieren sie die mühsame Konversion von lesbaren Spezifikationen zu technischen Definitionen unter Verwendung des Ko-/Rekursors. Mithilfe der Ko-/Rekursor-Theoreme wird anschließend bewiesen und als Theorem vermerkt, dass die generierte Definition tatsächlich die Benutzereingabe erfüllt.

Acknowledgements

First of all, I want to thank Professor Tobias Nipkow for allowing me to participate in this work and compile this thesis under his supervision. His undergraduate lecture introduced me to the wonderful world of functional programming and, ultimately, automated theorem proving. He also made it possible for me to present my results at the *Isabelle Workshop 2014* in Vienna.

I am very grateful to my advisors Jasmin Blanchette and Dmitriy Traytel, with both of whom it was highly enjoyable collaborating. The numerous examples they provided helped a lot in analyzing the problems at hand and implementing as well as testing the solution. I also appreciate very much the countless valuable comments on drafts of this thesis as well as other insightful communication on the subject at hand.

Contents

1 Introduction

Recursive functions are a substantial means of expressing recurring computations in functional programming languages. One such language is *higher-order logic* (*HOL*), forming the foundation of the popular theorem proving environment *Isabelle/HOL*. Isabelle is a generic theorem prover and proof checker [10] capable of reasoning about different object logics including, but not limited to, HOL. It follows the LCF philosophy [6], featuring a small inference kernel, and uses a definitional approach which discourages the introduction of additional axioms and instead strives to use definitions whenever possible. This is tedious, but there is hope that minimizing the amount of code and axiomatization that needs to be trusted avoids inconsistencies. Thus, unlike in other theorem provers, Isabelle/HOL's algebraic (co)datatypes are not intrinsic to the object logic (such as in Coq [5]) or brought into being by axiomatization (as in PVS [11]), but explicitly constructed in HOL and checked by Isabelle's inference kernel. The method of choice for this construction is the category theoretical notion of a *bounded natural functor* (*BNF*), "an enriched type constructor satisfying specific properties preserved by interesting categorical operations." [13]

Isabelle/HOL's new (co)datatype package [13] provides means to define compositional algebraic (co)datatypes, but until recently, there was no convenient way of iterating (co)recursively over these types. For some cases, there have been methods to do so, including limited commands covering special cases or directly using low-level constructions and theorems generated by **datatype** and **codatatype**.

When trying to define (co)recursive functions, there is a central problem. Users could, in principle, introduce functions axiomatically, but this is dangerous: consider the example wrong :: *nat* ⇒ *nat* over the datatype *nat* of natural numbers specified by

$$\mathsf{wrong}\ n\ =\ 1\ +\ \mathsf{wrong}\ n.$$

This definition is clearly inconsistent with the properties of *nat* since it implies $0 = 1$, and therefore must not be established as an axiom. Because it is impossible to decide whether an arbitrary recursive specification is consistent (as a consequence to Rice's theorem), the definitional approach is justified: We do not postulate the existence of a function that fulfills the user specification, but rather construct one that does (i.e. find a definition f $=$ rhs for f such that rhs does not contain f). The *primitive* prefix to (co)recursion denotes useful classes of functions where constructing such a definition is always possible, which is why they're detached as an important special case.

Primitive recursion means that in each recursive descent, exactly one constructor is peeled off the function argument through which recursion is performed. In particular, for finite inputs, i.e. having only a finite number of constructors (like all datatype values), this forces the recursion to terminate[1] in any case. As a consequence, no inconsistencies can be introduced: Termination implies that a function application can at some point be written as

[1] Note that Isabelle does not have any notion of computation — whenever this thesis mentions "termination" of a function, it should strictly say "termination of a recursive specification", i.e. *if we were* to repeatedly unfold the specification, this would eventually yield a term not containing the recursive function. More formally, this means that there is a well-founded partial order on the function arguments such that the arguments become smaller with each unfolding, rendering the function's result fully specified.

a term not containing the original invocation, and since this term is unambiguous, the definition preserves consistency. A well-known example is the length function over lists, which can be defined as follows:

```
primrec length :: α list ⇒ int where
  length Nil = 0
| length (Cons _ xs) = 1 + length xs
```

*Primitive **co**recursion* is dual in the sense that while primitive recursion removes one constructor per iteration, primitive corecursion *adds* one layer of constructors during each corecursive descent. We shall call this property *productivity*. The main difference to datatypes is that codatatypes can have infinite values and thus, it is desirable to *not* require a corecursive function to terminate. Consistency is guaranteed nonetheless due to productivity: Since arbitrarily precise finite approximations of a primitively corecursive function's result can be obtained by repeatedly unfolding its specification, and codatatype values are uniquely characterized by their observations, there is exactly one value that matches the specification for each given input.

An example is the function nats, which takes a natural number n and outputs an infinite stream of *all* numbers that are greater than or equal to n:

```
primcorec nats :: nat ⇒ nat stream where
  shd (nats n) = n
| stl (nats n) = nats (n + 1)
```

To demonstrate that there is a non-productive specification that implies False, consider the simple example

```
f :: bool stream
f = smap Not f
```

that is *not* primitively corecursive. Since streams are always infinite and thus non-empty, this identity implies $b = \neg b$ for some boolean b. On the other hand, there are non-productive corecursive specifications hat do not lead to a contradiction. Lochbihler and Hölzl recently showed how to generalize the well-known filter function (which is not productive) to infinite lists in a consistent way [8].

The command **primrec** is intended as a compatible analogue and, eventually, replacement for Berghofer and Wenzel's homonymous **primrec** [2] for the old datatype package.

The command **primcorec** and its more powerful variant **primcorecursive** are capable of defining potentially non-terminating corecursive functions over codatatypes, using a variety of syntaxes suitable to different kinds of tasks and personal preferences.

The author's contribution within the scope of this thesis was the implementation of both commands' term-level functionality, some details of which are described in sections 4 and 7. His advisors provided the underlying requirements, notably: nested-to-mutual reduction, proof tactics and interfaces to the BNF data structures.

The *ITP 2014* conference paper "Truly modular (co)datatypes for Isabelle/HOL" [3] describes some of this work along with other recent Isabelle development concerning (co)datatypes.

The *Isabelle Workshop 2014* paper "Primitively (co)recursive definitions for Isabelle/HOL" [9] is based on a draft of this thesis.

2 Datatypes

One of the most fundamental concepts in most functional programming languages is that of an *algebraic datatype*. For most purposes, they are the preferred way of creating new types and combining existing ones into more complex (and useful) composite types. Probably the most popular example is the datatype of finite lists, which can be defined by

$$\begin{aligned}\textbf{datatype}\ \alpha\ \textit{list}\ (\textbf{map}\colon \textit{map}) \ &= \ \mathsf{null}\colon \mathsf{Nil} \\ &\mid \mathsf{Cons}\ (\mathsf{hd}\colon \alpha)\ (\mathsf{tl}\colon \alpha\ \textit{list})\end{aligned}$$

using Isabelle's **datatype**[1] command. This invocation defines a new type α *list* with one type parameter α, together with two constructors:

- Nil, which indicates an empty list and can be detected using the *discriminator* null :: α *list* $\Rightarrow$ *bool*;
- Cons, taking two parameters of type α and α *list*, which constructs a new list from a single item (the new list's head) and the rest of the list (its tail). The functions hd :: α *list* $\Rightarrow$ α and tl :: α *list* $\Rightarrow$ α *list* that extract Cons's arguments are called *selectors*.

This simple example shows a basic concept of algebraic datatypes: They can recursively contain a number of other types (including themselves). A function that takes a (recursive) datatype as one of its argument types can thus re-apply itself (or, in general, another function) to some term containing the value of the datatype that is enclosed in the function argument. Inevitably, the problem of possible non-termination (and thus, potential inconsistency) arises.

Isabelle can solve this problem by requiring the user to provide proof that their specification will definitely terminate. However, many important functions over datatypes fall into a class for which there are simple (easily automatable) criteria that guarantee termination. This class is the set of *primitively recursive functions*, described in detail in section 3.

