Chapter 18

The Philosophy of Computer Languages

Graham White

1 Introduction: Two Semantic Projects

Consider a theory whose aim is to say, for a given language, what each of its expressions means. Call it a semantics of that language. What might be required of such a theory before it was allowed that it accomplished its aim or did so in an optimal way? (Travis 1986)

This is a question which belongs to a well-established philosophical program: the Davidson–Dummett program of using a formal semantics – a "theory of meaning," as these philosophers call it – in order to attack philosophical questions about the relation between language and reality, or between mind and language (see, for example, Wiggins 1997).

There is also, among theoretical computer scientists, a similar area of study: it is usually called "the semantics of computer languages," or often simply "semantics." Its aim is, likewise, to develop a formal account of the meaning of a given computer language, and to use that account to answer interesting questions.

These questions might be severely practical – one might, for example, want to formally verify, using such a semantics, that the language in question did what it was claimed to do. However, one might also want to answer more general questions: one might want to design a computer language, and an intended semantics for that language is often a good place to start. Or one might want to classify existing computer languages: it is clear that some of them are more similar to each other than to others, and that some languages are merely notational variants of each other, but how do we put such observations on a more formal footing? And, finally, we might be tempted to say something about the nature of computation on the basis of the semantics of the languages in which we express computations.

I would claim that the semantics of computer language has considerable philosophical interest: it has a different motivation to the usual philosophical approach to computation via Turing machines, and, correspondingly, it yields different insights. One of the main difference is that the semantics of programming languages is concerned with the languages in which people actually program, and, particularly, with the languages which have been found to be good to program in; it is thus inescapably connected with the practice of programming. It is emphatically not a discipline which is developed out of some *a priori* notion of computation. And, in fact, it has connections with areas of philosophy which may seem surprising; Quine's concept of referential transparency, for example, is an important part of the programming-language enterprise. Some of this material is quite technical, and is also generally unfamiliar to philosophers (even to those who know the usual technical repertoire of philosophical logic). I have segregated many of the more technical details into sections of their own, which can, at first reading, be omitted.

2 History

The semantics of programming languages grew up in a particular historical context, and it is worth spending some time describing it: it was developed by a group of philosophically literate mathematicians and computer scientists, and the philosophical influences are quite evident. They are also little known: the history of computing has tended to focus very much on the very early days, and, at that, mostly on the history of hardware, so that the history of these topics is doubly neglected.

2.1 The first programming languages

In the early days of computers, programming was done by directly writing machine instructions: this was difficult and error-prone. Programming languages were invented to allow programmers to write in a more comprehensible form: Fortran dates from 1954, and made it possible to write programs in a notation very like that of standard mathematics (Backus 1981). Although the Fortran designers paid very little attention to theory – Backus, the leader of the project, says "we simply made up the language as we went along" (1981: 30) – both syntax and semantics soon became important factors in the design of programming languages. Algol was designed over the period 1958–60 (Naur 1981; Perlis 1981), Lisp from 1958 to 1962 (McCarthy 1981), and many of the difficulties of developing these languages were due to two factors: it was difficult to define the syntax of a language at all precisely, and the semantics of these languages seemed utterly mysterious. The latter was an extremely serious problem: without some sort of semantics, it was hard to say what counted as a correct implementation of these languages. Although Lisp eventually achieved a precise semantics, it was designed by starting from the implementation and then attempting to find mathematical structure in the resulting language: several of the Lisp primitives were called after hardware features of the machine that it was originally implemented on (McCarthy 1981: 175), whereas its original semantics was "ramshackle" (Landin 2000). Nevertheless, it was also possible to see that the rewards for a precise syntax – or, more ambitiously, a precise semantics – were extremely high: Algol "proved to be an object of stunning beauty" (Perlis 1981: 88).

2.2 Algol-like languages

What, then, do these programming languages look like? We will describe a generic language, quite similar to Algol; since Algol has had an enormous influence on language design its features can be found in many others. These languages have some similarity to formal languages like first-order logic: like these languages, they have variables, to which values can be assigned, and they have both predicates and functions. And many of the basic operations of programming can be viewed as the assignment of values to variables, so one might think that these operations, too, could be viewed in this way.

