Foundational Extensible Corecursion: A Proof Assistant Perspective

Jasmin Christian Blanchette, Andrei Popescu, Dmitriy Traytel

Foundational Extensible Corecursion

A Proof Assistant Perspective

Jasmin Christian Blanchette
Inria & LORIA, Nancy, France
Max-Planck-Institut für Informatik, Saarbrücken, Germany
jasmin.blanchette@inria.fr

Andrei Popescu
Department of Computer Science, School of Science and Technology, Middlesex University, UK
a.popescu@mdx.ac.uk

Dmitriy Traytel
Technische Universität München, Germany
traytel@in.tum.de

Abstract

This paper presents a formalized framework for defining corecursive functions safely in a total setting, based on corecursion up-to and relational parametricity. The end product is a general corecursor that allows corecursive (and even recursive) calls under "friendly" operations, including constructors. Friendly corecursive functions can be registered as such, thereby increasing the corecursor's expressiveness. The metatheory is formalized in the Isabelle proof assistant and forms the core of a prototype tool. The corecursor is derived from first principles, without requiring new axioms or extensions of the logic.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Mechanical theorem proving, Model theory

General Terms Theory, Verification

Keywords (Co)recursion, parametricity, proof assistants, higher-order logic, Isabelle

1. Introduction

Total functional programming is a discipline that ensures computations always terminate. It is invaluable in a proof assistant, where nonterminating definitions such as $\mathsf{f}\ x = \mathsf{f}\ x + 1$ can be interpreted in such a way as to yield contradictions. Hence, most assistants will accept recursive functions only if they can be shown to terminate. Similar concerns arise in specification languages and verifying compilers.

However, some processes need to run forever, without their being inconsistent. An important class of total programs has been identified under the heading of *productive coprogramming* [1, 8, 62]: These are functions that progressively reveal parts of their (potentially infinite) output. For example, given a type of infinite streams constructed by SCons, the definition

$$\mathsf{natsFrom}\ n = \mathsf{SCons}\ n\ (\mathsf{natsFrom}\ (n+1))$$

falls within this fragment, since each call to $\mathsf{natsFrom}$ produces one constructor before entering the nested call.

The above definition is legitimate only if objects are allowed to be infinite. This may be self-evident in a nonstrict functional language such as Haskell, but in a total setting we must carefully distinguish between the well-founded inductive (or algebraic) datatypes and the non-well-founded coinductive (or coalgebraic) datatypes—often simply called *datatypes* and *codatatypes*, respectively. *Recursive* functions consume datatype values, peeling off constructors as they proceed; *corecursive* functions produce codatatype values, consisting of finitely or infinitely many constructors. And in the same way that *induction* is available as a proof principle to reason about datatypes and terminating recursive functions, *coinduction* supports reasoning about codatatypes and productive corecursive functions.

Despite their reputation for esotericism, codatatypes have an important role to play in both the theory and the metatheory of programming. On the theory side, they allow a direct embedding of a large class of nonstrict functional programs in a *total* logic. In conjunction with interactive proofs and code generators, this enables certified functional programming [11]. On the metatheory side, codatatypes conveniently capture infinite, possibly branching processes. Major proof developments rely on them, including those associated with a C compiler [41], a Java compiler [42], and the Java memory model [43].

Codatatypes are supported by an increasing number of proof assistants, including Agda [18], Coq [13], Isabelle/HOL [48], Isabelle/ZF [49, 50], Matita [7], and PVS [21]. They are also present in the CoALP dialect of logic programming [27] and in the Dafny specification language [40]. But the ability to introduce codatatypes is not worth much without adequate support for defining meaningful functions that operate on them. For most systems, this support can be characterized as work in progress. The key question they all must answer is: *What right-hand sides can be safely allowed in function definitions?*

Generally, there are two main ways to support recursive and corecursive functions in a proof assistant or similar system:

The intrinsic approach: A syntactic criterion is built into the logic: termination for recursive specifications, productivity (or guardedness) for corecursive specifications. The termination or productivity checker is part of the system's trusted code base.

The foundational approach: The (co)recursive specifications are reduced to a fixpoint construction inside the given logic, which permits a simple definition of the form $\mathsf{f} = \ldots$, where f does not occur in the right-hand side. The original equations are then derived as theorems from this internal definition by dedicated proof tactics.

Systems favoring the intrinsic approach include the proof assistants Agda and Coq, as well as tools such as CoALP and Dafny. The main hurdle for their users is that syntactic criteria are inflexible; the specification must be massaged so that it falls within a given syntactic fragment, even though the desired property (termination or productivity) is semantic. But perhaps more troubling for systems that process theorems, soundness is not obvious at all and very tedious to ensure; as a result, there is a history of critical bugs in termination and productivity checkers, as we will see when we review related work (Section 7). Indeed, Abel [4] observed that

> Maybe the time is ripe to switch to a more semantical notion of termination and guardedness. The syntactic guard condition gets you somewhere, but then needs a lot of extensions and patching to work satisfactor[il]y in practice. Formal verification of it becomes too difficult, and only intuitive justification is prone to errors.

In contrast to Agda and Coq, proof assistants based on higher-order logic (HOL), such as HOL4, HOL Light, and Isabelle/HOL, generally adhere to the foundational approach. Their logic is expressive enough to accommodate the (co)algebraic constructions underlying (co)datatypes and (co)recursive functions in terms of functors on the category of sets [60]. The main drawback of this approach is that it requires a lot of work, both conceptual and implementational. Moreover, it is not available for all systems, since it requires an expressive enough logic.

Because every step must be justified, foundational definitional principles tend to be more restrictive than their intrinsic counterparts. As a telling example, codatatypes were introduced in Isabelle/HOL only recently, almost two decades after their inception in Coq, and they are still missing in other HOL systems. Before the work reported in this paper, corecursion was limited to the primitive case, in which self-calls occur under exactly one constructor.

That primitive corecursion (or the slightly extended version supported by Coq) is too restrictive is an observation that has been made repeatedly by researchers who use corecursion in Coq and now also Isabelle. Lochbihler and Hölzl dedicated a paper [44] to ad hoc techniques for defining operations on corecursive lists in Isabelle. Only after introducing a lot of machinery do they manage to define their central example—lfilter, a filter function on lazy (coinductive) lists—and derive suitable reasoning principles.

We contend that it is possible to combine advanced features as found in Agda and Coq with the fundamentalism of Isabelle. The lack of built-in support for corecursion, an apparent weakness, reveals itself as a strength as we proceed to introduce rich notions of corecursion, without extending the type system or adding axioms.

In this paper, we formalize a highly expressive corecursion framework that extends primitive corecursion in the following ways: It allows corecursive calls under several constructors; it allows "friendly" operations in the context around or between the constructors and around the corecursive calls; importantly, it supports blending terminating recursive calls with guarded corecursive calls. This general corecursor is accompanied by a corresponding, equally general coinduction principle that makes reasoning about it convenient. The corecursor and the coinduction principle grow in expressiveness during the interaction with the user, by learning new friendly contexts. In process algebra terminology [58], both corecursion and coinduction take place "up to" friendly contexts. The constructions draw heavily from category theory.

Before presenting the technical details, we first show through examples how a primitive corecursor can be incrementally enriched to accept ever richer notions of corecursive call context (Section 2). This is made possible by the modular bookkeeping of additional structure for the involved type constructors, including a relator structure. This structure can be exploited to prove parametricity theorems, which allow to mix operations freely in the corecursive call contexts, in the style of coinduction up-to. Each new corecursive definition is a potential future participant (Section 3).

This extensible corecursor gracefully handles codatatypes with nesting through arbitrary type constructors (e.g., for infinite-depth Rose trees nested through finite or infinite lists). Thanks to the framework's modularity, function specifications can combine corecursion with recursion, yielding quite expressive mixed fixpoint definitions (Section 4). This is inspired by the Dafny tool, but our approach is semantically founded and hence provably consistent.

Our framework is implemented in Isabelle/HOL, as a combination of a generic proof development parameterized by arbitrary type constructors and a tool for instantiating the metatheory to user-specified instances (Sections 5 and 6). It is available online along with the examples from this paper [16].

Techniques such as corecursion and coinduction up-to have been known for years in the process algebra community, before they were embraced and perfected by category theorists (Section 7). This work is part of a wider program aiming at bringing insight from category theory into proof assistants [14, 15, 17, 60]. The main contributions of this paper are the following:

- We represent in higher-order logic a framework for corecursion that evolves by user interaction.
- We identify a sound fragment of mixed recursive–corecursive specifications, integrate it in our framework, and present several examples that motivate this feature.
- We implement the above in Isabelle/HOL within an interactive loop that maintains the recursive–corecursive infrastructure.
- We use this infrastructure to automatically derive many examples that are problematic in other proof assistants.

A distinguishing feature of our framework is that it does not require the user to provide type annotations. On the design space, it lies between the restrictive primitive corecursion and the expressive but more bureaucratic approaches such as clock variables [8, 20] and sized types [2], combining expressiveness and ease of use. The identification of this "sweet spot" can also be seen as a contribution.

2. Motivating Examples

We demonstrate the expressiveness of our corecursor framework by examples, adopting the user's perspective. The case studies by Rutten [57] and Hinze [28] on stream calculi serve as our starting point. Streams of natural numbers can be defined as

$$\mathsf{codatatype}\ \mathsf{Stream} = \mathsf{SCons}\ (\mathsf{head}\colon \mathsf{Nat})\ (\mathsf{tail}\colon \mathsf{Stream})$$

where $\mathsf{SCons} : \mathsf{Nat} \to \mathsf{Stream} \to \mathsf{Stream}$ is a constructor and $\mathsf{head} : \mathsf{Stream} \to \mathsf{Nat}$, $\mathsf{tail} : \mathsf{Stream} \to \mathsf{Stream}$ are selectors. The examples were chosen to show the main difficulties that arise in practice.

2.1 Corecursion Up-to

As our first example of a corecursive function definition, we consider the pointwise sum of two streams:

$$xs \oplus ys = \underline{\mathsf{SCons}}\ (\mathsf{head}\ xs + \mathsf{head}\ ys)\ (\mathsf{tail}\ xs \oplus \mathsf{tail}\ ys)$$

The specification is productive, since the corecursive call occurs directly under the stream constructor, which acts as a guard (shown underlined). 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.

These syntactic restrictions can be relaxed to allow conditional statements and `let` expressions [14], but despite such tricks primitive corecursion remains hopelessly primitive. The syntactic restriction for admissible corecursive definitions in Coq is more permissive

in that it allows for an arbitrary number of constructors to guard the corecursive calls, as in the following definition:

$$\mathsf{oneTwos} = \underline{\mathsf{SCons}}\ 1\ (\underline{\mathsf{SCons}}\ 2\ \mathsf{oneTwos})$$

Our framework achieves the same result by registering SCons as a friendly operation. Intuitively, an operation is *friendly* if it 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) of its arguments before producing an SCons. Because the operation preserves productivity, it can safely surround the guarding constructor.

The rigorous definition of friendliness will capture this intuition in a parametricity property that needs to be discharged, either by the user or automatically. In exchange, the framework yields a strengthened corecursor incorporating the new operation.

The constructor SCons is friendly, since it does not even need to inspect its arguments to produce a constructor. By contrast, the selector tail is not friendly—it must destruct two layers of constructors to produce one:

$$\mathsf{tail}\ xs = \underline{\mathsf{SCons}}\ (\mathsf{head}\ (\mathsf{tail}\ xs))\ (\mathsf{tail}\ (\mathsf{tail}\ xs))$$

The presence of unfriendly operations in the corecursive call context is enough to break productivity, as in the example

$$\mathsf{stallA} = \mathsf{SCons}\ 1\ (\mathsf{tail}\ \mathsf{stallA})$$

which stalls immediately after producing one constructor, leaving tail stallA unspecified.

Another instructive example is the function that keeps every other element in a stream:

$$\mathsf{everyOther}\ xs = \underline{\mathsf{SCons}}\ (\mathsf{head}\ xs)\ (\mathsf{everyOther}\ (\mathsf{tail}\ (\mathsf{tail}\ xs)))$$

The function is not friendly, despite being primitively corecursive. It also breaks productivity: The function

$$\mathsf{stallB} = \mathsf{SCons}\ 1\ (\mathsf{everyOther}\ \mathsf{stallB})$$

stalls after producing two constructors.

Going back to our first example, we observe that the operation $\oplus$ is friendly. Hence, it is allowed to participate in corecursive call contexts when defining new functions. In this respect, the framework is more permissive than Coq's built-in syntactic check. For example, we can define the stream of Fibonacci numbers in either of the following two ways:

$$\mathsf{fibA} = \underline{\mathsf{SCons}}\ 0\ (\mathsf{SCons}\ 1\ \mathsf{fibA} \oplus \mathsf{fibA})$$

$$\mathsf{fibB} = \underline{\mathsf{SCons}}\ 0\ (\mathsf{SCons}\ 1\ \mathsf{fibB}) \oplus \underline{\mathsf{SCons}}\ 0\ \mathsf{fibB}$$

Friendly operations are allowed to appear both under the constructor guard (as in fibA) and around it (as in fibB). Two guards are necessary in the second example—one for each branch of $\oplus$. Without rephrasing the specification, fibB cannot be expressed in Rutten's format of behavioral differential equations [57] or in Hinze's syntactic restriction [28], nor via Agda copatterns [5, 6].

