Logical Methods in Computer Science
Vol. 10(4:18)2014, pp. 1–23
www.lmcs-online.org

Submitted Dec. 23, 2013
Published Dec. 27, 2014

GLOBAL SEMANTIC TYPING FOR INDUCTIVE AND COINDUCTIVE COMPUTING

DANIEL LEIVANT

Indiana University Bloomington
e-mail address: leivant@indiana.edu

Abstract. Inductive and coinductive types are commonly construed as ontological (Church-style) types, with canonical semantical interpretation. When studying programs in the context of global ("uninterpreted") semantics, it is preferable to think of types as semantical properties (Curry-style). A purely logical framework for reasoning about semantic types is provided by intrinsic theories, introduced by the author in 2002, which fit tightly with syntactic, semantic, and proof theoretic fundamentals of formal logic, with potential applications in implicit computational complexity as well as extraction of programs from proofs.

Intrinsic theories have been considered so far for inductive data, and we presently extend that framework to data defined using both inductive and coinductive closures. Our first main result is a Canonicity Theorem, showing that the global definition of program typing, via the usual (Tarskian) semantics of first-order logic, agrees with their operational semantics in the intended ("canonical") model.

The paper's other main result is a proof theoretic calibration of intrinsic theories: every intrinsic theory is interpretable in (a conservative extension of) first-order arithmetic. This means that quantification over infinite data objects does not lead, on its own, to proof-theoretic strength beyond that of Peano Arithmetic.

1. Introduction

1.1. A motivation: termination of equational programs.

We refer to the well-known dichotomy between the *canonical* and *global* interpretations of proofs and programs, often referred to as "interpreted" and "uninterpreted," respectively. The former is exemplified by Peano's Arithmetic, whose canonical model is the standard structure of the natural numbers with basic operations, and by programming languages with primitive types for integers, strings, etc. Thus, the axioms of Peano's Arithmetic (PA) are intended to contribute to the delineation of a particular model, whereas the axioms of Group Theory are intended to describe a class of models, a task they perform successfully by definition.

The limitative properties of canonical axiomatization and computing, e.g. the high complexity of program termination in the canonical model, let alone the complexity of

2012 ACM CCS: [**Theory of computation**]: Models of computation; Logic; Semantics and reasoning.
Key words and phrases: Inductive and coinductive types, equational programs, intrinsic theories, global model theory.

DOI:10.2168/LMCS-10(4:18)2014
© D. Leivant
Creative Commons

semantic truth of first-order formulas of arithmetic, justify a reconsideration of canonically intended theories, such as PA, as global theories with unintended, "non-standard," models. Such non-standard models have "non-standard elements," but the machinery of Tarskian semantics makes no syntactic distinction between intended and non-standard elements, and consequently no explicit distinction between canonical and non-standard models.

A trivial remedy is to enrich the vocabulary with type-identifiers. Indeed, that is precisely Peano's original axiomatization of arithmetic [26]: his context is an abstract universe of objects and sets, and the natural numbers form a particular collection N within that broader universe. The type N is thus construed semantically, as a collection of pre-existing objects, which happen to satisfy certain properties. This is in perfect agreement with the brand of typing introduced by Haskell Curry [7, 37]: a function f has type $\tau \rightarrow \sigma$ 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.

Semantic types reflect a global perspective, in that they can be considered for any domain of discourse. In contrast, Church's approach [6] construes types as inherent properties of objects: a function is of type $\tau \rightarrow \sigma$ when its domain consists of the objects of τ, and its codomain of objects of type σ. That is, Church's types are related to the presence of a canonical model.[1]

The distinction between ontological and semantic typing can be made for arbitrary inductive data types D, such as the booleans, strings, finite trees, and lists of natural numbers. Each inductive data-type is contained in the term algebra generated by a set $\mathcal{C}$ of constructors, a syntactic representation that suggests a global semantics for such types. Namely, D is a "global predicate," that assigns to each structure $\mathcal{S}$ (for a vocabulary containing $\mathcal{C}$) the set of denotations of closed $\mathcal{C}$-terms. Such global semantics is a well-known organizing principle for descriptive and computational devices over a class of structures, such as all finite graphs [8, 10].

The global viewpoint of types is of particular interest with respect to *programs* over inductive data. Each such program P may be of type $D \rightarrow D$ in one structure and not in another; e.g. if P is non-total on $\mathbb{N}$, then it is of type $\mathtt{N} \rightarrow \mathtt{N}$ in the flat-domain structure with $\mathtt{N}$ interpreted as $\mathbb{N}_\perp$, but not when $\mathtt{N}$ is interpreted as $\mathbb{N}$.

In [19] we showed that a program P computes a total function in the canonical structure iff P has a unique solution, with respect to Tarskian semantics, in *every* reasonable model of P. See §4.4 below for background and discussion.

Within the global framework, it makes sense to consider formal theories for proving global typing properties of equational programs. We focus here on 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 [19], where they were dubbed *intrinsic theories.* Among other benefits, they support attractive proof-theoretic characterizations of major complexity classes, such as the provable functions of Peano Arithmetic and the primitive recursive functions [19, 20].

The rest of the paper is organized as follows. In §2 we define *data-systems,* i.e. collections of data-types obtained by both inductive and coinductive definitions. Starting with the syntactic framework, which generalizes term algebras to potentially-infinite *hyper-terms,*

[1]The difference between semantic and ontological typing disciplines is thus significant in ways that phrases such as "explicit" and "implicit" do not convey.

we give an operational semantics for equational programs over hyper-terms. Section 3 describes the equational programs we wish to focus on, including their semantics. Section 4 presents a proof of our first main result, a *Canonicity Theorem* 4.7 matching the Tarskian semantics of equational programs with their operational semantics. Section 5 describes intrinsic theories, a simple proof theoretic setting for reasoning about equational and typing properties of equational programs. Finally, §6 presents our other main theorem 6.5, stating that intrinsic theories, even in the presence of coinductive types, are interpretable in a conservative extension of Peano's Arithmetic, and are thus of the same proof theoretic strength as Peano's Arithmetic.[2]

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.* Given a constructor-vocabulary $\mathcal{C}$, a *hyper-term (over $\mathcal{C}$)* is an ordered tree of constructors, possibly infinite, where each node with constructor $\mathbf{c}$ of arity r has exactly r children. We write $H_{\mathcal{C}}$ for the set of hyper-terms over $\mathcal{C}$. For a structure $\mathcal{S}$, we write $|\mathcal{S}|$ for the universe of the structure.

The *replete $\mathcal{C}$-structure* is the structure $\mathcal{H}_{\mathcal{C}}$ with[3]

(1) $\mathcal{C}$ as vocabulary;
(2) $|\mathcal{H}_{\mathcal{C}}| = H_{\mathcal{C}}$; and
(3) a syntactic interpretation for each identifier $\mathbf{c} \in \mathcal{C}$: $[\![\mathbf{c}]\!](a_1 \dots a_r)$ is the tree with $\mathbf{c}$ at the root and $a_1 \dots a_r$ as immediate sub-trees.

2.2. **Inductive data systems.** An inductive type is defined by its generative closure rules. For example, the rules for $\mathbb{N}$ are $\mathtt{N}(\mathtt{0})$ and $\forall x\, \mathtt{N}(x) \to \mathtt{N}(\mathtt{s}(x))$ (we'll often omit the universal quantifier in the statements of such rules). Similarly, words in $\{0,1\}^*$, construed as terms generated from a nullary constructor $\mathtt{e}$ and unary $\mathtt{0}$ and $\mathtt{1}$, are generated by the three rules $\mathtt{W}(\mathtt{e})$, $\forall x\, \mathtt{W}(x) \to \mathtt{W}(\mathtt{0}(x))$, and $\mathtt{W}(x) \to \mathtt{W}(\mathtt{1}(x))$. If $\mathtt{G}$ names a type G, then the type T of binary trees with leaves in G is generated by the rules $\mathtt{G}(x) \to \mathtt{T}(x)$, and $\mathtt{T}(x) \wedge \mathtt{T}(y) \to \mathtt{T}(\mathtt{p}(x,y))$, where $\mathtt{p}$ is a binary constructor (for pairing).

Several types can be generated jointly (i.e. simultaneously), for example: the set of $\mathtt{01}$-strings with no adjacent $\mathtt{1}$'s is obtained 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 sequence $\vec{\mathbf{D}} = (\mathbf{D}_1 \dots \mathbf{D}_k)$ of unary relation-identifiers, dubbed *type identifiers*;
(2) A set of *construction rules,* each one of the form

$$\forall \vec{y}\, \mathbf{Q}_1(y_1) \wedge \cdots \wedge \mathbf{Q}_r(y_r) \;\to\; \mathbf{D}_i(\mathbf{c}(y_1 \cdots y_r)) \tag{2.1}$$

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

[2]The reader familiar with rich type systems, such as those of Coq, Agda, or Nuprl, will notice that Theorem 6.5 is stated for a type system without infinite-branching type-constructors, such as W-types.

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

These rules delineate the intended meaning of the inductive types $\vec{\mathbf{D}}$ from below, as $\mathbf{D}_i$ is built up by the construction rules.

Conjoining the composition rules, we obtain a single. The following variant, equivalent to that conjunction in constructive (intuitionistic) first-order logic, will be useful:

$$\psi_1 \vee \cdots \vee \psi_k \;\rightarrow\; \mathbf{D}_i(x) \tag{2.2}$$

where each ψ_i, with x a free variable, is of the form

$$\exists y_1 \ldots y_r \;\; x = \mathbf{c}(\vec{y}) \;\wedge\; \mathbf{Q}_1(y_1) \;\wedge \cdots \wedge \mathbf{Q}_r(y_r) \tag{2.3}$$

where $y_1 \ldots y_r$ are distinct variables. We call a formula of the form (2.3) a *constructor-statement (for x)*.

To focus on the essentials, we do not consider several important type constructions, such as parametric types, dependent types, sum types, polymorphism, and W-types.

2.3. **Coinductive deconstruction rules.** Inductive construction rules state sufficient reasons for asserting that a (finite) hyper-term has a given type, given the types of its immediate sub-terms. The intended semantics of an inductive type D is thus the smallest set of hyper-terms closed under those rules. Coinductive deconstruction rules state necessary conditions for a term to have a given type, by implying possible combinations for the types of its immediate sub-terms. The intended semantics is the largest set of hyper-terms satisfying those conditions.

