# Foundational Extensible Corecursion

Jasmin Christian Blanchette Andrei Popescu Dmitriy Traytel
Technische Universität München, Germany
{blanchet,popescua,traytel}@in.tum.de

## Abstract

This paper presents a theoretical 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 well-behaved operations, including constructors. Corecursive functions that are well behaved can be registered as such, thereby increasing the corecursor's expressiveness. To the extensible corecursor corresponds an equally flexible coinduction principle. The metatheory is formalized in the Isabelle proof assistant and forms the core of a prototype tool. The approach is foundational: The corecursor is derived from first principles, without requiring new axioms or extensions of the logic. This ensures that no inconsistencies can be introduced by omissions in a termination or productivity check.

***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*** Algorithms, 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 yield contradictions. Hence, most proof 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, 57]: 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 natsFrom produces one constructor before entering the nested call. Not only is the equation consistent, it also fully specifies the function's behavior.

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 a finite number of 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 over 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 [22, 35], 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 [34], a Java compiler [36], and the Java memory model [37]. Beyond programming, a number of papers illustrate how coinductive methods can lead to more elegant solutions than traditional approaches. A recent instance is a formal proof of the completeness theorem for classical first-order logic [16], based on possibly infinite derivation trees.

Codatatypes are supported by an increasing number of proof assistants, including Agda [17], Coq [13], Isabelle/HOL [42], Isabelle/ZF [43, 44], Matita [7], and PVS [18]. They are also present in the CoALP dialect of logic programming [24] and in the Dafny specification language [33].

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 a function definition?*

Generally, there are two main approaches 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, 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 derived as theorems from this internal definition, using 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 principal hurdle for their users is that syntactic criteria are often inflexible; the specification must be massaged so that it falls within a supported syntactic fragment, which is all the more aggravating because the desired property (termination or productivity) is semantic. But perhaps more troubling in systems that process theorems, soundness is not obvious at all and extremely 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 5). Indeed, Abel [2] 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 satisfactory 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, 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 [55]. 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 formally justified, foundational definitional principles tend to be simpler and 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 from the other HOL systems; and corecursion is limited to the primitive case, in which corecursive calls occur under exactly one constructor.

We are interested in lifting these limitations and 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. This flexibility is especially welcome given the rapid pace of research in the area.

In this paper, we build a highly expressive corecursor that extends primitive corecursion in the following ways: It allows corecursive calls under several constructors; it allows well-behaved operators in the context around or between the constructors and around the corecursive calls; and it supports terminating recursive calls under well-behaved operators, in conjunction with guarded corecursive calls. This general corecursor is accompanied by a corresponding, equally general coinduction principle that makes reasoning about it convenient. Both the corecursor and the coinduction principle can be derived in higher-order logic. The underlying 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 ensure the suitability of operators as participants to the 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). Moreover, thanks to the framework's modularity, function specifications can combine corecursion with recursion, yielding quite expressive mixed fixpoint definitions. This is inspired by the Dafny tool, but our approach is semantically founded and hence provably consistent.

The complete metatheory is formalized in Isabelle/HOL, in a generic proof development parameterized by arbitrary type constructors with additional structure. This development forms the basis of a prototype definitional tool (Section 4). Both are publicly available [15].

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 5). Our work adapts these insights in the context of proof assistants. The main contributions we make in this paper are the following:

- We introduce a semantic notion of safe context for corecursive calls, consisting of well-behaved operations, and matched by a coinduction up-to principle.
- We identify a sound fragment of mixed recursive–corecursive specifications and integrate it into the up-to framework.
- We formally derive the corecursor and coinduction principles in higher-order logic.

## 2. Motivating Examples

We demonstrate the expressiveness of the corecursor framework by examples, adopting the user's perspective. The case studies by Rutten [52] and Hinze [25] on stream calculi serve as our starting point. Streams of natural numbers can be defined as

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

where $\mathsf{SCons} : \mathsf{Nat} \to \mathsf{Stream} \to \mathsf{Stream}$ is the constructor and $\mathsf{head} : \mathsf{Stream} \to \mathsf{Nat}$, $\mathsf{tail} : \mathsf{Stream} \to \mathsf{Stream}$ are its selectors.

### 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 well-behaved operation. Intuitively, an operation is well behaved if it needs to destruct at most one constructor to produce one constructor. For streams, such an operation is allowed to inspect the head and the tail (but not the tail's tail) of its arguments before producing an SCons. Well-behaved operations preserve productivity; for this reason, they can safely surround the guarding constructor.

The rigorous definition will capture this intuition in a parametricity property that must be discharged upon registration of an operation as well behaved. In exchange, the framework yields a strengthened corecursor that incorporates the new operation.

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

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

The presence of non-well-behaved 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 $\mathsf{tail}\ \mathsf{stallA}$ unspecified.

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

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

The function not well behaved, despite being primitive corecursive. It also breaks productivity: $\mathsf{stallB} = \mathsf{SCons}\ 1\ (\mathsf{everyOther}\ \mathsf{stallB})$ stalls after producing two constructors.

