Primitively (Co)recursive Definitions for Isabelle/HOL

Lorenz Panny, Jasmin Christian Blanchette, and Dmitriy Traytel

Fakultät für Informatik, Technische Universität München, Germany

Abstract. Isabelle/HOL has recently been enriched with a definitional package for datatypes and codatatypes. The package introduces the specified types and derives auxiliary constants and characteristic theorems, notably (co)recursors and (co)induction principles. We now introduce support for high-level specifications of (co)recursive functions, in the form of three commands: **primrec**, **primcorec**, and **primcorecursive**. The commands internally reduce the specifications to arguments to the (co)recursors and generate a number of theorems about the function definition, automating a tedious process.

1 Introduction

Recursive functions are the primary mechanism for expressing recurring computations in functional programming languages. The newly introduced (co)datatype package for Isabelle/HOL [**?**, **?**] provides methods to define datatypes and codatatypes, but until recently there was no convenient way of defining functions over these types. This paper introduces commands to allow users to define primitively recursive functions over datatypes and primitively corecursive functions over codatatypes.

Primitive recursion means that in each recursive descent, exactly one constructor is peeled off the function argument through which recursion is performed. Since datatypes are finite, this ensures that the recursion eventually terminates. A simple 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 corecursion is dual: Instead of peeling off one constructor at each iteration, it *produces* one layer of constructors. The generated value may be finite or infinite, depending on whether the corecursive call chain terminates.

The new command **primrec** is modeled after the command of the same name developed by Berghofer and Wenzel [**?**] for the old datatype package, but it additionally supports recursion through non-datatypes using a map function. Both the old and the new **primrec** share the same command name. Internally, the new implementation described here is used if the recursion is performed on a datatype introduced by **datatype_new**; otherwise, the old implementation is used. In contrast, the command **primcorec** and its more powerful variant **primcorecursive** are completely new additions.

The paper is structured in two parts: Sections 2 to 4 cover datatypes, **primrec**, and its implementation; Sections 5 to 7 cover codatatypes, **primcorec**, and its implementation. The text of this paper is based largely on a draft of the first author's B.Sc. thesis [**?**]. It also incorporates passages from the (co)datatype package's user manual [**?**].

2 Background: Datatypes

A fundamental concept in most, if not all, typed functional programming languages is that of freely generated datatypes. They are the main way to create new types and combine existing types into more complex (and useful) types. Probably the most popular example is the datatype of (finite) lists over an element type α, which can be defined by

> **datatype_new** α *list* (**map**: map) = null: Nil | Cons (hd: α) (tl: α *list*)

The **datatype_new** command is provided by the new (co)datatype package. Here, it introduces the type α *list* generated freely by two constructors:

- Nil, which represents an empty list and can be tested using null :: α *list* $\Rightarrow$ *bool*;
- Cons, which takes two arguments of types α and α *list* and *cons*tructs a new list from a single item (the new list's *h*e*ad*) and the rest of the list (its *t*ai*l*).

This simple example shows a basic concept of 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 to the nested value.

The **datatype_new** command supports the definition of mutually recursive datatypes. These are characterized by the simultaneous specification of two or more datatypes that recursively contain each other in some of their constructor arguments. A typical example is the type of finitely branching trees of finite depth and associated (finite) forests that do not use a universal list type:

> **datatype_new** α *tree0* = Node0 (tval0: α) (children0: α *forest*)
> **and** α *forest* = FNil | FCons (fhd: α *tree0*) (ftl: α *forest*)

The command requires that each set of mutually recursive datatypes have identical type arguments (here, α). Another, more complex example is a representation for a simple arithmetic expression language:

> **datatype_new** *expr* = XSum *expr_sum* | XProd *expr_prod* | XConst *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 makes it possible to reason more modularly about the datatype and to reuse existing libraries. For example, the more modular way to specify a type such as α *tree0* reuses the generic *list* type constructor instead of introducing a dedicated *forest* type:

> **datatype_new** α *tree* = Node (tval: α) (children: α *tree list*)

Similarly, binary trees can be defined via

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

where α *btree* is nested inside the *option* type. In general, nested recursion arises when a datatype recursively occurs under an existing type constructor.

The **datatype_new** command allows combinations of mutual and nested recursion through datatypes, through the right-hand side of the function type $\alpha \Rightarrow \beta$, and through other well-behaved type constructors such as *fset* (finite set), as long as the defined type is nonempty. These well-behaved type constructors are called *bounded natural functors* (*BNFs*) [**?**]. BNFs are equipped with a map function (or functorial action), one or more set functions (or natural transformations), and a cardinality bound that makes the (co)datatype constructions possible.

3 Specification of Primitively Recursive Functions

Recursive functions over datatypes can be specified using the **primrec** command, which supports primitive recursion, or using the more general **fun**, **function**, or **partial_ function** commands [**?**,**?**]. Here, the focus is on **primrec**. Despite its relative simplicity, it provides a number of advantages over the alternatives:

- *It never emits any proof obligations.* In particular, it accepts certain mutually recursive functions that are rejected by **fun** and for which a manual termination proof is necessary with **function**.
- *It is fast and lightweight.* The internal reduction of user specifications to arguments of the recursor generated by **datatype_new** is a straightforward syntactic transformation that requires no induction nor any other sophisticated reasoning principle.
- *It has few dependencies.* As a result, it is loaded at an early stage in the Isabelle/HOL bootstrapping process, before any of its rivals.

