PROGRAMMING LANGUAGES

AS

MATHEMATICAL THEORIES

Raymond Turner

3BAbstract

That computer science is somehow a mathematical activity was a view held by many of the pioneers of the subject, especially those who were concerned with its foundations. At face value it might mean that the actual activity of programming is a mathematical one. Indeed, at least in some form, this has been held. But here we explore a different gloss on it. We explore the claim that programming languages are (semantically) mathematical theories. This will force us to discuss the normative nature of semantics, the nature of mathematical theories, the role of theoretical computer science and the relationship between semantic theory and language design.

Introduction

The design and semantic definition of programming languages has occupied computer scientists for almost half a century. Design questions centre upon the style or paradigm of the language, e.g. functional, logic, imperative or object oriented. More detailed issues concern the nature and content of its type system, its model of storage and its underlying control mechanisms. Semantic questions relate to the form and nature of programming language semantics (Tennent, 1981; Stoy, 1977; Milne, 1976; Fernandez, 2004). For instance, how is the semantic content of a language determined and how is it expressed?

Presumably, one cannot entirely divorce the design of a language from its semantic content; one is not just designing a language in order to construct meaningless strings of symbols. A programming language is a vehicle for the expression of ideas and for the articulation of solutions to problems; and surely issues of meaning are central to this. But should semantic considerations enter the picture very early on in the process of design, or should they come as an afterthought; i.e. should we first design the language and then proceed to supply it with a semantic definition?

An influential perspective on this issue is to be found in one the most important early papers on the semantics of programming languages (Strachey C. , 2000).

I am not only temperamentally a Platonist and prone to talking about abstracts if I think they throw light on a discussion, but I also regard syntactical problems as essentially irrelevant to programming languages at their present state of development. In a rough and ready sort of way, it seems to be fair to think of the semantics as being what we want to say and the syntax

as how to say it. In these terms the urgent task in programming languages is to explore the field of semantic possibilities….When we have discovered the main outlines and the principal peaks we can go about describing a suitable neat and satisfactory notation for them. But first we must try to get a better understanding of the processes of computing and their description in programming languages. In computing we have what I believe to be a new field of mathematics which is at least as important as that opened up by the discovery (or should it be invention) of the calculus.

Apparently, *the field of semantic possibilities* must be laid out prior to the design of any actual language i.e., its syntax. More explicitly, the things that we may refer to and manipulate, and the processes we may call upon to control them, needs to be settled before any actual syntax is defined. We shall call this the *Semantics First* (**SF**) principle. According to it, one does not design a language and then proceed to its semantic definition as a post-hoc endeavour; semantics must come first.

This leads to the second part of Strachey's advice. In the last sentence of the quote he takes computing to be a new branch of mathematics. At face value this might be taken to mean that the activity of programming is somehow a mathematical one. This has certainly been suggested elsewhere (Hoare, 1969) and criticized by several authors e.g. (Colburn T. R., 2000; Fetzer, 1988; Colburn T. , 2007). But, whatever its merits, this does not seem to be what Strachey is concerned with. The early part of the quote suggests that he is referring to programming languages and their underlying structures. And his remark seems best interpreted to mean that (semantically) programming languages are, in some way, mathematical structures. Indeed, this is in line with other publications (Strachey C. , 1965) where the underlying ontology of a language is taken to consist of mathematical objects. This particular perspective found its more exact formulation in denotational semantics (Stoy, 1977; Milne, 1976), where the theory of complete lattices supplied the background mathematical framework. This has since been expanded to other frameworks including category theory (Oles, 1982; Crole, 1993).

However, we shall interpret this more broadly i.e., in a way that is neutral with respect to the host theory of mathematical structures (e.g. set theory, category theory, or something else). We shall take it to mean that programming languages are, via their provided semantics, mathematical theories in their own right. We shall refer to this principle as the Mathematical Thesis (**MT**).

Exactly what **MT** and **SF** amount to, whether they are true, how they are connected, and what follows from them, will form the main focus of this paper. But before we embark on any consideration of these, we need to

clarify what we understand by the terms *mathematical theory* and *semantics*.

Mathematical Theories

The nature of mathematical theories is one of the central concerns of the philosophy of mathematics (Shapiro, 2004), and it is not one that we can sensibly address here. But we do need to say something; otherwise our claim is left hanging in the air. Roughly, we shall be concerned with theories that are axiomatic in the logical sense. While we shall make a few general remarks about the nature of these theories, we shall largely confine ourselves to illustrating matters and drawing out significant points by reference to some common examples.