Going back to our first example, we observe that the operation $\oplus$ is well behaved. 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 syntactic restriction. For example, we can define the stream of Fibonacci numbers in either of the following two ways:

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

Well-behaved operations are allowed to appear both under the constructor guard (as in fibA) and around it (as in fibB). Notice that two guards are necessary in the second example—one for each branch of the $\oplus$ operator. Incidentally, we are not aware of any other framework that allows such definitions. Without rephrasing the specification, fibB cannot be expressed in Rutten's format of behavioral differential equations [52] or in Hinze's syntactic restriction [25], nor via Agda copatterns [5, 6].

Many useful operations are well behaved 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 well behaved, we can employ it to define stream exponentiation, which also turns out to be well behaved:

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

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):[1]

$$\mathsf{facA} = \underline{\mathsf{SCons}}\ 1\ \mathsf{facA} \otimes \underline{\mathsf{SCons}}\ 1\ \mathsf{facA}$$
$$\mathsf{facB} = \mathsf{exp}\ (\underline{\mathsf{SCons}}\ 0\ \mathsf{facB})$$

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

The arguments of well-behaved operations are not restricted to the Stream type. For example, we can define the well-behaved supremum of a finite set of streams by primitive corecursion:

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

Here, fimage gives the image of a finite set under a function, and $\bigsqcup X$ is the maximum of a finite set of naturals or 0 if $X$ is empty.

### 2.2 Nested Corecursion Up-to

Although we use streams are 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:

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

[1] These definitions also constitute tentative contributions to the popular "Evolution of a Haskell Programmer" collection [49].

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 $\oplus$:

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

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 criterion for primitive corecursion for nested codatatypes requires the corecursive call to be applied through map, which is the case for $\boxplus$. Moreover, by virtue of being well behaved, $\boxplus$ can be used in the definition of the shuffle product of trees:

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

Again, the corecursive call takes place inside map, but this time also in the context of $\boxplus$. The specification of $\boxtimes$ is corecursive up-to and well behaved.

### 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 could be unfolded to arbitrary finite depth, ultimately yielding a purely corecursive definition. An example is the primes function taken from Di Gianantonio and Miculan [20]:

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

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 [20].)

Mixed recursion–corecursion also allows us to give a definition of factorials without involving any auxiliary stream operations:

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

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 also occur under well-behaved 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 of Catalan numbers: $C_1, C_2, \ldots$, where $C_i = \frac{1}{n+1}\binom{2n}{n}$. 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 apparently unobjectionable specification in which the corecursive call

is guarded by SCons and the unguarded call's argument strictly decreases toward 0:

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

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 first cousin once removed of the infamous $\mathsf{f}\ x = \mathsf{f}\ x+1$ example mentioned in the introduction. The framework rejects this specification on the grounds that the tail selector in the recursive call context is not well behaved.

### 2.4 Coinduction

Specifying corecursive functions is only the beginning. Once the specification has been accepted as productive, we presumably 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 on streams by providing a relation $R$ that relates the left-hand side with the right-hand side (first premise) and that constitutes a bisimulation (second premise). Streams that are related by a bisimulation cannot be distinguished by taking observations (i.e., by applying the head and tail selectors); therefore they must be equal. In other words, equality is the largest bisimulation.

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 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 must show that $\mathsf{head}\ s = \mathsf{head}\ t$ and $R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)$. The first property can be discharged by a simple calculation. For the second one:

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

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 well-behaved 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 well-behaved operation in the sense that our framework refines the definition of the congruence closure cl. (Strictly speaking, a fresh symbol $\mathsf{cl}'$ 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 further 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 $\mathsf{cl}\ R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)$:

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

$$\begin{aligned}&\mathsf{cl}\ R\ (\mathsf{tail}\ s)\ (\mathsf{tail}\ t)\\ &\longleftrightarrow \mathsf{cl}\ R\ (\mathsf{tail}\ (\mathsf{exp}\ (xs \oplus ys)))\ (\mathsf{tail}\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys))\\ &\longleftrightarrow \mathsf{cl}\ R\ ((\mathsf{tail}\ xs \oplus \mathsf{tail}\ ys) \otimes \mathsf{exp}\ (xs \oplus ys))\\ &\qquad (\mathsf{exp}\ xs \otimes (\mathsf{tail}\ ys \otimes \mathsf{exp}\ ys) \oplus (\mathsf{tail}\ xs \otimes \mathsf{exp}\ xs) \otimes \mathsf{exp}\ ys)\\ &\overset{*}{\longleftrightarrow} \mathsf{cl}\ R\ ((\mathsf{tail}\ xs \otimes \mathsf{exp}\ (xs \oplus ys) \oplus \mathsf{tail}\ ys \otimes \mathsf{exp}\ (xs \oplus ys))\\ &\quad{}^{\oplus}\!\longleftarrow (\mathsf{tail}\ xs \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys) \oplus \mathsf{tail}\ ys \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)\\ &\longleftarrow \mathsf{cl}\ R\ (\mathsf{tail}\ xs \otimes \mathsf{exp}\ (xs \oplus ys))\ (\mathsf{tail}\ xs \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)) \wedge\\ &\quad{}^{\otimes}\!\longleftarrow \mathsf{cl}\ R\ (\mathsf{tail}\ ys \otimes \mathsf{exp}\ (xs \oplus ys))\ (\mathsf{tail}\ ys \otimes (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys))\\ &\longleftarrow \mathsf{cl}\ R\ (\mathsf{tail}\ xs)\ (\mathsf{tail}\ xs) \wedge \mathsf{cl}\ R\ (\mathsf{tail}\ ys)\ (\mathsf{tail}\ ys) \wedge\\ &\qquad \mathsf{cl}\ R\ (\mathsf{exp}\ (xs \oplus ys))\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)\\ &\longleftarrow R\ (\mathsf{exp}\ (xs \oplus ys))\ (\mathsf{exp}\ xs \otimes \mathsf{exp}\ ys)\end{aligned}$$