The **datatype** command allows the definition of *mutually recursive datatypes*. These are characterised by the simultaneous specification of two or more datatypes that recursively contain each other in some of their constructor arguments. A typical example is a type of finitely branching trees of finite depth and associated (finite) forests:

$$\begin{aligned}\textbf{datatype}\ \alpha\ \textit{tree}_{\mathsf{M}} \ &= \ \mathsf{TEmpty}_{\mathsf{M}} \\ &\mid \mathsf{TNode}_{\mathsf{M}}\ (\mathsf{tval}_{\mathsf{M}}\colon \alpha)\ (\mathsf{children}_{\mathsf{M}}\colon \alpha\ \textit{forest}) \\ \textbf{and}\ \alpha\ \textit{forest} \ &= \ \mathsf{fempty}\colon \mathsf{FNil} \\ &\mid \mathsf{FCons}\ (\mathsf{fhd}\colon \alpha\ \textit{tree}_{\mathsf{M}})\ (\mathsf{ftl}\colon \alpha\ \textit{forest})\end{aligned}$$

[1] In the current Isabelle release, the new **datatype** is suffixed with "**_new**" to distinguish it from the old command. In this thesis, the suffix is omitted and **datatype** shall always denote **datatype_new**.

The internal constructions require that each set of mutually recursive datatypes have identical type parameters (in this case, α *tree*$_{\mathsf{M}}$ and α *forest* share α).

Another, more involved example is the following representation for a simple arithmetic expression language:

datatype *expr* = Expr_Sum *expr_sum*
| Expr_Prod *expr_prod*
| Expr_Const *nat*
and *expr_sum* = Sum *expr expr*
and *expr_prod* = Prod *expr expr*

In addition to mutual recursion, it is often desirable to incorporate existing datatypes into newly defined types. This enables reuse of existing datatypes and, notably, theorems and functions concerning these types. In many cases, it also enables a more intuitive way of specifying a datatype and eases understanding the type's structure. For example, a type equivalent to α *tree*$_{\mathsf{M}}$ can be obtained by reusing the generic *list* type instead of creating a dedicated *forest* type:

datatype α *tree* = tempty: TEmpty | TNode (tval: α) (children: α *tree list*)

This construction is an example of a *nested recursive datatype*. In general, we speak of nested recursion when a datatype (in this case: α *tree*) recursively occurs under an existing type constructor (*list*). As a second example, non-empty binary trees can be defined via

datatype α *option* = is_none: None | Some (the: α)
datatype α *btree* = BNode (bval: α) (left: α *btree option*) (right: α *btree option*),

where the α *btree* type is nested inside the *option* type constructor.

The internal constructions support arbitrary combinations of mutual and nested recursion, with one notable exception: Whenever types occur nested inside the function type $\alpha \Rightarrow \beta$, they must appear on the function's right-hand side, that is, in its result type. For example,

datatype α *wrong* = L α | N (α *wrong* $\Rightarrow$ α)

is rejected by the **datatype** command, whereas

datatype α *ftree* = FTLeaf α | FTNode ($\alpha \Rightarrow \alpha$ *ftree*)

is permitted.

3 Primitively recursive functions

As noted in section 2, *primitively recursive functions* are always guaranteed to terminate. This stems from the requirement that the recursion keeps pattern-matching the same argument position and the argument passed in this position becomes smaller (in terms of the total number of nested constructors) in each recursive descent. Since datatype values always have a finite number of constructors, this implies that at some point, recursion will hit a base case and terminate, thus ensuring consistency.

3.1 Simple recursion

A *primitively recursive* function f over a datatype $(\alpha_1\ \ldots\ \alpha_\tau)\ t$ is characterised by peeling off one of $(\alpha_1\ \ldots\ \alpha_\tau)\ t$'s constructors in each recursive descent. The definition of such functions via the **primrec** command is of the general form

primrec f :: $\ldots \Rightarrow (\alpha_1\ \ldots\ \alpha_\tau)\ t \Rightarrow\ \ldots \Rightarrow \beta$ **where**
  f $\ldots$ (C $a_1\ \ldots\ a_i$) $\ldots$ = $\ldots$
| f $\ldots$ (D $a_1\ \ldots\ a_j$) $\ldots$ = $\ldots$
| $\ldots$,

where C, D, etc. are distinct constructors of the recursive type. The right-hand sides may involve recursively passing some of the constructor arguments a_1 to a_k to f. To be able to guarantee that the recursive function argument becomes syntactically smaller in each descent, it is required that the constructor arguments shall not be modified by passing them to a different function — especially not by introducing a constructor, as this would permit inconsistent definitions like

wrong (C x) = wrong (C x) + 1.

The usage of other arguments that are not contained in the constructor pattern is not constrained in any way; they may vary arbitrarily.

The constructor pattern (C $a_1\ \ldots\ a_n$) may be located at an arbitrary argument position, but its position needs to be consistent throughout the whole function. To keep the implementation simple, no further pattern matching in the constructor pattern is allowed — the pattern needs to be a single, fully applied constructor.

Other examples are the app and rev functions on lists that append two lists over the same type or reverse the order of a list's elements, respectively:

primrec app :: α *list* $\Rightarrow$ α *list* $\Rightarrow$ α *list* **where**
  app Nil = id
| app (Cons x xs) = Cons x $\circ$ app xs

primrec rev :: α *list* $\Rightarrow$ α *list* **where**
  rev Nil = Nil
| rev (Cons x xs) = app (rev xs) (Cons x xs)

The **primrec** command expects to find an equation for each of the recursive datatype's constructors — otherwise, a warning is printed. Since there are cases where the specification is intended to be incomplete, the *nonexhaustive* option can be used to suppress this warning. For example:

```
primrec (nonexhaustive) last :: α list ⇒ α where
  last (Cons x xs) = (if null xs then x else last xs)
```

The behaviour of last on input Nil is then unspecified.

3.2 Mutual recursion

In the case of mutually recursive datatypes $(\alpha_1 \ \ldots \ \alpha_\tau)\ t_1$ to $(\alpha_1 \ \ldots \ \alpha_\tau)\ t_\mu$ (that is, they contain one another, wrapped in constructors), one can define *mutually recursive* functions. This is done by the following general command:

```
primrec
  f1 :: ... ⇒ (α1 ... ατ) t1 ⇒ β1 and
  ... and
  fμ :: ... ⇒ (α1 ... ατ) tμ ⇒ βμ
where
  f1 ... (C a1 ... ai) ... = ...
| f1 ... (D a1 ... aj) ... = ...
| ...
| fμ ... (E a1 ... ak) ... = ...
| fμ ... (F a1 ... aℓ) ... = ...
| ...
```

Note that for one single type, this reduces to the "simple recursion" described above. The right-hand sides may contain recursive calls to any of the functions f_n, passing it any pattern-matched constructor argument.

For example, arithmetic expressions represented by the datatype introduced in section 2 can be evaluated to a natural number using

```
primrec
  eval :: expr ⇒ nat and
  eval_sum :: expr_sum ⇒ nat and
  eval_prod :: expr_prod ⇒ nat
where
  eval (Expr_Sum s) = eval_sum s
| eval (Expr_Prod p) = eval_prod p
| eval (Expr_Const c) = c
| eval_sum (Sum a b) = eval a + eval b
| eval_prod (Prod a b) = eval a * eval b.
```

3.3 Nested recursion

In addition to the mechanisms described in the previous sections, some datatypes also allow *nested recursion*. This is possible whenever a datatype contains another embedded

inside an existing type constructor. An example is the datatype of non-empty, finitely branching trees of finite depth, defined in section 2. It contains itself in the second constructor argument, nested inside the *list* type constructor. In cases like this, it is possible to recurse through the wrapping type's *map function* (which is just called map for *list*s), for example:

```
primrec mirror :: α tree ⇒ α tree where
  mirror (TNode x cs) = TNode x (rev (map mirror cs))
```

There are some exceptions to the rule that a recursive function must be *directly* applied to an unmodified constructor argument (which generally carries over to nested recursion in a natural way). This is due to many users preferring to write

$$\mathsf{map}\ (g \circ \ldots \circ \mathsf{f})\ x,$$

where f is one of the functions to be defined, rather than

$$\mathsf{map}\ g\ (\mathsf{map}\ \ldots\ (\mathsf{map}\ \mathsf{f}\ x)\ldots).$$

The latter variant is already covered by the syntax permitted according to the above; thus, since these specifications are equivalent, we do not risk inconsistency by additionally supporting the first input style.