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

$$\mathtt{W}^\omega(x) \rightarrow (\exists y\, \mathtt{W}^\omega(y) \wedge x = \mathtt{0}(y)) \vee (\exists y\, \mathtt{W}^\omega(y) \wedge x = \mathtt{1}(y)) \tag{2.4}$$

Note that this is not quite captured by the implications $\mathtt{W}^\omega(\mathtt{0}x) \rightarrow \mathtt{W}^\omega(x)$ and $\mathtt{W}^\omega(\mathtt{1}x) \rightarrow \mathtt{W}^\omega(x)$, since these do not guarantee that every element of $\mathtt{W}^\omega$ is of one of the two forms considered.

Moreover, using a destructor function in stating deconstruction rules fails to differentiate between cases of the argument's main constructor. For 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 deconstruction rules

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

and

$$\mathtt{E}(x) \;\rightarrow\; \exists y\, \mathtt{Z}(y) \wedge x = \mathtt{1}(y)$$

These rules cannot be captured using a destructor, since the latter does not differentiate between cases for the input's main constructor.

These observations motivate the following.

Definition 2.1. A *deconstruction 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 *deconstruction rule,* of the form

$$\mathbf{D}_i(x) \;\rightarrow\; \psi_1 \vee \cdots \vee \psi_k \tag{2.5}$$

where each ψ_i is a constructor-statement (as in (2.3) above).

2.4. **General data-systems.** We proceed to define data-systems, in which data-types may be defined by any sequential nesting of induction and coinduction. Descriptive and deductive tools for such definitions have been studied extensively, e.g. referring to typed lambda calculi, with operators μ for smallest fixpoint and ν for greatest fixpoint. The Common Algebraic Specification Language CASL was used as a unifying standard in the algebraic specification community, and extended to coalgebraic data [29, 32, 23, 34]. Several frameworks combining inductive and coinductive data, such as [25], strive to be comprehensive, including various syntactic distinctions and categories, in contrast to our minimalistic approach.

Definition 2.2. A *data-system* $\mathcal{D}$ over a constructor vocabulary $\mathcal{C}$ 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 a *type-bundle,* and designated as either *inductive* or *coinductive.*
(2) For each inductive bundle $\vec{\mathbf{D}}_i$, an inductive definition of $\vec{\mathbf{D}}_i$ from the types in $\vec{\mathbf{D}}_j$, $j < i$.
(3) For each coinductive bundle $\vec{\mathbf{D}}_i$ a *coinductive definition* of $\vec{\mathbf{D}}_i$ from $\vec{\mathbf{D}}_j$, $j < i$.

Definition 2.3. We say that a data-system $\vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_k$ is Σ_n (Π_n) if $\vec{\mathbf{D}}_k$ is inductive (respectively, coinductive), and the list of bundles alternates $n{-}1$ times between inductive and coinductive bundles. (This choice of notation will become evident in Theorem 6.3.) That is, a single bundle is Σ_1 (Π_1) if it is inductive (respectively, coinductive); if $\vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_k$ is Σ_n then $\vec{\mathbf{D}}_1, \dots, \vec{\mathbf{D}}_k, \vec{\mathbf{D}}_{k+1}$ is Σ_n if $\vec{\mathbf{D}}_{k+1}$ is inductive, and Π_{n+1} if $\vec{\mathbf{D}}_{k+1}$ is coinductive; if $\vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_k$ is Π_n then $\vec{\mathbf{D}}_1, \dots, \vec{\mathbf{D}}_k, \vec{\mathbf{D}}_{k+1}$ is Π_n if $\vec{\mathbf{D}}_{k+1}$ is coinductive, and Σ_{n+1} if $\vec{\mathbf{D}}_{k+1}$ is inductive.

A data system $\mathcal{D} = \vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_k$ has *rank* n if it is Σ_n or Π_n. A data-type $\mathbf{D}_{ij}$ of $\mathcal{D}$ has rank n (in $\mathcal{D}$) if the data-system $\vec{\mathbf{D}}_1 \dots \vec{\mathbf{D}}_i$ has rank n.

2.5. **Examples of data-systems.**

(1) Let $\mathcal{C}$ consist of the identifiers `0`, `1`, `e`, `s`, and `p`, of arities 0,0,0,1, and 2, respectively. Consider the following Σ_3 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 `B`'s and `N`'s starting with `B` or, respectively, `N`), and finally an inductive `L` for lists of such streams. The defining formulas are, in simplified form,
$$\mathtt{B}(\mathtt{0}) \quad \mathtt{B}(\mathtt{1})$$
$$\mathtt{N}(\mathtt{0}) \quad \forall y\, \mathtt{N}(y) \to \mathtt{N}(\mathtt{s}(y))$$
$$\mathtt{F}(x) \to \exists y, z\ (x = \mathtt{p}(y, z)) \ \wedge\ \mathtt{B}(y) \ \wedge\ \mathtt{S}(z)$$
$$\mathtt{S}(x) \to \exists y, z\ (x = \mathtt{p}(y, z)) \ \wedge\ \mathtt{N}(y) \ \wedge\ \mathtt{F}(z)$$
$$\mathtt{L}(\mathtt{e}) \quad \forall y, z\, \mathtt{F}(y) \wedge \mathtt{L}(z) \to \mathtt{L}(\mathtt{p}(y, z)) \quad \forall y, z\, \mathtt{S}(y) \wedge \mathtt{L}(z) \to \mathtt{L}(\mathtt{p}(y, z))$$
Note that constructors `p` and `0` are reused for different data-types. This is in agreement with our untyped, generic approach, where the data-objects are untyped.
(2) Let the constructors be `0`, `1`, `s`, `p`, and `d`, of arities 0,0,1,2 and 3 respectively. Consider the Π_2 data system $((\mathtt{N}), (\mathtt{T}), (\mathtt{D}))$, with inductive `N` (natural numbers), coinductive `T` (finite or infinite 2-3 trees with leaves in `N`), and coinductive `D` (infinite binary trees

with nodes decorated by elements of $\mathtt{T}$). The inductive definition of $\mathtt{N}$ is as above; the coinductive definitions of $\mathtt{T}$ and $\mathtt{D}$ are

$$\begin{array}{rcl}\mathtt{T}(x) & \rightarrow & \mathtt{N}(x) \\ & & \vee\, (\exists y_1, y_2\;\; x = \mathtt{p}(y_1, y_2) \;\wedge\; \mathtt{T}(y_1) \;\wedge\; \mathtt{T}(y_2)) \\ & & \vee\, (\exists y_1, y_2, y_3\;\; x = \mathtt{d}(y_1, y_2, y_3) \;\wedge\; \mathtt{T}(y_1) \;\wedge\; \mathtt{T}(y_2) \;\wedge\; \mathtt{T}(y_3))\end{array}$$

and

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

Note that we construe a "tree of trees" not as a higher-order object, but simply as a tree of constructors, suitably parsed.

3. Programs over data-systems

3.1. **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 $\mathbf{t}_1 \ldots \mathbf{t}_r$ are terms, then so is $\mathbf{g}\,\mathbf{t}_1 \cdots \mathbf{t}_r$. We use informally the parenthesized notation $\mathbf{g}(\mathbf{t}_1, \ldots, \mathbf{t}_r)$, when convenient.[4] 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.[5] We write $|\mathbf{t}|$ for the height of a term $\mathbf{t}$.

We adopt equational programs, in the style of Herbrand-Gödel, as computation model. See for example [16] for a classical exposition. Such programs are sometimes dubbed "computation rules" [3, 35]. There are easy inter-translations between equational programs and program-terms such as those of $\mathbf{FLR}_0$ [22]. We prefer however to focus on equational programs because they integrate easily into logical calculi, and are naturally construed as axioms. In fact, 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 $\mathbf{f}(\mathbf{t}_1, \ldots, \mathbf{t}_k) = \mathbf{q}$, where $\mathbf{f}$ is a program-function of arity $k \geq 0$, $\mathbf{t}_1 \ldots \mathbf{t}_k$ is a list of base-terms with no variable repeating, 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. Given a program P, we call the program-terms that use the function-identifiers occurring in P P*-terms.*

The requirement that program-equations have no repeating variable in the input is essential when the input may be infinite, for else the applicability of such an equation might depend on two inputs being identical, a condition which is not decidable.

Programs of arity 0 can be used to define objects. For example, the singleton program T consisting of the equation $\mathtt{f} = \mathtt{sss0}$ 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. We can similarly construct nullary programs defining hyper-terms, such as the program I consisting of the single equation $\mathtt{i} = \mathtt{s}(\mathtt{i})$. The infinite hyper-term $\mathtt{s}^\omega$ is the unique hyper-term solution of

[4] Note that if $\mathbf{g}$ is nullary, it is itself a term, whereas with formal parentheses we'd have $\mathbf{g}()$.

[5] Data-terms are often referred to as *values*, and base-terms as *patterns.*

this equation. But that uniqueness does not extend to arbitrary structures, of course. For example, we may have $\mathtt{s}$ interpreted as the identity function, and the equation above is modeled over the ordinals with $\mathtt{s}$ interpreted as $\lambda x.1+x$, and $\mathtt{i}$ as any infinite ordinal.

3.2. **Operational semantics of programs.** A program $(P, \mathbf{f})$ *computes* a partial-function $g : \bar{\mathcal{C}} \rightharpoonup \bar{\mathcal{C}}$ when $g(p) = q$ iff the equation $\mathbf{f}(p) = q$ is derivable from P in equational logic. However, replete structures have infinite terms, so the output of a program over $H_{\mathcal{C}}$ must be computed piecemeal from finite information about the input values.

To formally describe computation over infinite data, with a modicum of syntactic machinery, we posit that each program over $\mathcal{C}$ has defining equations for destructors and a discriminator. That is, if the given vocabulary's constructors are $\mathbf{c}_1 \ldots \mathbf{c}_k$, with m their maximal arity, then the program-functions include the unary identifiers $\pi_{i,m}$ $(i = 1..m)$ and δ (destructors and discriminator), and all programs contain, for each constructor $\mathbf{c}$ of arity r the equations

$$\begin{array}{rcll} \pi_{i,m}(\mathbf{c}(x_1,\ldots,x_r)) & = & x_i & (i = 1..r) \\ \pi_{i,m}(\mathbf{c}(x_1,\ldots,x_r)) & = & \mathbf{c}(x_1,\ldots,x_r) & (i = r{+}1..m) \\ \delta(\mathbf{c}_i(\vec{\mathbf{t}}), x_1, \ldots, x_k) & = & x_i & (i = 1..k) \end{array} \tag{3.1}$$

We call a repeated composition of destructors a *deep-destructor,* and construe it as an address in hyper-terms.