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 [15] 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 (which is the case in Isabelle/HOL). The formalization [15] 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 an arbitrary (bounded) functor. The approach is quite general and does not rely on any particular grammar for specifying codatatypes.

### 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 $\mathsf{F} : \mathsf{Set}^n \to \mathsf{Set}$ operating on sets (or perhaps on 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 which commutes with binary-relation identity and composition.

Following standard notation from category theory, we will 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 hence $n = 1$), Frel is exactly rel from Section 2.4. It is well known that the positive type constructors defined by standard means (basic types, composition, least or greatest fixpoints) have canonical functorial and relator structure. This is crucial for the foundational construction of user-specified (co)datatypes in Isabelle/HOL [55].

But even non-positive type constructors $\mathsf{G} : \mathsf{Set}^n \to \mathsf{Set}$ exhibit a relator-like structure $\mathsf{Grel} : \prod_{\overline{A},\overline{B}\in\mathsf{Set}^n} (\overline{A} \to \overline{B}) \to (\mathsf{G}\ \overline{A} \to \mathsf{G}\ \overline{B})$ (which need not commute with relation composition, though). 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 $\forall a_1 \in A_1.\ \forall b_1 \in B_1.\ R_1\ a_1\ b_1 \longrightarrow R_2\ (f\ a_1)\ (f\ b_1)$. A polymorphic function $c : \prod_{\overline{A}\in\mathsf{Set}^n} \mathsf{G}\ \overline{A}$, $c$ is called *parametric* [46, 58] if $\forall \overline{A}, \overline{B} \in \mathsf{Set}^n.\ \forall R : \overline{A} \to \overline{B} \to \mathsf{Bool}.\ \mathsf{Grel}\ R\ c_{\overline{A}}\ c_{\overline{B}}$. The maintenance of relator-like structures is very helpful for automating theorem transfer along isomorphisms and quotients [27].

Here we explore an additional important benefit of maintaining functorial and relator structure for type constructors: the possibility to incrementally extend the corecursor in reaction to user input.

Throughout this section, we assume that all the considered type constructors are both functors and relators, include basic functors such as identity, constant, sum, and product, and are closed under least fixpoints (initial algebras) and greatest fixpoints (final coalgebras). To emphasize the central role played by their functorial structure, we simply call them *functors*. Examples of such classes of type constructors include the datafunctors [23], the containers [1], and the bounded natural functors [55].

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 presented 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 $\mathsf{F} = (\lambda A.\ \mathsf{Nat} \times A)$.

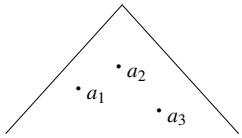



Figure 1: An element $x$ of $\mathsf{F}\ A$ with content items $a_1, a_2, a_3$

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 the tree-like drawing of 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 $f : A \to B$, the functorial action associated with F sends any $x$ into 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 [55].

### 3.2 Primitive Corecursion

The codatatype that defines J also 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 $\mathsf{corecPrim} : \prod_{A\in\mathsf{Set}} (A \to \mathsf{F}\ A) \to A \to \mathsf{J}$ characterized by the equation $\mathsf{corecPrim}\ s\ a = \mathsf{ctor}\ (\mathsf{F}\ (\mathsf{corecPrim}\ s)\ (s\ a))$. 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 identify the curried and uncurried forms of functions, counting on implicit coercions between the two.

### 3.3 The Corecursion State

Given any functor $\Sigma : \mathsf{Set} \to \mathsf{Set}$, we define its *free-monad functor* $\Sigma^*$ by $\Sigma^* A = \mathsf{lfp}\ (\lambda B.\ A + \Sigma\ B)$. We write $\mathsf{leaf} : A \to \Sigma^* A$ and $\mathsf{op} : \Sigma\ (\Sigma^* A) \to \Sigma^* A$ for the left and right injections into $\Sigma^* A$.[2]

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 $\mathsf{K}_i$, a function $f_i : \mathsf{K}_i\ \mathsf{J} \to \mathsf{J}$;
- a polymorphic function $\Lambda : \prod_{A\in\mathsf{Set}} \Sigma\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\Sigma^* A)$.