Many useful operations are friendly and can therefore participate in further definitions. Following Rutten, the shuffle product $\otimes$ of two streams is defined in terms of $\oplus$. Shuffle product being itself friendly, we can employ it to define stream exponentiation, which also turns out to be friendly:

$$\begin{aligned} xs \otimes ys &= \underline{\mathsf{SCons}}\ (\mathsf{head}\ xs \times \mathsf{head}\ ys) \\ &\qquad ((xs \otimes \mathsf{tail}\ ys) \oplus (\mathsf{tail}\ xs \otimes ys)) \end{aligned}$$

$$\mathsf{exp}\ xs = \underline{\mathsf{SCons}}\ (2\ \hat{}\ \mathsf{head}\ xs)\ (\mathsf{tail}\ xs \otimes \mathsf{exp}\ xs)$$

Next, we use the defined and registered operations to specify two streams of factorials of natural numbers facA (starting at 1) and facB (starting at 0):

$$\mathsf{facA} = \underline{\mathsf{SCons}}\ 1\ \mathsf{facA} \otimes \underline{\mathsf{SCons}}\ 1\ \mathsf{facA}$$

$$\mathsf{facB} = \mathsf{exp}\ (\underline{\mathsf{SCons}}\ 0\ \mathsf{facB})$$

Computing the first few terms of facA manually should convince the reader that productivity and efficiency are not synonymous.

The arguments of friendly operations are not restricted to the Stream type. Let fimage give the image of a finite set under a function and $\bigsqcup X$ be the maximum of a finite set of naturals or 0 if X is empty. We can define the (friendly) supremum of a finite set of streams by primitive corecursion:

$$\mathsf{sup}\ X = \underline{\mathsf{SCons}}\ (\textstyle\bigsqcup\ (\mathsf{fimage}\ \mathsf{head}\ X))\ (\mathsf{sup}\ (\mathsf{fimage}\ \mathsf{tail}\ X))$$

2.2 Nested Corecursion Up-to

Although we use streams as our main example, the framework generally supports arbitrary codatatypes with multiple curried constructors and nesting through other type constructors. To demonstrate this last feature, we introduce the type of finitely branching Rose trees of potentially infinite depth with numeric labels:

$$\mathtt{codatatype}\ \mathsf{Tree} = \mathsf{Node}\ (\mathsf{val}\colon \mathsf{Nat})\ (\mathsf{sub}\colon \mathsf{List}\ \mathsf{Tree})$$

The type Tree has a single constructor $\mathsf{Node} : \mathsf{Nat} \to \mathsf{List}\ \mathsf{Tree} \to \mathsf{Tree}$ and two selectors $\mathsf{val} : \mathsf{Tree} \to \mathsf{Nat}$ and $\mathsf{sub} : \mathsf{Tree} \to \mathsf{List}\ \mathsf{Tree}$. The recursive occurrence of Tree is nested in the familiar polymorphic datatype of finite lists.

We first define the pointwise sum of two trees analogously to $\oplus$:

$$\begin{aligned} t \boxplus u = \underline{\mathsf{Node}}\ &(\mathsf{val}\ t + \mathsf{val}\ u) \\ &(\mathsf{map}\ (\lambda(t', u').\ t' \boxplus u')\ (\mathsf{zip}\ (\mathsf{sub}\ t)\ (\mathsf{sub}\ u))) \end{aligned}$$

Here, map is the standard map function on lists, and zip converts two parallel lists into a list of pairs, truncating the longer list if necessary. The operation $\boxplus$ is defined by primitive corecursion. Notice that the corecursive call is nested through map. This is a reflection of the target type, Tree, having its fixpoint definition nested through List. Moreover, by virtue of being friendly, $\boxplus$ can be used to define the shuffle product of trees:

$$\begin{aligned} t \boxtimes u = \underline{\mathsf{Node}}\ &(\mathsf{val}\ xs \times \mathsf{val}\ ys) \\ &(\mathsf{map}\ (\lambda(t', u').\ (t \boxtimes u') \boxplus (t' \boxtimes u))\ (\mathsf{zip}\ (\mathsf{sub}\ t)\ (\mathsf{sub}\ u))) \end{aligned}$$

The corecursive call takes place inside map, but also in the context of $\boxplus$. The specification of $\boxtimes$ is corecursive up-to (more precisely, up to $\boxplus$) and friendly.

2.3 Mixed Recursion–Corecursion

It is often convenient to let a corecursive function perform some finite computation before producing a constructor. With mixed recursion–corecursion, a finite number of unguarded recursive calls perform this calculation before reaching a guarded corecursive call.

The intuitive criterion for accepting such definitions is that the unguarded recursive call can be unfolded to arbitrary finite depth, ultimately yielding a purely corecursive definition. An example is the primes function taken from Di Gianantonio and Miculan [23]:

$$\begin{aligned} \mathsf{primes}\ m\ n = \mathtt{if}\ &(m = 0 \wedge n > 1) \vee \mathsf{gcd}\ m\ n = 1 \\ &\mathtt{then}\ \underline{\mathsf{SCons}}\ n\ (\mathsf{primes}\ (m \times n)\ (n+1)) \\ &\mathtt{else}\ \mathsf{primes}\ m\ (n+1) \end{aligned}$$

When called with $m = 1$ and $n = 2$, this function computes the stream of prime numbers. The unguarded call in the `else` branch increments its second argument n until it is coprime to the first argument m (i.e., the greatest common divisor of m and n is 1). For any positive integers m and n, the numbers m and $m \times n + 1$ are coprime, yielding an upper bound on the number of times n is increased. Hence, the function will take the `else` branch at most finitely often before taking the `then` branch and producing one constructor. There is a slight complication when $m = 0$ and $n > 1$: Without the first disjunct in the `if` condition, the function could stall. (This corner case was overlooked in the original example [23].)

Mixed recursion–corecursion makes the following (somewhat contrived) definition of factorials possible,

$$\begin{aligned}\mathsf{facC}\ n\ a\ i = {} & \mathtt{if}\ i = 0\ \mathtt{then}\ \underline{\mathsf{SCons}}\ a\ (\mathsf{facC}\ (n+1)\ 1\ (n+1))\\ & \mathtt{else}\ \mathsf{facC}\ n\ (a \times i)\ (i-1)\end{aligned}$$

The recursion in the `else` branch computes the next factorial by means of an accumulator a and a decreasing counter i. When the counter reaches 0, facC corecursively produces a constructor with the accumulated value and resets the accumulator and the counter.

Unguarded calls may even occur under friendly operations:

$$\begin{aligned}\mathsf{cat}\ n = {} & \mathtt{if}\ n > 0\ \mathtt{then}\ \mathsf{cat}\ (n-1) \oplus \underline{\mathsf{SCons}}\ 0\ (\mathsf{cat}\ (n+1))\\ & \mathtt{else}\ \underline{\mathsf{SCons}}\ 1\ (\mathsf{cat}\ 1)\end{aligned}$$

The call cat 1 computes the stream $C_1, C_2, \ldots$ of Catalan numbers, where $C_n = \frac{1}{n+1}\binom{2n}{n}$. This fact is far from obvious. Productivity is not entirely obvious either, but it is guaranteed by the framework.

When mixing recursion and corecursion, it is easy to get things wrong in the absence of solid foundations. Consider this specification, in which the corecursive call is guarded by SCons and the unguarded call's argument strictly decreases toward 0:

$$\begin{aligned}\mathsf{nasty}\ n = {} & \mathtt{if}\ n < 2\ \mathtt{then}\ \underline{\mathsf{SCons}}\ n\ (\mathsf{nasty}\ (n+1))\\ & \mathtt{else}\ \mathsf{inc}\ (\mathsf{tail}\ (\mathsf{nasty}\ (n-1)))\end{aligned}$$

Here, $\mathsf{inc} = \mathsf{smap}\ (\lambda x.\ x+1)$ and smap is the map function on streams. A simple calculation reveals that this specification is inconsistent because the tail selector before the unguarded call destructs the freshly produced constructor from the other branch:

$$\begin{aligned}\mathsf{nasty}\ 2 &= \mathsf{inc}\ (\mathsf{tail}\ (\mathsf{nasty}\ 1)) = \mathsf{inc}\ (\mathsf{tail}\ (\mathsf{SCons}\ 1\ (\mathsf{nasty}\ 2)))\\ &= \mathsf{inc}\ (\mathsf{nasty}\ 2)\end{aligned}$$

This is a close relative of the $\mathsf{f}\ x = \mathsf{f}\ x + 1$ example from the introduction. Our framework rejects this specification because the tail selector in the recursive call context is not friendly.

We conclude this subsection with a practical example from the literature. Given the polymorphic type

$$\mathtt{codatatype}\ \mathsf{LList}\ A = \mathsf{LNil} \mid \mathsf{LCons}\ (\mathsf{head}\colon A)\ (\mathsf{tail}\colon \mathsf{LList}\ A)$$

of lazy lists, the task is to define the function $\mathsf{lfilter} : (A \to \mathsf{Bool}) \to \mathsf{LList}\ A \to \mathsf{LList}\ A$ that filters out all elements failing to satisfy the given predicate. Thanks to the support for mixed recursion–corecursion, the framework turns what was for Lochbihler and Hölzl [44] a research problem into a routine exercise:

$$\begin{aligned}\mathsf{lfilter}\ P\ xs = {} & \mathtt{if}\ \forall x \in xs.\ \neg\ P\ x\\ & \quad \mathtt{then}\ \mathsf{LNil}\\ & \quad \mathtt{else\ if}\ P\ (\mathsf{head}\ xs)\\ & \quad\quad \mathtt{then}\ \underline{\mathsf{LCons}}\ (\mathsf{head}\ xs)\ (\mathsf{lfilter}\ P\ (\mathsf{tail}\ xs))\\ & \quad\quad \mathtt{else}\ \mathsf{lfilter}\ P\ (\mathsf{tail}\ xs)\end{aligned}$$

The first self-call is corecursive and guarded by LCons, whereas the second self-call is terminating, because the number of "false" elements until reaching the next "true" element (whose existence is guaranteed by the first `if` condition) decreases by one. In fact, in Isabelle the function can be introduced without proving termination of the second call by exploiting its tail-recursive nature [16, 36].

2.4 Coinduction Up-to

Once a corecursive specification has been accepted as productive, we normally want to reason about it. In proof assistants, codatatypes are accompanied by a notion of structural coinduction that matches primitively corecursive functions. For nonprimitive specifications, our framework provides the more advanced proof principle of coinduction up to congruence—or simply *coinduction up-to*.

The structural coinduction principle for streams is as follows:

$$\frac{R\ l\ r \quad \forall s\ t.\ R\ s\ t \longrightarrow \mathsf{head}\ s = \mathsf{head}\ t \wedge R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)}{l = r}$$

Coinduction allows us to prove an equality $l = r$ on streams by providing a relation R that relates l and r (first premise) and that constitutes a bisimulation (second premise). Streams that are related by a bisimulation cannot be distinguished by taking observations (via the selectors head and tail); hence they must be equal.

Creativity is generally required to instantiate R with a bisimulation. However, given a goal $l = r$, the following canonical candidate often works: $\lambda s\ t.\ \exists \overline{xs}.\ s = l \wedge t = r$, where $\overline{xs}$ are the variables occurring free in l or r. As a rehearsal, let us prove that the primitively corecursive operation $\oplus$ is commutative.

Proposition 1. $xs \oplus ys = ys \oplus xs$.

Proof. We first show that $R = \lambda s\ t.\ \exists xs\ ys.\ s = xs \oplus ys \wedge t = ys \oplus xs$ is a bisimulation. We fix two streams s and t for which we assume $R\ s\ t$ (i.e., there exist two streams xs and ys such that $s = xs \oplus ys$ and $t = ys \oplus xs$). Next, we show that $\mathsf{head}\ s = \mathsf{head}\ t$ and $R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)$. The first property is easy. For the second one:

$$\begin{aligned}& R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)\\ \longleftrightarrow\ & R\ (\mathsf{tail}\ (xs \oplus ys))\ (\mathsf{tail}\ (ys \oplus xs))\\ \longleftrightarrow\ & R\ (\mathsf{tail}\ xs \oplus \mathsf{tail}\ ys))\ (\mathsf{tail}\ ys \oplus \mathsf{tail}\ xs)\\ \longleftrightarrow\ & \exists xs'\ ys'.\ \mathsf{tail}\ xs \oplus \mathsf{tail}\ ys = xs' \oplus ys' \wedge\\ & \qquad \mathsf{tail}\ ys \oplus \mathsf{tail}\ xs = ys' \oplus xs'\end{aligned}$$

The last formula can be shown to hold by selecting $xs' = \mathsf{tail}\ xs$ and $ys' = \mathsf{tail}\ ys$. Moreover, $R\ (xs \oplus ys)\ (ys \oplus xs)$ holds. Therefore, the thesis follows by structural coinduction. □

If we attempt to prove the commutativity of $\otimes$ analogously, we eventually encounter a formula of the form $R\ (\cdots \oplus \cdots)\ (\cdots \oplus \cdots)$, because $\otimes$ is defined in terms of $\oplus$. Since R mentions only $\otimes$ but not $\oplus$, we are stuck. An ad hoc solution would be to replace the canonical R with a bisimulation that allows for descending under $\oplus$. However, this would be needed for almost every property about $\otimes$.