A *valuation* is a function η from a finite set of variables to $H_{\mathcal{C}}$. If $\vec{\mathbf{v}}$ is a list of r distinct variables, and $\vec{t}$ a list of r hyper-terms, then $[\vec{\mathbf{v}} \leftarrow \vec{t}]$ is the valuation η defined by $\eta(\mathbf{v}_i) = t_i$.

We posit the presence in $\mathcal{C}$ of at least one nullary constructor $\mathtt{o}$; indeed, adding a nullary constructor to $\mathcal{C}$ does not impact the rest of the discussion. For a constructor $\mathbf{c}$ we write $\mathbf{c}^o$ for the term $\mathbf{c}(\mathtt{o},\ldots,\mathtt{o})$. For a deep-destructor Π we define[6]

$$\Pi^o(x) = \delta(\Pi(x), \mathbf{c}_1^o, \ldots, \mathbf{c}_k^o)$$

That is, $\Pi^o(x)$ identifies the constructor of x at address Π.

Definition 3.1. We say that a set Γ of equations *locally infer* an equation $\mathbf{t} = \mathbf{q}$ between program-terms if, for each deep-destructor Π, the equation $\Pi^o(\mathbf{t}) = \Pi^o(\mathbf{q})$ is derivable in equational logic from Γ. We write then $\Gamma \vdash^\omega \mathbf{t} = \mathbf{q}$.

The *diagram* of a valuation η is the set Δ_η of equations of the form $\Pi^o(\mathtt{v}) = \mathbf{c}^o$ where $\mathtt{v}$ is in the domain of η, Π a deep-destructor, and $\mathbf{c}$ the main constructor of $\Pi(\eta(\mathtt{v}))$. That is, Δ_η conveys, node by node, the structure of the hyper-term $\eta(\mathtt{v})$.

An r-ary program $(P, \mathbf{f})$ *locally-computes* a partial-function $g : H_{\mathcal{C}}{}^r \rightharpoonup H_{\mathcal{C}}$ when, for every $\vec{t} \in H_{\mathcal{C}}{}^r$ and $q \in H_{\mathcal{C}}$, $g(t_1,\ldots,t_r) = q$ iff $P, \Delta_\eta \vdash^\omega \mathbf{f}(\mathtt{v}_1 \ldots \mathtt{v}_r) = \mathtt{u}$, where $\eta = [\vec{\mathtt{v}}, \mathtt{u} \leftarrow \vec{t}, q]$.

The notion of local-computability is motivated solely by the presence of infinite data. For finite hyper-terms local-computability is equivalent to computability, as we now show. For a data-term $\mathbf{t}$ of $\mathcal{C}$ let $\hat{\mathbf{t}}$ be corresponding hyper-term, i.e. the syntax-tree of $\mathbf{t}$.

Proposition 3.2. *Let* $\mathbf{t}$ *and* $\mathbf{q}$ *be data-terms.*

$$P,\, \Delta_{[\mathtt{u},\mathtt{v} \leftarrow \hat{\mathbf{t}}, \hat{\mathbf{q}}]} \vdash^\omega \mathbf{f}(\mathtt{v}) = \mathtt{u} \tag{3.2}$$

iff

$$P \vdash \mathbf{f}(\mathbf{t}) = \mathbf{q} \tag{3.3}$$

[6]Here again we stipulate that $\mathcal{C} = \{\mathbf{c}_1, \ldots \mathbf{c}_k\}$; also $r_i = \text{arity}(\mathbf{c}_i)$

By structural induction on data-terms we have, in equational logic, and using the defining equations for the destructors and discriminator (3.1),

$$\mathtt{u} = \mathbf{t},\ \mathtt{v} = \mathbf{q} \vdash \Delta_{[\mathrm{u},\mathrm{v} \leftarrow \hat{\mathbf{t}}, \hat{\mathbf{q}}]}$$

So (3.2) implies

$$P,\ \mathtt{u} = \mathbf{t}, \mathtt{v} = \mathbf{q} \vdash^{\omega} \mathbf{f}(\mathtt{v}) = \mathtt{u}$$

i.e.

$$P \vdash^{\omega} \mathbf{f}(\mathbf{t}) = \mathbf{q}$$

By induction on $\mathbf{q}$, and using again (3.1), this implies (3.3).

For the converse, assume (3.3). Using induction on the length of equational derivations, for all terms $\mathbf{r}, \mathbf{s}$, if $P \vdash \{\mathbf{t}, \mathbf{q}/\mathtt{u}, \mathtt{v}\}\,(\mathbf{r} = \mathbf{s})$ then

$$P,\ \Delta_{[\mathrm{u},\mathrm{v} \leftarrow \hat{\mathbf{t}}, \hat{\mathbf{q}}]} \vdash^{\omega} \mathbf{r} = \mathbf{s}$$

In particular, we conclude (3.2).

3.3. **Equational vs. Turing computation.** The equivalence of equational programs over $\mathbb{N}$ with the μ-recursive functions was implicit already in [9], and explicit in [14]. Their equivalence with λ-definability [5, 15] and hence with Turing computability [40] followed readily. 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 nodes 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 $\hat{h}(f) = (h(\check{f}))^{\wedge}$.

It is easy (albeit tedious) to see that 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.

4. Matching Tarskian semantics and operational semantics

4.1. **$\mathcal{D}$-correct structures.** We have focused so far on the canonical setup for data-systems $\mathcal{D}$, with hyper-terms as objects. We now consider arbitrary structures. We call a structure $\mathcal{S}$ a $\mathcal{D}$-*structure* if its vocabulary (i.e. signature, symbol set) contains the constructor- and type-identifiers of $\mathcal{D}$. In a $\mathcal{D}$-structure $\mathcal{S}$ we may have a finite or infinite regression of constructor-eliminations, regardless of the nature of the structure elements. For example, if $\mathtt{f}$ is a unary constructor, and $g = \mathtt{f}_{\mathcal{S}}$ is its interpretation in $\mathcal{S}$, we might have an element $v_0 \in |\mathcal{S}|$ for which there is a $v_1 \in |\mathcal{S}|$ with $v_0 = g(v_1)$, and more generally elements $v_i \in |\mathcal{S}|$ $(i = 0, 1, \ldots)$ where $v_i = g(v_{i+1})$. In general g need not be injective, and so v_{i+1} need not be uniquely determined by v_i.

We say that a $\mathcal{D}$-structure $\mathcal{S}$ is $\mathcal{D}$-*correct* (or just *correct* when in no danger of confusion) if

(1) $\mathcal{S}$ is *separated for* $\mathcal{C}$, that is, the interpretations in $\mathcal{S}$ of the constructors are all injective and have pairwise-disjoint codomains.[7] Note that $\mathcal{H}_\mathcal{C}$ satisfies this property.
(2) If $\vec{\mathbf{D}}_i = \langle \mathbf{D}_{i1} \ldots \mathbf{D}_{im} \rangle$ is inductive, then $\langle [\![\mathbf{D}_{i1}]\!], \ldots, [\![\mathbf{D}_{im}]\!] \rangle$ is the minimal m-tuple of subsets of $|\mathcal{S}|$ closed under the construction rules for $\vec{\mathbf{D}}_i$, given the sets $[\![\vec{\mathbf{D}}_1]\!] \ \ldots \ [\![\vec{\mathbf{D}}_{i-1}]\!]$
(3) Dually, if $\vec{\mathbf{D}}_i$ is coinductive, then $[\![\vec{\mathbf{D}}_i]\!]$ is the largest vector of subsets of $H_\mathcal{C}$ closed under the deconstruction rules for $\vec{\mathbf{D}}_i$, given the sets $[\![\vec{\mathbf{D}}_1]\!] \ \ldots \ [\![\vec{\mathbf{D}}_{i-1}]\!]$.

The *canonical model* $\mathcal{A} \equiv \mathcal{A}_\mathcal{D} \equiv [\![\mathcal{D}]\!]$ of a data-system $\mathcal{D}$ is the $\mathcal{D}$-correct expansion of the replete structure $\mathcal{H}_\mathcal{C}$. Note that inductive and coinductive types are given their canonical interpretation in every $\mathcal{D}$-correct structures, but such structures may have elements that are outside all types. Indeed, that possibility is the motivation of intrinsic theories in the first place: one deals with "anomalies" of computation (divergence when an inductive output is expected, non-productiveness when a coinductive output is expected) not by partiality, but by allowing output which is not typed. A single element $\perp$ denoting divergence does not suffice (see the proof in [19] of Theorem 4.5 below).

4.2. **Decomposition in data-correct structures.** Let $\mathcal{S}$ be a $\mathcal{D}$-structure, and consider an element a of $|\mathcal{S}|$. A *$\mathcal{C}$-decomposition of a* is a finitely-branching tree T of elements of $|\mathcal{S}| \times \mathcal{C}$ such that

(1) The root of T is of the form $\langle a, \mathbf{c} \rangle$ with $\mathbf{c} \in \mathcal{C}$;
(2) if $\langle b_i, \mathbf{c}_i \rangle$ $(i = 1..r)$ are the children in T of a node $\langle b, \mathbf{c} \rangle$ of T, then $b = \mathbf{c}_\mathcal{S}(b_1, .., b_r)$.

If a has a $\mathcal{C}$-decomposition, we say that it is *$\mathcal{C}$-decomposable.* Put differently, a is $\mathcal{C}$-decomposable iff it is in the range of a partial mapping $\varphi : \ H_\mathcal{C} \rightharpoonup |\mathcal{S}|$ that satisfies $\varphi(\mathbf{c}(t_1 \ldots t_r)) = \mathbf{c}_\mathcal{S}(\varphi(t_1) \ldots \varphi(t_r))$.

Obviously, an element $a \in |\mathcal{S}|$ may have multiple $\mathcal{C}$-decompositions, and even uncountably many: it suffices to take the structure with two elements a, b and two constant functions $\lambda x.a$ and $\lambda x.b$.

Recall that a $\mathcal{D}$-structure $\mathcal{S}$ is separated if the interpretations in $\mathcal{S}$ of the constructors $\mathbf{c} \in \mathcal{C}$ are injective and with disjoint codomains.

