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.

Jean-Yves Marion (Ed.): Second Workshop on
Developments in Implicit Computational Complexity (DICE 2011)
EPTCS 75, 2012, pp. 1–14, doi:10.4204/EPTCS.75.1

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.[1] 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 μ for smallest fixpoint and ν 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 $\mathscr{C}$ of function identifiers, referred to as *constructors*, each assigned an *arity* ≥ 0 (as usual, constructors of arity 0 are *object-identifiers*). We posit an infinite set $\mathscr{X}$ of *variables*, and an infinite set $\mathscr{F}$ of function-identifiers, dubbed *program-functions*, and assigned arities ≥ 0 as well. The sets $\mathscr{C}$, $\mathscr{X}$ and $\mathscr{F}$ are, of course, disjoint.

If $\mathscr{E}$ is a set consisting of function-identifiers and (possibly) variables, we write $\overline{\mathscr{E}}$ for the set of terms containing $\mathscr{E}$ and closed under application: if $g \in \mathscr{E}$ is a function-identifier of arity r, and $t_1 \ldots t_r$ are terms, then so is $g\,t_1 \cdots t_r$. We use informally the parenthesized notation $g(t_1, \ldots, t_r)$, when convenient.[2] We refer to elements of $\mathscr{C}$, $\overline{\mathscr{C} \cup \mathscr{X}}$ and $\overline{\mathscr{C} \cup \mathscr{X} \cup \mathscr{F}}$ 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 $\mathbf{FLR}_0$ [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 φ is a strictly-positive subformula of ψ if φ 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 $\mathbf{f}(\mathbf{t}_1 \dots \mathbf{t}_k) = \mathbf{q}$, where $\mathbf{f}$ is a program-function of arity $k \geq 0$, $\mathbf{t}_1 \dots \mathbf{t}_k$ are base-terms, and $\mathbf{q}$ is a program-term. The left-hand side of a program equation is its *definiendum*. Two program-equations are *compatible* if their definiendums cannot be unified. A *program-body* is a finite set of pairwise-compatible program-equations. A program $(P, \mathbf{f})$ (of arity k) consists of a program-body P and a program-function $\mathbf{f}$ (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 $\mathbf{c}_1 \dots \mathbf{c}_k$, with m the maximal arity, then the program-functions include the unary identifiers $\pi_{i,m}$ ($i = 1..m$) and δ_k, and the program contains the equations (for $\mathbf{c}$ an r-ary constructor)

$$\begin{array}{rcll} \pi_{i,m}(\mathbf{c}(x_1,\dots,x_r) & = & x_i & (i = 1..r) \\ \pi_{i,m}(\mathbf{c}(x_1,\dots,x_r)) & = & \mathbf{c}(x_1,\dots,x_r) & (i = r{+}1..m) \\ \delta_k(\mathbf{c}_i(\vec{t}),x_1,\dots,x_k) & = & x_i & i = 1..k \end{array}$$

Thus δ_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 $(P, \mathbf{f})$ is a program for a unary function over $\mathbb{N}$, say, then it computes the partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ where $f(p) = q$ just in case the equation $\mathbf{f}(\bar{p}) = \bar{q}$ is derivable from P in equational logic. (We write $\bar{n}$ for the n'th numeral, i.e. the data-term $\mathbf{ss}\cdots\mathbf{s0}$ with n $\mathbf{s}$'s.

The partiality of computable functions is most commonly addressed by either allowing partial structures [9, 1, 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 $\mathtt{0}, \mathtt{s}$ consisting of the two equations[4] $\mathtt{f}(\mathtt{0}) = \mathtt{0}$ and $\mathtt{f}(\mathtt{ss}x) = \mathtt{f}(\mathtt{sss}x)$. Thus P provides no instructions for input 1, and diverges for input ≥ 2. The latter conditions are captured by the statement that there are structures which model the equations P, and where the terms $\mathtt{f}(\mathtt{s0})$ and $\mathtt{f}(\mathtt{ss0})$ 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 $\mathscr{C}$ be a collection of structures. A *global relation* (of arity r) over $\mathscr{C}$ is a mapping $\mathscr{P}$ that assigns to each structure $\mathscr{S}$ in $\mathscr{C}$ an r-ary relation over the universe $|\mathscr{S}|$ of $\mathscr{S}$. For example, if $\mathscr{C}$ is the collection of all structures over a given vocabulary V, then a first-order V-formula φ, with free variables among $x_1 \dots x_r$, defines the predicate $\lambda x_1 \dots x_r \varphi$ that to each V-structure $\mathscr{S}$ assigns the relations