A more reusable solution is to strengthen the coinduction principle upon registration of a new friendly operation. The strengthening mirrors the acquired possibility of the new operation to appear in the corecursive call context. It is technically represented by a congruence closure $\mathsf{cl} : (\mathsf{Stream} \to \mathsf{Stream} \to \mathsf{Bool}) \to \mathsf{Stream} \to \mathsf{Stream} \to \mathsf{Bool}$. The coinduction up-to principle is almost identical to structural coinduction, except that the corecursive application of R is replaced by $\mathsf{cl}\ R$:

$$\frac{R\ l\ r \quad \forall s\ t.\ R\ s\ t \longrightarrow \mathsf{head}\ s = \mathsf{head}\ t \wedge \mathsf{cl}\ R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)}{l = r}$$

The principle evolves with every newly registered friendly operation in the sense that our framework refines the definition of the congruence closure cl. (Strictly speaking, a fresh symbol cl' is introduced each time.) For example, after registering SCons and $\oplus$, $\mathsf{cl}\ R$ is the least reflexive, symmetric, transitive relation containing R and satisfying the rules

$$\frac{x = y \quad \mathsf{cl}\ R\ xs\ ys}{\mathsf{cl}\ R\ (\mathsf{SCons}\ x\ xs)\ (\mathsf{SCons}\ y\ ys)} \qquad \frac{\mathsf{cl}\ R\ xs\ ys \quad \mathsf{cl}\ R\ xs'\ ys'}{\mathsf{cl}\ R\ (xs \oplus xs')\ (ys \oplus ys')}$$

After defining and registering $\otimes$, the relation $\mathsf{cl}\ R$ is extended to also satisfy

$$\frac{\mathsf{cl}\ R\ xs\ ys \quad \mathsf{cl}\ R\ xs'\ ys'}{\mathsf{cl}\ R\ (xs \otimes xs')\ (ys \otimes ys')}$$

Let us apply the strengthened coinduction principle to prove the distributivity of stream exponentiation over pointwise addition:

Proposition 2. $\mathsf{exp}\ (xs \oplus ys) = \mathsf{exp}\ xs \otimes \mathsf{exp}\ ys$.

Proof. We first show that $R = \lambda s\ t.\ \exists xs\ ys.\ s = \mathsf{exp}\ (xs \oplus ys) \wedge t = \mathsf{exp}\ xs \otimes \mathsf{exp}\ ys$ is a bisimulation. We fix two streams s and t for which we assume $R\ s\ t$ (i.e., there exist two streams xs and ys such

that $s = \mathsf{exp}\ (xs \oplus ys)$ and $t = \mathsf{exp}\ xs \otimes \mathsf{exp}\ ys$). Next, we show that $\mathsf{head}\ s = \mathsf{head}\ t$ and $\mathsf{cl}\ R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)$:

$$\begin{aligned}
&\mathsf{head}\ s = \mathsf{head}\ (\mathsf{exp}\ (xs \oplus ys)) = 2\,\hat{}\,\mathsf{head}\ (xs \oplus ys)\\
&\quad = 2\,\hat{}\,(\mathsf{head}\ xs + \mathsf{head}\ ys) = 2\,\hat{}\,\mathsf{head}\ xs \times 2\,\hat{}\,\mathsf{head}\ ys\\
&\quad = \mathsf{head}\ (\mathsf{exp}\ xs) \times \mathsf{head}\ (\mathsf{exp}\ ys)\\
&\quad = \mathsf{head}\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys) = \mathsf{head}\ t
\end{aligned}$$

$$\begin{aligned}
&\mathsf{cl}\ R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)\\
&\leftrightarrow \mathsf{cl}\ R\ (\mathsf{tail}\ (\mathsf{exp}\ (xs \oplus ys)))\ (\mathsf{tail}\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys))\\
&\leftrightarrow \mathsf{cl}\ R\ ((\mathsf{tail}\ xs \oplus \mathsf{tail}\ ys) \otimes \mathsf{exp}\ (xs \oplus ys))\\
&\quad (\mathsf{exp}\ xs \otimes (\mathsf{tail}\ ys \otimes \mathsf{exp}\ ys) \oplus (\mathsf{tail}\ xs \otimes \mathsf{exp}\ xs) \otimes \mathsf{exp}\ ys)\\
&\overset{*}{\leftrightarrow} \mathsf{cl}\ R\ ((\mathsf{tail}\ xs \otimes \mathsf{exp}\ (xs \oplus ys) \oplus \mathsf{tail}\ ys \otimes \mathsf{exp}\ (xs \oplus ys))\\
&\quad {}_{\oplus}\ (\mathsf{tail}\ xs \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys) \oplus \mathsf{tail}\ ys \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)\\
&\overset{\oplus}{\leftarrow} \mathsf{cl}\ R\ (\mathsf{tail}\ xs \otimes \mathsf{exp}\ (xs \oplus ys))\ (\mathsf{tail}\ xs \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)) \wedge\\
&\quad {}_{\otimes}\ \mathsf{cl}\ R\ (\mathsf{tail}\ ys \otimes \mathsf{exp}\ (xs \oplus ys))\ (\mathsf{tail}\ ys \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys))\\
&\overset{\otimes}{\leftarrow} \mathsf{cl}\ R\ (\mathsf{tail}\ xs)\ (\mathsf{tail}\ xs) \wedge \mathsf{cl}\ R\ (\mathsf{tail}\ ys)\ (\mathsf{tail}\ ys) \wedge\\
&\quad \mathsf{cl}\ R\ (\mathsf{exp}\ (xs \oplus ys))\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)\\
&\leftarrow R\ (\mathsf{exp}\ (xs \oplus ys))\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)
\end{aligned}$$

The step marked with $*$ appeals to associativity and commutativity of $\oplus$ and $\otimes$ as well as distributivity of $\otimes$ over $\oplus$. These properties are likewise proved by coinduction up-to. The implications marked with $\oplus$ and $\otimes$ are justified by the respective congruence rules. The last implication uses reflexivity and expands R to its closure $\mathsf{cl}\ R$.

Finally, it is easy to see that $R\ (\mathsf{exp}\ (xs \oplus ys))\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)$ holds. Therefore, the thesis follows by coinduction up-to. □

The formalization accompanying this paper [16] also contains proofs of $\mathsf{facA} = \mathsf{facC}\ 1\ 1\ 1 = \mathsf{smap}\ \mathsf{fac}\ (\mathsf{natsFrom}\ 1)$, $\mathsf{facB} = \mathsf{SCons}\ 1\ \mathsf{facA}$, and $\mathsf{fibA} = \mathsf{fibB}$, where fac is the factorial on Nat.

Nested corecursion up-to is also reflected with a suitable strengthened coinduction rule. For Tree, this strengthening takes place under the rel operator on list, similarly to the corecursive calls occurring nested in the map function:

$$\frac{R\ l\ r \quad \forall s\ t.\ R\ s\ t \longrightarrow \mathsf{val}\ s = \mathsf{val}\ t \wedge \mathsf{rel}\ (\mathsf{cl}\ R)\ (\mathsf{sub}\ s)\ (\mathsf{sub}\ t)}{l = r}$$

The rel R operator lifts the binary predicate $R : A \to B \to \mathsf{Bool}$ to a predicate $\mathsf{List}\ A \to \mathsf{List}\ B \to \mathsf{Bool}$. More precisely, $\mathsf{rel}\ R\ xs\ ys$ holds if and only if xs and ys have the same length and parallel elements of xs and ys are related by R. This nested coinduction rule is convenient provided there is some infrastructure to descend under rel (as is the case in Isabelle/HOL). The formalization establishes several arithmetic properties of $\boxplus$ and $\boxtimes$.

3. Extensible Corecursors

We now describe the definitional and proof mechanisms that substantiate flexible corecursive definitions in the style of Section 2. They are based on the modular maintenance of infrastructure for the corecursor associated with a codatatype, with the possibility of open-ended incremental improvement. We present the approach for an arbitrary codatatype given as the greatest fixpoint of a (bounded) functor. The approach is quite general and does not rely on any particular grammar for specifying codatatypes.

Extensibility is an integral feature of the framework. In principle, an implementation could redo the constructions from scratch each time a friendly operation is registered, but it would give rise to a quadratic number of definitions, slowing down the proof assistant. The incremental approach is also more flexible and future-proof, allowing mixed fixpoints and composition with other (co)recursors.

3.1 Functors and Relators

Functional programming languages and proof assistants necessarily maintain a database of the user-defined types or, more generally, type constructors, which can be thought as functions $\mathsf{F} : \mathsf{Set}^n \to \mathsf{Set}$ on the class of sets (or perhaps of ordered sets). It is often useful to maintain more structure along with these type constructors:

- a functorial action $\mathsf{Fmap} : \prod_{\overline{A},\overline{B} \in \mathsf{Set}^n} \prod_{i=1}^{n} (A_i \to B_i) \to \mathsf{F}\ \overline{A} \to \mathsf{F}\ \overline{B}$, i.e., a polymorphic function of the indicated type that commutes with identity $\mathsf{id}_A : A \to A$ and composition;
- a relator $\mathsf{Frel} : \prod_{\overline{A},\overline{B} \in \mathsf{Set}^n} \prod_{i=1}^{n} (A_i \to B_i \to \mathsf{Bool}) \to \mathsf{F}\ \overline{A} \to \mathsf{F}\ \overline{B} \to \mathsf{Bool}$, i.e., a polymorphic function of the indicated type that commutes with binary-relation identity and composition.

Following standard notation from category theory, we write F instead of Fmap. Given binary relations $R_i : A_i \to B_i \to \mathsf{Bool}$ for $1 \le i \le n$, we think of $\mathsf{Frel}\ \overline{R} : \mathsf{F}\ \overline{A} \to \mathsf{F}\ \overline{B} \to \mathsf{Bool}$ as the natural lifting of R along F; for example, if F is List (and $n = 1$), Frel lifts a relation on elements to the componentwise relation on lists, defined conjunctively, and also requiring equal lengths. It is well known that the positive type constructors defined by standard means (basic types, composition, least and greatest fixpoints) have canonical functorial and relator structure. This is crucial for the foundational construction of user-specified (co)datatypes in Isabelle/HOL [60].

But even nonpositive type constructors $\mathsf{G} : \mathsf{Set}^n \to \mathsf{Set}$ exhibit a relator-like structure

$$\mathsf{Grel} : \textstyle\prod_{\overline{A},\overline{B} \in \mathsf{Set}^n} (\overline{A} \to \overline{B} \to \mathsf{Bool}) \to (\mathsf{G}\ \overline{A} \to \mathsf{G}\ \overline{B} \to \mathsf{Bool})$$

(which need not commute with relation composition, though). Above, $\overline{A} \to \overline{B} \to \mathsf{Bool}$ consists of tuples $(R_i : A_i \to B_i \to \mathsf{Bool})_{i \in \overline{1,n}}$ of relations, where $\overline{A} = (A_i)_{i \in \overline{1,n}}$ and $\overline{B} = (B_i)_{i \in \overline{1,n}}$. For example, if $\mathsf{G} : \mathsf{Set}^2 \to \mathsf{Set}$ is the function space constructor $\mathsf{G}\ (A_1, A_2) = A_1 \to A_2$ and $f \in \mathsf{G}\ (A_1, A_2)$, $g \in \mathsf{G}\ (B_1, B_2)$, $R_1 : A_1 \to B_1 \to \mathsf{Bool}$, and $R_2 : A_2 \to B_2 \to \mathsf{Bool}$, then $\mathsf{Grel}\ R_1\ R_2\ f\ g$ is defined as $\forall a_1 \in A_1.\ \forall b_1 \in B_1.\ R_1\ a_1\ b_1 \longrightarrow R_2\ (f\ a_1)\ (g\ b_1)$.

