TOWARDS A PROGRAMMING LANGUAGE ONTOLOGY

Raymond Turner
Department of Computer Science,
University of Essex, UK

Amnon H Eden
Department of Computer Science,
University of Essex, UK
and Center For Inquiry

September 9, 2005

Abstract

We examine the role of semantic theory in determining the ontology of programming languages. We explore how different semantic perspectives result in different ontologies. In particular, we compare the ontological implications of set-theoretic versus type-theoretic semantics.

1 Programming Languages

Programming languages (PLs) combine two, not always distinct, facilities: data structures (e.g., numbers, lists, trees, finite sets, objects) and control structures (e.g., assignment, iteration, procedure calls, recursion) that operate on these data structures. Data and control structures come in all shapes and sizes [12], [10], [8]. In particular, they are immensely varied in their style of presentation and conceptual content. For example, logic based languages such as Prolog are based upon Horn clause logic with a very primitive type structure. Functional ones such as MirandaTM have no imperative features and use recursion equations and inductive data types as their means of constructing programs. Imperative languages such as C and Algol employ a whole range of data types and are characterised by the presence of assignment and its associated imperative constructs. Object-oriented languages such as Smalltalk insist that everything is an object or, at very least, can be naturally represented as such. In all of them, we can express very complex ideas and construct extremely complicated and intricate algorithms. Indeed, PLs may be usefully thought of as theories of problem solving in the sense that each language proffers a means of solving...
problems with given methods of representation and problem solving strategies enshrined in their techniques of program construction. To varying degrees, whichever language one chooses, it forces one to solve problems within a given conceptual framework (technically, a programming paradigm). This underlying conceptual framework is what we have in mind when we talk about the ontology of a PL. Our objective in this paper is to say something about such ontologies: what is the best way of characterising and formalising them? Unfortunately, the very diversity of PLs makes it hard to see how to start any such investigation. Consequently, at the outset we say a little about our approach [16], [7] to ontology.

2 Ontological Perspectives

Since PLs are formal languages, they have much in common with the formal languages used in mathematics and physics. Indeed, many have been inspired by the languages of formal logic. Consequently, we shall explore some of the ontological options that have been suggested for these logically expressed languages. We begin with a famous one.

Quine [13], [14] distinguishes between the ontological commitments of a scientific theory and the actual choice of a theory itself. The latter is to be decided in terms of overall scientific adequacy i.e., explanatory power, simplicity, elegance etc. However, once the theory has been selected, within it, existence is determined by what the theory says exists i.e.,

To be is to be the value of a bound variable

In theories stated within the languages of formal logic, this unpacks in terms of existential quantification. For example, Peano arithmetic is committed to zero and the successor of any number; second order arithmetic, to numbers and sets of numbers; and Russell’s simple type theory to sets, sets of sets and sets of sets of sets etc. This also applies to scientific theories expressed within such languages. So for example, axiomatic accounts of Newtonian physics are committed to the mathematics required to express the physics together with any of the actual physical notions that do not form part of the underlying pure mathematics. The same is true for any axiomatic formulation of any branch of science. Each theory has its own existential commitments and these are explicitly laid out by the axioms and rules of the theory-and these determine its underlying ontology.

It would be pleasant to be able to apply these ontological guidelines to PLs i.e., to be able to read off the underlying ontology via its existential structure. But before we explore this possibility there is one aspect of the Quinean perspective that we do not want to adopt. In so far as he believes that scientific theories are attempts to describe an independently existing reality, the languages of our theories, and their subsequent ontologies, are ways of describing that reality. This is a realist perspective. And although we have suggested that PLs may be usefully thought of as theories of problem solving, we do not wish to defend a
realist position about such theories-and for very obvious reasons. Fortunately, Putnam, Carnap and Strawson [18] can be brought to the rescue. Although they accept the internal existential commitments of a formalised ontology, it is not taken to describe an independently existing reality. It is thus stripped of its realist roots. For them, like their predecessor Kant, Ontology is concerned with belief systems/conceptual frameworks and their articulation within the language of the theory: so-called Internal Metaphysics [16]. While the Quinean criteria of existence within such theories remain relevant, such systems are no longer realist. This is closer to the kind of stance we would wish to defend about our so-called theories of problem solving: they are by their very nature alternative conceptual frameworks. With this much clarified, we can return to our goal of applying these ontological guidelines to PLs.

Unfortunately, as soon as we try to do so, we are presented with an obstacle: unlike logic languages, no programming language is explicitly distilled as a pure logical theory, not even PROLOG. We are rarely given anything like an axiomatisation that exhibits its ontological commitments. So unlike logical languages, uncovering the ontological underpinnings of a PL is not quite as simple as searching through the axioms. In general, we are only presented with a syntax of the programming language together with some informal account of its semantics. What we are rarely given is anything like an axiomatisation that displays its existential commitments\(^1\). So this approach offers little direct help.

A closely allied alternative line of enquiry is offered by Frege and Dummett [1], [2], [4]. Instead of focusing on an axiomatic account of its constructs, it associates ontology with semantics. It can be crudely summarized as follows.

What there is, is what is required to furnish a semantic account of the language

In other words, the ontological implications of a language are to be identified with the entities required to provide its constructs with a semantics. For logical languages, such semantic accounts are normally given in the form of its model-theoretic characterisation. Indeed, the axioms of the theory are often justified by direct appeal to its semantics. Consequently, for these languages, the two ontological strategies are two sides of the same coin. This suggests that we might be able to apply the Quinean perspective via its semantic dimension.

But once more, things are a little less straightforward when one tries to apply this ontological criterion to PLs. Indeed, for them, even the statement of this semantic strategy is a little vague; presumably, it all depends upon what one means by semantic account. These are not logical languages whose semantics can be given using Tarski satisfaction. Moreover, the notion that PLs have or need a formal semantics came much after their invention and use.

\(^1\)The nearest we get to any axiomatic account of a PL is in the work of Manna and Hoare. However, they were mainly concerned with program correctness and devised ways of attaching predicate information to programs; information intended to record the before and after states of the execution of a piece of program text. Moreover, while the presentation of Hoare [5] is axiomatic, it does not capture all the ontological commitments of a language; it only records the impact of the constructs on some underlying notion of state.
Fortunately for us, semantics of PLs has now been under investigation for several decades, and so the semantic approach offers us a strategy to move forward. While several flavours of semantics are on offer, Denotational Semantics (DS)\textsuperscript{[15]}, \textsuperscript{[17]}, \textsuperscript{[26]} is not only the market leader, but is also the most ontologically self-conscious. Consequently, DS will form our starting point. Eventually, by following this trail and abstracting the central features, we shall be led back to the primary Quinean perspective.

3 Set-Theoretic Semantics

It was Frege\textsuperscript{[4]} who pioneered the idea that meaning should be unpacked in terms of mathematical notions of some kind. In his case he was concerned with analysing natural language as a way of throwing light on the nature of mathematical argumentation. For him, all natural language constituents (proper names, predicative expressions, adjectives and quantifiers) are to be compositionally assigned a denotation. For example, proper names have simple objects as their referents while predicative expressions are taken to refer to certain kinds of functions. While it is the case that his notion of function is not the one found in modern set theory, it is the ancestor of all such mathematical accounts. Indeed, his perspective in natural language semantics was later extended and developed by Montague\textsuperscript{[19]}. However, by the time that Montague developed his semantics, Zermelo Fraenkel set theory had moved to its current dominant semantic position and semantics had come to mean set-theoretic semantics\textsuperscript{[20]}, \textsuperscript{[24]}.

This semantic background was inherited by the researchers developing semantic theories for PLs and was one of the driving conceptual forces behind the Scott-Strachey paradigm of DS. Consequently, in DS every data item and control feature is taken to denote a mathematical object of some kind. At a deeper level, all the central notions of the underlying mathematics are sets. For example, the functions employed in the semantics are sets of ordered pairs. This is such a common characterisation of the notion of function that, as an assumption, it is rarely made explicit and it even seems somewhat pedantic to spell it out. However, it will play a crucial role in our evaluation of DS as providing an acceptable ontology for PLs.

In order to be more explicit about matters, and draw out some of its underlying assumptions, we need to sketch the DS approach. We are advocating a strategy of ontological classification with DS at its heart and, although we shall discard a great deal of the underlying mathematical foundations, we shall maintain the essence of DS, as well as the methodology of recovering the ontology from it. For this reason we need to spend a little time developing the basic ideas and machinery of DS.
3.1 SIL: Simple Imperative Language

We begin with a simple imperative language (SIL). Although it is a toy language, it will suffice to illustrate some of the central aspects of DS. An equally comprehensive analysis of a real language would not only be impractical, it would actually obscure many of the central conceptual issues. We shall later enrich SIL to illustrate some others that are significant from the perspective of PL ontology.

SIL has three syntactic categories: Booleans, numbers and commands. Booleans are generated by constants and the propositional connectives. The expression language contains zero and the successor operation while the actual command language, is generated by simple assignment statements, conditionals, sequencing and a while loop; it is given formally by the following BNF syntax.

\[
\begin{align*}
  b & ::= x \mid t \mid f \mid b \land b \mid \neg b \\
  e & ::= x \mid 0 \mid e^+ \\
  c & ::= x := e \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid c ; c \mid \textbf{while } b \textbf{ do } c
\end{align*}
\]

We shall use \(B\) as the set of Boolean expressions, \(E\) as the set of numerical expressions and \(C\) as the set of commands. The denotational semantics requires the following sets for the denotations of these syntactic categories; we shall see why shortly.

\[
\begin{align*}
  \text{Variables} & = \{x_1, x_2, x_3, \ldots\} \\
  \text{Bool} & = \{\text{true, false}\} \\
  \text{Values} & = \text{Nat} \\
  \text{Nat} & = \{0, 1, 2, 3\ldots\} \\
  \text{State} & = \text{Variables} \rightarrow \text{Values}
\end{align*}
\]

Here, and throughout DS, \(A \rightarrow B\) denotes some set of functions from the set \(A\) to the set \(B\). In addition, we require the following semantic functions for the various syntactic categories. Observe that they all require the state as an argument.

\[
\begin{align*}
  B & : B \rightarrow (\text{State} \rightarrow \text{Bool}) \\
  E & : E \rightarrow (\text{State} \rightarrow \text{Values}) \\
  C & : C \rightarrow (\text{State} \rightarrow \text{State})
\end{align*}
\]

Note that all three functions depend upon the state but only commands may change it. (This may not be so for programming languages, such as C, whose expressions also have side effects.) These are defined by recursion on the syntax. We shall illustrate two of the cases to draw attention to some ontological and mathematical requirements of any such set-theoretic semantics.

For ontology, one of the important semantic clauses is that for the assignment command. The reason is because it concerns the structure of the underlying notion of state and because assignments are the most distinguishing characteristics
Assignment changes the state: the Update function forces the value of $x$ in the new state, to be equal to the value of $e$ in the old one. As we shall see in a moment, the structure of the state is part of the ontological background of any imperative language.

Next consider the while loop. In DS this is unpacked in terms of the conditional as follows.

$$C[\text{while } b \text{ do } c] = \text{if } B[b]s \text{ then } C[\text{while } b \text{ do } c](C[c]s) \text{ else } s$$

The important aspect of this is that it is interpreted as a recursive definition i.e.,

$$F(s) = \text{if } B[b]s \text{ then } F(C[c]s) \text{ else } s$$

This is usually expressed by saying that the function $F$ is a fixed-point of the implicit functional expression.

To provide mathematically support for such semantics, we need to use a class of functions, indeed theory of mathematical structures and their associated functions, that is guaranteed to support such recursive definitions. Such structures were supplied by Scott’s Theory of Computation [15]: his theory of Domains\textsuperscript{2}. In DS all the sets employed are taken to be domains\textsuperscript{3}. Of particular importance, is the domain of functions from one domain to another. In domain theory, the set

$$D \rightarrow D'$$

is not taken to be the set of all functions from $D$ to $D'$ but just the set of Continuous ones\textsuperscript{4}.

Given this mathematical background, we can now read off the ontological commitments of our language. To provide DS for SIL, we require the sets Variables, Values, Bool and Nat to be domains. These are taken to be flat domains (i.e. sets with a bottom element added). Finally, we take

$$State = Variables \rightarrow Values$$

\textsuperscript{2}These are partly ordered sets $<D, \sqsubseteq>$ where $D$ is a set and $\sqsubseteq$ is a partial ordering, with a least element, such that any $\omega$-chain of elements has a least upper bound.

\textsuperscript{3}Or some other mathematical framework that supports all the constructions in a set-theoretic setting. The last 30 years has seen the development of many variations and alternative frameworks including those where the space of functions is cut down to just the computable ones-according to some indexing. But for us these variations are not philosophically significant since it is on the underlying set theory that we shall concentrate.

\textsuperscript{4}i.e. the ones that both preserve the orderings in $D$ and $D'$ and preserve the least upper bounds of $\omega$-chains.
to be the domain of continuous functions from the domain of variables to the
domain of values. This furnishes us with our set-theoretic ontology for the
language. It is important to observe that the domains required for the semantics
of a language do not just correspond to the data structures of that language.
The state domain is an important example of one that does not. They include
all the domains that we need to provide the semantic values for the constructs
of the language.

3.2 SILD

Of course, different languages require different domains i.e., they have different
ontological requirements. Not only do they have different data structures, they
require different notions of state. More specifically, in SIL, the state binds vari-
able directly to values. However, in most actual languages there is a difference
between declaring a variable and assigning a value to it. To reflect this we enrich
SIL by the addition of declarations

\[ d ::= declare \ x \mid d; d \]

i.e., a declaration is either simple (a variable is declared) or it is a sequence of
such. Call this language SILD. To model it, we bind variables/identifiers to
locations; such associations being called Environments. States are then reinter-
preted as continuous functions from Locations to Values.

\[ Env = Variables \rightarrow Loc \]
\[ States = Loc \rightarrow Values \]

where \( Loc \) is the domain of locations. In addition, we must introduce a new
semantic function for our new syntactic category of declarations. Here \( D \) is the
syntactic domain of declarations.

\[ \mathcal{D} : D \rightarrow [Env \rightarrow Env] \]

i.e., declarations change the environment. Variables, when declared, are now
bound to locations.

\[ \mathcal{D}[declare \ x]v = New[v, x] \]

The function \( New \) returns a new environment identical to the old but where \( x \) is
now attached to an unused location i.e. one not in the range of \( v \). Sequences of
declarations are unpacked via functional composition. However, we cannot stop
here since environments now effect the whole language and must be passed as
arguments. Consequently, our semantic functions now take the following shape.

\[ C : C \rightarrow (Env \rightarrow (State \rightarrow State)) \]
\[ E : E \rightarrow (Env \rightarrow (State \rightarrow Value)) \]
\[ B : B \rightarrow (Env \rightarrow (State \rightarrowBool)) \]
i.e., we have not only added new semantic domains but we must change all the semantic clauses. For example, in the clause for assignment, it is the value in the locations that is changed.

\[ C[x := e] vs = Update[s, v(x), E[e] vs] \]

This better reflects most imperative languages where declarations assign locations to variables and assignments change the state by updating the contents of such locations. This gives us a richer ontological picture and one that accurately reflects actual languages.

Furthermore, with these in place it makes more sense to add procedures to our declarations and procedure calls to our command language. For simplicity we include only procedures that take no arguments. These have the semantic import of changing the state. So we need the following domain to stand for these:

\[ Proc = State \rightarrow State \]

Presumably, when declared their locations (memory addresses) are stored as values in other locations and so the domain of values needs to be enriched.

\[ Values = Nat \oplus Proc \]

where \( \oplus \) is the disjoint union of the two. The reader should now be beginning to see how the sets/domains required to carry out the DS of a language are furnishing its ontology. Notice that we have flat domains, disjoints unions of domains and function space domains.

We have included procedures to draw attention to a technical problem that we shall have to deal with in any alternative framework. Since locations may now contain procedures, we have an indirect form of self application i.e. our domains are recursively defined. For example

\[ State = Variables \rightarrow [Nat \oplus [State \rightarrow State]] \]

Where the function space is the classical one (i.e., contains all the functions) this is a problem: it is just too big. Indeed, in general it is not sufficient to cut things down to just the continuous ones. However, there is a solution to this equation in domain theory. While the technical details of the solution are not philosophically germane, it is important to note that the existence of such solutions is necessary for the mathematical coherence of DS. Any alternative account of DS that is not based upon set/domain theory must address this issue.

\[ S_0 = Nat \]
\[ S_{n+1} = Variables \rightarrow [Nat \oplus [S_n \rightarrow S_n]] \]

\( State \), is then taken to be the inverse limit of this sequence.

\[ \text{The solution may be constructed by iterating the function space construction i.e.} \]
\[ S_0 = Nat \]
\[ S_{n+1} = Variables \rightarrow [Nat \oplus [S_n \rightarrow S_n]] \]

\( State \), is then taken to be the inverse limit of this sequence.

8
Domain theory provides an underlying mathematical system in which to articulate the DS semantics of PLs. Thus, according to our semantic motto, it is a candidate ontology for PLs. Indeed, it fulfills two tasks. Firstly, at the level of individual PLs, DS enables us to pinpoint the domains the language commits us to. For example, **SIL**, requires the flat domains `Variables`, `Values`, `Bool`, `Nat` and `State = Variables \rightarrow Values`. On the other hand, **SILD**, in addition to the flat domains, requires the following structures

\[
\begin{align*}
Env & = Variables \rightarrow Loc \\
States & = Loc \rightarrow Values
\end{align*}
\]

Finally, the addition of procedures adds the following domains with all their mathematical side effects.

\[
\begin{align*}
Proc & = State \rightarrow State \\
Values & = Nat \oplus Proc
\end{align*}
\]

So the underlying ontological demands of each language are articulated in terms of the domains required and the relationships between them. Moreover, over and above such individual language ontologies, the theory of domains provides the general ontological setting: it provides the general mathematical framework in which DS is situated.

This is a very attractive picture with semantics and ontology close bedfellows. It is not only a very satisfying approach to semantics but an insightful and useful way of pinpointing the underlying ontological structures of PLs. We have spent some time on it because, in any alternative approach to semantics and ontology, we shall want to maintain much of its top level framework. Indeed, it is only with the underlying theory, the general ontological setting of domain theory, that we shall take issue. Much of the semantic structure of DS will remain.

### 4 A Computer Science Perspective

There are three underlying assumptions of DS that need to be made explicit and reflected upon. These are:

- **Compositionality**
- **Extensionality**
- **Completeness**

*Compositionality* insists that the meaning of a complex expression is systematically put together from the meanings of its parts. In DS it unpacks as the demand that the semantic functions are defined by recursion on the syntactic structure of the language. It is largely uncontroversial, and in some form it is accepted in all versions of formal semantics. Indeed, it is such a fundamental
assumption of semantics in all its guises, that we shall not pause to question it. It is the others that need further reflection.

*Extensionality* relates primarily to the interpretation of programs; it insists that two programs are equal iff their denotations are the same. This means that their equality is given in terms of the functions they compute. It is enforced in DS by making the denotations of programs be functions.

The third assumption, *Completeness* is more of a consequence of the fact that the semantics is set-theoretic than a separate principle. It relates to the fact that all the semantic domains e.g., the natural numbers, are interpreted as sets given in extension.

One of the fundamental ontological questions about DS concerns the role of sets\(^6\). Some ontologists now accept *sets* as a basic ontological structure. They put them alongside individuals, properties and events, thereby taking them to be part of our conceptual framework. Within this framework, extensionality and completeness are natural assumptions. Indeed, they almost follow as corollaries.

Despite this, there is some reason to believe that sets should not form the basis of PL ontology. Indeed, there are some obvious reservations that motivate the following questions. Does the set-theoretic account of programming languages satisfy the intuitions of the computer scientist (CS)? Is this style of semantics definitive for the working CS? Does it define a language for her/him? We suspect that in all three cases the answer is negative. And the reason is because set-theoretic constructs do not reflect the intuitions of the CS whereas any worthy ontology should. A philosophically-correct ontology should reflect what the CS thinks she is talking about. It should be committed to data structures, programs and algorithms: things that can be computed with. Indeed, a possible ontological motto of CS might be

*What there is, is whatever can be computed with*

In other words, for the CS existence coincides with what can be implemented. This along with all our mottos is a bit vague but the intention is clear: computer scientists think in terms of data structures and programs. These and only these things exist in their conceptual universe. With these concerns to hand, consider the last two principles of DS.

Extensionality is interpreted in DS as the demand that programs denote set-theoretic functions. But from a CS viewpoint, programs are not best understood as sets. For a start, we cannot compute with infinite functions given in extension. Moreover, properties of programs such as efficiency, flexibility and elegance are, from the CS view of things, absolutely essential. But these properties are obliterated by the DS. In particular, whatever the appropriate notion of equality for algorithms and programs, it should be decidable (at least semi-decidable). This is certainly not delivered by its set-theoretic interpretation i.e. two programs are equal if they denote the same set-theoretic function.

\(^6\)We shall not so much reflect upon domain theory and the fact that the functions are continuous but on the fact that domains are sets and continuous functions are set-theoretic ones.
This is not to say that the set-theoretic interpretations are not worthwhile for metamathematical purposes; they are. When we wish to explore the functions computed by a program and set up theories of equivalence that abstract from the details of a program, functions are good. Domain theory is also very elegant. All this is accepted and endorsed. However, the set-theoretic interpretation does not represent the way CS treat programs and does not reflect the underlying ontology of CS; while modelling them as infinite sets may bring about some metamathematical insight, it cannot be definitional\textsuperscript{7}.

Furthermore, the set-theoretic ontology causes the problem of self-application. Consider again the requirement

\[
\text{State} = \text{Variables} \rightarrow [\text{Nat} \oplus [\text{State} \rightarrow \text{State}]]
\]

This is certainly a problem where the function space is treated set-theoretically, but, as we shall see, it is not one when the space is the space of algorithms or programs.

Completeness is equally problematic from a CS perspective. Consider the data type of natural numbers. A CS does not see this as an infinite set given in extension. For her, \textit{being a natural number}, is an operational notion characterised by its rules of generation: 0 is a number and for any number, its successor is a number. Much the same is true of numbers, lists, trees etc. For the CS, these are operational notions that are not to be characterised set-theoretically. CS are Aristotelian about infinity (in a manner not unlike the justification for the extendable tape of the Turing machine). They allow for the possibility that we can keep adding one, but not for the possibility that we can finish the process and thereby put everything into a completed totality.

If we are right, set theory cannot form the basis for PL ontology and, more generally, an ontology of computer science. As we have already said, we are not saying that the set-theoretic models are not worthwhile for metamathematical purposes. Indeed, the situation is not that dissimilar to that to be found in the possible world models of intuitionistic logic that employ sets and possible worlds. Such models are not for the constructive mathematician, but the classical one. They do not reflect what the constructive mathematician takes the logical connectives to mean. Constructive logicians and mathematicians appeal to notions such as \textit{operation} and \textit{constructive proof} as ways of explaining the basic logical connectives. The situation with PLs is similar: like constructivists CS have their own ontology.

5 Computational Ontology

So what does? In this section we shall suggest and sketch the beginnings of an alternative ontological framework. The idea is simple enough: we return to the

\textsuperscript{7}Some of these objections can be met by appealing to a different foundation namely category theory. Although these models are more flexible, it is still not that clear that they can furnish the ontology of the working computer scientists who as we shall claim shortly, has an ontology of their own. \cite{26} provides a fine summary of some of these alternatives with some insightful comments about their different roles and properties.
intuitions of computer science itself. CS build programs and systems from data types. So instead of sets we propose to turn matters around and use data types as our basic ontology. Once this has been sketched, we shall indicate how DS can be redone in terms of them—of course sacrificing the offending principles of extensionality and completeness in the process.

But to begin with we need to be clear about one aspect of this program: we cannot just use the CS notions of data type and program as we find them in PLs. There are two reasons for this. Firstly, our ontology must be theoretically secure. If it is to be a rival to domain theory, then it needs to be as rigorously formulated as the latter. More importantly, we need to know what our underlying ontology amounts to. For this we require a formal ontology. One way of unpacking this, and the standard way, is to formulate axiomatic theories of our data objects. This will be our approach. In the following we shall only be able to sketch the general idea but see [21], [22], [25] for more details.

First consider completeness and how this impinges on the development of a computationally acceptable ontological framework. We illustrate with the natural numbers. We shall not adopt the set-theoretic interpretation but rather a type-theoretic one governed by the following rules.

\[
\begin{align*}
\text{N}_0 & \quad \text{Nat type} \\
\text{N}_1 & \quad 0 : \text{Nat} \\
\text{N}_2 & \quad a : \text{Nat} \quad a^+ : \text{Nat}
\end{align*}
\]

i.e., we are proposing to replace the set of natural numbers with the type of natural numbers as it is used in computing science. This captures its role in type-checking. But in general, type inference is not enough to nail down a decent theory of numbers. We require other rules to determine its logical content [23], [25]. For example, we must have a principle of induction

\[
\begin{align*}
\phi[0] & \quad x : \text{Nat} \vdash \phi[x] \rightarrow \phi[x^+] \\
\hline
\phi[x] & \quad x : \text{Nat} \vdash \phi[x]
\end{align*}
\]

where \( \phi \) is a wff (well-formed formula) in our formal language. Without such a theory we would not be able to support recursion and fixed-points.

In short, we are advocating that \( \text{Nat} \) be taken as a primitive notion i.e., \( \text{Nat} \) is to be taken as a basic type and not interpreted in set-theoretic terms. In a similar way we might formulate an axiomatic theory of lists.

\[
\begin{align*}
\text{L}_0 & \quad T \text{ type} \\
& \quad \frac{}{\text{List}(T) \text{ type}} \\
\text{L}_1 & \quad T \text{ type} \\
& \quad \frac{\text{empty} : \text{List}(T)}{\text{List}(T)} \\
\text{L}_2 & \quad a : T \\
& \quad \frac{b : \text{List}(T)}{a \oplus b : \text{List}(T)} \\
\text{L}_3 & \quad \Gamma \vdash \phi[\text{empty}] \\
& \quad \frac{\Gamma, x : T, y : \text{List}(T) \vdash \phi[y] \rightarrow \phi[x \oplus y]}{\Gamma, x : \text{List}(T) \vdash \phi[x]}
\end{align*}
\]

The same remarks apply here about these rules being definitional for the type i.e., all our types are given axiomatically by their rules and, while they can be
modelled as sets, they are not primarily to be interpreted as such. They are to be taken as sui-generis.

For the second issue i.e., the extensional interpretation of programs as functions, we replace the set of functions from a set $A$ to a set $B$ with the type of operations from the type $A$ to the type $B$. These are determined by the following rules:

$$
\begin{align*}
T \text{ type} & \quad S \text{ type} \\
\overrightarrow{T} & \Rightarrow S \text{ type} \\
\lambda x : A \cdot t : \overrightarrow{T} & \Rightarrow S \\
f : T \Rightarrow S & \quad a : T \\
f a : S & \\
\end{align*}
$$

This is a simple axiomatisation of the typed lambda calculus [6] but we are not imposing the following rule of extensionality.

$$
\begin{align*}
f & : A \rightarrow B \\
g & : A \rightarrow B \\
x & : A \vdash fx = gx \\
\overrightarrow{f} & = \overrightarrow{g}
\end{align*}
$$

In a similar way, constructors such as Cartesian products and disjoint unions are also given a type-theoretic analysis.

Finally, the need for self application is solved in a fashion that CS would recognise immediately: we code our programs. So for example, we might add a type $D$ such that

$$
\begin{align*}
a & : D \\
\pi[a] & : D \rightarrow D \\
f & : D \rightarrow D \\
\overrightarrow{\text{Code}(f)} & : D \\
\pi(\overrightarrow{\text{Code}(f)}) & = \overrightarrow{f}
\end{align*}
$$

i.e. a type where its space of operations can be injected into it.

In summary, we are advocating an axiomatic account of data types are taken as sui-generis. However, we are not just letting in any old type theory. It must reflect the fact that it is a theory of data items\(^8\). Moreover, the types are not to be understood in set-theoretic terms but as new primitive notions. This is similar to their status in constructive mathematics [21], [22], [9] \(^9\). We claim that such accounts better capture the operational notions that CS operate with - i.e. such a rule based account is much closer to the intuitions of CS about data types.

## 6 Computational Semantics

We shall now illustrate how DS can be carried out within such a mathematical setting. We shall re-examine the semantics of SIL and its extensions, and

\(^8\)The acid test of any such theory is whether its basic notions are interpretable in Peano arithmetic (PA) where all the basic relations and functions are interpreted as $\Sigma$ relations of PA.

\(^9\)There are some obvious connections with constructivism but also some major differences. In particular, we are not advocating a change of logic. Indeed, we assume classical logic. Our interest is in the mathematics that directly reflects the CS intuitions.
indicate how this mathematical framework can be applied to provide a computationally acceptable DS i.e., how much of the distinctive flavour of DS can be preserved using only computationally acceptable notions. It is important to realise here that we are not promoting a semantics that is closer to an implementation such as Stack semantics[11]. Our semantics will be at the same abstract level only the underlying theory will be changed.

We set up the semantic functions just as before but where throughout we replace domains with types. In particular $\text{Nat}$ is interpreted as the type of natural numbers given by the above rules and the notions of disjoint union and function space in

$$Env = \text{Variables} \rightarrow \text{Loc}$$

$$State = \text{Loc} \rightarrow \text{Values}$$

$$Values = \text{Nat} \oplus \text{Proc}$$

$$\text{Proc} = \text{State} \rightarrow \text{State}$$

are replaced by their type-theoretic ones.

The problem of self application is solved, as we have indicated, by setting up types where the function spaces are mapped into the base type via coding. The classic case of this is the natural numbers i.e., the recursive model where

$$\pi \colon N \rightarrow (N \rightarrow N) \rightarrow N$$

where $\pi$ associates with every number the Turing machine with that Goedel number and $\text{code}$ just codes the Turing machine as a number. This interpretation is part of the CS ontological view of the world. In contrast, the domain theoretic one provides the set theorists perspective on such issues.

So the essence of DS remains intact. The semantic functions and the structure of the semantics including the structure of the relationships between the various entities remains. Moreover, we can read off the ontology of the language just as before. What has changed is the underlying ontological theory. We claim that it is much closer in spirit to the pre-theoretical ontology of computer science. Indeed, on the assumption that the ontology of a PL is determined by its semantics, and the best semantic theory is the one that best captures the pre-theoretical intuitions of the CS, we have the beginnings of an axiomatic account of the underlying CS ontology. This brings us full circle back to Quine.

References


