Truly Modular (Co)datatypes for Isabelle/HOL

Jasmin Christian Blanchette[1], Johannes Hölzl[1], Andreas Lochbihler[2], Lorenz Panny[1], Andrei Popescu[1,3], and Dmitriy Traytel[1]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Institute of Information Security, ETH Zurich, Switzerland
[3] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania

Abstract. We extended Isabelle/HOL with a pair of definitional commands for datatypes and codatatypes. They support mutual and nested (co)recursion through well-behaved type constructors, including mixed recursion–corecursion, and are complemented by syntaxes for introducing primitive (co)recursive functions and by a general proof method for reasoning coinductively. As a case study, we ported Isabelle's Coinductive library to use the new commands, eliminating the need for tedious ad hoc constructions.

1 Introduction

Coinductive methods are becoming widespread in computer science. In proof assistants such as Agda and Coq, codatatypes and coinduction are intrinsic to the logical calculus. Formalizations involving programming language semantics, such as the CompCert verified C compiler [17], use codatatypes to represent potentially infinite execution traces. The literature also abounds with "coinductive pearls"—papers that demonstrate how coinductive methods lead to more elegant solutions than traditional approaches.

Thus far, provers based on higher-order logic (HOL) have mostly stood on the sidelines of these developments. Isabelle/HOL provides a few manually derived codatatypes (e.g., lazy lists) in the Coinductive entry of the *Archive of Formal Proofs* [18]. This library forms the basis of JinjaThreads [19], a verified compiler for a Java-like language, and of the formalization of the Java memory model [21]. The manual constructions are heavy, requiring hundreds of lines for each codatatype.

Even in the realm of datatypes, there is room for improvement. Isabelle's datatype package was developed by Berghofer and Wenzel [4], who drew on the work of Melham [22], Gunter [11, 12], Paulson [24], and Harrison [14]. The package supports positive recursion through functions and reduces nested recursion through datatypes to mutual recursion, but otherwise allows no nesting. For example, it rejects definitions such as

$$\textbf{datatype}\ \alpha\ \mathit{tree}_{\mathsf{FS}} = \mathsf{Tree}_{\mathsf{FS}}\ \alpha\ (\alpha\ \mathit{tree}_{\mathsf{FS}}\ \mathit{fset})$$

where *fset* designates finite sets (a non-datatype). Moreover, the reduction of nested to mutual recursion makes it difficult to specify recursive functions modularly.

We introduce a definitional package for datatypes and codatatypes that addresses the issues noted above. The key notion is that of a *bounded natural functor* (BNF), a type constructor equipped with map and set functions and a cardinality bound (Section 2). BNFs are closed under composition and least and greatest fixpoints and are expressible in HOL. Users can register well-behaved type constructors such as *fset* as BNFs.

The BNF-based **datatype** and **codatatype** commands provide many conveniences such as automatically generated discriminators, selectors, map and set functions, and relators (Sections 3 and 4). Thus, the command

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

defines the type α *llist* of lazy lists over α, with constructors LNil :: α *llist* and LCons :: $\alpha \Rightarrow \alpha$ *llist* $\Rightarrow \alpha$ *llist*, a discriminator lnull :: α *llist* $\Rightarrow$ *bool*, selectors lhd :: α *llist* $\Rightarrow \alpha$ and ltl :: α *llist* $\Rightarrow \alpha$ *llist*, a set function lset :: α *llist* $\Rightarrow \alpha$ *set*, a map function lmap :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *llist* $\Rightarrow \beta$ *llist*, and a relator lrel :: $(\alpha \Rightarrow \beta \Rightarrow$ *bool*$) \Rightarrow \alpha$ *llist* $\Rightarrow \beta$ *llist* $\Rightarrow$ *bool*. Intuitively, the **codatatype** keyword indicates that the constructors can be applied repeatedly to produce infinite values—e.g., LCons 0 (LCons 1 (LCons 2 ...)).

Nesting makes it possible to mix recursion and corecursion arbitrarily. The next commands introduce the types of Rose trees with finite or possibly infinite branching (*list* vs. *llist*) and with finite or possibly infinite paths (**datatype** vs. **codatatype**):

datatype α *tree* = Tree (lab: α) (sub: α *tree list*)
datatype α *tree*$_\omega$ = Tree$_\omega$ (lab$_\omega$: α) (sub$_\omega$: α *tree*$_\omega$ *llist*)
codatatype α *ltree* = LTree (llab: α) (lsub: α *ltree list*)
codatatype α *ltree*$_\omega$ = LTree$_\omega$ (llab$_\omega$: α) (lsub$_\omega$: α *ltree*$_\omega$ *llist*)

Primitive (co)recursive functions can be specified using **primrec** and **primcorec** (Sections 5 and 6). The function below constructs a possibly infinite tree by repeatedly applying $f :: \alpha \Rightarrow \alpha$ *llist* to x. It relies on lmap to construct the nested *llist* modularly:

primcorec iterate_ltree$_\omega$ :: $(\alpha \Rightarrow \alpha$ *llist*$) \Rightarrow \alpha \Rightarrow \alpha$ *ltree*$_\omega$ **where**
  iterate_ltree$_\omega$ $f\ x$ = LTree$_\omega$ x (lmap (iterate_ltree$_\omega$ f) ($f\ x$))

An analogous definition is possible for α *ltree*, using *list*'s map instead of lmap.

For datatypes that recurse through other datatypes, and similarly for codatatypes, old-style mutual definitions are also allowed. For the above example, this would mean defining iterate_ltree$_\omega$ by mutual corecursion with iterate_lforest$_\omega$ :: $(\alpha \Rightarrow \alpha$ *llist*$) \Rightarrow \alpha$ *llist* $\Rightarrow \alpha$ *ltree*$_\omega$ *llist*. Despites its lack of modularity, the approach is useful both for compatibility and for expressing specifications in a more flexible style. The package generates suitable (co)induction rules to facilitate reasoning about the definition.

Reasoning coinductively is notoriously tedious in Isabelle, because the *coinduct* method requires the user to provide a witness relation. Our new *coinduction* method eliminates the boilerplate that has plagued earlier developments (Section 7). It is now possible to have one-line proofs **by** *coinduction auto*. To show the package in action, we present a theory of stream processors, which combine a least and a greatest fixpoint (Section 8). In addition, we describe our experience porting the Coinductive library to use the new package (Section 9). A formal development accompanies this paper [8].

The package has been part of Isabelle starting with version 2013. The implementation is a significant piece of engineering, at over 17 500 lines of Standard ML code and 2 500 lines of Isabelle formalization. Most of the features described here are available in 2013-2; a few are present only in the development repository. In the current implementation, the BNF-based **datatype** and **primrec** commands are suffixed with **_new** to avoid clashes with the old package. The input syntax and the generated constants and theorems are documented in the user's manual [5].

2 Low-Level Constructions

At the lowest level, each (co)datatype has a single unary constructor. Multiple curried constructors are modeled by disjoint sums ($+$) of products ($\times$). A (co)datatype definition corresponds to a fixpoint equation. For example, the equation $\beta = \mathit{unit} + \alpha \times \beta$ specifies either (finite) lists or lazy lists, depending on which fixpoint is chosen.

Bounded natural functors (BNFs) are a semantic criterion for where (co)recursion may appear on the right-hand side of an equation. The theory of BNFs is described in a previous paper [28] and in Traytel's M.Sc. thesis [29]. We refer to either of these for a discussion of related work. Here, we focus on implementational aspects.

There is a large gap between the low-level view and the end products presented to the user. The necessary infrastructure—including support for multiple curried constructors, generation of high-level characteristic theorems, and commands for specifying functions—constitutes a new contribution and is described in Sections 3 to 6.