A polymorphic function $c : \prod_{\overline{A} \in \mathsf{Set}^n} \mathsf{G}\ \overline{A}$ is called *parametric* [52, 63] if

$$\forall \overline{A}\ \overline{B} \in \mathsf{Set}^n.\ \forall \overline{R} : \overline{A} \to \overline{B} \to \mathsf{Bool}.\ \ \mathsf{Grel}\ \overline{R}\ c_{\overline{A}}\ c_{\overline{B}}$$

The maintenance of relator-like structures is helpful for automating theorem transfer along isomorphisms and quotients [31]. Here we explore an additional benefit of maintaining functorial and relator structure for type constructors: the possibility to extend the corecursor in reaction to user input.

We assume that all the considered type constructors are both functors and relators, that they include basic functors such as identity, constant, sum, and product, and that they are closed under least and greatest fixpoints (initial algebras and final coalgebras). Examples of such classes of type constructors are the datafunctors [26], the containers [1], and the bounded natural functors [60].

We focus on the case of a unary codatatype-generating functor $\mathsf{F} : \mathsf{Set} \to \mathsf{Set}$. The codatatype of interest will be its greatest fixpoint (or final coalgebra) $\mathsf{J} = \mathsf{gfp}\ \mathsf{F}$. This generic situation already covers the vast majority of interesting codatatypes, since F can represent arbitrarily complex nesting. For example, if $\mathsf{F} = \lambda A.\ \mathsf{Nat} \times \mathsf{List}\ A$, then J corresponds to the Tree codatatype introduced in Section 2.2. The extension to mutually defined codatatypes is straightforward but tedious. Our examples will take J to be the Stream type from Section 2, with $\mathsf{F} = \lambda A.\ \mathsf{Nat} \times A$.

Given a set A, it will be useful to think of the elements $x \in \mathsf{F}\ A$ as consisting of a *shape* together with *content* that fills the shape with elements of A, as suggested by Figure 1. If $\mathsf{F}\ A = \mathsf{Nat} \times A$, the shape of $x = (n, a)$ is $(n, \_)$ and the content is a; if $\mathsf{F}\ A = \mathsf{List}\ A$, the shape of $x = [x_1, \ldots, x_n]$ is the n-slot container $[\_, \ldots, \_]$ and the content consists of the x_i's. According to this view, for each $f : A \to B$, F's functorial action sends any x to an element $\mathsf{F}\ f\ x$ of the same shape as x but with each content item a replaced by $f\ a$. Technically, this view can be supported by custom notions such as containers [1] or, more simply, via a parametric function of type $\prod_{A \in \mathsf{Set}} \mathsf{F}\ A \to \mathsf{Set}\ A$ that collects the content elements [60].

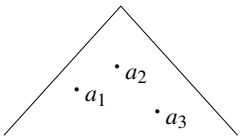


Figure 1: An element x of $\mathsf{F}\ A$ with content items a_1, a_2, a_3

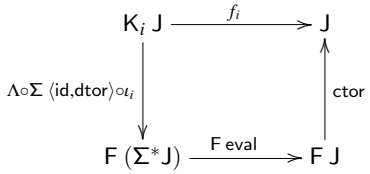


Figure 2: The friendliness condition

3.2 Primitive Corecursion

The `codatatype` command that defines J introduces the constructor and destructor bijections $\mathsf{ctor} : \mathsf{F}\ \mathsf{J} \to \mathsf{J}$ and $\mathsf{dtor} : \mathsf{J} \to \mathsf{F}\ \mathsf{J}$ and the primitive corecursor $\mathsf{corecPrim} : \prod_{A \in \mathsf{Set}} (A \to \mathsf{F}\ A) \to A \to \mathsf{J}$ satisfying $\mathsf{corecPrim}\ s\ a = \mathsf{ctor}\ (\mathsf{F}\ (\mathsf{corecPrim}\ s)\ (s\ a))$. In elements $x \in \mathsf{F}\ A$, the occurrences of content items $a \in A$ in the shape of x captures the positioning of the corecursive calls.

Example 3. Modulo currying, the pointwise sum of streams $\oplus$ is definable as $\mathsf{corecPrim}\ s$, by taking $s : \mathsf{Stream}^2 \to \mathsf{Nat} \times \mathsf{Stream}^2$ to be $\lambda(xs, ys).\ (\mathsf{head}\ xs + \mathsf{head}\ ys, (\mathsf{tail}\ xs, \mathsf{tail}\ ys))$.

In Example 3 and elsewhere, we lighten notation by identifying curried and uncurried functions, counting on implicit coercions.

3.3 The Corecursion State

Given a functor $\Sigma : \mathsf{Set} \to \mathsf{Set}$, we define Σ^*, the *free-monad functor* over $\lambda B.\ \mathsf{J} + \Sigma\ B$, by

$$\Sigma^* A = \mathsf{lfp}\ (\lambda B.\ A + \mathsf{J} + \Sigma\ B)$$

We write $\mathsf{vleaf} : A \to \Sigma^* A$, $\mathsf{cleaf} : \mathsf{J} \to \Sigma^* A$, and $\mathsf{op} : \Sigma\ (\Sigma^* A) \to \Sigma^* A$ for the first, second, and third injections into $\Sigma^* A$. These functions are in fact polymorphic; for example, vleaf has type $\prod_{A \in \mathsf{Set}} A \to \Sigma^* A$. We omit the set parameters of polymorphic functions since they can be inferred from the arguments.

At any given moment, we maintain the following data associated with J, which we call a *corecursion state*:

- a finite number of functors $\mathsf{K}_1, \ldots, \mathsf{K}_n : \mathsf{Set} \to \mathsf{Set}$ and, for each K_i, a function $f_i : \mathsf{K}_i\ \mathsf{J} \to \mathsf{J}$;
- a polymorphic function $\Lambda : \prod_{A \in \mathsf{Set}} \Sigma\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\Sigma^* A)$.

We call the f_i's the *friendly operations* and define their collective *signature functor* Σ by

$$\Sigma\ A = \mathsf{K}_1\ A + \cdots + \mathsf{K}_n\ A$$

where $\iota_i : \mathsf{K}_i \to \Sigma$ is the standard embedding of K_i into Σ. We call Λ the *corecursor seed*.

The corecursion state is subject to the following conditions:

Parametricity: Λ is parametric.

Friendliness: Each f_i satisfies the characteristic equation

$$f_i\ x = \mathsf{ctor}\ (\mathsf{F}\ \mathsf{eval}\ (\Lambda\ (\Sigma\ \langle \mathsf{id}, \mathsf{dtor} \rangle\ (\iota_i\ x))))$$

The convolution operator $\langle \_, \_ \rangle$ builds a function $\langle f, g \rangle : B \to C \times D$ from two functions $f : B \to C$ and $g : B \to D$, and $\mathsf{eval} : \Sigma^* \mathsf{J} \to \mathsf{J}$ is the canonical evaluation function defined recursively (using the primitive recursor associated with Σ^*):

$$\begin{array}{l} \mathsf{eval}\ (\mathsf{vleaf}\ j) = j \\ \mathsf{eval}\ (\mathsf{cleaf}\ j) = j \\ \mathsf{eval}\ (\mathsf{op}\ z) = \mathsf{case}\ z\ \mathsf{of}\ \iota_i\ t \Rightarrow f_i\ (\mathsf{K}_i\ \mathsf{eval}\ t) \end{array}$$

Notice that eval is applied recursively to t by lifting it through the functor K_i. Functions having the type of Λ and assumed parametric (or, equivalently, assumed to be natural transformations) are known in category theory as abstract GSOS rules. They were introduced by Turi and Plotkin [61] and further studied by Bartels [9], Jacobs [33], Hinze and James [29], Milius et al. [46], and others.

Thus, a corecursion state is a triple $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$. As we will see in Section 3.6, the state evolves as users define and register new functions. The f_i's are the operations that have been registered as potential participants in corecursive call contexts. Since f_i has type $\mathsf{K}_i\ \mathsf{J} \to \mathsf{J}$, we think of K_i as encoding the arity of f_i. Then Σ, the sum of the K_i's, represents the signature consisting of all the f_i's. Thus, for each A, $\Sigma^* A$ represents the set of formal expressions over Σ and $A + \mathsf{J}$, i.e., the trees built starting from two kinds of leaves—"variables" in A and "constants" in J—by applying operation symbols corresponding to the f_i's. Finally, eval evaluates in J the formal expressions of $\Sigma^* \mathsf{J}$ by recursively applying the functions f_i.

If the functors K_i are restricted to be finite monomials $\lambda A.\ A^{k_i}$, the functor Σ can be seen as a standard algebraic signature and $(\Sigma^* A, \mathsf{op})$ as the standard term algebra for this signature, over the variables A and the constants J. However, we allow K_i to be more exotic; for example, $\mathsf{K}_i\ A$ can be A^{Nat} (representing an infinitary operation) or one of $\mathsf{List}\ A$ and $\mathsf{FinSet}\ A$ (representing an operation taking a varying finite number of ordered or unordered arguments).

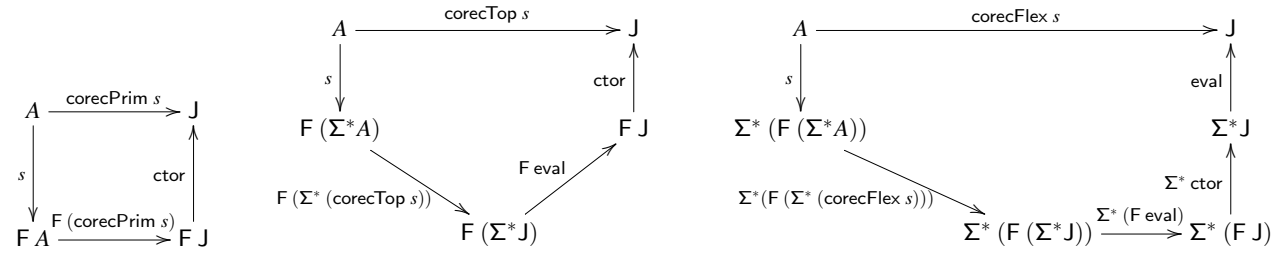
But what guarantees that the f_i's are indeed safe as contexts for corecursive calls? In particular, how can the framework exclude tail while allowing SCons, $\oplus$, and $\otimes$? This is where the parametricity and friendliness conditions on the state enter the picture.

We start with friendliness. Assume $x \in \mathsf{K}_i\ \mathsf{J}$, which is unambiguously represented in $\Sigma\ \mathsf{J}$ as $\iota_i\ x$. Let $j_1, \ldots, j_m \in \mathsf{J}$ be the content items of $\iota_i\ x$ (placed in various slots in the shape of x). To evaluate f_i on x, we first corecursively unfold the j_l's while also keeping the originals, thus replacing each j_l with $(j_l, \mathsf{dtor}\ j_l)$. Then we apply the transformation Λ to obtain an element of $\mathsf{F}\ (\Sigma^* \mathsf{J})$, which has an F-shape at the top (the first produced observable data) and for each slot in this shape an element of $\Sigma^* \mathsf{J}$, i.e., a formal-expression tree having leaves in J and built using operation symbols from the signature (the corecursive continuation):

$$\mathsf{K}_i\ \mathsf{J} \xrightarrow{\iota_i} \Sigma\ \mathsf{J} \xrightarrow{\Sigma\ \langle \mathsf{id}, \mathsf{dtor} \rangle} \Sigma\ (\mathsf{J} \times \mathsf{F}\ \mathsf{J}) \xrightarrow{\Lambda} \mathsf{F}\ (\Sigma^* \mathsf{J})$$

Next, we evaluate the formal expressions (from $\Sigma^* \mathsf{J}$) located in the slots. This is achieved by applying eval, which corecursively calls the f_i's under the functor F. Finally, the result (an element of $\mathsf{F}\ \mathsf{J}$) is guarded with ctor. In summary, Λ is a schematic representation of the mutually corecursive behavior of the friendly operations up to the production of the first observable data. This intuition is captured by the friendliness condition, which states that the diagram in Figure 2 commutes for each f_i. (If we preferred the destructor view, we could replace the right upward ctor arrow with a downward dtor arrow without changing the diagram's meaning.)

It suffices to peel off one layer of the arguments j_i (by applying dtor) for a friendly operation f_i to produce, via Λ, one layer of the result and to delegate the rest of the computation to a context consisting of a combination of friendly operations (an element of $\Sigma^* \mathsf{J}$). But how can we formally express that exploring one layer is enough, i.e., that applying $\Lambda : \mathsf{J} \times \mathsf{F}\ \mathsf{J} \to \mathsf{F}\ (\Sigma^* \mathsf{J})$ to $(j_i, \mathsf{dtor}\ j_i)$ does not leed to a deeper exploration? An elegant way of capturing this is to require that Λ, a polymorphic function, operates without



(a) Primitive corecursion (b) Top-guarded corecursion up-to (c) Flexibly guarded corecursion up-to

Figure 3: The corecursors

analyzing J, i.e., that it operates in the same way on $A \times \mathsf{F}\,A \to \mathsf{F}\,(\Sigma^* A)$ for any set A. This requirement is precisely parametricity.