[2] The functions leaf and op are in fact polymorphic; for example, leaf has type $\prod_{A\in\mathsf{Set}} A \to \Sigma^* A$. We often omit the set parameters of polymorphic functions if they can be inferred from the context, writing leaf and op instead of $\mathsf{leaf}_A$ and $\mathsf{op}_A$.

We call the $f_i$'s the *well-behaved operations* and define their collective *signature functor* $\Sigma$ as $\lambda A.\ \mathsf{K}_1\ A + \cdots + \mathsf{K}_n\ A$, where $\iota_i : \mathsf{K}_i \to \Sigma$ is the standard embedding of $\mathsf{K}_i$ into $\Sigma$. We call $\Lambda$ the *corecursor seed*. The corecursion state is subject to the following conditions:

**Parametricity:** $\Lambda$ is parametric.

**Well-behavedness:** 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 $\Sigma^*$):

$$\begin{array}{lcl} \mathsf{eval}\ (\mathsf{leaf}\ j) & = & j \\ \mathsf{eval}\ (\mathsf{op}\ z) & = & \mathsf{case}\ z\ \mathsf{of}\ \iota_i\ t \Rightarrow f_i\ (\mathsf{K}_i\ \mathsf{eval}\ t) \end{array}$$

Functions having the type of $\Lambda$ and additionally assumed parametric (or, equivalently, assumed to be natural transformations) are known in the category theory community as "abstract GSOS rules." They were introduced by Turi and Plotkin [56] and further studied by Bartels [9], Jacobs [29], and Milius et al. [40], among others.

Thus, a corecursion state is a triple $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$. As we will see in Section 3.6, the state will evolve as the user defines new functions and registers them. The $f_i$'s are the operations that have been registered as safe for participating in the context of corecursion calls. Since $f_i$ has type $\mathsf{K}_i\ \mathsf{J} \to \mathsf{J}$, we think of $\mathsf{K}_i$ as encoding the arity of $f_i$. Then $\Sigma$, the sum of the $\mathsf{K}_i$'s, represents the signature consisting of all the $f_i$'s. Thus, for each $A$, $\Sigma^* A$ represents the set of formal expressions over $\Sigma$ and $A$, i.e., the trees built starting from the "variables" in $A$ as leaves by applying operations symbols corresponding to the $f_i$'s. Finally, eval evaluates in J the formal expressions of $\Sigma^*\,\mathsf{J}$ by applying the functions $f_i$ recursively.

If the functors $\mathsf{K}_i$ are restricted to be finite monomials $\lambda A.\ A^{k_i}$, the functor $\Sigma$ can be seen as a standard algebraic signature and $(\Sigma^* A, \mathsf{op})$ as the standard term algebra for this signature, over the variables $A$. However, we allow $\mathsf{K}_i$ to be more exotic; for example, $\mathsf{K}_i\ A$ can be $A^{\mathsf{Nat}}$ (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 well-behavedness conditions on the state enter the picture.

We start with well-behavedness. Assume $x \in \mathsf{K}_i$, which is unambiguously represented in $\Sigma$ 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 destruct 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 $\Lambda$ 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):

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

In summary, $\Lambda$ is a schematic representation of the mutually corecursive behavior of the well-behaved operations up to the production of the first observable data. This intuition is made formal in the well-behavedness condition, which states that the diagram in Figure 2 commutes for each $f_i$. (We could replace the right upward arrow labeled by ctor with a downward arrow labeled by dtor without changing the diagram's meaning. However, we consistently prefer the constructor view in our exposition.)

In the above explanations, we saw that it suffices to peel off one layer of the arguments $j_i$ (by applying dtor) for a well-behaved

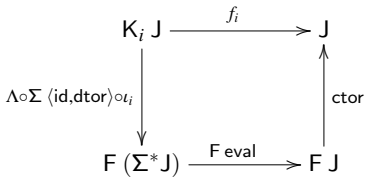



Figure 2: The well-behavedness condition

operation $f_i$ to produce, via $\Lambda$, one layer of the result and to delegate the rest of the computation to a context consisting of a combination of well-behaved operations (an element of $\Sigma^*\,\mathsf{J}$). But how to 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 result in a deeper exploration? An elegant way of capturing this is to require that $\Lambda$, which is a polymorphic function, operates without 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 well-behaved operations $\overline{f}$ are a redundant piece of data in the state $(\overline{\mathsf{K}}, \overline{f}, \Lambda)$, since, assuming $\Lambda$ parametric, we can prove that there exists a unique tuple $\overline{f}$ that satisfies the well-behavedness condition. In other words, the operations $\overline{f}$ could be derived on a per-need basis.