Bounded Natural Functors. An n-ary BNF is a type constructor equipped with a map function, n set functions, and a cardinal bound that satisfy certain properties. For example, the map and set functions associated with α *llist* are lmap and lset, and the relator lrel extends binary predicates over elements to binary predicates over lazy lists:

$$\mathsf{lrel}\ R\ \mathit{xs}\ \mathit{ys} = (\exists \mathit{zs}.\ \mathsf{lset}\ \mathit{zs} \subseteq \{(x, y) \mid R\ x\ y\} \wedge \mathsf{lmap}\ \mathsf{fst}\ \mathit{zs} = \mathit{xs} \wedge \mathsf{lmap}\ \mathsf{snd}\ \mathit{zs} = \mathit{ys})$$

Additionally, lbd bounds the number of elements returned by lset; it may not depend on α's cardinality. To prove that *llist* is a BNF, the greatest fixpoint operation discharges the following proof obligations:[1]

$$\begin{array}{lll} \mathsf{lmap}\ \mathsf{id} = \mathsf{id} & \mathsf{lmap}\ (f \circ g) = \mathsf{lmap}\ f \circ \mathsf{lmap}\ g & \dfrac{\bigwedge x.\ x \in \mathsf{lset}\ \mathit{xs} \Longrightarrow f\ x = g\ x}{\mathsf{lmap}\ f\ \mathit{xs} = \mathsf{lmap}\ g\ \mathit{xs}} \\ |\mathsf{lset}\ \mathit{xs}| \leq_{\mathsf{o}} \mathsf{lbd} & \mathsf{lset} \circ \mathsf{lmap}\ f = \mathsf{image}\ f \circ \mathsf{lset} & \\ \aleph_0 \leq_{\mathsf{o}} \mathsf{lbd} & \mathsf{lrel}\ R \circ\circ \mathsf{lrel}\ S \sqsubseteq \mathsf{lrel}\ (R \circ\circ S) & \end{array}$$

(The operator $\leq_{\mathsf{o}}$ is a well-order on ordinals [6], and $\circ\circ$ denotes the relational composition of binary predicates.) Internally, the package stores BNFs as an ML structure that combines the functions, the basic properties, and derived facts such as $\mathsf{lrel}\ R \circ\circ \mathsf{lrel}\ S = \mathsf{lrel}\ (R \circ\circ S)$, $\mathsf{lrel}\ (\mathsf{op} =) = (\mathsf{op} =)$, and $R \sqsubseteq S \Longrightarrow \mathsf{lrel}\ R \sqsubseteq \mathsf{lrel}\ S$.

Given an n-ary BNF, the n type variables associated with set functions, and on which the map function acts, are *live*; any other variables are *dead*. The notation $\sigma\ \langle \overline{\alpha}\,|\,\Delta \rangle$ stands for a BNF of type σ depending on the (ordered) list of live variables $\overline{\alpha}$ and the set of dead variables Δ. Nested (co)recursion can only take place through live variables.

A two-step procedure constructs (co)datatypes as solutions to fixpoint equations:

1. Construct the BNFs for the right-hand sides of the equations by composition.
2. Perform the least or greatest fixpoint operation on the BNFs.

Whereas codatatypes are always nonempty, some datatype definitions must be rejected in HOL. For example, the type of infinite streams can be defined only as a codatatype: **codatatype** $\alpha\ \mathit{stream} = \mathsf{SCons}\ (\mathsf{shd}\colon \alpha)\ (\mathsf{stl}\colon \alpha\ \mathit{stream})$. In the general BNF setting, each functor must keep track of its nonemptiness witnesses [7].

[1] The list of proof obligations has evolved since our previous work [28]. The redundant cardinality condition $|\{\mathit{xs} \mid \mathsf{lset}\ \mathit{xs} \subseteq A\}| \leq_{\mathsf{o}} (|A| + 2)^{\mathsf{lbd}}$ has been removed, and the preservation of weak pullbacks has been reformulated as a simpler property of the relator.

The Fixpoint Operations. The LFP operation constructs a least fixpoint solution $\tau_1, \ldots, \tau_n$ to n mutual fixpoint equations $\beta_j = \sigma_j$. Its input consists of n BNFs sharing the same live variables [28, 29]:

LFP : n $(m+n)$-ary BNFs $\sigma_j \langle \overline{\alpha}, \overline{\beta} \,|\, \Delta_j \rangle \rightarrow$
- n m-ary BNFs $\tau_j \langle \overline{\alpha} \,|\, \Delta_1 \cup \cdots \cup \Delta_m \rangle$ for newly defined types τ_j
- n constructors $\mathsf{ctor\_}\tau_j :: \sigma_j[\overline{\beta} \mapsto \overline{\tau}] \Rightarrow \tau_j$
- n iterators $\mathsf{iter\_}\tau_j :: (\sigma_1 \Rightarrow \beta_1) \Rightarrow \cdots \Rightarrow (\sigma_n \Rightarrow \beta_n) \Rightarrow \tau_j \Rightarrow \beta_j$
- characteristic theorems including an induction rule

(The fixpoint variables β_j are harmlessly reused as result types of the iterators.) The contract for GFP, the greatest fixpoint, is identical except that coiterators and coinduction replace iterators and induction. The coiterator $\mathsf{coiter\_}\tau_j$ has type $(\beta_1 \Rightarrow \sigma_1) \Rightarrow \cdots \Rightarrow (\beta_n \Rightarrow \sigma_n) \Rightarrow \beta_j \Rightarrow \tau_j$. An iterator consumes a datatype, peeling off one constructor at a time; a coiterator produces a codatatype, delivering one constructor at a time.

LFP defines algebras and morphisms based on the equation system. The fixpoint, or initial algebra, is defined abstractly by well-founded recursion on a sufficiently large cardinal. In contrast, GFP builds a concrete tree structure. An abstract approach is also possible for GFP [26]; preliminary results indicate that it is simpler and more efficient.

The BNF approach to nesting scales much better than the old package's reduction to mutual recursion [29, Appendix B]. On the other hand, the LFP and GFP operations scale poorly in the number of mutual types; $n \approx 8$ is often the limit in practice. Reducing mutual recursion to nested recursion would circumvent the problem.

The ML functions that implement BNF operations all adhere to the same pattern: They introduce constants, state their properties, and discharge the proof obligations using dedicated tactics. About one fifth of the code base is devoted to tactics. They rely almost exclusively on resolution and unfolding, which makes them fast and reliable.

Methodologically, we developed the package gradually, starting with the formalization of a fixed abstract example $\beta = (\alpha, \beta, \gamma)\, F_0$ and $\gamma = (\alpha, \beta, \gamma)\, G_0$ specifying $\alpha\, F$ and $\alpha\, G$. We axiomatized the BNF structure and verified the closure under LFP and GFP using structured Isar proofs. We then expanded the proofs to detailed **apply** scripts [8]. Finally, we translated the scripts into tactics and generalized them for arbitrary m and n.

The Composition Pipeline. Composing functors together is widely perceived as being trivial, and accordingly it has received little attention in the literature, including our previous paper [28]. Nevertheless, an implementation must perform a carefully orchestrated sequence of steps to construct BNFs for the types occurring on the right-hand sides of fixpoint equations. This is achieved by four operations:

COMPOSE : m-ary BNF $\sigma \langle \overline{\alpha} \,|\, \Delta \rangle$ and m n-ary BNFs $\tau_i \langle \overline{\beta} \,|\, \Theta_i \rangle \rightarrow$
n-ary BNF $\sigma[\overline{\alpha} \mapsto \overline{\tau}] \langle \overline{\beta} \,|\, \Delta \cup \Theta_1 \cup \cdots \cup \Theta_m \rangle$

KILL : m-ary BNF $\sigma \langle \overline{\alpha} \,|\, \Delta \rangle$ and $k \leq m \rightarrow$
$(m-k)$-ary BNF $\sigma \langle \alpha_{k+1}, \ldots, \alpha_m \,|\, \Delta \cup \{\alpha_1, \ldots, \alpha_k\} \rangle$

LIFT : m-ary BNF $\sigma \langle \overline{\alpha} \,|\, \Delta \rangle$ and n type variables $\overline{\beta} \rightarrow$
$(m+n)$-ary BNF $\sigma \langle \overline{\beta}, \overline{\alpha} \,|\, \Delta \rangle$

PERMUTE : m-ary BNF $\sigma \langle \overline{\alpha} \,|\, \Delta \rangle$ and permutation π of $\{1, \ldots, m\} \rightarrow$
m-ary BNF $\sigma \langle \alpha_{\pi(1)}, \ldots, \alpha_{\pi(m)} \,|\, \Delta \rangle$

COMPOSE operates on BNFs normalized to share the same live variables; the other operations perform this normalization. Complex types are proved to be BNFs by applying normalization followed by COMPOSE recursively. The base cases are manually registered as BNFs. These include constant $\alpha\,\langle|\,\alpha\rangle$, identity $\alpha\,\langle\alpha\,|\rangle$, sum $\alpha+\beta\,\langle\alpha,\beta\,|\rangle$, product $\alpha\times\beta\,\langle\alpha,\beta\,|\rangle$, and restricted function space $\alpha\Rightarrow\beta\,\langle\beta\,|\,\alpha\rangle$. Users can register further types, such as those introduced by the new package for non-free datatypes [27].