Strictly speaking, the friendly operations $\overline{f}$ are a redundant piece of data in the state $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$, since, assuming Λ parametric, we can prove that there exists a unique tuple $\overline{f}$ that satisfies the friendliness condition. Hence, in principle, the operations $\overline{f}$ could be derived on a per-need basis. However, in the context of proof assistants, these operations must be available as part of the state, since a user will directly formulate their corecursive definitions in terms of these operations.

Example 4. Let $\mathsf{J} = \mathsf{Stream}$ and assume that $\mathsf{SCons} : \mathsf{Nat} \times \mathsf{Stream} \to \mathsf{Stream}$ and $\oplus : \mathsf{Stream}^2 \to \mathsf{Stream}$ are the only friendly operations registered so far. Then $\mathsf{K}_1 = \lambda B.\ \mathsf{Nat} \times B$, $f_1 = \mathsf{SCons}$, $\mathsf{K}_2 = \lambda B.\ B^2$, and $f_2 = \oplus$. Moreover, $\Sigma^* A = \mathsf{lfp}\,(\lambda B.\ A + \mathsf{Stream} + (\mathsf{Nat} \times B + B^2))$ consists of formal-expression trees with leaves in A and Stream and built using arity-correct applications of operation symbols corresponding to SCons and $\oplus$, written $\boxed{\mathsf{SCons}}$ and $\boxed{\oplus}$. Given $n \in \mathsf{Nat}$ and $a,\ b \in A$, an example of such a tree is $\mathsf{vleaf}\ a\ \boxed{\oplus}\ \boxed{\mathsf{SCons}}\ (n, \mathsf{vleaf}\ a\ \boxed{\oplus}\ \mathsf{vleaf}\ b)$. If additionally $A = \mathsf{Stream}$, then eval applied to this tree is $a \oplus \mathsf{SCons}\ n\ (a \oplus b)$.

But what is Λ? As we show below, we need not worry about the global definition of Λ, since both Σ and Λ will be updated incrementally when registering new operations as friendly. Nonetheless, a global definition of Λ for SCons and $\oplus$ follows:

$$\begin{array}{l}\Lambda\ z = \mathsf{case}\ z\ \mathsf{of}\\ \quad \iota_1\ (n, (a, (m, a'))) \Rightarrow (n, \boxed{\mathsf{SCons}}\ (m, \mathsf{vleaf}\ a'))\\ \quad \iota_2\ ((a, (m, a')),\ (b, (n, b'))) \Rightarrow (m+n, \mathsf{vleaf}\ a'\ \boxed{\oplus}\ \mathsf{vleaf}\ b')\end{array}$$

Informally, SCons and $\oplus$ exhibit the following behaviors:

- to evaluate SCons on a number n and an item a with $(\mathsf{head}\ a, \mathsf{tail}\ a) = (m, a')$, produce n and evaluate SCons on m and a', i.e., output $\mathsf{SCons}\ n\ (\mathsf{SCons}\ m\ a') = \mathsf{SCons}\ n\ a$;
- to evaluate $\oplus$ on a, b with $(\mathsf{head}\ a, \mathsf{tail}\ a) = (m, a')$ and $(\mathsf{head}\ b, \mathsf{tail}\ b) = (n, b')$, produce $m+n$ and evaluate $\oplus$ on a' and b', i.e., output $\mathsf{SCons}\ (m+n)\ (a' \oplus b')$.

A natural question at this point is why we need "constant" leaves $\mathsf{cleaf}\ j$ in $\Sigma^* A$, given that the eval function is defined on $\Sigma^* \mathsf{J}$ and operates on $\mathsf{cleaf}\ j$ in the same way as it does on $\mathsf{vleaf}\ j$. The answer is that constant leaves allow Λ to produce results that concretely refer to J, which offers greater flexibility. To illustrate this, let us change our operation $\oplus$ by replacing, in the right-hand side of its corecursive call, the argument tail *ys* with a fixed stream, oneTwos:

$$xs \oplus ys = \boxed{\mathsf{SCons}}\ (\mathsf{head}\ xs + \mathsf{head}\ ys)\ (\mathsf{tail}\ xs \oplus \mathsf{oneTwos})$$

To capture this as friendly, we need to change the ι_2 case of Λ to $(m+n, \mathsf{vleaf}\ a'\ \boxed{\oplus}\ \mathsf{cleaf}\ \mathsf{oneTwos})$. One could achieve the above by registering the constant operation oneTwos as friendly. However, we want all elements of J to be a priori registered as friendly. This is precisely what cleaf offers.

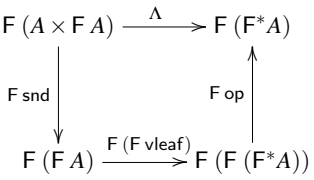


Figure 4: Definition of Λ for the initial state

3.4 Corecursion Up-to

A corecursion state $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$ for an F-defined codatatype J consists of a collection of operations $f_i : \mathsf{K}_i\ \mathsf{J} \to \mathsf{J}$ that satisfy the friendliness properties expressed in terms of a parametric function Λ. We are now ready to harvest the crop of this setting: a corecursion principle for defining functions having J as codomain.

The principle will be represented by two corecursors, corecTop and corecFlex. Although subsumed by the latter, the former is interesting in its own right and will give us the opportunity to illustrate some fine points. Below we list the types of these corecursors along with that of the primitive corecursor for comparison:

Primitive corecursor:
$\mathsf{corecPrim} : \prod_{A \in \mathsf{Set}} (A \to \mathsf{F}\,A) \to A \to \mathsf{J}$

Top-guarded corecursor up-to:
$\mathsf{corecTop} : \prod_{A \in \mathsf{Set}} (A \to \mathsf{F}\,(\Sigma^* A)) \to A \to \mathsf{J}$

Flexibly guarded corecursor up-to:
$\mathsf{corecFlex} : \prod_{A \in \mathsf{Set}} (A \to \Sigma^*\,(\mathsf{F}\,(\Sigma^* A))) \to A \to \mathsf{J}$

Figure 3 presents the diagrams whose commutativity properties give the characteristic equations of these corecursors.

Each corecursor implements a contract of the following form: If, for each $a \in A$, one provides the intended corecursive behavior of $g\ a$ represented as $s\ a$, where s is a function from A, one obtains the function $g : A \to \mathsf{J}$ (as the corresponding corecursor applied to s) satisfying a suitable fixpoint equation matching this behavior.

The codomain of s is the key to understanding the expressiveness of each corecursor. The intended corecursive calls are represented by A, and the call context is represented by the surrounding combination of functors (involving F, Σ^*, or both):

- for corecPrim, the allowed call contexts consist of a single constructor guard (represented by F);
- for corecTop, they consist of a constructor guard (represented by F) followed by any combination of friendly operations f_i (represented by Σ^*);
- for corecFlex, they consist of any combination of friendly operations satisfying the condition that on every path leading

to a corecursive call there exists at least one constructor guard (represented by $\Sigma^*\ (\mathsf{F}\ (\Sigma^*\_))$).

We can see the computation of $g\ a$ by following the diagrams in Figure 3 counterclockwise from their left-top corners. The application $s\ a$ first builds the call context syntactically. Then g is applied corecursively on the leaves. Finally, the call context is evaluated: For corecPrim, it consist only of the guard (ctor); for corecTop, it involves the evaluation of the friendly operations (which may also include several occurrences of the guard) and ends with the evaluation of the top guard; for corecFlex, the evaluation of the guard is interspersed with that of the other friendly operations.

Example 5. For each example from Section 2.1, we give the corecursors that can handle it (assuming the necessary friendly operations were registered):

$\oplus$, everyOther:	corecFlex, corecTop, corecPrim
oneTwos, fibA, $\otimes$, exp, sup:	corecFlex, corecTop
fibB, facA, facB:	corecFlex

(Note that everyOther is definable in our framework, but is not friendly, meaning that it cannot participate in call contexts of other corecursive definitions.) With the usual identification of $\mathsf{Unit} \to \mathsf{J}$ and J, we can define fibA and facA as

$$\mathsf{corecTop}\ (\lambda u : \mathsf{Unit}.\ (0, \boxed{\mathsf{SCons}}\ (1, \mathsf{vleaf}\ u)\ \boxed{\oplus}\ \mathsf{vleaf}\ u))$$
$$\mathsf{corecFlex}\ (\lambda u : \mathsf{Unit}.\ \mathsf{vleaf}\ (1, \mathsf{vleaf}\ u)\ \boxed{\otimes}\ \mathsf{vleaf}\ (1, \mathsf{vleaf}\ u))$$

Let us compare fibA's specification $\mathsf{fibA} = \mathsf{SCons}\ 0\ (\mathsf{SCons}\ 1\ \mathsf{fibA} \oplus \mathsf{fibA})$ with its definition in terms of corecTop. The outer SCons guard (with 0 as first argument) corresponds to the outer pair $(0, \_)$. The inner SCons and $\oplus$ are interpreted as friendly operations and represented by the symbols $\boxed{\mathsf{SCons}}$ and $\boxed{\oplus}$ (cf. Example 4). Finally, the corecursive calls of fibA are captured by vleaf u.

The desired specification can be obtained from the corecTop form by the characteristic equation of corecTop (for $A = \mathsf{Unit}$) and the properties of eval as follows, where we simply write s, fibA, and vleaf for their applications to the unique element $()$ of Unit, namely $s\ ()$, fibA $()$, and vleaf $()$:

$$\begin{aligned}
&\mathsf{fibA}\\
=\ &\{\text{by the commutativity of Figure 3b, with } \mathsf{fibA} = \mathsf{corecTop}\ s\}\\
&\mathsf{ctor}\ (\mathsf{F}\ (\mathsf{eval} \circ \Sigma^*\ \mathsf{fibA})\ s)\\
=\ &\{\text{by the definitions of F and } s\}\\
&\mathsf{SCons}\ 0\ ((\mathsf{eval} \circ \Sigma^*\ \mathsf{fibA})\ (\boxed{\mathsf{SCons}}\ (1, \mathsf{vleaf})\ \boxed{\oplus}\ (\mathsf{vleaf})))\\
=\ &\{\text{by the definition of } \Sigma^*\}\\
&\mathsf{SCons}\ 0\ (\mathsf{eval}\ (\boxed{\mathsf{SCons}}\ (1, \mathsf{vleaf}\ \mathsf{fibA})\ \boxed{\oplus}\ (\mathsf{vleaf}\ \mathsf{fibA}))\\
=\ &\{\text{by the definition of eval}\}\\
&\mathsf{SCons}\ 0\ (\mathsf{SCons}\ 1\ \mathsf{fibA} \oplus \mathsf{fibA})
\end{aligned}$$

The elimination of the corecTop infrastructure relies on simplification rules for the involved operators and can be fully automatized.

Parametricity and friendliness are crucial for proving that the corecursors actually exist:

Theorem 6. There exist the polymorphic functions corecTop and corecFlex making the diagrams in Figures 3b and 3c commute. Moreover, for each s of appropriate type, corecTop s or corecFlex s is the unique function making its diagram commute.

Theorem 6 is a known result from the category theory literature: The corecTop s version follows from the results in Bartels's thesis [10], whereas the corecFlex s version was recently (and independently) proved by Milius et al. [46, Theorem 2.16].

3.5 Initializing the Corecursion State

The simplest relaxation of primitive corecursion is the allowance of multiple constructors in the call context, in the style of Coq, as in the definition of oneTwos (Section 2.1). Since this idea is independent of the choice of codatatype J, we realize it when bootstrapping the corecursion state. Upon defining a codatatype J, we take the following initial corecursion state $\mathsf{initState} = (\overline{\mathsf{K}}, \overline{f}, \Lambda)$:

- $\overline{K}$ is a singleton consisting of (a copy of) F;
- $\overline{f}$ is a singleton consisting of ctor;
- $\Lambda : \prod_{A \in \mathsf{Set}} \mathsf{F}\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\mathsf{F}^*\ A)$ is defined as $\mathsf{F}\ (\mathsf{op} \circ \mathsf{F}\ \mathsf{vleaf} \circ \mathsf{snd})$, where snd is the second product projection.

Recall that the seed Λ is designed to schematically represent the corecursive behavior of the registered operations by describing how they produce one layer of observable data. The definition in Figure 4 depicts this for ctor and instantiates to the schematic behavior of SCons presented at the end of Example 4.

Theorem 7. initState is a well-formed corecursion state—i.e., it satisfies parametricity and friendliness.

3.6 Advancing the Corecursion State

The role of a corecursion state $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$ for J is to provide infrastructure for flexible corecursive definitions of functions g between arbitrary sets A and J. If nothing else is known about A, this is the end of the story. However, assume that J is a component of A, in that A is constructed from J (possibly along with other components). For example, A could be List J, or $\mathsf{J} \times (\mathsf{Nat} \to \mathsf{List}\ \mathsf{J})$. We capture this abstractly by assuming $A = \mathsf{K}\ \mathsf{J}$ for some functor K.

