Implicit complexity for coinductive data: a characterization of corecurrence

Daniel Leivant
Indiana University and Loria Nancy
leivant@indiana.edu

Ramyaa Ramyaa
Indiana University and Universitat Munchen
ramyaa@indiana.edu

We propose a framework for reasoning about programs that manipulate coinductive data as well as inductive data. Our approach is based on using equational programs, which support a seamless combination of computation and reasoning, and using productivity (fairness) as the fundamental assertion, rather than bi-simulation. The latter is expressible in terms of the former.

As an application to this framework, we give an implicit characterization of corecurrence: a function is definable using corecurrence iff its productivity is provable using coinduction for formulas in which data-predicates do not occur negatively. This is an analog, albeit in weaker form, of a characterization of recurrence (i.e. primitive recursion) in [13].

1 Introduction

Coinductive data has been recognized for nearly two decades as a powerful framework for dealing with infinite objects of evolving and computational nature, such as streams, and — more generally — the behavior of unbounded processes and dynamic systems.

We consider computation over “data-systems”, in which data-types may be defined both inductively and co-inductively. As our main computation model we use equational programs, since these have immediate kinship with formal theories: a program’s equations can be viewed as axioms, and computations are simply derivations in equational logic. In the first part of this paper we develop some building blocks for this project. We consider the global semantics of programs \( P \) over a data-system, that is their behavior as “uninterpreted programs” over all structures for the vocabulary of the data-system. This approach was developed for inductive data in [12]; here we extend it to data-systems in general, including coinductive constructions. It is orthogonal to category theoretical methods in the study of coinduction, which seek to characterize the intended (canonical) model.

An important benefit of streamlined proof systems for reasoning about programs is their use for characterizing major computational complexity classes. Such characterizations fall within the realm of implicit computational complexity, where one delineates complexity classes without reference to computational resources such as time and space. In particular, there are illuminating characterizations of complexity classes in terms of the strength of proof methods needed to prove termination (see e.g. [3,10,13]). Such results lend insight into the significance of complexity classes, provide natural frameworks for programming within given complexity boundaries, and yield static analysis tools for guaranteeing complexity. Implicit characterizations have further potential benefit for coinductive data, because they might clarify complexity notions that are dual to traditional notions of computational complexity such as Polynomial Time.

The primitive recursive functions over the set \( \mathbb{N} \) of natural numbers were characterized proof theoretically already by Parsons [18], who proved that a function is primitive recursive iff it is provable in Peano’s Arithmetic with induction restricted to existential formulas.
In [11, 12] we developed intrinsic theories, a generic framework for reasoning about equational computing over inductive data, and in [13] we used it to characterize the primitive recursive functions in terms of induction for a particular class of formulas. Call a formula unipolar if it does not use data-predicates (i.e. references to data) in both positive and negative position; an example are the positive formulas, in which data-predicates do not occur in a negative position. In [13] we proved that a computable function is primitive recursive iff it is provably correct in the intrinsic theory for \( \mathbb{N} \) with induction restricted to unipolar formulas. In fact we proved more. The forward implication can refer to a very weak formalism, namely, every primitive recursive function is provable, using minimal logic, by induction for formulas in which data-predicates appear only strictly-positively. On the other hand, for the backwards implication we proved that if a computable function is provable, using classical logic, by induction on unipolar formulas, then it is primitive recursive.

We establish here a dual characterization for coinductive data, but where both implication refer to a weak deductive calculus: a computable function over boolean streams is primitive corecursive (i.e. definable using explicit definitions and corecurrence) iff it is provable using minimal logic, by coinduction for formulas built from only conjunction, disjunction, and existential quantification. At present we do not know whether this result can be strengthen to show that every equational program over streams which is provable, using classical logic and unipolar coinduction is primitive-corecursive.

2 Equational programs over data systems

2.1 Equational programs

We describe a generic framework for data-types that are defined using induction, coinduction, or a mix thereof. Such frameworks are well-known for typed lambda calculi, with operators \( \mu \) for smallest fixpoint and \( \nu \) for greatest fixpoint. Our present approach is to express computational behavior of programs via global semantics, thereby dispensing with partial functions; and to define types semantically, via first order axiomatics, dispensing with explicit fixpoint operators.

A constructor-vocabulary is a finite set \( C \) of function identifiers, referred to as constructors, each assigned an arity \( \geq 0 \) (as usual, constructors of arity 0 are object-identifiers). We posit an infinite set \( X \) of variables, and an infinite set \( F \) of function-identifiers, dubbed program-functions, and assigned arities \( \geq 0 \) as well. The sets \( C, X \) and \( F \) are, of course, disjoint.

If \( E \) is a set consisting of function-identifiers and (possibly) variables, we write \( \overline{E} \) for the set of terms containing \( E \) and closed under application: if \( g \in E \) is a function-identifier of arity \( r \), and \( t_1 \ldots t_r \) are terms, then so is \( gt_1 \ldots t_r \). We use informally the parenthesized notation \( g(t_1, \ldots, t_r) \), when convenient.\(^2\)

We refer to elements of \( \overline{C} \), \( C \cup X \) and \( C \cup X \cup F \) as data-terms, base-terms, and program-terms, respectively.\(^3\)

As in [11, 12], we use an equational computation model, in the style of Herbrand-Gödel, familiar from the extensive literature on algebraic semantics of programs. There are easy inter-translations between equational programs and program-terms such as those of FLR\(_0\) [14]. We prefer to focus on equational programs because they integrate easily into logical calculi, and are naturally construed as mathematical theories (with each equation as an axiom). Codifying equations by terms is, in fact, a

\(^1\)Recall that \( \phi \) is a strictly-positive subformula of \( \psi \) if \( \phi \) is not in the scope of a negation or the negative scope of an implication.