As an example, consider the type $(\alpha\Rightarrow\beta)+\gamma\times\alpha$. The recursive calls on the arguments to $+$ return two BNFs, $\alpha\Rightarrow\beta\,\langle\beta\,|\,\alpha\rangle$ and $\gamma\times\alpha\,\langle\gamma,\alpha\,|\rangle$. Since α is dead in $\alpha\Rightarrow\beta$, it must be killed in $\gamma\times\alpha$ as well. This is achieved by permuting α to be the first variable and killing it, yielding $\gamma\times\alpha\,\langle\gamma\,|\,\alpha\rangle$. Next, both BNFs are lifted to have the same set of live variables: $\alpha\Rightarrow\beta\,\langle\gamma,\beta\,|\,\alpha\rangle$ and $\gamma\times\alpha\,\langle\beta,\gamma\,|\,\alpha\rangle$. Another permutation ensures that the live variables appear in the same order: $\alpha\Rightarrow\beta\,\langle\beta,\gamma\,|\,\alpha\rangle$. At this point, the BNF for $+$ can be composed with the normalized BNFs to produce $(\alpha\Rightarrow\beta)+\gamma\times\alpha\,\langle\beta,\gamma\,|\,\alpha\rangle$.

The compositional approach to BNF construction keeps the tactics simple at the expense of performance. Composition initially took seconds even for simple examples. By inlining intermediate definitions and deriving auxiliary BNF facts lazily, we were able to address the main bottlenecks. Nevertheless, Brian Huffman has demonstrated in a private prototype that the monolithic approach is feasible and less heavy.

Nested-to-Mutual Reduction. The old **datatype** command reduces nested recursion to mutual recursion, as proposed by Gunter [11]. Given a nested datatype specification such as $\alpha\ \mathit{tree} = \mathsf{Tree}\,\alpha\,(\alpha\ \mathit{tree}\ \mathit{list})$, the old command first unfolds the definition of *list*, resulting in the mutual specification of trees and "lists of trees," as if the user had entered

datatype $\alpha\ \mathit{tree} = \mathsf{Tree}\,\alpha\,(\alpha\ \mathit{treelist})$ **and** $\alpha\ \mathit{treelist} = \mathsf{Nil} \mid \mathsf{Cons}\,(\alpha\ \mathit{tree})\,(\alpha\ \mathit{treelist})$

In a second step, the package translates all occurrences of $\alpha\ \mathit{treelist}$ into the more palatable $\alpha\ \mathit{tree}\ \mathit{list}$ via an isomorphism. As a result, the induction principle and the input syntax to **primrec** have an unmistakable mutual flavor.

For compatibility, and for the benefit of users who prefer the mutual approach, the new package also implements a nested-to-mutual reduction operation, N2M, that constructs old-style induction principles and iterators from new-style ones:

N2M : n $(m+n)$-ary BNFs $\sigma_j\,\langle\overline{\alpha},\overline{\beta}\,|\,\Delta_j\rangle$ and n datatypes τ_j $\rightarrow$
- n iterators $\mathsf{n2m\_iter\_}\tau_j :: (\sigma_1\Rightarrow\beta_1)\Rightarrow\cdots\Rightarrow(\sigma_n\Rightarrow\beta_n)\Rightarrow\tau_j\Rightarrow\beta_j$
- characteristic theorems including an induction rule

Like LFP and GFP, the N2M operation takes a system of equations $\beta_j=\sigma_j$ given as normalized BNFs. In addition, it expects a list of datatypes τ_j that solve the equations and that may nest each other (e.g., $\alpha\ \mathit{tree}$ and $\alpha\ \mathit{tree}\ \mathit{list}$). The operation is dual for codatatypes. Its implementation is a single ML function that reverses some of the function and implication arrows when operating on codatatypes.

The **primrec** and **primcorec** commands invoke N2M when they detect nested (co)datatypes used as if they were mutual. In addition, **datatype_compat** relies on N2M to register new-style nested datatypes as old-style datatypes, which is useful for interfacing with existing unported infrastructure. In contrast to Gunter's approach, N2M does not introduce any new types. Instead, it efficiently composes existing artifacts: (co)iterators and (co)induction rules. The (admittedly technical) description below is to our knowledge the first account of such an operation.

As an abstract example that captures most of the complexity of N2M, let $\alpha\ F$ be the LFP of $\beta = (\alpha, \beta)\, F_0$ and $\alpha\ G$ be the LFP of $\beta = (\alpha, \beta\, F)\, G_0$.[2] These two definitions reflect the modular, nested view: First $\alpha\ F$ is defined as an LFP, becoming a BNF in its own right; then $\alpha\ G$ is defined as an LFP using an equation that nests F. The resulting iterator for $\alpha\ G$, iter_G, has type $((\alpha, \gamma\, F)\, G_0 \Rightarrow \gamma) \Rightarrow \alpha\ G \Rightarrow \gamma$, and its characteristic equation recurses through the F components of G using map_F.

On the other hand, if we defined $\alpha\ G_{\mathsf{M}}$ ($\simeq \alpha\ G$) and $\alpha\ GF_{\mathsf{M}}$ ($\simeq \alpha\ G\ F$) together, in the old-style mutually recursive fashion, as the LFP of $\beta = (\alpha, \gamma)\, G_0$ and $\gamma = (\beta, \gamma)\, F_0$, we would obtain two iterators with the following types:

$$\mathsf{iter\_G_M} :: ((\alpha, \gamma)\, G_0 \Rightarrow \beta) \Rightarrow ((\beta, \gamma)\, F_0 \Rightarrow \gamma) \Rightarrow \alpha\ G_{\mathsf{M}} \Rightarrow \beta$$
$$\mathsf{iter\_GF_M} :: ((\alpha, \gamma)\, G_0 \Rightarrow \beta) \Rightarrow ((\beta, \gamma)\, F_0 \Rightarrow \gamma) \Rightarrow \alpha\ GF_{\mathsf{M}} \Rightarrow \gamma$$

These are more flexible: $\mathsf{iter\_GF_M}$ offers the choice of indicating recursive behavior other than a map for the $\alpha\ GF_{\mathsf{M}}$ components of $\alpha\ G_{\mathsf{M}}$. The gap is filled by N2M, which defines "mutualized" iterators by combining the standard iterators for F and G. It does not introduce any new types $\alpha\ G_{\mathsf{M}}$ and $\alpha\ GF_{\mathsf{M}}$ but works with the existing ones:

$$\mathsf{n2m\_iter\_G} :: ((\alpha, \gamma)\, G_0 \Rightarrow \beta) \Rightarrow ((\beta, \gamma)\, F_0 \Rightarrow \gamma) \Rightarrow \alpha\ G \Rightarrow \beta$$
$$\mathsf{n2m\_iter\_G\_F} :: ((\alpha, \gamma)\, G_0 \Rightarrow \beta) \Rightarrow ((\beta, \gamma)\, F_0 \Rightarrow \gamma) \Rightarrow \alpha\ G\ F \Rightarrow \gamma$$
$$\mathsf{n2m\_iter\_G}\ \ g\ f = \mathsf{iter\_G}\ (g \circ \mathsf{map\_G_0}\ \mathsf{id}\ (\mathsf{iter\_F}\ f))$$
$$\mathsf{n2m\_iter\_G\_F}\ g\ f = \mathsf{iter\_F}\ (f \circ \mathsf{map\_F_0}\ (\mathsf{n2m\_iter\_G}\ g\ f)\ \mathsf{id})$$