In this case, we have a fruitful situation of which we can profit for improving the corecursion state, and hence improving the flexibility of future corecursive definitions. Under some uniformity assumptions, g itself can be registered as friendly.

More precisely, assume that $g : \mathsf{K}\ \mathsf{J} \to \mathsf{J}$ is defined by $g = \mathsf{corecTop}\ s$ and that s can be proved uniform in the following sense: There exists a parametric function

$$\rho : \textstyle\prod_{A \in \mathsf{Set}} \mathsf{K}\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\Sigma^*\ (\mathsf{K}\ A))$$

such that $s = \rho \circ \mathsf{K}\langle \mathsf{id}, \mathsf{dtor}\rangle$ (Figure 5a). Then we can integrate g as a friendly operation as follows.

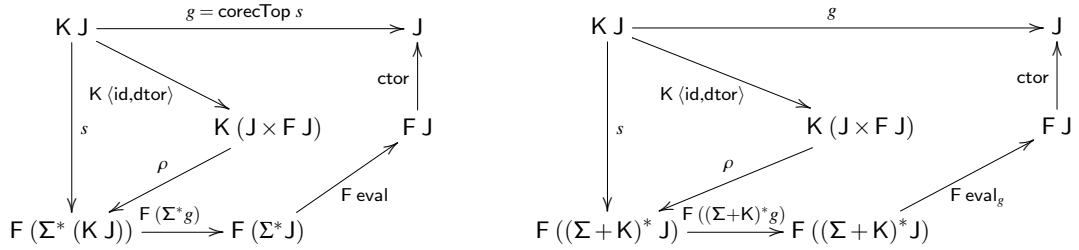
We define $\mathsf{nextState}_g\ (\overline{\mathsf{K}}, \overline{f}, \Lambda)$, the "next" corecursion state triggered by g, as $(\overline{\mathsf{K}'}, \overline{f'}, \Lambda')$, where

- $\overline{\mathsf{K}'} = (\mathsf{K}_1, \ldots, \mathsf{K}_n, \mathsf{K})$ (similarly to Σ versus $\overline{\mathsf{K}}$, we write Σ' for the signature functor of K'; note that we essentially have $\Sigma' = \Sigma + \mathsf{K}$);
- $\overline{f'} = (f_1, \ldots, f_n, g)$;
- $\Lambda' : \prod_{A \in \mathsf{Set}} \Sigma'\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\Sigma'^* A)$ is defined as $[\mathsf{F}\ \mathsf{embL} \circ \Lambda, \mathsf{F}\ \mathsf{embR} \circ \rho]$, where $[\_, \_]$ is the case operator on sums, which builds a function $[u, v] : B + C \to D$ from two functions $u : B \to D$ and $v : C \to D$, and $\mathsf{embL} : \Sigma^* A \to \Sigma'^* A$ and $\mathsf{embR} : \Sigma^*\ (\mathsf{K}\ A) \to \Sigma'^* A$ are the natural embeddings into $\Sigma'^* A$.

Theorem 8. If $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$ is a well-formed corecursion state, then so is $\mathsf{nextState}_g\ (\overline{\mathsf{K}}, \overline{f}, \Lambda)$.

In summary, we have the following scenario triggering the state's advancement:

1. One defines a new operation $g = \mathsf{corecTop}\ s$.
2. One shows that s factors through a parametric function ρ and $\mathsf{K}\ \langle \mathsf{id}, \mathsf{dtor}\rangle$ (as in Figure 5a); in other words, one shows that g's corecursive behavior s can be decomposed into a one-step destruction of the arguments and a parametric transformation (which is independent of J).
3. The corecursion state is updated by $\mathsf{nextState}_g$.



(a) Version defined with the general-purpose corecursor

(b) Freely mixing version

Figure 5: A new friendly operation g

Example 9. Assume that SCons and $\oplus$ are registered as friendly at the time of defining $\otimes$ (cf. Example 4). Then $\mathsf{K} = \lambda A.\ A^2$ and $\otimes = \mathsf{corecTop}\ s$, where

$$s = \lambda(xs, ys).\ (\mathsf{head}\ xs \times \mathsf{head}\ ys,\ \mathsf{vleaf}\ (xs, \mathsf{tail}\ ys)\ \boxed{\oplus}\ \mathsf{vleaf}\ (\mathsf{tail}\ xs, ys))$$

The function s can be recast into $\rho \circ \mathsf{K}\ \langle \mathsf{id}, \langle \mathsf{head}, \mathsf{tail} \rangle \rangle$, where

$$\rho : \textstyle\prod_{A \in \mathsf{Set}} (A \times (\mathsf{Nat} \times A))^2 \to \mathsf{Nat} \times \Sigma^*\ A^2$$

is defined by

$$\rho\ ((a, (m, a')), (b, (n, b'))) = (m \times n,\ \mathsf{vleaf}\ (a, b')\ \boxed{\oplus}\ \mathsf{vleaf}\ (a', b))$$

which is clearly parametric. Determining ρ from s and $\mathsf{K}\ \langle \mathsf{id}, \langle \mathsf{head}, \mathsf{tail} \rangle \rangle$ can be done in a syntax-directed fashion.

Above, the new operation $g : \mathsf{K}\ \mathsf{J} \to \mathsf{J}$ was defined using corecTop, and the domain of g was treated as any arbitrary domain. It turns out there is more opportunity for taking advantage of the form $\mathsf{K}\ \mathsf{J}$ of the domain: We can allow the corecursive calls of g to mix freely with occurrences of the other friendly operations, i.e., treat g as friendly already when defining it. To this end, we slightly change the codomain of ρ, replacing $\Sigma^*\ (\mathsf{K}\ A)$ with $(\Sigma + \mathsf{K})^* A$, i.e., $\Sigma'^* A$. We now assume $\rho : \prod_{A \in \mathsf{Set}} \mathsf{K}\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ ((\Sigma + \mathsf{K})^* A)$ and have the following improved version of Theorem 8.

Theorem 10. Assume $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$ is a well-formed corecursion state, $s = \rho \circ \mathsf{K}\langle \mathsf{id}, \mathsf{dtor} \rangle$ (as in Figure 5b), and ρ is parametric. Then there exists a unique function $g : \mathsf{K}\ \mathsf{J} \to \mathsf{J}$ that makes the (outer) diagram in Figure 5b commute and such that $\mathsf{nextState}_g\ (\overline{\mathsf{K}}, \overline{f}, \Lambda) = (\overline{\mathsf{K}'}, \overline{f'}, \Lambda')$ is again a well-formed corecursion state, where

- $\overline{\mathsf{K}'} = (\mathsf{K}_1, \ldots, \mathsf{K}_n, \mathsf{K})$ (hence $\Sigma' = \Sigma + \mathsf{K}$);
- $\overline{f'} = (f_1, \ldots, f_n, g)$;
- $\Lambda' : \prod_{A \in \mathsf{Set}} \Sigma'\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\Sigma'^* A)$ is defined as $[\mathsf{F}\ \mathsf{embL} \circ \Lambda, \rho]$.

In Figure 5b, the function $\mathsf{eval}_g : (\Sigma + \mathsf{K})^* \mathsf{J} \to \mathsf{J}$ is the natural extension of both $\mathsf{eval} : \Sigma^* \mathsf{J} \to \mathsf{J}$ and g.

The above theorem allows us to define and integrate as friendly functions such as

$$xs \heartsuit ys = \underline{\mathsf{SCons}}\ (\mathsf{head}\ xs \times \mathsf{head}\ ys)\ ((((xs \heartsuit \mathsf{tail}\ ys) \oplus (\mathsf{tail}\ xs \otimes ys)) \heartsuit ys) \otimes zs)$$

whose corecursive calls mix freely with $\oplus$ and $\otimes$.

In Theorem 8, the definition of the new operation is decoupled from the parametric decomposition of its corecursion law, which ensures friendliness. We may define g as $\mathsf{corecTop}\ s$ regardless of whether s decomposes as $\rho \circ \mathsf{K}\langle \mathsf{id}, \mathsf{dtor} \rangle$ for a parametric ρ; but we can register g as friendly only if such a decomposition is possible. On the other hand, in Theorem 10, the very existence of g depends on s being parametrically decomposable: The behavior of g needs to be a priori known as friendly, because g itself can participate in the call contexts for g. Thus, the following definition is valid, and yields a friendly operation, according to Theorem 10:

$$g\ xs = \underline{\mathsf{SCons}}\ (\mathsf{head}\ xs)\ (g\ (g\ (\mathsf{tail}\ xs)))$$

By contrast, the next definition is not valid due to the nonexistence of a suitable ρ:

$$g\ xs = \underline{\mathsf{SCons}}\ (\mathsf{head}\ xs)\ (g\ (g\ (\mathsf{tail}\ (\mathsf{tail}\ xs))))$$

In fact, it is not productive, let alone friendly.

3.7 Coinduction Up-to

In a proof assistant, specification mechanisms are not very useful unless they are complemented by suitable reasoning infrastructure. The natural counterpart of corecursion up-to is coinduction up-to. In our incremental framework, the expressiveness of coinduction up-to grows together with that of corecursion up-to.

We start with structural coinduction [56], allowing to prove two elements of J equal by exhibiting an F-bisimulation, i.e., a binary relation R on J such that whenever two elements j_1 and j_2 are related, their dtor-unfoldings are componentwise related by R:

$$\frac{R\ j_1\ j_2 \quad \forall j_1\ j_2 \in \mathsf{J}.\ R\ j_1\ j_2 \longrightarrow \mathsf{Frel}\ R\ (\mathsf{dtor}\ j_1)\ (\mathsf{dtor}\ j_2)}{j_1 = j_2}$$

Recall that our type constructors are not only functors but also relators. The notion of "componentwise relationship" refers to F's relator structure Frel.

Upon integrating a new operation g (Section 3.6), the coinduction rule is made more flexible by allowing the dtor-unfoldings to be componentwise related not only by R but more generally by a closure of R that takes g into account.

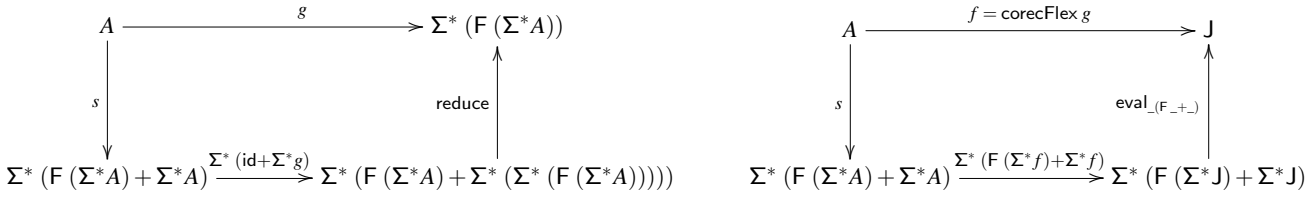
For a corecursion state $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$ and a relation $R : \mathsf{J} \to \mathsf{J} \to \mathsf{Bool}$, we define $\mathsf{cl}_{\overline{f}}\ R$, the $\overline{f}$-congruence closure of R, as the smallest equivalence relation that includes R and is compatible with each $f_i : \mathsf{K}_i\ \mathsf{J} \to \mathsf{J}$: $\forall z_1\ z_2 \in \mathsf{K}_i\ \mathsf{J}.\ \mathsf{Krel}_i\ R\ z_1\ z_2 \longrightarrow \mathsf{cl}_{\overline{f}}\ R\ (f_i\ z_1)\ (f_i\ z_2)$, where Krel_i is the relator associated with K_i.

The next theorem supplies the reasoning counterpart of the definition principle stated in Theorem 6. It can be inferred from recent, more abstract results [54].

Theorem 11. The following coinduction rule up to $\overline{f}$ holds in the corecursion state $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$:

$$\frac{R\ j_1\ j_2 \quad \forall j_1\ j_2 \in \mathsf{J}.\ R\ j_1\ j_2 \longrightarrow \mathsf{Frel}\ (\mathsf{cl}_{\overline{f}}\ R)\ (\mathsf{dtor}\ j_1)\ (\mathsf{dtor}\ j_2)}{j_1 = j_2}$$

Coinduction up to $\overline{f}$ is the ideal abstraction for proving equalities involving functions defined by corecursion up to $\overline{f}$: For example, a proof of commutativity for $\otimes$ naturally relies on contexts involving $\oplus$, because $\otimes$'s corecursive behavior (i.e., $\otimes$'s dtor-unfolding) depends on $\oplus$.



(a) Assumption

(b) Conclusion (via corecFlex)

Figure 6: Mixed fixpoint

4. Mixed Fixpoints

When we write fixpoint equations to define a function f, we often want to distinguish corecursive calls from calls that are sound for other reasons—for example, if they terminate. We model this situation abstractly by a function $s : A \to \Sigma^* (\mathsf{F} (\Sigma^* A) + \Sigma^* A)$. As usual, for each a, the shape of $s\ a$ represents the calling context for $f\ a$, with the occurrences of the content items a' in $s\ a$ representing calls to $f\ a'$. The new twist is that we now distinguish guarded calls (captured by the left-hand side of $+$) from possibly unguarded ones (the right-hand side of $+$).