Similar reasoning applies to map function arguments using λ-abstraction in lieu of composition, resulting in (sub)terms like

$$\mathsf{map}\ (\lambda v.\ g\ (\ldots\ (\mathsf{f}\ v)\ldots))\ x.$$

This form is especially useful when the mapped function expects more non-recursive arguments after its recursive parameter:

```
primrec tree_apply :: (α ⇒ α) tree ⇒ α ⇒ α tree where
  tree_apply (TNode v cs) x = TNode (v x) (map (λt. tree_apply t (v x)) cs)
```

generates a function that creates a tree of values from a tree of endomorphisms and an initial value, storing all intermediate values along the paths from the root to the leaves.[1]

3.4 Nested-as-mutual recursion

The possibility to use nested recursive datatypes in function definitions as if they were mutually recursive exists mainly for compatibility with the old datatype package, but it is useful in its own right. *Nested-to-mutual reduction* [3] intuitively corresponds to unfolding the nesting type's definition inside the nested recursive type's definition. For example,

[1] Note that this definition would otherwise need to be written as

$$\ldots\ \mathsf{map}\ ((\lambda r.\ r\ (v\ x)) \circ \mathsf{tree\_apply})\ cs\ \ldots,$$

which is rather cumbersome, or even worse, without making use of the first exception:

$$\ldots\ \mathsf{map}\ (\lambda r.\ r\ (v\ x))\ (\mathsf{map}\ \mathsf{tree\_apply}\ cs)\ \ldots$$

```
primrec
  tree_apply_M :: (α ⇒ α) tree ⇒ α ⇒ α tree and
  trees_apply :: (α ⇒ α) tree list ⇒ α ⇒ α tree list
where
  tree_apply_M (Node f cs) = Node (f x) (trees_apply cs (f x))
| trees_apply Nil _ = Nil
| trees_apply (Cons t ts) x = Cons (tree_apply_M t x) (trees_apply ts x)
```

is a mutually recursive implementation of tree_apply over the *nested* recursive datatype $(\alpha \Rightarrow \alpha)$ *tree*. The types that the nested-to-mutual reduction simulates are isomorphic to the $tree_M$ and *forest* types presented in section 2, instantiated with $\alpha \Rightarrow \alpha$.

3.5 Recursion through function types

A noteworthy special case of nested recursion is recursion through $\Rightarrow$, the function type. It was noted in section 2 that a datatype may only occur recursively on a function arrow's right-hand side, as in the *ftree* example. The function type's map function is composition, as can be seen below in FTNode's argument:

```
primrec ftree_map :: (α ⇒ α) ⇒ α ftree ⇒ α ftree where
  ftree_map f (FTLeaf x) = FTLeaf x
| ftree_map f (FTNode φ) = FTNode (ftree_map f ∘ φ)
```

For convenience and verbosity, specifying the equivalent expression

```
(λv. ftree_map f (φ v))
```

as FTNode's argument is (similarly to the special syntaxes for map function arguments described in section 3.3) supported as well.

4 primrec's implementation

Before the new **primrec** command's introduction, users had to provide suitable arguments to a new-style datatype's *recursor* in order to perform recursion. Using the theorems associated with the recursor, it is then possible to prove statements about the newly defined function. The **primrec** command automates this process: Given high-level specifications of a function's desired behaviour, it assembles a recursor-based definition and introduces the user specification as theorems.

4.1 Recursors

A datatype's *recursor* performs the most general variant of primitive recursion over this type. Its properties are perhaps best explained by looking at an example: Consider the datatypes α *list* and α *tree* from section 2, which have

$$\mathsf{rec\_list} :: \beta \Rightarrow (\alpha \Rightarrow \alpha\ \mathit{list} \Rightarrow \beta \Rightarrow \beta) \Rightarrow \alpha\ \mathit{list} \Rightarrow \beta$$
$$\mathsf{rec\_tree} :: \beta \Rightarrow (\alpha \Rightarrow (\alpha\ \mathit{tree} \times \beta)\ \mathit{list} \Rightarrow \beta) \Rightarrow \alpha\ \mathit{tree} \Rightarrow \beta$$

as their recursors. In the following, we shall refer to all but the last of the recursor's arguments as its *behavioural functions*. The recursors' semantics are illustrated by the following theorems:

list.rec:

$$\mathsf{rec\_list}\ n\ c\ \mathsf{Nil} = n$$
$$\mathsf{rec\_list}\ n\ c\ (\mathsf{Cons}\ x\ xs) = c\ x\ xs\ (\mathsf{rec\_list}\ n\ c\ xs)$$

tree.rec:

$$\mathsf{rec\_tree}\ e\ n\ \mathsf{TEmpty} = e$$
$$\mathsf{rec\_tree}\ e\ n\ (\mathsf{TNode}\ v\ cs) = n\ x\ (\mathsf{map}\ (\lambda t.\ (t,\ \mathsf{rec\_tree}\ e\ n\ t)\ cs))$$

The first equation for each recursor is simple: it states that the first behavioural function is the constant function value for the base case, Nil or TEmpty. The second equation describes the recursor's behaviour for a composite input: In Cons's case, the constructor arguments x and xs are passed to the function c together with the result of passing the list's tail, xs, to the same instantiation of the recursor. To each child node of a TNode, the result of passing this child to the recursive function is paired to the node, giving the type (α *tree* $\times \beta$) *list*.

In general, a set of mutually recursive datatypes' recursors take one function per constructor as their arguments. They serve as a description of how the constructor arguments and, possibly, recursive calls' results are combined to give the desired return values. The behavioural function for a constructor is passed

- for a constructor argument through which recursion cannot be performed: the constructor argument's unmodified value;

- for a constructor argument through which mutual recursion can be performed: the constructor argument's value ("unmodified") as well as the result of passing this value to a recursive call ("modified");
- for a constructor argument through which nested recursion through a map function can be performed: a tuple containing the original constructor argument's unmodified value and, just like for mutual recursion, the recursive call's result.

Given all these arguments, the recursor returns a function with the desired semantics.

Each datatype partaking in mutual recursion has its own recursor, but they only differ in their return type and last argument's type, which is always the type that the recursor corresponds to (and, equivalently, consumes). Each recursor serves thus as its associated type's entry point to mutual recursion.

Let us take a look at an example: The length function for lists can be defined as follows:

definition length :: α *list* $\Rightarrow$ *nat* **where**
  length $\equiv$ rec_list 0 (λ_ _ r. $1 + r$)

Most likely, the user will wish to have more readable properties of length available than this (technical) definition. Using the *list.rec* theorems, they can derive the usual characterization presented in the introduction:

lemma length_Nil: length Nil $=$ 0
  unfolding length_def list.rec ..

lemma length_Cons: length (Cons x xs) $=$ $1 +$ length xs
  unfolding length_def list.rec ..

4.2 General procedure

The **primrec** implementation's main goal is automating the process sketched in section 4.1. The handling of specifications consists of the following steps, performed in order:

1. From each equation, extract
   - the function that the equation talks about;
   - the recursive type (deduced from the function's argument types and the constructor pattern position);[1]
   - the pattern-matched constructor and its arguments' names;
   - names and types of other (non-pattern) arguments left and right of the constructor pattern;
   - the defined function's real result type after the constructor pattern has been moved to the first argument position;
   - the equation's right-hand side term;

[1] Note that there are generally more than one datatype arguments to a primitively recursive function, but only one of them — that we can't identify using only the function's type information — is the recursive type.

- the unmodified user specification that is proved as a theorem later on.

2. Use the list of recursive types and corresponding result types to get the relevant information (constructors, recursors, theorems) from the datatype database. In this step, the nested-to-mutual reduction (if needed, as can be detected from the recursive types and forms of recursive calls) is performed transparently to the following steps.

3. With the help of this information, traverse the right-hand side of each equation in order to find recursive calls and replace them by a non-recursive term that will eventually use the additional arguments the recursor provides to describe the recursion. This process is explained in more detail in section 4.3.