However, programming languages also have features which are strikingly different from the sort of logical languages familiar to philosophers. In large part, these other features come from a need to control the structure of programs: programs are extraordinary large entities, and programmers can only keep control of this complexity by making programs out of smaller components, which can be individually constructed and tested and, if possible, reused in many different programs. Programs, then, tend to be made of hierarchically nested components; there are various names for these components, but we can – following Algol usage – call them *blocks.*

Furthermore, unlike the logical languages which philosophers are familiar with – namely, variants of untyped first-order logic – modern programming languages are typed: variables, and the values that they take on, have types. This is partly for practical reasons: many programming errors can be detected automatically, simply by checking the types of the entities involved. But there are also rather deeper reasons: programming does not fit very well into a set-theoretic view of things, since sets – even finite sets – are collections without any extra structure, and, however we may choose to store data in a computer, we always do so in some structured way (the data may be ordered, or arranged on the leaves of a tree, or the items may be mapped to integers, and so on).

There is a final difference, which is extremely far-reaching. Most programming languages allow programs to perform actions that change the values of variables, or which have other irreversible effects (input or output, for example); we say that these actions have side-effects. These features add further complications to the task of giving semantics to these programming languages.

2.3 The development of semantics

The first steps towards the semantics of these languages were taken, in the 1960s, by a group – Peter Landin, Dana Scott, and others – associated with Christopher Strachey (Scott 1977; Landin 2000). They provided what is called a denotational semantics: that is, rather than describe the operations that pieces of code perform, they associated mathematical objects – denotations, or semantic values – to the syntactic entities of a programming language. Values are assigned to entities on all scales: the variables (and constants) of the language get values, of course, but so do assignment statements – the parts of programs which give values to variables – as do blocks and subroutines and, finally, the entire program. This requirement – that programs should have semantic values on all scales – is part of the basic program of denotational semantics: it is motivated by the view that programming constructs which have well-defined semantic values will be easy to reason about, and it has, over the years, shown itself to be quite justified.

This requirement means that the semantic values assigned to these entities must belong to a quite intricate system. To see this, consider a particular case of programming entities: namely, those that are sometimes called *subroutines.* These are pieces of code that have parameters: they are invoked with particular values of their parameters, and they then perform various actions on them. A subroutine, then, can be thought of as a sort of function: its arguments are the semantic values of its parameters, and its value is the semantic value of the expression that it returns. So the semantic value of a subroutine must be a function type: it maps its argument types to its return type.

But now consider a subroutine which takes a subroutine as a parameter. Such things occur frequently in the normal practice of programming: for example, we might want to write a subroutine which constructed a button in a user interface. Buttons can perform various actions, and – because we are writing a subroutine which we can use for constructing all sorts of buttons – we want to be able to give the code for the action to the button code as a parameter. The "code for the action," of course, is itself a subroutine: so the code for the button is a subroutine which has a subroutine as one of its parameters. The need for "higher-order" entities of this sort seems to be natural and pervasive in programming: our example is from user-interface programming, but there is a differently motivated example in Abelson et al. 1996: 21–31.

A subroutine with a subroutine as argument, then, can be considered as a piece of code which takes the subroutine and returns a value: so its semantic value will map the semantic value of its subroutine parameter to the semantic value of its result. The semantic value of subroutine-calling code, then, is a function which takes another function as an argument: in technical terms, it is a functional.

We must also remember that the functions corresponding to subroutines cannot, in general, be everywhere defined. We know, from the theory of recursive functions, that any reasonably expressive programming language must have programming constructs – looping, recursion, or both – which cannot be guaranteed to give a result in all cases. This must be accommodated in the semantic values of such subroutines.

2.3.1 Technical interlude*: semantic values in detail*

The initial stages of this accommodation are quite easy to see: we can deal with partial functions from, let us say, the integers to the integers by regarding them as functions from **int** – the usual integers – to $\mathbf{int}_\perp$ – the integers together with an extra element, $\perp$, which is the value of the function when the computation fails to terminate. But now a subroutine which takes such a subroutine as argument must itself have a type which is not merely the type of functions ($\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$, but rather the type of functions ($\mathbf{int} \to \mathbf{int}_\perp) \to \mathbf{int}_\perp$ (which is, one should point out, much more complex than $((\mathbf{int} \to \mathbf{int}) \to \mathbf{int})_\perp$). The moral is clear: although we only have to add a single extra element to our base types, the changes required at higher types become progressively more complex.