$$\{\langle a_1 \dots a_r \rangle \mid \mathscr{S}, [\vec{x} := \vec{a}] \models \varphi\}$$

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 $\mathscr{C}$ is defined analogously. For example, each typed λ-term of type $o \to o$, with identifiers in V as primitives, defines a global function over the class of V-structures. E.g., if $\mathtt{c}$, $\mathtt{f}$ and $\mathtt{g}$ are V-identifiers for functions of arity 0,1 and 2 respectively, then the term $\lambda x_{,1}, x_2\ \mathtt{g}(\mathtt{f}(x_1), \mathtt{g}(x_2, \mathtt{c}))$ defines the global function that to each V-structure $\mathscr{S}$ 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 $\mathscr{S}$ of the identifiers $\mathtt{c}, \mathtt{f}$ and $\mathtt{g}$.

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 $\mathscr{P}$ 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 $\mathtt{t} = \mathtt{sss0}$ defines 3, in the sense that in every model $\mathscr{S}$ of T (over a vocabulary with $\mathtt{t}$ as an identifier), the interpretation of the identifier $\mathtt{t}$ 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 $\mathtt{ind} = \mathtt{s}(\mathtt{ind})$. 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 $\mathtt{ind}$ as a new identifier, so as to satisfy the equation I.[5] But I is modeled in any structure where $\mathtt{s}$ is interpreted as identity, and $\mathtt{ind}$ as any structure element. Thus the interpretation of $\mathtt{ind}$ is not unique. For a more interesting example, consider the structure consisting of countable ordinals, with $\mathtt{s}$ interpreted as the function $\lambda x.1 + x$. Then I holds whenever $\mathtt{ind}$ 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 $\mathbf{b}(0:x, 0:y) = 0:\mathbf{b}(x,y)$ and $\mathbf{b}(1:x, 1:y) = 1:\mathbf{b}(x,y)$. If P also defines constant identifiers $\mathtt{a}$ and $\mathtt{b}$ as some streams, then we have $P \models S(\mathtt{a}) \wedge S(\mathtt{b}) \to S(\mathbf{b}(\mathtt{a}, \mathtt{b}))$ just in case there is a bi-simulation between the streams denoted by $\mathtt{a}$ and $\mathtt{b}$, i.e. they are equal as elements of the coalgebra of boolean streams. If the equality $\mathtt{a} = \mathtt{b}$ is provable using the traditional coinduction rule for bi-simulation then the implication $(P) \to S(\mathbf{b}(\mathtt{a}, \mathtt{b}))$ 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 $\mathtt{0}$ and $\mathtt{s}$). 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-

[5] As usual, when a structure is an expansion of another they have the same universe.

ward. Given a structure $\mathscr{S}$ (for a vocabulary including the constructors in hand), a program $(P,\mathbf{f})$ (unary say) computes the partial function $g:\ \mathbb{N}\rightharpoonup\mathbb{N}$ given by: $g(n)=m$ iff $P\vdash \mathbf{f}(\bar{n})=\bar{m}$, i.e. the equation is deducible from P in equational logic. (We write $\bar{n}$ for the n'th unary numeral $\mathtt{s}^{[n]}(\mathtt{0})$.)