3.1 Simple Recursion

A primitively recursive function over a datatype $\bar{\tau}\ \kappa$ peels off one constructor in each recursive descent. For simply recursive types, the function definitions are of the form

$$\begin{array}{l}
\textbf{primrec}\ \mathsf{f} :: \bar{\sigma} \Rightarrow \bar{\tau}\ \kappa \Rightarrow \bar{\upsilon} \Rightarrow \beta\ \textbf{where} \\
\quad \mathsf{f}\ \bar{w}_1\ (\mathsf{C}_1\ \bar{x}_1)\ \bar{y}_1\ = t_1 \\
\qquad \vdots \\
\mid \mathsf{f}\ \bar{w}_m\ (\mathsf{C}_m\ \bar{x}_m)\ \bar{y}_m\ = t_m
\end{array}$$

where the C_i's are κ's constructors and the $\bar{x}_i$'s are their arguments (in curried form). The right-hand sides t_i may involve recursive calls of the form $\mathsf{f}\ \bar{w}\ x_{ij}\ \bar{y}$. The constructor patterns $\mathsf{C}_i\ \bar{x}_i$ may appear at an arbitrary argument position, but its position must be consistent among the equations for f. No further pattern matching in the constructor pattern is supported, nor is pattern matching allowed for the remaining arguments to f.

The functions app (append) and rev (reverse) are examples of primitively recursive functions:

```
primrec app :: α list ⇒ α list ⇒ α list where
  app Nil = id
| app (Cons x xs) = Cons x ∘ app xs

primrec rev :: α list ⇒ α list where
  rev Nil = Nil
| rev (Cons x xs) = app (rev xs) (Cons x Nil)
```

The **primrec** command expects to find an equation for each of the datatype's constructors; otherwise, it prints a warning. The *nonexhaustive* option can be used to suppress the warning:

```
primrec (nonexhaustive) hd :: α list ⇒ α where
  hd (Const x _) = x
```

3.2 Mutual Recursion

Recursion over m mutually recursive datatypes $\bar{\tau}\,\kappa_1, \ldots, \bar{\tau}\,\kappa_m$ generally requires m mutually recursive functions. This is achieved using the syntax

$$
\begin{array}{l}
\textbf{primrec} \\
\quad \mathsf{f}_1 :: \cdots \Rightarrow \bar{\tau}\,\kappa_1 \Rightarrow \cdots \Rightarrow \sigma_1 \ \textbf{and} \\
\qquad \vdots \\
\quad \mathsf{f}_m :: \cdots \Rightarrow \bar{\tau}\,\kappa_m \Rightarrow \cdots \Rightarrow \sigma_m \\
\textbf{where} \\
\quad \mathsf{f}_1 \ \ldots \ (\mathsf{C}_{11}\ \bar{x}_{11}) \ \ldots = \ldots \\
\qquad \vdots \\
\mid \mathsf{f}_1 \ \ldots \ (\mathsf{C}_{1k_1}\ \bar{x}_{1k_1}) \ \ldots = \ldots \\
\qquad \vdots \\
\mid \mathsf{f}_m \ \ldots \ (\mathsf{C}_{m1}\ \bar{x}_{m1}) \ \ldots = \ldots \\
\qquad \vdots \\
\mid \mathsf{f}_m \ \ldots \ (\mathsf{C}_{mk_m}\ \bar{x}_{mk_m}) \ \ldots = \ldots
\end{array}
$$

The right-hand sides may contain recursive calls to any of the functions f_i. For example, arithmetic expressions represented by the datatypes introduced in Section 2 can be evaluated recursively as follows:

```
primrec
  eval :: expr ⇒ nat and
  eval_sum :: expr_sum ⇒ nat and
  eval_prod :: expr_prod ⇒ nat
where
  eval (XSum s) = eval_sum s
| eval (XProd p) = eval_prod p
| eval (XConst 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 subsections, some datatypes are defined by nested recursion. This arises whenever a datatype is embedded inside another type constructor. For example, the *tree* datatype contains itself in the second argument to the Node constructor, nested inside the *list* datatype. In such cases, it is possible to recurse through the nesting BNF's map function (which is simply called map for *list*):

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

As a rule, the indirect recursive call must be a map term that follows the structure of the nesting at the type level (e.g., map f). The functions being mapped must be either the identity (id or $\lambda x.\ x$) or the recursive function being defined (e.g., f), and the argument to the map term must be an unmodified constructor argument (e.g., *cs*). There are some exceptions: Instead of map g (map … (map f x)…), where f is one of the functions being defined, users are allowed to write the more natural map $(g \circ \cdots \circ \mathsf{f})\ x$ or even map $(\lambda v.\ g\ (\ldots\ (\mathsf{f}\ v)\ \ldots))\ x$. This last form is especially useful when the mapped function expect nonrecursive arguments after its recursive argument, as in the next example:

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

The tree_apply function builds a tree of values from a tree of self-mappings and an initial value, storing all intermediate values along the paths from the root to the leaves. For this definition, the λ-abstraction style is more intuitive than the alternatives:

```
Node (v x) (map ((λr. r (v x)) ∘ tree_apply) cs)
Node (v x) (map (λr. r (v x)) (map tree_apply cs))
```

3.4 Nested-as-Mutual Recursion

For compatibility with the old datatype package, but also because this can be convenient in its own right, users can treat nested recursive datatypes as if they were mutually recursive. This is supported if the recursion takes place through other (new-style) datatypes. For example:

```
primrec
  filter_subtree :: (α ⇒ bool) ⇒ α tree ⇒ α tree and
  filter_subtrees :: (α ⇒ bool) ⇒ α tree list ⇒ α tree list
where
  filter_subtree p (Node x ts) = Node x (if p x then filter_subtrees p ts else Nil)
| filter_subtrees _ Nil = Nil
| filter_subtrees p (Cons t ts) = Cons (filter_subtree p t) (filter_subtrees p ts)
```

4 Implementation of Primitively Recursive Functions

In the absence of **primrec**, users could define recursive functions using suitable arguments to *recursors* associated with the datatypes on which they want to recurse. Using the theorems associated with the recursors, it is then possible to prove statements about the defined functions. The **primrec** command automates this process: Given high-level specifications of a function's desired behavior, it synthesizes a recursor-based definition and derives the user specification as theorems.

4.1 Recursors

Recursors encode the most general form of primitive recursion over a datatype. For example, the types α *list* and α *tree* introduced in Section 2 are equipped with the recursors

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

characterized by the following equations:

list.rec:

$$\mathsf{rec\_list}\ n\ \_\ \mathsf{Nil} = n$$
$$\mathsf{rec\_list}\ n\ c\ (\mathsf{Cons}\ y\ ys) = c\ y\ ys\ (\mathsf{rec\_list}\ n\ c\ ys)$$

tree.rec:

$$\mathsf{rec\_tree}\ e\ n\ \mathsf{TEmpty} = e$$
$$\mathsf{rec\_tree}\ e\ n\ (\mathsf{Node}\ x\ ts) = n\ x\ (\mathsf{map}\ (\lambda t.\ (t, \mathsf{rec\_tree}\ e\ n\ t))\ ts)$$

Given a recursor that takes k arguments, we will refer to the first $k-1$ arguments as its *behavioral functions*. For *list*, the behavioral functions n and c encode the behavior for the Nil and Cons cases, respectively. Notice that c is given not only the head (y) and tail (ys) of the list but also the result of the recursive call of rec_list $n\ c$ on the tail. For *tree*, the results of the recursive calls are paired with their arguments inside a list, as reflected in the type $(\alpha\ \mathit{tree} \times \beta)\ \mathit{list}$.

Using the recursor for lists, the length function can be defined as follows:

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

From this definition and the *list.rec* properties, it is easy to derive a high-level characterization of the length function:

lemma *length_simps* [*simp*] :
  length Nil $= 0$
  length (Cons $x\ xs$) $= 1 +$ length xs
  unfolding *length_def list.rec* **by** *safe*

A set of recursors associated with mutually recursive datatypes takes one behavioral function per constructor as arguments. They serve as a description of how the constructor arguments and the results of recursive calls are combined to give the desired return values. In general:

- Constructor arguments on which recursion cannot be performed are passed as is to the behavioral functions.
- Constructor arguments x on which direct or mutual recursion can be performed are passed together with the result y of passing x to a recursive call.
- Constructor arguments on which nested recursion through a map function can be performed contain pairs (x, y) that combine the original value x before the recursion (the *pre* value) and the recursive call's result y (the *post* value).

In the mutual case, each datatype partaking in mutual recursion has its own recursor, but the recursors differ only in the last argument and in the result type. Each recursor serves thus as its type's entry point to mutual recursion.

4.2 General Procedure

The general procedure implemented by **primrec** performs the following steps:

1. From each equation, extract the following information:
   - the function that the equation talks about;
   - the recursive type (deduced from the function's argument types and the position of the constructor pattern);
   - the pattern-matched constructor and the names of its arguments;
   - the names and types of the other arguments;
   - the equation's right-hand side.
2. Query the datatype database to obtain the relevant information (constructors, recursors, theorems) about the recursive types. If nested-as-mutual recursion takes place (as can be detected from the recursive types and the shape of the recursive calls on the right-hand sides), suitable recursors and theorems are derived at this stage [**?**].
3. Traverse the right-hand side of each equation in order to locate recursive calls and replace them by a nonrecursive term that will eventually use the additional arguments the recursor provides to describe the recursion.
4. Introduce λ-abstractions in front of each of these modified right-hand sides and use them as behavioral functions, filling up missing specifications with `undefined`.
5. For each recursor, permute the resulting term's arguments using λ-abstractions to pull the constructor to the front and define the desired function as equal to this term.
6. Using the recursor theorems, prove that the definition fulfills the user specification (the function's characteristic theorems).

4.3 Eliminating Recursive Calls

The second step of the procedure described above depends on information about the structure of recursion in the upcoming definition. This poses a chicken-and-egg problem, since it is specifically this step that supplies all the information about the involved types and their properties. To overcome this difficulty, the process of eliminating recursion from the specification is divided into two steps. First, everything that looks like a recursive call is extracted and passed to the underlying machinery, and the structure of the calls is checked against the information from the datatype database. In the second step, the exact datatype information is used to replace the recursive terms by a recursor-based equivalent.

In more detail, for an equation

$$\mathsf{f}_m\ \bar{l}\ (\mathsf{C}\ \bar{x})\ \bar{r} = \mathit{rhs}$$

the following steps are performed:

1. Starting with t set to rhs, do the following:
   1.1. If t is of the form $\lambda v.\ t'$, recursively apply this procedure to t'.
   1.2. If t is not a function application, stop. Otherwise, write t as $G\ a_1\ \ldots\ a_k$, where G is not an application.
   1.3. If none of the a_j's 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's.
   1.4. Define g as the partial application of G to the longest prefix $a_1, \ldots, a_j$ such that none of its elements are a 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's as a subterm, stop. Otherwise, g is recursively applied to x_i and t is a recursive call.
2. Traverse the right-hand side rhs again to convert any legal recursive calls to terms that use the additional arguments passed by the recursor. This involves applying the following transformations to a term $g\ x_i$:
   2.1. If g does not contain any of the f_j's and either no recursion or mutual recursion can be performed through x_i, replace x_i by the corresponding x_i argument to the behavioral function.
   2.2. If g does not contain any of the f_j's and nested recursion can be performed through x_i, replace x_i by $\mathsf{map}\ \mathsf{fst}\ z_i$, where map is x_i's type's map function and z_i is the argument to the behavioral function that contains pairs (x, y) of pre and post values.
   2.3. If g is one of the f_j, replace $g\ x_i$ by the corresponding y_i argument to the behavioral function.
   2.4. If g is of the form $\mathsf{map}\ (h_1 \circ \cdots \circ h_k \circ \mathsf{f}_j)$, where map is the map function of x_i's type, this is a nested recursive call. Replace $g\ x_i$ by $\mathsf{map}\ (h_1 \circ \cdots \circ h_k \circ \mathsf{snd})\ z_i$, where z_i is the argument to the behavioral function that contains pairs (x, y) of pre and post values.

To simplify the description, we assumed that the map function is a constant that takes a single function argument. In general, it may take several arguments and have a more complicated structure.

4.4 Reordering the Recursive Function's Arguments

In its original form, a recursor instantiation provides a single argument, of the type the recursor consumes. For a recursive type τ, this gives us functions $\mathsf{f} :: \tau \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n$ by instantiating the recursor's return type with $\sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n$. However, **primrec** should allow arguments before the recursive argument. This is made possible by permuting the argument. A specification $\mathsf{f} :: \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_{k-1} \Rightarrow \tau \Rightarrow \sigma_{k+1} \Rightarrow \cdots \Rightarrow \sigma_n$ is internally converted to $\mathsf{f0} :: \tau \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_{k-1} \Rightarrow \sigma_{k+1} \Rightarrow \cdots \Rightarrow \sigma_n$. Once a definition for f0 is obtained, f can easily be defined as $\mathsf{f} = (\lambda a_1 \ldots a_{k-1}\, x.\ \mathsf{f0}\ x\ a_1 \ldots a_{k-1})$.

5 Background: Codatatypes

The syntax for defining codatatypes is almost identical to that for datatypes. But unlike datatypes, codatatypes can be built by applying constructors infinitely, resulting in infinite terms. This is reflected in the generated characteristic theorems: A coinduction (or bisimulation) principle replaces induction. As a result, there need not be a base case that forces (primitively) recursive functions to terminate. Thus, the type of infinite streams of data can be defined using the command

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

whereas the corresponding “datatype” would be empty (and hence impossible to define in HOL). Every instance of α *stream* has an infinite number of constructors. This implies that any function that constructs a stream one constructor at a time must call itself an infinite number of times.

Other examples are the coinductive counterparts to *list* and *nat*:

codatatype α *llist* (**map**: lmap) = lnull: LNil | LCons (lhd: α) (ltl: α *llist*)
codatatype *enat* = EZero | ESuc (epred: *enat*)

The type α *llist* represents lazy (or coinductive) lists with a finite or infinite number of elements, whereas *enat* holds the extended natural numbers, consisting of finite terms of the form ESuc^k EZero and of the special value ESuc (ESuc ...) representing ∞.

Analogously to datatypes, codatatypes support mutually corecursive and nested corecursive definitions. Examples include infinitely branching trees of potentially infinite depth, which can be defined by simply substituting **codatatype** for **datatype_new** and *llist* for *list* in the mutual and nested examples from Section 2:

codatatype α *ltree0* = LNode0 (ltval0: α) (lchildren0: α *lforest*)
  and α *lforest* = LFNil | LFCons (lfhd: α *ltree0*) (lftl: α *lforest*)
codatatype α *ltree* = LNode (ltval: α) (lchildren: α *ltree llist*)

6 Specification of Primitively Corecursive Functions

Corecursive functions can be specified using **primcorec** and **primcorecursive**, which support primitive corecursion, or using the more general **partial_function** command. Alternatives based on domain theory and topology are described by Lochbihler and Hölzl [**?**]. Here, the focus is on **primcorec** and **primcorecursive**.

Whereas recursive functions consume datatype values one constructor at a time, corecursive functions produce potentially infinite codatatype values one constructor at a time. Partly reflecting a lack of agreement among proponents of coinductive methods, Isabelle supports three competing syntaxes for specifying a function f:

- The *destructor view* specifies f by implications of the form $\ldots \Longrightarrow \mathsf{is\_C}_j\ (\mathsf{f}\ \bar{x})$ and equations of the form $\mathsf{un\_C}_{ji}\ (\mathsf{f}\ \bar{x}) = \ldots$. This style is popular in the coalgebraic literature.
- The *constructor view* specifies f by equations of the form $\ldots \Longrightarrow \mathsf{f}\ \bar{x} = \mathsf{C}_j \ldots$. This style is often more concise than the previous one.
- The *code view* specifies f by a single equation of the form $\mathsf{f}\ \bar{x} = \ldots$, 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.

6.1 The Destructor View

Specifications for the function $\mathsf{f} :: \bar{\sigma} \Rightarrow \bar{\tau}\ \kappa$ in the *destructor view* consist of two kinds of formula. *Discriminator formulas* have the form

$$P_1\ \bar{x} \Longrightarrow \cdots \Longrightarrow P_n\ \bar{x} \Longrightarrow \mathsf{is\_C}_i\ (\mathsf{f}\ \bar{x})$$

where the P_j's are some condition on the function arguments $\bar{x}$ and $\mathsf{is\_C}_i$ is a discriminator for the codatatype κ. *Selector equations* have the form

$$\mathsf{un\_C}_{ij}\ (\mathsf{f}\ \bar{x}) = g\ \bar{x}$$

where $\mathsf{un\_C}_{ij}$ is the jth selector for C_i. The discriminator formulas specify the conditions under which the function produces a constructor C_i, whereas the selector equations specify the arguments to C_i.

For the function to be well defined, the discriminator formulas' conditions must be mutually exclusive. The **primcorec** attempts to discharge the corresponding proof obligations automatically. If this fails, users can fall back on the longer form **primcorecursive**, which passes the burden of proof to them. Thus, **primcorec** … can be seen as an abbreviation for **primcorecursive** … **by** auto?.

As an example, consider the lapp function that concatenates two (potentially infinite) lists of the same type. It can be specificed in the destructor view as follows:

```
primcorec lapp :: α llist ⇒ α llist ⇒ α llist where
  lnull xs ⟹ lnull ys ⟹ 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)
```

The specification is slightly ambiguous because there is no discriminator formula for the $\neg$ lnull case (i.e., the LCons case). When only one constructor case is left out, **primcorec** fills in the gap in the obvious way:

$$\neg\ \mathsf{lnull}\ xs \vee \neg\ \mathsf{lnull}\ ys \Longrightarrow \neg\ \mathsf{lnull}\ (\mathsf{lapp}\ xs\ ys)$$

Syntactic restrictions on the selector equations ensure that progress is made with each corecursive call—i.e., the function is *productive*. Productivity guarantees that even when the result of lapp is infinite, prefixes of arbitrary finite length can be computed by expanding a finite number of corecursive calls. This, in turn, ensures that the function is well defined.

The main syntactic restriction on the selector equations, beyond the fixed format of the left-hand side, is that any corecursive call on the right-hand side either occupies the entire right-hand side or appears as a branch in a 'if–then–else', 'case–of', or 'let–in' construct. Because of this restriction, the following specification must be rejected:

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

Some codatatypes reuse the same selector functions for several constructors. Ambiguities can then arise when connecting the selectors to the corresponding constructors. To resolve this issue, **primcorec** accepts selector equations of the form

get (f ...) $= \ldots$ **of** C

where get is an ambiguous selector and C is a constructor.

To avoid the need of tedious manual specification of an 'else' predicate for the discriminator formulas, a single 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

$P\ x \Longrightarrow \ldots$
$Q\ x \Longrightarrow \ldots$
$\_ \Longrightarrow \ldots$

the last equation's condition is taken to be $\neg\ P\ x \wedge \neg\ Q\ x$.

A related functionality is provided by the *sequential* option. It causes the discriminator formula conditions for a function to apply in sequence, rather than independently of each other. This relieves the user (or *auto*) from having to show mutual exclusion, but the generated theorems then feature more complicated conditions.

Finally, the *exhaustive* option signals that the discriminator formula premises cover all cases. Specifying this option adds another proof obligation, which again is either solved automatically by **primcorec** or left to the user by **primcorecursive**. In exchange, stronger theorems are generated about the discriminators, with $\longleftrightarrow$ in place of $\Longrightarrow$. Another way to enable this behavior is to specify _ as the last condition, in which case exhaustiveness is syntactically trivial.

6.2 The Constructor View

The *constructor view* combines the discriminator formula and the selector equations associated with a constructor in a single equation. The general form is

$$P_1\ \bar{x} \Longrightarrow \cdots \Longrightarrow P_n\ \bar{x} \Longrightarrow \mathsf{f}\ \bar{x} = \mathsf{C}_i\ (g_1\ \bar{x}) \ldots (g_k\ \bar{x})$$

The P_i's are conditions just like in the destructor view, and C_i is a constructor. The lapp function can be defined as follows:

```
primcorec lapp :: α llist ⇒ α llist ⇒ α llist where
  lnull xs ⟹ lnull ys ⟹ lapp xs ys = LNil
| _ ⟹ lapp xs ys = LCons (if lnull xs then lhd ys else lhd xs)
                           (if lnull xs then ys else lapp (ltl xs) ys)
```

Due to the close correspondence between constructor and destructor view, the constraints on constructor arguments are the same as the requirements to a selector equation right-hand side. Additionally, just like each discriminator must occur in at most one discriminator formulas, there must be at most one equation by constructor.

6.3 The Code View

Specifications in the code view consist of a single equation of the form

$$\mathsf{f}\,\bar{x} = g\,\bar{x}$$

with conditionals encoded as 'if–then–else' and 'case–of' on the right-hand side. This form is suitable for Isabelle's code generator, which can work only with unconditional equations. As an input format, it also provides some more flexibility, sometimes leading to simpler definitions than are possible with the other two views. This is clearly the case for lapp:

```
primcorec lapp :: α llist ⇒ α llist ⇒ α llist ⇒ α llist where
  lapp xs ys = (case xs of LNil ⇒ ys | LCons x xs′ ⇒ LCons x (lapp xs′ ys))
```

In general, the right-hand side may involve arbitrarily nested conditional and 'let–in' expressions. The branches must either be protected by a constructor or involve no corecursion. Unlike with the other two views, the conditionals may be arbitrarily nested, and the same constructor may occur in several branches.

6.4 Mutual Corecursion

Functions that return values of mutually corecursive codatatypes must be defined together. The following example specifies a function treeify $m\ n$ that constructs a full n-ary tree of depth m whose values are all () and where both m and n are extended natural numbers (i.e., may be ∞), together with an auxiliary function treeifys for forests:

```
primcorec
  treeify :: enat ⇒ enat ⇒ unit ltree0 and
  treeifys :: enat ⇒ enat ⇒ enat ⇒ unit lforest
where
  ltval0 (treeify m n) = ()
| lchildren0 (treeify m n) = treeifys (epred m) n n
| m = EZero ∨ i = EZero ⟹ treeifys m n i = LFNil
| _ ⟹ treeifys m n i = LFCons (treeify m n) (treeifys m n (epred i))
```

The example shows that it is possible to mix the views: treeify is expressed in the destructor view, whereas the auxiliary treeifys is in the constructor view.

6.5 Nested Corecursion

For nested corecursion, we consider a more elaborate example: the definition of a monadic structure for *ltree*. Unfortunately, the standard way to define the bind operator $\gg\!= :: \alpha\ ltree \Rightarrow (\alpha \Rightarrow \beta\ ltree) \Rightarrow \beta\ ltree$, as is done for the monad instance `Monad Tree` in the Haskell library `Data.Tree`, is not primitively corecursive:

$$\begin{aligned} t \gg\!= f = (&\text{case } f\ (\mathsf{ltval}\ t) \text{ of } \mathsf{LNode}\ b\ us \Rightarrow \\ &\quad \mathsf{LNode}\ b\ (\mathsf{lapp}\ us\ (\mathsf{lmap}\ (\lambda t.\ t \gg\!= f)\ (\mathsf{lchildren}\ t)))) \end{aligned}$$

The problem is that the corecursive call to $\gg\!=$ nested through lmap occurs not directly as an argument to the produced constructor LNode, but as an argument to lapp. Fortunately, we can make the specification primitively corecursive by moving the corecursive call past lapp. We use the sum type to distinguish between the subtrees *us* spawned by f (Inl) and the subtrees $\mathsf{lchildren}\ t$ to which the corecursive call is to be applied (Inr):

$$\begin{aligned} &\textbf{primcorec} \gg\!= :: \alpha\ ltree \Rightarrow (\alpha \Rightarrow \beta\ ltree) \Rightarrow \beta\ ltree\ \textbf{where} \\ &\quad t \gg\!= f = (\text{case } f\ (\mathsf{ltval}\ t) \text{ of } \mathsf{LNode}\ b\ us \Rightarrow \\ &\qquad \mathsf{LNode}\ b\ (\mathsf{lmap}\ (\lambda ut.\ \text{case } ut \text{ of } \mathsf{Inl}\ u \Rightarrow u \mid \mathsf{Inr}\ t \Rightarrow t \gg\!= f) \\ &\qquad\qquad (\mathsf{lapp}\ (\mathsf{lmap}\ \mathsf{Inl}\ us)\ (\mathsf{lmap}\ \mathsf{Inr}\ (\mathsf{lchildren}\ t))))) \end{aligned}$$

From the lemma $\mathsf{lmap}\ f\ (\mathsf{lapp}\ xs\ ys) = \mathsf{lapp}\ (\mathsf{lmap}\ f\ xs)\ (\mathsf{lmap}\ f\ ys)$, it is easy to derive the Haskell-style nonprimitively corecursive specification. Given $\mathsf{return}\ x$ defined as $\mathsf{LNode}\ x\ \mathsf{LNil}$, proving the monadic laws

$$\begin{aligned} &\mathsf{return}\ x \gg\!= f\ = f\ x \\ &t \gg\!= \mathsf{return}\ = t \\ &t \gg\!= f \gg\!= g\ = t \gg\!= (\lambda x.\ f\ x \gg\!= g) \end{aligned}$$

is an interesting exercise in coinduction.

6.6 Nested-as-Mutual Corecursion

To demonstrate the convenience of nested-as-mutual coinduction, let us define the $\gg\!=$ function again, but this time mutually with the ternary mixfix operator $\_ \nabla \_ \gg \_$ that returns a lazy list, instead of reusing lapp:

$$\begin{aligned} &\textbf{primcorec} \\ &\quad \gg\!= :: \alpha\ ltree \Rightarrow (\alpha \Rightarrow \beta\ ltree) \Rightarrow \beta\ ltree\ \textbf{and} \\ &\quad \_ \nabla \_ \gg \_ :: \beta\ ltree\ llist \Rightarrow \alpha\ ltree\ llist \Rightarrow (\alpha \Rightarrow \beta\ ltree) \Rightarrow \beta\ ltree\ llist \\ &\textbf{where} \\ &\quad t \gg\!= f = (\text{case } f\ (\mathsf{ltval}\ t) \text{ of } \mathsf{LNode}\ b\ us \Rightarrow \mathsf{LNode}\ b\ (us\ \nabla\ \mathsf{lchildren}\ t \gg f)) \\ &\mid us\ \nabla\ ts \gg f = (\text{case } us \text{ of} \\ &\qquad\qquad \mathsf{LNil} \Rightarrow (\text{case } ts \text{ of} \\ &\qquad\qquad\qquad \mathsf{LNil} \Rightarrow \mathsf{LNil} \\ &\qquad\qquad\qquad \mid \mathsf{LCons}\ t\ ts' \Rightarrow \mathsf{LCons}\ (t \gg\!= f)\ (us\ \nabla\ ts' \gg f)) \\ &\qquad\qquad \mid \mathsf{LCons}\ u\ us' \Rightarrow \mathsf{LCons}\ u\ (us'\ \nabla\ ts \gg f) \end{aligned}$$

The new operator's intended semantics is

$us \nabla ts \gg f = \mathsf{lapp}\ us\ (\mathsf{lmap}\ (\lambda t.\ t \ggg f)\ ts)$

When defining functions mutually, we are not restricted to corecursive calls through the map function only. Thus, we can use the required primitive corecursion scheme directly, without working around the syntactic restriction by using the sum type as above. The **primcorec** command creates the corresponding mutual coinduction rule to reason about $\ggg$ and _ ∇ _ $\gg$ _, allowing us to prove $\ggg$ equal to its version from Section 6.5.

The above specification could easily be adapted for the mutual codataypes *ltree0* and *lforest*, by substituting *ltree0* for *ltree*, *lforest* for *ltree llist*, ltval0 for ltval, etc.

6.7 Generated Theorems

Regardless of the user's choice of input syntax, the **primcorec** command generates characteristic theorems for all three views. For example, the definition

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

given in the constructor view produces the following theorems (among others):

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

The *iterate_while.disc_iff* theorems are produced because the '_' wildcard implicitly enables the *exhaustive* option.

The *disc*, *disc_iff*, and *sel* theorems are registered as simplification rules. This is not done for the *code* and *ctr* theorems by default because they can loop.

7 Implementation of Primitively Corecursive Functions

The implementation of **primcorec** (and **primcorecursive**) follows the same general principle as that of **primrec**: From a user specification, **primcorec** synthesizes a low-level definition based on *corecursors* and derives theorems about the specified function from the characteristic theorems associated with the codatatype and its corecursor.

7.1 Corecursors

Corecursors encode the most general form of primitive corecursion over a codatatype. The types α *llist* and α *ltree* introduced in Section 5 are equipped with the corecursors

$$\begin{aligned}&\mathsf{corec\_llist} :: (\alpha \Rightarrow bool) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow bool) \Rightarrow (\alpha \Rightarrow \beta\ llist) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \\ &\qquad\qquad \alpha \Rightarrow \beta\ llist \\ &\mathsf{corec\_ltree} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow (\beta\ ltree + \alpha)\ llist) \Rightarrow \alpha \Rightarrow \beta\ ltree\end{aligned}$$