**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 well-behaved 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, $\Sigma^* = \mathsf{lfp}\ (\lambda B.\ A + (\mathsf{Nat} \times B + B^2))$ consists of formal-expression trees with leaves in $A$ and built using arity-correct applications of operation symbols corresponding to SCons and $\oplus$, denoted by $\boxed{\mathsf{SCons}}$ and $\boxed{\oplus}$. Given $n \in \mathsf{Nat}$ and $a, b \in A$, an example of such a tree is $\mathsf{leaf}\ a\ \boxed{\oplus}\ \boxed{\mathsf{SCons}}\ (n, \mathsf{leaf}\ a\ \boxed{\oplus}\ \mathsf{leaf}\ b)$. If additionally $A = \mathsf{J}$, then eval applied to the above tree is $a\ \oplus\ \mathsf{SCons}\ n\ (a\ \oplus\ b)$.

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

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

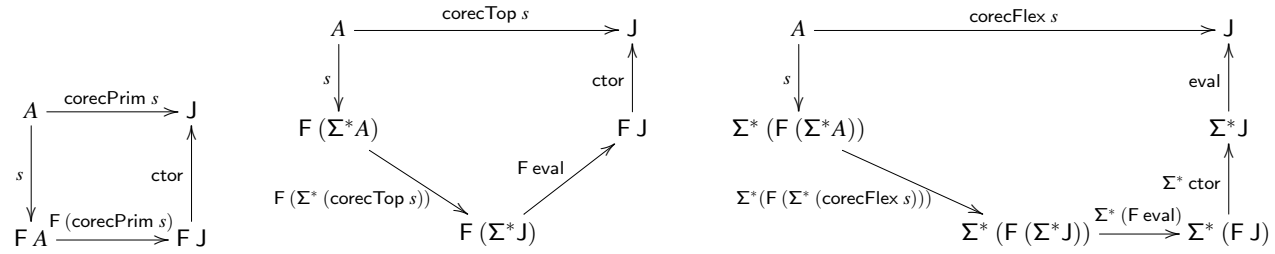
Informally, SCons and $\oplus$ exhibit the following behaviors:

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

### 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 on J, $f_i : \mathsf{K}_i\ \mathsf{J} \to \mathsf{J}$, that satisfy the well-behavedness properties expressed in terms of a parametric function $\Lambda$. 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:




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

Figure 3: The corecursors

**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 $\mathsf{F}$, $\Sigma^*$, 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 well-behaved operations $f_i$ (represented by $\Sigma^*$);
- for corecFlex, they consist of any combination of well-behaved 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 well-behaved operators (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 well-behaved operations.

**Example 5.** For each of the examples from Section 2.1, we give the weakest of the three recursors that can handle it (assuming that the necessary well-behaved operations have been registered):

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

Making the usual identification between $\mathsf{Unit} \to \mathsf{J}$ and J, we can define fibA and facA as follows:

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

Let us look at fibA more closely, comparing its 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 well-behaved operations and represented by the symbols $\boxed{\mathsf{SCons}}$ and $\boxed{\oplus}$ (cf. Example 4). Finally, the corecursive calls of fibA are captured by leaf $u$.

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 leaf instead of their applications to the unique element $()$ of Unit, namely $s\ ()$, fibA $()$, and leaf $()$:

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

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

Parametricity and well-behavedness are crucial for proving that our corecursors actually exist:

**Theorem 6.** There exist the polymorphic functions corecTop and corecFlex making the diagrams in Figure 3b and 3c commute. Moreover, for each $s$ of the appropriate type, corecTop $s$ or corecFlex $s$ is the unique function making the corresponding 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 very recently (and independently) proved by Milius et al. [40, 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.

More precisely, 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;

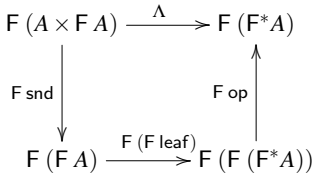



Figure 4: Definition of Λ for the initial state

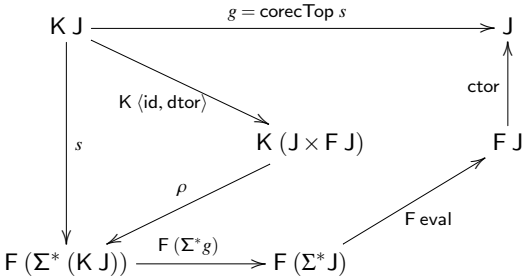



Figure 5: A new well-behaved operation $g$

- $\Lambda : \prod_{A \in \mathsf{Set}} \mathsf{F}\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\mathsf{F}^*\ A)$ is defined as $\mathsf{F}\ (\mathsf{op} \circ \mathsf{F}\ \mathsf{leaf} \circ \mathsf{snd})$, where $\mathsf{snd}$ is the second product projection.

Recall that the seed $\Lambda$ 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 well-behavedness.

### 3.6 Advancing the Corecursion State

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

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

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 to be uniform in the following sense: There exists a parametric function $\rho : \prod_{A \in \mathsf{Set}} \mathsf{K}\ (A \times \mathsf{F}\ A) \to \mathsf{F}\ (\Sigma^*\ (\mathsf{K}\ A))$ such that $s = \rho \circ \mathsf{K}\langle \mathsf{id},\ \mathsf{dtor} \rangle$ (Figure 5). Then we can integrate $g$ as a well-defined 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 $\Sigma$ versus $\overline{\mathsf{K}}$, we write $\Sigma'$ for the signature functor of $\mathsf{K}'$; 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 $[\Lambda \circ \mathsf{F}\ \mathsf{embL},\ \rho \circ \mathsf{F}\ \mathsf{embR}]$ 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 $\mathsf{embL} : \Sigma^* A \to \Sigma'^* A$, $\mathsf{embR} : \Sigma^*\ (\mathsf{K}\ A) \to \Sigma'^* A$ are the natural embeddings into $\Sigma'^* A$.