Let $\mathscr{S}$ be a structure whose vocabulary contains at least the constructors in hand. Consider fresh 0-ary identifiers $\mathtt{v}_a$, one for each $a\in|\mathscr{S}|$ (i.e. element of the universe of $\mathscr{S}$). In keeping with the terminology of Model Theory, we define the *diagram* of $\mathscr{S}$ to be the theory[6]

$$\begin{array}{rcl}\mathrm{Diag}(\mathscr{S}) & = & \{\mathtt{v}_a=\mathbf{c}(\mathtt{v}_{b_1}\cdots\mathtt{v}_{b_r})\ \mid \\ & & \qquad a=\mathbf{c}_{\mathscr{S}}(b_1\cdots b_r)\quad \mathbf{c}\text{ an }r\text{-ary constructor}\ \}\end{array}$$

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 Γ is any set of equations, and $\mathbf{t}$ and $\mathbf{t}'$ are terms, we write $\Gamma\vdash^{\omega}\mathbf{t}=\mathbf{t}'$ if for all deep-destructors Π we have (in equational logic) $\Gamma,\ \mathrm{Diag}\vdash\delta((\Pi(\mathbf{t}),\vec{x})=\delta((\Pi(\mathbf{t}'))$. That is, one can establish equationally the observational equivalence of $\mathbf{t}$ and $\mathbf{t}'$, i.e. the stepwise equality of finite approximations of the two terms.

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

We say that a k-ary program $(P,\mathbf{f})$ *computes over* $\mathscr{S}$ the partial-function $f:|\mathscr{S}|^k\rightharpoonup|\mathscr{S}|$ when for every $\vec{a},b\in|\mathscr{S}|$ we have $f(\vec{a})=b$ just in case $P\cup\mathrm{Diag}(\mathscr{S})\vdash^{\omega}\mathbf{f}(\mathtt{v}_a)=\mathtt{v}_b$.

Examples. Consider as constructors two unary functions ("successors") $\mathtt{0}$ and $\mathtt{1}$. Let $\mathscr{S}$ be the structure of the ω-words over $\{0,1\}$, with the obvious interpretation of the constructors. Writing a for $(01)^{\omega}$ and b for $(10)^{\omega}$, the diagram of $\mathscr{S}$ includes the equations $v_a=0v_b$, and $v_b=1v_a$. In this simple case these equations could be used to define a and 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 $\mathbf{f}(0w)=1\,\mathbf{f}(w)$, $\mathbf{f}(1w)=0\,\mathbf{f}(w)$ defines the function $\mathit{flip}:\ |\mathscr{S}|\to|\mathscr{S}|$. We have $\mathit{flip}((01)^{\omega})=(10)^{\omega}$, because we can easily see that

$$P,\ v_a=0v_b,\ v_b=1v_a\vdash^{\omega}\mathit{flip}(v_a)=v_b$$

We also have for $e=$ the digitwise flip of c above that

$$P,\ \mathrm{Diag}(\mathscr{S})\vdash^{\omega}\mathit{flip}(c)=e$$

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

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 μ and ν 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

[6] We write $\mathbf{c}_{\mathscr{S}}$ for the interpretation of the identifier $\mathbf{c}$ in the structure $\mathscr{S}$.

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 $\mathscr{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_k\}$ be a set of constructors as above, where $\mathbf{c}_i$ is of arity $r_i = \underline{arity}(\mathbf{c}_i)$. A *data-system* over $\mathscr{C}$ 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 $\mathscr{C}_n \subseteq \mathscr{C}$ of constructors.
2. For each constructor $\mathbf{c}$, of arity r say, a non-empty finite set of *functional types* τ, each of the form $E_1 \times \cdots \times E_r \to E_0$, 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_i. We say then that $\mathbf{c}$ *has type* τ.