\(^2\)In particular, when \( g \) is of arity 0, it is itself a term, whereas with parentheses we have \( g() \) (with \( r = 0 \) arguments) as a term.

\(^3\)Data-terms are often referred to as values, and base-terms as patterns.
conceptual detour, since the computational behavior of such terms is itself spelled out using equations or rewrite-rules.

A program-equation is an equation of the form \( f(t_1 \ldots t_k) = q \), where \( f \) is a program-function of arity \( k \geq 0 \), \( t_1 \ldots t_k \) are base-terms, and \( q \) is a program-term. The left-hand side of a program equation is its definieum. Two program-equations are compatible if their definieum cannot be unified. A program-body is a finite set of pairwise-compatible program-equations. A program \(( P, f )\) (of arity \( k \)) consists of a program-body \( P \) and a program-function \( f \) (of arity \( k \)) dubbed the program’s principal-function. We identify each program with its program-body when in no danger of confusion.

We posit that every program over a given constructor-vocabulary has equations for destructors, as well as a discriminator. That is, if the given vocabulary’s constructors are \( c_1 \ldots c_k \), with \( m \) the maximal arity, then the program-functions include the unary identifiers \( \pi_{i,m} (i = 1 \ldots m) \) and \( \delta_k \), and the program contains the equations (for \( c \) an \( r \)-ary constructor)

\[
\begin{align*}
\pi_{i,m} (c(x_1, \ldots, x_r)) &= x_i & (i = 1 \ldots r) \\
\pi_{i,m} (c(x_1, \ldots)) &= c(x_1, \ldots) & (i = r+1 \ldots m) \\
\delta_k (c_i(\overline{f}), x_1, \ldots, x_k) &= x_i & i = 1 \ldots k
\end{align*}
\]

Thus \( \delta_k \) is a definition-by-cases operation, depending on the main constructor of the first argument. We call a composition of \( n \) destructors (\( n \geq 0 \)) a deep destructor.

It is easy to define the denotational semantics of an equational program for the canonical interpretation of inductive data. If \(( P, f )\) is a program for a unary function over \( \mathbb{N} \), say, then it computes the partial function \( f : \mathbb{N} \rightarrow \mathbb{N} \) where \( f(p) = q \) just in case the equation \( f(\overline{p}) = \overline{q} \) is derivable from \( P \) in equational logic. (We write \( \overline{n} \) for the \( n \)’th numeral, i.e. the data-term \( \overline{s} \ldots \overline{s} \overline{0} \) with \( n \overline{s} \)’s.

The partiality of computable functions is most commonly addressed by either allowing partial structures \([9, 11, 16]\), or by referring to domains, in which an object \( \perp \) denotes divergence. Yet another approach, adopted here, is based on the “global” behavior of programs in all (usual, non-partial) structures. For example, consider the program \( P \) over the constructors \( 0, s \) consisting of the two equations

\[ f(0) = 0 \text{ and } f(ssx) = f(sssx). \]

Thus \( P \) provides no instructions for input 1, and diverges for input \( \geq 2 \). The latter conditions are captured by the statement that there are structures which model the equations \( P \), and where the terms \( f(s0) \) and \( f(ss0) \) are not equal to any numeral.

### 2.2 Global semantics

The concept of global relations, which was present implicitly in mathematical logic for long, came to prominence in Finite Model Theory in the 1980s. Let \( \mathcal{C} \) be a collection of structures. A global relation (of arity \( r \)) over \( \mathcal{C} \) is a mapping \( \mathcal{P} \) that assigns to each structure \( \mathcal{S} \) in \( \mathcal{C} \) an \( r \)-ary relation over the universe \( |\mathcal{S}| \) of \( \mathcal{S} \). For example, if \( \mathcal{C} \) is the collection of all structures over a given vocabulary \( V \), then a first-order \( V \)-formula \( \varphi \), with free variables among \( x_1 \ldots x_r \), defines the predicate \( \lambda x_1 \ldots x_r \varphi \) that to each \( V \)-structure \( \mathcal{S} \) assigns the relations

\[ \{(a_1 \ldots a_r) \mid \mathcal{S}, [\vec{x} := \vec{a}] \models \varphi \} \]

The notion that a formula delineates uniformly subsets of structures is implicit in \([24]\) and \([2]\). Alternative phrases used include generalized relations, data base queries, global relations, global predicates, uniformly defined relations, predicates over oracles, and predicates.)