Recursion, also, needs a suitable treatment. Consider a recursively defined subroutine, for example:

```
function f(x:int): int
begin
  if (x = 0) then f := 1 else
  (f := x * f(x – 1))
end
```

We can regard this as saying that the subroutine f is a fixed point of a certain operation, namely the operation which takes a subroutine F as input and returns, as output, the subroutine G, defined by

```
function G(x:int): int
begin
  if (x = 0) then G :=1 else
  (G := x * F(x – 1))
end
```

So the semantic value of f must be fixed under the semantic counterpart of the operation $F \mapsto G$; and thus, to handle recursion, the appropriate semantic domains must be closed under certain fixed-point operations. Finally, the need to accommodate assignment statements brings another complication. Consider the following subroutine:

```
function f(x: int): int
begin
  y := x;
  f := y + 1
end
```

which first assigns the value of its argument to a global variable y, and then returns y + 1. Consider also the simpler subroutine

```
function g(x: int): int
begin
  g := x + 1;
end
```

f and g yield the same values for the same arguments, but they are not substitutable, one for the other: g changes the values of a global variable, which f does not. (We say that f has *side-effects.*)

Accommodating this sort of behavior, and still preserving the compositional nature of our semantics, makes the type system for our semantic values somewhat complex and intricate – see Tennent 1994: 250ff for details.

The development of semantics, then, is a process of progressive elaboration of semantic values. We might think of it like this: originally, we have a straightforward conception of what the values of programming entities are: variables stand for their values, subroutines stand for functions from parameters to return values, and so on. We may call this original conception the intended semantics. However, it proves impossible to preserve substitutivity with this intended semantics, so we have to progressively elaborate the semantic values that we assign to programs and their parts; in the process, these semantic values become further and further removed from the original, intended semantics.

The divergences are caused by phenomena that can be viewed, when measured against the intended values, as a lack of referential transparency. The intended values of subroutines such as these ought to be given by the maps, from arguments to return values, that they induce, but, as we have seen, subroutines with the same intended values might not be intersubstitutable: and such a failure of substitutivity is, in Quinean terms, described as referential opacity.

3 The Uses of Semantics

A working semantics on these lines can, indeed, be achieved (Stoy 1977; Tennent 1994), and such semantic accounts of programming language have been widely used.

However, the uses are not as direct as one might imagine. It is rarely expedient, for example, to establish correctness for a particular program by examining the semantic values of it and its components: these semantic values are usually extremely complex. They must necessarily be complex: since it is possible to decide whether a given program terminates or not, purely on the basis of its semantic value, there must be facts about the semantic values of programs that are as difficult to establish as the halting problem (i.e. undecidable).

On the other hand, semantics has a large number of metatheoretical uses. One can, for example, establish equivalences between programs, and, more generally, one can develop, and semantically justify, logics (the so-called Floyd–Hoare logics) for reasoning about programs (Tennent 1994: 196ff; Jones 1992); and, unlike direct reasoning with semantic values, these logics are, for typical problems, easy to work with.

There is another, less formal but very pervasive, use of semantics. The practice of programming involves a great deal of substitution: replacement of one subroutine by another (or one object by another, one library by another, and so on). We like to have languages in which substitutions like this are easy to justify: if we can be sure that, if two items "behave the same" (in some suitably informal sense) they can safely be substituted for each other. In the development of semantics, it very soon became apparent that the semantic properties of languages were decisive for this question: that certain semantic properties made substitution behave well.

3.0.1 Technical interlude*: an issue in programming-language design*

Here is an example of the sort of guidance that semantics can give in language design. Suppose we define a subroutine – call it S, and that we later invoke it. Suppose also that, in the code defining S, there is a global variable x. Suppose, finally, that we change the definition of x between the time that S is defined and the time that it is invoked. Which value do we use for x? There are two obvious choices:

1. the value it had when S was defined: this is called *lexical binding*, and
2. the value it had when S was invoked: this is called *dynamic binding*.

It turns out (Stoy 1977: 46ff) that lexical binding gives a language much better substitution properties, and, in fact, languages with dynamic binding – the typesetting language TeX, for example – are terribly difficult to program with. More generally, it seems to be the case that, if a language has clean, elegant semantic properties, then it will be easy to program in.

3.1 Identity of programs

We use semantics, then, to conclude facts about the behavior of programs on the basis of mathematical properties of their semantic values. We could, for example, observe that, if the semantic values of programs *P* and *Q* were different, then the programs themselves must be different.

This is more subtle than it might seem. What do we mean by identity and difference between programs? A trivial answer would be that it simply consisted in the identity or difference of their source code: but this is rarely of any interest. Programs can vary a good deal, in a merely notational way, and still remain "essentially the same" (whatever that might mean).

A better criterion for the identity of programs is that of *observational equivalence*; philosophically, it can be regarded as a sort of functionalism (see Lycan 1995, Block 1995). One definition is as follows: Two programs, *P* and *Q*, are observationally equivalent if and only if, whenever the inputs of *P* and *Q* are the same, then so are their outputs.

We can define this also for constituents of programs (subroutines, statements, blocks, and the like: these are generically called *program* *phrases*). We define equivalence by observing what happens when phrases are substituted for each other in programs (here a program with a phrase deleted is called a *program context*):

> Two program phrases, f and g, are observationally equivalent if and only if, for any program context $P(\cdot)$, the two programs $P(f)$ and $P(g)$ are observationally equivalent.

Observational equivalence is an extremely versatile property. If we think of anything which might be (in the nontechnical sense) an "observation" of the behavior of a program – stimulating it with certain input, checking the output, and so on – we can automate this observation by writing another program to perform it. This other program will provide a program context with which we can test the program that we are interested in: and thus our definition of observational equivalence can be regarded as an automated version of the everyday concept of observation.

However, observational equivalence is a difficult property to establish, since it talks about what happens when a program fragment is substituted into any program context at all, and in most cases we have no grasp of this totality of program concepts.

If our semantics respects observational equivalence, then we call it *fully abstract*:

> A semantic valuation $\upsilon(\cdot)$ is fully abstract if, whenever program phrases f and g are observationally equivalent, $\upsilon(f)$ and $\upsilon(g)$ are the same.

Full abstraction is, of course, an important concept, because we are interested in observational equivalence. But its interest is rather wider than that: a fully abstract semantics will, in some way, reflect the essential structure of programs, abstracting away from notational or implementational details (Tennent 1994: 242). Of course, we can – rather fraudulently – define fully abstract semantics by starting with a non-fully abstract semantics and imposing equivalence relations on it; but unless we have independent access to the model thus constructed, it would do us no good. In the case when we can find a fully abstract model, and characterize it in some meaningful way – for example, in terms of games (Abramsky et al. 1994; Hyland & Ong 2000) – we have a mathematical object which tells us a great deal about the deep structure of a particular programming language.

3.2 Functional programming

We have been describing an approach to the theory of programming languages which simply seeks to analyze the usual languages that people program in. We might, though, take a different approach to language design: we might want the theory to be more prescriptive, and design programming languages so that they had a good, perspicuous, metatheory.

One of the features which give languages a good metatheory is referential transparency: the property that terms of the language, which stand for the same entities, can always be substituted for each other. Languages with side-effects – such as statements that change the values of variables – do not have referential transparency. A term of such a language might stand for, let us say, a number, but might also, in the course of evaluating that number, change the values of a particular variable; another term might evaluate to the same number, and might change the values of other variables; and it is easy to see that these two terms, even though they evaluated to the same numbers, could not be substituted for each other.