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

Example. Let $\mathscr{C}$ consist of the identifiers $\mathtt{0}, \mathtt{1}, [], \mathtt{s}, \mathtt{t}$, and $\mathtt{c}$, 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 $\mathtt{s}/\mathtt{t}$-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.

$$\begin{array}{l} \mathtt{0} : B \quad \mathtt{0} : N \\ \mathtt{1} : B \\ [] : L \\ \mathtt{s} : N \to N \quad \mathtt{s} : J \to J \\ \mathtt{t} : J \to J \\ \mathtt{c} : N \times S \to S \\ \mathtt{c} : S \times L \to L \end{array}$$

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 onthological (Church style) properties. □

The *canonical model* $\mathscr{A} = [\![\mathscr{D}]\!]$ of a data-system $\mathscr{D}$ 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 $[\![D_1]\!]$ … $[\![D_{n-1}]\!]$ by a finite number of application of the constructors in $\mathscr{C}_n$; dually, if D_n is coinductive, then $[\![D_n]\!]$ is the set of finite and infinite terms obtained from $[\![D_1]\!]$ … $[\![D_{n-1}]\!]$ 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 $\mathscr{C}_n$ 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 $\mathtt{N}$ as the set of numeral denotations.

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

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 $\mathscr{D}$, $\mathbf{IT}(\mathscr{D})$, has

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

$$\forall x_1 \dots x_r, y_1 \dots y_r \;\; \mathbf{c}(\vec{x}) = \mathbf{c}(\vec{y}) \to \bigwedge_i x_i = y_i$$

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

$$\forall \vec{x}, \vec{y} \;\; \mathbf{c}\vec{x} \neq \mathbf{d}\vec{y}$$

  for each distinct constructors $\mathbf{c}, \mathbf{d}$; and

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

$$\frac{E_1(x_1) \quad \cdots \quad E_r(x_r)}{E_0(\mathbf{c}x_1 \cdots x_r)}$$

  These rules delineate the intended meaning of E_0 from below.

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

$$\frac{E_0(\mathbf{c}x_1 \cdots x_r)}{E_i(x_i)}$$

  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[7] $\varphi \equiv \varphi[z]$, the rule

$$\frac{D_n(\mathbf{t}) \quad Cmp_n[\varphi]}{\varphi[\mathbf{t}]}$$

where

$$Cmp_n[\varphi] \quad \equiv \quad \frac{D_n(\mathbf{t}) \quad \left\{ \begin{array}{c} \{E_1^\varphi(x_1)\} \cdots \{E_r^\varphi(x_r)\} \\ \vdots \\ \varphi[\mathbf{c}(x_1 \cdots x_r)] \end{array} \right\}_{c:E_1\times\cdots\times E_r\to D_n}}{\varphi[\mathbf{t}]}$$

Here $E_i^\varphi(u)$ 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 $D_n(\mathbf{t}) \to \varphi[\mathbf{t}]$.

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

$$\frac{\varphi[\mathbf{t}] \quad Dcm_n[\varphi]}{D_n(\mathbf{t})} \tag{1}$$

where

$$Dcm_n[\varphi] \quad \equiv \quad \begin{array}{c} \{\varphi[x]\} \\ \vdots \\ \bigvee \{ \exists z_1 \ldots z_r.(\wedge_i E_i^\varphi(z_i)) \wedge x = \mathbf{c}(\vec{z}) \mid \mathbf{c} : E_1 \times \cdots \times E_r \to D_n \} \end{array}$$

(Here Q_i^φ is defined as for the induction template above.)

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

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^∞ 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 φ 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: $(P, \mathbf{f})$ 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 $\mathbf{IT}(\mathscr{D})$ 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

[7] We use the bracket notation $\varphi[t]$ to stand for the correct substitution in φ of t for the free occurrences of some fixed variable z.

each constructor **c**, one has a clause

$$f(\mathbf{c}(x_1 \dots x_r), \vec{y}) \quad = \quad g_c(e_1 \dots e_r, \vec{y}) \qquad r = \underline{\text{arity}}(\mathbf{c}) \quad e_i =_{\text{df}} f(x_i, \vec{y}) \tag{2}$$

Here each g_c is a previously defined function of appropriate arity. Using a discriminator *case* function, the template above can be summarized as

$$\begin{aligned} f(x, \vec{y}) \quad &= \quad case(x, e_1 \dots e_k) \\ &\qquad\qquad e_i =_{\text{df}} f(\pi_i(x), \vec{x}) \end{aligned}$$

(Recall that π_i is the i'th destructor.)