---

\(^4\)We omit some parentheses for readability.
A global r-ary function over \( \mathcal{C} \) is defined analogously. For example, each typed \( \lambda \)-term of type \( o \rightarrow o \), with identifiers in \( V \) as primitives, defines a global function over the class of \( V \)-structures. E.g., if \( c, f \) and \( g \) are \( V \)-identifiers for functions of arity 0, 1 and 2 respectively, then the term \( \lambda x_1, x_2 \ g(f(x_1), g(x_2, c)) \) defines the global function that to each \( V \)-structure \( \mathcal{S} \) assigns the mapping \( \langle x_1, x_2 \rangle \mapsto g(f(x_1), g(x_2, c)) \), where \( c, f \) and \( g \) are the interpretations in \( \mathcal{S} \) of the identifiers \( c, f \) and \( g \).

The starting point of Descriptive Computational Complexity [7] is that programs used as acceptors define global relations. When those global relations can be defined also by certain logical formulas, one obtains machine-independent characterizations of computational complexity classes. For instance, Fagin [6] and Jones & Selman [8] proved that a predicate \( P \) over finite structures is defined by a program running in nondeterministic polynomial time (NP) iff it is defined by a purely existential second order formula.

Programs of arity 0 can be used to define objects. For example, the singleton program \( T \) consisting of the equation \( t = \text{ss}0 \) defines 3, in the sense that in every model \( \mathcal{S} \) of \( T \) (over a vocabulary with \( t \) as an identifier), the interpretation of the identifier \( t \) is the same as that of the numeral for 3. Consider instead a 0-ary program defining an infinite term (i.e. essentially a stream), for instance the singleton program \( I \) consisting of \( \text{ind} = \text{ss}(\text{ind}) \). This does not have any solution in the free algebra of the unary numerals, that is: the free algebra cannot be expanded into the richer vocabulary with \( \text{ind} \) as a new identifier, so as to satisfy the equation \( I[5] \) But \( I \) is modeled in any structure where \( s \) is interpreted as identity, and \( \text{ind} \) as any structure element. Thus the interpretation of \( \text{ind} \) is not unique. For a more interesting example, consider the structure consisting of countable ordinals, with \( s \) interpreted as the function \( \lambda x. 1 + x \). Then \( I \) holds whenever \( \text{ind} \) is interpreted as an infinite ordinal.

It follows that in our context bi-simulation, while guaranteeing true equality for the canonical model, implies in general only equivalent computational behavior. Indeed, in the global semantic context bi-simulation is not a sound inference rule, since for example two distinct objects can unfold to exactly the same stream of digits (i.e. be observationally equivalent). However, bi-simulation leads to an equivalence relation, which can be captured by a function \( bsm \). Consider the program consisting of the two equations \( b(0 : x, 0 : y) = 0 : b(x,y) \) and \( b(1 : x, 1 : y) = 1 : b(x,y) \). If \( P \) also defines constant identifiers \( a \) and \( b \) as some streams, then we have \( P \models S(a) \wedge S(b) \rightarrow S(b(a, b)) \) just in case there is a bi-simulation between the streams denoted by \( a \) and \( b \), i.e. they are equal as elements of the coalgebra of boolean streams. If the equality \( a = b \) is provable using the traditional coinduction rule for bi-simulation then the implication \( (P) \rightarrow S(b(a, b)) \) is provable in our deductive calculus below. Thus our framework supports all common forms of reasoning about coinductive data.

### 2.3 Semantics of programs

The global semantic approach to equational programs, considered for inductive data in [12], is of interest as an alternative alternative to the “canonical-structure” approach. Under the global semantics approach the notion of correctness of programs is simple, direct, and informative. Here a program over inductive data is said to be correct if it maps, in every structure, inductive data to inductive data. This turns out to be equivalent to the program termination (for all input) in the intended structure (e.g. \( \mathbb{N} \) when the constructors are \( 0 \) and \( s \)). For programs over co-inductive data, which we address here, correctness will turn out to be equivalent to productivity (sometimes dubbed fairness): if the input is a stream, then the program will have a stream as output, without stalling.

The semantics of equational programs for inductive data, such as the natural numbers, is straightfor-
ward. Given a structure $\mathcal{S}$ (for a vocabulary including the constructors in hand), a program $(P, f)$ (unary say) computes the partial function $g : \mathbb{N} \rightarrow \mathbb{N}$ given by: $g(n) = m$ if $P \vdash f(\bar{m}) = \bar{m}$, i.e. the equation is deducible from $P$ in equational logic. (We write $\bar{n}$ for the $n$th unary numeral $\varepsilon([n])$.)

Let $\mathcal{S}$ be a structure whose vocabulary contains at least the constructors in hand. Consider fresh 0-ary identifiers $v_a$, one for each $a \in \mathcal{S}$ (i.e. element of the universe of $\mathcal{S}$). In keeping with the terminology of Model Theory, we define the diagram of $\mathcal{S}$ to be the theory

$$
\text{Diag}(\mathcal{S}) = \left\{ v_a = e(v_{b_1} \cdots v_{b_r}) \mid a = e_{\mathcal{S}}(b_1 \cdots b_r) \text{ e an } r\text{-ary constructor} \right\}
$$

In the presence of coinductive data-types, data may be infinite, and so the operational semantics of equational programs must compute the output piecemeal from finite information about the input. If $\Gamma$ is any set of equations, and $t$ and $t'$ are terms, we write $\Gamma \vdash^\omega t = t'$ if for all deep-destructors $\Pi$ we have (in equational logic) $\Gamma, \text{Diag} \vdash \delta(\Pi(t), \bar{x}) = \delta(\Pi(t'))$. That is, one can establish equationally the observational equivalence of $t$ and $t'$, i.e. the stepwise equality of finite approximations of the two terms.