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

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 $\rho$ and $\mathsf{K}\ \langle \mathsf{id},\ \mathsf{dtor} \rangle$ (as in Figure 5); in other words, one shows that $g$'s corecursive behavior $s$ decomposes into a one-step destruction of the arguments and a parametric transformation (which is independent of $\mathsf{J}$).
3. The corecursion state is updated by $\mathsf{nextState}_g$.

**Example 9.** The operations onetwos, $\oplus$, $\otimes$, and exp from Section 2.1 are covered by this scenario. For example, assume that SCons and $\oplus$ are registered as well behaved at the time of defining $\otimes$ (cf. Example 4). Then $\mathsf{K} = (\lambda A.\ A^2)$ and $\otimes = \mathsf{corecTop}\ s$, where

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

The function $s$ decomposes 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 $\rho\ ((a, (m, a')), (b, (n, b'))) = (m \times n, (a, b')\boxed{\oplus}(a', b))$, which is clearly parametric. The process of determining $\rho$ from $s$ and $\mathsf{K}\ \langle \mathsf{id},\ \langle \mathsf{head},\ \mathsf{tail} \rangle \rangle$ is directed by the syntax.

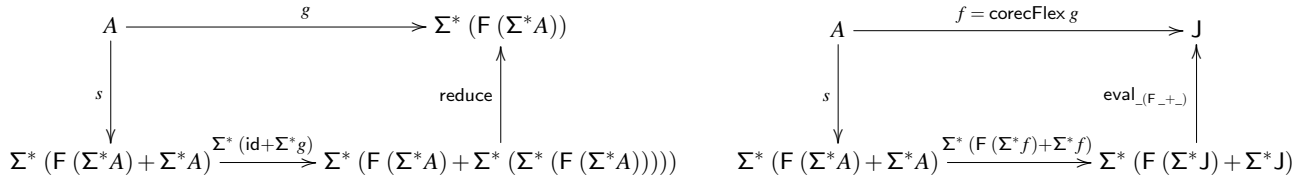
### 3.7 Mixed Fixpoints

When we write one or more fixpoint equations in the attempt of defining a function $f$, we may wish to distinguish between the corecursive calls and the calls that are sound for other reasons (e.g., they lead to eventual termination). We can 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 between guarded calls (captured by the left-hand side of $+$) and possibly unguarded ones (captured by the right-hand side of $+$).

We are interested in defining a function $f$ having the behavior indicated by $s$, i.e., making the diagram in Figure 6b commute. In the figure, $+$ denotes the map function that builds a function $u + v : B + C \to D + E$ 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 non-corecursive calls eventually lead to a corecursive call, we will be able to employ corecFlex. This precondition can be phrased in terms of a suitable fixpoint equation, depicted in Figure 6a. According to the diagram, 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 here 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: $\mathsf{reduce} = \mathsf{flat} \circ \Sigma^*[\mathsf{leaf}, \mathsf{flat}]$, where $\mathsf{flat} : \prod_{A \in \mathsf{Set}} \Sigma^*\ (\Sigma^* A) \to A$ is the standard join operation of the $\Sigma^*$-monad.




(a) Assumption (b) Conclusion

Figure 6: Mixed fixpoint

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

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

**Theorem 10.** 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 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 11.** 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 $g : \mathsf{Nat} \times \mathsf{Nat} \to \Sigma^* (\mathsf{Nat} \times \Sigma^* \mathsf{Nat} + \Sigma^* \mathsf{Nat})$ by

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

In essence, $g$ behaves like (the intended) $f$ except that the guarded calls are left symbolic, whereas the unguarded calls are interpreted as actual calls to $g$. One can show that $g$ is well defined by a standard termination argument; in Isabelle/HOL, it can be done using the `function` command [32]. This characteristic of $g$ is essentially the commutativity of determined by $s$ as in Figure 6a, where $s : \mathsf{Nat} \times \mathsf{Nat} \to \Sigma^* (\mathsf{Nat} \times \Sigma^* \mathsf{Nat} + \Sigma^* \mathsf{Nat})$ is defined as follows (with Inl and Inr being the left and right sum embeddings):

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

Setting $\mathsf{primes} = \mathsf{corecFlex}\ g$ yields the desired characteristic equation for primes after applying the necessary simplification rules (cf. 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 well-behaved operation $\oplus$. However, we do require that the unguarded calls occur in contexts formed by well-behaved operations alone. After unfolding all the unguarded calls, the resulting context that is to be handled corecursively must be well behaved. This is what precludes unsound definitions such as that of nasty.

### 3.8 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 [51], which allows 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 $\mathsf{Krel}_i$ is the relator associated with $\mathsf{K}_i$.

The next theorem supplies the reasoning counterpart of the definition principle stated in Theorem 6. It can be inferred from recent, more abstract results [48].

**Theorem 12.** 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, the proof of commutativity for $\otimes$ in Section 2.4 naturally relies on contexts involving $\oplus$, because $\otimes$'s corecursive behavior (i.e., $\otimes$'s dtor-unfolding) depends on $\oplus$.

## 4. Formalization and Implementation

The metatheory presented in Section 3 and the examples of Section 2 have been formalized in Isabelle/HOL [15]. Higher-order logic does not allow quantification over type constructors; instead, we fix arbitrary type constructors, axiomatize their functorial and relator structure, and develop the metatheory from there.

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 well-behaved operation.

The base part starts by axiomatizing a functor F and defines a codatatype with nesting through F: `codatatype` J = ctor (F J). (In general, J could depend on type variables, but this is an orthogonal concern that would only clutter the formalization.) Then the formalization defines the free algebra over F and the basic corecursor seed $\Lambda$ for initializing the state with ctor as well behaved (Section 3.5). It also needs to lift $\Lambda$ to the free algebra, a technicality that was omitted in the presentation. Then it defines eval as well as other necessary infrastructure (Section 3.3). Finally, it introduces corecTop and corecFlex (Section 3.4) and derives the corresponding coinduction principle (Section 3.8).

From a high-level point of view, the step has a somewhat similar structure. It axiomatizes a domain functor K and a parametric function $\rho$ associated with the new well-behaved operation $g$ to integrate. Then it extends the signature to include K, defines the extended corecursor seed $\Lambda'$, and lifts $\Lambda'$ to the free algebra. Next, it defines the parameterized $\mathsf{eval}_g$ and other infrastructure (Section 3.6). Finally, it introduces corecTop and corecFlex for the new state and derives the coinduction principle.

The most courageous users can make the metatheory work for their application by replacing the axiomatizations and the codatatype definition with the desired artifacts and iterating the step as many times as necessary by copying the theory files and editing them. This tedious (and temerarious) process is automated by a prototype tool, written as a shell script. The stream and tree examples presented in Section 2 have been developed in this way. The parametricity proof obligations were discharged with the help of Isabelle's parametricity prover [27]. The mixed recursion–corecursion definitions were done using Isabelle's facility for defining terminating recursive functions [32]. The result is fully definitional, which ensures that no inconsistencies are introduced by mistake.

Even with this prototype tool, we are far from the level of automation and convenience expected by most potential users. The situation is in many ways similar to that of codatatypes a few years ago, after the development of their formalized metatheory [55] but before their materialization in Isabelle/HOL [14].

We have a fairly clear vision for the implementation, inspired by our experience with the metatheory, the prototype tool, and the case studies. The core of the framework will take the form of a `corec` command that allows users to specify a function f corecursively and that performs the following steps (cf. Example 5):

1. Parse the 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 *well_behaved* option to `corec` will additionally invoke the following procedure (cf. Example 9):

4. Extract a polymorphic function $\rho$ from the specification of f.
5. Automatically prove $\rho$ parametric or pass the proof obligation to the user.
6. Derive the new strengthened corecursor and the corresponding coinduction principle.

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

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

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

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

corec (well_behaved) ⊗ : 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
```

## 5. 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 have been around for years, starting with process algebra [50, 53] and later receiving incarnations in the abstract language of category theory [9, 29, 31, 40, 48, 56]. 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 [40, 48]. Nonetheless, there are some novel aspects, motivated by our target application, including the tight connection between corecursion and coinduction up-to and the support for mixed recursion–corecursion.

Category theory provides a gigantic body of abstract results, which can be used provide elegant modular solutions to many concrete problems. 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 [23] illustrated the categorical approach to (co)datatypes in PVS via axiomatic declarations of various flavors of trees with associated (co)recursors and proof principles. Popescu and Gunter proposed incremental coinduction for a deeply embedded proof system in Isabelle/HOL [45]. Hur et al. [28] extended Winskel's [59] and Moss's [41] parameterized coinduction and studied applications to Agda, Coq, and Isabelle/HOL. Endrullis et al. [21] developed a method to perform up-to coinduction in Coq adapting insight from behavioral logic [47]. However, 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 [39], Di Gianantonio and Miculan [20], Huffman [26], and Lochbihler and Hölzl [38]. These are not directly comparable to our work because they do not guarantee productivity or otherwise offer total programming. They also force the user to switch to a different, richer universe of domains or to define ordered structures and perform continuity proofs (although Matthews shows that this process can be partly automated). Remarkably, Di Gianantonio and Miculan's methodology allows mixed recursive–corecursive definitions.

Strictly speaking, our approach does not guarantee productivity either. This is an inherent limitation of the semantic (or shallow embedded) approach in a system such as Isabelle/HOL that does

not specify a computational model (unlike Agda and Coq). Productivity can be argued informally by inspecting the characteristic equations of the corecursors.

***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 was recently demonstrated for Agda [54] and Coq [19].[3] For users, such syntactic criteria are also inflexible; for example, Coq allows more than one constructor to appear as guards but is otherwise limited to primitive corecursion.

To the best of our knowledge, the only deployed system that explicitly supports mixed recursive–corecursive definitions in a total setting is Dafny. Leino and Moskal's paper [33] triggered our interest in the topic. Unfortunately, the paper is not entirely clear about the supported fragment. A naive reading 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 section.

***Type Systems.*** A more flexible alternative to syntactic criteria is to have users annotate the functions' types with information that controls termination and productivity. Two competing approaches are sized types [3] and clock variables [8]. Size types are implemented in MiniAgda [4] and in newer versions of Agda, in conjunction with a destructor-oriented syntax (copatterns) for corecursion [5, 6]. Sized types and clock variables are part of an effort to achieve more modularity and a higher degree of trustworthiness, by moving away from purely syntactic criteria and toward semantic properties. By carefully tracking sizes, they allow for more general contexts than our well-behavedness criterion. Their main disadvantages are that they require an extension to the type system and that they burden the types, complicating user formalizations. Finally, their combination with other features such as dependent types is highly nontrivial; without extremely rigorous paper proofs that capture the complexity of the formalisms implemented in real-world proof assistants, it is difficult to rule out inconsistencies.

## 6. Conclusion

We have presented a 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, foundational constructions. The characteristic theorems are derived from an internal constructor, rather than stated as axioms. Coinduction being somewhat counterintuitive, we argue that these safeguards are well worth the effort.

Although we emphasized the foundational nature of the framework, many of our ideas equally apply to systems with a built-in notion of codatatypes and corecursion, such as Agda, CoALP, Coq, Dafny, Matita, and PVS. In particular, one could imagine extending the productivity check of a proof assistant such as Coq to allow corecursion under well-behaved operations, linking a syntactic criterion to a semantic property, as a lightweight alternative to the notationally heavier sized types and clock variables. The emerging infrastructure for parametricity in Coq [12, 30] would likely be a useful building block.

Many directions for future work suggest themselves. To reach users, we want to transform the crude prototype tool into a solid implementation inside Isabelle/HOL. On the theoretical front, some research is necessary to accommodate mutually corecursive functions. It would also be desirable to provide more general notions of well-behaved operation, with constructor counting (e.g., "consumes $m$, produces $n$ constructors"), although the interaction with coinduction is not clear. Finally, we believe it should be possible to integrate recursion and corecursion more tightly, without enforcing a strict distinction between recursive and corecursive calls—but how to express these ideas formally and generally in a foundational proof assistant is currently unclear to us.

[3] In all fairness, we should mention that critical bugs were also found in the primitive definitional mechanism of our proof assistant of choice, Isabelle. 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.

## Acknowledgments

Tobias Nipkow made this work possible. Stefan Milius guided us through his highly relevant work on abstract GSOS rules. Andreas Abel and Rustan Leino were always ready to discuss their tools and share their paper drafts and examples with us. Mark Summerfield suggested many textual improvements.

Blanchette is supported by the Deutsche Forschungsgemeinschaft (DFG) project Hardening the Hammer (grant Ni 491/14-1). Popescu is supported by the DFG project Security Type Systems and Deduction (grant Ni 491/13-2) as part of the program Reliably Secure Software Systems (RS$^3$, priority program 1496). Traytel is supported by the DFG program Program and Model Analysis (PUMA, doctorate program 1480). The authors are listed in alphabetical order regardless of individual contributions or seniority.

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

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

[4] 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.

[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. Formal development associated with this paper. `http://www21.in.tum.de/~blanchet/fouco_devel.zip`, 2014.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, pp. 103–117. Springer, 2010.

[23] 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.

[24] J. Heras, E. Komendantskaya, and M. Schmidt. (co)recursion in logic programming: Lazy vs eager. *Theor. Pract. Log. Prog.*, to appear.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.

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

[35] P. Letouzey. Extraction in Coq: An overview. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, eds., *CiE 2008*, vol. 5028 of *LNCS*, pp. 359–369. Springer, 2008.

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

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

[38] 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. 93–110. Springer, 2014.

[39] 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.

[40] 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.

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

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

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

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

[45] 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.

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

[47] G. Roşu and D. Lucanu. Circular coinduction: A proof theoretical foundation. In *CALCO'09*, pp. 127–144, 2009.

[48] J. Rot, M. M. Bonsangue, and J. J. M. M. Rutten. Coalgebraic bisimulation-up-to. In *SOFSEM*, pp. 369–381, 2013.

[49] F. Ruehr. The evolution of a Haskell programmer. `http://www.willamette.edu/~fruehr/haskell/evolution.html`.

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

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

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

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

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

[55] 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.

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

[57] 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.

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

[59] G. Winskel. A note on model checking the modal $\nu$-calculus. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, eds., *ICALP89*, vol. 372 of *LNCS*, pp. 761–772. Springer, 1989.