So we might consider designing a programming language in which we could not change the values of variables. What would such a language look like? There could be variables, and we could have definitions that assigned values to variables: however, once we have let a variable have a certain value, we could not subsequently change it. We could also have subroutines: subroutines would take parameters and return values. Because we have no assignment statements, the result returned by a subroutine on particular arguments depends only on the values of its arguments: the same subroutine, evaluated on the same argument, always yields the same result. Subroutines, then, are extensional: they give the same results on arguments with the same referents, and in this respect they behave like mathematical functions.

Following Quine, it is usual to refer to languages with this property as *referentially transparent.*

As we have seen, programming needs higher-order constructs. These languages are no exception: we can let higher-order entities (in this case, functions and higher-order functionals) be the values of variables, we can pass them as parameters to subroutines, and so on. Following Quine's slogan "to be is to be the value of a variable" – a slogan explicitly used by the pioneers of programming-language semantics – we can, and do, give a rough ontology to our language: the entities that can be the values of variables, that are passed to and returned by subroutines, are usually known as "first-class citizens," and they will figure largely in any account of the semantics of the language. These languages – known as functional languages – generally have a large array of such higher-order constructs. Lisp is such a language, but is semantically somewhat impure: modern, more principled versions are untyped languages such as Scheme, and typed languages such as ML.

3.2.1 Technical interlude*: the lambda calculus*

There is an alternative description of these languages. Consider a subroutine, such as the following:

```
begin function f(x, y):
  return 3 * x + 2 * y
end
```

This is the subroutine which takes two parameters, x and y, and returns 3x + 2y. We can give an alternative description of this – in more mathematical notation – as a term in the λ-calculus:

$$\lambda \mathrm{x}.\lambda \mathrm{y}.(3x + 2y)$$

This rough analogy can be made precise: we can set up a correspondence between functional programs and λ-calculus terms, which is compositional and extensional, in such a way that we can obtain a semantics for our programs from a semantics for the λ-calculus.

We can go on from here. It is known that terms in a suitable λ-calculus can be used to encode proofs in higher-order intuitionistic logic (Lambek & Scott 1986); this is called the *Curry–Howard correspondence.* This suggests that functional programs can also be considered to be proofs in that logic: and such in fact is the case. The correspondence between programs and proofs is illuminating in its own right. Consider a subroutine which takes a parameter – say x – and which computes, for example, $3x + 2$. This corresponds to the lambda-term

$$\lambda \mathrm{x} :\mathbf{int}.3x + 2,$$

which corresponds to a proof of the proposition

$$\forall x :\mathbf{int}\exists y :\mathbf{int}$$

but not, of course, just *any* proof: it is the proof which takes the integer x introduced by ∀, computes $3x + 2$, and then uses *that* integer as a premise in ∃-introduction.

4 Conclusions

We have surveyed a rather large area of theoretical computer science; we now have to consider its philosophical relevance. There are, of course, several points of direct relevance: programming-language semantics tells us a great deal about the processes of abstraction involved in programming computers, and also about the nature of algorithms (for which Moschovakis 2001 is a good comparison). However, there are less direct points of interest. The overall goal of programming semantics is quite similar to the philosophical project of developing a theory of meaning: however, the methods and results are strikingly difficult. To a large extent this is because the philosophical project has been developed in isolation, with unsophisticated technical tools, and with the aid of a very small number of examples, none of them either large or complex. The practical necessities of producing a useful body of theory have made it impossible for programming-language semantics to indulge in any of these luxuries. So, the comparative use of programming-language semantics is, perhaps, more interesting than its direct use.

4.1 What aren't we interested in

Programming-language semantics was developed in order to address certain specific needs, and one must understand the biases resulting from those needs to be able to understand the theory and its place in the world.

We know the mechanism The first is obvious: we know all about the mechanisms of computers, because, after all, we made them. This contrasts very strongly with the situation in the philosophy of language and in linguistics, where we have very little information about underlying neurophysiological mechanisms.

We design the systems We also design computers, their operating systems, and the programming languages that we use on them. Many of the choices that we make when we do this are reflections of our needs, rather than of the nature of computation as such; for example, operating systems are extremely modular, and most programming languages have a great deal of support for modularity, simply because modularity makes computers much easier to program. Modularity does not seem to be entailed by the nature of computation as such. By contrast, Fodor's work (Fodor 1983) deploys much more transcendental premises: he is attempting to establish that any minds such as ours must be modular, whatever their mechanisms and however those mechanisms might have arisen. The semantics of programming languages can, of course, neither prove nor disprove the validity of a program like Fodor's – though, of course, it might provide illuminating results on the nature of modularity and on its formal analysis.

Reference is not problematic If we are using computers to solve a problem in the real world, we generally know what the expressions of our programming language stand for: we have, if we are sensible, set things up that way.

Foundationalism is not interesting We may, in principle, *know* the physical processes that the expressions of our programming languages result in, when they are suitably compiled and run. However, we are very rarely interested: looking at computers on that sort of level would submerge the interesting features in an ocean of low-level detail. We would be unable to distinguish, *on that level*, between operations which were performed by the operating system and those which were performed by programs; of the programs running on a real computer, by far the majority would be concerned with trivial housekeeping tasks, rather than anything we were interested in: of the instructions which execute in an interesting program, by far the majority of them would be concerned with rather dull tasks such as redrawing windows on the screen, interacting with the operating system, and so on. On the level of machine-level instructions, none of these processes would be distinguishable from each other in any tractable way.

4.2 What are we interested in?

So what are the interesting problems?

Making languages different There is a huge number of different programming languages, and there are also genuine differences between them. If we were to analyze these languages using the methods of recursive function theory, or by representing programs written in them as Turing-machine programs, we would find (since programming languages are generally Turing complete) that they were indistinguishable from each other. So we want a theory that is finely grained enough to be able to represent the genuine differences between languages. On the other hand, we do not want to differentiate languages that are merely typographical variants of each other, or which differ simply by trivial definitional extensions: we want, that is, a theory that is sensitive to *genuine* differences between languages, and only to those. We would also like to go on and construct a *taxonomy* of languages: that is, we would like to arrange languages in some sort of formal scheme in which we could describe how to get from one language to another by regularly varying theoretical parameters of some sort.

Attaining abstraction There is a common theme running through all of these considerations. When we are designing, or using, a high-level programming language, we are concerned about attaining a sufficient degree of abstraction. Low-level, detailed, grounded descriptions of our systems are unproblematic, but we do not want these: we want to be able to forget about such merely implementational details in order to program, and reason about programs, at the level we are interested in. In order to do this, we need to be able to design languages to do it; and in order to do that, we need some sort of theoretical conception of what these languages should look like. Thus, our semantics should give us a view of computational processes which is equally as abstract as the languages that we want to design. Abstraction, then, is an achievement.

This contrasts sharply with the traditional task of the philosophy of language. Here we start with a high-level view – a speaker's intuitions about language – and we attempt to find a suitably grounded account of this high-level view (Dummett 1991: 13). Attaining the abstract view is not a problem: grounding it is. In the semantics of computation, on the other hand, we already have a grounded view of our subject-matter: it is the construction of a suitably abstract view that is the major difficulty.

4.3 The technical tools

The technical tools used also differ strongly from those current in the philosophy of language community. Programming-language semanticists use higher-order logic and the mathematical theory of categories: linguistic philosophers use first-order logic and set theory. This is a fairly profound difference in mathematical cultures, but it also has to do with the difference between the problems that these communities are addressing.

Intuitionistic logic Semantics uses intuitionist logic a great deal: we have seen, above, that the semantics of functional programming looks very like the proof theory of higher-order intuitionist logic, because programs correspond to proofs of certain propositions. For this, we must use intuitionist, rather than classical, logic: because we want to make programs correspond to proofs, there must be a large number of essentially different proofs of the same proposition. We can rephrase this in terms of equivalence of proofs: we should be able to define a notion of proof equivalence which disregards merely notational variation, but which is not so coarse that the set of equivalence classes becomes trivial. It turns out that, for technical reasons, we can do this for intuitionist logic, but not for classical logic (Girard 1991).

However, this use of intuitionist logic is less ideological than it might seem. We are using it because we need a finely-grained proof theory, and it would be perfectly possible for an ideologically classical logician to use intuitionist logic for these purposes, only because, for computational purposes, one needed a fine-grained proof theory.

In a similar way, we use higher-order logic: higher-order constructions are pervasive in programming, and it is appropriate to have a metatheory which reflects that. However, this preference, again, is not straightforwardly ideological: these higher-order entities are, after all, algorithms, and carry no taint of the infinite. Correspondingly, there are constructive models of set theory in which sets are modeled by equivalence relations on subsets of the integers: we can, in such models, carry out all of the constructions needed to develop programming-language semantics, although we cannot quite accommodate all of traditional higher-order logic (Robinson 1989; Hyland 1982). There is very little that a constructivist can find about such models to object to.

Category theory is also part of the semantic toolkit (see Tennent 1994: 290ff for some examples). But this is hardly any surprise: category theory has found wide application in areas of mathematics – algebraic topology, algebraic geometry, and proof theory – where one wants to disregard "implementational" (or merely notational) detail, and concentrate on the essential features of a situation.

This can be rephrased in more traditional philosophical terms as follows. There is a well-known example, due to Benacerraf (1965), which is (slightly rephrased) as follows: consider two mathematicians (A and B) who both talk about ordered pairs, except that A encodes the ordered pair <*x*, *y*> as {*x*, {*x*, *y*}}, whereas B encodes it as {*y*, {*x*, *y*}}. Now – though A and B clearly each have accounts of ordered pairs which are mathematically adequate – they do not seem to be talking about the same objects (and, in fact, they can be made to disagree by asking them stupid questions of the form ("is *x* a member of <*x*, *y*>?")). One approach to this would be to invoke a difference between specification and implementation, and to say that A and B were simply using different implementations of a single specification. Of course, to do that we need to have some way of making these specifications explicit: and category theory gives us that. We can, given two sets X and Y, specify their Cartesian product $X \times Y$ (the set of ordered pairs with members in each set) in terms of the two maps $X \times Y \to X$ and $X \times Y \to Y$, and of the properties of these two maps. And this characterization turns out to characterize the construction exactly, without involving any purely implementational decisions.

We can think of category theory as ruling out the stupid questions which differentiated between A's and B's mathematics: that is, of giving us a distinction between observable and unobservable properties of mathematical constructions. The observable properties are those which can be expressed in terms of mappings ("morphisms," in category-theoretic terms) between mathematical objects, and in terms of identities between those morphisms; the unobservable ones need identities between objects (Bénabou 1985). It is no surprise, then, that programming-language semantics, which is intimately tied to the observable properties of computer programs, also uses category theory to express that notion of observability.

4.4 Theories of meaning

Finally, we should compare these semantic theories with the philosophical project of a theory of meaning. We should recall that the goal of Davidson's program was to develop an axiomatic theory which would yield, for each sentence of a natural language, a suitable instance of the schema (Dummett 1991: 63)

S is true if and only if **A**.

Truth is not particularly salient in the semantics of programming languages, but we do have an important central notion: that of observational equivalence. So, if we do have a fully abstract semantics, then it can (after suitable manipulation to express it in philosopher-friendly terms) be construed as a sort of counterpart of a theory of meaning: it is a mathematical theory from which we can derive a great number of conclusions about observational equivalence of computer languages. And there are such fully abstract semantics.

However, there are one or two caveats to be made. There is a presumption that, when one had achieved a theory of meaning, one could simply examine to see what its "central notion" was (Dummett 1991: 34). But these semantic theories are possibly a little more recalcitrant: they assign mathematical objects – semantic values – to program phrases, but these mathematical objects do not wear their meanings on their sleeve: there is still room for considerable argument about what they mean. We may, it is true, present mathematical objects using vocabulary which is sufficiently tendentious to make one think that they have a clear and obvious meaning: but this would be *merely* tendentious, having to do with a particular presentation of those objects.

The other caveat is this. The semantics of programming languages has paid particular attention to the question of full abstraction: this concept has been somewhat neglected in the philosophy of language, where the problems have seemed to be those of finding rich enough semantic values to hold all of the components of meaning that we want. However, full abstraction ought to play a role in the philosophy of language as well: as Quine said, there should be no identity without identity (Quine 1969: 23), and the semantic values that we assign to sentence fragments should, in some way, respect the identities of the meanings that we are trying to model.

References

Abelson, Harold, Sussman, Gerald Jay, and Sussman, Julie. 1996. *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA: MIT Press. [An outstanding, and semantically aware, programming textbook, using the functional language Scheme.]

Abramsky, Samson, Jagadeesan, Radha, and Malacaria, Pasquale. 1994. "Full abstraction for PCF (extended abstract)." In M. Hagiya and J. C. Mitchell, eds., *Theoretical Aspects of Computer Software.* Lecture Notes in Computer Science no. 789. New York: Springer-Verlag.

Backus, John. 1981. "The history of Fortran I, II and III." In R. L. Wexelblat, ed., *History of Programming Languages.* New York: Academic Press. From the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978.

Bénabou, Jean. 1985. "Fibred categories and the foundations of naïve category theory." *Journal of Symbolic Logic* 50: 10–37.

Benacerraf, Paul. 1965. "What numbers cannot be." *Philosophical Review* 1974.

Block, Ned. 1995. "Functionalism (2)." In Guttenplan 1995.

Fodor, Jerry A. 1983. *The Modularity of Mind: An Essay on Faculty Psychology.* Cambridge, MA: MIT Press.

Dummett, Michael. 1991. *The Logical Basis of Metaphysics.* London: Duckworth.

Girard, Jean-Yves. 1991. "A new constructive logic: classical logic." *Mathematical Structures In Computer Science* 1(3): 255–96.

Guttenplan, Samuel, ed. 1995. *A Companion to the Philosophy of Mind.* Oxford: Blackwell.

Hyland, J. M. E. and Ong, C.-H. L. 2000. "On full abstraction for PCF." *Information and Computation* 163: 285–408.

Hyland, Martin. 1982. "The effective topos." In A. S. Troelstra and D. van Dalen, eds., *L. E. J. Brouwer Centenary Symposium.* Studies in Logic and the Foundations of Mathematics no. 110. Amsterdam: North-Holland.

Jones, C. B. 1992. *The Search for Tractable Ways of Reasoning About Programs.* Tech. rept. UMCS-92-4-4. Dept. of Computer Science, University of Manchester.

Lambek, J. and Scott, P. J. 1986. *Introduction to Higher Order Categorical Logic.* Cambridge Studies in Advanced Mathematics no. 7. Cambridge: Cambridge University Press.

Landin, Peter J. 2000. "My years with Strachey." *Higher Order and Symbolic Computation* 13: 75–6.

Lycan, William G. 1995. "Functionalism (1)." In Guttenplan 1995.

McCarthy, John. 1981. "History of Lisp." In R. L. Wexelblat, ed., *History of Programming Languages.* New York: Academic Press. From the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978.

Moschovakis, Yiannis N. 2001. What is an algorithm? *Pages 919–936 of*: Engquist, Björn, & Schmid, Wilfried (eds), *Mathematics unlimited – 2001 and Beyond.* Berlin: Springer.

Naur, Peter. 1981. "The European side of the last phase of the development of Algol." In R. L. Wexelblat, ed., *History of Programming Languages.* New York: Academic Press. From the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978.

Perlis, Alan J. 1981. "The American side of the development of Algol." In R. L. Wexelblat, ed., *History of Programming Languages.* New York: Academic Press. From the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978.

Quine, Willard van Ormond. 1969. "Speaking of objects." In *Ontological Relativity and Other Essays.* New York: Columbia University Press.

Robinson, E. 1989. "How Complete is PER?" In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science.* IEEE Press: Washington.

Scott, Dana S. 1977. "Introduction." In Stoy 1977.

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

Tennent, R. D. 1994. "Denotational semantics." S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds., *Handbook of Logic in Computer Science*, vol. 3. Oxford: Oxford University Press.

Travis, Charles. 1986. "Introduction." In C. Travis, ed., *Meaning and Interpretation.* Oxford: Blackwell.

Wiggins, David. 1997. "Meaning and theories of meaning: meaning and truth conditions from Frege's grand design to Davidson's." In B. Hale and C. Wright, eds., *A Companion to the Philosophy of Language.* Oxford: Blackwell.