Proposition 4.1. *If $\mathcal{S}$ is a separated structure for $\mathcal{C}$, then each element $a \in |\mathcal{S}|$ has at most one decomposition.*

Proof. Let t, t' be decompositions of $a \in |\mathcal{S}|$. We prove by induction on n that if $\langle b, \mathbf{c} \rangle$ is at address α of t of height n, then it is also at address α of t'. The induction's basis and step follow outright from the assumption that $\mathcal{S}$ is separated. □

If t is a $\mathcal{C}$-decomposition of a, let $\check{t}$ be the hyper-term obtained from t by replacing each node $\langle b, \mathbf{c} \rangle$ by $\mathbf{c}$. We call $\check{t}$ a *constructor-decomposition* (for short, a *decomposition*) of a. From the proof of proposition 4.1 it follows that an element a of a separated structure has at most one decomposition, which we denote (when it exists) by $\check{a}$.

Proposition 4.2. *Suppose $\mathcal{S}$ is a $\mathcal{D}$-correct structure. If $a \in |\mathcal{S}|$ has type $\mathbf{D}$ in $\mathcal{S}$, then it has a decomposition, which has type $\mathbf{D}$ in $\mathcal{A}$.*

Conversely, if $t \in H_\mathcal{C}$ has type $\mathbf{D}$ in $\mathcal{A}$, then every $a \in |\mathcal{S}|$ which has t as decomposition, is of type $\mathbf{D}$ in $\mathcal{S}$.

[7] For $\mathbb{N}$ these are Peano's Third and Fourth Axioms.

Proof. We prove the Proposition by cumulative induction on the rank of $\mathbf{D}$ in $\mathcal{D}$. Suppose the statement holds for types of rank $< n$. For each type $\mathbf{D}$ of $\mathcal{D}$ define

$$A(\mathbf{D}) = \{a \in |\mathcal{S}| \mid a \text{ has a decomposition, which is in } \mathbf{D}_{\mathcal{A}} \}$$

and

$$S(\mathbf{D}) = \{t \in H_{\mathcal{C}} \mid t \text{ is the decomposition of some } a \in \mathbf{D}_{\mathcal{S}} \}$$

Suppose $\mathbf{D}$ is in an inductive bundle $\vec{\mathbf{D}}_i$ of $\mathcal{D}$. The sequence of sets $\langle A(\mathbf{D}_{ij})\rangle_j$ satisfies the inductive closure condition of $\vec{\mathbf{D}}_i$. To see this, consider a rule of $\mathcal{D}$ for $\vec{\mathbf{D}}_i$, say (w.l.o.g.)

$$\mathbf{D}(y_1) \wedge \mathbf{D}'(y_2) \wedge \mathbf{E}(y_3) \rightarrow \mathbf{D}(\mathbf{c}(y_1, y_2, y_3))$$

where $\mathbf{D}'$ is another type in $\vec{\mathbf{D}}_i$ and $\mathbf{E}$ is a type of rank $< n$. We show that

$$(y_1 \in A(\mathbf{D})) \ \wedge \ (y_2 \in A(\mathbf{D}')) \ \wedge \ (y_3 \in \mathbf{E}_{\mathcal{S}}) \ \rightarrow \ \mathbf{c}(y_1, y_2, y_3) \in A(\mathbf{D})$$

The first two premises mean that y_1 and y_2 have decompositions $\check{y}_1 \in \mathbf{D}_{\mathcal{A}}$ and $\check{y}_2 \in \mathbf{D}'_{\mathcal{A}}$, and the third premise implies that $\check{y}_3 \in \mathbf{E}_{\mathcal{S}}$ by IH, since $\mathbf{E}$ is of rank $< n$. So the hyper-term $\mathbf{c}(\check{y}_1, \check{y}_2, \check{y}_3)$ is in $\mathbf{D}_{\mathcal{A}}$, since $\mathcal{A}$ is $\mathcal{D}$-correct. That hyper-term is the decomposition of $\mathbf{c}(y_1, y_2, y_3)$, proving that the latter is in $A(\mathbf{D})$.

Since $\langle(\mathbf{D}_{ij})_{\mathcal{S}}\rangle_j$ is the smallest fixpoint of those conditions (given that $\mathcal{S}$ is $\mathcal{D}$-correct), it follows that $\mathbf{D}_{\mathcal{S}} \subseteq A(\mathbf{D})$, i.e. every element of $|\mathcal{S}|$ of type $\mathbf{D}$ in $\mathcal{S}$ has a decomposition, which furthermore is of type $\mathbf{D}$ in $\mathcal{A}$.

For the converse, we observe that the sequence of sets $\langle S(\mathbf{D}_{i,j})\rangle_j$ is closed under the inductive closure conditions of the bundle $\vec{\mathbf{D}}_i$. To see this, consider again a rule

$$\mathbf{D}(y_1) \wedge \mathbf{D}'(y_2) \wedge \mathbf{E}(y_3) \rightarrow \mathbf{D}(\mathbf{c}(y_1, y_2, y_3)$$

as above. Assume the premise of

$$(y_1 \in S(\mathbf{D})) \ \wedge \ (y_2 \in S(\mathbf{D}')) \ \wedge \ (y_3 \in \mathbf{E}_{\mathcal{A}}) \ \rightarrow \ \mathbf{c}(y_1, y_2, y_3) \in S(\mathbf{D})$$

The first two conjuncts mean that y_1 and y_2 are decompositions of some $a_1 \in \mathbf{D}_{\mathcal{S}}$ and $a_2 \in \mathbf{D}'_{\mathcal{S}}$, and the third implies, by IH, that y_3 is the decomposition of some $a_3 \in \mathbf{E}_{\mathcal{S}}$. The hyper-term $\mathbf{c}(y_1, y_2, y_3)$ is the decomposition of $\mathbf{c}_{\mathcal{S}}(a_1, a_2, a_3)$, which is in $\mathbf{D}_{\mathcal{S}}$, since $\mathcal{S}$ is $\mathcal{D}$-correct. This concludes the case where $\mathbf{D}$ is an inductive type.

Suppose now that $\mathbf{D}$ is coinductive. Then $\langle A(\mathbf{D}_{ij})\rangle_j$ satisfies the coinductive closure condition of the bundle $\mathbf{D}_i$. To see this, consider the rule of $\mathcal{D}$ for $\mathbf{D}$, say

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

where each ψ_i is a constructor-statement. Assume (w.l.o.g.) that $k = 1$ and

$$\psi_1 \qquad \equiv \qquad \exists y_1, y_2, y_3 \; x = \mathbf{c}(y_1, y_2, y_3) \ \wedge \mathbf{D}(y_1) \wedge \mathbf{D}'(y_2) \wedge \mathbf{E}(y_3) \tag{4.1}$$

where $\mathbf{D}'$ and $\mathbf{E}$ are as above. We show that

$$a \in A(\mathbf{D}) \ \ \rightarrow \ \ (\ \exists y_1 \in A(\mathbf{D}) \ \exists y_2 \in A(\mathbf{D}') \ \exists y_3 \in \mathbf{E}_{\mathcal{S}} \quad a = \mathbf{c}_{\mathcal{S}}(y_1, y_2, y_3)\ )$$

An element $a \in A(\mathbf{D})$ has $\check{a} \in \mathbf{D}_{\mathcal{A}}$. Since $\mathcal{A}$ is $\mathcal{D}$-correct, $\check{a}$ must be $\mathbf{c}(t_1, t_2, t_3)$ for some $t_1 \in \mathbf{D}_{\mathcal{A}}$, $t_2 \in \mathbf{D}'_{\mathcal{A}}$, and $t_3 \in \mathbf{E}_{\mathcal{A}}$. Since $\check{a}$ is the decomposition of a, this means that $a = \mathbf{c}_{\mathcal{S}}(b_1, b_2, b_3)$, where $t_i = \check{b}_i$. So $b_1 \in A(\mathbf{D})$, $b_2 \in A(\mathbf{D}')$, by the definition of the function A, and $b_3 \in \mathbf{E}_{\mathcal{S}}$ by IH, since $\mathbf{E}$ is of rank $< n$.

Since $\mathcal{S}$ is $\mathcal{D}$-correct, $\mathbf{D}_{\mathcal{S}}$ is the greatest set closed under the closure conditions for the bundle $\vec{\mathbf{D}}$; it therefore has $A(\mathbf{D})$ as a subset. That is, every element a of $\mathcal{S}$ whose decomposition is of type $\mathbf{D}$ in $\mathcal{A}$, is of type $\mathbf{D}$ in $\mathcal{S}$.

For the converse, we similarly prove that $\langle S(\mathbf{D}_{i,j})\rangle_j$ is closed under the coinductive closure conditions of the bundle $\vec{\mathbf{D}}_i$. Suppose again that the coinductive rule for $\mathbf{D}$ is (4.1) above. We show that for every hyper-term t

$$t \in S(\mathbf{D}) \quad \to \quad (\exists y_1 \in S(\mathbf{D})\ \exists y_2 \in S(\mathbf{D}')\ \exists y_3 \in \mathbf{E}_\mathcal{S}\ a = \mathbf{c}_\mathcal{S}(y_1, y_2, y_3))$$

Suppose $t \in S(\mathbf{D})$, i.e. t is the deconstruction of some $a \in \mathbf{D}_\mathcal{S}$. Since $\mathcal{S}$ is $\mathcal{D}$-correct, a must be $\mathbf{c}(b_1, b_2, b_3)$ for some $b_1 \in \mathbf{D}_\mathcal{S}$, $b_2 \in \mathbf{D}'_\mathcal{S}$, and $b_3 \in \mathbf{E}_\mathcal{S}$. So t must be of the form $\mathbf{c}(t_1, t_2, t_3)$ where $t_1 \in S(\mathbf{D})$, $t_2 \in S(\mathbf{D}')$, by definition of $S(\cdots)$, and $t_3 \in \mathbf{E}_\mathcal{S}$ (by IH).

Since $\mathbf{D}_\mathcal{A}$ is the greatest subset of $H_\mathcal{C}$ closed under the rule for $\mathbf{D}$, is follows that it has $S(\mathbf{D})$ as a subset. That is, if a hyper-term t is the decomposition of an element of $\mathbf{D}_\mathcal{S}$, then t is of type $\mathbf{D}$ in $\mathcal{A}$. □

Corollary 4.3. *For any two $\mathcal{D}$-correct structures $\mathcal{S}$ and $\mathcal{Q}$, if $a \in |\mathcal{S}|$ and $b \in |\mathcal{Q}|$ have the same decomposition, then they have the same types in $\mathcal{S}$ and $\mathcal{Q}$.*

4.3. **Typing statements.**

Definition 4.4. Given a data-system $\mathcal{D}$ over $\mathcal{C}$, with $\mathbf{D}_1, \ldots, \mathbf{D}_r$ and $\mathbf{E}$ among its type-identifiers, we say that a partial function $g : H_\mathcal{C}{}^r \rightharpoonup H_\mathcal{C}$ is *of type* $(\times_{j\in J}\mathbf{D}_j) \to \mathbf{E}$ if $a_j \in (\mathbf{D}_j)_\mathcal{A}$ $(j \in J)$ jointly imply that $g(\vec{a})$ is definable and in $\mathbf{E}_\mathcal{A}$.

