Minds & Machines (2007) 17:203–216
DOI 10.1007/s11023-007-9062-6

# Understanding Programming Languages

**Raymond Turner**

Published online: 8 June 2007


**Abstract** We document the influence on programming language semantics of the Platonism/formalism divide in the philosophy of mathematics.

**Keywords** Programming language semantics · Operational · Denotational

## Semantic Requirements

An implementation on a particular machine cannot serve as a definitional vehicle for a programming language. For one thing, there are too many extraneous details that pertain to the target machine i.e., it is far too idiosyncratic to serve as a definition of a machine independent language. Indeed, it is not clear that all such implementation details are graspable. Certainly, someone trying to learn a language from studying an implementation would find it hard to distinguish the wood from the trees. And here the wood is the machine independent language, and a semantics is intended to define it. However, it is not that the semantic account will not involve a machine; at some level that is precisely what it does, but it is a formalisation of the underlying abstract machine of the language, an idealisation, a mathematical abstraction that is intended to guide the construction of actual implementations and facilitate a notion of compiler correctness.

Of course, the requirements on a formal semantics go beyond their application to compiler construction. To understand the constructs of the language, a potential user requires a precise definition to guide and inform her. From the other side, a competent programmer has knowledge of the underlying machine and this knowledge underlies the construction of correct programs. Finally, the theoretician, who wishes to explore the semantic properties of the language, can only do so on the basis of a formal definition.

R. Turner (✉)
Department of Computer Science, University of Essex, Colchester, UK
e-mail: turnr@essex.ac.uk

This general picture is the one that generated the whole semantic enterprise and few question it. However, what should count as an acceptable semantic account, has been queried. There are a large number of approaches to semantics: e.g., operational (Landin 1964; Plotkin 1981), denotational (Scott and Strachey 1974; Stoy 1977; Milne and Strachey 1977; Winskel 1993) and category theoretic (Rydeheard and Burstall 1988) and even so-called axiomatic (Hoare 1969),[1] that reflect some aspects of this dispute. But the differences between them do not all raise significant philosophical questions. Allegedly, the most important ones concern operational and denotational semantics. But are there any significant philosophical differences between these? To address this, we need to briefly say what they are.

## Operational Semantics

In the early sixties, Landin (1964) provided an account of programming language semantics in which programming languages were given their semantic content via an interpretation into the Lambda Calculus, (Church 1941) together with an abstract machine (the SECD machine) that provided the means of evaluating Lambda expressions. This formed the basis for the purest version of operational semantics (OS, Landin 1966; Reynolds et al. 1972).

The syntax of the calculus is minimal. The language supports abstraction, a form of implicit function definition, and application, a form of function call. Using abstraction and application, Landin demonstrated how programming languages could be systematically interpreted in the calculus. At the core of this interpretation is the modelling of procedure definition and procedure call as abstraction and application. Furthermore, by syntactically sugaring the calculus, all the constructs of programming languages can be interpreted. In particular, the calculus is known to support a means of unpacking recursive definitions via its *fix-point* operator.

In addition to the syntax, the calculus comes equipped with an associated set of reduction rules that provide the means of computation in the calculus i.e., computation is expression reduction/evaluation, and this is used to model the evaluation of expressions in programming languages. Furthermore, the SECD machine provides an abstract machine implementation of reduction.[2] Fortunately, the system of reduction rules satisfies the Church-Rosser property i.e., where termination is reached the order of evaluation is irrelevant. There is also an alternative axiomatic account of the calculus that provides rules of equality rather than reduction rules (Hindley and Seldin 1986). The two accounts are equivalent in the sense that this notion of equality is equivalent to the one determined by reduction. However, different languages enforce different reduction strategies. These can be built into the semantic description in various ways (see Plotkin 1975).

More contemporary approaches, pioneered by Gordon Plotkin in (Plotkin 1981), provide accounts in terms of state transition systems. Even natural deduction

[1] (White 2004) provides a fine survey.

[2] Note that any such evaluation mechanism has to be measured against the axioms/reduction rules. These have semantic priority.

formulations (Burstall and Honsell 1988) have been advocated. Still others bring types and the typed Lambda calculus into the picture (Barendregt 1992).

But whatever version of OS is adopted, the conceptually significant aspect is that a programming language is interpreted in another language (e.g. the Lambda Calculus) that may have some associated theory. The latter might be given as rules for an abstract machine, state transitions or even rules for equality and termination. But the semantics is still taken to be translational i.e., one language is interpreted in another.

## Denotational Semantics

However, from a more traditional semantic perspective, the operational approach is taken to be unsatisfactory. According to this tradition, and one that is supported by a large fraction of the logical and theoretical computer science communities, to semantically specify a formal language, at some point in the interpretation process, a mathematical semantics is required. While the difference between the two versions of semantics is not put in philosophical terms, it is still taken that OS is essentially translational, whereas mathematical approaches aim at the abstract mathematical world.

> The motivation for the Scott-Strachey approach to semantics stems from a recognition of the fact that programs and the objects they manipulate are in some sense concrete realisations or implementations of abstract mathematical objects.(McGettrick 1980).

Apparently, what distinguishes *real* semantics is that it is mathematical i.e., programming languages refer to (or are notations for) abstract mathematical objects. In particular, semantics has almost become synonymous with set-theoretic semantics, and one way of criticising a formalism in computer science is to point to its lack of a such a semantics. In (Stoy 1977), this criticism is levelled at Landin's operational approach i.e., the Lambda Calculus could not be employed as a semantic medium because it itself lacked a set-theoretic semantics.[3]

> We can apparently get quite a long way expounding the properties of a language with purely syntactic rules and transformations.....One such language is the Lambda Calculus and, as we shall see, it can be presented solely as a formal system with syntactic conversion rules.....But we must remember that when working like this all we are doing is manipulating symbols-we have no idea at all of what we are talking about. To solve any real problem, we must give some semantic interpretation. We must say, for example, ''these symbols represent the integers''. (Stoy 1977)

Such an objection is a fairly standard one: replacing one language with another (the *meta-language*) does not yield a semantic account. Apparently, it does not get at

[3] The unyped nature of the calculus made it hard to see how a set-theoretic model could be found in ZF. This was Scott's main objection (Scott 1970).

*what we are talking about*. Providing a further meta-meta-language to interpret the first meta-language does not help; this will still be in need of interpretation. Indeed, this seems only to succeed in generating a potentially infinite sequence of interpreting languages. Nor does the presence of associated rules or axioms move us far forward: uninterpreted axiom systems are still taken to be in need of a semantic justification; the rules themselves need to be interpreted, and any such interpretation relies on a further interpretation that relies on more rules[4] etc.

In contrast, the language of set-theory is taken to refer beyond syntax to the abstract world of sets, an objective world, independent of language and our knowledge of it. Hence, it is taken to block any such regress of languages. This is the implicit picture that drives the striving for set-theoretic semantics.

But for this perspective to have any real force, sets must somehow be distinguished from formal systems such as the Lambda Calculus. The standard way in which this is done involves a platonist view of set-theory-and consequently a platonist view of programming languages. Indeed, the other pioneer of *denotational semantics* (DS), Christopher Strachey, seemingly held a platonist position about the underlying ontology of programming languages (Strachey 1966; Strachey 1967). Although he was not explicit about the mathematical content of this ontology, his subsequent collaboration with Scott set the scene for the development of DS (Scott and Strachey 1974; Stoy 1977; Milne and Strachey 1977; Winskel 1993). Here, every data-structure and control feature is taken to denote a set of some kind: programs denote set-theoretic functions and data types are modelled as infinite sets.

The original objection to the Lambda Calculus as an acceptable vehicle for semantics now becomes somewhat more specific. At the time, its lack of a set-theoretic semantics meant that there was no way out of the layers of languages. Neither the axioms of the Lambda Calculus nor the provision of an evaluation machine, would suffice. However, shortly after this objection was voiced, Scott (1970, 1972) found a set theoretic model for the Lambda Calculus. It consisted of two isomorphic sets $D$ and $D \rightarrow D$, a class[5] of functions from $D$ to itself that is closed under the functions definable by the Lambda language. In this mathematical setting it is possible to interpret all the lambda terms in such a way that provably equal Lambda terms denote the same object in the model.

Presumably, with this original objection to operational semantics lifted, one could adopt Landin's approach in the sure knowledge that everything was securely based on set-theory. But things did not quite proceed in that way. Instead, programming languages were semantically unpacked directly in set-theoretic terms with the Lambda Calculus playing only a housekeeping role. However, the difference between this direct approach and the indirect one via the Lambda Calculus, is not philosophically significant. The important philosophical detail is the set-theoretic basis of the resulting semantics: ultimately, every aspect of a programming language is interpreted as a set.

[4] We shall say more about this later when we discuss formalism a bit more carefully.

[5] The sets were complete lattices and the function space the space of continuous functions. Since then there have been many variations on this theme.

## Set Theory and Platonism

To what extent can set-theoretic semantics fulfill our original list of semantic requirements for programming languages? For example, can it really serve as a guide to the implementer and user? Could it be used to teach someone the language? How well does it capture what the user of the language knows about the underlying abstract machine? Part of the answer to these questions depends upon her knowledge of set-theory: for someone with no such knowledge it would be useless; they are not going to make much progress unless they know enough set-theory to follow the semantic definitions of the constructs. Unfortunately, very few users or implementers do. And this is a rather significant constraint upon the use of set-theory as a semantic medium. However, on the face of it, this is less a philosophical issue than a matter of education.

To draw out the central philosophical concern, assume that our user, an expert set-theorist, is given the DS of a programming language L and is asked to write a compiler for a given machine. How does the set-theoretic knowledge guide her? More generally, what exactly does she know as result of grasping the set-theoretic semantics? Clearly, this depends upon what this set-theoretic knowledge amounts to: what is the nature of the knowledge provided by the semantics? If there is any substance to the above metaphysical claims about set-theory, it must go beyond the syntax of the language of set-theory; indeed, it must go beyond its axioms. Otherwise, we will have not reached the abstract universe of sets. Gödel gives us some insight into what is involved.

> Despite their remoteness from sense experience, we do have something like a perception also of the objects of set-theory, as is seen from the fact that the axioms force themselves upon us as being true. I don't see any reason why we should have less confidence in this kind of perception i.e., in mathematical intuition than in sense perception, which induces us to build up physical theories and to expect that future sense perceptions will agree with them, and moreover, to believe that a question not decidable now has meaning and may be decided in the future (Gödel 198?).

Gödel draws an analogy with the perception of physical objects; sets are *perceived* in an analogous way but what is *perceived* is neither the axioms and rules, nor the expressions that generate sets, but the sets themselves.[6]

> It should be noted that mathematical intuition need not be conceived of as a faculty giving an immediate knowledge of the objects concerned. Rather it seems that, as in the case of physical experience, we form our ideas also of those objects on the basis of something else which is immediately given. Only this something else here is not, or not primarily, the sensations. That something besides the sensations actually is immediately given follows (independently of mathematics) from the fact that even our ideas referring to physical objects contain constituents qualitatively different from sensations or

[6] It would seem that such knowledge must be taken as *knowledge of* sets rather than knowledge that some proposition about sets is true. At least this is the interpretation that leads to Platonism.

> mere combinations of sensations, e.g., the idea of object itself, whereas, on the other hand, by our thinking we cannot create any qualitatively new elements, but only reproduce and combine those that are given. Evidently the ''given'' underlying mathematics is closely related to the abstract elements contained in our empirical ideas. It by no means follows, however, that the data of this second kind, because they cannot be associated with actions of certain things upon our sense organs, are something purely subjective, as Kant asserted. Rather they, too, may represent an aspect of objective reality, but, as opposed to the sensations, their presence in us may be due to another kind of relationship between ourselves and reality.(Gödel 1983).

While this is hard to unpack, he does seem to be suggesting that we may only be able to explain our knowledge of sets in terms of another relation to reality than that of a perceiving subject.

Maddy (1990) defends a version of something like Gödel's position in which she attempts to provide an account of a notion of *perception* for sets. However, ultimately, she rejects it in favour of another Gödelian theme, i.e., a form of mathematical naturalism (Maddy 1997) in which the power and consequences of set-theoretic axioms (e.g. their consequences in main stream mathematics) determine their truth.

But whatever the merits of Gödel's perspective, and we shall come to this shortly, there is already a puzzle about the difference between DS and OS. Why should the Lambda Calculus and set-theory be metaphysically different? Why is set-theory taken to stop the regress of languages but the Lambda Calculus taken not to do so? After all, they are both formal mathematical theories. Even under a platonic interpretation, it seems hard to see how the difference could be made out. It is certainly not clear that Gödel would have supported such a distinction. Referring to Turing's analysis of mechanical computability, Gödel writes:

> [Turing] has shown that the computable functions defined in this way are exactly those for which you can construct a machine with a finite number of parts which will do the following thing. If you write down any number $n_1,..,n_r$ on a slip of paper and put the slip into the machine and turn the crank, then after a finite number of turns the machine will stop and the value of the function for the argument $n_1,..,n_r$ will be printed on the paper (Gödel 193?).

In his Gibbs Lecture he writes:

> The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such definition, which, however, all lead to the same concept. The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing (Gödel 1951/1995).

During this period, Gödel thought that Turing's account of the notion of finite procedure was definitive. Consequently, one assumes that he would have assigned

the notion of *finite procedure* a similar metaphysical status to sets. In Wang's words, Gödel saw the problem of defining computability as:

> an excellent example of a concept which did not appear sharp to us but has become so as a result of a careful reflection (Wang 1974).

Furthermore, the approach to computability given by the Lambda Calculus is extensionally the same as that provided by Turing machines. Indeed, a properly formulated version of the (Partial) Lambda Calculus has a direct recursive model i.e., a model in Turing machines. Consequently, although they are quite different mathematically, it is hard to see how there could be any difference in metaphysical status between set-theoretic and recursive models of the Lambda Calculus i.e., if platonism about mathematical objects is correct, it is hard to see a metaphysical difference between DS and OS.

## The Epistemological Challenge

However, there is a serious epistemological problem with any form of mathematical platonism, including that of Gödel. At the heart of this is Benacerraf's *epistemological challenge* (Benacerraf 1996). In its original form, this argument has two premises.

A. The abstract nature of mathematical entities

Mathematical entities are mind and language independent. They bear no spatio-temporal relations to us. They undergo no physical interactions with us or anything we can observe.

B. The causal theory of knowledge

This adds a new clause to the traditional view of knowledge as justified true belief: to count as knowledge, what makes the belief true must be appropriately causally responsible for that belief. This additional clause was taken to be necessary to overcome Gettier's objection to the traditional view.

If A represents the true nature of mathematical entities, we have little chance of having any causal connections with them. Is it even clear what it means to causally interact with such causally inert entities? Consequently, on this account of mathematical knowledge, it seems very hard to see how we could ever come to possess any.

If this argument is sound, then it raises a major problem for DS as a semantic medium. To see why, suppose we are given the set-theoretic semantics of a programming language L. We shall use term *sense of a program* of L to mean what someone who understands L grasps about its programs (see Dummett 1975, 1976; Miller 1998). Suppose further that the programs of L have set-theoretic objects (i.e., functions) as their senses. So that someone who understands the programs of L has knowledge of the sets that form their senses. It follows that if the causal theory of knowledge is correct, then there is no way in which a programmer can grasp the senses of the programs of L. One or other of our two premises has to be given up.

However, the debate has moved on somewhat.[7] In particular, the causal theory of knowledge no longer attracts the same level of support and other extensions/modifications to the traditional view have been considered. One such is *Reliabilism* which may be summarised as follows.

C. One knows some proposition just in case it is true, one believes that it is true, and one has arrived at this belief through some reliable process.

So what are the consequences of replacing B with C? Although not as explicit as the causal theory about how knowledge is acquired, this still leaves a substantial challenge for the platonist: what is the reliable process that enables us to gain knowledge of abstract objects?

Indeed, as Maddy (1990) suggests, this seems to indicate that the exact form of the theory of knowledge is not the critical issue. Rather, it is the more general concern of how we obtain knowledge of such abstract objects. We can perhaps capture this by employing a principle enunciated by Dummett (1975, 1976) in his arguments for antirealism in theories of meaning.

D. If a piece of knowledge is ascribed to a person, then it must be in principle possible for that person to have acquired that knowledge.

D gets at a generalisation that applies to all theories of knowledge. *A*,*D* is a general form of the epistemological challenge. While it does not say how knowledge of abstract sets is to be obtained, it does demand that we provide some account of how, in principle, that knowledge may be obtained and, given that our objects are abstract objects, it is a significant challenge.

Field puts matters slightly differently but with much the same final request.

> The relevant facts about how the platonist conceives of mathematical objects include their mind-independence and language independence; the fact that they bear no spatio-temporal relations to us; the fact that they do not undergo any physical inactions (exchanges of energy momentum and the like) with us or anything we can observe; etc.....I refrain from making any sweeping statements about the impossibility of the required explanation. However, I am not optimistic about the prospects of supplying it. (Field 1982).

So, while this does not have the force of the original Benacerraf challenge, it still raises a substantial explanatory task for set-theoretic semantics, indeed, for any form of abstract mathematical semantics.

## The Manifestation Challenge

A second problem for platonism is posed by a modification of an anti-realist argument employed in the philosophy of language (Miller 1998). The argument relies for its credibility on Wittgenstein's view that understanding consists of some practical ability i.e.,

[7] Maddy (1990) clearly summarises, it up until 1990.

W. If speakers possess a piece of linguistic knowledge, then that knowledge should be manifested in speakers' use of the language.

If the programs of L have set-theoretic functions as their senses, and speakers possess this knowledge, then knowledge of these senses should be manifested in the use of L. But, how could such knowledge be manifested in the use of L? In particular, it is hard to see how a difference between knowledge of the axioms of ZF and knowledge of the actual sets, could be so manifested.

In answer, platonists argue that set-theorists, without reference to any axiomatic system, make correct judgements that constitute knowledge of sets i.e., they are correct in the sense that they are subsequently proven from the ZF axioms. And, this is often cited as something that is in need of explanation (Maddy 1990, 1997) and platonism is taken to be part of the answer: if platonism were false, how could judgements made by set-theorists be so reliable. But again, how is it possible to demonstrate, in terms of their mathematical practice, that set-theorists make such judgements without reference to and use of any axiom system?

Presumably, one might try to force a wedge between knowledge of the axioms and knowledge of the sets, by reference to the standard concept of set that is encoded in concepts such as the *limitation of size* principle and *cumulative hierarchy*. Indeed, these are often used to provide external justification for the axioms of ZF. In particular, the cumulative hierarchy is sometimes taken to be a picture of the abstract world of sets whereas the axioms of ZF only constitute a formal system. But is this difference sustainable? It cannot simply be that one is a set of axioms and the other is a more abstract structure. After all, the cumulative hierarchy can be put on an axiomatic footing via the direct axiomatisation obtained by using Dana Scott's theory of sets and stages (Scott 1970; Potter 2004). Indeed, in the latter, the concept of the cumulative hierarchy is made precise in a *faithful* way (in the sense of (Feferman 1986)). This makes a justification of the axioms more formally exact. Whatever the independent platonic realm of sets is like, it really does seem difficult to imagine how knowledge about it could ever be manifested in a way that is not matched by some axiomatic account. This is yet another difficult challenge for the platonist.

While not completely compelling, these arguments throw up substantial challenges to the platonist and to the possibility of a platonist account of programming language semantics. At very least they suggest that it would be prudent to look for alternatives.

## Game Formalism

One alternative approach advocates an account of set-theory in which we locate our understanding of sets, not in the sets themselves, but in the axioms and rules of set-theory. This would be an operational view of set-theory. Our understanding of the programming language would then be unpacked in terms of these rules i.e., via the semantic clauses through the axioms and theorems of set-theory, including the proof of any required theorems. Such understanding is constituted by the rules and axioms

of the underlying semantic theory: when we teach a language, we teach the semantic rules etc.

However, by removing the metaphysical and ontological status of set-theory, and replacing it with this axiomatic perspective, we have removed much of the attraction of a semantic account based on set-theory: its semantic prowess, as more than just an axiomatic theory, is lost. But once this is acknowledged, philosophically, there is little to choose between a metaphysically stripped version of DS and Landin's original account. Indeed, once this step is taken, it is even possible to completely dispense with any intermediate primitive theory (e.g. sets or finite procedures) and axiomatise programming languages directly. We can even mimic the denotational semantics in an axiomatic way. But once again, the particular choice of target theory (set or recursive) or whether the semantics is direct or indirect, does not impinge upon the central philosophical issues. Philosophically, all that matters is that we have come full circle and returned to a variant of OS in which meaning and understanding are located in sets of rules. However, we have made some progress in that we can now situate it in its general philosophical setting.

Underlying this view of meaning and understanding is a form of *game formalism*. On this view, understanding both games and mathematics depends only on understanding their rules. It is not germane to ask about the ontology of either: if mathematical languages refer to anything, it is extraneous to mathematics itself. More generally, according to game formalism, mathematical theorems have no truth value in the sense that there is no reference to entities outside of the formal system.

To many this is counter intuitive and leads to difficulties in explaining why mathematics can be used to explain and model the world i.e., why it is so applicable. For example, if mathematical entities or systems are invented games, why are they so useful in the natural sciences? How can they be used to make true assertions about the physical world? More significantly for PL semantics, we still do not have any decisive answer to the original worries about such an approach to semantics: no matter how many object/meta levels of language we incorporate, we never escape from syntax to meanings. Even sets of rules associated with such languages do not address the problem since the rules themselves need to be interpreted-presumably by an interpreter that relies on further rules for its interpretation. Apparently, platonism offers us an escape from this spiral of languages/systems of rules; unfortunately, it has some rather unpleasant epistemological consequences.

We appear to have reached one of the traditional walls in the philosophy of mathematics: the platonist/formalist divide.[8] However, there seems to be a difference between the working attitudes of computer scientists and mathematicians. It has been said (Hersh 1997) that mathematicians are platonists during the week and formalist at weekends. They think about abstract structures in a platonic way during the week, but get cold feet at weekends and resort to a form of formalism. In contrast, the working computer scientist is a formalist in her working week. She manipulates logical and computational systems, write compilers and programs, uses proof systems to prove properties of programs and systems, designs

[8] There are several other anti-realist options but we have no space to consider their impact on the current issue. One of some significance is provided by Wittgenstein's account of concept formation.

databases etc. All of these activities are heavily syntax directed. However, when she is challenged about the content and meaning of the formalism employed, she defers to set-theory. She recoils from this syntactic world claiming that their real meaning is given in the universe of sets i.e., computer scientists seem to be formalists during the week and platonists at weekends. Of course, this caricature does not move us forward in the aforementioned debate. Nor can we hope to resolve it here. After all this is a hard traditional problem in the philosophy of mathematics that, despite many years of discussion, does not have a settled solution.

### Wittgenstein on Rules

However, an entirely different way forward is offered by Wittgenstein in his account of rule-following (Wittgenstein 1939, 1953, 1978). His formulation of the problem gets to the heart of the general issues of meaning, understanding and using a language. Clearly, space does not allow us to do justice to Wittgenstein's ideas on rule following and nor to the controversies and criticisms surrounding it. Nor can we develop anything like an adequate approach to the meaning of programming languages based on his rule-following considerations. We can only sketch a possible way forward.

Our starting point is the original objection to operational semantics: the layers of language problem. Imagine that a student is instructed to evaluate the simple program

$$
\begin{aligned}
&x = 5;\\
&y = 0;\\
&\textbf{while } x > 0 \textbf{ do } y := y + 3; x := x - 2.
\end{aligned}
$$

But suppose that, instead of the series of values for the variables $x$ and $y$,

$$(5, 0), (3, 3), (1, 6)$$

the student returns the series

$$(5, 0), (3, 3), (1, 5).$$

Further suppose that the student, upon being corrected, says that throughout her evaluation, she just *went on the same way* i.e., she followed the rule as she sees it always in the same way. How might we reply? Well, we might appeal to an interpreter/compiler for the language in order to justify the claim that a mistake has been made. But this just pushes the problem back one layer i.e., into understanding another language i.e., the language in which the interpreter/compiler has been written. But how are we to pin this down? By another set of rules? Do we ever get to a set of unambiguous rules? Wittgenstein puts matters as follows.

> ... every course of action can be made out to accord with the rule. But if everything can be made out to accord with the rule, then it can also be made out to conflict with it. And so there would be neither accord nor conflict. (Wittgenstein 1953)

It is at this point that the platonist appeals to set-theory as the arbitrator. However, rather than seeing platonism as a way out of the rule-following considerations, Wittgenstein is intent on liberating us from the bewitchment of it. His reflections on rule-following are used to argue against a referential conception of meaning i.e., one that associates the meaning of an expression with the knowledge of the criteria for identifying its referent. The rule following considerations actually argue against there being any such platonic referents. For example, in the case of our program P, at any given moment, the meaning attributed to P is not able to impose any logical constraint on what must be taken as the result of its correct application, and thus cannot intensionally fix any specific entity. Indeed, his considerations are aimed at eliminating the need to posit any such entity; there is nothing required beyond the actual application of the rule/program. In particular, one favoured interpretation of the rule-following considerations argues in favour of a *community* interpretation i.e., it is an empirical fact that all those people who have received a certain type of training, or more generally, all those who share a certain *form of life*, agree in the practice of conferring meaning to the expressions of their language (Pasquale 1994).[9]

This view of the relationship between mathematics and meaning is intended to undermine the view of meaning propagated by platonism and to save a hidden truth of formalism.

> Is it already mathematical alchemy, that mathematical propositions are regarded as statements about mathematical objects, and mathematics as the exploration of those objects. In a certain sense it is not possible to appeal to the meaning of signs in mathematics, just because it is only mathematics that gives their meaning.(Wittgenstein 1978).

This throws some light on the semantic issues surrounding programming languages. Any formally adequate mathematical theory can serve as the semantic vehicle for defining programming languages. What programming languages are taken to mean will be informed, indeed constituted, by the *form of life* that is the practice of the chosen mathematical theory-be it the Lambda Calculus, set-theory or something else.

## Some Conclusions

While we have not reached any final conclusions about the nature of semantics, we located the different approaches to the semantics of programming languages in a general philosophical setting. We have demonstrated that the semantic problem of programming languages provides a new twist to the platonist/formalist/conventionalist debate. In particular, it seems clear that the formalist/platonist perspectives do

[9] A controversial but interesting reading of the Wittgenstein's account of *rule-following* is Kripke's. Apparently, Wittgenstein is voicing a skeptical paradox and offering a skeptical solution. This interpretation drives one to conclude that there is *no fact of the matter* as to the correct application of a rule. However, most contemporary interpretations do not see Wittgenstein as arguing for such a skeptical position.

not line up with the operational/denotational divide in programming language semantics. One could hold either philosophical position with respect to any of the so-called operational or denotational accounts. Much the same is true of the more conventionalist stance of Wittgenstein: any form of semantics, be it based on the Lambda calculus or set-theory, constitutes a *form of life* that is constituted by the appropriate training.

Indeed, the difference between the various approaches to semantics is technical not conceptual or philosophical. And even here, the exact technical differences are not easy to articulate.

## References


Barendregt, H. P. (1992). Lambda Calculus With Types. Handbook of Logic in Computer Science. In S. Abramsky, D. M. Gabbay, & T. S. E. Maibaum (Eds.), Oxford Science Publications (pp. 118–310).

Benacerraf, P. (1996). What mathematical truth could not be -I.1973. Reprinted in Benacerraf and His Critics, Morton, Adam (Ed.), Blackwell.

Burstall, R., & Honsell, F. (1988). *A natural deduction treatment of operational semantics. In Proceedings of the 8th Conf. on Foundations of Software Technology and Theoretical Computer Science, volume LNCS, Vol. 338* (pp. 250–269). NewYork: Springer-Verlag.

Church, A. (1941). The calculi of lambda conversion. Princeton: Princeton University Press.

Dummett, M. (1975) What is a theory of meaning ? In mind and language. Oxford: Oxford University Press.

Dummett, M. (1976). What is a theory of meaning II. In J. McDowell, & G. Evans (Eds.), *Truth and meaning: Essays in semantics* (pp. vii-xxiii). Oxford: Clarendon Press.

Feferman et al (Eds.) (1986, 1990, 1995), Gödel. Collected works, Vols. I–III. Oxford: Oxford University Press.

Feferman, S. (1979). Constructive theories of functions and classes. In: M. Boffa, & D. van Dalen, K. Mc Aloon (Eds.), *Logic Colloquium '78* (pp. 159–225). Amsterdam: North Holland.

Field, H. (1982). *Realism, mathematics, and modality*. Oxford: Basil Blackwell.

Gödel, K. (1931) Undecidable diophantine propositions. In Collected Works III:164–175.

Gödel, K. (1951/1995). Some basic theorems on the foundations of mathematics and their implications. In *Collected works III* (pp. 304–323). Oxford: Oxford University Press.

Gödel K. (1983) What is cantor's continuum problem? reprinted in Benacerraf and Putnam's collection philosophy of mathematics (2nd ed.). Cambridge University Press.

Hersh, R. (1997). What is mathematics, really? London: Vintage.

Hindley, J. R., & Seldin, J. P. (1986). *Introduction to combinators and the λ-calculus*. London: London Mathematical Society Texts.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM, 12*(10), 576585, October.

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal, 6*, 308–320.

Landin, P. J. (1966). The next 700 programming languages Landin PJ. *Communications of the ACM, 9*(3), 157–403.

Miller, A. (1998). Philosophy of language (London: University College London Press/Routledge, Fundamentals of Philosophy Series, xviii + 348 pp).

Maddy, P. (1990). Realism in mathematics. Oxford: Oxford University Press.

Maddy, P. (1997). Naturalism in mathematics. Oxford: Oxford University Press.

McGettrick, A. D. (1980). *The Definition of Programming Languages*. New York: Cambridge University Press, NY, USA, ISBN:0521226317.

Milne, R., & Strachey, C. A. (1977). *Theory of programming language semantics*. New York, NY, USA: Halsted Press, ISBN:0470989068.

Moore, G. H. (1982). Zermelo's axiom of choice: Its origins, development, and influence. Berlin: Springer-Verlag.

Pasquale, F. (1994). Wittgenstein's philosophy of mathematics. London: Routledge.

Plotkin, G. D., (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September.
Plotkin, G. D. (1975). Call-by-name, call-by-value, and the $\lambda$-calculus. *Theoretical Computer Science, 1*, 125–159.
Potter, M. (2004). *Set Theory and Its Philosophy: a Critical Introduction*. Oxford: Oxford University Press. ISBN 0199270414.
Reynolds, J. (1998). Definitional interpreters for higher-order programming languages. Higher-order and symbolic computation, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
Rydeheard, D. E., & Burstall, R. M. (1988). Computational category theory. Prentice Hall: New York.
Scott, D. S. (1970). *Outline of a mathematical theory of computation. Technical Monograph PRG-2*. Oxford: Oxford University Computing Laboratory, England, November.
Scott, D. S. (1970). OWHY. Unpublished manuscript.
Scott, D. S. (1972). Continuous lattices. In F. W. Lawvere, (Eds.), Toposes, algebraic geometry, and logic, number 274 in Lecture notes in mathematics (pp. 97–136), Springer-Verlag.
Scott, D. S. (1974). Axiomatizing set-theory. In Jech, J. Thomas, (Ed.), Axiomatic Set Theory II, Proceedings of Symposia in Pure Mathematics 13. American Mathematical Society: 20714
Shapiro, S. (1997). Philosophy of mathematics: Structure and ontology (p. xii + 279).Oxford: Oxford University Press.
Stoy, J. E. (1977). Denotational semantics: The Scott-Strachey approach to programming language theory. Cambridge, MA: MIT Press.
Strachey, C. (1967). *Fundamental concepts in programming languages Strachey*. Oxford: C. Oxford University Press, Oxford.
Strachey, C. (1966). Towards a formal semantics. In Formal Language Description Languages for Computer Programming, North Holland, pp. 198–220.
Scott, D. S., & Strachey, P. (1974). The varieties of programming languages. Oxford Computer Lab
Scott, D. S., & Strachey, C. (1971). Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab.
White, G. (2004). The philosophy of computer languages. In L. Floridi (Ed.), *The blackwell guide to the philosophy of computing & information*. Oxford: Blackwell.
Wang, H. (1974). *From mathematics to philosophy*. London: Routledge & Kegan Paul
Wittgenstein, L. (1939). Wittgenstein's lectures on the foundations of mathematics, Cora Diamond (Ed.), Cambridge
Wittgenstein, L. (1978). Remarks on the Foundation of Mathematics, (3rd ed.). In G. H. von Wright, R. Rhees, & G. E. M. Anscombe, trans. G. E. M. Anscombe. Basil Blackwell, Oxford.
Wittgenstein, L. (1953). *Philosophical Investigations*. Oxford: Basil Blackwell
Winskel, G. (1993). *Formal semantics of programming languages*. Cambridge: MIT press