Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic

Julian Biendarra[1], Jasmin Christian Blanchette[2,3(✉)], Aymeric Bouzy[4], Martin Desharnais[5], Mathias Fleury[3], Johannes Hölzl[6], Ondřej Kunčar[1], Andreas Lochbihler[7], Fabian Meier[8], Lorenz Panny[9], Andrei Popescu[10,11], Christian Sternagel[12], René Thiemann[12], and Dmitriy Traytel[7]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Vrije Universiteit Amsterdam, The Netherlands
j.c.blanchette@vu.nl
[3] Max-Planck-Institut für Informatik, Saarland Informatics Campus, Saarbrücken, Germany
[4] InstantJob, Paris, France
[5] Ludwig-Maximilians-Universität München, Germany
[6] Carnegie Mellon University, Pittsburgh, USA
[7] Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
[8] Google, Zurich, Switzerland
[9] Technische Universiteit Eindhoven, The Netherlands
[10] Middlesex University London, UK
[11] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania
[12] Universität Innsbruck, Austria

Abstract. We describe a line of work that started in 2011 towards enriching Isabelle/HOL's language with coinductive datatypes, which allow infinite values, and with a more expressive notion of inductive datatype than previously supported by any system based on higher-order logic. These (co)datatypes are complemented by definitional principles for (co)recursive functions and reasoning principles for (co)induction. In contrast with other systems offering codatatypes, no additional axioms or logic extensions are necessary with our approach.

1 Introduction

Rich specification mechanisms are crucial to the usability of proof assistants—in particular, mechanisms for defining inductive datatypes, recursive functions, and inductive predicates. Datatypes and recursive functions are inspired by typed functional programming languages from the ML family. Inductive predicates are reminiscent of Prolog.

Coinductive methods are becoming increasingly widespread in computer science. Coinductive datatypes, corecursive functions, and coinductive predicates are useful to represent potentially infinite data and processes and to reason about them. Coinductive datatypes, or *codatatypes*, are freely generated by their constructors, but in contrast to datatypes, infinite constructor terms are also legitimate values for codatatypes. Corecursion makes it possible to build such values. A simple example is the "lazy" (or coinductive) list $\mathsf{LCons}\ 0\ (\mathsf{LCons}\ 1\ (\mathsf{LCons}\ 2\ \ldots))$ that enumerates the natural numbers. It can be specified via the corecursive equation $\mathsf{enum}\ n = \mathsf{LCons}\ n\ (\mathsf{enum}\ (n+1))$.

In 2011, we started an effort to enrich the Isabelle/HOL proof assistant with definitional mechanisms for codatatypes and corecursion. Until then, Isabelle/HOL and the

other main systems based on higher-order logic (HOL4, HOL Light, and ProofPower–HOL) provided at most (inductive) datatypes, recursive functions, and (co)inductive predicates. Our aim was to support formalizations such as Lochbihler's verified compiler for a Java-like language [32] and his mathematization of the Java memory model [33], both of which rely on codatatypes to represent infinite traces.

Creating a monolithic codatatype package to supplement Isabelle/HOL's existing datatype package [4] was not an attractive prospect, because many applications need to mix datatypes and codatatypes, as in the following nested (co)recursive specification:

datatype $\alpha\ \mathit{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha\ \mathit{list})$
codatatype $\alpha\ \mathit{ltree} = \mathsf{LNode}\ \alpha\ ((\alpha\ \mathit{ltree})\ \mathit{list})$

The first command introduces a polymorphic type of finite lists over an element type α, freely generated by the constructors $\mathsf{Nil} : \alpha\ \mathit{list}$ and $\mathsf{Cons} : \alpha \to \alpha\ \mathit{list} \to \alpha\ \mathit{list}$. The second command introduces a type of finitely branching trees of possibly infinite depth. For example, the infinite tree $\mathsf{LNode}\ 0\ (\mathsf{Cons}\ (\mathsf{LNode}\ 0\ (\mathsf{Cons}\ \ldots\ \mathsf{Nil}))\ \mathsf{Nil})$ specified by $t = \mathsf{LNode}\ 0\ (\mathsf{Cons}\ t\ \mathsf{Nil})$ is valid. Ideally, (co)datatypes should also be allowed to (co)recurse through well-behaved nonfree type constructors, such as the finite set constructor *fset*:

codatatype $\alpha\ \mathit{ltree}_{\mathsf{fs}} = \mathsf{LNode}_{\mathsf{fs}}\ \alpha\ ((\alpha\ \mathit{ltree}_{\mathsf{fs}})\ \mathit{fset})$

In this paper, we present the various new definitional packages for (co)datatypes and (co)recursive functions that today support Isabelle users with their formalizations. The theoretical cornerstone underlying these is a semantic criterion we call *bounded natural functors* (BNF, Sect. 3). The criterion is met by construction for a large class of datatypes and codatatypes (such as *list*, *ltree*, and $\mathit{ltree}_{\mathsf{fs}}$) and by bounded sets and bounded multisets. On the right-hand side of a **datatype** or **codatatype** command, recursion is allowed under arbitrary type constructors that are BNFs. This flexibility is not available in other proof assistants.

The **datatype** and **codatatype** commands construct a type as a solution to a fixpoint equation (Sect. 4). For example, $\alpha\ \mathit{list}$ is the solution for β in the equation $\beta \cong \mathit{unit} + \alpha \times \beta$, where *unit* is a singleton type, whereas $+$ and $\times$ are the type constructors for sum (disjoint union) and product (pairs), respectively. To ensure that the new types are nonempty, the commands must also synthesize a witness (Sect. 5).

The above mechanisms are complemented by commands for defining primitively (co)recursive functions over (co)datatypes (Sect. 6). But primitive (co)recursion is very restrictive in practice. For general (nonprimitive) well-founded recursion, Isabelle/HOL already provided the **fun** and **function** commands [29]; our new datatypes work well with them. For nonprimitive corecursion, we designed and implemented a definitional mechanism based on the notion of corecursion up to "friendly" operations (Sect. 7).

In *nonuniform* datatypes, the type arguments may vary recursively. They arise in the implementation of efficient functional data structures. We designed commands that reduce a large class of nonuniform datatypes, and nonuniform codatatypes, to their uniform counterparts (Sect. 8).

We integrated the new (co)datatypes with various Isabelle tools, including the Lifting and Transfer tools, which transfer definitions and theorems across isomorphisms, the Nitpick counterexample generator, and the Sledgehammer proof tool (Sect. 9). The

new (co)datatypes are widely used, including in our own work—codatatypes for their convenience, and the new datatypes for their flexibility and scalability (Sect. 10).