If $(P, \mathbf{f})$ is a program that computes the partial-function g above, we also say that P *is of type* $(\times_i \mathbf{D}_i) \to \mathbf{E}$.

Note that each function, including the constructors, can have multiple types. Also, a program may compute a non-total mapping over $H_\mathcal{C}$, and still be of type $\mathbf{D} \to \mathbf{E}$, i.e. compute a total function from type $\mathbf{D}$ to type $\mathbf{E}$.

When a (total) function $f : H_\mathcal{C} \to H_\mathcal{C}$ fails to be of a type $\mathbf{D} \to \mathbf{E}$ there must be some $d \in [\![\mathbf{D}]\!]$ for which $f(d) \notin [\![\mathbf{E}]\!]$. Thus the value $f(d)$ can represent divergence with respect to computation over $[\![\mathbf{D}]\!]$, as for example when $[\![\mathbf{D}]\!] = \mathbb{N}$ and $[\![\mathbf{E}]\!] = \mathbb{N}_\perp$ with $f(d) = \perp$. However, to adequately capture the computational behavior of equational programs, multiple representations of divergence might be necessary; see [19] for examples and discussion.

The partiality of computable functions is commonly addressed either by allowing partial structures [17, 1, 24], 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.

4.4. **Canonicity for inductive data.** Definition 3.1 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 semantics, 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 a (total) function g as *computable* just in case there is a program for which g is the unique solution.[8] It is rather easy to show that every computable

[8]This proposal was made to Gödel in personal communication, and reported in [9]. A modified proposal, incorporating an operational-semantics ingredient, was made in [12].

function is indeed the unique solution of a program. But the converse fails. In fact, Herbrand's definition yields precisely the hyper-arithmetical functions [30].[9] But Herbrand's ingenious idea to relate computability of a program to the unicity of its solution is still in force, provided one refers collectively to all $\mathcal{D}$-correct structures:[10]

Theorem 4.5. (Canonicity Theorem for $\mathbb{N}$) [19] *An equational program $(P, \mathbf{f})$ over $\mathbb{N}$ computes a total function iff the formula $\mathtt{N}(x) \rightarrow \mathtt{N}(\mathbf{f}(x))$ is true in every $\mathbb{N}$-correct model of P.*

4.5. **Canonicity for Data Systems.** We generalize Theorem 4.5 to all data-systems. Given a (unary) program $(P, \mathbf{f})$ over a data-system $\mathcal{D}$, and a valuation η, we construct a canonical model $\mathcal{M}(P, \eta)$ to serve as "test-structure" for the program P and the valuation η as input.

We define the equivalence relation $\approx_{P,\eta}$ over hyper-terms to hold between t and q iff Δ_η, P locally infer $t = q$, in the sense of Definition 3.1. When safe, we write $\approx$ for $\approx_{P,\eta}$.

Let $\mathcal{Q}(P, \eta)$ be the structure whose universe is the quotient $H_\mathcal{C}/\approx$, and where each function-identifier (constructor or program-function) is interpreted as symbolic application: for an r-ary identifier $\mathbf{f}$, $\mathbf{f}_\mathcal{Q}$ maps equivalence classes $[\mathbf{t}_i]_\approx$ to $[\mathbf{f}(\vec{\mathbf{t}})]_\approx$. This symbolic interpretation of the constructors guarantees that the structure is separated for $\mathcal{C}$. Let now $\mathcal{M}(P, \eta)$ be the $\mathcal{D}$-correct expansion of $\mathcal{Q}(P, \eta)$, i.e. the expansion of $\mathcal{Q}(P, \eta)$ to the full vocabulary of $\mathcal{D}$, with type-identifiers, where inductive types are interpreted as the minimal subsets of $H_\mathcal{C}$ closed under their closure conditions, and the coinductive types as the maximal subsets closed under their closure conditions.

Lemma 4.6. *$\mathcal{M}(P, \eta)$ is a model of P.*

Proof. If $\mathbf{f}(\vec{\mathbf{t}}) = \mathbf{q}$ is an equation in P, then $\mathbf{f}(\vec{\mathbf{t}}) \approx \mathbf{q}$ is immediate from the definition of $\approx_{P,\eta}$. Thus

$$[\mathbf{f}(\vec{\mathbf{t}})]_\approx = [\mathbf{q}]_\approx$$

Also, by structural induction on terms, one easily proves that

$$[\![\mathbf{t}]\!]_{\mathcal{M}(P,\eta)} = [\mathbf{t}]_\approx$$

for each term $\mathbf{t}$, since function-identifiers are interpreted in $\mathcal{M}(P, \eta)$ symbolically.

We conclude

$$[\![\mathbf{f}(\vec{\mathbf{t}})]\!]_{\mathcal{M}(P,\eta)} = [\![\mathbf{q}]\!]_{\mathcal{M}(P,\eta)}$$

□

Theorem 4.7. (Canonicity Theorem for Data Systems) *Let $\mathcal{D}$ be a data-system over $\mathcal{C}$, and $\mathbf{D}$, $\mathbf{E}$ two type-identifiers of $\mathcal{D}$. Let $(P, \mathbf{f})$ be an equational program over $\mathcal{C}$ computing a partial-function $g : H_\mathcal{C} \rightharpoonup H_\mathcal{C}$.*

The following are equivalent:

(1) $g : \mathbf{D}_\mathcal{A} \to \mathbf{E}_\mathcal{A}$
(2) $\mathbf{D}(x) \to \mathbf{E}(\mathbf{f}(x))$ *is true in every $\mathcal{D}$-correct model of P.*

The equivalence above generalizes to arities $\neq 1$.

[9] The first counter-example to Herbrand's proposal is probably due to Kalmar [13]. A simple example of a program whose unique solution is not computable was given by Kreisel, quoted in [30].

[10] Of course, the important correction of Herbrand's equational computing is Gödel's radical change of perspective, from Tarskian semantics to operational (rewrite rules).

Proof. We show that (1) and (2) are also equivalent to
(3) For all valuations η, $\mathcal{M}(P,\eta) \models \mathbf{D}(x) \ \rightarrow \ \mathbf{E}(\mathbf{f}(x))$.

(1) implies (2): Assume (1), and let $\mathcal{S}$ be a $\mathcal{D}$-correct model of P. Consider an element $a \in \mathbf{D}_{\mathcal{S}}$. By Proposition 4.2 a has a decomposition $\breve{a}$. Moreover, since $\mathcal{S}$ is $\mathcal{D}$-correct, the closure conditions justifying $a \in \mathbf{D}_{\mathcal{S}}$ also justify $\breve{a} \in \mathbf{D}_{\mathcal{A}}$. By (1), this implies that $g(\breve{a}) \in \mathbf{E}_{\mathcal{A}}$.

Since g is computed by P we have, for each deep-destructor Π, that an equation $\Pi^o(\mathbf{f}(\mathtt{v})) = \mathbf{c}^o$ is derivable in equational logic from P and Δ_η, where $\mathbf{c}$ is the main constructor of $\Pi(g(\breve{a}))$. Since $\breve{a}$ is the decomposition of a, all equations Δ_η are true in $\mathcal{S}$. But $\mathcal{S}$ is known to be a model of P, so $\Pi^o(\mathbf{f}(\mathtt{v})) = \mathbf{c}^o$ is true in $\mathcal{S}$ with $\mathtt{v}$ bound to a. This being the case for every deep-destructor Π, it follows that $\mathbf{f}_{\mathcal{S}}(a)$ has the same decomposition as $g(\breve{a})$. But $g(\breve{a}) \in \mathbf{E}_{\mathcal{A}}$ and $\mathcal{S}$ is $\mathcal{D}$-correct, so $\mathbf{f}_{\mathcal{S}}(a) \in \mathbf{E}_{\mathcal{S}}$, proving (2).

(2) implies (3): $\mathcal{M}(P,\eta)$ is $\mathcal{D}$-correct by definition. It is a model of P by Lemma 4.6. So (3) is a special case of (2).

(3) implies (1): Assume (3). Consider input $a \in \mathbf{D}_{\mathcal{A}}$, and let $\eta(\mathtt{v}) = a$. The class $[\mathtt{v}]_{\approx}$ has then the same decomposition as a, and since $\mathcal{M}(P,\eta)$ is $\mathcal{D}$-correct, it must have type $\mathbf{D}$ in $\mathcal{M}(P,\eta)$, by Proposition 4.2. By (3) it follows that

$$\mathbf{f}_{\mathcal{M}(P,\eta)}([\mathtt{v}]_{\approx}) \in \ \mathbf{E}_{\mathcal{M}(P,\eta)}$$

But

$$\mathbf{f}_{\mathcal{M}(P,\eta)}([\mathtt{v}]_{\approx}) = [\mathbf{f}(\mathtt{v})]_{\approx}$$

by definition of $\mathcal{M}(P,\eta)$. Since g is computed by $(P,\mathbf{f})$, we have $\Pi^o(\mathbf{f}(\mathtt{v})) = \Pi^o(g(a))$ for all deep-destructors Π. So $g(a)$ has the same decomposition as $[\mathbf{f}(\mathtt{v})]_{\approx}$, and therefore is in $\mathbf{E}_{\mathcal{A}}$. $\square$

5. Intrinsic theories

5.1. **Intrinsic theories for inductive data.** *Intrinsic theories* for inductive data-types were introduced in [19]. 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 productivity. In particular, declarative programs are considered as formal theories. This departs from two longstanding approaches to reasoning about programs and their termination, namely programs as modal operators [36, 27, 11], and programs (and their computation traces) as explicit mathematical objects [16, 17].

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 $\mathcal{D}$.
- **Separation axioms for $\mathcal{C}$,** stating that the constructors are injective and have pairwise-disjoint codomains. These imply that all data-terms are distinct.