4. Apply λ-abstractions to each right-hand side to make it accept the additional arguments the recursor provides to the behavioural function.

5. Use each of these modified right-hand sides as an argument to the recursors (filling up missing specifications with undefined).

6. For each recursor, permute the resulting term's arguments in order to reverse the constructor pattern shifting using simple λ wrappers (section 4.4) and define the desired function as equal to this term.

7. Using the recursor theorems, prove that this definition does in fact fulfill the user specification (the function's *characteristic theorems*).

4.3 Eliminating recursive calls

The second step mentioned above depends on information about the structure of recursion in the upcoming definition. This poses a chicken-and-egg problem, since it's specifically this step that supplies all the information about the involved types and their properties. In order to overcome this difficulty, the process of eliminating recursion from the specification is splitted into two parts. First, everything that *looks like* a recursive call is extracted and passed to the underlying machinery. During this step, the structure of the calls is checked against the information from the datatype database and invalid specifications are rejected. After that, the exact datatype information is used to accurately substitute the recursive terms by a recursor-based equivalent.

In more detail, for an equation

$$\mathsf{f}_m\ l_1\ \ldots\ l_p\ (\mathsf{C}\ x_1\ \ldots\ x_n)\ r_1\ \ldots\ r_q\ =\ rhs,$$

the following steps are performed:[2]

1. Starting with t set to rhs, do the following.
   a) If t is a λ-abstraction, that is, of the form $\lambda v.\ t'$, apply this procedure to t' instead.
   b) If t is not a function application, stop. Otherwise, write t as $G\ a_1\ \ldots\ a_k$ with G not an application.

[2] We assume for simplicity that map takes only one argument. In general, its structure is more complicated.

c) If none of the a_j is a constructor argument x_i, this is not a recursive call. Since there might be recursive calls in composite subterms, recursively apply this procedure to G and each of the a_j.

d) Define g as the partial application of G to the first a_j, excluding the constructor argument x_i. This means that

$$t = G\, a_1 \ldots x_i \ldots a_k = g\, x_i \ldots.$$

If g does not contain any of the f_j as a subterm, stop. Otherwise, g is recursively applied to x_i and t is a recursive call.

2. Query the datatype database.

3. Using the newly obtained information, traverse the right-hand side rhs again to convert any legal recursive calls to equivalent terms which use the additional arguments provided by the recursor. This involves applying the following transformations to a subterm $g\ x_i$ (terms that are not of this form are left unmodified except for possible recursive applications of this procedure to their subterms):

   - If g does not contain any of the f_j and either no recursion or mutual recursion can be performed through x_i, replace x_i by the "unmodified x_i" argument to the behavioural function.
   - If g does not contain any of the functions f_j and nested recursion can be performed through x_i, substitute x_i by

     $\mathsf{map}\ \mathsf{fst}\ y_i$,

     where map is x_i's type's map function and y_i is the argument to the behavioural function that pairs x_i and its recursive call's results.
   - If g is one of the f_j, replace x_i by the "modified x_i" argument to the behavioural function.
   - If g is of the form

     $\mathsf{map}\ (h_1 \circ \ldots \circ h_k \circ \mathsf{f}_j)$,

     where map is the map function of x_i's type, this is a nested recursive call. Substitute $g\ x_i$ by

     $\mathsf{map}\ (h_1 \circ \ldots \circ h_k \circ \mathsf{snd})\ y_i$,

     where y_i is again the combination of x_i and its recursive call's results.

4.4 Arbitrary constructor pattern position

In its original form, a recursor instantiation only permits one single argument that is of the type the recursor consumes. For a recursive type τ, this already gives us functions

$$\mathsf{f} :: \tau \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$$

free of charge by instantiating the recursor's return type as $\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$. It is generally desirable to be able to pass other arguments to a function before supplying the recursive argument. This is made possible by converting all input to functions of the former

shape — that is, by permuting the arguments and their types, a specification like

$$\mathsf{f} :: \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{k-1} \Rightarrow \tau \Rightarrow \alpha_{k+1} \Rightarrow \ldots \Rightarrow \alpha_n$$

is internally converted to

$$\mathsf{f'} :: \tau \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{k-1} \Rightarrow \alpha_{k+1} \Rightarrow \ldots \Rightarrow \alpha_n$$

and all further processing is performed with regard to this specification. Once a definition for f' is obtained, one for f is easily generated by adding a thin wrapper to f' that reverses the initial permutation of arguments:

$$\mathsf{f} = (\lambda a_1 \ \ldots \ a_{k-1} \ x. \ \mathsf{f'} \ x \ a_1 \ \ldots \ a_{k-1})$$

5 Codatatypes

Codatatypes are in many ways similar to datatypes, with one major difference: They permit infinite values, i.e. instances with an infinite number of constructors. The definition of a codatatype is not any more complicated than for datatypes and most of the time looks identical with **datatype** replaced by **codatatype**. The most notable difference is that corecursive codatatypes need not have a base case constructor that permits the type's values to be finite. Thus, for example, a type that holds infinite streams of data can be defined via

codatatype α *stream* (**map**: smap) = SCons (shd: α) (stl: α *stream*).

Note that every object of this type has an infinite number of constructors, which implies that any function that *returns* a stream is forced to be non-terminating.

Although a non-corecursive codatatype isn't any more expressive than a non-recursive datatype, it can be beneficial to use one. An example is the *complex* codatatype from Isabelle's standard library that makes extensive use of **primcorec**'s destructor view syntax (section 6.2) to specify a complex-valued function's result separated into its real and imaginary parts.

Other popular examples are the codatatype analogues to *list* and *nat*, namely

codatatype α *llist* (**map**: lmap) = lnull: LNil | LCons (lhd: α) (ltl: α *llist*),

the type of *lazy lists* with an either finite or infinite number of elements, and

codatatype *enat* = EZero | nonzero: ESuc (epred: *enat*),

a codatatype that is capable of representing $\mathbb{N} \cup \{\infty\}$, the *extended natural numbers*.[1] As shown in section 6, a lazy list's length can be represented by *enat*.

Analogically to datatypes, it is possible for codatatypes to support mutual and nested corecursion. Examples include *infinitely branching trees* of potentially infinite depth that can be defined by simply substituting **codatatype** for **datatype** and *llist* for *list* in the examples from section 2:

codatatype α *ltree*$_M$ = ltempty$_M$: LTEmpty$_M$
| LTNode$_M$ (ltval$_M$: α) (lchildren$_M$: α *lforest*)
and α *lforest* = lfempty: LFNil
| LFCons (lfhd: α *ltree*$_M$) (lftl: α *lforest*)

is a possible mutually corecursive realization, and

codatatype α *ltree* = LTNode (ltval: α) (lchildren: α *ltree llist*)

is a nested corecursive variant.

[1] Technically, the latter corresponds more closely to

codatatype *enat*′ = finite: Nat *nat* | Infty.

Because there is only one instance of *enat* with an infinite number of constructors (ESuc ESuc ...) that can be understood as ∞ and each finite *enat* bijectively maps to a *nat*, it becomes clear that these types can represent the same set of values.

6 Primitively corecursive functions

As noted in section 1, termination of a (co)recursive function generally implies consistency. For codatatypes, however, it is highly desirable to permit non-terminating functions (since existence of infinite values is the main advantage to datatypes): For instance, a transformation of some infinite stream is inherently non-terminating as it never stops consuming and producing pieces of data. Thus, we can't use termination (and, consequently, primitive recursion) as the constraint that ensures consistency of a function definition. The natural alternative is *productivity*, which means that in each recursive call, at least one constructor must be emitted by the corecursive function. Note that, just as for primitively recursive functions, this property can be verified by purely syntactic means.

To support varying user preferences and allow for more flexibility, Isabelle supports various ways of describing primitively corecursive functions. These so-called *views* are:

- The *destructor* view, which is the closest to the constants and theorems that **primcorec** internally uses. It has two different kinds of formulae, corresponding to a codatatype's discriminators and selectors, respectively.
- The *constructor* view, which combines related destructor view formulae into a single equation. This view's syntax is comparable to Haskell or Standard ML's.
- The *code* view, which is primarily intended for Isabelle's code generator, but has many uses in its own right. It's syntax consists of a single, unconditional equation that has an arbitrary nesting of conditionals on its right-hand side.