Crucially, all our specification mechanisms follow the *definitional approach*, as is typical in Isabelle/HOL and the HOL family of systems. This means that the desired types and terms are explicitly constructed and introduced using more primitive mechanisms and their characteristic properties are derived as theorems. This guarantees that they introduce no inconsistencies, reducing the amount of code that must be trusted. The main drawback of this approach is that it puts a heavy burden on the mechanisms' designers and implementers. For example, the (**co**)**datatype** commands explicitly construct solutions to fixpoint equations and nonemptiness witnesses, and the constructions must be performed efficiently. Other approaches—such as the intrinsic approach, where the specification mechanism is built directly into the logic, and the axiomatic approach, where types and terms are added to the signature and characterized by axioms—require less work but do not guard against inconsistencies [4, Sect. 1].

The work described in this paper was first presented in conference and journal publications between 2012 and 2017 [7,9,10,12,13,15–18,46,50,51]. The current text is partly based on these papers. The source code consists of about 29 000 lines of Standard ML distributed as part of Isabelle and the *Archive of Formal Proofs* [47]. It is complemented by Isabelle lemma libraries necessary for the constructions, notably a theory of cardinals [15]. We refer to our earlier papers [10, 13, 18, 51] for discussions of related work.

2 Isabelle/HOL

Isabelle [39] is a generic proof assistant whose metalogic is an intuitionistic fragment of polymorphic higher-order logic. The types τ are built from type variables $\alpha, \beta, \ldots$ and type constructors, written infix or postfix (e.g., $\to$, *list*). All types are inhabited. Terms t, u are built from variables x, constants c, abstractions $\lambda x.\, t$, and applications $t\, u$. Types are usually left implicit. Constants may be functions. A formula is a term of type *prop*. The metalogical operators are $\bigwedge$, $\Rightarrow$, and $\equiv$, for universal quantification, implication, and equality. The notation $\bigwedge x.\, t$ abbreviates $\bigwedge(\lambda x.\, t)$. Internally, λ is the only binder.

Isabelle/HOL is the instantiation of Isabelle with classical higher-order logic (HOL) extended with type classes as its object logic, complete with a Boolean type *bool*, an equality predicate ($=$), the usual connectives ($\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$) and quantifiers ($\forall$, $\exists$), and Hilbert's choice operator. HOL formulas, of type *bool*, are embedded in the metalogic. The distinction between *prop* and *bool* is not essential to understand this paper.

Isabelle/HOL offers two primitive definitional mechanisms: The **typedef** command introduces a type that is isomorphic to a nonempty subset of an existing type, and the **definition** command introduces a constant as equal to an existing term. Other commands, such as **datatype** and **function**, build on these primitives.

Proofs are expressed either as a sequence of low-level *tactics* that manipulate the proof state directly or in a declarative format called Isar [53]. Basic tactics rely on resolution and higher-order unification. Other useful tactics include the *simplifier*, which rewrites terms using conditional oriented equations, and the *classical reasoner*, which applies introduction and elimination rules in the style of natural deduction. Specialized tactics can be written in Standard ML, Isabelle's main implementation language.

3 Bounded Natural Functors

An n-ary *bounded natural functor* (BNF) [12, 51, 52] is an $(n+k)$-ary type constructor equipped with a map function (or functorial action), a relator, n set functions (natural transformations), and a cardinal bound that satisfy certain properties. For example, *list* is a unary BNF. Its relator $\mathsf{rel} : (\alpha \to \beta \to \mathit{bool}) \to \alpha\ \mathit{list} \to \beta\ \mathit{list} \to \mathit{bool}$ extends binary predicates over elements to binary predicates over parallel lists: $\mathsf{rel}\ R\ \mathit{xs}\ \mathit{ys}$ is true if and only if the lists xs and ys have the same length and the elements of the two lists are elementwise related by R. Moreover, the cardinal bound bd constrains the number of elements returned by the set function set; it cannot depend on α's cardinality. To prove that *list* is a BNF, the **datatype** command discharges the following proof obligations:

$$\mathsf{map}\ \mathsf{id} = \mathsf{id} \qquad \mathsf{map}\ (f \circ g) = \mathsf{map}\ f \circ \mathsf{map}\ g \qquad \frac{\bigwedge x.\ x \in \mathsf{set}\ \mathit{xs} \Longrightarrow f\ x = g\ x}{\mathsf{map}\ f\ \mathit{xs} = \mathsf{map}\ g\ \mathit{xs}}$$

$$|\mathsf{set}\ \mathit{xs}| \leq_{\mathsf{o}} \mathsf{bd} \qquad \mathsf{set} \circ \mathsf{map}\ f = \mathsf{image}\ f \circ \mathsf{set}$$

$$\aleph_0 \leq_{\mathsf{o}} \mathsf{bd} \qquad \mathsf{rel}\ R \circ\!\circ \mathsf{rel}\ S \sqsubseteq \mathsf{rel}\ (R \circ\!\circ S)$$

$$\mathsf{rel}\ R\ \mathit{xs}\ \mathit{ys} \longleftrightarrow \exists \mathit{ps}.\ \mathsf{set}\ \mathit{ps} \subseteq \{(\mathit{xs}, \mathit{ys}).\ R\ \mathit{xs}\ \mathit{ys}\} \wedge \mathsf{map}\ \mathsf{fst}\ \mathit{ps} = \mathit{xs} \wedge \mathsf{map}\ \mathsf{snd}\ \mathit{ps} = \mathit{ys}$$

The operator $\leq_{\mathsf{o}}$ is a well-order on ordinals [15], $\sqsubseteq$ denotes implication lifted to binary predicates, $\circ\!\circ$ denotes the relational composition of binary predicates, fst and snd denote the left and right pair projections, and the horizontal bar denotes implication ($\Longrightarrow$).

The class of BNFs is closed under composition, initial algebra (for datatypes), and final coalgebra (for codatatypes). The last two operations correspond to least and greatest fixpoints, respectively. Given an n-ary BNF, the n type variables associated with set functions, and on which the map function acts, are *live*; the remaining k type variables are *dead*. For example, the function type $\alpha \to \beta$ is a unary BNF on β; the variable α is dead. Nested (co)recursion can only take place through live variables.

Composition of functors is widely perceived as being trivial. Nevertheless, the implementation must perform a carefully orchestrated sequence of steps to construct BNFs and discharge the emerging proof obligations for the types occurring on the right-hand sides of fixpoint equations. This is achieved by four operations: Composition proper works on normalized BNFs that share the same live variables, whereas the other three operations achieve this normalization by adding, killing, or permuting live variables.

4 Datatypes and Codatatypes

The **datatype** and **codatatype** commands [12] state and solve fixpoint equations. Then they define the constructor, discriminator, and selector constants and derive various theorems involving the constructors. The command for introducing lazy lists follows:

codatatype $\alpha\ \mathit{llist} = \mathsf{lnull}\colon \mathsf{LNil} \mid \mathsf{LCons}\ (\mathsf{lhd}\colon \alpha)\ (\mathsf{ltl}\colon \alpha\ \mathit{list})$

The constructors are LNil and LCons. The discriminator lnull tests whether a lazy list is LNil. The selectors lhd and ltl return the head or tail of a non-LNil lazy list.

The **datatype** command also introduces a *recursor*, which can be used to define primitively recursive functions. The list recursor has type $\beta \to (\alpha \to \alpha\ \mathit{list} \times \beta \to \beta) \to \alpha\ \mathit{list} \to \beta$ and is characterized by the following theorems:

$$\mathsf{rec}\ n\ c\ \mathsf{Nil} = n \qquad \mathsf{rec}\ n\ c\ (\mathsf{Cons}\ x\ \mathit{xs}) = c\ x\ (\mathit{xs}, \mathsf{rec}\ n\ c\ \mathit{xs})$$

In general, for a datatype equipped with m constructors, the recursor takes one argument corresponding to each constructor, followed by a datatype value, and returns a value of an arbitrary type β. The corresponding induction principle has one hypothesis per constructor. For example, for lists it is as follows:

$$\frac{P\ \mathsf{Nil} \quad \bigwedge x\ xs.\ P\ xs \Rightarrow P\ (\mathsf{Cons}\ x\ xs)}{P\ t}$$

Recursive functions *consume* datatype values, peeling off constructors as they proceed. In contrast, corecursive functions *produce* codatatype values, consisting of finitely or infinitely many constructors, one constructor at a time. For each codatatype, a corresponding *corecursor* embodies this principle. It works as follows: Given a codatatype τ with m constructors, $m-1$ predicates sequentially determine which constructor to produce. Moreover, for each argument to each constructor, a function specifies how to construct it from an abstract value of type α that captures the tuple of arguments given to the corecursive function. For corecursive constructor arguments, the function has type $\alpha \to \tau + \alpha$ and returns either a value (τ) that stops the corecursion or a tuple of arguments (α) to a corecursive call. Thus, the corecursor for lazy lists has type

$$(\alpha \to \mathit{bool}) \to (\alpha \to \beta) \to (\alpha \to \beta\ \mathit{llist} + \alpha) \to \alpha \to \beta\ \mathit{llist}$$

and is characterized as follows, where lnl and lnr are the injections into the sum type:

$$\begin{aligned} n\ a &\Rightarrow \mathsf{corec}\ n\ h\ t\ a = \mathsf{LNil} \\ \neg\ n\ a &\Rightarrow \mathsf{corec}\ n\ h\ t\ a = \mathsf{LCons}\ (h\ a)\ (\mathsf{case}\ t\ a\ \mathsf{of}\ \mathsf{lnl}\ xs \Rightarrow xs \mid \mathsf{lnr}\ a' \Rightarrow \mathsf{corec}\ n\ h\ t\ a') \end{aligned}$$

The coinduction principle can be used to prove equalities $l = r$. It is parameterized by a relation R that relates l and r and is closed under application of destructors. Such a relation is called a *bisimulation*. For lazy lists, we have the following principle:

$$\frac{R\ xs\ ys \qquad \begin{array}{r} \bigwedge xs\ ys.\ R\ xs\ ys \Rightarrow \mathsf{lnull}\ xs \leftrightarrow \mathsf{lnull}\ ys \wedge (\neg\ \mathsf{lnull}\ xs \wedge \neg\ \mathsf{lnull}\ ys \longrightarrow \\ \mathsf{lhd}\ xs = \mathsf{lhd}\ ys \wedge R\ (\mathsf{ltl}\ xs)\ (\mathsf{ltl}\ ys)) \end{array}}{xs = ys}$$

5 Nonemptiness Witnesses

The **typedef** primitive requires a nonemptiness witnesses before it introduces the desired type in HOL. Thus, the **datatype** and **codatatype** commands, which build on **typedef**, must provide such a witness [18]. For **datatype**, this is nontrivial. For example, the following inductive specification of "finite streams" must be rejected because it would lead to an empty datatype, one without a nonemptiness witness:

datatype $\alpha\ \mathit{fstream} = \mathsf{FSCons}\ \alpha\ (\alpha\ \mathit{fstream})$

If we substituted **codatatype** for **datatype**, the infinite value $\mathsf{FSCons}\ x\ (\mathsf{FSCons}\ x\ \ldots)$ would be a suitable witness, given a value x of type α.

While checking nonemptiness appears to be an easy reachability test, nested recursion complicates the picture, as shown by this attempt to define infinitely branching trees with finite branches by nested recursion via a codatatype of (infinite) streams:

codatatype α *stream* $=$ SCons α (α *stream*)
datatype α *tree* $=$ Node α ((α *tree*) *stream*)

The second definition should fail: To get a witness for α *tree*, we would need a witness for (α *tree*) *stream*, and vice versa. Replacing streams with finite lists should make the definition acceptable because the empty list stops the recursion. So even though codatatype specifications are never empty, here the datatype provides a better witness (the empty list) than the codatatype (which requires an α *tree* to build an (α *tree*) *stream*).

Mutual, nested datatype specifications and their nonemptiness witnesses can be arbitrarily complex. Consider the following commands:

datatype (α,β) *tree* $=$ Leaf β | Branch (($\alpha+(\alpha,\beta)$ *tree*) *stream*)
codatatype (α,β) *ltree* $=$ LNode β (($\alpha+(\alpha,\beta)$ *ltree*) *stream*)
datatype
$t_1 = \mathsf{T}_{11}$ (((t_1,t_2) *ltree*) *stream*) | T_{12} ($t_1\times(t_2+t_3)$ *stream*) **and**
$t_2 = \mathsf{T}_2$ (($t_1\times t_2$) *list*) **and**
$t_3 = \mathsf{T}_3$ ((t_1, (t_3,t_3) *tree*) *tree*)

The definitions are legitimate, but the last group of mutually recursive datatypes should be rejected if t_2 is replaced by t_3 in the constructor T_{11}.

What makes the problem interesting is the open-ended nature of our setting. BNFs form a *semantic* class that is not syntactically predetermined. In particular, they are not restricted to polynomial functors (sums of products); the user can register new type constructors as BNFs after discharging the BNF proof obligations.

Our solution exploits the package's abstract, functorial view of types. Each (co)datatype, and more generally each functor (type constructor) that participates in a definition, carries its own witnesses. Operations such as functorial composition, initial algebra, and final coalgebra derive their witnesses from those of the operands. Each computational step performed by the package is certified in HOL.

The solution is complete: Given precise information about the functors participating in a definition, all nonempty datatypes are identified as such. A corollary is that the nonemptiness of open-ended, mutual, nested (co)datatypes is decidable. The proof relies on a notion of possibly infinite derivation trees, which can be captured formally as a codatatype. We proved the key results in Isabelle/HOL for an arbitrary unary functor, using the **datatype** and **codatatype** commands to formalize their own metatheory.

6 Primitive Recursion and Corecursion

Primitively recursive functions can be defined by providing suitable arguments to the relevant recursor, and similarly for corecursive functions. The **primrec** and **primcorec** commands automate this process: From the recursive equations specified by the user, they synthesize a (co)recursor-based definition [12, 41]. For example, the command

primrec length : α *list* $\to$ *nat* **where**
length Nil $= 0$
| length (Cons x xs) $=$ $1+$ length xs

synthesizes the definition $\mathsf{length} = \mathsf{rec}\ 0\ (\lambda x\, xs\, n.\ 1 + n)$ and derives the specified equations as theorems, exploiting the recursor's characteristic theorems (Sect. 4).

To qualify as primitive, recursive calls must be directly applied to constructor arguments (e.g., *xs* in the second equation for length). Dually, primitive corecursive calls must occur under exactly one constructor—and possibly some 'if–then–else', 'case', and 'let' constructs—as in the next example:

primcorec $\mathsf{lappend} : \alpha\ \mathit{llist} \to \alpha\ \mathit{llist} \to \alpha\ \mathit{llist}$ **where**
$\mathsf{lappend}\ \mathit{xs}\ \mathit{ys} = (\mathsf{case}\ \mathit{xs}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathit{ys} \mid \mathsf{LCons}\ x\ \mathit{xs} \Rightarrow \mathsf{LCons}\ x\ (\mathsf{lappend}\ \mathit{xs}\ \mathit{ys}))$

With both **primrec** and **primcorec**, an interesting scenario arises for types defined by (co)recursion through a BNF. The (co)recursive calls must then appear inside the map function associated with the BNF. For example:

primrec $\mathsf{height\_tree_{fs}} : \alpha\ \mathit{tree}_{\mathsf{fs}} \to \mathit{nat}$ **where**
$\mathsf{height\_tree_{fs}}\ (\mathsf{Node_{fs}}\ x\ T) = 1 + \bigsqcup(\mathsf{fimage}\ \mathsf{height\_tree_{fs}}\ T)$

Here, $\alpha\ \mathit{tree}_{\mathsf{fs}}$ is the datatype constructed by $\mathsf{Node_{fs}} : \alpha \to (\alpha\ \mathit{tree}_{\mathsf{fs}})\ \mathit{fset} \to \alpha\ \mathit{tree}_{\mathsf{fs}}$, $\bigsqcup N$ stands for the maximum of N, and the map function fimage gives the image of a finite set under a function. From the specified equation, the command synthesizes the definition $\mathsf{height\_tree_{fs}} = \mathsf{rec\_tree_{fs}}\ (\lambda x\, TN.\ 1 + \bigsqcup(\mathsf{fimage}\ \mathsf{snd}\ TN))$. From this definition and $\mathit{tree}_{\mathsf{fs}}$'s recursor theorems, it derives the original equation as a theorem. Notice how the argument $T : (\alpha\ \mathit{tree}_{\mathsf{fs}})\ \mathit{fset}$ becomes $TN : (\alpha\ \mathit{tree}_{\mathsf{fs}} \times \mathit{nat})\ \mathit{fset}$, where the second pair components (extracted by snd) store the result of the corresponding recursive calls.

7 Corecursion up to Friendly Operations

Primitive corecursion is very restrictive. To work around this, Lochbihler and Hölzl dedicated an entire paper [35] to ad hoc techniques for defining operations on lazy lists; and when formalizing formal languages coinductively, Traytel [50] needed to recast the nonprimitive specifications of concatenation and iteration into specifications that can be processed by the **primcorec** command.

Consider the codatatype of streams (infinite lazy lists), with the constructor SCons and the selectors shd and stl:

codatatype $\alpha\ \mathit{stream} = \mathsf{SCons}\ (\mathsf{shd}\colon \alpha)\ (\mathsf{stl}\colon \alpha\ \mathit{stream})$

Primitive corecursion is expressive enough to define operations such as the componentwise addition of two streams of numbers:

primcorec $\oplus : \mathit{nat\ stream} \to \mathit{nat\ stream} \to \mathit{nat\ stream}$ **where**
$\mathit{xs} \oplus \mathit{ys} = \underline{\mathsf{SCons}}\ (\mathsf{shd}\ \mathit{xs} + \mathsf{shd}\ \mathit{ys})\ (\mathsf{stl}\ \mathit{xs} \oplus \mathsf{stl}\ \mathit{ys})$

Intuitively, the evaluation of $\oplus$ makes some progress with each corecursive call, since the call occurs directly under the constructor, which acts as a *guard* (shown underlined). The specification is *productive* and unambiguously characterizes a function. Moreover, it is primitively corecursive, because the topmost symbol on the right-hand side is a constructor and the corecursive call appears directly as an argument to it.

Although these syntactic restrictions can be relaxed to allow conditional statements and 'let' expressions, primitive corecursion remains hopelessly primitive. The syntactic criterion for admissible corecursive definitions in Coq [5] is more permissive in that it allows for an arbitrary number of constructors to guard the corecursive calls, as in the following definition: $\mathsf{oneTwos} = \underline{\mathsf{SCons}}\ 1\ (\underline{\mathsf{SCons}}\ 2\ \mathsf{oneTwos})$.

We designed and implemented a framework, code-named AmiCo, that can be used to define such functions and reason about them [10, 17]. It achieves the same result as Coq by registering SCons as a *friendly* operation, or a *friend*. Intuitively, a friend needs to destruct at most one constructor of input to produce one constructor of output. For streams, such an operation may inspect the head and the tail (but not the tail's tail)—i.e., it may explore at most one layer of its arguments before producing an SCons. Because the operation preserves productivity, it can safely surround the guarding constructor.

But how can we formally express that operators such as SCons and $\oplus$ only explore at most one layer? Inspired by "up to" techniques in category theory [1, 37], we require that the corecursor argument is a composition of an optional destructor and a "surface" function that does not explore its codatatype argument. Formally, the surface must be polymorphic and relationally parametric [43] in that argument.

Our **corec** command generalizes **primcorec** to allow corecursion under friendly operations. The codatatype constructors are automatically registered as friends. Other operations can be registered as friends either after their definition—using the dedicated **friend_of_corec** command, which takes as input either their definition or another proved equation—or at definition time, by passing the **friend** option to **corec**:

corec (**friend**) $\oplus$: *nat stream* $\to$ *nat stream* $\to$ *nat stream* **where**
$xs \oplus ys = \underline{\mathsf{SCons}}\ (\mathsf{shd}\ xs + \mathsf{shd}\ ys)\ (\mathsf{stl}\ xs \oplus \mathsf{stl}\ ys)$

The command synthesizes the corecursor argument and surface functions, defines $\oplus$ in terms of the corecursor, and derives the user's equation as a theorem. It additionally checks that $\oplus$ meets the criteria on friends and registers it as such.

After registering friends, the corecursor becomes more expressive, allowing corecursive calls surrounded by any combinations of friends. In other words, the corecursor gradually grows to recognize more friends, going well beyond the syntactic criterion implemented in Coq and other systems. For example, the shuffle product $\otimes$ of two streams is defined in terms of $\oplus$, and already goes beyond the corecursive definition capabilities of Coq. Shuffle product being itself friendly, we can employ it to define stream exponentiation, which is also friendly:

corec (**friend**) $\otimes$: *nat stream* $\to$ *nat stream* $\to$ *nat stream* **where**
$xs \otimes ys = \underline{\mathsf{SCons}}\ (\mathsf{shd}\ xs \times \mathsf{shd}\ ys)\ ((xs \otimes \mathsf{stl}\ ys) \oplus (\mathsf{stl}\ xs \otimes ys))$
corec (**friend**) exp : *nat stream* $\to$ *nat stream* **where**
$\mathsf{exp}\ xs = \underline{\mathsf{SCons}}\ (2\ \hat{}\ \mathsf{shd}\ xs)\ (\mathsf{stl}\ xs \otimes \mathsf{exp}\ xs)$

Friends also form a basis for soundly combining recursion with corecursion. The following definition exhibits both recursion on the naturals and corecursion on streams:

corec cat : *nat* $\to$ *nat stream* **where**
$\mathsf{cat}\ n = (\text{if } n > 0 \text{ then } \mathsf{cat}\ (n-1) \oplus \underline{\mathsf{SCons}}\ 0\ (\mathsf{cat}\ (n+1)) \text{ else } \underline{\mathsf{SCons}}\ 1\ (\mathsf{cat}\ 1))$

The call cat 1 computes the stream $C_1, C_2, \ldots$ of Catalan numbers, where $C_n = \frac{1}{n+1}\binom{2n}{n}$.

The first self-call, $\mathsf{cat}\ (n-1)$, is recursive, whereas the others are corecursive. Both recursive and corecursive calls are required to appear in friendly contexts, whereas only the corecursive calls are required to be guarded. In exchange, the recursive calls should be terminating: They should eventually lead to either a base case or a corecursive call. AmiCo automatically marks unguarded calls as recursive and attempts to prove their termination using Isabelle/HOL's termination prover [19]. Users also have the option to discharge the proof obligation manually.

8 Nonuniform Datatypes and Codatatypes

Nonuniform (co)datatypes are recursively defined types in which the type arguments vary recursively. Powerlists and powerstreams are prominent specimens:

nonuniform_datatype $\alpha\ \mathit{plist} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha)\ \mathit{plist})$
nonuniform_codatatype $\alpha\ \mathit{pstream} = \mathsf{SCons}\ \alpha\ ((\alpha \times \alpha)\ \mathit{pstream})$

The type $\alpha\ \mathit{plist}$ is freely generated by $\mathsf{Nil} : \alpha\ \mathit{plist}$ and $\mathsf{Cons} : \alpha \to (\alpha \times \alpha)\ \mathit{plist} \to \alpha\ \mathit{plist}$. When Cons is applied several times, the product type constructors ($\times$) accumulate to create pairs, pairs of pairs, and so on. Thus, any powerlist of length 3 will have the form

$$\mathsf{Cons}\ a\ (\mathsf{Cons}\ (b_1, b_2)\ (\mathsf{Cons}\ ((c_{11}, c_{12}),\ (c_{21}, c_{22}))\ \mathsf{Nil}))$$

Similarly, the type *pstream* contains only infinite values of the form

$$\mathsf{SCons}\ a\ (\mathsf{SCons}\ (b_1, b_2)\ (\mathsf{SCons}\ ((c_{11}, c_{12}),\ (c_{21}, c_{22}))\ \ldots))$$

Nonuniform datatypes arise in the implementation of efficient functional data structures such as finger trees [23], and they underlie Okasaki's bootstrapping and implicit recursive slowdown optimization techniques [40]. Agda, Coq, Lean, and Matita allow nonuniform definitions, but these are built into the logic, with all the risks and limitations that this entails [17, Sect. 1]. For systems based on HOL, until recently no dedicated support existed for nonuniform types, probably because they were widely believed to lie beyond the logic's simple polymorphism. Building on the BNF infrastructure, we disproved this folklore belief by showing how to define a large class of nonuniform datatypes by reduction to their uniform counterparts within HOL [13, 36].

Our constructions allow variations along several axes for both datatypes and codatatypes. They allow multiple recursive occurrences, with different type arguments:

nonuniform_datatype $\alpha\ \mathit{plist}' = \mathsf{Nil} \mid \mathsf{Cons}_1\ \alpha\ (\alpha\ \mathit{plist}') \mid \mathsf{Cons}_2\ \alpha\ ((\alpha \times \alpha)\ \mathit{plist}')$

They allow multiple type arguments, which may all vary independently of the others. Moreover, they allow the presence of uniform or nonuniform (co)datatypes and other BNFs both around the type arguments and around the recursive type occurrences:

nonuniform_datatype $\alpha\ \mathit{crazy} = \mathsf{Node}\ \alpha\ (((((\alpha\ \mathit{pstream})\ \mathit{fset})\ \mathit{crazy})\ \mathit{fset})\ \mathit{list})$

Once a nonuniform datatype has been introduced, users want to define functions that recurse on it and carry out proofs by induction involving these functions—and similarly for codatatypes. A uniform datatype definition generates an induction theorem and a recursor. Nonuniform datatypes pose a challenge, because neither the induction theorem nor the recursor can be expressed in HOL, due to its limited polymorphism. For example,

the induction principle for *plist* should look like this:

$$\bigwedge Q.\ Q\ \mathsf{Nil} \wedge \big(\bigwedge x\ xs.\ Q\ xs \Rightarrow Q\ (\mathsf{Cons}\ x\ xs)\big) \Rightarrow \bigwedge ys.\ Q\ ys$$

However, this formula is not typable in HOL, because the second and third occurrences of the variable Q need different types: $(\alpha \times \alpha)\ plist \to bool$ versus $\alpha\ plist \to bool$. Our solution is to replace the theorem by a procedure parameterized by a polymorphic property $\varphi_\alpha : \alpha\ plist \to bool$. For *plist*, the procedure transforms a proof goal of the form $\varphi_\alpha\ ys$ into two subgoals $\varphi_\alpha\ \mathsf{Nil}$ and $\bigwedge x\ xs.\ \varphi_{\alpha\times\alpha}\ xs \Rightarrow \varphi_\alpha\ (\mathsf{Cons}\ x\ xs)$. A weak form of parametricity is needed to recursively transfer properties about φ_α to properties about $\varphi_{\alpha\times\alpha}$. Our approach to (co)recursion is similar.

9 Tool Integration

Lifting and Transfer. Isabelle/HOL's Lifting and Transfer tools [26] provide automation for working with type abstractions introduced via the **typedef** command. Lifting defines constants on the newly introduced abstract type from constants on the original raw type. Transfer reduces proof goals about the abstract type to goals about the raw type. Both tools are centered around parametricity and relators.

The BNF infrastructure serves as an abundant supply of relator constants, their properties, and parametricity theorems about the constructors, 'case' combinators, recursors, and the BNF map, set, and relator constants. The interaction between Lifting, Transfer, and the BNF and (co)datatype databases is implemented using Isabelle's plugin mechanism. Plugins are callbacks that are executed upon every update to the BNF or (co)datatype database, as well as for all existing database entries at the moment of the registration of the plugin. The Lifting and Transfer plugins derive and register properties in the format accepted by those tools from the corresponding properties in the BNF and (co)datatype databases.

To enable nested recursion through types introduced by **typedef**, we must register the types as BNFs. The BNF structure can often be lifted from the raw type to the abstract type in a canonical way. The command **lift_bnf** automates this lifting based on a few properties of the carved-out subset: Essentially, the subset must be closed under $\mathsf{map}\ f$ for any f, where map is the map function of the raw type's BNF. If the carved out subset is the entire type, the **copy_bnf** command performs the trivial lifting of the BNF structure. This command is particularly useful to register types defined via Isabelle/HOL's **record** command, which are type copies of some product type, as BNFs.

Size, Countability, Comparators, Show, and Hash. For each finitary datatype τ, the size plugin generates a function $\mathsf{size} : \tau \to nat$. The **fun** and **function** commands [29] rely on size to prove termination of recursive functions on datatypes.

The *countable_datatype* tactic can be used to prove the countability of many datatypes, building on the countability of the types appearing in their definitions.

The **derive** command [46], provided by the *Archive of Formal Proofs* [47], automatically generates comparators, show functions, and hash functions for a specified datatype and can be extended to generate other operations. The mechanism is inspired by Haskell's **deriving** mechanism, with the important difference that it also provides theorems about the operations it introduces.

Nitpick and Sledgehammer. Nitpick [6, 8] is a counterexample generator for Isabelle/HOL that builds on Kodkod [49], a SAT-based first-order relational model finder. Nitpick supported codatatypes even before the introduction of a **codatatype** command. Users could define custom codatatypes from first principles and tell Nitpick to employ its efficient first-order relational axiomatization of ω-regular values (e.g., cyclic values).

Sledgehammer integrates automatic theorem provers in Isabelle/HOL to provide one-click proof automation. Some automatic provers have native support for datatypes [28, 38, 42]; for these, Sledgehammer generates native definitions, which are often more efficient and complete than first-order axiomatizations. Blanchette also collaborated with the developers of the SMT solver CVC4 to add codatatypes to their solver [42].

10 Applications

Coinductive. Lochbihler's Coinductive library [31] defines general-purpose codatatypes, notably extended natural numbers ($\mathbb{N} \uplus \{\infty\}$), lazy lists, and streams. It also provides related functions and a large collection of lemmas about these. Back in 2010, every codatatype was constructed manually—including its constructors and corecursor—and operations were defined directly in terms of the corecursor. Today, the codatatypes are defined with **codatatype** and most functions with **primcorec**, leading to considerably shorter definitions and proofs [12]. The library is used in several applications, including in Hölzl's formalization of Markov chains and processes [24, 25] and in Lochbihler's JinjaThreads project to verify a Java compiler and formalize the Java memory model [30, 32, 33].

Coinductive Languages. Rutten [44] views formal languages as infinite tries—i.e., prefix trees branching over the alphabet with Boolean labels at the nodes indicating whether the path from the root denotes a word in the language. Traytel [50] formalized these tries in Isabelle as

codatatype $\alpha\ lang\ =\ \mathsf{Lang}\ bool\ (\alpha \to \alpha\ lang)$

a type that nests corecursion through the right-hand side of the function space arrow ($\to$). He also defined regular operations on them as corecursive functions and proved by coinduction that the defined operations form a Kleene algebra.

Completeness of First-Order Logic. Gödel's completeness theorem [21] is a central result about first-order logic. Blanchette, Popescu, and Traytel [9, 14, 16] formalized a Beth–Hintikka-style proof [27] in Isabelle/HOL. It depends on a Gentzen or tableau system and performs a search that builds either a finite deduction tree yielding a proof (or refutation, depending on the system) or an infinite tree from which a countermodel (or model) can be extracted.

Even in the most formalistic textbooks, potentially infinite trees are defined rigorously (e.g., as prefix-closed sets), but the reasoning is performed informally, disregarding the definition and relying on the intuitive notion of trees. By contrast, the formalization relies on $\alpha\ ltree_{\mathsf{fs}}$ (Sect. 1), a codatatype of finitely branching, possibly infinite trees with nodes labeled by elements in a set α of inference rules. One could argue that trees are intuitive and do not need a formal treatment, but the same holds for the syntax of formulas, which is treated very rigorously in most textbooks.

The core of the proof establishes an abstract property of possibly infinite derivation trees, independently of the concrete syntax or inference rules. This separation of concerns simplifies the presentation. The abstract proof can be instantiated for a wide range of Gentzen and tableau systems as well as variants of first-order logic.

IsaFoR and CeTA. The IsaFoR (Isabelle Formalization of Rewriting) formal library, developed by Sternagel, Thiemann, and their colleagues, is a collection of abstract results and concrete techniques from the term rewriting literature. It forms the basis of the CeTA (Certified Termination Analysis) certifier [48] for proofs of (non)termination, (non)confluence, and other properties of term rewriting systems. Termination proofs are represented by complicated mutually and nested recursive datatypes.

One of the benefits of the modular, BNF-based approach is its scalability. The previous approach [4, 22] implemented in Isabelle/HOL consisted in reducing specifications with nested recursion to mutually recursive specifications, which scales poorly (and only allows nesting through datatypes). After the introduction of the new **datatype** command in 2014, Thiemann observed that the IsaFoR session *Proof-Checker* compiled in 10 minutes on his computer, compared with 50 minutes previously.

Generative Probabilistic Values. Lochbihler [34] proposed generative probabilistic values (GPVs) as a semantic domain for probabilistic input–output systems, which he uses to formalize and verify cryptographic algorithms. Conceptually, each GPV chooses probabilistically between failing, terminating with a result of type α, and continuing by producing an output γ and transitioning into a reactive probabilistic value, which waits for a response ρ of the environment before moving to the generative successor state. Lochbihler modeled GPVs as a codatatype (α, γ, ρ) *gpv* and defined a monadic language on GPVs similar to a coroutine monad:

$$\textbf{codatatype}\ (\alpha,\gamma,\rho)\ gpv\ =\ \mathsf{GPV}\ (\mathsf{unGPV}\colon (\alpha + \gamma \times (\rho \to (\alpha,\gamma,\rho)\ gpv))\ spmf)$$

This codatatype definition exploits the full generality that BNFs provide as it corecurses through the nonfree type constructor *spmf* of discrete subprobability distributions and through the function space ($\to$), products ($\times$), and sums ($+$).

The definition of the 'while' loop corecurses through the monadic sequencing operator $\gg\!=_{gpv}$ and is accepted by **corec** after $\gg\!=_{gpv}$ has been registered as a friend (Sect. 7):

$$\begin{aligned}&\textbf{corec}\ \mathsf{while} : (\sigma \to bool) \to (\sigma \to (\sigma,\gamma,\rho)\ gpv) \to \sigma \to (\sigma,\gamma,\rho)\ gpv\ \textbf{where}\\ &\quad \mathsf{while}\ g\ b\ s =\\ &\quad \mathsf{GPV}\ (\mathsf{map}_{spmf}\ (\mathsf{map}_{+}\ \mathsf{id}\ (\mathsf{map}_{\times}\ \mathsf{id}\ (\lambda x\ r.\ x\ r \gg\!=_{gpv} \mathsf{while}\ g\ b)))\ (\mathsf{search}\ g\ b\ s))\end{aligned}$$

The auxiliary operation search $g\ b\ s$ iterates the loop body b starting from state s until the loop guard g is falsified or the first interaction is found. It is defined as the least fixpoint of the recursive specification in the *spmf* monad below. The search is needed to expose the constructor guard in while's definition. The recursion in search must be manually separated from the corecursion as the recursion is not well founded, so search is not the only solution—e.g., it is unspecified for $g\ s = \mathsf{True}$ and $b\ s = \mathsf{GPV}\ (\mathsf{return}_{spmf}\ (\mathsf{Inl}\ s))$.

$$\begin{aligned}&\mathsf{search}\ g\ b\ s = (\mathsf{if}\ g\ s\ \mathsf{then}\\ &\qquad \mathsf{unGPV}\ (b\ s) \gg\!=_{spmf} (\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Inl}\ s' \Rightarrow \mathsf{search}\ g\ b\ s' \mid \_ \Rightarrow \mathsf{return}_{spmf}\ x)\\ &\quad \mathsf{else}\ \mathsf{return}_{spmf}\ (\mathsf{Inl}\ s))\end{aligned}$$

Nested and Hereditary Multisets. Blanchette, Fleury, and Traytel [7, 11] formalized a collection of results about (finite) nested multisets, as a case study for BNFs. Nested multisets can be defined simply, exploiting the BNF structure of *multiset*:

datatype α *nmultiset* $=$ Elem α | MSet $((\alpha$ *nmultiset*$)$ *multiset*$)$

This type forms the basis of their formalization of Dershowitz and Manna's nested multiset order [20]. If we omit the Elem case, we obtain the hereditary multisets instead:

datatype *hmultiset* $=$ HMSet $($*hmultiset multiset*$)$

This type is similar to hereditarily finite sets, a model of set theory without the axiom of infinity, but with multisets instead of finite sets. Indeed, we can replace *multiset* with *fset* to obtain the hereditarily finite sets.

It is easy to embed *hmultiset* in α *nmultiset*, and using the Lifting and Transfer tools, we can lift definitions and results from the larger type to the smaller type, such as the definition of the nested multiset order. Hereditary multisets offer a convenient syntactic representation for ordinals below ϵ_0, which can be expressed in Cantor normal form:

$$\alpha ::= \omega^{\alpha_1} \cdot c_1 + \cdots + \omega^{\alpha_n} \cdot c_n$$

where $c_i \in \mathbb{N}^{>0}$ and $\alpha_1 > \cdots > \alpha_n$. The correspondence with hereditary multisets is straightforward:

$$\alpha ::= \{\underbrace{\alpha_1, \ldots, \alpha_1}_{c_1 \text{ occurrences}}, \ldots, \underbrace{\alpha_n, \ldots, \alpha_n}_{c_n \text{ occurrences}}\}$$

The coefficients c_i are represented by multiset multiplicities, and the ω exponents are the multiset's members. Thus, $\{\} = 0$; $\{0\} = \{\{\}\} = \omega^0 = 1$; $\{0,0,0\} = \{\{\},\{\},\{\}\} = \omega^0 \cdot 3 = 3$; $\{1\} = \{\{\{\}\}\} = \omega^1 = \omega$; and $\{\omega\} = \{\{\{\}\}\}\} = \omega^\omega$.

The hereditary multisets were used to represent syntactic ordinals in a proof of Goodstein's theorem [7, 11], in an ongoing proof of the decidability of unary PCF (programming computable functions) [7, 11], and in a formalization of transfinite Knuth–Bendix orders [2, 3].

11 Conclusion

It is widely recognized that proof automation is important for usability of a proof assistant, but it is not the only factor. Many formalizations depend on an expressive specification language. The axiomatic approach, which is favored in some subcommunities, is considered unreliable in others. Extending the logic is also a problematic option: Not only must the metatheory be extended, but the existing tools must be adapted. Moreover, the developers and users of the system must be convinced of the correctness and necessity of the extension.

Our challenge was to combine specification mechanisms that are both expressive and trustworthy, without introducing new axioms or changing the logic. We believe we have succeeded as far as (co)datatypes and (co)recursion are concerned, but more could be done, notably for nonfree datatypes [45]. Our new commands, based on the notion of a bounded natural functor, probably constitute the largest definitional package to have been implemented in a proof assistant. Makarius Wenzel [54], Isabelle's lead developer, jocularly called it "one of the greatest engineering projects since Stonehenge!"

Acknowledgments. We first want to acknowledge the support and encouragement of past and current bosses: David Basin, Wan Fokkink, Stephan Merz, Aart Middeldorp, Tobias Nipkow, and Christoph Weidenbach. We are grateful to the FroCoS 2017 program chairs, Clare Dixon and Marcelo Finger, and to the program committee for giving us this opportunity to present our research. We are also indebted to Andreas Abel, Stefan Berghofer, Sascha Böhme, Lukas Bulwahn, Elsa Gunter, Florian Haftmann, Martin Hofmann, Brian Huffman, Lars Hupel, Alexander Krauss, Peter Lammich, Rustan Leino, Stefan Milius, Lutz Schröder, Mark Summerfield, Christian Urban, Daniel Wand, and Makarius Wenzel, and to dozens of anonymous reviewers (including those who rejected our manuscript "Witnessing (co)datatypes" [18] six times).

Blanchette was supported by the Deutsche Forschungsgemeinschaft (DFG) projects "Quis Custodiet" (NI 491/11-2) and "Den Hammer härten" (NI 491/14-1). He also received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Hölzl was supported by the DFG project "Verifikation probabilistischer Modelle in interaktiven Theorembeweisern" (NI 491/15-1). Kunčar and Popescu were supported by the DFG project "Security Type Systems and Deduction" (NI 491/13-2 and NI 491/13-3) as part of the program Reliably Secure Software Systems (RS^3, priority program 1496). Kunčar was also supported by the DFG project "Integration der Logik HOL mit den Programmiersprachen ML und Haskell" (NI 491/10-2). Lochbihler was supported by the Swiss National Science Foundation (SNSF) grant "Formalising Computational Soundness for Protocol Implementations" (153217). Popescu was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) starting grant "VOWS: Verification of Web-based Systems" (EP/N019547/1). Sternagel and Thiemann were supported by the Austrian Science Fund (FWF): P27502 and Y757. Traytel was supported by the DFG program "Programm- und Modell-Analyse" (PUMA, doctorate program 1480). The authors are listed alphabetically.

References

[1] Bartels, F.: Generalised coinduction. Math. Struct. Comp. Sci. 13(2), 321–348 (2003)

[2] Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: Formalization of Knuth–Bendix orders for lambda-free higher-order terms. Archive of Formal Proofs (2016), `http://isa-afp.org/entries/Lambda_Free_KBOs.shtml`, Formal proof development

[3] Becker, H., Blanchette, J.C., Waldmann, U., Wand, D.: A transfinite Knuth–Bendix order for lambda-free higher-order terms. In: de Moura, L. (ed.) CADE-26. LNCS, Springer (2017)

[4] Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin-Mohring, C., Théry, L. (eds.) TPHOLs '99. LNCS, vol. 1690, pp. 19–36. Springer (1999)

[5] Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004)

[6] Blanchette, J.C.: Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions. Softw. Qual. J. 21(1), 101–126 (2013)

[7] Blanchette, J.C., Fleury, M., Traytel, D.: Nested multisets, hereditary multisets, and syntactic ordinals in Isabelle/HOL. In: Miller, D. (ed.) FSCD 2017. LIPIcs, vol. 84, pp. 11:1–11:17. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2017)

[8] Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer (2010)

[9] Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reasoning 58(1), 149–179 (2017)

[10] Blanchette, J.C., Bouzy, A., Lochbihler, A., Popescu, A., Traytel, D.: Friends with benefits: Implementing corecursion in foundational proof assistants. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 111–140. Springer (2017)
[11] Blanchette, J.C., Fleury, M., Traytel, D.: Formalization of nested multisets, hereditary multisets, and syntactic ordinals. Archive of Formal Proofs (2016), http://isa-afp.org/entries/Nested_Multisets_Ordinals.shtml, Formal proof development
[12] 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, vol. 8558, pp. 93–110. Springer (2014)
[13] Blanchette, J.C., Meier, F., Popescu, A., Traytel, D.: Foundational nonuniform (co)datatypes for higher-order logic. In: Ouaknine, J. (ed.) LICS 2017. IEEE Computer Society (2017)
[14] Blanchette, J.C., Popescu, A., Traytel, D.: Abstract completeness. Archive of Formal Proofs (2014), http://isa-afp.org/entries/Abstract_Completeness.shtml, Formal proof development
[15] Blanchette, J.C., Popescu, A., Traytel, D.: Cardinals in Isabelle/HOL. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 111–127. Springer (2014)
[16] Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness: A coinductive pearl. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 46–60. Springer (2014)
[17] Blanchette, J.C., Popescu, A., Traytel, D.: Foundational extensible corecursion—A proof assistant perspective. In: Fisher, K., Reppy, J.H. (eds.) ICFP '15. pp. 192–204. ACM (2015)
[18] Blanchette, J.C., Popescu, A., Traytel, D.: Witnessing (co)datatypes. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 359–382. Springer (2015)
[19] Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer (2007)
[20] Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM 22(8), 465–476 (1979)
[21] Gödel, K.: Über die Vollständigkeit des Logikkalküls. Ph.D. thesis, Universität Wien (1929)
[22] Gunter, E.L.: Why we can't have SML-style `datatype` declarations in HOL. In: TPHOLs '92. IFIP Transactions, vol. A-20, pp. 561–568. North-Holland/Elsevier (1993)
[23] Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. J. Funct. Program. 16(2), 197–217 (2006)
[24] Hölzl, J.: Markov chains and Markov decision processes in Isabelle/HOL. Accepted in J. Autom. Reasoning
[25] Hölzl, J.: Markov processes in Isabelle/HOL. In: Bertot, Y., Vafeiadis, V. (eds.) CPP 2017. pp. 100–111. ACM (2017)
[26] 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)
[27] Kleene, S.C.: Mathematical Logic. John Wiley & Sons (1967)
[28] Kovács, L., Robillard, S., Voronkov, A.: Coming to terms with quantified reasoning. In: Castagna, G., Gordon, A.D. (eds.) POPL 2017. pp. 260–270. ACM (2017)
[29] Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 589–603. Springer (2006)
[30] Lochbihler, A.: Jinja with threads. Archive of Formal Proofs (2007), http://isa-afp.org/entries/JinjaThreads.shtml, Formal proof development
[31] Lochbihler, A.: Coinductive. Archive of Formal Proofs (2010), http://afp.sf.net/entries/Coinductive.shtml, Formal proof development
[32] Lochbihler, A.: Verifying a compiler for Java threads. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 427–447. Springer (2010)

[33] Lochbihler, A.: Making the Java memory model safe. ACM Trans. Program. Lang. Syst. 35(4), 12:1–65 (2014)
[34] Lochbihler, A.: Probabilistic functions and cryptographic oracles in higher-order logic. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 503–531. Springer (2016)
[35] Lochbihler, A., Hölzl, J.: Recursive functions on lazy lists via domains and topologies. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 341–357. Springer (2014)
[36] Meier, F.: Non-Uniform Datatypes in Isabelle/HOL. M.Sc. thesis, ETH Zürich (2016)
[37] Milius, S., Moss, L.S., Schwencke, D.: Abstract GSOS rules and a modular treatment of recursive definitions. Log. Meth. Comput. Sci. 9(3) (2013)
[38] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)
[39] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
[40] Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
[41] Panny, L.: Primitively (Co)recursive Function Definitions for Isabelle/HOL. B.Sc. thesis, Technische Universität München (2014)
[42] Reynolds, A., Blanchette, J.C.: A decision procedure for (co)datatypes in SMT solvers. J. Autom. Reasoning 58(3), 341–362 (2017)
[43] Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP '83. pp. 513–523 (1983)
[44] Rutten, J.J.M.M.: Automata and coinduction (an exercise in coalgebra). In: Sangiorgi, D., de Simone, R. (eds.) CONCUR '98. LNCS, vol. 1466, pp. 194–218. Springer (1998)
[45] 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)
[46] Sternagel, C., Thiemann, R.: Deriving comparators and show functions in Isabelle/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 421–437. Springer (2015)
[47] Sternagel, C., Thiemann, R.: Deriving class instances for datatypes. Archive of Formal Proofs (2015), http://isa-afp.org/entries/Deriving.shtml, Formal proof development
[48] Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer (2009)
[49] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer (2007)
[50] Traytel, D.: Formal languages, formally and coinductively. In: Kesner, D., Pientka, B. (eds.) FSCD 2016. LIPIcs, vol. 52, pp. 31:1–31:17. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2016)
[51] 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 Computer Society (2012)
[52] Traytel, D.: A Category Theory Based (Co)datatype Package for Isabelle/HOL. M.Sc. thesis, Technische Universität München (2012)
[53] Wenzel, M.: Isabelle/Isar—A generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof: Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric, vol. 10(23). Uniwersytet w Białymstoku (2007)
[54] Wenzel, M.: Re: [isabelle] "Unfolding" the sum-of-products encoding of datatypes (2015), https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2015-November/msg00082.html