Dually, a definition by *corecurrence* builds up the output: it produces the top constructor of the output, and proceeds to produce that constructor's arguments:

$$f(\vec{x}) = c_h(\vec{x}, e_1 \dots e_r) \qquad \begin{array}{rcl} r & = & \underline{\text{arity}}(h(\vec{x})) \\ e_i & =_{\text{df}} & f(\vec{g}_i(\vec{x})) \end{array} \tag{3}$$

This template can be summarized by

$$f(\vec{x}) = cocase(h(\vec{x}), e_1 \dots e_k) \qquad e_i =_{\text{df}} f(\vec{g}_i(\vec{x}))$$

where $cocase(u, \vec{v})$ 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 $\vec{f} = \langle f_1 \dots f_k \rangle$ of functions:

$$f_j(\vec{x}) = cocase(h_j(\vec{x}), e_1 \dots e_k) \qquad e_i =_{\text{df}} f_{\ell_i}(\vec{g}_{ij}(\vec{x}))$$

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 *cons*, the output's main constructor need not be specified, and (3) can be conveyed by applying destructors to the output:

$$\pi_i(f(\vec{x})) \quad = \quad f(\vec{g}_i(\vec{x})) \quad i = 0, 1 \tag{4}$$

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 *cons* to be one of these constructors.

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

Example. Boolean streams form a simple data system of the kind mentioned above: CONS is the unique non-constant constructor, which we denote by an infixed colon. The remaining constructs are the nullary **0** and **1**, and the data-predicates are the inductive (and finite) B (booleans) and the coinductive S (streams). The rules are

$$\frac{}{B(\mathbf{0})} \qquad \frac{}{B(\mathbf{1})} \qquad \frac{S(x:y)}{B(x)} \qquad \frac{S(x:y)}{S(y)}$$

The constructor *cons* has the the two destructors $hd : S \to B$ and $tl : S \to S$.

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

$$\begin{aligned} hd(f(x,\vec{y})) &= g_0(x,\vec{y}) \\ tl(f(x,\vec{y})) &= f(g_1(x,\vec{y}),\vec{y}) \end{aligned}$$

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

$$hd(even(x)) = hd(x); \quad tl(even(x)) = even(tl(tl(x))).$$

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 $\mathscr{S}$ 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 $\mathscr{S}$'s universe, then $S(even(x))$.

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

$$\frac{\begin{array}{c}\{\varphi[x]\} \\ \vdots \\ \varphi[\mathbf{t}] \quad \exists z_0, z_1.(B(z_0) \wedge \varphi[z_1] \wedge x = z_0 : z_1\end{array}}{S(\mathbf{t})} \tag{5}$$

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, $x = 1 : merge(x, not(x))$, 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 *strongly positive* if built using conjunction, disjunction, and $\exists$ as the only logical operations. A formula is *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 $\mathscr{S}m$ consisting of just streams of booleans as data-type, and refer to the intrinsic theory for it, based on minimal logic. We write $\mathbf{IT}^+$ 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 $\mathbf{IT}^+$, then f is provable in $\mathbf{IT}^+$.*

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 $\mathbf{IT}^+$, with $\mathscr{D}_0$ a derivation of $B(g_0(u))$ from $S(u)$ and P_0, and $\mathscr{D}_1$ deriving $S(g_1(u))$ from $S(u)$ and P_1. Consider (P, f) where P is $P_0 \cup P_1$ 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 $\varphi[z]$ be the strictly-positive formula $\exists y S(y) \wedge f(y) = z$. Then $S(f(x))$ is derived from assumptions $S(x)$ and P by coinduction on φ, since the premises of coinduction follow from these assumptions:

- From $S(x)$ we have $S(x) \wedge f(x) = f(x)$, and so $\varphi[f(x)]$.
- Assuming $\varphi[x]$ we have $S(y) \wedge f(y) = x$ for some y, i.e. $g_0(y) : g_1(y) = x$. But $S(y)$ implies $B(g_0(y))$ by $\mathscr{D}_0$, and $S(g_1(y))$ by $\mathscr{D}_1$. Using $\mathscr{D}_0$ and $\mathscr{D}_1$ for $u = g_1(y)$, we get from $S(g_1(y))$ that $\varphi[g_1(y)]$. Taking $z_0 = g_0(y)$ and $z_1 = g_1(y)$, we thus have $f(x) = z_0 : z_1 \wedge B(z_0) \wedge \varphi[z_1]$, concluding the other premise of the coinduction.