If $t'$ is a data term, then $\Gamma \vdash^\omega t = t'$ is clearly equivalent (by discourse-level induction on $|t'|$) to $\Gamma, \text{Diag} \vdash t = t'$.

We say that a $k$-ary program $(P, f)$ computes over $\mathcal{S}$ the partial-function $f : \mathcal{S}^k \rightarrow \mathcal{S}$ when for every $\bar{a}, b \in \mathcal{S}$ we have $f(\bar{a}) = b$ just in case $P \cup \text{Diag}(\mathcal{S}) \vdash^\omega f(v_a) = v_b$.

**Examples.** Consider as constructors two unary functions ("successors") 0 and 1. Let $\mathcal{S}$ be the structure of the $\omega$-words over $\{0, 1\}$, with the obvious interpretation of the constructors. Writing $a$ for $(01)^\omega$ and $b$ for $(10)^\omega$, the diagram of $\mathcal{S}$ includes the equations $v_a = 0v_b$, and $v_b = 1v_a$. In this simple case these equations could be used to define $a$, $b$, but if $c$ and $d$ are the binary expansions of $\pi/4$ and $(\pi - 2)/2$, then the equation $v_c = 1v_d$ is also in the diagram, with not much to say about what $c$ and $d$ really are.

The unary program consisting of the two equations $f(0w) = 1f(w), f(1w) = 0f(w)$ defines the function $\text{flip} : \mathcal{S} \rightarrow \mathcal{S}$. We have $\text{flip}((01)^\omega) = (10)^\omega$, because we can easily see that

$$
P, \text{Diag}(\mathcal{S}) \vdash^\omega \text{flip}(v_a) = v_b
$$

We also have for $e = \text{the digitwise flip of } c$ above that

$$
P, \text{Diag}(\mathcal{S}) \vdash^\omega \text{flip}(c) = e
$$

However, as we take deeper destructors for the two terms, the equational proof needed here will use increasingly large (albeit finite) portions of $\text{Diag}(\mathcal{S})$.

### 2.4 Data systems

So far we have considered abstract structures, with no a priori restriction on the behavior of constructor-identifiers. We now proceed to define data-types, needed to reflect the intended computational behavior of programs. We use reserved relation-identifiers (i.e. predicate symbols) for data-types, and convey their defining properties by axioms (closure conditions) rather than via $\mu$ and $v$ fixpoint operators. This allows us to incorporate data types seamlessly into the (first order) deductive machinery.

Descriptive and deductive tools for inductive and coinductive data are not new, of course. For instance, the Common Algebraic Specification Language CASL has been used as a unifying standard in
the algebraic specification community, and extended to coalgebraic data \[20, 21, 15, 22\]. Several frameworks combining inductive and coinductive data, such as \[17\], strive to be comprehensive, including various syntactic distinctions and categories, whereas our approach is minimalist. Such minimalism is made possible by combining the global semantic approach with a semantic (i.e. Curry-style) view of types, by which types indicate semantic properties of pre-existing objects, as opposed to the ontological (Church-style) view, by which types precede objects, with each object coming with a pre-assigned type.

Let \( C = \{ c_1, \ldots, c_k \} \) be a set of constructors as above, where \( c_i \) is of arity \( r_i = \text{arity}(c_i) \). A data-system over \( C \) consists of

1. A list \( D_1 \ldots D_k \) (the order matters) of unary relation-identifiers, where each \( D_n \) is designated as either an inductive-predicate or a coinductive-predicate, and associated a set \( C_n \subseteq C \) of constructors.

2. For each constructor \( c \), of arity \( r \) say, a non-empty finite set of functional types \( \tau \), each of the form \( E_1 \times \cdots \times E_r \rightarrow E_0 \), where each \( E_i \) is one of the \( D_j \)'s. Here we require that no \( E_i \) comes after \( E_0 \) in the given listing of the predicates \( D_j \). We say then that \( c \) has type \( \tau \).

The data-systems defined above do not accommodate simultaneous inductive or coinductive definitions, but a straightforward generalization does.

**Example.** Let \( C \) consist of the identifiers \( 0, 1, [], s, t, \) and \( c \), of arities \( 0,0,0,1,1, \) and \( 2 \), respectively. Consider the following (ordered) list of predicates: inductive predicate \( B \) (for booleans) and \( N \) (natural numbers), coinductive predicates \( J \) (infinite \( s/t \)-words) and \( S \) (streams of natural numbers), and an inductive predicate \( L \) (lists of such streams).

The association of types to constructors is as follows.

\[
\begin{align*}
0 & : B \\
1 & : B \\
[] & : L \\
s & : N \rightarrow N \\
t & : J \rightarrow J \\
c & : N \times S \rightarrow S \\
c & : S \times L \rightarrow L
\end{align*}
\]

Note that constructors are being reused for different data-types. This is in agreement with our untyped, generic approach, where the intended type information is conveyed by the data-predicates. In other words, data-types are explicitly conveyed in the formalism’s syntax as semantic (Curry style) rather than ontological (Church style) properties.