characterized by the following equations:

llist.corec:

$$\begin{aligned}&n\ a \Longrightarrow \mathsf{corec\_llist}\ n\ h\ s\ e\ c\ a = \mathsf{LNil} \\ &\neg\ n\ a \Longrightarrow \mathsf{corec\_llist}\ n\ h\ s\ e\ c\ a = \\ &\qquad \mathsf{LCons}\ (h\ a)\ (\mathsf{if}\ s\ a\ \mathsf{then}\ e\ a\ \mathsf{else}\ \mathsf{corec\_llist}\ n\ h\ s\ e\ c\ (c\ a))\end{aligned}$$

ltree.corec:

$$\begin{aligned}&\mathsf{corec\_ltree}\ l\ c\ a = \\ &\mathsf{LNode}\ (l\ a)\ (\mathsf{lmap}\ (\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Inl}\ t \Rightarrow t \mid \mathsf{Inr}\ r \Rightarrow \mathsf{corec\_ltree}\ l\ c\ r)\ (c\ a))\end{aligned}$$

Given a corecursor that takes k arguments, we will refer to the first $k-1$ arguments as its *behavioral functions*.

For *llist*, the predicate n indicates whether a LNil constructor should be produced. Next follows h, a function that computes the head of a nonempty list from the input a. The next three functions, s, e, and c, specify the list's tail: s ("stop?") is a predicate that determines whether the corecursion ends with a noncorecursive term or continues further; e ("end") gives the noncorecursive tail if s is satisfied; c ("continue") computes the argument to a corecursive call if s is not satisfied, specifying a corecursive tail.

For *ltree*, the behavioral function constructs a list of sum values that specify how to produce each subtree individually. An Inl value is interpreted as a literal noncorecursive result, whereas an Inr value is passed to a corecursive call.

In general, if the codatatype σ has n constructors, the corecursor expects $n-1$ predicates $p_j :: \alpha \Rightarrow bool$ that are tested in sequence to determine which constructor should be produced. Additionally, for each constructor argument of type τ, one of the following cases applies:

- If τ does not contain σ (or any of its mutually corecursive types), no corecursion is possible and the behavioral function is simply a function $g :: \alpha \Rightarrow \tau$ that returns the constructor argument's unconditional value.
- If τ's outermost type constructor is σ (or a mutually recursive type), this constructor argument allows direct (or mutual) corecursion. The corecursor expects three behavioral functions: $s :: \alpha \Rightarrow bool$ ("stop?"), $e :: \alpha \Rightarrow \tau$ ("end"), and $c :: \alpha \Rightarrow \alpha$ ("continue").
- If τ nests σ under one or more BNFs, the corecursor takes one argument that returns a value of τ's nesting type wrapped around $\sigma + \alpha$. The sum type represents either a noncorecursive constant result or a corecursive call's argument.

7.2 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. The code view is reduced to the constructor view, which in turn is reduced to the destructor view. Each reduction works by disassembling the input as far as necessary and creating equivalent specifications in the next input style. These specifications are then passed down to the parsing functions for said 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. When it comes to generating the function's characteristic theorems, the path of reductions is traversed backward: The code-view theorems are derived from the constructor-view theorems, which are in turn derived from the destructor-view theorems.

Constructor View to Destructor View. An equation

$$P_1\ \bar{x} \implies \cdots \implies P_n\ \bar{x} \implies \mathsf{f}\ \bar{x} = \mathsf{C}\ t_1 \ldots t_m$$

expressed in constructor view is reduced to the equivalent destructor view

$$\begin{array}{l} P_1\ \bar{x} \implies \cdots \implies P_n\ \bar{x} \implies \mathsf{is\_C}\ (\mathsf{f}\ \bar{x}) \\ \mathsf{un\_C}_1\ (\mathsf{f}\ \bar{x}) = t_1 \\ \qquad \vdots \\ \mathsf{un\_C}_m\ (\mathsf{f}\ \bar{x}) = t_m \end{array}$$

where $\mathsf{is\_C}$ is the m-ary constructor C's discriminator and $\mathsf{un\_C}_i$ are its selectors.

Code View to Constructor View. Recall that a specification in code view consists of a single equation, potentially having many case distinctions via 'if–then–else', 'case–of', or 'let–in' expressions. 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, noncorecursive branches that are not guarded by a constructor are expanded using 'case'. For example, a term xs of type $\alpha\ llist$ becomes

$$(\mathsf{case}\ xs\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathsf{LNil} \mid \mathsf{LCons}\ x\ xs' \Rightarrow \mathsf{LCons}\ x\ xs')$$

Along each of the paths, the set of conditions that need to be fulfilled to reach the current node is carried along. After the formulas have been collected, they are ready to be combined into one single equation per constructor.

7.3 Generating Theorems in All Three Views

The syntax reductions described in the previous subsection induce the need to traverse the reduction's path backward. For the view chosen by the user to specify the function, the statements of the theorems are given by the specification; for the others, suitable statements are generated, as explained below.

Constructor View from Destructor View. Essentially, this just uses the reduction described in Section 7.2 in reverse. Each constructor and its associated discriminator formula premises and selector equation right-hand sides are collected and combined to form a constructor view. However, no equation is generated if some selector equations are missing for a given constructor (e.g., if the head of a list is specified but not the tail).

Code View from Constructor View. Analogously, this step reverses the reduction in Section 7.2. It takes the constructor-view 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, which throws an exception at run time if none of the cases apply. Either way, the newly assembled terms are proved as theorems, exploiting the constructor-view theorems.

7.4 General Procedure

The general procedure implemented by **primcorec** performs the following steps:

1. Use the functions' types to query the involved constructors, discriminators, and selectors from the codatatype database.
2. From each supplied formula's structure, determine the kind of formula. For constructor or code equations, call the respective reduction functions to extract the same internal, destructor-based representation from all three input syntaxes.
   During this step, the interpretation of '_' wildcards and the *sequential* option are performed and any implicit discriminator formulas are generated.
   After this step, for each (specified or generated) formula, we have:
   - the function's name, type, and its arguments' names and types as it 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 formulas and selector equations are a list of premises and the right-hand side.
3. The selector equation right-hand sides are scanned for corecursive calls and their structure is recorded.
4. Using this new information, get the rest of the codatatypes' information (corecursors, theorems, types of corecursion, etc.) from the codatatype database. The nested-to-mutual reduction is performed in this step if necessary.
5. Definitions for the specified functions are obtained. This involves translating selector equation right-hand sides to behavioral functions.
6. Any exclusiveness and exhaustiveness properties are assembled.
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.

7.5 Eliminating Corecursive Calls

Corecursion is simpler than recursion in at least one respect: Whereas **primrec** needs complicated logic to locate (direct and indirect) recursive calls, the syntactic restrictions that ensure productivity also ensure that **primcorec** knows where to find the corecursive calls. 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 equation right-hand side—except for 'if–then–else', 'case–of', and 'let–in'—it suffices to traverse the terms to determine for each leaf whether it is a corecursive call and, if so, replace it by a suitable argument of the behavioral function:

- If the constructor argument does not allow any corecursion, its selector equation right-hand side is converted to a behavioral function by λ-abstracting the function arguments.
- For mutual corecursion, we need to generate three behavioral functions: The predicate s ("stop?") is created by substituting either True for a noncorecursive leaf or False for a corecursive leaf; e ("end") is formed by replacing corecursive leaves by $\mathsf{undefined}$; and c ("continue") is obtained by substituting $\mathsf{undefined}$s for noncorecursive leaves and a tuple of the corecursive call's function arguments for corecursive leaves.
- 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 noncorecursive result or a corecursive call. The behavioral function is obtained by replacing a noncorecursive leaf y by $\mathsf{map}\ \mathsf{Inl}\ y$, where map is the nesting type's map function, and a nested corecursive call $\mathsf{map}\ (\mathsf{f} \circ h_1 \circ \cdots \circ h_k)\ a$ by $\mathsf{map}\ (\mathsf{Inr} \circ h_1 \circ \cdots \circ h_k)\ a$.

Like for **primrec**, we assumed that the map function is a constant that takes a single function argument. In general, it may take several argument and have a more complicated structure.

7.6 Supporting Functions with No or Multiple Arguments

The corecursor takes only one argument $a :: \alpha$. When defining a n-ary primitively corecursive function (where $n \geq 0$), an n-tuple of arguments is passed to the corecursor $\mathsf{corec\_}\kappa$ and, consequently, the behavioral functions. The definition is curried:

$$\lambda x_1 \ldots x_n.\ \mathsf{corec\_}\kappa\ (\lambda(x_1,\ldots,x_n).\ \ldots) \ldots (\lambda(x_1,\ldots,x_n).\ \ldots)\ (x_1,\ldots,x_n)$$

8 Conclusion

Isabelle's new (co)datatype package makes it possible to specify a large class of types using a convenient syntax reminescent of typed functional programming languages. The new **primrec**, **primcorec**, and **primcorecursive** commands complement the package by allowing users to specify arbitrary (co)recursive functions on the (co)datatypes.

Like the rest of the (co)datatype package, the commands are fully definitional: They analyze the specifications entered by the user, synthesize definitions in terms of internal (co)recursors, and generate characteristic theorems, all of this without introducing axioms or extending the logic.

Nested recursion is handled truly modularly. Nonetheless, the older approach of reducing nested recursion to mutual recursion is also supported. The two approaches can be arbitrarily combined for both recursive and corecursive functions. Internally, suitable (co)recursors and (co)induction principles are derived to make this possible.

For future work, we are interested in stronger forms of corecursion as well as mixed recursive–corecursive definitions. We have some ideas already, but we need a solid theoretical foundation so that Isabelle's inference kernel can accept the definitions.

Acknowledgment. Tobias Nipkow made this work possible by allowing the senior authors to neglect their duties while they designed and implemented the new (co)datatype package. Andrei Popescu masterminded the metatheory behind the package, without which there would be no (co)datatypes, no (co)recursors, and hence no (co)recursive function definitions. Florian Haftmann, Christian Urban, and Makarius Wenzel provided general advice on Isabelle and package writing. Johannes Hölzl and Andreas Lochbihler gave precious feedback at the early stages of **primcorec**'s implementation.

Blanchette is supported by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant Ni 491/14-1). Traytel is supported by the DFG program Program and Model Analysis (PUMA, doctorate program 1480). The senior authors are listed alphabetically regardless of individual contributions or relative seniority.

References

1. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs '99. LNCS, vol. 1690, pp. 19–36. Springer (1999)
2. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, Springer (2014)
3. Blanchette, J.C., Panny, L., Popescu, A., Traytel, D.: Defining (co)datatypes in Isabelle/HOL. `http://isabelle.in.tum.de/dist/Isabelle/doc/datatypes.pdf` (2013)
4. Krauss, A.: Recursive definitions of monadic functions. EPTCS 43, 1–13 (2010)
5. Krauss, A.: Defining recursive functions in Isabelle/HOL. `http://isabelle.in.tum.de/doc/functions.pdf` (2013)
6. Lochbihler, A., Hölzl, J.: Recursive functions on lazy lists via domains and topologies. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, Springer (2014)
7. Panny, L.: Primitively (Co)recursive Function Definitions for Isabelle/HOL. B.Sc. thesis draft, Technische Universität München (2014)
8. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: LICS 2012, pp. 596–605. IEEE (2012)