□

4.3 From coinduction to corecurrence

We proceed to show the converse of Proposition 2, namely that corecurrence captures strongly-positive coinduction. If P is an equational program, let us write $\mathbf{IT}^+(P)$ for the natural deduction calculus for $\mathbf{IT}^+$, augmented with the program P in the guise of an inference rule:[8] If $\mathbf{t} = \mathbf{t}'$ is an equation in P, then

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

are inferences, where α is any atomic formula. Clearly, a formula φ is derivable in $\mathbf{IT}^+(P)$ from assumptions $\vec{\psi}$ iff φ is derivable in $\mathbf{IT}^+$ from $\vec{\psi}$ 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* $\mathbf{IT}^+(P)$ *can be converted to a derivation without logical detours.*

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

We define a relation $\mathscr{S},\ \eta,\ \sigma \Vdash \varphi$, i.e. *the stream* σ *realizes the formula* φ in the interpretation $(\mathscr{S}, \eta)$ consisting of a model of $\mathbf{IT}^+$ and of P, and an environment η in it. The definition is by induction on φ. This relation is defined by structural recurrence on the formula φ. For a stream σ we define the streams σ_i $i \geq 0$ inductively, jointly with the streams σ'_i. The intent is that σ_0 consists of the even-positioned entries of σ, σ_1 of the even-positioned entries of the remaining entries, etc. $\sigma_0 = \mathit{even}(\sigma)$, $\sigma'_0 = \mathit{odd}(\sigma)$, $\sigma_{i+1} = \mathit{even}(\sigma'_i)$, $\sigma'_{i+1} = \mathit{odd}(\sigma'_i)$.

- $\mathscr{S},\ \eta,\ \sigma \Vdash S(\mathbf{t})$ iff $\sigma = [\![\mathbf{t}]\!]_{\mathscr{S},\eta} X$ and $\sigma \in S_{\mathscr{S}}$.

[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 [4] and subsequent works.

- $\mathscr{S},\ \eta,\ \sigma \Vdash \mathbf{t} = \mathbf{t}'$ iff $\sigma = [\![\mathbf{t}]\!]_{\mathscr{S},\eta} X = [\![\mathbf{t}']\!]_{\mathscr{S},\eta} X$.
- $\mathscr{S},\ \eta,\ \sigma \Vdash \varphi_0 \wedge \varphi_1$ iff $\sigma_i \Vdash_{\mathscr{S},\eta} X\varphi_i$, $i = 0, 1$.
- $\mathscr{S}\ \eta,\ \sigma \Vdash \varphi_0 \vee \varphi_1$ iff $\mathscr{S} <\ \eta,\ tl\sigma \Vdash \varphi_{hd\sigma}$.
- $\mathscr{S},\ \eta,\ \sigma \Vdash \exists x\varphi$ iff $\mathscr{S},\ \eta[x := \sigma_0],\ \sigma_1 \Vdash \varphi$.

LEMMA **4** *j Suppose* $\mathbf{IT}^+(P) \vdash \wedge_i \psi_i[\vec{x}] \to \varphi[\vec{x}]$. *Then there is a primitive corecursive function* f_0 *such that for all models* $\mathscr{S}$ *of* P, *and for all streams* $\vec{\tau}$ *and* σ_i, *if*

$$\mathscr{S},\ [\vec{x} := \vec{\tau}],\ \sigma_i \Vdash \psi_i,$$

then

$$\mathscr{S},\ [\vec{x} := \vec{\tau}],\ f_0(\vec{\tau}, \vec{\sigma}) \Vdash \varphi.$$

More precisely, there is a primitive corecursive program P_0 *(which computes* f *above), such that every model of* P *can be expanded to a model of* P_0, *where* f_0 *has the property above.*

Proof. Let $\mathscr{D}$ be a derivation of $\psi[\vec{x}] \to \varphi[\vec{x}]$ in $\mathbf{IT}^+(P)$. By Lemma 3 we may assume that $\mathscr{D}$ is detour-free, and with all formulas strongly-positive. The Lemma is proved by structural induction on $\mathscr{D}$. For the base cases f is the identity. The cases where the main inference of $\mathscr{D}$ is a logical rule are immediate from the definition of $\Vdash$. The cases of Data-elimination rule (decomposition) are immediate since the destructors functions are initial primitive corecursive functions. The case of the rewrite rules based on P is assured by the fact that $\mathscr{S}$ is assumed to be a model of P.

The case of interest is where the main inference rule of $\mathscr{D}$ is Coinduction (for strongly-positive formulas):

$$\frac{\varphi[\mathbf{t}] \quad \exists z_0, z_1.(B(z_0) \wedge \varphi[z_1] \wedge x = z_0 : z_1 \quad \overset{\{\varphi[x]\}}{\cdots}}{S(\mathbf{t})} \tag{6}$$

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

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

from an environment $\vec{u}$, a stream u' assigned to x, realizers $\vec{v}$ for the open assumptions, and a realizer v' for $\varphi[x]$ in the environment $(\vec{v}, v')$. Let j and j' be the functions that extract from a realizer for φ' (in a given environment) the boolean $z_0 = hd(x)$, and the realizer of $z_1 = tl(x)$, respectively.

If $\vec{u}$ are the variables free in $\mathscr{D}$, define by corecurrence

$$r(\vec{u}, \vec{v}, w) = j(w) : r(\vec{u}, \vec{v}, j'(h(\vec{u}, \vec{v}, w)))$$

Thus, if $\vec{u}$ are streams, and $\vec{v}$ are realizers for the open assumptions of $\mathscr{D}$ for the environment $\vec{u}$, then

$$r(\vec{u}, \vec{v}, g(\vec{u}, \vec{v}))$$

is the value of $\mathbf{t}$, and therefore a realizer of $S(\mathbf{t})$, i.e. the conclusion of 5. □

THEOREM **5** *A function over streams is primitive corecursive iff it is computable by some equational program which is provable in* $\mathbf{IT}^+$.

Proof. If a function is primitive corecursive then its primitive corecursive definition is provable in $\mathbf{IT}^+$, by Proposition 2.

Conversely, suppose f is a function computable by some equational programs $(P,\mathbf{f})$ which is provable in $\mathbf{IT}^+$, i.e. there is a derivation of $\mathbf{IT}^+(P)$ of the formula $S(x) \to S(\mathbf{f}(x))$. From Lemma 4 it follows that there is a primitive corecursive program $(P_0,\mathbf{f}_0)$ such that in all models $\mathscr{S}$ of P, a realizer of $S(\sigma)$, i.e. σ itself, is mapped by f_0 to a realizer of $S(\mathbf{f}(x))$, i.e. the value of $\mathbf{f}(x)$ 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. □

References

[1] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella & Andrzej Tarlecki (2002): *CASL: the Common Algebraic Specification Language*. *Theor. Comput. Sci.* 286(2), pp. 153–196.

[2] Jon Barwise & Yanis Moschovakis (1978): *Global inductive definability*. *Journal of Symbolic Logic* 43, pp. 521–534.

[3] Samuel Buss (1986): *Bounded Arithmetic*. Bibliopolis, Naples.

[4] Gilles Dowek, Thérèse Hardin & Claude Kirchner (2003): *Theorem Proving Modulo*. *J. Autom. Reasoning* 31(1), pp. 33–72.

[5] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara & Jan Willem Klop (2007): *Productivity of Stream Definitions*. In Erzsébet Csuhaj-Varjú & Zoltán Ésik, editors: *FCT*, *Lecture Notes in Computer Science* 4639, Springer, pp. 274–287, doi:10.1007/978-3-540-74240-1_24.

[6] Ronald Fagin (1974): *Generalized first order spectra and polynomial time recognizable sets*. In R. Karp, editor: *Complexity of Computation*, SIAM-AMS, pp. 43–73.

[7] Neil Immerman (1989): *Descriptive and Computational Complexity*. In: *FCT*, pp. 244–245.

[8] N.G. Jones & A.L. Selman (1974): *Turing machines and the spectra of first-order formulas*. *Journal of Symbolic Logic* 39, pp. 139–150.

[9] Stephen C. Kleene (1969): *Formalized Recursive Functions and Formalized Realizability*. *Memoirs of the AMS* 89, American Mathematical Society, Providence.

[10] Daniel Leivant (1994): *A foundational delineation of poly-time*. *Information and Computation* 110, pp. 391–420.

[11] Daniel Leivant (1995): *Intrinsic theories and computational complexity*. In D. Leivant, editor: *Logic and Computational Complexity*, LNCS, Springer-Verlag, Berlin, pp. 177–194.

[12] Daniel Leivant (2002): *Intrinsic reasoning about functional programs I: First order theories*. *Annals of Pure and Applied Logic* 114, pp. 117–153, doi:10.1016/S0168-0072(01)00078-1.

[13] Daniel Leivant (2004): *Intrinsic reasoning about functional programs II: unipolar induction and primitive-recursion*. *Theor. Comput. Sci.* 318(1-2), pp. 181–196, doi:10.1016/j.tcs.2003.11.002.

[14] Yiannis N. Moschovakis (1989): *The Formal Language of Recursion*. *J. Symb. Log.* 54(4), pp. 1216–1252, doi:10.2307/2274814.

[15] Till Mossakowski, Lutz Schröder, Markus Roggenbach & Horst Reichel (2006): *Algebraic-coalgebraic specification in CoCasl*. *J. Log. Algebr. Program.* 67(1-2), pp. 146–197, doi:10.1016/j.jlap.2005.09.006. Available at http://dx.doi.org/10.1016/j.jlap.2005.09.006.

[16] Peter D. Mosses (2004): *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*. *Lecture Notes in Computer Science* 2960, Springer, doi:10.1007/b96103.

[17] Peter Padawitz (2000): *Swinging types=functions+relations+transition systems*. *Theor. Comput. Sci.* 243(1-2), pp. 93–165, doi:10.1016/S0304-3975(00)00171-7.

[18] Charles Parsons (1970): *On a number-theoretic choice schema and its relation to induction*. In A. Kino, J. Myhill & R. Vesley, editors: *Intuitionism and Proof Theory*, North-Holland, Amsterdam, pp. 459–473, doi:10.1016/S0049-237X(08)70771-7.

[19] D. Prawitz (1965): *Natural Deduction*. Almqvist and Wiksell, Uppsala.

[20] Horst Reichel (1999): *A Uniform Model Theory for the Specification of Data and Process Types*. In Didier Bert, Christine Choppy & Peter D. Mosses, editors: *WADT*, *Lecture Notes in Computer Science* 1827, Springer, pp. 348–365, doi:10.1007/978-3-540-44616-3_20.

[21] Jan Rothe, Hendrik Tews & Bart Jacobs (2001): *The Coalgebraic Class Specification Language CCSL*. *J. UCS* 7(2), pp. 175–193. Available at http://www.jucs.org/jucs_7_2/the_coalgebraic_class_specification.

[22] Lutz Schröder (2008): *Bootstrapping Inductive and Coinductive Types in HasCASL*. *Logical Methods in Computer Science* 4(4), doi:10.2168/LMCS-4(4:17)2008. Available at http://dx.doi.org/10.2168/LMCS-4(4:17)2008.

[23] Ben A. Sijtsma (1989): *On the Productivity of Recursive List Definitions*. *ACM Trans. Program. Lang. Syst.* 11(4), pp. 633–649, doi:10.1145/69558.69563.

[24] Alfred Tarski (1952): *Some notions and methods on the borderline of algebra and metamathematics*. In: *Proceedings of the International Congress of Mathematicians I*, American Mathematical Society, Providence, RI, pp. 705–720.