The canonical model \( \mathcal{A} = [\mathcal{D}] \) of a data-system \( \mathcal{D} \) consists of interpretations \( [D_n] \) \((n = 1..k)\) of the data-predicates as sets of finite and infinite terms, obtained by discourse-level recurrence, as follows. If \( D_n \) is inductive, then \( [D_n] \) is the set of terms obtained from \( [D_1] \ldots [D_{n-1}] \) by a finite number of application of the constructors in \( C_n \); dually, if \( D_n \) is coinductive, then \( [D_n] \) is the set of finite and infinite terms obtained from \( [D_1] \ldots [D_{n-1}] \) by such applications. These terms are trees labeled by constructors, where any node labeled by a constructor of arity \( r \) has \( r \) children. Note that if the (non-empty) set \( C_n \) of constructors associated with \( D_n \) has no 0-ary constructors, then for an inductive \( D_n \) the set \( [D_n] \) is empty, whereas for a coinductive \( D_n \) it is a nonempty set of infinite terms.

2.5 Adequacy of Global semantics
Herbrand famously proposed to define the computable functions (over \( \mathbb{N} \)) as those that are unique solutions of equational programs. That definition yields in fact all the hyper-arithmetical functions, a far larger class. But Herbrand was not far off: he only needed to adopt a global approach, rather than restrict attention to the standard structure of the natural numbers. Indeed, in [12] we observed the following. We say that a structure is data-correct for \( \mathbb{N} \) if it interprets the identifier \( \mathbb{N} \) as the set of numeral denotations.

**Theorem 1 (Semantic Adequacy Theorem for Inductive Data)** An equational program \((P, \Gamma)\) over \( \mathbb{N} \) computes a total function iff the formula \( \forall x (x \to \mathbb{N}(\Gamma(x))) \) is true in every model of \( P \) which is data-correct for \( \mathbb{N} \).

The proof in [12] of the nontrivial direction of Theorem 1 proceeds by constructing a “test-model” for the program \( P \). One starts with an extended term model, using the program-functions in \( P \) as well the constructors, and takes the quotient of that term model over the equivalence relation of equality-derived-from \( P \).

### 3 Intrinsic Theories

Intrinsic theories, introduced in [11][12] for inductive data, are skeletal first-order theories whose interest lies in a natural and streamlined formalization of reasoning about equational computing. For example, the intrinsic theory for the natural numbers is suited for incorporating equational programs as axioms, and while it has the same provably computable functions as Peano’s Arithmetic, it has a more immediate formalization of the notion of provable computability. For background, rationale, and examples, we refer to [12].

The intrinsic theory for a data-system \( \mathcal{D} \), \( \text{IT}(\mathcal{D}) \), has

- The rules of \( \mathcal{D} \);
- **Injectiveness axioms** stating that the constructors are injective, i.e. for each \( c \in \mathcal{C} \), of arity \( r \),

\[
\forall x_1 \ldots x_r, y_1 \ldots y_r \ c(\vec{x}) = c(\vec{y}) \rightarrow \bigwedge_i x_i = y_i
\]

- **Separation axioms** stating that the constructors have disjoint images:

\[
\forall \vec{x}, \vec{y} \ c \vec{x} \neq d \vec{y}
\]

for each distinct constructors \( c, d \); and

- For each constructor \( c \), and type \( E_1 \times \cdots \times E_r \rightarrow E_0 \) for \( c \), with \( E_0 \) an inductive predicate, the corresponding clause in the inductive definition of \( E_0 \). That is, the *data-introduction* rule

\[
\begin{array}{c}
E_1(x_1) \cdots E_r(x_r) \\
\hline
E_0(c.x_1 \cdots x_r)
\end{array}
\]

These rules delineate the intended meaning of \( E_0 \) from below.

- For each constructor \( c \), and type \( E_1 \times \cdots \times E_r \rightarrow E_0 \) for \( c \), with \( E_0 \) a co-inductive predicate, the corresponding clause in the co-inductive definition of \( E_0 \). That is, the *data-elimination* rule

\[
\begin{array}{c}
E_0(c.x_1 \cdots x_r) \\
\hline
E_i(x_i)
\end{array}
\]

These rules delineate the intended meaning of a coinductive \( E_0 \) from above.
• For each inductive data-predicate $D_n$ as above, a data-elimination (i.e. Induction) rule: for each formula $\phi \equiv \varphi[z]$, the rule

$$
\frac{D_n(t) \quad \text{Cmp}_n[\phi]}{\varphi[t]}
$$

where

$$
\text{Cmp}_n[\phi] \equiv \frac{\{E^\varphi_i(x_1)\} \cdots \{E^\varphi_r(x_r)\}}{D_n(t) \quad \{\varphi[c(x_1, \ldots, x_r)]\} \quad \text{c : } E_1 \times \cdots \times E_r \rightarrow D_n}
$$

Here $E^\varphi_i(u)$ is $\varphi[u]$ if $E_i$ is $D_n$, and is $E_i(u)$ otherwise. (These open assumptions are closed by the inference.)

That is, if $\varphi[u]$ has the same closure properties under the constructors as $D_n$, then $D_n(t) \rightarrow \varphi[t]$.

