Preprint of a chapter in: Gordana Dodig-Crnkovic, Susan Stuart (eds.) *Computation, Information, Cognition—The Nexus and the Liminal.* Cambridge, UK: Cambridge Scholars Press, forthcoming.

Copyright notice. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

CHAPTER TEN

TOWARDS A PROGRAMMING LANGUAGE ONTOLOGY

RAYMOND TURNER AND AMNON H. EDEN

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.

Key terms: Philosophy of computer science
Related terms: Mathematical logic, type theory

1. Introduction

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. Such structures (Naur 1981; McCarthy 1981; Landin 1964) 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 Miranda™ have no imperative features and use recursion equations and inductive data types as their means of constructing programs. Imperative languages such as 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, each language forces one to solve problems within a given conceptual framework (its *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 (Smith 2004; Johansson 1989) to ontology.

2. Ontological Perspectives

Since PLs are formal languages, they have much in common with the formal languages used in logic, 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 languages. We begin with a famous one.

Quine (1961; 1969) 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. 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 classes of numbers; and Russell's simple type theory to classes, classes of classes and classes of classes of classes 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.

Unfortunately, as soon as we try to do so, we are presented with an obstacle: unlike logical languages, no programming language is explicitly distilled as a pure logical theory, not even PROLOG. We are rarely given anything like an axiomatization 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 axiomatization that displays its existential commitments ([1]). So this approach offers little direct help.

A closely allied alternative line of enquiry is offered by Frege (1952) and Dummett (1973; 1991). Instead of focusing on an axiomatic account of its constructs, it associates ontology with semantics. The ontological implications of a language are to be identified with the entities required to provide its constructs with a semantic interpretation. 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, with PLs, matters are less straightforward. 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. 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) (Stoy 1977; Strachey 1973; White 2004) 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.

[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 (1969) 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.

3. Set-Theoretic Semantics

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.

We illustrate matters with a simple imperative language (**SIL**). Although it is a toy language, it will suffice to illustrate some of the central aspects of DS and how the ontology of the language may be linked to it. **SIL** has three syntactic categories: Booleans, expressions and commands. We shall use *B* as the set of Boolean expressions, *E* as the set of expressions and *C* as the set of commands. The denotational semantics requires the following sets for the denotations of these syntactic categories.

$$\begin{aligned} Variables &= \{x_1, x_2, x_3, \ldots\} \\ Bool &= \{true, false\} \\ Nat &= \{0, 1, 2, \ldots\} \\ State &= Variables \rightarrow Nat \end{aligned}$$

where $A \rightarrow B$ 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.

$$\begin{aligned} \mathcal{B} &: B \rightarrow (State \rightarrow Bool) \\ \mathcal{E} &: E \rightarrow (State \rightarrow State) \\ \mathcal{C} &: C \rightarrow (Variable \rightarrow State) \end{aligned}$$

To illustrate the semantics we shall concentrate on the most interesting part i.e., the command language. This is generated by simple assignment statements, conditionals, sequencing and a while loop; it is given formally by the following BNF syntax.

$$c ::= x{:=}e \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid c \; ; \; c \mid \textbf{while } b \textbf{ do } c$$

where *e* is an expression and *b* a boolean expression. The semantic functions are defined by recursion on the syntax. We shall illustrate with the command language.

$$\begin{aligned} \mathcal{C}[x{:=}e]\,s &= Update[s,\, x,\, \mathcal{E}[e]\,s] \\ \mathcal{C}[\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\,s &= Cond(\mathcal{B}[b]\,s,\, \mathcal{C}[c_1]\,s,\, \mathcal{C}[c_2]\,s] \\ \mathcal{C}[c_1;c_2]\,s &= \mathcal{C}[c_1](\mathcal{C}[c_2]\,s) \\ \mathcal{C}[\textbf{while } b \textbf{ do } c]\,s &= \textbf{if } \mathcal{B}[b]\,s \\ &\qquad \textbf{then } \mathcal{C}[\textbf{while } b \textbf{ do } c](\mathcal{C}[c]\,s]) \textbf{ else } s \end{aligned}$$

The $Update$ function changes the state: it forces the value of x in the new state, to be equal to the value of e in the old one. $Cond$ chooses the first or second component according to the value of the Boolean; it is given the undefined value ($\perp$) if the latter is. Sequencing is interpreted as functional composition ($\circ$) while the **while** loop is unpacked in terms of the conditional whose important aspect is that it is interpreted as a recursive definition:

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

This is usually expressed by saying that the function $\mathcal{F}$ 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 (Stoy 1977), i.e., his theory of *Domains*[2]. In DS all the sets employed are taken to be domains[3]. Of

[2] These are partly ordered sets $\langle D,\sqsubseteq\rangle$ where D is a set and $\sqsubseteq$ is a partial ordering, with a least element $\perp$ such that any ω-chain of elements has a least upper bound.

[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

particular importance, is the domain of functions from one domain to another. In domain theory, the set $A \rightarrow B$ is not taken to be the set of all functions from A to B but just the set of *continuous* ones ([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 $Variables$, Nat, $Bool$ to be domains. These are taken to be flat domains (i.e. sets with a bottom element added). Finally, we take $State = Variables \rightarrow Nat$ to be the domain of continuous functions from the domain of variables to the domain of values. Of course, different programming languages require different domains i.e., they have different ontological requirements. Indeed, enrichments to the language such as the addition of declarations and procedures will effect the structure of domains considerably ([5]).

Domain theory provides an underlying mathematical system in which to articulate the DS semantics of PLs and is a candidate ontology for PLs. Firstly, at the level of individual PLs, the underlying ontological demands of each language is articulated in terms of the domains required and the relationships between them. Moreover, the theory of domains provides the general ontological setting: it provides the general mathematical framework in which DS is situated. This is an attractive picture: it is not only a mathematically elegant approach to semantics but an insightful and useful way of pinpointing the underlying ontological structures of PLs. We shall try to maintain the majority of these features.

---

philosophically significant since it is on the underlying set theory that we shall concentrate.

[4] i.e. the ones that both preserve the orderings in D and D' and preserve the least upper bounds of ω-chains.

[5] The set theoretic approach requires certain domains to contain copies of their own function spaces. Instead we adopt recursive models in the sense of recursive function theory.

4. A Computer Science Perspective

Observe that there are three underlying assumptions of DS: Compositionality, Extensionality and 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 set theoretic functions.

The third assumption, *Completeness* is more of a consequence of the fact that the semantics is set-theoretic than a separate principle. It demands that all the semantic items be 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, we have some reservations. 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. It seems that set-theoretic constructs do not reflect the intuitions of the computer scientist. More exactly, any philosophically worthy ontology should capture what the computer scientist talks about and employs about i.e., data structures, programs and algorithms. 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 such. For a start, we cannot compute with infinite functions given in extension. Moreover, properties of programs such as efficiency, flexibility and elegance

[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.

are, from the CS view of things, absolutely essential. But these properties are obliterated by DS. In particular, whatever the appropriate notion of equality for algorithms and programs, it should be 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. 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 appropriate. However, the set-theoretic interpretation does not represent the way computer scientists 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 ([7]).

Completeness is equally problematic from a CS perspective. Consider the data type of natural numbers. A computer scientists does not see this as an infinite set given in extension. For her, *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 for 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. 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 *operation* and *constructive proof* as ways of explaining the basic logical connectives. The situation with PLs is similar. While, for their own purposes, set theorists are free to interpret computational notions in their terms, it is clear that computer scientists have their own ontology.

[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. White (2004) provides a fine summary of some of these alternatives with some insightful comments about their different roles and properties.

5. Computational Ontology and Semantics

In this section we shall suggest and sketch the beginnings of an alternative ontological framework. The idea is simple enough: we return to the intuitions of computer science itself. 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 (Turner 1993; Turner 1996; Turner 2005) 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.

$$\mathbf{N}_0 \;\; Nat \; type \qquad\qquad \mathbf{N}_2 \;\; \frac{a : Nat}{a^+ : Nat}$$

$$\mathbf{N}_1 \;\; 0 : Nat$$

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 (Turner 2001; Turner 2005). For example, we must have a principle of induction

$$\frac{\phi[0] \quad x : Nat \;\vdash \phi[x] \rightarrow \phi[x^+]}{x : Nat \;\vdash \phi[x]}$$

where ϕ 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 Nat is taken as a primitive notion, i.e., Nat is a basic type and is not interpreted in set-theoretic terms. The same applies to

all types, namely, all our types are defined axiomatically by their rules and, while they can be modelled as sets, they need not be interpreted as such. In particular, we replace the set of functions from A to B with the type of *operations* from the type A to the type B (Hindley & Seldin 1986). In a similar way, constructors such as Cartesian products and disjoint unions are also given a type-theoretic analysis.

We shall now illustrate how DS can be carried out within such a mathematical setting. We shall re-examine the semantics of **SIL**, and indicate how this mathematical framework can be applied to provide a computationally acceptable DS. It is important to realise here that we are not promoting a semantics that is closer to an implementation such as Stack semantics (Milne & Strachey 1977). Our semantics will be at the same abstract level only the underlying theory will be changed.

For the interpretation of programs in **SIL**, we must explicitly deal with partiality; programs may not terminate and so may not produce a value. Consequently, $t{:}T$ is now to be understood as asserting that *t*, if it is defined, is of type *T*. With this in mind, we introduce a new type

$$State \rightsquigarrow State$$

i.e., the type of partial operations from states to states. This is determined by the following introduction and elimination rules.

$$\frac{x : Variables \quad v : Nat}{Update(x, v) : State \rightsquigarrow State}$$

$$\frac{f : State \rightsquigarrow State \quad g : State \rightsquigarrow State}{f \circ g : State \rightsquigarrow State}$$

$$\frac{b : Bool \quad f : State \rightsquigarrow State \quad g : State \rightsquigarrow State}{Cond(b, f, g) : State \rightsquigarrow State}$$

$$\frac{b : Bool \rightsquigarrow State \quad f : State \rightsquigarrow State}{While\ b\ do\ f : State \rightsquigarrow State}$$

$$\frac{f : State \rightsquigarrow State \quad a : State}{Apply(f, a) : State}$$

The notion of being defined/having a value (expressed as a down arrow) is determined by rules such as the following:

$$\frac{Apply(t, s) \downarrow}{t \downarrow} \qquad \frac{Apply(t, s) \downarrow}{s \downarrow}$$

Such rules enforce strictness ([8]). Finally, we add some rules that determine the equality criteria. We illustrate with the most important one i.e., the one for the while loop.

$$\frac{b : Bool \rightsquigarrow State \quad f : State \rightsquigarrow State \quad s : State}{(While\ b\ do\ f)s \approx Cond(bs, (While\ b\ do\ f)(fs), s)}$$

where $\approx$ denotes partial equality, i.e.

$$a \approx b \triangleq (a \downarrow \wedge\ b \downarrow) \rightarrow a = b$$

Informally, this asserts that, where both sides of the equality are defined, they are equal. The semantics of **SIL** can now proceed much as before but with these constructs replacing the set theoretic ones, e.g.,

$$\mathcal{C}[\textbf{while}\ b\ \textbf{do}\ c]s \approx\ While\ \mathcal{B}[b]s\ do\ \mathcal{C}[c]s$$

This has something in common with axiomatic semantics in the Hoare style but we have concentrated more on the logic of the programs themselves. Although very much a sketch of such a theory, it provides enough details for the reader to get the conceptual point being made: we are advocating an axiomatic account of data types where their status is similar to that of constructive mathematics (Turner 1993; Turner 1996; Martin-Löf 1975) ([9]). We claim that such accounts better capture the fundamental notions that computer scientists operate with. However, the essence of DS remains intact. The semantic functions and the structure of the semantics including the structure of the

[8] Other ways which impose some more operational information are also possible.

[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.

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 we have the beginnings of an axiomatic account of underlying CS ontology. As such, this brings us full circle back to Quine: the types of the logical language employed in the axiomatic underpinnings of the denotational semantics determine the underlying ontology of the language.

Acknowledgements

The authors wish to thank the Royal Academy of Engineering for supporting this research.

References

Dummett, Michael. 1973. *Frege's Philosophy of Language*. London: Duckworth.

—. 1991. The Logical Basis of Metaphysics. London: Duckworth.

Frege, Gottlob. 1952. "On sense and reference". In *Translations from the Philosophical Writings of Gottlob Frege*. Edited by Peter Geach and Max Black.

Hoare, Tony. 1969. "An axiomatic basis for computer programming." *Communications of the ACM* 12: 576–580.

Hindley, J. Roger, and Jonathan P. Seldin. 1986. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge: Cambridge University Press.

Johansson, Ingvar. 1989. Ontological Investigations: An Inquiry into Nature, Man and Society. New York and London: Routledge.

Landin, Peter J. 1964. "The mechanical evaluation of expressions." *Computer* 6:308–320.

Martin-Löf, Per. 1975. "An intuitionistic theory of types, predicative part". In *Logic Colloquium '73*. Edited by H. E. Rose and J. C. Shepherdson. Amsterdam: North Holland.

McCarthy, John. 1981. "History of Lisp". In *History of Programming Languages*. Edited by Richard L. Wexelblat. New York: Academic Press.

Milne, Robert E, and Christopher Strachey. 1977. *A Theory of Programming Language Semantics*. New York: Wiley.

Naur, Peter. 1981. "The European side of the last phase of the development of Algol." In *History of Programming Languages*. Edited by Richard L. Wexelblat. New York: Academic Press.

Quine, Willard Van Orman. 1961. "On what there is." In *From a Logical Point of View*. Cambridge: Harvard University Press.

—. 1969. “Speaking of objects.” In *Ontological Relativity and Other Essays*. New York: Columbia University Press.

Stoy, Joseph E. 1977. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Cambridge: MIT Press.

Smith, Barry. “Ontology”. In *The Blackwell Guide to the Philosophy of Computing and Information*. Edited by Luciano Floridi. Malden: Blackwell, 2004.

Strachey, Christopher. 1973. *The Varieties of Programming Language*. Oxford: Oxford Computer Lab.

Thomason, Richard H., ed. 1974. *Formal Philosophy: Selected papers of Richard Montague*. Yale: Yale University Press, .

Turner, Raymond. 1993. “Lazy theories of operations and types”. *J. Logic Computation* 3(1): 77–102.

—. 1996. “Weak theories of operations and types.” *J. Logic Computation* 6(1): 5–31.

—. 2001. “Type inference for set theory.” *Theoretical Computer Science* 266(1-2): 951–974.

—. 2005. “The foundations of specification.” *J. Logic Computation* 15(5): 623–663.

White, Graham. 2004. “The philosophy of computer languages.” In *The Blackwell Guide to the Philosophy of Computing & Information*. Edited by Luciano Floridi. Malden: Blackwell, 2004.