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

$$Const[\vec{\varphi}] \;\rightarrow\; (\wedge_i \forall x\, \mathbf{D}_i(x) \rightarrow \varphi_i[x]) \tag{5.1}$$

  where $Const[\vec{\varphi}]$ is the conjunction of the construction rules for the bundle, with each $\mathbf{D}_i(\mathbf{t})$ replaced by $\varphi_i[\mathbf{t}]$. The formulas $\vec{\varphi}$ are the *induction-formulas* of the delineation.

Example: 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 type-identifier W. Here we have the

- inductive closure rules:

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

- and inductive-delineation:

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

Definition 5.1. A unary program $(P, \mathbf{f})$ is *provably of type* $\mathbf{D} \rightarrow \mathbf{E}$ in a theory $\mathbf{T}$ if $\mathbf{D}(x) \rightarrow \mathbf{E}(\mathbf{f}(x))$ is provable in $\mathbf{T}$ from the universal closure of the equations in P.[11]

Theorem 5.2. [20, 19].

(1) *A function f over $\mathbb{N}$ has a program provably of type* $\mathtt{N} \rightarrow \mathtt{N}$ *in the intrinsic theory* $\mathbf{IT}(\mathbb{N})$ *iff it is a provably-recursive function of Peano's Arithmetic, i.e. a function definable using primitive-recursion in finite types.*

(2) *f has a program proved to be of type* $\mathtt{N} \rightarrow \mathtt{N}$ *using only formulas in which* $\mathbb{N}$ *does not occur negatively iff f is a primitive-recursive function.*

Note that this characterization of the provable functions of PA involves no particular choice of base functions (such as addition and multiplication). See [19] for examples and discussion.

5.2. **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 Separation axioms, the inductive construction rules and coinductive deconstruction 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 construction rules for an inductive bundle $\vec{\mathbf{D}}$, then it contains $\vec{\mathbf{D}}$:

$$Const[\vec{\varphi}] \;\rightarrow\; (\wedge_i \forall x\, \mathbf{D}_i(x) \rightarrow \varphi_i[x])$$

  where $Const[\vec{\varphi}]$ is the conjunction of the construction rules for the bundle, with each $\mathbf{D}_i(\mathbf{t})$ replaced by $\varphi_i[\mathbf{t}]$.

[11] Universal closure is needed, since the logic here is first-order, rather than equational.

- **Coinductive delineation (data-introduction, Coinduction)**: If a vector $\vec{\varphi}[x]$ of first order formulas satisfies the deconstruction rule for a coinductive bundle $\vec{\mathbf{D}}$, then it is contained in $\vec{\mathbf{D}}$:
$$\mathit{Deconst}[\vec{\varphi}] \;\rightarrow\; (\wedge_i \forall x\; \varphi_i[x] \rightarrow \mathbf{D}_i(x)) \tag{5.2}$$
where $\mathit{Deconst}[\vec{\varphi}]$ is the conjunction of the deconstruction rules for the bundle, with each $\mathbf{D}_i(\mathbf{t})$ replaced by $\varphi_i[\mathbf{t}]$.

A characterization result, analogous to Theorem 5.2(2), was proved in [21]: A function over a coinductive type is definable using corecurrence iff its productivity is provable using coinduction for formulas in which type-identifiers do not occur negatively. The proof in [21] is for streams, the general result will be proved elsewhere, as well as an analog Theorem 5.2(1).

The phrase *coinduction* is often mentioned in reference to the principle enunciated by David Park, *"To prove two processes observationally equivalent, show that they are bisimilar"* (see e.g [33]). The phrase "observational equivalent" is sometimes taken to mean "equal." Park's principle is not directly derivable in the intrinsic theory of given coinductive types because the equality primitive of intrinsic theories is untyped, acting as a rewrite rule (a *definitional equality* in the sense of Martin-Löf's type theory). However, intrinsic theories do derive Park's principle for equality-in-a-type. Consider the following program for a fresh function identifier `eq`.

$$\begin{array}{lcll} \mathtt{eq}(\mathbf{c}(x_1,\ldots,x_r),\mathbf{c}(y_1,\ldots,y_r)) & = & \mathbf{c}(\mathtt{eq}(x_1,y_1),\ldots,\mathtt{eq}(x_r,y_r)) & \\ & & & \mathbf{c} \text{ a constructor of arity } r \\ \mathtt{eq}(\mathbf{c}(x_1,\ldots,x_r),\mathbf{d}(y_1,\ldots,y_t)) & = & \boldsymbol{\xi} & \mathbf{c}, \mathbf{d} \text{ distinct constructors} \end{array}$$

Here $\boldsymbol{\xi}$ is a nullary constructor, not used in any type definition of the data system. That is, `eq` maps two equal hyper-terms into their common value, and maps two distinct hyper-terms into a hyper-term containing $\boldsymbol{\xi}$, which is therefore in no type. Now define an equality relation for a coinductive type $\mathbf{D}$ by

$$x \doteq_{\mathbf{D}} y \quad \equiv \quad \mathbf{D}(\mathtt{eq}(x,y))$$

Of course, this equality is undecidable, as indeed should be the case for infinite hyper-terms.

Given a bi-simulation between x and y, the program for `eq` can be used to obtain the premises of coinduction for the unary predicate $\lambda x.\mathbf{D}(\mathtt{eq}(x,y))$ (i.e. with y as parameter). Our Coinduction scheme (5.2) then implies $\mathbf{D}(\mathtt{eq}(x,y))$, i.e. $x \doteq_{\mathbf{D}} y$.

6. Proof theoretic strength

6.1. **Innocuous function quantification.** 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, their deductive machinery does not imply the existence of any particular coinductive object, as would be the case, for example, in the presence of some forms of the Axiom of Choice or of a comprehension principle. Coinductive objects can be specified, of course, by programs, but such programs are treated as axioms, i.e. assumptions.

We show next that, as a consequence, any intrinsic theory **T** is interpretable in a formal theory whose proof theoretic strength is no greater than that of Peano Arithmetic.

We take as starting point 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.[12] 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).

Let **PRA*** be **PRA** augmented with function variables and quantifiers over them, as well as free variables for functionals (i.e. functions from numeric functions to numeric functions.) The set of *terms* is built by type-correct explicit definition (i.e. composition and application) from number-, function-, and functional-variables, starting with 0 and identifiers for all primitive-recursive functions. The theory has as axiom schema the Principle of Explicit Definition: for each term $\mathbf{t}[\vec{x}, \vec{f}]$ of the extended language, with number variables $\vec{x}$ and function variables $\vec{f}$,

$$\forall \vec{f}\, \exists g\, \forall \vec{x}\;\; g(\vec{x}) = \mathbf{t}[\vec{x}, \vec{f}]$$

There are no further axioms stipulating the existence of additional functions.

The schema of Induction applies now to all formulas in the extended language.

Lemma 6.1. *The theory* **PRA*** *is conservative over* **PRA**. *That is, if a formula in the language of* **PRA** *is provable in* **PRA***, *then it is provable already in* **PRA**.

Consequently, **PRA*** *is no stronger, proof-theoretically, than* **PA**.

Proof. The proof is virtually the same as that in [39, Prop. 1.14, p. 453], that **E-HA**$^\omega$ is conservative over **HA**.[13] The use of classical logic, rather than constructive (intuitionistic) logic, makes here no difference, and **PRA*** is a sub-theory of (the classical counterpart of) **E-HA**$^\omega$. □

The main result of this section, and the second of the paper, is the following evaluation of the proof theoretic strength of intrinsic theories, which turns out to be surprisingly modest.

Theorem 6.2. (Arithmetic interpretability) *Every intrinsic theory is interpretable in* **PRA***.

As will become clear from the proof of Theorem 6.3 below, Theorem 6.2 depends on our avoiding infinite-branching type constructions, such as W-types.

6.2. **Representing data by numeric functions.** We posit canonical primitive-recursive coding-scheme $\langle \cdots \rangle$ for sequences of natural numbers. More generally, we assume that basic syntactic operations on finite data-terms, such as application and sub-term extraction, are represented by primitive recursive functions. See e.g. [16, 31] for details, related notations, and proofs of the closure of the primitive-recursive functions and predicates under major operations, such as bounded quantification and minimization.

For each constructor $\mathbf{c}$, let $\mathbf{c}^\sharp$ be a distinct numeric code. We say that a function $f : \mathbb{N} \to \mathbb{N}$ *represents* a hyper-term $t \in H_{\mathcal{C}}$ if f maps addresses $a = \langle a_0 \cdots a_k \rangle \in \mathbb{N}$ to the code $\mathbf{c}^\sharp$ of the constructor $\mathbf{c}$ at address a of t, whenever such a constructor exists. (We could insist that $f(a)$ be some flag, say 0, when t has no constructor at address a, thereby

[12] See e.g. [38] for details and related discussions.

[13] I am grateful to Ulrich Kohlenbach for pointing me to that reference.

determining f uniquely from t; but this would be of no use to us, and would imply the undecidability of determining whether two computable functions represent the same *finite* term.)

For example, the finite term $\mathtt{p}(\mathtt{e}, \mathtt{0}(\mathtt{e}))$ is represented by f provided $f\langle\rangle = \mathtt{p}^\sharp$, $f\langle 0\rangle = \mathtt{e}^\sharp$, $f\langle 1\rangle = \mathtt{0}^\sharp$, and $f\langle 1, 0\rangle = \mathtt{e}^\sharp$. Similarly, the infinite 01-word $(\mathtt{0},\mathtt{1})^\omega = \mathtt{0101}\cdots$ is represented by f provided $f\langle 0^{2n}\rangle = \mathtt{0}^\sharp$ and $f\langle 0^{2n+1}\rangle = \mathtt{1}^\sharp$ $(n \geq 0)$.