• For each coinductive data-predicate $D_n$, a data-introduction (i.e. coinduction) rule: for each formula $\varphi[z]$,

$$
\frac{\varphi[t] \quad \text{Dcm}_n[\phi]}{D_n(t)}
$$

where

$$
\text{Dcm}_n[\phi] \equiv \frac{\{\varphi[x]\}}{\sqrt{\{\exists z_1 \ldots z_r. (\land E_i^\varphi_i(z_i)) \land x = c(\bar{z}) \mid c : E_1 \times \cdots \times E_r \rightarrow D_n\}}
$$

(Here $Q^\varphi$ is defined as for the induction template above.)

That is, if $\varphi$ has the same closure properties under data decomposition (i.e. the destructors) as $D_n$, then $\varphi[t] \rightarrow D_n(t)$.

**Note.** Since our approach here is generic to all structures, the bounding condition in the statement of Coinduction is necessary. Consider for example the coinductive data $W^{\infty}$ of infinite 0-1 words, i.e. the coinductive data predicate built from unary function identifiers 0 and 1, considered above. Taking the eigen formula $\varphi$ of Coinduction to be $x = x$, we would get, absent the bounding condition, $\forall x W^{\infty}(x)$, which is not valid in models of the intrinsic theory for $W$.

From the injectiveness and separation axioms it follows that it is innocuous to use identifiers for destructors and discriminator functions, as above.

Theorem [1] justifies a concept of provable correctness of programs: $(P, f)$ is provably correct in a given formal theory if the formula above is not merely true in all data-correct models of $P$, but is indeed provable in the intrinsic theory $\text{IT}(\varphi)$ from (the universal closure of) $P$, as an axiom.

### 4 Corecurrence and strictly-positive coinduction

#### 4.1 Functions definition by corecurrence

A function definition by recurrence uses its input by eager evaluation: it consumes the top constructor of the input to select the definition-case, and proceeds to consume that constructor’s arguments. That is, for

7We use the bracket notation $\varphi[t]$ to stand for the correct substitution in $\varphi$ of $t$ for the free occurrences of some fixed variable $z$.
each constructor \( c \), one has a clause
\[
f(c(x_1 \ldots x_r), \vec{y}) = g_c(e_1 \ldots e_r, \vec{y}) \quad r = \text{arity}(c) \quad e_i = \text{df} f(x_i, \vec{y})
\] (2)

Here each \( g_c \) is a previously defined function of appropriate arity. Using a discriminator \textit{case} function, the template above can be summarized as
\[
f(x, \vec{y}) = \text{case}(x, e_1 \ldots e_k) \\
e_i = \text{df} f(\pi_i(x), \vec{x})
\]
(Recall that \( \pi_i \) is the \( i \)th destructor.)

Dually, a definition by \textit{corecurrence} builds up the output: it produces the top constructor of the output, and proceeds to produce that constructor’s arguments:
\[
f(\vec{x}) = c_h(\vec{x}, e_1 \ldots e_r) \\
r = \text{arity}(h(\vec{x})) \\
e_i = \text{df} f(\vec{g}_i(\vec{x}))
\] (3)

This template can be summarized by
\[
f(\vec{x}) = \text{cocase}(h(\vec{x}), e_1 \ldots e_k) \\
e_i = \text{df} f(\vec{g}_i(\vec{x}))
\]
where \( \text{cocase}(u, \vec{v}) \) returns the main constructor \( c \) of \( u \), of arity \( r \) say, applied to the first \( r \) of the remaining arguments \( \vec{v} \).

More generally, we use corecurrence to define as above not a single function \( f \), but a vector \( \vec{f} = \langle f_1 \ldots f_k \rangle \) of functions:
\[
f_j(\vec{x}) = \text{cocase}(h_j(\vec{x}), e_1 \ldots e_k) \\
e_i = \text{df} f(\vec{g}_{ij}(\vec{x}))
\]

The distinction in (2) between the recurrence argument and the parameters \( \vec{y} \) disappears in (3) because the focus of the definition shifts to the output, which plays a role analogous to the recurrence argument of the recurrence schema.

When we have just one constructor, e.g. a binary function \textit{cons}, the output’s main constructor need not be specified, and (3) can be conveyed by applying destructors to the output:
\[
\pi_i f(\vec{x}) = f(\vec{g}_i(\vec{x})) \quad i = 0, 1
\] (4)

Such use of destructors is common in presentations of corecurrence, but it fails to capture corecurrence for arbitrary coinductive data. Of course, each case can be coded using streams, just as all inductive data can be coded using the natural numbers.

In our untyped setting the values \( f(\vec{g}_0(\vec{x})) \) and \( f(\vec{g}_1(\vec{x})) \) have the same standing. Streams over a finite base set \( A \) can be construed as a restricted form of (3), with each \( a \in A \) taken as a nullary constructor, and requiring the first argument of \textit{cons} to be one of these constructors.

A function over the given data-system is \textit{primitive corecursive} if it is generated from the constructors and destructors by composition and corecurrence.

\textbf{Example.} Boolean streams form a simple data system of the kind mentioned above: \textit{CONS} is the unique non-constant constructor, which we denote by an infixed colon. The remaining constructs are the nullary \textit{0} and \textit{1}, and the data-predicates are the inductive (and finite) \textit{B} (booleans) and the coinductive \textit{S} (streams). The rules are
\[
\begin{array}{cccc}
\text{B}(0) & \text{B}(1) & \text{S}(x : y) & \text{S}(x : y) \\
\text{B}(x) & \text{B}(y) & \text{S}(x) & \text{S}(y) \\
\end{array}
\]
The constructor `cons` has the two destructors `hd : S → B` and `tl : S → S`.