We want to define a function f with the behavior indicated by s, i.e., making the diagram in Figure 6b commute. In the diagram, $+$ denotes the map function $u + v : B + C \to D + E$ built from two functions $u : B \to D$ and $v : C \to E$. In the absence of pervasive guards, we cannot employ the corecursors directly to define f. However, if we can show that the noncorecursive calls eventually lead to a corecursive call, we will be able to employ corecFlex. This precondition can be expressed in terms of a fixpoint equation. According to Figure 6a, the call to g (shown on the base arrow) happens only on the right-hand side of $+$, meaning that the intended corecursive calls are ignored when "computing" the fixpoint g. Our goal is to show that the remaining calls behave properly.

The functions reduce and eval that complete the diagrams of Figure 6 are the expected ones:

- The elements of $\Sigma^* (\mathsf{F} (\Sigma^* A))$ are formal-expression trees guarded on every path to the leaves, and so are the elements $\Sigma^* (\mathsf{F} (\Sigma^* A) + \Sigma^* (\Sigma^* (\mathsf{F} (\Sigma^* A))))$, but with a more restricted shape; reduce embeds the latter in the former:

  $$\mathsf{reduce} = \mathsf{flat} \circ \Sigma^* [\mathsf{vleaf}, \mathsf{flat}]$$

  where $\mathsf{flat} : \prod_{A \in \mathsf{Set}} \Sigma^* (\Sigma^* A) \to A$ is the standard join operation of the Σ^*-monad.

- $\mathsf{eval}_{\_(\mathsf{F}\ \_+\_)}$ evaluates all the formal operations of Σ^*:

  $$\mathsf{eval}_{\_(\mathsf{F}\ \_+\_)} = \mathsf{eval} \circ \Sigma^* [\mathsf{ctor} \circ \mathsf{F}\ \mathsf{eval}, \mathsf{eval}]$$

Theorem 12. If there exists (a unique) $g : A \to \Sigma^* (\mathsf{F} (\Sigma^* A))$ such that the diagram in Figure 6a commutes, there exists (a unique) $f : A \to \mathsf{J}$ such that the diagram in Figure 6b commutes, namely, $\mathsf{corecFlex}\ g$.

The theorem certifies the following procedure for making sense of a mixed fixpoint definition of a function f:

1. Separate the guarded and the unguarded calls (as shown in the codomain $\Sigma^* (\mathsf{F} (\Sigma^* A) + \Sigma^* A)$ of s).
2. Prove that the unguarded calls eventually terminate or lead to guarded calls (as witnessed by g).
3. Pass the unfolded guarded calls to the corecursor—i.e., take $f = \mathsf{corecFlex}\ g$.

Example 13. The above procedure can be applied to define facC, $\mathsf{primes} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Stream}$, and $\mathsf{cat} : \mathsf{Nat} \to \mathsf{Stream}$, while avoiding the unsound nasty (Section 2.3). A simple analysis reveals that the first self-call to primes is guarded while the second is not. We define $g : \mathsf{Nat} \times \mathsf{Nat} \to \Sigma^* (\mathsf{Nat} \times \Sigma^* (\mathsf{Nat} \times \mathsf{Nat}))$ by

$$\begin{aligned} g\ (m, n) = {} & \mathtt{if}\ (m = 0 \wedge n > 1) \vee \mathsf{gcd}\ m\ n = 1 \\ & \mathtt{then}\ \mathsf{vleaf}\ (n, \mathsf{vleaf}\ (m \times n, n + 1)) \\ & \mathtt{else}\ g\ (m, n + 1) \end{aligned}$$

In essence, g behaves like the function f we want to define (here, primes), except that the guarded calls are left symbolic, whereas the unguarded calls are interpreted as actual calls to g. We can show that g is well defined by a standard termination argument. This characteristic equation of g is the commutativity of the diagram determined by s as in Figure 6a, where $s : \mathsf{Nat} \times \mathsf{Nat} \to \Sigma^* (\mathsf{Nat} \times \Sigma^* (\mathsf{Nat} \times \mathsf{Nat}) + \Sigma^* (\mathsf{Nat} \times \mathsf{Nat}))$ is defined as follows:

$$\begin{aligned} s\ (m, n) = {} & \mathtt{if}\ (m = 0 \wedge n > 1) \vee \mathsf{gcd}\ m\ n = 1 \\ & \mathtt{then}\ \mathsf{vleaf}\ (\mathsf{Inl}\ (n, \mathsf{vleaf}\ (m \times n, n + 1))) \\ & \mathtt{else}\ \mathsf{vleaf}\ (\mathsf{Inr}\ (\mathsf{vleaf}\ (m, n + 1))) \end{aligned}$$

where Inl and Inr are the left and right sum embeddings. Setting $\mathsf{primes} = \mathsf{corecFlex}\ g$ yields the desired characteristic equation for primes after simplification (as illustrated in Example 4).

The primes example has all unguarded calls in tail form, which makes the associated function g tail-recursive. This need not be the case, as shown by the cat example, whose unguarded calls occur under the friendly operation $\oplus$. However, we do require that the unguarded calls occur in contexts formed by friendly operations alone. This requirement guarantees that after unfolding all the unguarded calls, the resulting context that is to be handled corecursively is friendly. This precludes unsound definitions such as nasty.

5. Formalization

We formalized the metatheory of Sections 3 and 4 in Isabelle/HOL. The results have been proved in higher-order logic with Infinity, Choice, and a mechanism for defining types by exhibiting nonempty subsets of existing types. The logic is comparable to Zermelo set theory with Choice (ZC) but weaker than ZFC. The development would work for any class of functors that are relators (or closed under weak pullbacks), contain basic functors (identity, (co)products, etc.) and are closed under intersection and composition, and have initial algebras and final coalgebra that can be represented in higher-order logic. However, our Isabelle development focuses on a specific class: the bounded natural functors [60].

The formalization consists of two parts: The *base* derives a corecursor up-to from a primitive corecursor; the *step* starts with a corecursor up-to and integrates an additional friendly operation.

The base part starts by axiomatizing a functor F and defines a codatatype with nesting through F: $\mathtt{codatatype}\ \mathsf{J} = \mathsf{ctor}\ (\mathsf{F}\ \mathsf{J})$. In general, J could depend on type variables, but this is an orthogonal concern. Then the formalization defines the free algebra over F and the basic corecursor seed Λ for initializing the state with ctor as friendly (Section 3.5). It also needs to lift Λ to the free algebra, a

technicality that was omitted in the presentation. Then it defines eval and other necessary structure (Section 3.3). Finally, it introduces corecTop and corecFlex (Section 3.4) and derives the corresponding coinduction principle (Section 3.7).

From a high-level point of view, the step part has a somewhat similar structure to the base. It axiomatizes a domain functor K and a parametric function ρ associated with the new friendly operation g to integrate. Then it extends the signature to include K, defines the extended corecursor seed Λ', and lifts Λ' to the free algebra. Next, it defines the parameterized eval_g and other infrastructure (Section 3.6). Finally, it introduces corecTop and corecFlex for the new state and derives the coinduction principle.

6. Prototype Implementation

The process of instantiating the metatheory to particular user-specified codatatypes is automated by a prototype tool. The user points to a particular codatatype [14]. The tool takes over and instantiates the generic corecursor to the indicated type, providing the concrete corecursion and mixed recursion–corecursion theorems. The stream and tree examples presented in Section 2 have all been obtained with this tool. As a larger case study, we formalized all the examples from the extended version of Hinze and James's study [29]. The parametricity proof obligations were discharged by Isabelle's parametricity prover [31]. The mixed recursion–corecursion definitions were done using Isabelle's facility for defining terminating recursive functions [36].

Our tool currently lacks syntactic sugar. It still requires some boilerplate from the user, namely the explic invocation of the corecursor and the parametricity prover. These are just a few extra lines of script per definition, and therefore the tool is also usable in the current form. Following the design of its primitive ancestor [14], the envisioned user-friendly `corec` command will automate the following steps (cf. Example 5):

1. Parse the user's corecursive specification of f and synthesize arguments to the current, most powerful corecursor.
2. Define f in terms of the corecursor.
3. Derive the original specification from the corecursor theorems.

Passing the `friendly` option to `corec` will additionally invoke the following procedure (cf. Example 9):

4. Extract a polymorphic function ρ from the specification of f.
5. Automatically prove ρ parametric or pass the proof obligation to the user.
6. Derive the strengthened corecursor and its coinduction rule.

The `corec` command will be complemented by an additional command, tentatively called `corec_friendly`, for registering arbitrary operations f (not necessarily defined using `corec`) as friendly. The command will ask the user to provide a corecursive specification of f as a lemma of the form $\mathsf{f}\ \bar{x} = \mathsf{ctor}\ \ldots$ and then perform steps 4 to 6. The `corec` command will become increasingly stronger as more operations are registered.

The following Isabelle-like theory fragment gives a flavor of the envisioned functionality from the user's point of view:

```
codatatype Stream A = SCons (head: A) (tail: Stream A)

corec (friendly) ⊕ : Stream → Stream → Stream
  xs ⊕ ys = SCons (head xs + head ys) (tail xs ⊕ tail ys)

corec (friendly) ⊗ : Stream → Stream → Stream
  xs ⊗ ys = SCons (head xs × head ys)
                  ((xs ⊗ tail ys) ⊕ (tail xs ⊗ ys))

lemma ⊕_commute: xs ⊕ ys = ys ⊕ xs
  by (coinduction arbitrary: xs ys rule: stream.coinduct) auto

lemma ⊗_commute: xs ⊗ ys = ys ⊗ xs
proof (coinduction arbitrary: xs ys rule: stream.coinduct_upto)
  case Eq_stream thus ?case unfolding tail_⊗
    by (subst ⊕_commute) (auto intro: stream.cl_⊕)
qed
```

7. Related Work

There is a lot of relevant work, concerning both the metatheory and applications in proof assistants and similar systems. We referenced some of the most closely related work in the earlier sections. Here is an attempt at a more systematic overview.

Category Theory. The notions of corecursion and coinduction up-to started with process algebra [55, 58] before they were recast in the abstract language of category theory [9, 29, 33, 35, 46, 54, 61]. Our approach owes a lot to this theoretical work, and indeed formalizes some state-of-the-art category theoretical results on corecursion and coinduction up-to [46, 54]. Besides adapting existing results to higher-order logic within an incremental corecursor cycle, we have extended the state of the art with a sound mechanism for mixing recursion with corecursion up-to.

Category theory provides an impressive body of abstract results that can be applied to solve concrete problems elegantly. Proof assistants have a lot to benefit from category theory, as we hope to have demonstrated with this paper. There has been prior work on integrating coinduction up-to techniques from category theory into these tools. Hensel and Jacobs [26] illustrated the categorical approach to (co)datatypes in PVS via axiomatic declarations of various flavors of trees with (co)recursors and proof principles. Popescu and Gunter proposed incremental coinduction for a deeply embedded proof system in Isabelle/HOL [51]. Hur et al. [32] extended Winskel's [64] and Moss's [47] parameterized coinduction and studied applications to Agda, Coq, and Isabelle/HOL. Endrullis et al. [25] developed a method to perform up-to coinduction in Coq inspired by behavioral logic [53]. To our knowledge, no prior work has realized corecursion up-to in a proof assistant.

Ordered Structures and Convergence. A number of approaches to define functions on infinite types are based on domain theory, or more generally on ordered structures and notions of convergence, including Matthews [45], Di Gianantonio and Miculan [23], Huffman [30], and Lochbihler and Hölzl [44]. These do not capture total programming or productivity; instead, the user must switch to a richer universe of domains and continuous computations.

Strictly speaking, our approach does not guarantee productivity either. This is an inherent limitation of the semantic (shallow embedded) approach in HOL systems, which do not specify a computational model (unlike Agda and Coq). Productivity can be argued informally by inspecting the characteristic corecursion equations.

Syntactic Criteria. Proof assistants based on type theory include checkers for termination of recursion functions and productivity of corecursive functions. These checkers are part of the system's trusted code base; bugs can lead to inconsistencies, as we saw for Agda [59] and Coq [22].[1] For users, built-in syntactic criteria are inflexible, due to their inability to evolve by incorporating semantic information; for example, Coq allows more than one constructor to appear as guards but is otherwise limited to primitive corecursion.

[1] In all fairness, we should mention that critical bugs were also found in the primitive definitional mechanism of our proof assistant of choice [38, 39]. Our point is not that brand B is superior to brand A, but rather that it is generally desirable to minimize the amount of trusted code.

To the best of our knowledge, the only deployed system that explicitly supports mixed recursive–corecursive definitions is Dafny. Leino and Moskal's paper [40] triggered our interest in the topic. However, a naive reading of the paper suggests that the inconsistent nasty example from Section 2.3 is allowed, as was the case with earlier versions of Dafny. Newer versions reject not only nasty but also the legitimate cat function from the same subsection.

Type Systems. A more flexible alternative to syntactic criteria is to have users annotate the functions' types with information that controls termination and productivity. Approaches in these category include fair reactive programming [19, 24, 37], clock variables [8, 20], and sized types [2]. Sized types are implemented in MiniAgda [3] and in newer versions of Agda, in conjunction with a destructor-oriented (copattern) syntax for corecursion [5]. These approaches, often featuring a blend of type systems and notions of convergence, achieve a higher modularity and trustworthiness, by moving away from purely syntactic criteria and toward semantic properties. By carefully tracking sizes and timers, they allow for more general corecursive call contexts than friendliness; for example, given suitable annotations, everyOther can participate in certain corecursive call contexts.

Our criterion captures a "1–1" contract: A friendly function can destroy *one* constructor to produce at least *one* constructor. The function double mapping the stream $a_1, a_2, \ldots$ to $a_1, a_1, a_2, a_2, \ldots$ is friendly, but it would be more precisely described by a 1–2 contract. The function everyOther mapping $a_1, a_2, a_3, a_4, \ldots$ to $a_1, a_3, \ldots$ is not friendly; it would require a 2–1 contract. And although everyOther $\circ$ double satisfies a 1–1 contract (2–1 $\circ$ 1–2 = 1–1), our corec command must reject the definition zeros = SCons 0 (everyOther (double zeros)) because the unfriendly function everyOther appears in the call context.

In exchange for their flexibility, clock variables and sized types require extending the type system and burden the types. The general contracts must be specified by the user and complicate the up-to corecursion principle; the contract arithmetic would have to be captured in the principle, giving rise to new proof obligations. By contrast, friendly functions can be freely combined. This is the main reason why we claim it is a "sweet spot."

There is a prospect of embedding our lighter approach into such heavier but more precise frameworks. Our friendly operations possibly form the maximal class of context functions requiring no annotations (in general), amounting to a lightweight subsystem of Krishnaswami and Benton's type system [37].

8. Conclusion

We presented a formalized framework for deriving rich corecursors that can be used to define total functions producing codatatypes. The corecursors gain in expressiveness with each new corecursive function definition that satisfies a semantic criterion. They constitute a significant improvement over the state of the art in the world of proof assistants based on higher-order logic, including HOL4, HOL Light, Isabelle/HOL, and PVS. Trustworthiness is attained at the cost of elaborate constructions. Coinduction being somewhat counterintuitive, we argue that these safeguards are well worth the effort. As future work, we want to transform our prototype tool into a solid implementation inside Isabelle/HOL.

Although we advocate the foundational approach, many ideas equally apply to systems with built-in codatatypes and corecursion. One could imagine extending the productivity check of Coq to allow corecursion under friendly operations, linking a syntactic criterion to a semantic property, as a lightweight alternative to clock variables and sized types. The emerging infrastructure for parametricity in Coq [12, 34] would likely be a useful building block.

Acknowledgment. Tobias Nipkow made this work possible. Stefan Milius guided us through his highly relevant work on abstract GSOS rules. Aymeric Bouzy constructed the example featured at the end of Section 3.3, which led us to enrich the framework with the cleaf injection. Andreas Abel and Rustan Leino discussed their tools and shared their paper drafts and examples with us. Mark Summerfield suggested many textual improvements. Reviewers provided very useful comments, suggested improvements of the presentation, and found technical typos. Blanchette was partially supported by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant NI 491/14-1). Popescu was partially 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 was supported by the DFG program Program and Model Analysis (PUMA, doctorate program 1480). The authors are listed alphabetically.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005.

[2] A. Abel. Termination checking with types. *RAIRO—Theor. Inf. Appl.*, 38(4):277–319, 2004.

[3] A. Abel. MiniAgda: Integrating sized and dependent types. In A. Bove, E. Komendantskaya, and M. Niqui, eds., *PAR 2010*, vol. 43 of *EPTCS*, pp. 14–28, 2010.

[4] A. Abel. Re: [Coq-Club] Propositional extensionality is inconsistent in Coq, 2013. Archived at `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00147.html`.

[5] A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In G. Morrisett and T. Uustalu, eds., *ICFP '13*, pp. 185–196. ACM, 2013.

[6] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In R. Giacobazzi and R. Cousot, eds., *POPL 2013*, pp. 27–38, 2013.

[7] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The Matita interactive theorem prover. In N. Bjørner and V. Sofronie-Stokkermans, eds., *CADE-23*, vol. 6803 of *LNCS*, pp. 64–69. Springer, 2011.

[8] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In G. Morrisett and T. Uustalu, eds., *ICFP '13*, pp. 197–208. ACM, 2013.

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

[10] F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats: Distributive Laws in Coalgebraic Modelling*. Ph.D. thesis, Vrije Universiteit Amsterdam, 2004.

[11] N. Benton. The proof assistant as an integrated development environment. In C.-c. Shan, ed., *APLAS 2013*, vol. 8301 of *LNCS*, pp. 307–314. Springer, 2013.

[12] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free: Parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012.

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

[14] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, eds., *ITP 2014*, vol. 8558 of *LNCS*, pp. 93–110. Springer, 2014.

[15] J. C. Blanchette, A. Popescu, and D. Traytel. Unified classical logic completeness: A coinductive pearl. In S. Demri, D. Kapur, and C. Weidenbach, eds., *IJCAR 2014*, vol. 8562 of *LNCS*, pp. 46–60. Springer, 2014.

[16] J. C. Blanchette, A. Popescu, and D. Traytel. Formalization associated with this paper. `https://github.com/dtraytel/fouco`, 2015.

[17] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In J. Vitek, ed., *ESOP 2015*, vol. 9032 of *LNCS*, pp. 359–382. Springer, 2015.

[18] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda—A functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, eds., *TPHOLs 2009*, vol. 5674 of *LNCS*, pp. 73–78. Springer, 2009.

[19] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. In S. Jagannathan and P. Sewell, eds., *POPL '14*, pp. 361–372. ACM, 2014.

[20] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. Programming and reasoning with guarded recursion for coinductive types. In A. M. Pitts, ed., *FoSSaCS 2015*, vol. 9034 of *LNCS*, pp. 407–421. Springer, 2015.

[21] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. WIFT '95, 1995.

[22] M. Dénès. [Coq-Club] Propositional extensionality is inconsistent in Coq, 2013. Archived at `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`.

[23] P. Di Gianantonio and M. Miculan. A unifying approach to recursive and co-recursive definitions. In H. Geuvers and F. Wiedijk, eds., *TYPES 2002*, vol. 2646 of *LNCS*, pp. 148–161. Springer, 2003.

[24] C. Elliott and P. Hudak. Functional reactive animation. In S. L. P. Jones, M. Tofte, and A. M. Berman, eds., *ICFP '97*, pp. 263–273. ACM, 1997.

[25] J. Endrullis, D. Hendriks, and M. Bodin. Circular coinduction in Coq using bisimulation-up-to techniques. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, eds., *ITP 2013*, vol. 7998 of *LNCS*, pp. 354–369. Springer, 2013.

[26] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. In E. Moggi and G. Rosolini, eds., *CTCS '97*, vol. 1290 of *LNCS*, pp. 220–241. Springer, 1997.

[27] J. Heras, E. Komendantskaya, and M. Schmidt. (Co)recursion in logic programming: Lazy vs eager. *Theor. Pract. Log. Prog.*, 14(4-5), 2014. Supplementary material.

[28] R. Hinze. Concrete stream calculus—An extended study. *J. Funct. Program.*, 20:463–535, 2010.

[29] R. Hinze and D. W. H. James. Proving the unique fixed-point principle correct—An adventure with category theory. In *ICFP '11*, pp. 359–371, 2011. Extended version available at `http://www.cs.ox.ac.uk/people/daniel.james/unique/unique-tech.pdf`.

[30] B. Huffman. A purely definitional universal domain. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, eds., *TPHOLs 2009*, vol. 5674 of *LNCS*, pp. 260–275. Springer, 2009.

[31] B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, eds., *CPP 2013*, vol. 8307 of *LNCS*, pp. 131–146. Springer, 2013.

[32] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In R. Giacobazzi and R. Cousot, eds., *POPL '13*, pp. 193–206. ACM, 2013.

[33] B. Jacobs. Distributive laws for the coinductive solution of recursive equations. *Inf. Comput.*, 204(4):561–587, 2006.

[34] C. Keller and M. Lasson. Parametricity in an impredicative sort. In P. Cégielski and A. Durand, eds., *CSL 2012*, vol. 16 of *LIPIcs*, pp. 381–395. Schloss Dagstuhl, 2012.

[35] B. Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011.

[36] A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, eds., *IJCAR 2006*, vol. 4130 of *LNCS*, pp. 589–603. Springer, 2006.

[37] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *LICS 2011*, pp. 257–266. IEEE, 2011.

[38] O. Kunčar. Correctness of Isabelle's cyclicity checker—Implementability of overloading in proof assistants. In X. Leroy and A. Tiu, eds., *CPP 2015*, pp. 85–94. ACM, 2015.

[39] O. Kunčar and A. Popescu. A consistent foundation for Isabelle/HOL. In C. Urban and X. Zhang, eds., *ITP 2015*, LNCS. Springer, 2015.

[40] K. R. M. Leino and M. Moskal. Co-induction simply—Automatic co-inductive proofs in a program verifier. In C. B. Jones, P. Pihlajasaari, and J. Sun, eds., *FM 2014*, vol. 8442 of *LNCS*, pp. 382–398. Springer, 2014.

[41] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[42] A. Lochbihler. Verifying a compiler for Java threads. In A. D. Gordon, ed., *ESOP 2010*, vol. 6012 of *LNCS*, pp. 427–447. Springer, 2010.

[43] A. Lochbihler. Making the Java memory model safe. *ACM Trans. Program. Lang. Syst.*, 35(4):12:1–65, 2014.

[44] A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. In G. Klein and R. Gamboa, eds., *ITP 2014*, vol. 8558 of *LNCS*, pp. 341–357. Springer, 2014.

[45] J. Matthews. Recursive function definition over coinductive types. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, eds., *TPHOLs '99*, vol. 1690 of *LNCS*, pp. 73–90. Springer, 1999.

[46] S. Milius, L. S. Moss, and D. Schwencke. Abstract GSOS rules and a modular treatment of recursive definitions. *Log. Meth. Comput. Sci.*, 9(3), 2013.

[47] L. S. Moss. Parametric corecursion. *Theor. Comput. Sci.*, 260(1-2):139–163, 2001.

[48] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[49] L. C. Paulson. Set theory for verification: I. From foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993.

[50] L. C. Paulson. Set theory for verification: II. Induction and recursion. *J. Autom. Reasoning*, 15(2):167–215, 1995.

[51] A. Popescu and E. L. Gunter. Incremental pattern-based coinduction for process algebra and its Isabelle formalization. In C.-H. L. Ong, ed., *FoSSaCS 2010*, vol. 6014 of *LNCS*, pp. 109–127. Springer, 2010.

[52] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP '83*, pp. 513–523, 1983.

[53] G. Roşu and D. Lucanu. Circular coinduction—A proof theoretical foundation. In A. Kurz, M. Lenisa, and A. Tarlecki, eds., *CALCO 2009*, vol. 5728 of *LNCS*, pp. 127–144. Springer, 2009.

[54] J. Rot, M. M. Bonsangue, and J. J. M. M. Rutten. Coalgebraic bisimulation-up-to. In P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. R. Nawrocki, and H. Sack, eds., *SOFSEM 2013*, vol. 7741 of *LNCS*, pp. 369–381. Springer, 2013.

[55] J. J. M. M. Rutten. Processes as terms: Non-well-founded models for bisimulation. *Math. Struct. Comp. Sci.*, 2(3):257–275, 1992.

[56] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.

[57] J. J. M. M. Rutten. A coinductive calculus of streams. *Math. Struct. Comp. Sci.*, 15(1):93–147, 2005.

[58] D. Sangiorgi. On the bisimulation proof method. *Math. Struct. Comp. Sci.*, 8(5):447–479, 1998.

[59] D. Traytel. [Agda] Agda's copatterns incompatible with initial algebras, 2014. Archived at `https://lists.chalmers.se/pipermail/agda/2014/006759.html`.

[60] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS 2012*, pp. 596–605. IEEE, 2012.

[61] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *LICS 1997*, pp. 280–291. IEEE, 1997.

[62] D. A. Turner. Elementary strong functional programming. In P. H. Hartel and M. J. Plasmeijer, eds., *FPLE '95*, vol. 1022 of *LNCS*, pp. 1–13. Springer, 1995.

[63] P. Wadler. Theorems for free! In *FPCA '89*, pp. 347–359. ACM, 1989.

[64] G. Winskel. A note on model checking the modal ν-calculus. *Theor. Comput. Sci.*, 83(1):157–167, 1991.