It follows that an r-ary constructor $\mathbf{c}$ is represented by a functional $\mathbf{c}^\Box$ provided

$$\begin{array}{rcl} \mathbf{c}^\Box(f_1, \ldots, f_r)\langle\rangle & = & \mathbf{c}^\# \\ \mathbf{c}^\Box(f_1, \ldots, f_r)(\langle i\rangle * a) & = & f_i(a) \quad i = 1..r \end{array}$$

6.3. **Representing types by formulas.** Consider a purely co-inductive data-system. One can state that a hyper-term t is of type $\mathbf{D}$ by asserting the existence of a correct type decoration of the nodes of t, with the root assigned type $\mathbf{D}$. The correctness of the decoration can be expressed by a single numeric $\forall$, but we would need to have an existential function-quantifier to state, in the first place, the existence of the decoration. We show here that no such function quantification is needed. Referring to Definition 2.3, we have:

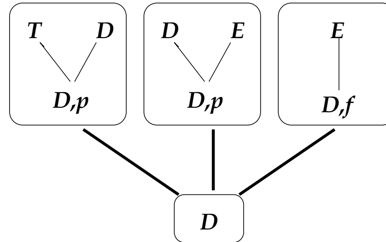
Theorem 6.3. *Σ_n types are defined by Σ^0_n formulas, and Π_n types by Π^0_n formulas.*

Proof. The proof is by induction on $n \geq 1$. If a type $\mathbf{D}$ is Σ_n, i.e. is in a bundle defined inductively from Π_{n-1} types (where we take Π_0 to be empty), then a hyper-term t has type $\mathbf{D}$ iff there is a finite deduction establishing $\mathbf{D}(t)$ from typing-statements of the form $\mathbf{E}_j(t_j)$, with each $\mathbf{E}_j$ a type of lower rank, and $t_j = \Pi(t)$ for some deep-destructor Π. By IH each $\mathbf{E}_j$ is defined by some Π^0_{n-1} formula $\mathbf{E}^\Box[t_j]$ and the correctness of the finite type-derivation is clearly a primitive-recursive predicate. Thus $\mathbf{D}$ is definable by existential quantification over Π^0_{n-1} formulas, i.e. by a Σ^0_n formula $\mathbf{D}^\Box$.

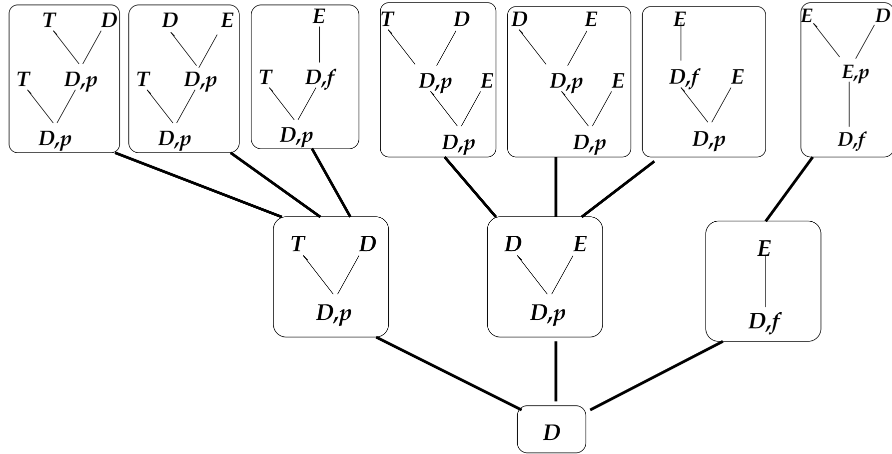
Consider now a Π_n type $\mathbf{D}$. We shall concretize the general argument using a running example, with types $\mathtt{D}$ and $\mathtt{E}$ defined by a common coinduction, and $\mathtt{T}$ a (previously defined) Σ_{n-1} type:

$$\begin{array}{rcl} \mathtt{D}(x) & \to & \exists y,z\; x = \mathtt{p}(y,z) \wedge \mathtt{D}(y) \wedge \mathtt{E}(z) \\ & & \vee\; \exists y,z\; x = \mathtt{p}(y,z) \wedge \mathtt{T}(y) \wedge \mathtt{D}(z) \\ & & \vee\; \exists y\; x = \mathtt{f}(y) \wedge \mathtt{E}(y) \\ \mathtt{E}(x) & \to & \exists y,z\; x = \mathtt{p}(y,z) \wedge \mathtt{E}(y) \wedge \mathtt{D}(z) \end{array}$$

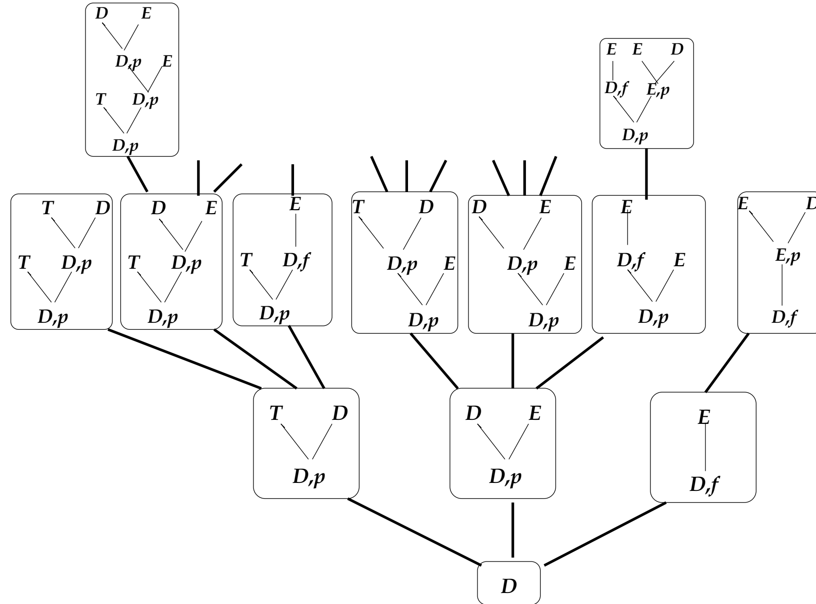
The decomposition rule for each type has a number of constructor-statements as choices, in our example $\mathtt{D}$ has three and $\mathtt{E}$ one. Each choice determines a main constructor, and types for the component. The spelling out of $\mathtt{D}$ into three options can be represented graphically:



We continue an expansion of all typing options for a hyper-term in D. That is, we construct a tree T_D, where a node of height h consists of a finite tree (of height $\leq h$), with types at the leaves, and a pair of a type and a constructor at internal nodes. Each such node represents a possible partial typing of a hyper-term of type D. The children of each such node N are the local expansions of the lowermost-leftmost unexpanded leaf, with a type in the bundle considered. (E.g., in our running example, the leaves with type T are not expandable, and are left alone.) Put differently, the leaves are expanded in a breadth-first order. (We refrain from expanding all expandable leaves at each step, because the resulting tree, albeit finitely-branching, would have unbounded degree.)



A few nodes of height 3 are given here:



Note that the tree T_D is primitive recursive, i.e. there is a primitive-recursive function that, for every address α, gives (a numeric code for) the node at address α.

A hyper-term $t \in H_{\mathcal{C}}$ is *consistent* with a node N as above if its constructor-decomposition is consistent with the tree of constructors in N, and for every deep-destructor Π, if N has

at address Π a type $\mathbf{E}$ of lower rank, then $\Pi(t)$ has type $\mathbf{E}$. The consistency of a hyper-term t with a node N is thus definable by a Σ^0_{n-1} formula.

A hyper-term has type $\mathtt{D}$ iff there is an unbounded (i.e. infinite or terminating) branch of the tree T above, every node of which is consistent with t. The existence of such a branch is equivalent, by Weak König's Lemma, to the existence, for every $h > 0$, of a node N of height h in T_D, which is consistent with t. Since consistency of t with N is definable by a Σ^0_{n-1} formula, this property is Π^0_n. □

6.4. **Interpretation of terms.** For an r-ary constructor $\mathbf{c}$ let $\hat{\mathbf{c}}$ denote the PR functional that maps functions $f_0, \ldots, f_{r-1}$ representing hyper-terms $t_0, \ldots, t_{r-1}$ to a function representing the hyper-term obtained by rooting $t_0, \ldots, t_{r-1}$ from the symbol $\mathbf{c}$, i.e.

$$\begin{aligned} \hat{\mathbf{c}}(f_0, \ldots, f_{r-1})\langle\rangle &= \mathbf{c}^\sharp \\ \hat{\mathbf{c}}(f_0, \ldots, f_{r-1})\langle i\rangle * a &= f_i(a) \end{aligned}$$

Recall that we posit the presence in the vocabulary of $\mathbf{PRA}^*$ of identifiers for all PR functionals, in particular $\hat{\mathbf{c}}$.

Next, we define a mapping $\mathbf{t} \mapsto \mathbf{t}^\square$ from terms of $\mathbf{T}$ to terms of $\mathbf{PRA}^*$. We posit that the identifiers of $\mathbf{PRA}^*$ for PR functions and functionals are disjoint from the program identifiers of intrinsic theories.

- For a variable x of $\mathbf{T}$ (intended to range over hyper-terms) we let $x^\square$ be a fresh unary function variable of $\mathbf{PRA}^*$ (intended to range over functions representing hyper-terms).
- For a constructor $\mathbf{c}$ of arity $r \geq 0$, let $(\mathbf{c}(\mathbf{t}_1 \ldots \mathbf{t}_r))^\square =_{\text{df}} \hat{\mathbf{c}}(\mathbf{t}_1^\square \ldots \mathbf{t}_r^\square)$.
- For a program-function $\mathbf{f}$ of arity $r \geq 0$ (i.e. a free variable denoting a function between hyper-terms), let $(\mathbf{f}(\mathbf{t}_1 \ldots \mathbf{t}_r))^\square =_{\text{df}} \mathbf{f}^\square(\mathbf{t}_1^\square \ldots \mathbf{t}_r^\square)$, where $\mathbf{f}^\square$ is a fresh functional variable of $\mathbf{PRA}^*$, of arity r.

6.5. **Interpretation of formulas.** Finally, we define a mapping $\varphi \mapsto \varphi^\square$ from formulas of $\mathbf{T}$ (possibly with program-functions) to formulas of $\mathbf{PRA}^*$. Let $\mathit{Htm}[g]$ be a PR formula stating that the function g represents a hyper-term.

- $(\mathbf{t} = \mathbf{q})^\square$ is $\mathbf{t}^\square = \mathbf{q}^\square$.
- $(\mathbf{D}(\mathbf{t}))^\square$ is $\mathbf{D}^\square[\mathbf{t}^\square]$, where $\mathbf{D}^\square$ is the arithmetic formula (possibly with free function and functional variables) that defines $\mathbf{D}$ (Theorem 6.3).
- $(\varphi \wedge \psi)^\square$ is $\varphi^\square \wedge \psi^\square$, and similarly for the other connectives.
- $(\forall x\, \varphi)^\square$ is $\forall x^\square\ \mathit{Htm}[x^\square] \rightarrow \varphi^\square$; $(\exists x\, \varphi)^\square$ is $\exists x^\square\ \mathit{Htm}[x^\square] \wedge \varphi^\square$.