Since there is a single non-constant constructor here, corecursion can be formulated using the destructors, as in the template:

\[
\begin{align*}
\text{hd}(f(x, \bar{y})) &= g_0(x, \bar{y}) \\
\text{tl}(f(x, \bar{y})) &= f(g_1(x, \bar{y}), \bar{y})
\end{align*}
\]

For example, we can define by corecurrence a function `even`:

\[
\text{hd}(\text{even}(x)) = \text{hd}(x); \quad \text{tl}(\text{even}(x)) = \text{even}(\text{tl}(\text{tl}(x))).
\]

The function `even` is productive (i.e. fair, see [23, 5]), in the sense that it maps streams to streams.

More precisely, in every model \( \mathcal{S} \) of the data-system, expanded to interpret `even` while satisfying its equational definition, if \( S(x) \) holds for \( x \) bound to an element \( a \) of \( \mathcal{S} \)'s universe, then \( S(\text{even}(x)) \).

The generic coinduction rule (1) specializes for boolean streams to the following.

\[
\{ \varphi[x] \}
\]

\[
\vdots
\]

\[
\varphi[t] \quad \exists z_0, z_1. (B(z_0) \land \varphi[z_1] \land x = z_0 : z_1)
\]

\[
S(t)
\]

While corecurrence is dual to recurrence, it is computationally weaker in some ways. Recurrence allows coding of computation traces, so that cumulative (course-of-value) recurrence is implementable using simple recurrence. In contrast, a cumulative variant of corecursion, using at any given point the output stream so far, is not captured by standard corecurrence. For example, the definition of the Morse-Thue sequence, \( x = 1 : \text{merge}(x, \text{not}(x)) \), is not a legal corecurrence.

### 4.2 Strictly-positive coinduction captures corecurrence

Consider the intrinsic theory for a coinductive datatype, such as the boolean streams. We call a formula \textit{strongly positive} if built using conjunction, disjunction, and \( \exists \) as the only logical operations. A formula is \textit{unipolar} if it does not have both positive and negative occurrences of data-predicates. As mentioned in the Introduction above, we know that a function over \( \mathbb{N} \) is primitive recursive iff it is provably correct, using classical logic, in the intrinsic theory for \( \mathbb{N} \) with induction restricted to unipolar formulas; and also iff it is provably correct, using minimal logic, in the intrinsic theory for \( \mathbb{N} \) with induction restricted to strongly-positive formulas.

Here we prove for the primitive corecursive functions an analog of the latter characterization. For concreteness and expository economy, we focus on the data-system \( \mathcal{S}m \) consisting of just streams of booleans as data-type, and refer to the intrinsic theory for it, based on minimal logic. We write \( \text{IT}^+ \) for that theory, with coinduction restricted to strictly-positive formulas.

**Proposition 2** If a k-ary \( f \) is defined by corecursion from functions provable in \( \text{IT}^+ \), then \( f \) is provable in \( \text{IT}^+ \).

**Proof.** Suppose that \( f \) is defined by

\[
f(x) = g_0(x) : f(g_1(x))
\]
Let \((P_0, g_0)\) and \((P_1, g_1)\) be programs (with no common function-identifiers) that are provable in \(\text{IT}^+\), with \(\mathcal{D}_0\) a derivation of \(B(g_0(u))\) from \(S(u)\) and \(P_0\), and \(\mathcal{D}_1\) deriving \(S(g_1(u))\) from \(S(u)\) and \(P_1\). Consider \((P, f)\) where \(P = P_0 \cup P_1\) augmented with the corecursive definition of \(f\) from \(g_0\) and \(g_1\). Then \(S(f(x))\) is derived from \(S(x)\) and \(P\), as follows.

Let \(\phi[z]\) be the strictly-positive formula \(\exists y S(y) \land f(y) = z\). Then \(S(f(x))\) is derived from assumptions \(S(x)\) and \(P\) by coinduction on \(\phi\), since the premises of coinduction follow from these assumptions:

- From \(S(x)\) we have \(S(x) \land f(x) = f(x)\), and so \(\phi[f(x)]\).
- Assuming \(\phi[x]\) we have \(S(y) \land f(y) = x\) for some \(y\), i.e. \(g_0(y) : g_1(y) = x\). But \(S(y)\) implies \(B(g_0(y))\) by \(\mathcal{D}_0\), and \(S(g_1(y))\) by \(\mathcal{D}_1\). Using \(\mathcal{D}_0\) and \(\mathcal{D}_1\) for \(u = g_1(y)\), we get from \(S(g_1(y))\) that \(\phi[g_1(y)]\).

Taking \(z_0 = g_0(y)\) and \(z_1 = g_1(y)\), we thus have \(f(x) = z_0 \land B(z_0) \land \phi[z_1]\), concluding the other premise of the coinduction.

\[\square\]

### 4.3 From coinduction to corecurrence

We proceed to show the converse of Proposition\(^8\) namely that corecurrence captures strongly-positive coinduction. If \(P\) is an equational program, let us write \(\text{IT}^+(P)\) for the natural deduction calculus for \(\text{IT}^+\), augmented with the program \(P\) in the guise of an inference rule.\(^3\) If \(t = t'\) is an equation in \(P\), then

\[
\frac{\alpha[t']}{\alpha[t]} \quad \text{and} \quad \frac{\alpha[t]}{\alpha[t']}
\]

are inferences, where \(\alpha\) is any atomic formula. Clearly, a formula \(\phi\) is derivable in \(\text{IT}^+(P)\) from assumptions \(\bar{\phi}\) iff \(\phi\) is derivable in \(\text{IT}^+\) from \(\bar{\phi}\) plus (the universal closure of) \(P\).

A basic observation is the following, where we refer to the usual notion of logical detours in natural deduction derivations [19]. Recall that a logical detour arises when the major premise of an elimination rule (for a logical operator) is derived by an introduction rule.

**Lemma 3** 1. Every derivation of \(\text{IT}^+(P)\) can be converted to a derivation without logical detours.

2. If \(\mathcal{D}\) is a derivation of \(\text{IT}^+(P)\) without logical detours, proving a strongly-positive formula from strongly-positive assumptions, then every formula in \(\mathcal{D}\) is strongly-positive.

**Proof.** Part (1) is proved as for first-order logic [19]. Part (2) follows by a straightforward structural induction, using the fact that coinduction is restricted to strongly-positive formulas, and that the logic is minimal.\[\square\]

We define a relation \(\mathcal{S}, \eta, \sigma \vdash \phi\), i.e. the stream \(\sigma\) realizes the formula \(\phi\) in the interpretation \((\mathcal{S}, \eta)\) consisting of a model of \(\text{IT}^+\) and of \(P\), and an environment \(\eta\) in it. The definition is by induction on \(\phi\). This relation is defined by structural recurrence on the formula \(\phi\). For a stream \(\sigma\) we define the streams \(\sigma_i\) \(i \geq 0\) inductively, jointly with the streams \(\sigma'_i\). The intent is that \(\sigma_0\) consists of the even-positioned entries of \(\sigma\), \(\sigma_1\) of the even-positioned entries of the remaining entries, etc. \(\sigma_0 = \text{even}(\sigma)\), \(\sigma'_0 = \text{odd}(\sigma)\), \(\sigma_{i+1} = \text{even}(\sigma'_i)\), \(\sigma'_{i+1} = \text{odd}(\sigma'_i)\).

- \(\mathcal{S}, \eta, \sigma \vdash S(t) \iff \sigma = [t]_{\mathcal{S}, \eta} X\) and \(\sigma \in S_x\).

---

\(^8\)This deductive style has been used in research on the Curry-Howard morphism for higher-order logic, e.g. [10]; it was dubbed “deduction modulo” in [31] and subsequent works.
\[ S(t) \]

By IH applied to the left sub-derivation, there is a primitive corecursive function \( g(u, v) \) yielding a stream \( \sigma \) realizing \( \varphi[t] \), from an environment \( u \) and realizers \( v \) for the open assumptions. By IH applied to the right sub-derivation, there is a primitive corecursive function \( h(u, u', v, v') \) yielding a stream realizing

\[
\varphi'[x] \equiv \exists z_0, z_1. (B(z_0) \land \varphi[z_1] \land x = z_0 : z_1)
\]

from an environment \( u \), a stream \( u' \) assigned to \( t \), realizers \( v \) for the open assumptions, and a realizer \( v' \) for \( \varphi[x] \) in the environment \( (v, v') \). Let \( j \) and \( j' \) be the functions that extract from a realizer for \( \varphi' \) (in a given environment) the boolean \( z_0 = hd(x) \), and the realizer of \( z_1 = tl(x) \), respectively.

If \( u \) are the variables free in \( \mathcal{D} \), define by corecurrence

\[
r(u, v, w) = j(w) : r(u, v, j'(h(u, v, w)))
\]

Thus, if \( u \) are streams, and \( v \) are realizers for the open assumptions of \( \mathcal{D} \) for the environment \( u \), then

\[
r(u, v, g(u, v))
\]

is the value of \( t \), and therefore a realizer of \( S(t) \), i.e. the conclusion of Lemma 4.

\[ \square \]
**Theorem 5** A function over streams is primitive corecursive iff it is computable by some equational program which is provable in IT$^+$. 

**Proof.** If a function is primitive corecursive then its primitive corecursive definition is provable in IT$^+$, by Proposition \[2\]

Conversely, suppose $f$ is a function computable by some equational programs $(P,f)$ which is provable in IT$^+$, i.e. there is a derivation of IT$^+(P)$ of the formula $S(x) \rightarrow S(f(x))$. From Lemma \[4\] it follows that there is a primitive corecursive program $(P_0,f_0)$ such that in all models $\mathcal{V}$ of $P$, a realizer of $S(\sigma)$, i.e. $\sigma$ itself, is mapped by $f_0$ to a realizer of $S(f(\sigma))$, i.e. the value of $f(\sigma)$ in the structure. Since $f$ is computed by $P$ in the canonical structure, the above holds there too, i.e. $f$ is primitive-corecursive in the canonical structure. 

\[\square\]

**References**


Implicit corecurrence