N2M also outputs an induction principle corresponding to mutualized recursion. The operation first derives the low-level relator induction rule—a higher-order version of parallel induction on two values of the same shape (e.g., lists of the same length)—for each input BNF. The relator induction rules for $\alpha\ G$ and $\alpha\ G\ F$ are as follows:

$$\frac{\bigwedge x\ x'.\ \mathsf{rel\_G_0}\ P\ (\mathsf{rel\_F}\ R)\ x\ x' \Longrightarrow R\ (\mathsf{ctor\_G}\ x)\ (\mathsf{ctor\_G}\ x')}{\mathsf{rel\_G}\ P \sqsubseteq R}$$

$$\frac{\bigwedge y\ y'.\ \mathsf{rel\_F_0}\ R\ S\ y\ y' \Longrightarrow S\ (\mathsf{ctor\_F}\ y)\ (\mathsf{ctor\_F}\ y')}{\mathsf{rel\_F}\ R \sqsubseteq S}$$

The relators are compositional, enabling a modular proof of the mutualized relator induction from the relator inductions for G and F and relator monotonicity of G_0 and F_0:

$$\frac{\begin{array}{l}\bigwedge x\ x'.\ \mathsf{rel\_G_0}\ P\ S\ x\ x' \Longrightarrow R\ (\mathsf{ctor\_G}\ x)\ (\mathsf{ctor\_G}\ x') \\ \bigwedge y\ y'.\ \mathsf{rel\_F_0}\ R\ S\ y\ y' \Longrightarrow S\ (\mathsf{ctor\_F}\ y)\ (\mathsf{ctor\_F}\ y')\end{array}}{\mathsf{rel\_G}\ P \sqsubseteq R \wedge \mathsf{rel\_F}\ (\mathsf{rel\_G}\ P) \sqsubseteq S}$$

The standard induction rule is derived by instantiating $P :: \alpha \Rightarrow \alpha' \Rightarrow \mathit{bool}$ with equality, followed by some massaging. Coinduction is dual, with $\Longrightarrow$ and $\sqsubseteq$ reversed.

3 Types with Free Constructors

Datatypes and codatatypes are instances of types equipped with free constructors. Such types are interesting in their own right, irrespective of whether they support (co)induction; for example, pattern matching requires only distinctness and injectivity.

[2] It may help to think of these types more concretely by taking

$$F := \mathit{list} \qquad G := \mathit{tree} \qquad (\alpha, \beta)\, F_0 := \mathit{unit} + \alpha \times \beta \qquad (\alpha, \beta)\, G_0 := \alpha \times \beta$$

We have extended Isabelle with a database of freely constructed types. Users can enter the **wrap_free_constructors** command to register custom types, by listing the constructors and proving exhaustiveness, distinctness, and injectivity. In exchange, Isabelle generates constants for case expressions, discriminators, and selectors—collectively called *destructors*—as well as a wealth of theorems about constructors and destructors. The **datatype** and **codatatype** commands use this functionality internally.

The case constant is defined via the definite description operator (ι)—for example, case_list $n\ c\ xs = (\iota z.\ xs = \mathsf{Nil} \wedge z = n \vee (\exists y\ ys.\ xs = \mathsf{Cons}\ y\ ys \wedge z = c\ y\ ys))$. Syntax translations render case_list $n\ c\ xs$ as an ML-style case expression.

Given a type τ constructed by $\mathsf{C}_1, \ldots, \mathsf{C}_m$, its *discriminators* are constants $\mathsf{is\_C}_1, \ldots, \mathsf{is\_C}_m :: \tau \Rightarrow \mathit{bool}$ such that $\mathsf{is\_C}_i\ (\mathsf{C}_j\ \bar{x})$ if and only if $i = j$. No discriminators are needed if $m = 1$. For the $m = 2$ case, Isabelle generates a single discriminator and uses its negation for the second constructor by default. For nullary constructors C_i, Isabelle can use $\lambda x.\ x = \mathsf{C}_i$ as the discriminator. In addition, for each n-ary constructor $\mathsf{C}_i :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau$, n *selectors* $\mathsf{un\_C}_{ij} :: \tau \Rightarrow \tau_j$ extract its arguments. Users can reuse selector names across constructors. They can also specify a default value for constructors on which a selector would otherwise be unspecified.

The example below defines four selectors and assigns reasonable default values. The mid selector returns the third argument of Node2 $x\ l\ r$ as a default value:

datatype $\alpha\ \mathit{tree}_{23} =$
  Leaf (**defaults** left: Leaf mid: Leaf right: Leaf)
| Node2 (val: α) (left: $\alpha\ \mathit{tree}_{23}$) (right: $\alpha\ \mathit{tree}_{23}$) (**defaults** mid: $\lambda x\ l\ r.\ r$)
| Node3 (val: α) (left: $\alpha\ \mathit{tree}_{23}$) (mid: $\alpha\ \mathit{tree}_{23}$) (right: $\alpha\ \mathit{tree}_{23}$)

4 (Co)datatypes

The **datatype** and **codatatype** commands share the same input syntax, consisting of a list of mutually (co)recursive types to define, their desired constructors, and optional information such as custom names for destructors. They perform the following steps:

1. Formulate and solve the fixpoint equations using LFP or GFP.
2. Define the constructor constants.
3. Generate the destructors and the free constructor theorems.
4. Derive the high-level map, set, and relator theorems.
5. Define the high-level (co)recursor constants.
6. Derive the high-level (co)recursor theorems and (co)induction rules.

Step 1 relies on the fixpoint and composition operations described in Section 2 to produce the desired types and low-level constants and theorems. Step 2 defines high-level constructors that untangle sums of products—for example, $\mathsf{Nil} = \mathsf{ctor\_list}\ (\mathsf{Inl}\ ())$ and $\mathsf{Cons}\ x\ xs = \mathsf{ctor\_list}\ (\mathsf{Inr}\ (x, xs))$. Step 3 amounts to an invocation of **wrap_free_constructors**, described in Section 3. Step 4 reformulates the low-level map, set, and relator theorems in terms of constructors; a selection is shown for $\alpha\ \mathit{list}$ below:

list.map:	$\mathsf{map}\ f\ \mathsf{Nil} = \mathsf{Nil}$	$\mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$
list.set:	$\mathsf{set}\ \mathsf{Nil} = \{\}$	$\mathsf{set}\ (\mathsf{Cons}\ x\ xs) = \{x\} \cup \mathsf{set}\ xs$
list.rel_inject:	$\mathsf{rel}\ R\ \mathsf{Nil}\ \mathsf{Nil}$	$\mathsf{rel}\ R\ (\mathsf{Cons}\ x\ xs)\ (\mathsf{Cons}\ y\ ys) \longleftrightarrow R\ x\ y \wedge \mathsf{rel}\ R\ xs\ ys$

Datatypes and codatatypes differ at step 5. For an m-constructor datatype, the high-level iterator takes m curried functions as arguments (whereas the low-level version takes one function with a sum-of-product domain). For convenience, a recursor is defined in terms of the iterator to provide each recursive constructor argument's value both before and after the recursion. The list recursor has type $\beta \Rightarrow (\alpha \Rightarrow \alpha\ list \Rightarrow \beta \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta$. The corresponding induction rule has one hypothesis per constructor:

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

list.induct: $\quad \dfrac{P\ \mathsf{Nil} \quad \bigwedge x\ xs.\ P\ xs \Longrightarrow P\ (\mathsf{Cons}\ x\ xs)}{P\ t}$

For nested recursion beyond sums of products, the map and set functions of the type constructors through which recursion takes place appear in the high-level theorems:

tree$_\omega$.rec: $\quad \mathsf{rec\_tree}_\omega\ f\ (\mathsf{Tree}_\omega\ x\ ts) = f\ x\ (\mathsf{lmap}\ (\lambda t.\ (t, \mathsf{rec\_tree}_\omega\ f\ t))\ ts)$

tree$_\omega$.induct: $\quad \dfrac{\bigwedge x\ ts.\ (\bigwedge t.\ t \in \mathsf{lset}\ ts \Longrightarrow P\ t) \Longrightarrow P\ (\mathsf{Tree}_\omega\ x\ ts)}{P\ t}$

As for corecursion, for an m-constructor codatatype, $m-1$ predicates sequentially determine which constructor to produce. Moreover, for each constructor argument, a function specifies how to construct it from an abstract value of type α. For corecursive arguments, the function has type $\alpha \Rightarrow \tau + \alpha$ and returns either a value that stops the corecursion or a tuple of arguments to a corecursive call. At the high level, such functions are presented as three arguments *stop* $:: \alpha \Rightarrow bool$, *end* $:: \alpha \Rightarrow \tau$, and *continue* $:: \alpha \Rightarrow \alpha$, abbreviated to s, e, c below. For example, the high-level corecursor for lazy lists has the type $(\alpha \Rightarrow bool) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow bool) \Rightarrow (\alpha \Rightarrow \beta\ llist) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \beta\ llist$:

llist.corec: $\quad n\ a \Longrightarrow \mathsf{corec\_llist}\ n\ h\ s\ e\ c\ a = \mathsf{LNil}$

$\neg\ n\ a \Longrightarrow \mathsf{corec\_llist}\ n\ h\ s\ e\ c\ a =$

$\qquad \mathsf{LCons}\ (h\ a)\ (\mathsf{if}\ s\ a\ \mathsf{then}\ e\ a\ \mathsf{else}\ \mathsf{corec\_llist}\ n\ h\ s\ e\ c\ (c\ a))$

Nested corecursion is expressed using the map functions of the nesting type constructors. The coinduction rule uses the relators to lift a coinduction witness R. For example:

ltree.corec: $\quad \mathsf{corec\_ltree}\ l\ s\ a =$

$\mathsf{LTree}\ (l\ a)\ (\mathsf{map}\ (\mathsf{case\_sum}\ (\lambda t.\ t)\ (\mathsf{corec\_ltree}\ l\ s))\ (s\ a))$

ltree.coinduct: $\quad \dfrac{R\ t\ u \quad \bigwedge t\ u.\ R\ t\ u \Longrightarrow \mathsf{llab}\ t = \mathsf{llab}\ u \wedge \mathsf{rel}\ R\ (\mathsf{lsub}\ t)\ (\mathsf{lsub}\ u)}{t = u}$

5 Recursive Functions

Primitive recursive functions can be defined by providing suitable arguments to the recursors. The **primrec** command automates this process: From recursive equations specified by the user, it synthesizes a recursor-based definition.

The main advantage of the new implementation of **primrec** over the old one is its support for nested recursion through map functions [23]. For example:

primrec $\mathsf{height\_tree}_{\mathsf{FS}} :: \alpha\ tree_{\mathsf{FS}} \Rightarrow nat$ **where**

$\mathsf{height\_tree}_{\mathsf{FS}}\ (\mathsf{Tree}_{\mathsf{FS}}\ \_\ T) = 1 + \bigsqcup \mathsf{fset}\ (\mathsf{fimage}\ \mathsf{height\_tree}_{\mathsf{FS}}\ T)$

In the above, $\alpha\ tree_{\mathsf{FS}}$ is the datatype constructed by $\mathsf{Tree}_{\mathsf{FS}} :: \alpha \Rightarrow \alpha\ tree_{\mathsf{FS}}\ fset \Rightarrow \alpha\ tree_{\mathsf{FS}}$ (Section 1), $\bigsqcup N$ stands for the maximum of N, fset injects $\alpha\ fset$ into $\alpha\ set$, and the map function fimage gives the image of a finite set under a function. From the specified equation, the command synthesizes the definition

$$\mathsf{height\_tree}_{\mathsf{FS}} = \mathsf{rec\_tree}_{\mathsf{FS}}\ (\lambda\_\ TN.\ 1 + \bigsqcup \mathsf{fset}\ (\mathsf{fimage\ snd}\ TN))$$

From this definition and the $tree_{\mathsf{FS}}.rec$ theorems, it derives the original specification as a theorem. Notice how the argument $T :: \alpha\ tree_{\mathsf{FS}}\ fset$ becomes $TN :: (\alpha\ tree_{\mathsf{FS}} \times nat)\ fset$, where the second pair component stores the result of the corresponding recursive call.

Briefly, constructor arguments x are transformed as follows. Nonrecursive arguments appear unchanged in the recursor and can be used directly. Directly or mutually recursive arguments appear as two values: the original value x and the value y after the recursive call to f. Calls f x are replaced by y. Nested recursive arguments appear as a single argument but with pairs inside the nesting type constructors. The syntactic transformation must follow the map functions and eventually apply fst or snd, depending on whether a recursive call takes place. Naked occurrences of x without map are replaced by a suitable "map fst" term; for example, if the constant 1 were changed to fcard T in the specification above, the definition would have fcard (fimage fst TN) in its place.

The implemented procedure is somewhat more complicated. The recursor generally defines functions of type $\alpha\ tree_{\mathsf{FS}} \Rightarrow \beta$, but **primrec** needs to process n-ary functions that recurse on their jth argument. This is handled internally by moving the jth argument to the front and by instantiating β with an $(n-1)$-ary function type.

For recursion through functions, the map function is function composition ($\circ$). Instead of $f \circ g$, **primrec** also allows the convenient (and backward compatible) syntax $\lambda x.\ f\ (g\ x)$. More generally, $\lambda x_1 \ldots x_n.\ f\ (g\ x_1 \ldots x_n)$ expands to $(\mathrm{op} \circ (\ldots\ (\mathrm{op} \circ f) \ldots))\ g$.

Thanks to the N2M operation described in Section 2, users can also define mutually recursive functions on nested datatypes, as they would have done with the old package:

primrec height_tree :: $\alpha\ tree \Rightarrow nat$ **and** height_forest :: $\alpha\ tree\ list \Rightarrow nat$
where
  height_tree (Tree _ ts) = 1 + height_forest ts
| height_forest Nil = 0
| height_forest (Cons t ts) = height_tree t ⊔ height_forest ts

Internally, the following steps are performed:

1. Formulate and solve the fixpoint equations using N2M.
2. Define the high-level (co)recursor constants.
3. Derive the high-level (co)recursor theorems and (co)induction rules.

Step 1 produces low-level constants and theorems. Steps 2 and 3 are performed by the same machinery as when declaring mutually recursive datatypes (Section 4).

6 Corecursive Functions

The **primcorec** command is the main mechanism to introduce functions that produce potentially infinite codatatype values [23]. The command supports three competing syntaxes, or *views*: *destructor*, *constructor*, and *code*. Irrespective of the view chosen for input, the command generates the characteristic theorems associated with all three views.

The Destructor View. The coinduction literature tends to favor the destructor view, perhaps because it best reflects the duality between datatypes and codatatypes [1, 16]. The append function on lazy lists will serve as illustration:

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

The first formula, called the *discriminator formula*, gives the condition on which LNil should be produced. For an m-constructor datatype, up to m discriminator formulas can be given. If exactly $m-1$ formulas are stated (as in the example above), the last one is implicitly understood, with as its condition the complement of the other conditions.

The last two formulas, the *selector equations*, describe the behavior of the function when an LCons is produced. They are implicitly conditional on $\neg$ lnull *xs* $\vee$ $\neg$ lnull *ys*. The right-hand sides consist of 'let', 'if', or 'case' expressions whose leaves are either corecursive calls or arbitrary non-corecursive terms. This restriction ensures that the definition qualifies as primitive corecursive. The selector patterns on the left ensure that the function is productive and hence admissible [16].

With nesting, the corecursive calls appear under a map function, in much the same way as for **primrec**. Intuitive λ syntaxes for corecursion via functions are supported. The nested-to-mutual reduction is available for corecursion through codatatypes.

Proof obligations are emitted to ensure that the conditions are mutually exclusive. These are normally given to *auto* but can also be proved manually. Alternatively, users can specify the **sequential** option to have the conditions apply in sequence.

The conditions need not be exhaustive, in which case the function's behavior is left underspecified. If the conditions are syntactically detected to be exhaustive, or if the user enables the **exhaustive** option and discharges its proof obligation, the package generates stronger theorems—notably, discriminator formulas with $\longleftrightarrow$ instead of $\Longrightarrow$.

The Constructor View. The constructor view can be thought of as an abbreviation for the destructor view. It involves a single conditional equation per constructor:

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

The wildcard _ stands for the complement of the previous conditions.

The constructor view is convenient as input and sometimes for reasoning, but the equations are generally not suitable as simplification rules since they can loop. Compare this with the discriminator formulas and the selector equations of the destructor view, which can be safely registered as simplification rules.

The Code View. The code view is a variant of the constructor view in which the conditions are expressed using 'if' and 'case' expressions. Its primary purpose is for interfacing with Isabelle's code generator, which cannot cope with conditional equations.

The code view that **primcorec** generates from a destructor or constructor view is simply an equation that tests the conditions sequentially using 'if':

$\mathsf{lapp}\ xs\ ys = (\text{if } \mathsf{lnull}\ xs \wedge \mathsf{lnull}\ ys \text{ then } \mathsf{LNil}$
$\text{else } \mathsf{LCons}\,(\mathsf{lhd}\,(\text{if } \mathsf{lnull}\ xs \text{ then } ys \text{ else } xs))\,(\text{if } \mathsf{lnull}\ xs \text{ then } \mathsf{ltl}\ ys \text{ else } \mathsf{lapp}\,(\mathsf{ltl}\ xs)\ ys))$

If the cases are not known to be exhaustive, an additional 'if' branch ensures that the generated code throws an exception when none of the conditions are met.

The code view has a further purpose besides code generation: It provides a more flexible input format, with nested 'let', 'if', and 'case' expressions outside the constructors, multiple occurrences of the same constructors, and non-corecursive branches without constructor guards. This makes the code view the natural choice for append:

primcorec lapp :: α *llist* $\Rightarrow$ α *llist* $\Rightarrow$ α *llist* **where**
$\mathsf{lapp}\ xs\ ys = (\text{case } xs \text{ of } \mathsf{LNil} \Rightarrow ys \mid \mathsf{LCons}\ x\ xs \Rightarrow \mathsf{LCons}\ x\ (\mathsf{lapp}\ xs\ ys))$

The package reduces this specification to the following constructor view:

$\mathsf{lnull}\ xs \Longrightarrow \mathsf{lnull}\ ys \Longrightarrow \mathsf{lapp}\ xs\ ys = \mathsf{LNil}$
$\_ \Longrightarrow \mathsf{lapp}\ xs\ ys = \mathsf{LCons}\ (\text{case } xs \text{ of } \mathsf{LNil} \Rightarrow \mathsf{lhd}\ ys \mid \mathsf{LCons}\ x\ \_ \Rightarrow x)$
$(\text{case } xs \text{ of } \mathsf{LNil} \Rightarrow \mathsf{ltl}\ ys \mid \mathsf{LCons}\ \_\ xs \Rightarrow \mathsf{lapp}\ xs\ ys)$

In general, the reduction proceeds as follows:

1. Expand branches t of the code equation that are not guarded by a constructor to the term $(\text{case } t \text{ of } \mathsf{C}_1\ \bar{x}_1 \Rightarrow \mathsf{C}_1\ \bar{x}_1 \mid \cdots \mid \mathsf{C}_m\ \bar{x}_m \Rightarrow \mathsf{C}_m\ \bar{x}_m)$, yielding an equation χ.
2. Gather the conditions Φ_i associated with the branches guarded by C_i by traversing χ, with 'case' expressions recast as 'if's.
3. Generate the constructor equations $\bigvee\Phi_i\ \bar{x} \Longrightarrow \mathsf{f}\ \bar{x} = \mathsf{C}_i\ (\mathsf{un\_C}_{i1}\ \chi) \ldots (\mathsf{un\_C}_{ij}\ \chi)$, taking care of moving the $\mathsf{un\_C}_{ij}$'s under the conditionals and of simplifying them.

For the append example, step 1 expands the *ys* in the first 'case' branch to the term $(\text{case } ys \text{ of } \mathsf{LNil} \Rightarrow \mathsf{LNil} \mid \mathsf{LCons}\ y\ ys \Rightarrow \mathsf{LCons}\ y\ ys)$.

Finally, although **primcorec** does not allow pattern matching on the left-hand side, the **simps_of_case** command developed by Gerwin Klein and Lars Noschinski can be used to generate the pattern-matching equations from the code view—in our example, $\mathsf{lapp}\ \mathsf{LNil}\ ys = ys$ and $\mathsf{lapp}\ (\mathsf{LCons}\ x\ xs)\ ys = \mathsf{LCons}\ x\ (\mathsf{lapp}\ xs\ ys)$.

7 Coinduction Proof Method

The previous sections focused on the infrastructure for defining coinductive objects. Also important are the user-level proof methods, the building blocks of reasoning. The new method *coinduction* provides more automation over the existing *coinduct*, following a suggestion formulated in Lochbihler's Ph.D. thesis [20, Section 7.2]. The method handles arbitrary predicates equipped with suitable coinduction theorems. In particular, it can be used to prove equality of codatatypes by exhibiting a bisimulation.

A coinduction rule for a codatatype contains a free bisimulation relation variable R in its premises, which does not occur in the conclusion. The *coinduct* method crudely leaves R uninstantiated; the user is expected to provide the instantiation. However, the choice of the bisimulation is often canonical, as illustrated by the following proof:

lemma
assumes $\mathsf{infinite}\ (\mathsf{lset}\ xs)$
shows $\mathsf{lapp}\ xs\ ys = xs$

proof *coinduct*
  def $R = \lambda l\ r.\ \exists xs.\ l = \mathsf{lapp}\ xs\ ys \wedge r = xs \wedge \mathsf{infinite}\ (\mathsf{lset}\ xs)$
  with *assms* **show** $R\ (\mathsf{lapp}\ xs\ ys)\ xs$ **by** *auto*

  fix $l\ r$ **assume** $R\ l\ r$
  then obtain xs **where** $l = \mathsf{lapp}\ xs\ ys \wedge r = xs \wedge \mathsf{infinite}\ (\mathsf{lset}\ xs)$ **by** *auto*
  thus $\mathsf{lnull}\ l = \mathsf{lnull}\ r \wedge (\neg\ \mathsf{lnull}\ l \longrightarrow \neg\ \mathsf{lnull}\ r \longrightarrow \mathsf{lhd}\ l = \mathsf{lhd}\ r \wedge R\ (\mathsf{ltl}\ l)\ (\mathsf{ltl}\ r))$
    by *auto*
qed

The new method performs the steps highlighted in gray automatically, making a one-line proof possible: **by** (*coinduction arbitrary*: *xs*) *auto*.

In general, given a goal $P \Longrightarrow \mathsf{q}\ t_1 \ldots t_n$, the method selects the rule q.*coinduct* and takes $\lambda z_1 \ldots z_n.\ \exists x_1 \ldots x_m.\ z_1 = t_1 \wedge \cdots \wedge z_n = t_n \wedge P$ as the coinduction witness R. The variables x_i are those specified as being *arbitrary* and may freely appear in $P, t_1, \ldots, t_n$. After applying the instantiated rule, the method discharges the premise $R\ t_1 \ldots t_n$ by reflexivity and using the assumption P. Then it unpacks the existential quantifiers from R.

8 Example: Stream Processors

Stream processors were introduced by Hancock et al. [13] and have rapidly become the standard example for demonstrating mixed fixpoints in proof assistants [1–3, 9, 10]. Thanks to the new (co)datatype package, Isabelle finally joins this good company.

A stream processor represents a continuous transformation on streams—that is, a function of type $\alpha\ stream \Rightarrow \beta\ stream$ that consumes at most a finite prefix of the input stream before producing an element of output. The datatype sp_1 captures a single iteration of this process. The codatatype sp_ω nests sp_1 to produce an entire stream:

datatype $(\alpha, \beta, \delta)\ sp_1 = \mathsf{Get}\ (\alpha \Rightarrow (\alpha, \beta, \delta)\ sp_1) \mid \mathsf{Put}\ \beta\ \delta$
codatatype $(\alpha, \beta)\ sp_\omega = \mathsf{SP}\ (\mathsf{unSP}\colon (\alpha, \beta, (\alpha, \beta)\ sp_\omega)\ sp_1)$

Values of type sp_1 are finite-depth trees with inner nodes Get and leaf nodes Put. Each inner node has $|\alpha|$ children, one for each possible input α. The Put constructor carries the output element of type β and a continuation of type δ. The definition of sp_ω instantiates the continuation type to a stream processor $(\alpha, \beta)\ sp_\omega$.

In contrast to Isabelle, Agda supports the simultaneous mutual definition of sp_1 and sp_ω with annotations on constructor arguments indicating whether they are to be understood coinductively [10]. Least fixpoints are taken before greatest fixpoints, which is appropriate for this example. Isabelle's nested approach is arguably more explicit, more flexible, and more consistent with the literature.

The semantics of sp_ω is given by two functions: run_1 recurses on sp_1 (i.e., consumes an sp_1), and run_ω, corecurses on *stream* (i.e., produces a *stream*, defined in Section 2):

primrec $\mathsf{run}_1 :: (\alpha, \beta, \delta)\ sp_1 \Rightarrow \alpha\ stream \Rightarrow (\beta \times \delta) \times \alpha\ stream$ **where**
  $\mathsf{run}_1\ (\mathsf{Get}\ f)\quad s = \mathsf{run}_1\ (f\ (\mathsf{shd}\ s))\ (\mathsf{stl}\ s)$
  $\mathsf{run}_1\ (\mathsf{Put}\ x\ q)\ s = ((x, q), s)$

primcorec $\mathsf{run}_\omega :: (\alpha, \beta)\ sp_\omega \Rightarrow \alpha\ stream \Rightarrow \beta\ stream$ **where**
  $\mathsf{run}_\omega\ q\ s = (\mathsf{let}\ ((x, q'), s') = \mathsf{run}_1\ (\mathsf{unSP}\ q)\ s\ \mathsf{in}\ \mathsf{SCons}\ x\ (\mathsf{run}_\omega\ q'\ s'))$

These definitions illustrate some of the conveniences of **primrec** and **primcorec**. For run_1, the modular way to nest the recursive call of run_1 through functions would rely on composition—i.e., $(\mathsf{run}_1 \circ f)\ (\mathsf{shd}\ s)\ (\mathsf{stl}\ s)$. The **primrec** command allows us not only to expand the term $\mathsf{run}_1 \circ f$ to $\lambda x.\ \mathsf{run}_1\ (f\ x)$ but also to β-reduce it. For run_ω, the constructor view makes it possible to call run_1 only once, assign the result in a 'let', and use this result to specify both arguments of the produced constructor.

The stream processor copy outputs the input stream:

primcorec $\mathsf{copy} :: (\alpha, \alpha)\, sp_\omega$ **where** $\mathsf{copy} = \mathsf{SP}\ (\mathsf{Get}\ (\lambda a.\ \mathsf{Put}\ a\ \mathsf{copy}))$

The nested sp_1 value is built directly with corecursion under constructors as an alternative to the modular approach: $\mathsf{copy} = \mathsf{SP}\ (\mathsf{map\_sp}_1\ \mathsf{id}\ (\lambda\_.\ \mathsf{copy})\ (\mathsf{Get}\ (\lambda a.\ \mathsf{Put}\ a\ ())))$. The lemma $\mathsf{run}_\omega\ \mathsf{copy}\ s = s$ is easy to prove using *coinduction* and *auto*.

Since stream processors represent functions, it makes sense to compose them:

function $\circ_1 :: (\beta, \gamma, \delta)\, sp_1 \Rightarrow (\alpha, \beta, (\alpha, \beta)\, sp_\omega)\, sp_1 \Rightarrow (\alpha, \gamma, \delta \times (\alpha, \beta)\, sp_\omega)\, sp_1$ **where**
$\mathsf{Put}\ b\ q \circ_1 p \quad = \mathsf{Put}\ b\ (q, \mathsf{SP}\ p)$
$\mathsf{Get}\ f \circ_1 \mathsf{Put}\ b\ q = f\ b \circ_1 \mathsf{unSP}\ q$
$\mathsf{Get}\ f \circ_1 \mathsf{Get}\ g \quad = \mathsf{Get}\ (\lambda a.\ \mathsf{Get}\ f \circ_1 g\ a)$
by *pat_completeness auto*
termination by (*relation* lex_prod sub sub) *auto*

primcorec $\circ_\omega :: (\beta, \gamma)\, sp_\omega \Rightarrow (\alpha, \beta)\, sp_\omega \Rightarrow (\alpha, \gamma)\, sp_\omega$ **where**
$\mathsf{unSP}\ (q \circ_\omega q') = \mathsf{map\_sp}_1\ (\lambda b.\ b)\ (\lambda(q, q').\ q \circ_\omega q')\ (\mathsf{unSP}\ q \circ_1 \mathsf{unSP}\ q')$

The corecursion applies $\circ_\omega$ nested through the map function $\mathsf{map\_sp}_1$ to the result of finite preprocessing by the recursion $\circ_1$. The $\circ_1$ operator is defined using **function**, which emits proof obligations concerning pattern matching and termination.

9 Case Study: Porting the Coinductive Library

To evaluate the merits of the new definitional package, and to benefit from them, we have ported existing coinductive developments to the new approach. The Coinductive library [18] defines four codatatypes and related functions and comprises a large collection of lemmas. Originally, the codatatypes were manually defined as follows:

- extended naturals *enat* as **datatype** $enat = \mathsf{enat}\ nat \mid \infty$;
- lazy lists α *llist* using Paulson's construction [25];
- terminated lazy lists (α, β) *tllist* as the quotient of α *llist* $\times\, \beta$ over the equivalence relation that ignores the second component if and only if the first one is infinite;
- streams α *stream* as the subtype of infinite lazy lists.

Table 1 presents the types and the evaluation's statistics. The third column gives the lines of code for the definitions, lemmas, and proofs that were needed to define the type, the constructors, the corecursors, and the case constants, and to prove the free constructor theorems and the coinduction rule for equality. For *enat*, we kept the old definition because the datatype view is useful. Hence, we still derive the corecursor and the coinduction rules manually, but we generate the free constructor theorems with the **wrap_free_constructors** command (Section 3), saving 6 lines. In contrast, the other three types are now defined with **codatatype** in 33 lines instead of 774, among which 28 are for *tllist* because the default value for TNil's selector applied to TCons requires

Codatatype	Constructors	Lines of code for definition	Number of lemmas	Lines of code per lemma
enat	0 | eSuc *enat*	200 → 194	31 → 57	8.42 → 5.79
α *llist*	LNil | LCons α (α *llist*)	503 → 3	527 → 597	9.86 → 6.44
(α, β) *tllist*	TNil β | TCons α ((α, β) *tllist*)	169 → 28 + 120	121 → 200	6.05 → 4.95
α *stream*	SCons α (α *stream*)	102 → 2 + 96	64 → 159	3.11 → 3.47
Total		974 → 227 + 216	743 → 1013	8.60 → 5.65

Table 1. Statistics on porting Coinductive to the new package (before → after)

unbounded recursion. However, we lost the connection between *llist*, *tllist*, and *stream*, on which the function definitions and proofs relied. Therefore, we manually set up the lifting and transfer packages [15]; the line counts are shown behind plus signs (+).

The type definitions are just a small fraction of the library; most of the work went into proving properties of the functions. The fourth column shows the number of lemmas that we have proved for the functions on each type. There are 36% more than before, which might be surprising at first, since the old figures include the manual type constructions. Three reasons explain the increase. First, changes in the views increase the counts. Coinductive originally favored the code and constructor views, following Paulson [25], whereas the new package expresses coinduction and other properties in terms of the destructors (Sections 4 and 6). We proved additional lemmas for our functions that reflect the destructor view. Second, the manual setup for lifting and transfer accounts for 36 new lemmas. Third, the porting has been distributed over six months such that we continuously incorporated our insights into the package's implementation. During this period, the *stream* part of the library grew significantly and *tllist* a little.

Therefore, the absolute numbers should not be taken too seriously. It is more instructive to examine how the proofs have changed. The last column of Table 1 gives the average length of a lemma, including its statement and its proof; shorter proofs indicate better automation. Usually, the statement takes between one and four lines, where two is the most common case. The port drastically reduced the length of the proofs: We now prove 36% more lemmas in 11% fewer lines.

Two improvements led to these savings. First, the *coinduction* method massages the proof obligation to fit the coinduction rule. Second, automation for coinduction proofs works best with the destructor view, as the destructors trigger rewriting. With the code and constructor style, we formerly had to manually unfold the equations, and pattern-matching equations obtained by **simps_of_case** needed manual case distinctions.

The destructor view also has some drawbacks. The proofs rely more on Isabelle's classical reasoner to solve subgoals that the simplifier can discharge in the other styles, and the reasoner often needs more guidance. We have not yet run into scalability issues, but we must supply a lengthy list of lemmas to the reasoner. The destructor style falls behind when we leave the coinductive world. For example, the recursive function lnth :: *nat* ⇒ α *llist* ⇒ α returns the element at a given index in a lazy list; clearly, there are no destructors on lnth's result type to trigger unfolding. Since induction proofs introduce constructors in the arguments, rewriting with pattern-matching equations obtained from the code view yields better automation. In summary, all three views are useful.

10 Conclusion

Codatatypes and corecursion have long been missing features in proof assistants based on higher-order logic. Isabelle's new (co)datatype definitional package finally addresses this deficiency, while generalizing and modularizing the support for datatypes. The package is already highly usable and is used not only for the Coinductive library but also in various ongoing developments by the authors. Although Isabelle is our vehicle, the approach is equally applicable to the other provers from the HOL family.

For future work, our priority is to integrate the package better with other Isabelle subsystems, including **fun** (for well-founded recursive definitions), lifting and transfer, and the counterexample generators. Another straightforward development would be to have the package produce even more theorems, notably for parametricity. There is also work in progress on supporting more general forms of corecursion and mixed recursion–corecursion. Finally, we expect that BNFs can be generalized to support non-free datatypes, including nominal types, but this remains to be investigated.

Acknowledgment. Tobias Nipkow and Makarius Wenzel encouraged us to implement the new package. Florian Haftmann and Christian Urban provided general advice on Isabelle and package writing. Brian Huffman suggested major simplifications to the internal constructions, many of which have yet to be implemented. Stefan Milius and Lutz Schröder found an elegant proof to eliminate one of the BNF assumptions. Lars Hupel and Mark Summerfield suggested many textual improvements.

Blanchette is supported by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant Ni 491/14-1). Hölzl is supported by the DFG project Verification of Probabilistic Models (grant Ni 491/15-1). Popescu is supported by the DFG project Security Type Systems and Deduction (grant Ni 491/13-2) as part of the program Reliably Secure Software Systems (RS^3, priority program 1496). Traytel is supported by the DFG program Program and Model Analysis (PUMA, doctorate program 1480). The authors are listed alphabetically.

References

1. Abel, A., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: Morrisett, G., Uustalu, T. (eds.) ICFP '13. pp. 185–196. ACM (2013)
2. Altenkirch, T., Danielsson, N.A., Löh, A., Oury, N.: ΠΣ: Dependent types without the sugar. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 40–55. Springer (2010)
3. Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: Morrisett, G., Uustalu, T. (eds.) ICFP '13. pp. 197–208. ACM (2013)
4. 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 (1999)
5. 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)
6. Blanchette, J.C., Popescu, A., Traytel, D.: Cardinals in Isabelle/HOL. `http://www21.in.tum.de/~blanchet/card.pdf` (2014), submitted to ITP 2014

7. Blanchette, J.C., Popescu, A., Traytel, D.: Witnessing (co)datatypes. http://www21.in.tum.de/~blanchet/wit.pdf (2014)
8. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Formalization accompanying this paper. http://www21.in.tum.de/~traytel/codata_impl.tar.gz
9. Capretta, V.: Wander types: A formalization of coinduction-recursion. Progress in Informatics (10), 47–64 (2013), shonan Meeting on Dependently Typed Programming
10. Danielsson, N.A., Altenkirch, T.: Subtyping, declaratively. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC. LNCS, vol. 6120, pp. 100–118. Springer (2010)
11. Gunter, E.L.: Why we can't have SML-style `datatype` declarations in HOL. In: Claesen, L.J.M., Gordon, M.J.C. (eds.) TPHOLs '92. IFIP Transactions, vol. A-20, pp. 561–568. North-Holland (1993)
12. Gunter, E.L.: A broader class of trees for recursive type definitions for HOL. In: Joyce, J.J., Seger, C.J.H. (eds.) HUG '93. LNCS, vol. 780, pp. 141–154. Springer (1994)
13. Hancock, P., Ghani, N., Pattinson, D.: Representations of stream processors using nested fixed points. Log. Meth. Comput. Sci. 5(3) (2009)
14. Harrison, J.: Inductive definitions: Automation and application. In: Schubert, E.T., Windley, P.J., Alves-Foss, J. (eds.) TPHOLs '95. LNCS, vol. 971, pp. 200–213. Springer (1995)
15. Huffman, B., Kunčar, O.: Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 131–146. Springer (2013)
16. Jacobs, B., Rutten, J.: A tutorial on (co)algebras and (co)induction. Bull. EATCS 62, 222–259 (1997)
17. Leroy, X.: A formally verified compiler back-end. J. Autom. Reas. 43(4), 363–446 (2009)
18. Lochbihler, A.: Coinductive. In: Klein, G., Nipkow, T., Paulson, L. (eds.) Archive of Formal Proofs. http://afp.sf.net/entries/Coinductive.shtml (2010)
19. Lochbihler, A.: Verifying a compiler for Java threads. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 427–447. Springer (2010)
20. Lochbihler, A.: A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler. Ph.D. thesis, Karlsruher Institut für Technologie (2012)
21. Lochbihler, A.: Making the Java memory model safe. ACM Trans. Program. Lang. Syst. 35(4), 12:1–65 (2014)
22. Melham, T.F.: Automating recursive type definitions in higher order logic. In: Birtwistle, G., Subrahmanyam, P.A. (eds.) Current Trends in Hardware Verification and Automated Theorem Proving, pp. 341–386. Springer (1989)
23. Panny, L.: Primitive (Co)recursive Function Definitions for Isabelle/HOL. B.Sc. thesis draft, Technische Universität München
24. Paulson, L.C.: A fixedpoint approach to implementing (co)inductive definitions. In: Bundy, A. (ed.) CADE-12. LNCS, vol. 814, pp. 148–161. Springer (1994)
25. Paulson, L.C.: Mechanizing coinduction and corecursion in higher-order logic. J. Log. Comput. 7(2), 175–204 (1997)
26. Rutten, J.J.M.M.: Universal coalgebra: A theory of systems. Theor. Comput. Sci. 249, 3–80 (2000)
27. Schropp, A., Popescu, A.: Nonfree datatypes in Isabelle/HOL: Animating a many-sorted metatheory. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 114–130. Springer (2013)
28. 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)
29. Traytel, D.: A Category Theory Based (Co)datatype Package for Isabelle/HOL. M.Sc. thesis, Technische Universität München (2012)