Regardless of the input syntax the user chooses, the defined functions' characteristic theorems are always generated in all three views.

An interesting implementation detail is that each of the input syntaxes is reduced to the same internal representation on which all transformations are performed. Afterwards, the generated theorems are lifted to the various syntax styles. This process is elaborated in sections 7.4 and 7.5. The following section describes the general appearance of primitive corecursion and gives some typical examples.

6.1 Primitive corecursion

The forms and shapes in which primitive corecursion may come is in many ways dual to primitive recursion for datatypes. The main difference is that whereas primitively recursive calls must be applied to an *unmodified* constructor argument from the function's *argument*, primitive *co*recursion must be the *outermost* function call in one of the *emitted* constructor's arguments. For example,

primcorec sllist :: α *stream* $\Rightarrow$ α *llist* **where**
  sllist s = LCons (shd s) (sllist (stl s))

is a primitively corecursive function that converts an infinite stream to an infinite list that contains the same elements. The corecursive call sllist (stl s) is the outermost function application in LCons's second argument.

A similar requirement applies to composition of (co)recursive functions in map arguments for the "nested" case: a primitively corecursive call must be the last/leftmost function in a composition chain.

In section 5, it was remarked that *enat* is capable of holding a *llist*'s length:

```
primcorec llength :: α llist ⇒ enat where
  lnull xs ⟹ llength xs = EZero
| _ ⟹ nonzero (llength xs)
| epred (llength xs) = llength (ltl xs)
```

introduces a function llength that is capable of computing this mapping. This specification is an instance of *destructor view* (section 6.2). The first two (*discriminator*) formulae specify conditions for each constructor to be emitted;[1] the third (*selector*) formula specifies the constructor's argument in the ESuc case.

An example for nested primitive corecursion is the function

```
primcorec ltree_map :: (α ⇒ β) ⇒ α ltree ⇒ β ltree where
  ltree_map f lt = (case lt of LTNode v cs ⇒ LTNode (f v) (lmap (ltree_map f) cs))
```

that applies a function to the value associated with each node in a (potentially) infinitely branching tree of (potentially) infinite depth.

Just like in the case of datatypes, **primcorec** supports nested-as-mutual corecursion, that is, using nested corecursive codatatypes as if they were mutually corecursive. The function defined above could alternatively be specified using nested-as-mutual corecursion:

```
primcorec
  ltree_map_M :: (α ⇒ β) ⇒ α ltree ⇒ β ltree and
  ltrees_map :: (α ⇒ β) ⇒ α ltree llist ⇒ β ltree llist
where
  ltree_map_M f lt = (case lt of LTNode v cs ⇒ LTNode (f v) (ltrees_map f cs))
| lnull lts ⟹ ltrees_map f lts = LNil
| _ ⟹ ltrees_map f lts = LCons (ltree_map_M f (lhd lts)) (ltrees_map f (ltl lts))
```

6.2 Destructor view

Specifications for the function f :: ... ⇒ α *t* in *destructor view* consist of formulae of the general form

P x_1 ... x_n ⟹ is_C$_i$ (f x_1 ... x_n),

where P :: ... ⇒ *bool* is some property of $x_1, \ldots, x_n$ and is_C$_i$ is a discriminator for the codatatype α *t*, or

un_C$_{ij}$ (f x_1 ... x_n) = F x_1 ... x_n,

where un_C$_{ij}$ is the jth selector for C$_i$.[2]

[1] The '_' serves as a wildcard that will match the negation of all predicates specified before; in this case, it is therefore equivalent to ¬ lnull xs.

[2] This is similar to the syntax used by Abel et al. in their work on *copatterns* [1].

The semantics of this specification is as follows: A formula of the first kind, called a *discriminator formula*, indicates premises for the function image to be constructed by C_i, and formulae of the second kind, called *selector formulae*, specify values $F\ x_1\ \ldots\ x_n$ for each of the constructor arguments given the premises for the associated discriminator formula have been met.

For the function to be well-defined, the discriminator formulae's preconditions need to be mutually exclusive, giving rise to proof obligations. **primcorec** tries to solve these automatically, but in case it fails, the user must resort to the more general **primcorecursive** command that passes the burden of proof to the user. Essentially, the following "equation" holds:

primcorec ... = **primcorecursive** ... **by** auto?

An example for destructor view is the lapp function that concatenates two (potentially infinite) lists over the same type. One of its possible representations is

primcorec lapp :: α *llist* $\Rightarrow$ α *llist* $\Rightarrow$ α *llist* **where**
  lnull xs $\wedge$ lnull ys $\Longrightarrow$ lnull (lapp xs ys)
| lhd (lapp xs ys) = (**if** lnull xs **then** lhd ys **else** lhd xs)
| ltl (lapp xs ys) = (**if** lnull xs **then** ys **else** lapp (ltl xs) ys).

Note that the discriminator formulae lead to the function's result being either constructed by one of the result type's constructors or, in exceptional cases, unspecified. Thus, when describing functions in this form, it is trivially guaranteed that a constructor is emitted.

Nevertheless, there is a potential problem: Corecursive calls' results must not be fed into another function in a selector formula right-hand side. Otherwise,

primcorec wrong :: *nat* $\Rightarrow$ *nat llist* **where**
  $\neg$ lnull (wrong n)
| lhd (wrong n) = n
| ltl (wrong n) = ltl (wrong $(n + 1)$),

listed in destructor view, would never produce a fully applied constructor. The issue is that even though the corecursive call is forced to output one, the following call to ltl strips it off again and an invocation like ltl (wrong 0) would loop forever. An exception to this requirement are corecursive calls inside **if** − **then** − **else**, **case** − **of** and **let** − **in**, which otherwise need to fulfill the same requirement, that is, corecursive calls must not serve as a function argument.

It is also notable that this style is the only one that allows partial specification of constructor arguments; missing selector formulae are simply ignored and no associated theorems are generated.

There is another speciality of this style: Due to a selector being applicable to more than one different constructors, an ambiguity in selector formulae can occur. To resolve this issue, **primcorec** accepts selector formulae of the form

sel (f ...) = ... **of** C,

where C is the constructor belonging to which the selector sel should be interpreted.

In the case of two constructors C_1 and C_2, another special syntax is supported: The discriminator formula for C_2 may use $\neg$ is_C_1 instead of is_C_2. This exception is due to some two-constructor codatatypes missing an explicit discriminator for the second constructor.

In order to avoid the need of tedious manual specification of an "else" predicate for the discriminator formulae, an underscore '_' is accepted as a "catch-all" wildcard. It is understood as the implicit negation of all conditions for the relevant function in previous equations. Thus, in

$$\begin{aligned} &P\,x \implies \mathsf{is\_C}_i\ (\mathsf{f}\,x)\\ &Q\,x \implies \mathsf{is\_C}_j\ (\mathsf{f}\,x)\\ &\_ \implies \mathsf{is\_C}\ (\mathsf{f}\,x), \end{aligned}$$

the last formula is interpreted as $\neg\, P\,x \,\wedge\, \neg\, Q\,x \implies \mathsf{is\_C}\ (\mathsf{f}\,x)$.

A related functionality is the *sequential* option to **primcorec**: it causes the discriminator formula conditions for a function to be interpreted not literally, but in an "**if**$\,-\,$**else if**" fashion. This means that each subsequent formula's premise implicitly includes the negation of all previously specified conditions. As a result, the exclusiveness proof obligations become trivially solvable, relieving the user from the burden of discharging them. In return, the generated theorems become more complicated and as a consequence, more unhandy to use.

The second option that **primcorec**(**ursive**) accepts is *exhaustive*. It signals that the discriminator formula premises are not only mutually exclusive (at most one of them is fulfilled), but in fact a partition of truth (*exactly* one of them is fulfilled). Specifying this option adds another proof obligation that is suppressed by **primcorec** if possible, and passed on to the user by **primcorecursive**. In return, somewhat stronger discriminator theorems (cf. section 6.5) are generated, with $\longleftrightarrow$ in place of $\implies$. Another consequence is that autogenerated code view (section 7.5.2) specifications are already complete and do not need an "else abort" branch.

In some cases when exhaustiveness is syntactically guaranteed (for example, when the catch-all premise '_' is used), **primcorec** automatically acts as if this option had been specified.

6.3 Constructor view

Constructor view is arguably the most intuitive for users with a background in functional programming due to its similarity to widespread syntax. In *constructor view*, there is only one kind of formula that combines discriminator and selector formulae into one single equation per constructor. Its general form is

$$\mathsf{P}\,x_1\,\ldots\,x_n \implies \mathsf{f}\,x_1\,\ldots\,x_n = \mathsf{C}_i\,(F_1\,x_1\,\ldots\,x_n)\,\ldots\,(F_k\,x_1\,\ldots\,x_n),$$

where P is a property just like before, C_i is one of f's return type's constructors and F_1 to F_k are some functions in $x_1, \ldots, x_n$ that describe the structure of C_i's arguments.

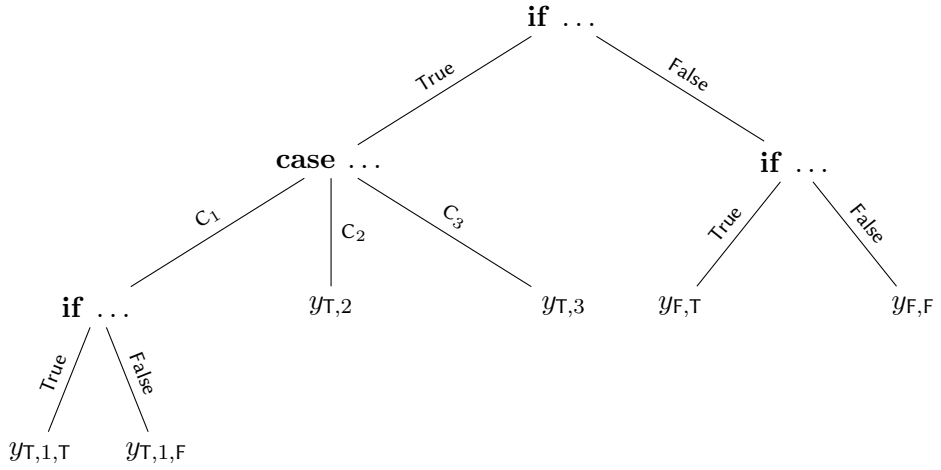


Figure 6.1: Example conditional tree

An example should be helpful: A definition of the lapp function in constructor view can be performed via

primcorec lapp :: α *llist* $\Rightarrow$ α *llist* $\Rightarrow$ α *llist* **where**
  lnull xs $\wedge$ lnull ys $\Longrightarrow$ lapp xs ys $=$ LNil
| _ $\Longrightarrow$ lapp xs ys $=$ LCons (**if** lnull xs **then** lhd ys **else** lhd xs)
(**if** lnull xs **then** ys **else** lapp (ltl xs) ys).

The constraints on constructor arguments are the same as the requirements to a selector formula right-hand side. Additionally, just like each discriminator must occur in exactly zero or one discriminator formulae, there must be at most one constructor-style equation for each of the codatatype's constructors.

6.4 Code view

The code view's main purpose is to interface with Isabelle's code generator, which supports no pattern matching (at function level) and requires that there is only one unconditional equation for the function's value. Since this interferes with the constructor view's policy that each right-hand side must consist of a fully applied constructor, this requirement was loosened to additionally allow the conditionals **if** $-$ **then** $-$ **else** and **case** $-$ **of** as well as **let** $-$ **in**-bindings *outside of constructors*[3] and merely require that each conditional *leaf* (in figure 6.1, these are the nodes labeled $y_{\ldots}$) starts with a constructor. Again, since the function is forced to emit at least one constructor per corecursive descent, productivity is guaranteed.

Specifications in this style consist of one single equation of the form

f x_1 ... x_n $=$ ...,

where the right-hand side may involve an arbitrary nesting of case distinctions and **let** $-$ **in**-bindings, as long as each of their body terms has a constructor at the beginning. In

[3] This is the main difference to the other views, where these constructs are permitted in constructor arguments or selector formulae, but they must be guarded by a constructor.

other words: all of the conditional tree's leaves have to fulfill the demands required from constructor-style right-hand sides. It is notable that this style allows the same constructor at the beginning different conditional leaves, contrary to the constructor view's requirements.

Let the lapp function serve, again, as an example:

primcorec lapp :: α *llist* $\Rightarrow$ α *llist* $\Rightarrow$ α *llist* $\Rightarrow$ α *llist* **where**
lapp xs ys $=$ (**case** xs **of** LNil $\Rightarrow$ ys | LCons x xs' $\Rightarrow$ LCons x (lapp xs' ys))

This variant can be considered the simplest among the definitions presented above, which makes it the most suitable choice of input syntax to describe this function.

6.5 Generated theorems

Regardless of the user's choice of input syntax, the **primcorec** command generates characteristic theorems in all of the syntax styles (section 7.5). For example, the constructor-style definition

primcorec (*exhaustive*) iterate_while :: ($\alpha \Rightarrow \alpha$ *option*) $\Rightarrow$ $\alpha \Rightarrow$ α *llist* **where**
is_none (f x) $\Longrightarrow$ iterate_while f x $=$ LNil
| _ $\Longrightarrow$ iterate_while f x $=$ LCons x (iterate_while f (the (f x)))

gives rise to the following theorems:

iterate_while.disc:
is_none (f x) $\Longrightarrow$ lnull (iterate_while f x)
$\neg$ is_none (f x) $\Longrightarrow$ $\neg$ lnull (iterate_while f x)

iterate_while.disc_iff:
lnull (iterate_while f x) $\longleftrightarrow$ is_none (f x)
$\neg$ lnull (iterate_while f x) $\longleftrightarrow$ $\neg$ is_none (f x)

iterate_while.sel:
$\neg$ is_none (f x) $\Longrightarrow$ lhd (iterate_while f x) $= x$
$\neg$ is_none (f x) $\Longrightarrow$ ltl (iterate_while f x) $=$ iterate_while f (the (f x)).

iterate_while.ctr:
is_none (f x) $\Longrightarrow$ iterate_while f x $=$ LNil
$\neg$ is_none (f x) $\Longrightarrow$
iterate_while f x $=$ LCons x (iterate_while f (the (f x)))

iterate_while.code:
iterate_while f x $=$
(**if** is_none (f x) **then** LNil **else** LCons x (iterate_while f (the (f x))))

The *iterate_while.disc_iff* theorems being produced is caused by the '_' implicitly enabling the *exhaustive* option for the relevant function.

Note that of the three syntaxes, only destructor-style theorems (that is, *....disc*, *....sel* and *....disc_iff*) are registered for simplification. The other forms potentially loop and can therefore lead to non-termination and memory exhaustion in the simplifier.

7 primcorec's implementation

Considering that codatatypes are in many ways analogue to datatypes, it is not surprising that the high-level view of **primcorec**'s operation is not too dissimilar from **primrec**'s.

Just like before, we parse the user input to synthesize a low-level definition based on the *corecursor* (which is obviously an analogue to a datatype's recursor) and use the codatatype's and corecursor's associated theorems to prove that our definition fulfills the user-specified properties.

7.1 Corecursors

Analogically to a datatype's recursor, a set of mutually corecursive codatatypes provides *corecursors* that serve essentially the same purpose. Again, it is probably best to have a look at an example: The α *llist* codatatype introduced in section 5 has the corecursor

corec_llist ::
$(\alpha \Rightarrow \mathit{bool}) \Rightarrow$
$(\alpha \Rightarrow \beta) \Rightarrow$
$(\alpha \Rightarrow \mathit{bool}) \Rightarrow (\alpha \Rightarrow \beta\ \mathit{llist}) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow$
$\alpha \Rightarrow \beta\ \mathit{llist}$.

The corecursor's behavioural functions may seem a bit obscure at first, but they actually have a well-considered, quite intuitive meaning described by the *llist.corec* theorems:

llist.corec:
$p\ a \implies \mathsf{corec\_llist}\ p\ g_{2,1}\ q_{2,2}\ g_{2,2}\ h_{2,2}\ a = \mathsf{LNil}$
$\neg\ p\ a \implies \mathsf{corec\_llist}\ p\ g_{2,1}\ q_{2,2}\ g_{2,2}\ h_{2,2}\ a =$
$\mathsf{LCons}\ (g_{2,1}\ a)\ (\mathbf{if}\ q_{2,2}\ a\ \mathbf{then}\ g_{2,2}\ a\ \mathbf{else}\ \mathsf{corec\_llist}\ p\ g_{2,1}\ q_{2,2}\ g_{2,2}\ h_{2,2}\ (h_{2,2}\ a))$

The following is perhaps best read with the destructor view in mind, since this view corresponds the most closely to the corecursor:

- The predicate p decides which of the codatatype's constructors is emitted. In this case, the result is LNil if and only if p is fulfilled.
- Next follow the corecursor arguments corresponding to a selector: $g_{2,1}$ describes the second constructor's first argument, the list's head. Due to its type being non-composite, no corecursion is possible in this constructor argument and the corresponding corecursor argument is just its value.
- The next three arguments, $q_{2,2}$ to $h_{2,2}$, specify the properties of the list's tail. The predicate $q_{2,2}$ determines whether the corecursion ends with a non-corecursive term or continues descending any further; $g_{2,2}$ is the constructor argument's value in the former case. If $q_{2,2}$ is not satisfied, $h_{2,2}$ returns a tuple of arguments that is corecursively passed to the corecursor's instantiation.

As an example for nested corecursion, consider the type of (possibly) infinitely branching trees of (potentially) infinite depth *ltree*, defined in section 5, which is equipped with

corec_ltree :: $(\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow (\beta\ \mathit{ltree} + \alpha)\ \mathit{llist}) \Rightarrow \alpha \Rightarrow \beta\ \mathit{ltree}$

as its corecursor. Again, the first argument returns just the constant value of Node's first argument. The second argument's purpose is explained by the theorem

ltree.corec:
corec_ltree $g_1\ g_2\ a\ =$
LTNode $(g_1\ a)$ (lmap $(\lambda x.$ **case** x **of** Inl $\ell \Rightarrow \ell$ | Inr $r \Rightarrow$ corec_ltree $g_1\ g_2\ r)\ (g_2\ a)$):

If the behavioural function returns an object of the sum type's left branch, it is interpreted as a literal non-corecursive result. Otherwise, the return value is fed back to the same corecursor instantiation as its input. The behavioural function can thus, given some input, decide for nested corecursion to terminate (by returning an Inl-constructed object) or to continue descending by returning the corecursive invocation's argument wrapped inside an Inr.

In general, if a codatatype σ has n constructors, its corecursor corec_σ :: $\ldots \Rightarrow \alpha \Rightarrow \sigma$ expects $n - 1$ predicates $p :: \alpha \Rightarrow \mathit{bool}$ that determine which constructor is emitted. These predicates are tested in sequence; the implicit precondition for the last constructor is the negation of all of the other predicates.

Additionally, to each constructor argument of type τ (or equivalently: each right-hand side of a selector formula), one of the following cases applies:

- If τ is not corecursive with σ, then no corecursion is possible and the behavioural function $g :: \alpha \Rightarrow \tau$ simply returns the constructor argument's unconditional value.
- If τ is mutually corecursive with σ, the corecursor expects three behavioural functions: A predicate $q :: \alpha \Rightarrow \mathit{bool}$("stop?") that returns True if and only if the constructor argument's value is a non-corecursive term; a function $g :: \alpha \Rightarrow \sigma$ ("end") that specifies the value for cases when the predicate is fulfilled; and a function $h :: \alpha \Rightarrow \beta$ ("continue"), where β is the mutually corecursive function's input type, that maps to the input for the corecursive invocation.
- If σ occurs in τ nested under one or more type constructors ν, the corecursor takes one behavioural function that returns a value of type $(\sigma + \alpha)\ \nu$. Just as described above, the sum type represents either a non-corecursive constant result or a corecursive call's argument.

7.2 General procedure

The high-level view of **primcorec**'s operation consists of the following steps:

1. Use the functions' type information to get the involved codatatypes' constructors, discriminators and selectors from the codatatype database.
2. From each supplied formula's structure, determine what kind of formula it is. For constructor- or code-style formulae, call the respective reduction functions to extract

the same internal representation from all three input syntaxes. The syntax reductions are described in section 7.4.

After this step, for each formula (which has perhaps been auto-generated during the reduction), we have:

- The function's name, type, and its arguments' names and types as they occured in this particular term
- The constructor that this formula is relevant to
- The original user input, and possibly — if this formula was obtained by reducing from a different view — the reduction's preimage.

The specific fields for discriminator and selector formulae are a list of premises and the right-hand side, respectively.

During this step, the interpretation of '_' wildcards and the *sequential* option are performed and any implicit discriminator formulae are generated.

3. Scan the selector formula right-hand sides for corecursive calls and record their structure.

4. Using this new information, get the rest of the codatatypes' information (corecursors, theorems, types of corecursion, ...) from the codatatype database. Just like for datatypes, a pending nested-to-mutual reduction is performed in this step (transparently to the following).

5. Obtain definitions for the specified functions. Note that this involves conversion of corecursive calls to a corecursor-based counterpart (see section 7.3).

6. Assemble the exclusiveness (and perhaps exhaustiveness) properties and, for **primcorec**'s case, prove them automatically.

7. From the definitions, exclusiveness and exhaustiveness theorems, and codatatype- and corecursor-related theorems, prove the functions' characteristic theorems in all of the syntax styles. Since the reductions are performed (and thus, yield specifications) only in one direction (section 7.4), we may need to assemble theorems in constructor view or code view at this point.

7.3 Eliminating corecursive calls

Contrary to what was the case for primitive recursion, the possible structure of primitive corecursion is fairly limited and, consequently, a bit easier to convert to a corecursor-based replacement. Since each constructor argument allows either mutual or nested corecursion, but not both, and a corecursive call must be the outermost function call in a selector formula right-hand side (except for **if** − **then** − **else**, **case** − **of** and **let** − **in**), it suffices to traverse the conditional tree induced by these structures (figure 6.1) to determine for each leaf whether it contains a corecursive call or not and substitute it by a suitable argument to the behavioural function:

- If the constructor argument does not allow any corecursion, its selector formula right-hand side is converted to a behavioural function by simply λ-abstracting with respect to the function arguments.
- In the case of mutual corecursion, we need to generate three corecursor arguments: the predicate q ("stop?") is created by substituting either True for a non-corecursive leaf or False for a corecursive leaf; g ("end") is formed by replacing corecursive leaves by undefined; and h ("continue") is similarly obtained by substituting undefineds for non-corecursive leaves and a tuple of the corecursive call's function arguments for corecursive leaves.[1]
- For nested corecursion, the corecursor combines the "stop?-end-continue" construction into a single argument that returns a nested sum type whose branches correspond to a non-corecursive result or a corecursive call. Therefore, similarly to mutual corecursion, the behavioural function can be obtained by replacing a non-corecursive leaf y by

  $$\mathsf{map}\ \mathsf{Inl}\ y,$$

  where map is the nesting type's map function,[2] and a nested corecursive call

  $$\mathsf{map}\ (\mathsf{f} \circ h_1 \circ \ldots \circ h_k)\ a$$

  by

  $$\mathsf{map}\ (\mathsf{Inr} \circ h_1 \circ \ldots \circ h_k)\ a.$$

7.4 Input syntax reductions

Despite the variety of input styles the **primcorec** command supports, the differences are mostly superficial. The internal constructions are common to the syntaxes, and in any case, the resulting theorems are generated in each of them.

This makes it possible to *reduce* the views one to another, using only structural (syntactic) transformations. Namely, the chain of reductions is

destructor $\longleftarrow$ *constructor* $\longleftarrow$ *code*.