Proposition 6.4. *The mapping $\varphi \mapsto \varphi^\square$ is semantically faithful; that is, for each formula $\varphi[\vec{x}, \vec{\mathbf{f}}]$ of $\mathbf{T}$, with free object variables among $\vec{x}$ and program-variables among $\vec{\mathbf{f}}$,*

$$\mathcal{A},\ [\vec{x} \leftarrow \vec{t},\ \vec{\mathbf{f}} \leftarrow \vec{g}] \models \varphi$$

iff for all unary functions $\vec{h}$ over $\mathbb{N}$ representing (respectively) the hyper-terms $\vec{t}$, and for all functionals $\vec{G}$ representing (respectively) the functions $\vec{g}$,

$$\mathcal{N},\ [\vec{x}^\square \leftarrow \vec{h},\ \vec{\mathbf{f}}^\square \leftarrow \vec{G}] \models \varphi^\square$$

In particular, if φ is a closed formula of $\mathbf{T}$, then φ is true in the canonical model $\mathcal{A}$ of the data-system iff $\varphi^\square$ is true in the standard model of $\mathbf{PRA}^*$.

Proof. The proof is straightforward by structural induction on φ. □

6.6. **An Interpretability Theorem.** We finally show that the interpretation is proof-theoretically faithful.

Theorem 6.5. *If a closed formula* φ *is provable in the intrinsic theory* $\mathbf{T}$*, then* $\varphi^{\Box}$ *is provable in* $\mathbf{PRA}^{*}$*.*

More generally: if a formula $\varphi[\vec{x}]$*, with free variables among* $\vec{x}$*, is provable in* $\mathbf{T}$*, then* $Htm[\vec{x}] \rightarrow \varphi^{\Box}[\vec{x}]$ *is provable in* $\mathbf{PRA}^{*}$*.*

Proof. The proof proceeds by structural induction on derivations.

Logic: The propositional and quantifier inferences are trivially pressured by the interpretation.

Separation: The case of the Separation Axioms is immediate by the definition of the interpretation.

Inductive construction: Consider the construction axiom (2.2) for an inductive bundle $\vec{\mathbf{D}}$,

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

where each ψ_i is a constructor-statement. The interpretation of (2.2) is

$$Htm[\vec{x}^{\Box}] \rightarrow (\psi_1^{\Box} \vee \cdots \vee \psi_k^{\Box} \;\rightarrow\; \mathbf{D}_i^{\Box}[x^{\Box}])$$

with

$$\psi_i^{\Box} \quad \text{of the form} \quad \exists y_1^{\Box} \ldots y_r^{\Box} \in Htm \;\; x^{\Box} = \mathbf{c}^{\Box}(\vec{y}) \;\wedge\; \mathbf{Q}_1^{\Box}[y_1] \;\wedge \cdots \wedge \mathbf{Q}_r^{\Box}[y_r]$$

Recall (from Theorem 6.3) that $\mathbf{D}_i^{\Box}[x^{\Box}]$ states the existence of a finite type derivation Δ of $\mathbf{D}[x^{\Box}]$ from statements of the form $\mathbf{E}[\Pi(x^{\Box})]$ with $\mathbf{E}$ of lower rank and Π a deep-destructor. Thus one of the decompositions $\psi_i^{\Box}$ must be true for $x^{\Box}$, with the correctness of the $\mathbf{Q}_j^{\Box}[y_j]$ true by induction on the height of Δ.

Induction: Given an inductive bundle $\vec{\mathbf{D}}$, the interpretation of $\vec{\mathbf{D}}$-induction for formulas $\vec{\varphi}$ (5.1) is

$$\mathbf{D}_i^{\Box}[x^{\Box}] \;\rightarrow\; (Const^{\Box}[\vec{\varphi}^{\Box}] \;\wedge\; Htm[x^{\Box}] \;\rightarrow\; \varphi_i^{\Box}[x^{\Box}]) \tag{6.1}$$

Recall that $\mathbf{D}_i^{\Box}[x^{\Box}]$ states the existence of a finite derivation Δ of $\mathbf{D}(x^{\Box})$ from formulas of the form $\mathbf{E}(\Pi(x^{\Box}))$, where $\mathbf{E}$ is of lower rank than $\mathbf{D}$, and Π is a deep-destructor. The conclusion of (6.1) is straightforward by cumulative (i.e. course-of-value) induction on the height of Δ.

Coinductive deconstruction: A deconstruction axiom (2.5) for a coinductive bundle $\vec{\mathbf{D}}$ has the interpretation

$$\mathbf{D}_i^{\Box}[x^{\Box}] \;\rightarrow\; \psi_1^{\Box} \vee \cdots \vee \psi_k^{\Box}$$

where each ψ_i is a constructor-statement. Recall that the definition of $\mathbf{D}_i^{\Box}$ in this case (Theorem 6.3) refers to the tree T_D of expansion-options for objects of type $\mathbf{D}$. Continuing our running example in the proof of Theorem 6.3, $\mathbf{D}_i^{\Box}[x^{\Box}]$ implies the existence, at every height, of an expansion of $\mathbf{D}$ which is consistent with the structure of $x^{\Box}$. Nodes of height 2 which are consistent with $x^{\Box}$ give its decomposition, say as $\mathtt{p}(y^{\Box}, z^{\Box})$. One of these nodes must have above it nodes of arbitrary height consistent with $x^{\Box}$.[14] If that node is the leftmost, giving $\mathtt{T}(y^{\Box})$ and $\mathtt{D}(z_0)$, then we have $\mathtt{D}^{\Box}[z^{\Box}]$ by assumption, and $\mathtt{T}^{\Box}[y^{\Box}]$ holds by the definition of $\mathtt{D}^{\Box}$ (since the node for $\mathtt{T}(y^{\Box})$ is a leaf of T_D).

[14]Note that we do not use here Weak König's Lemma, as we do not assert the existence of an infinite branch as a consequence.

Coinduction: Given a coinductive bundle $\vec{\mathbf{D}}$, the interpretation of $\vec{\mathbf{D}}$-coinduction for formulas $\vec{\varphi}$ (2.5) is

$$\varphi_i^{\Box}[x^{\Box}] \rightarrow ((Deconst^{\Box}[\vec{\varphi}^{\Box}] \wedge Htm[x^{\Box}]) \rightarrow \mathbf{D}_i^{\Box}[x^{\Box}]) \tag{6.2}$$

The conclusion of (6.2) is established by showing that the tree T_D (see the proof of Theorem 6.3) has a node consistent with $x^{\Box}$ at any given height h. This follows outright from the assumptions of (6.2) by *induction* on h. □

7. 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_2^0-conservative over intuitionistic arithmetic [19], and a particularly simple characterization of the primitive-recursive functions [20]. The latter application guided a dual characterization of the primitive corecursive functions in terms of intrinsic theories with positive coinduction [21].

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. Recent work by Berger and Seisenberg [4] has already explored similar ideas.

Finally, intrinsic theories are naturally amenable to ramification, leading to a transparent Curry-Howard link with ramified recurrence [2, 18] as well as ramified corecurrence [28].

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] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183(1):19–42, 2003.

[4] Ulrich Berger and Monika Seisenberger. Proofs, programs, processes. *Theory Comput. Syst.*, 51(3):313–329, 2012.

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

[6] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

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

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

[9] 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.

[10] Yuri Gurevich. Logic and the challenge of computer science. In *trends in theoretical computer science*, pages 1–57. Computer Science Press, Rockville, MD, 1988.
[11] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, Cabridge, MA, 2000.
[12] Jacque Herbrand. Sur la non-contradiction de l'arithmtique. *Journal für die reine und angewandte Mathematik*, 1932:1–8, 1932. English translation in [41] 618–628.
[13] 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.
[14] Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische annalen*, 112:727–742, 1936.
[15] Stephen C. Kleene. Lambda definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.
[16] Stephen C. Kleene. *Introduction to Metamathematics*. Wolters-Noordhof, Groningen, 1952.
[17] Stephen C. Kleene. *Formalized Recursive Functions and Formalized Realizability*, volume 89 of *Memoirs of the AMS*. American Mathematical Society, Providence, 1969.
[18] 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.
[19] Daniel Leivant. Intrinsic reasoning about functional programs I: First order theories. *Annals of Pure and Applied Logic*, 114:117–153, 2002.
[20] Daniel Leivant. Intrinsic reasoning about functional programs II: Unipolar induction and primitive-recursion. *Theor. Comput. Sci.*, 318(1-2):181–196, 2004.
[21] 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.
[22] Yiannis N. Moschovakis. The formal language of recursion. *J. Symb. Log.*, 54(4):1216–1252, 1989.
[23] Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-coalgebraic specification in CoCasl. *J. Log. Algebr. Program.*, 67(1-2):146–197, 2006.
[24] 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.
[25] Peter Padawitz. Swinging types=functions+relations+transition systems. *Theor. Comput. Sci.*, 243(1-2):93–165, 2000.
[26] Giuseppe Peano. *Arithmetices principia, novo methodo exposita*. Fratres Bocca, Torino, 1889. English translation in [41], 83–97.
[27] Vaughan R. Pratt. Semantical considerations on floyd-hoare logic. In *FOCS*, pages 109–121. IEEE Computer Society, 1976.
[28] Ramyaa Ramyaa and Daniel Leivant. Ramified corecurrence and logspace. *Electr. Notes Theor. Comput. Sci.*, 276:247–261, 2011.
[29] 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.
[30] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
[31] H.E. Rose. *Subrecursion*. Clarendon Press (Oxford University Press), Oxford, 1984.
[32] Jan Rothe, Hendrik Tews, and Bart Jacobs. The coalgebraic class specification language CCSL. *J. UCS*, 7(2):175–193, 2001.
[33] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.
[34] Lutz Schröder. Bootstrapping inductive and coinductive types in HasCASL. *Logical Methods in Computer Science*, 4(4), 2008.
[35] Helmut Schwichtenberg and Stanley S. Wainer. *Proofs and Computations*. Perspectives in logic. Cambridge University Press, 2012.
[36] Krister Segerberg. A completeness theorem in the modal logic of programs (preliminary report). *Notices American matheamtical society*, 24:A–552, 1977.
[37] 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.
[38] S. Simpson. *Subsystems of Second-Order Arithmetic*. Springer-Verlag, Berlin, 1999.

[39] A.S. Troelstra and D. van Dalen. *Constructivism in mathematics: an introduction. Vol. 2*. Constructivism in Mathematics. North-Holland, 1988.

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

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

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nd/2.0/` or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany