Global semantic typing for inductive and coinductive computing

Daniel Leivant

Indiana University Bloomington
leivant@indiana.edu

Abstract

Common data-types, such as $\mathbb{N}$, can be identified with term algebras. Thus each type can be construed as a global set; e.g. for $\mathbb{N}$ this global set is instantiated in each structure $\mathcal{S}$ to the denotations in $\mathcal{S}$ of the unary numerals. We can then consider each declarative program as an axiomatic theory, and assigns to it a semantic (Curry-style) type in each structure. This leads to the *intrinsic theories* of [18], which provide a purely logical framework for reasoning about programs and their types. The framework is of interest because of its close fit with syntactic, semantic, and proof theoretic fundamentals of formal logic.

This paper extends the framework to data given by coinductive as well as inductive declarations. We prove a Canonicity Theorem, stating that the denotational semantics of an equational program P, understood operationally, has type τ over the canonical model iff P, understood as a formula has type τ in every "data-correct" structure. In addition we show that every intrinsic theory is interpretable in a conservative extension of first-order arithmetic.

1998 ACM Subject Classification F.3 Logics and meanings of programs

Keywords and phrases Inductive and coinductive types, equational programs, intrinsic theories, global model theory

Digital Object Identifier 10.4230/LIPIcs.CSL.2013.469

1 Introduction

The notion of program typing, first introduced by Curry [5, 32], views types as semantic properties of pre-existing untyped objects. A function f has type $\tau \to \sigma$ if it maps objects of type τ to objects of type σ; f may well be defined for input values that are not of type τ. In contrast, Church [4] considered types as inherent properties of objects: a function has type $\tau \to \sigma$ when its domain consists of the objects of τ, and its codomain consists of objects of type σ. The difference between semantic and inherent typing is thus ontologically significant in a way that phrases such as "explicit" and "implicit" do not convey.

A distinction between inherent and semantic typing can also be made for inductive data types T, such as the booleans, natural numbers, and strings. While each such data-type has a canonical intended meaning, it is isomorphic to the term algebra over some set C of constructors. That syntactic representation suggests a *global semantics* for such types. Namely, T is a global predicate, that is a mapping that to each structure $\mathcal{S}$ (for a vocabulary V containing C) assigns the set of denotations of closed C-terms.

(Recall that global semantics is an organizing principle for descriptive and computational devices over a class of structures, such as all finite graphs [6, 9]. An example is Fagin's celebrated result that a global relation over finite structures is NP iff it is definable by an existential second-order formula [7].)

The global viewpoint is of particular interest with respect to *programs* over inductive data. Each such program P may be of type $T \to T$ in one V-structure and not in another;

Computer Science Logic 2013 (CSL'13).
Editor: Simona Ronchi Della Rocca; pp. 469–483

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

e.g. if P is non-total on $\mathbb{N}$, then it is of type $\mathtt{N} \to \mathtt{N}$ in the flat-domain structure with $\mathtt{N}$ interpreted as $\mathbb{N}_\perp$, but not when $\mathtt{N}$ is interpreted as $\mathbb{N}$. Already this simple observation resolves the status of an incorrect proposal by Herbrand, by which a set of equations P is said to compute a function $f : \mathbb{N} \to \mathbb{N}$ iff f is the unique solution of P.[1]

This proposal was corrected by Gödel [8], who replaced the Tarskian semantics of a set of equations (i.e. true or false in a given structure) by an operational semantics. But in fact Herbrand was right to equate operational semantics with Tarskian semantics, with one caveat: P computes f in the standard structure iff f has, by Tarskian semantics, type $N \to N$ in *every* model of P. We elaborate on this below.

Within the global framework, it makes sense to consider formal V-theories for proving global typing properties of equational programs. We adopt as programming model equational programs, since these mesh directly with formal reasoning: a program's equations can be construed as axioms, computations as derivations in equational logic, and types as formulas. Moreover, equational programs are amenable to term-model constructions, which turn out to be a useful meta-mathematical tool. Theories for reasoning directly about equational programs were developed in [18], where they were dubbed *intrinsic theories*. Among other benefits, they support attractive proof-theoretic characterizations of major function classes, such as the provable functions of Peano Arithmetic and the primitive recursive functions [18, 19].

In this paper we generalize the global semantics approach of [18] to *data-systems,* that is collections of data-types generated by both inductive and coinductive definitions. To do so we shall start by describing a syntactic framework, in analogy to the term algebras, in which a syntactic representation of the intended data-types is possible. We shall continue by giving an operational semantics for equational programs over data-systems, i.e. when data-objects may be infinite. We then prove a generalized Canonicity Theorem, which states that a program over a data-system is correctly typed in the standard structure (e.g. terminating for inductive output and fair for coinductive output) just in case such typing is correct by Tarskian semantics in all structures. Finally we show that the obvious first-order theories for proving correct-typing of equational programs are no stronger than Peano Arithmetic.

2 Data systems

2.1 Symbolic data

A *constructor-vocabulary* is a finite set $\mathcal{C}$ of function identifiers, referred to as *constructors,* each assigned an *arity* ≥ 0; as usual, constructors of arity 0 are *object-identifiers*. We'll posit the presence of a master constructor-vocabulary, and consider its sub-vocabularies. Given a constructor-vocabulary $\mathcal{C}$, the *replete term-set* for $\mathcal{C}$ is the set $R_\mathcal{C}$ consisting of all finite or infinite ordered trees of constructors, where each node with constructor $\mathbf{c}$ of arity r has exactly r children. Obviously $R_\mathcal{C}$ is definable coinductively, but we will be interested primarily in its subsets, defined both inductively and coinductively.

The *replete $\mathcal{C}$-structure* is the structure $\mathcal{R}_\mathcal{C}$ with[2]

[1] Actually, Herbrand's proposal also called $<$ for a constructive proof that a solution exists and is unique. But that $<$ additional condition is ill-formed, and cannot be replaced by provability in $<$ some intuitionistic theory, since that would imply that the computable total functions $<$ form a semi-decidable collection.

[2] We use typewriter font for actual identifiers, boldface for meta-level variables ranging over syntactic objects, and italics for other meta-level variables.

1. $\mathcal{C}$ as vocabulary (i.e. similarity-type);
2. $R_{\mathcal{C}}$ as universe; and
3. a syntactic interpretation of the constructors: for an r-ary $\mathbf{c} \in \mathcal{C}$ $\quad [\![\mathbf{c}]\!](a_1 \dots a_r)$ $\quad$ is the tree with $\mathbf{c}$ at the root and $a_1 \dots a_r$ as immediate sub-trees.

2.2 Equational programs

In addition to the set $\mathcal{C}$ of constructors we posit an infinite set $\mathcal{X}$ of *variables,* and an infinite set $\mathcal{F}$ of function-identifiers, dubbed *program-functions,* and assigned arities ≥ 0 as well. The sets $\mathcal{C}$, $\mathcal{X}$ and $\mathcal{F}$ are, of course, disjoint. If $\mathcal{E}$ is a set consisting of function-identifiers and (possibly) variables, we write $\bar{\mathcal{E}}$ for the set of terms generated from $\mathcal{E}$ by application: if $\mathbf{g} \in \mathcal{E}$ is a function-identifier of arity r, and $\mathbf{t}_1 \dots \mathbf{t}_r$ are terms, then so is $\mathbf{g}\,\mathbf{t}_1 \cdots \mathbf{t}_r$. We use informally the parenthesized notation $\mathbf{g}(\mathbf{t}_1, \dots, \mathbf{t}_r)$, when convenient.[3] We refer to elements of $\overline{\mathcal{C}}$, $\overline{\mathcal{C} \cup \mathcal{X}}$ and $\overline{\mathcal{C} \cup \mathcal{X} \cup \mathcal{F}}$ as *data-terms, base-terms,* and *program-terms,* respectively.[4]

We adopt equational programs, in the style of Herbrand-Gödel, as computation model. There are easy inter-translations between equational programs and program-terms such as those of $\mathbf{FLR}_0$ [21]. We prefer however to focus on equational programs because they integrate easily into logical calculi, and are naturally construed as axioms. Codifying equations by terms is a 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. Two program-equations are *compatible* if their left-hand sides 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.

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 $\mathcal{S}$ of T 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 such as 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, for example, in any structure where $\mathtt{s}$ is interpreted as identity, and $\mathtt{ind}$ as any structure element. Also, I is modeled over any ordinal $\alpha \succ \omega$, with $\mathtt{s}$ interpreted as the function $x \mapsto 1 + x$ and I as any infinite $\beta \in \alpha$.

2.3 Operational semantics of programs

If $(P, \mathbf{f})$ is a program over the set $\bar{\mathcal{C}}$ of data-terms (which are all finite) then we can say that it computes the partial function $\quad g : \bar{\mathcal{C}} \rightharpoonup \bar{\mathcal{C}} \quad$ when $g(p) = q$ iff the equation $\mathbf{f}(p) = q$ is derivable from P in equational logic. But non-trivial replete term-structures have infinite terms, so the output of a program over $R_{\mathcal{C}}$ must be computed piecemeal from finite information about the input values.

To express piecemeal computation of infinite data, without using extra constants or tools, we posit that each program over $\mathcal{C}$ has defining equations for destructors and a discriminator.

---

[3] In particular, when $\mathbf{g}$ is of arity 0, it is itself a term, whereas with parentheses we'd have $\mathbf{g}()$.
[4] Data-terms are often referred to as *values*, and base-terms as *patterns.*
[5] As usual, when a structure is an expansion of another they have the same universe.

That is, if the given vocabulary's k 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, and all programs contain the equations (for $\mathbf{c}$ an r-ary constructor)

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

We call a repeated composition of destructors a *deep destructor*. For $a \in R_{\mathcal{C}}$ and variable $\mathtt{v}$ let $\Delta_{a/\mathbf{v}}$ consist of all statements $\delta(\Pi(\mathtt{v}), x_1, \dots, x_k) = x_i$ where Π is a deep destructor, that are true when $\mathtt{v} = a$. That is, $\Delta_{a/\mathbf{v}}$ conveys, node by node, the structure of the syntactic tree a, using $\mathtt{v}$ as a name for a.

▶ **Definition 1.** We say that a unary program $(P, \mathbf{f})$ *computes* the partial function $g : R_{\mathcal{C}} \rightharpoonup R_{\mathcal{C}}$ when for every $a, b \in R_{\mathcal{C}}$ we have $g(a) = b$ iff for each deep-destructor Π the equation $\delta(\Pi(\mathbf{f}(\mathtt{v})), \vec{x}) = x_i$ is derivable in equational logic from P and $\Delta_{a/\mathbf{v}}$, where $\mathbf{c}_i$ is the main constructor of $\Pi(b)$.

That is, one can establish in equational logic the equality of $f(a)$ and of b at each "address" Π, given unbounded information about the structure of a.

The definition for programs of arity > 1 is similar.

Note that the piecemeal definition of computability is made necessary only by the presence of infinite data:

▶ Proposition 2. If, in definition (1) above, the terms a, b are finite, then $g(a) = b$ just in case $\mathbf{f}(a) = b$ is derivable from P in equational logic. ◀

The proof is straightforward, and omitted here since Proposition 2 is not used in the sequel.

2.4 Inductive Data systems

To motivate the general definition, let us consider first purely inductive types. One defines a single type by its closure rules: the natural numbers are given by the two rules $\mathtt{N}(\mathtt{0})$ and $\mathtt{N}(x) \to \mathtt{N}(\mathtt{s}(x))$. Similarly, words in $\{0, 1\}$ can be construed as terms using unary constructors $\mathtt{0}$ and $\mathtt{1}$, as well as nullary constructor $\mathtt{e}$, and generated by the rules $\mathtt{W}(\mathtt{e})$, $\mathtt{W}(x) \to \mathtt{W}(\mathtt{0}(x))$ and $\mathtt{W}(x) \to \mathtt{W}(\mathtt{1}(x))$. If $\mathtt{G}$ names a given type G, then the type of binary trees with leaves in G is generated by the rules $\mathtt{G}(x) \to \mathtt{T}(x)$, and $\mathtt{T}(x) \wedge \mathtt{T}(y) \to \mathtt{T}(\mathtt{p}(x, y))$.

We can similarly generate types jointly (i.e. simultaneously). For example, the following rules generate the 01-strings with no adjacent $\mathtt{1}$'s, by defining jointly the set (denoted by $\mathtt{E}$) of such strings that start with $\mathtt{1}$, and the set (denoted by $\mathtt{Z}$) of those that don't: $\mathtt{Z}(\mathtt{e})$, $\mathtt{Z}(x) \to \mathtt{Z}(\mathtt{0}(x))$, $\mathtt{Z}(x) \to \mathtt{E}(\mathtt{1}(x))$, and $\mathtt{E}(x) \to \mathtt{Z}(\mathtt{0}(x))$.

Generally, a definition of inductive types from given types $\vec{\mathbf{G}}$ consists of:

1. A list $\vec{\mathbf{D}}$ of unary relation-identifiers, dubbed *type identifiers*;
2. a set of *composition rules,* of the form

$$\frac{\mathbf{Q}_1(x_1) \quad \cdots \quad \mathbf{Q}_r(x_r)}{\mathbf{D}_i(\mathbf{c}\, x_1 \cdots x_r)}$$

where $\mathbf{c}$ is a constructor, and each $\mathbf{Q}_\ell$ is one of the data-predicates in $\vec{\mathbf{G}}, \vec{\mathbf{D}}$.

These rules delineate the intended meaning of inductive $\vec{\mathbf{D}}$ from below. Namely, elements of $\mathbf{D}_i$ are built up by the composition rules. Thus the data-predicates in $\vec{\mathbf{D}}$ are defined jointly, potentially using also the given types $\vec{\mathbf{G}}$.

Conjuncting the composition rules, we obtain a single rule, consisting of the the universal closure of conjunctions of implications into the data-type being defined.

2.5 Coinductive decomposition rules

Inductive composition rules state sufficient reasons to assert that a term has a given type, implied by the types of its immediate sub-terms. The intended semantics of an inductive type $\mathbf{D}$ is thus the smallest set of terms closed under those rules. Coinductive decomposition rules state necessary conditions for a term to have a given type, by implying the types of its immediate sub-terms. The intended semantics is the largest set of terms satisfying those conditions.

For instance, the type of ω-words over $0/1$ is given by the decomposition rule

$$\mathtt{W}^\omega(x) \to (\exists y\ \mathtt{W}^\omega(y) \wedge x = \mathtt{0}y) \vee (\exists y\ \mathtt{W}^\omega(y) \wedge x = \mathtt{1}y) \tag{1}$$

This is not quite captured by the implications $\mathtt{W}^\omega(\mathtt{0}x) \to \mathtt{W}^\omega(x)$ and $\mathtt{W}^\omega(\mathtt{1}x) \to \mathtt{W}^\omega(x)$, since these do not guarantee that every element of $\mathtt{W}^\omega$ is of the right form.

The implication $\mathtt{W}^\omega(x) \to \mathtt{W}^\omega(\pi_{1,1}(x))$ also fails to capture the rules (1), as shown by the following example. In analogy to the inductive definition above of the words with no adjacent $\mathtt{1}$'s, the ω-words over $0/1$ with no adjacent $\mathtt{1}$'s are delineated jointly by the two decomposition rules

$$\mathtt{Z}(x) \ \to\ (\exists y\ \mathtt{Z}(y) \wedge x = \mathtt{0}(y)) \ \vee\ (\exists y\ \mathtt{E}(y) \wedge x = \mathtt{0}(y))$$

and

$$\mathtt{E}(x) \ \to\ \exists y\ \mathtt{Z}(y) \wedge x = \mathtt{1}(y)$$

These rules cannot be captured using destructors, since those do not differentiate between cases for the input's main constructor.

These observations justify the following definition.

▶ **Definition 3.** A *decomposition definition of coinductive types from given types* $\vec{\mathbf{G}}$ consists of:

1. A list $\vec{\mathbf{D}}$ of *type identifiers*;
2. for each of the types $\mathbf{D}_i$ in $\vec{\mathbf{D}}$ a *decomposition rule,* of the form

$$\mathbf{D}_i(x) \ \to\ \psi_1 \vee \cdots \vee \psi_k$$

where each ψ_i is of the form

$$\exists y_1 \ldots y_r\ x = \mathbf{c}(\vec{y}) \ \wedge\ \mathbf{Q}_1(y_1) \ \wedge \cdots \wedge \mathbf{Q}_r(y_r)$$

Here $\mathbf{c}$ is a constructor of arity r, and each $\mathbf{Q}_\ell$ is one of the data-predicates in $\vec{\mathbf{G}}, \vec{\mathbf{D}}$. Thus, the single decomposition rule for each $\mathbf{D}_i$ is an implication from $\mathbf{D}_i$ to the disjunction of existential statements.

2.6 Data systems

We now define data-systems, where data-types can be defined by any combination of induction and coinduction. Descriptive and deductive tools for such definitions were studied extensively, e.g. referring to typed lambda calculi, with operators μ for smallest fixpoint and

ν for greatest fixpoint. 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 [27, 29, 22, 30]. Several frameworks combining inductive and coinductive data, such as [24], strive to be comprehensive, including various syntactic distinctions and categories, in contrast to our minimalistic approach.

▶ **Definition 4.** A *data-system* $\mathcal{D}$ over a set $\mathcal{C}$ of constructors consists of:

1. A double-list $\vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_k$ (the order matters) of unary relation-identifiers, dubbed *type-identifiers*, where each $\vec{\mathbf{D}}_i$ is dubbed a *data-bundle,* and designated as either *inductive* or *coinductive.*
2. For each inductive data-bundle $\vec{\mathbf{D}}_i$ an *inductive definition* given as a (finite) set of *data-composition* rules of the form

$$(\bigwedge_{\ell=1..r} \mathbf{Q}_\ell(x_\ell)) \to \mathbf{D}_{ij}(\mathbf{c}\, x_1 \cdots x_r)$$

   where each $\mathbf{Q}_\ell$ is one of the data-predicates in $\vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_i$. Note that r may be 0.
   These rules delineate the intended meaning of inductive $\vec{\mathbf{D}}_i$ from below. Namely, elements of $\mathbf{D}_{ij}$ are built up by the data-introduction rules. Thus the data-predicates in each $\vec{\mathbf{D}}_i$ are defined simultaneously, potentially using also previously defined predicates among $\vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_{i-1}$.
3. For each coinductive data-predicate $\mathbf{D}_{ij}$ a *coinductive definition* consisting of a *data-decomposition* rule, of the form

$$\mathbf{D}_{ij}(x) \;\to\; \psi_1 \vee \cdots \vee \psi_k$$

   where each ψ_ℓ is of the form

$$\exists y_1 \dots y_r \; x = \mathbf{c}(\vec{y}) \;\wedge\; \mathbf{Q}_1(y_1) \;\wedge\; \cdots \;\wedge\; \mathbf{Q}_r(y_r)$$

   with $\mathbf{c}$ a constructor, and each $\mathbf{Q}_\ell$ in $\vec{\mathbf{D}}_j$, $j \le i$.

2.7 Examples of data-systems

1. Let $\mathcal{C}$ consist of the identifiers `0`, `1`, `e`, `s`, and `c`, of arities 0,0,0,1, and 2, respectively. Consider the following data-system, for the double list $((\mathtt{B}), (\mathtt{N}), (\mathtt{F}, \mathtt{S}), (\mathtt{L}))$ with inductive `B` and `N` (booleans and natural numbers), coinductive `F` and `S` (streams with alternating boolean and numeral entries starting with booleans (respectively, with natural numbers)), and finally an inductive `L` for lists of such streams.

$$\mathtt{B}(\mathtt{0}) \quad \mathtt{B}(\mathtt{1})$$

$$\mathtt{N}(\mathtt{0}) \quad \mathtt{N}(x) \to \mathtt{N}(\mathtt{s}x)$$

$$\mathtt{F}(x) \to \exists y, z \; (x = \mathtt{c}yz) \;\wedge\; \mathtt{B}(y) \;\wedge\; \mathtt{S}(z)$$
$$\mathtt{S}(x) \to \exists y, z \; (x = \mathtt{c}yz) \;\wedge\; \mathtt{N}(y) \;\wedge\; \mathtt{F}(z)$$

$$\mathtt{L}(\mathtt{0}) \quad \mathtt{L}(\mathtt{e}) \quad \mathtt{F}(x) \wedge \mathtt{L}(y) \to \mathtt{L}(\mathbf{c}xy) \quad \mathtt{S}(x) \wedge \mathtt{L}(y) \to \mathtt{L}(\mathbf{c}xy)$$

   Note that constructors are 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, and the data-objects are untyped. In other words, data-types are semantic (Curry style) rather than ontological (Church style).

2. Here is a data system with a type for infinite binary trees with nodes decorated by finite/infinite binary trees with booleans as leaves. The constructors are `0`, `1`, `p`, and `d` of arities 0,0,2 and 3 respectively. The data-predicates are an inductive `B`, and two coinductive `D` (for trees) and `T` (for trees of trees). The composition rules for `B` are

$$\mathtt{B}(\mathtt{0}) \qquad \text{and} \qquad \mathtt{B}(\mathtt{1})$$

The decomposition rules for `T` and `D` (as a single bundle) are

$$\mathtt{D}(x) \to \mathtt{B}(x) \vee \exists y_1, y_2 \;\; x = \mathtt{p}(y_1, y_2) \quad \wedge \; \mathtt{D}(y_1) \; \wedge \; \mathtt{D}(y_2)$$

and

$$\mathtt{T}(x) \to \exists u, y_1, y_2 \;\; x = \mathtt{d}(u, y_1, y_2) \quad \wedge \; \mathtt{D}(u) \; \wedge \; \mathtt{T}(y_1) \; \wedge \; \mathtt{T}(y_2)$$

2.8 Computational completeness of equational programs

The equivalence of equational programs over $\mathbb{N}$ with the μ-recursive functions was implicit already in [8], and explicit in [13]. Their equivalence with λ-definability [3, 14] and hence with Turing computability [35] followed quickly. When equational programs are used over infinite data, a match with Turing machines must be based on an adequate representation of infinite data by functions over inductive data. For instance, each infinite 0/1 word w can be identified with the function $\hat{w} : \mathbb{N} \to \mathbb{B}$ defined by $\hat{w}(k) =$ the k'th constructor of w. Similarly, infinite binary trees with node decorated with 0/1 can be identified with functions from $\mathbb{W} = \{\mathtt{0}, \mathtt{1}\}^*$ to $\{\mathtt{0}, \mathtt{1}\}$. Conversely, a function $f : \mathbb{N} \to \mathbb{B}$ can be identified with the ω-word $\check{f}$ whose n'th entry is $f(n)$.

It follows that a functional $g : (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})$ can be identified with the function $\check{g} : \mathbb{B}^\omega \to \mathbb{B}^\omega$, defined by $\check{g}(w) = (g(\hat{w}))^\vee$. Conversely, a function $h : \mathbb{B}^\omega \to \mathbb{B}^\omega$ can be identified with the functional $\hat{h} : (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})$ defined by $\hat{h}(f) = (h(\check{f}))^\wedge$.

We state without proof the straightforward, albeit tedious, observation that the two notions are equivalent. (The Theorem and its proof are unrelated to other parts of the present paper.)

▶ **Theorem 5.** *A partial function $h : \mathbb{B}^\omega \to \mathbb{B}^\omega$ is computable by an equational program iff the functional $\hat{h}$ is computable by some oracle Turing machine.*

Dually, a functional $g : (\mathbb{N} \to \mathbb{B}) \to (\mathbb{N} \to \mathbb{B})$ is computable by an oracle Turing machine iff the function $\check{g}$ is computable by an equational program.

3 A Canonicity Theorem: operational semantic is equivalent to Tarskian semantic

3.1 Data-correct expansions and the canonical structure

Let $\mathcal{D}$ be a data-system with $\mathcal{C}$ as constructor-set, and $\mathcal{S}$ a structure over a vocabulary that includes all identifiers in $\mathcal{C}$, but not the type-identifiers of $\mathcal{D}$. The *data-correct expansion*[6] of $\mathcal{S}$ is the expansion to the full vocabulary of $\mathcal{D}$, with the data-predicates $\mathbf{D}_{ij}$ interpreted as follows. (Recall that the interpretation of the constructors is already given in $\mathcal{S}$.)

[6] As usual, we say that a structure $\mathcal{S}$ is an expansion of a structure $\mathcal{Q}$ if $\mathcal{S}$ differs from $\mathcal{Q}$ only in interpreting additional vocabulary identifiers. E.g. $\mathbb{N}$ with addition and multiplication is an expansion of $\mathbb{N}$ with addition only.

1. If $\vec{\mathbf{D}}_i$ is inductive, then the sets $[\![\mathbf{D}_{ij}]\!]$ are obtained from $[\![\vec{\mathbf{D}}_1]\!]$ ... $[\![\vec{\mathbf{D}}_{i-1}]\!]$ by the data composition rules for $\vec{\mathbf{D}}_i$. That is, each inductive bundle is interpreted as the minimal subsets of the $R_{\mathcal{C}}$ closed under the bundle's composition rules.
2. Dually, if $\vec{\mathbf{D}}_i$ is coinductive, then $[\![\mathbf{D}_{ij}]\!]$ are the sets of finite and infinite terms obtained from $[\![\vec{\mathbf{D}}_1]\!]$ ... $[\![\vec{\mathbf{D}}_{i-1}]\!]$ by the decomposition rules for $\vec{\mathbf{D}}_i$. That is, each coinductive data-bundle is interpreted as the maximal vector of subsets of $R_{\mathcal{C}}$ for which the decomposition rules are applicable (i.e. every element is subject to a decomposition rule) and true.

The *canonical model* $\mathcal{A} \equiv \mathcal{A}_{\mathcal{D}} = [\![\mathcal{D}]\!]$ of a data-system $\mathcal{D}$ is the data-correct expansion of the replete structure $\mathcal{R}_{\mathcal{C}}$.

3.2 Typing statements

Suppose $(P, \mathbf{f})$ is a program (unary, say) over $\mathcal{C}$. The program computes a partial function $g : R_{\mathcal{C}} \rightharpoonup R_{\mathcal{C}}$.

▶ **Definition 6.** Given a data-system $\mathcal{D}$ over $\mathcal{C}$, with $\mathbf{D}$ and $\mathbf{E}$ among its data-predicates, we say that g is *of type* $\mathbf{D} \to \mathbf{E}$ if for every $a \in [\![\mathbf{D}]\!]_{\mathcal{A}}$ the function g is defined for input a, and $g(a) \in [\![\mathbf{E}]\!]_{\mathcal{A}}$. We also say in that case that P *is of type* $\mathbf{D} \to \mathbf{E}$. The definition for $(P, \mathbf{f})$ of arity > 1 is similar.

Note that each function, including the constructors, can have multiple types. Also, a program may compute a non-total mapping over $R_{\mathcal{C}}$, and still be of type $\mathbf{D} \to \mathbf{E}$, i.e. compute a total function from type $\mathbf{D}$ to type $\mathbf{E}$. To adequately capture the computational behavior of equational programs, multiple representations of divergence might be necessary; see [18] for examples and discussion.

The partiality of computable functions is commonly addressed either by allowing partial structures [16, 1, 23], or by considering semantic domains, with an object $\perp$ denoting divergence. The approach here is based instead on the "global" behavior of programs in all structures.

When a function $f : R_{\mathcal{C}} \to R_{\mathcal{C}}$ fails to be of a type $\mathbf{D} \to \mathbf{E}$ then the restriction of f to $\mathbf{D}$ is a partial function. That is, values $f(a) \in R_{\mathcal{C}} - [\![\mathbf{E}]\!]$ correspond to the divergence of the program for input $a \in R_{\mathcal{C}}$.

3.3 Canonicity for inductive data

Definition 1 provides the computational semantics of a program $(P, \mathbf{f})$. But as a set of equations a program can be construed simply as a first-order formula, namely the conjunction of the universal closure of those equations. As such, a program has its Tarskian semantic, referring to arbitrary structures for the vocabulary in hand, that is the constructors and the program-functions used in P. A model of P is then just a structure that satisfies each equation in P.

Herbrand proposed to define the computable functions (over $\mathbb{N}$) as those that are unique solutions of equational programs.[7] It is rather easy to show that every computable function is indeed the unique solution of a set of equations. But the converse fails, as illustrated by the following example.[8] Let $G[x] \equiv \exists y.\, G_0(x, y)$ be undecidable, with G_0 decidable. Clearly

[7] This proposal was made to Gödel in personal communication, and reported in [8]. A modified proposal, incorporating an operational-semantics ingredient, was made in [11].

[8] The first counter-example to Herbrand's proposal is probably due to Kalmar [12]. The example given here is a simplification of an example of Kreisel, quoted in [28].

there is a program for the function f defined by

$$f(x,v) = \begin{cases} 1 & \text{if} \quad \exists y < v.\ G_0(x,y) \\ 2 \cdot f(x, v+1) & \text{otherwise} \end{cases}$$

If, for a given x, $\exists y.\ G_0(x,y)$, then $\lambda v.f(x,v)$ has a unique solution, with $f(x,0) > 0$. Otherwise $f(x,v) = 0$ is the unique solution. So if f were computable, then G would be decidable.

In fact, Herbrand's definition yields precisely the hyper-arithmetical functions [28]. But Herbrand was not far off: he only needed to refer collectively to all data-correct structures:

▶ **Theorem 7.** (Canonicity Theorem for $\mathbb{N}$) [18] *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 data-correct model of P.*

3.4 Canonicity Theorem for Data Systems

We generalize Theorem 7 to all data-systems. Let $\mathcal{D}$ be a data-system for a constructor set $\mathcal{C}$.

▶ **Theorem 8.** *(*Canonicity Theorem for Data Systems) *An equational program $(P, \mathbf{f})$ over $\mathcal{C}$ computes a function $f : \mathbf{D} \to \mathbf{E}$ (using the operational semantics of equational logic) iff the formula $\mathbf{D}(x) \to \mathbf{E}(\mathbf{f}(x))$ is true (in the sense of Tarskian semantic of first-order formulas) in every data-correct model of P.*

We present the proof in the rest of the present subsection. Given a program $(P, \mathbf{f})$ over a data-system $\mathcal{D}$, we construct a canonical model $\mathcal{M}(P)$ to serve as a "test-structure" for the program. Let $\mathcal{C}'$ consist of the program-functions in P, and $\mathbf{T}(P)$ be $\mathcal{R}_{\mathcal{B}}$ for the vocabulary $\mathcal{B} = \mathcal{C} \cup \mathcal{C}'$. Thus the elements of $\mathbf{T}(P)$ are finite and infinite terms built using both the constructors and the program-functions used in P. (Using only terms with a finite number of program-functions along each branch would suffice, but this restriction, albeit natural, is immaterial here.) Let $\approx_P$ be the binary relation over $\mathbf{T}(P)$ that holds between two terms $\mathbf{t}, \mathbf{t}' \in T(P)$ iff $P \vdash^{=} \Pi\mathbf{t} = \Pi\mathbf{t}'$ for every deep destructor Π; that is, the pointwise equality of $\mathbf{t}$ and $\mathbf{t}'$ can be proved from P in equational logic. This is trivially an equivalence relation.

Now define $\mathcal{B}(P)$ to be the structure for the vocabulary $\mathcal{C} \cup \mathcal{C}'$ whose universe is the quotient $\mathbf{T}(P)/\approx_P$, and where each function-identifier is interpreted as symbolic application: a function-identifier $\mathbf{g}$, unary say, maps each equivalence class $[\mathbf{t}]_\approx$ to the equivalence class $[\mathbf{g}(\mathbf{t})]_\approx$; and similarly for arities > 1.

Let $\mathcal{M}(P)$ be the data-correct expansion of $\mathcal{B}(P)$. The following observation implies an alternative, more direct, definition of $\mathcal{M}(P)$.

▶ **Lemma 9.** *Each data-predicate $\mathbf{D}$ is interpreted in $\mathcal{M}(P)$ as* $\{[a]_\approx \mid a \in [\![\mathbf{D}]\!]_{\mathcal{A}}\}$.

Proof. The closure conditions defining the sets $[\![\mathbf{D}]\!]_{\mathcal{M}(P)}$ for data-predicates $\mathbf{D}$ of $\mathcal{D}$ are the same as for the canonical model $\mathcal{A} = \mathcal{R}_{\mathcal{C}}$. ◀

Theorem 8 now follows from the following Lemma.

▶ **Lemma 10.** *The following are equivalent.*
1. *The program $(P, \mathbf{f})$ computes a function $g : \mathbf{D} \to \mathbf{E}$.*
2. *$\mathcal{M}(P) \models \forall x\ \mathbf{D}(x) \to \mathbf{E}(\mathbf{f}(x))$.*
3. *$\mathcal{S} \models \forall x\ \mathbf{D}(x) \to \mathbf{E}(\mathbf{f}(x))$ for every data-correct structure $\mathcal{S}$.*

The equivalence above lifts to arities > 1.

Proof. (1) implies (3) since the equational computation of P over $[\![\mathcal{D}]\!]$ remains correct in every data-correct model of P.

(3) implies (2), since $\mathcal{M}(P)$ is data-correct by definition.

Finally, towards proving that (2) implies (1), assume (2). Let $g : R_{\mathcal{C}} \to R_{\mathcal{C}}$ be the function computed by $(P, \mathbf{f})$, unary say. Taking an input $a \in R_{\mathcal{C}}$ such that $a \in [\![\mathbf{D}]\!]_{\mathcal{A}}$, we have $[a]_{\approx} \in [\![\mathbf{D}]\!]_{\mathcal{M}(P)}$, by Lemma 9. This implies by (2) that $[\mathbf{f}(a)] \in [\![\mathbf{E}]\!]_{\mathcal{M}(P)}$. But by Lemma 9 again, this implies that $\mathbf{f}(a) \approx b$ for some $b \in [\![\mathbf{E}]\!]_{\mathcal{S}}$, establishing (1). ◀

4 Intrinsic theories

4.1 Intrinsic theories for inductive data

Intrinsic theories for inductive data-types were introduced in [18]. They support unobstructed reference to partial functions and to non-denoting terms, common in functional and equational programming. Each intrinsic theory is intended to be a framework for reasoning about the typing properties of programs, including their termination and fairness. In particular, declarative programs are considered as formal theories. This contrasts with two longstanding approaches to reasoning about programs and their termination, namely programs as modal operators [31, 25, 10], and programs (and their computation traces) as explicit mathematical objects [15, 16].

Let $\mathcal{D}$ be a data-system consisting of a single inductive bundle $\vec{\mathbf{D}}$. The *intrinsic theory for $\mathcal{D}$,* is a first order theory over the vocabulary of $\mathcal{D}$, whose axioms are

- The closure rules of $\mathcal{D}$.
- **Inductive delineation (data-elimination, Induction)**, which are the dual of the closure rules. Namely, if a vector $\vec{\varphi}[x]$ of first order formulas satisfies the composition rules for $\vec{\mathbf{D}}$, then it contains $\vec{\mathbf{D}}$:

$$\mathit{Comp}[\vec{\varphi}] \;\to\; \wedge_i \forall x\, \mathbf{D}_i(x) \to \varphi_i[x]$$

  where $\mathit{Comp}[\vec{\varphi}]$ is the conjunction of the composition rules for the bundle, with each $\mathbf{D}_i(\mathbf{t})$ replaced by $\varphi_i[\mathbf{t}]$.
- **Separation Axioms,** stating that every constructor is injective, and $\mathbf{c}(\vec{x}) \neq \mathbf{d}(\vec{y})$ for all distinct constructors $\mathbf{c}$ and $\mathbf{d}$. These imply that all data-terms are distinct. The Separation Axioms are superfluous for

Examples: $\mathbb{N}$, i.e. $\mathcal{A}(\mathtt{0}, \mathtt{s})$.

The Intrinsic theory for $\mathbb{N}$ has for vocabulary the constructors $\mathtt{0}$ and $\mathtt{s}$, and a unary relation identifier N. The axioms, given as natural deduction rules, are

- Data-introduction:

$$\frac{}{N(\mathtt{0})} \qquad \frac{N(x)}{N(\mathtt{s}x)}$$

- Data-elimination:

$$\frac{N(\mathbf{t}) \quad \varphi[\mathtt{0}] \quad \begin{array}{c}\{\varphi[z]\}\\ \cdots \\ \varphi[\mathtt{s}(z)]\end{array}}{\varphi(\mathbf{t})}$$

Identifying $\mathbb{W} = \{\mathtt{0}, \mathtt{1}\}^*$ with the free algebra generated from the nullary constructor ε and the unary $\mathtt{0}$ and $\mathtt{1}$, the intrinsic theory $\mathbf{IT}(\mathbb{W})$ has as vocabulary these constructors and a unary data-predicate W. The deductive rules are:

- Data-introduction:

$$\frac{}{W(\varepsilon)} \qquad \frac{W(\mathbf{t})}{W(\mathtt{0}(\mathbf{t}))} \qquad \frac{W(\mathbf{t})}{W(\mathtt{1}(\mathbf{t}))}$$

- Data-elimination:

$$\frac{W(\mathbf{t}) \quad \varphi[\varepsilon] \quad \begin{matrix}\{\varphi[z]\}\\ \cdots\\ \varphi[\mathtt{0}(z)]\end{matrix} \quad \begin{matrix}\{\varphi[z]\}\\ \cdots\\ \varphi[\mathtt{1}(z)]\end{matrix}}{\varphi(\mathbf{t})}$$

4.2 Provable typing in intrinsic theories

▶ **Definition 11.** A unary program $(P, \mathbf{f})$ is *provably of type* $\mathbf{D} \to \mathbf{E}$ in a theory $\mathbf{T}$ if $\mathbf{D}(x) \to \mathbf{E}(\mathbf{f}(x)$ is provable in $\mathbf{T}$ from the universal closure of the equations in P.

For example, consider the doubling function $\mathtt{dbl}$ over $\mathbb{N}$ defined by the program $\mathtt{dbl}(\mathtt{0}) = \mathtt{0}$, $\mathtt{dbl}(\mathtt{s}(x)) = \mathtt{s}(\mathtt{s}(\mathtt{dbl}(x)))$. The following is a proof of $\mathtt{N}(x) \to \mathtt{N}(\mathtt{dbl}(x))$, using induction on the predicate $\varphi[z] \equiv \mathtt{N}(\mathtt{dbl}(z))$. The double-bars indicate the omission of trivial steps.

$$\dfrac{\dfrac{\mathtt{N}(\mathtt{0})}{\mathtt{N}(\mathtt{dbl}(\mathtt{0}))} \qquad \dfrac{\dfrac{\forall x\ \mathtt{dbl}(\mathtt{s}(x)) = \mathtt{s}(\mathtt{s}(\mathtt{dbl}(x)))}{\mathtt{dbl}(\mathtt{s}(z)) = \mathtt{s}(\mathtt{s}(\mathtt{dbl}(z)))} \qquad \dfrac{\mathtt{N}(\mathtt{dbl}(z))}{\mathtt{N}(\mathtt{s}(\mathtt{s}(\mathtt{dbl}(z))))}}{\mathtt{N}(\mathtt{dbl}(\mathtt{s}(z)))}}{\mathtt{N}(\mathtt{dbl}(x))}$$

In fact, we have:

▶ **Theorem 12.** [18]. *The provable programs of the intrinsic theory* $\mathbf{IT}(\mathbb{N})$ *for the natural numbers are precisely the provably-recursive programs of Peano's Arithmetic.*

Note that Theorem 12 gives a characterization of the provable functions of PA without involving any particular choice of base functions (such as additional and multiplication).

4.3 Intrinsic theories for arbitrary data-systems

Let $\mathcal{D}$ be a data-system. The *intrinsic theory for* $\mathcal{D}$, denoted $\mathbf{IT}(\mathcal{D})$, is a first order theory over the vocabulary of $\mathcal{D}$, whose axioms are the inductive composition rule and coinductive decomposition rules of $\mathcal{D}$, as well as their duals:

- **Inductive delineation (data-elimination, Induction)**: If a vector $\vec{\varphi}[x]$ of first order formulas satisfies the composition rules for an inductive bundle $\vec{\mathbf{D}}$, then it contains $\vec{\mathbf{D}}$:

$$\mathit{Comp}[\vec{\varphi}] \ \to\ \wedge_i \forall x\ \mathbf{D}_i(x) \to \varphi_i[x]$$

  where $\mathit{Comp}[\vec{\varphi}]$ is the conjunction of the composition rules for the bundle, with each $\mathbf{D}_i(\mathbf{t})$ replaced by $\varphi_i[\mathbf{t}]$.
- **Coinductive delineation (data-introduction, Coinduction)**: If a vector $\vec{\varphi}[x]$ of first order formulas satisfies the decomposition rule for a coinductive bundle $\vec{\mathbf{D}}$, then it is contained in $\vec{\mathbf{D}}$:

$$\mathit{Dec}[\vec{\varphi}] \ \to\ \wedge_i \forall x\ \varphi_i[x] \to \mathbf{D}_i(x)$$

  where $\mathit{Dec}[\vec{\varphi}]$ is the conjunction of the decomposition rules for the bundle, with each $\mathbf{D}_i(\mathbf{t})$ replaced by $\varphi_i[\mathbf{t}]$.
- *Separation* axioms, stating that every constructor is injective, and $\mathbf{c}(\vec{x}) \neq \mathbf{d}(\vec{y})$ for all distinct constructors $\mathbf{c}$ and $\mathbf{d}$.

5 Proof theoretic strength

Our general intrinsic theories refer to infinite basic objects (coinductive data), in contrast to intrinsic theories for inductive data only, as well as traditional arithmetical theories. However, the deductive machinery itself does not imply the existence of any particular coinductive object, as would be the case, for example, in the presence of some form of the Axiom of Choice or of a comprehension principle. As a consequence, any intrinsic theory, merging inductive and coinductive constructions in any way, is interpretable in a formal theory which proof theoretically is no stronger than Peano Arithmetic.

Consider the formalism **PRA** of Primitive Recursive Arithmetic, with function identifiers for all primitive recursive functions, and their defining equations as axioms. In addition, we have the Separation Axioms for $\mathbb{N}$ (as above), and the schema of Induction for all formulas.[9] Let $\mathbf{PRA}^*$ be **PRA** augmented with function variables and quantifiers over them. The schema of Induction applies now to all formulas in the extended language, but otherwise there are no axioms stipulating the existence of additional functions. It is well known that **PRA** is interpretable in Peano's Arithmetic (where only addition and multiplication are given as functions with their defining equations).

▶ **Lemma 13.** *The theory* $\mathbf{PRA}^*$ *is conservative over* **PRA**. *That is, if a formula in the language of* **PRA** *is provable in* $\mathbf{PRA}^*$*, then it is provable already in* **PRA**.

The proof is a simplified version of the proof in [34, §2.4.8] that the hereditarily recursive operators form a model of arithmetic in all finite types. Here it suffices to observe that the function quantifiers in $\mathbf{PRA}^*$ can be interpreted as ranging over the computable (or even the PR) functions.

Lemma 13 implies, in particular, that $\mathbf{PRA}^*$ is not proof-theoretically stronger than **PA**.

▶ **Theorem 14.** (Arithmetic interpretability) *Every intrinsic theory is interpretable in* $\mathbf{PRA}^*$.

Proof. Each $t \in R_{\mathcal{C}}$ can be represented by a function $f_t : \mathbb{N} \to \mathbb{N}$, that maps addresses $a = \langle b_1 \cdots b_k \rangle \in \mathbb{N}$ to the code $\mathbf{c}^\sharp$ of the constructor $\mathbf{c}$ at address $\langle b_1 \dots b_k \rangle$ of t, if such a constructor exists, and to a reserved code (0 say) if t has no constructor at address a. For instance, if $t = \mathtt{p}(\mathtt{e}, \mathtt{0}(\mathtt{e}))$ then $f_t\langle\rangle = \mathtt{p}^\sharp$, $f_t\langle 0\rangle = \mathtt{e}^\sharp$, $f_t\langle 1\rangle = \mathtt{0}^\sharp$, $f_t\langle 1, 0\rangle = \mathtt{e}^\sharp$, and $f_t(a) = 0$ for every other address a.

Suppose $\mathcal{D}$ is a data-system over $R_{\mathcal{C}}$, with successive bundles $\vec{\mathbf{D}}_1$, ..., $\vec{\mathbf{D}}_k$. If $\vec{\mathbf{D}}_1 = \langle \mathbf{D}_{11} \dots \mathbf{D}_{1\ell} \rangle$ is inductive, then we can define Σ^0_1 formulas $D_1 \dots D_\ell$, with a single free unary-function variable f, such that $D_i[f]$ is true just in case f codes a tree $t \in R_{\mathcal{C}}$ which is in $\mathbf{D}_i$.

If $\vec{\mathbf{D}}_1$ is coinductive, then the same holds with the formulas D_i taken to be Π^0_1. Next, we can define formulas $\vec{D}_2$ in the second level of the arithmetical hierarchy, with a free unary-function variable f, which are true of f iff it codes some $t \in R_{\mathcal{C}}$ in the bundle $\vec{\mathbf{D}}_2$. Thus, the entire data-system can be interpreted in $\mathbf{PRA}^*$ by formulas of some level $\leq k$ in the arithmetical hierarchy.

We can then define, for each (first-order) formula φ of an intrinsic theory $\mathbf{T}$, a formula $\varphi^\star$ of $\mathbf{PRA}^*$, such that φ is true in the canonical model of the data-system iff $\varphi^\star$ is true in the standard model of **PRA**.

[9] See e.g. [33] for details and related discussions.

Next we show that if φ is provable in the intrinsic theory, then $\varphi^\star$ is provable in $\mathbf{PRA}^*$. Indeed, it is easy to observe that induction for inductive data can be proved, for the $\star$-interpreted formulas, by induction. More interestingly, coinduction for coinductive data is also provable, for the interpretable formulas, by induction. For example, consider coinduction for the binary ω-words, represented by the data-predicate $\mathtt{W}$:

$$\frac{\varphi[\mathbf{t}] \qquad \forall x\ \varphi[x] \to \exists y\ \varphi[y] \wedge (x = \mathtt{0}(y) \vee x = \mathtt{1}(y))}{\mathtt{W}(\mathbf{t})}$$

For the Π^0_1 formula W interpreting $\mathtt{W}$, we need to prove $W[\mathbf{t}^\star]$ from the formula

$$\forall f\ (\ \varphi^\star[f] \to \exists g\ \varphi^\star[g] \wedge (f = \langle \mathtt{0}\rangle * g\ \vee\ f = \langle \mathtt{1}\rangle * g)\ ) \tag{2}$$

(Here $\langle u\rangle * g$ is the function that maps the root to u and an address $0^{\ell+1}$ to $g(0^\ell)$.) But recall that $W[\mathbf{t}^\star]$ states for every address a that the function denoted by $\mathbf{t}^\star$, has at each address a a certain (trivial) local property. This can now be proved by induction on the height n of a, using (2). The induction basis needs only the value of the function $\mathbf{t}^\star$ at the root, which is given by (2). The induction's step is similar.

Note that although we refer here to the iterated tails of $\mathbf{t}^\star$, thus seemingly to infinitely many functions, any function h among these can be referred to indirectly via $\exists u\ h = u * \mathbf{t}^\star$. ◀

6 Applications and further developments

Intrinsic theories provide a minimalist framework for reasoning about data and computation. The benefits were already evident when dealing with inductive data only, including a characterization of the provable functions of Peano's Arithmetic without singling out any functions beyond the constructors, a particularly simple proof of Kreisel's Theorem that classical arithmetic is Π^0_2-conservative over intuitionistic arithmetic [18], and a particularly simple characterization of the primitive-recursive functions [19]. The latter application guided a dual characterization of the primitive corecursive functions in terms of intrinsic theories with positive coinduction [20].

Intrinsic theories are also related to type theories, via Curry-Howard morphisms, providing an attractive framework for extraction of computational contents from proofs, using functional interpretations and realizability methods. The natural extension of the framework to coinductive methods, described here, suggests new directions in extracting such methods for coinductive data as well.

Finally, intrinsic theories are naturally amenable to ramification, leading to a transparent Curry-Howard link with ramified recurrence [2, 17] as well as ramified corecurrence [26].

References

1 Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

2 Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the poly-time functions, 1992.

3 Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

4 Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56—68, 1940.

5 Haskell Curry. First properties of functionality in combinatory logic. *Tohoku Mathematical Journal*, 41:371–401, 1936.

6 H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, Berlin, 1995.

7 Ronald Fagin. Generalized first order spectra and polynomial time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS, 1974.

8 Kurt Gödel. On undecidable propositions of formal mathematical systems. In Martin Davis, editor, *The Undecidable*. Raven, New York, 1965. Lecture notes taken by Kleene and Rosser at the Institute for Advanced Study, 1934.

9 Yuri Gurevich. Logic and the challenge of computer science. In *trends in theoretical computer science*, pages 1–57. Computer Science Press, Rockville, MD, 1988.

10 David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, Cabridge, MA, 2000.

11 Jacque Herbrand. Sur la non-contradiction de l'arithmétique. *Journal für die reine und angewandte Mathematik*, 1932:1—–8, 1932. English translation in [36] 618–628.

12 László Kalmár. Über ein Problem betreffend die Definition des Begriffes des allgemeine-rekursiven Funktion. *Zeit. mathematische Logik u Grund. der Mathematik*, 1:93–96, 1955.

13 Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische annalen*, 112:727–742, 1936.

14 Stephen C. Kleene. Lambda definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

15 Stephen C. Kleene. *Introduction to Metamathematics*. Wolters-Noordhof, Groningen, 1952.

16 Stephen C. Kleene. *Formalized Recursive Functions and Formalized Realizability*, volume 89 of *Memoirs of the AMS*. American Mathematical Society, Providence, 1969.

17 Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, Perspectives in Computer Science, pages 320–343. Birkhauser-Boston, New York, 1994. www.cs.indiana.edu/∼leivant/papers.

18 Daniel Leivant. Intrinsic reasoning about functional programs I: First order theories. *Annals of Pure and Applied Logic*, 114:117–153, 2002.

19 Daniel Leivant. Intrinsic reasoning about functional programs II: Unipolar induction and primitive-recursion. *Theor. Comput. Sci.*, 318(1-2):181–196, 2004.

20 Daniel Leivant and Ramyaa Ramyaa. Implicit complexity for coinductive data: a characterization of corecurrence. In Jean-Yves Marion, editor, *DICE*, volume 75 of *EPTCS*, pages 1–14, 2011.

21 Yiannis N. Moschovakis. The formal language of recursion. *J. Symb. Log.*, 54(4):1216–1252, 1989.

22 Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-coalgebraic specification in CoCasl. *J. Log. Algebr. Program.*, 67(1-2):146–197, 2006.

23 Peter D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.

24 Peter Padawitz. Swinging types=functions+relations+transition systems. *Theor. Comput. Sci.*, 243(1-2):93–165, 2000.

25 Vaughan R. Pratt. Semantical considerations on floyd-hoare logic. In *FOCS*, pages 109–121. IEEE Computer Society, 1976.

26 Ramyaa Ramyaa and Daniel Leivant. Ramified corecurrence and logspace. *Electr. Notes Theor. Comput. Sci.*, 276:247–261, 2011.

27 Horst Reichel. A uniform model theory for the specification of data and process types. In Didier Bert, Christine Choppy, and Peter D. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 348–365. Springer, 1999.

28 H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

29 Jan Rothe, Hendrik Tews, and Bart Jacobs. The coalgebraic class specification language CCSL. *J. UCS*, 7(2):175–193, 2001.

30 Lutz Schröder. Bootstrapping inductive and coinductive types in HasCASL. *Logical Methods in Computer Science*, 4(4), 2008.

31 Krister Segerberg. A completeness theorem in the modal logic of programs (preliminary report). *Notices American matheamtical society*, 24:A–552, 1977.

32 Jonathan Seldin. Curry's anticipation of the types used in programming languages. In *Proceedings of the Annual Meeting of the Canadian Society for History and Philosophy of Mathematics*, pages 143–163, Toronto, 2002.

33 S. Simpson. *Subsystems of Second-Order Arithmetic*. Springer-Verlag, Berlin, 1999.

34 A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Volume 344 of LNM. Springer-Verlag, Berlin, 1973.

35 Alan M. Turing. Computability and lambda-definability. *Journal of Symbolic Logic*, 2:153–163, 1937.

36 J. van Heijenoort. *From Frege to Gödel, A Source Book in Mathematical Logic*. Harvard University Press, Cambridge, MA, 1967.