Each reduction works by disassembling the input as far as necessary and creating equivalent specifications in the more primitive input style. These specifications are then passed down to the parsing functions for the given input style and processed as if the user had entered them. At the end of this procedure, the input syntaxes share common data structures holding the function specification's relevant details. In order to be able to recover the exact original term for the higher-style theorems, the user input is also stored in these structures. When it comes to generating the function's characteristic theorems, the path of reductions is traversed backwards: The code view theorems are derived from the constructor view theorems, which are in turn derived from the destructor-style theorems. If the user has

[1] Substituting undefined is necessary to remove occurrences of the corecursive function on the future definition's right-hand side for "end"; and to replace non-corecursive leaves by a term of the correct type for "continue".

[2] Like for **primrec**, we assumed for simplicity that the map function takes only one argument.

entered a destructor- or constructor-style specification, **primcorec** also synthesizes equivalent theorems in the missing input styles.

7.4.1 Constructor view to destructor view

Equations in constructor view correspond to an equivalent destructor view specification in a rather obvious way: A formula

$$\mathsf{P}\ x_1\ \ldots\ x_n\ \Longrightarrow\ \mathsf{f}\ x_1\ \ldots\ x_n\ =\ \mathsf{C}\ y_1\ \ldots\ y_m$$

is transformed into the equivalent set of destructor formulae

$$\begin{aligned}&\mathsf{P}\ x_1\ \ldots\ x_n\ \Longrightarrow\ \mathsf{is\_C}\ (\mathsf{f}\ x_1\ \ldots\ x_n)\\&\mathsf{un\_C}_1\ (\mathsf{f}\ x_1\ \ldots\ x_n)\ =\ y_1\\&\ldots\\&\mathsf{un\_C}_m\ (\mathsf{f}\ x_1\ \ldots\ x_n)\ =\ y_m,\end{aligned}$$

where is_C is the m-ary constructor C's discriminator and un_C$_i$ are its selectors.

7.4.2 Code view to constructor view

Recall that a specification in code view consists of a single equation, potentially having many case distinctions via **if** and **case**, and **let**-bindings. Since the constructor view requires that there is at most one equation for each constructor, we first need to group the leaves of these case distinctions by the constructor that is applied to the result. During this stage, non-corecursive branches (that do not start with a constructor) are expanded using **case**: For example, an expression $expr$ of type α *llist* becomes (**case** $expr$ **of** LNil $\Rightarrow$ LNil $\mid$ LCons $x\ xs \Rightarrow$ LCons $x\ xs$) to later fulfill the constructor view's syntax requirements. Along each of the paths, the set of conditions that need to be fulfilled to reach the current node is carried along. After the formulae have been collected, they are ready to be combined into one single equation per constructor. For example, the lapp specification shown in section 6.4 is internally processed as if

$$\begin{aligned}&\textbf{primcorec}\ \mathsf{lapp}\ ::\ \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist} \Rightarrow \alpha\ \mathit{llist}\ \textbf{where}\\&\quad \mathsf{lnull}\ xs\ \wedge\ \mathsf{lnull}\ ys\ \Longrightarrow\ \mathsf{lapp}\ xs\ ys\ =\ \mathsf{LNil}\\&\mid\ \_\ \Longrightarrow\ \mathsf{lapp}\ xs\ ys\ =\ \mathsf{LCons}\ (\textbf{case}\ xs\ \textbf{of}\ \mathsf{LNil} \Rightarrow \mathsf{lhd}\ ys\ \mid\ \mathsf{LCons}\ x\ xs' \Rightarrow x)\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{case}\ xs\ \textbf{of}\ \mathsf{LNil} \Rightarrow \mathsf{ltl}\ ys\ \mid\ \mathsf{LCons}\ x\ xs' \Rightarrow \mathsf{lapp}\ xs'\ ys)\end{aligned}$$

had been entered by the user. This specification is then passed down to the constructor-style machinery.

7.5 Producing theorems in other views

The syntax reductions described in section 7.4 induce the need to traverse the reduction's path backwards: that is, to deduce theorems in the less primitive styles from the more primitive specification that they were reduced to. If a representation in the higher style is available (because the lower-style specification was derived from it), this one is simply proved as is. Otherwise, **primcorec** automatically assembles one, using the following procedures:

7.5.1 Destructor view to constructor view

Essentially, the lifting of destructor view theorems to constructor view just uses the reduction described in section 7.4.1 in reverse. Each constructor and its associated discriminator formula premises and selector formulae right-hand sides are collected and combined to form a constructor-style specification. Note that this is not possible if some selector formulae are missing, which leads to no constructor-style theorem (and as a result, no code view theorem) being generated.

7.5.2 Constructor view to code view

Analogically, lifting from constructor view to code view reverses the reduction from section 7.4.2. It takes the constructor-style right-hand sides along with their preconditions and builds an **if** $-$ **then** $-$ **else if** tree from them.

If the "exhaustive" option is not specified, the generated theorem will have an "**else**" branch containing Code.abort, signaling the code generator that the function's result is unspecified.

In both cases, the newly assembled terms are then proved as theorems, exploiting the already generated lower-style theorems to make an easier (as opposed to directly using the corecursor-based definition and corecursor theorems) proof tactic possible.

7.6 Arbitrary number of arguments

As described in section 7.1, the corecursor internally only handles unary functions (just like a datatype's recursor). In order to be able to define n-ary primitively corecursive functions, an n-tuple of arguments is passed to the corecursor corec and, consequently, its arguments, and surrounded by a currying wrapper. Thus, the resulting definition has the form

$$\lambda a'_1\ \ldots\ a'_n.\ \mathsf{corec}\ (\lambda(a_1,\ \ldots,\ a_n).\ \ldots)\ \ldots\ (\lambda(a_1,\ \ldots,\ a_n).\ \ldots)\ (a'_1,\ \ldots,\ a'_n).$$

8 Conclusion

We presented new definitional Isabelle/HOL commands that enable specification of primitively (co)recursive functions over Isabelle's new BNF-based (co)datatypes, including mutual and nested (co)recursion, in intuitive syntaxes.

primrec is intended as, but not limited to, a replacement for Isabelle's old datatype package's command of the same name. In particular, it maintains full backward compatibility while providing additional, new functionality.

primcorec and its more general variant **primcorecursive** are an analogue to **primrec** for codatatypes. They permit a variety of intuitive input syntaxes, suitable to different needs and preferences.

The functionality presented in this thesis will be part of the forthcoming Isabelle2014 release and is currently available in the repository version of Isabelle. The *list* and *option* datatypes in the Isabelle standard library are already implemented using the new datatype package and thus make copious use of **primrec**. Lochbihler's Coinductive library [7], notably used in his work on the Java programming language, has been ported to use **primcorec** [3]. Traytel's implementation of formal languages using codatatypes [12] is another extensive user of **primcorec**.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *POPL '13: Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 27–38. ACM, 2013.

[2] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL — Lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36, 1999.

[3] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP 2014)*, Lecture Notes in Computer Science. Springer, 2014.

[4] Jasmin Christian Blanchette, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Defining (co)datatypes in Isabelle/HOL. `http://isabelle.in.tum.de/dist/Isabelle/doc/datatypes.pdf`, 2014.

[5] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1990.

[6] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[7] Andreas Lochbihler. Coinductive. *Archive of Formal Proofs*, February 2010. `http://afp.sf.net/entries/Coinductive.shtml`, Formal proof development.

[8] Andreas Lochbihler and Johannes Hölzl. Recursive functions on lazy lists via domains and topologies. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP 2014)*, Lecture Notes in Computer Science. Springer, 2014.

[9] Lorenz Panny, Jasmin Christian Blanchette, and Dmitriy Traytel. Primitively (co)recursive definitions for Isabelle/HOL. Isabelle Workshop 2014.

[10] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[11] Owre Sam and Shankar Natarajan. Abstract datatypes in PVS. Technical report, 1997.

[12] Dmitriy Traytel. A codatatype of formal languages. *Archive of Formal Proofs*, November 2013. `http://afp.sf.net/entries/Coinductive_Languages.shtml`, Formal proof development.

[13] Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic — Category theory applied to theorem proving. In *LICS 2012*, pages 596–605. IEEE, 2012.