Geometry began with the informal ideas of lines, planes and points; notions that were employed in measuring and surveying. Gradually, these were massaged into Euclidean geometry: a *mathematical theory* of these notions. Euclid's geometry was axiomatic but not formal in the sense of being expressed in a formal language, and this distinction will be important later. Euclidean geometry reached its modern rigorous formulation in the 20th century with Hilbert's axiomatisation.

A second, and much later example, is Peano arithmetic. Again, this consists of a group of axioms, informally expressed, but now about natural numbers. Of course, people counted before Peano arithmetic was formulated. Indeed, it was intended to be a theory of our intuitive notion of number, including the basis of counting. In its modern guises it is formulated in various versions of formal arithmetic. These theories are distinguished in terms of the power of quantification and the strength of the included induction principles.

ZF set theory (Jech, 1971) began with the informal notion of set that was operating in 19th century mathematics. It was developed into a standalone mathematical theory by Cantor who introduced the idea of an infinite set given in extension. It had some of the characteristics of the modern notion, but it was still not presented as an axiomatic theory. This emerged only in 20th century with the work of Zermelo and Fraenkel. The modern picture that drives the axioms of ZF is that of the cumulative hierarchy of sets: sets arranged in layers where each layer is generated by forming sets made of the elements of previous layers.

These axiomatic theories began with some informal concepts that are present in everyday applications and mathematical practice. In many

cases, the initial pre-axiomatic notions were quite loose, and most often the process of theory construction added substance and precision to the informal one. This feature is explicitly commented upon by Gödel in regard to Turing's analysis of *finite procedure or mechanical computability* (Turing, 1937). In the words of Wang (Wang, 1974.), 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*. The pre-theoretic analogues of such theories are not always sharp and decisive, and the informal picture is often far from complete. In this respect, the process of theory construction resembles the creation of a novel. And, as with the notion of *truth in the novel,* some things are determined (John did kill Mary) but not everything is (it is left open whether he killed Mary's dog). The mathematical process itself brings these theories into existence. They are in this sense, *definitional* theories.

Although all this is still quite vague, it captures something about what is demanded of an axiomatic theory for it to be considered mathematical. Arbitrary sets of rules and axioms will not do: to be mathematically worthy an axiomatic theory must capture some pre-theoretical intuitive notions in an elegant, useful and mathematically tractable manner. And this is roughly the notion of mathematical theory that we have in mind in the proposition that programming languages are mathematical theories (**MT**).

With this much ground cleared, we may now turn to the function and nature of *semantics*. This will take a few sections to unravel.

Normative Semantics

Syntax is given via a grammar of some sort e.g., context free, BNF, inference rules or syntax diagrams. But a grammar only pins down what the legal strings of the language are. It does not determine what they mean; this is the job of the semantics. We shall illustrate some issues with the following toy programming language.

$$P ::= x := E \mid \textbf{skip} \mid P; P \mid \textbf{if } B \textbf{ then } P \textbf{ else } P \mid \textbf{while } B \textbf{ do } P \mid$$
$$E ::= x \mid 0 \mid 1 \mid E + E \mid E * E \mid$$
$$B ::= x \mid \textbf{true} \mid \textbf{false} \mid E < E \mid \neg B \mid B \wedge B \mid$$

The expressions (E) are constructed from variables (x), 0 and 1 by addition and multiplication. The Boolean expressions (B) are constructed from variables; **true**, **false**, the ordering relation (<) on numbers, negation and conjunction. Finally, the programs of the language (P) are built from a simple assignment statement (x: =E) via sequencing (P;Q),

conditional programs (**if** *B* **then** *P* **else** *Q*) and while loops (**while** *B* **do** *P*). According to the grammar, with parenthesis added, the following program is legitimate, where *n* is an input variable.

$x := 0;\ \ y := 1;$
$\boldsymbol{while}\ x < n\ \boldsymbol{do}\ (x := x + 1; y := x * y)$

But in order to construct or understand this program, one needs to know more than the syntax of its host language; one must possess some semantic information about the language (Turner R. , 2007). Most importantly, in general, a semantic account of a language of any kind must tell us when we are using an expression correctly, and when we are not.

The fact that the expression means something implies that there is a whole set of normative truths about my behavior with that expression; namely, that my use of it is correct in application to certain objects and not in application to others. .... The normativity of meaning turns out to be, in other words, simply a new name for the familiar fact that, regardless of whether one thinks of meaning in truth-theoretic or assertion-theoretic terms, meaningful expressions possess conditions of correct use. Kripke's insight was to realize that this observation may be converted into a condition of adequacy on theories of the determination of meaning: any proposed candidate for the property in virtue of which an expression has meaning, must be such as to ground the 'normativity' of meaning-it ought to be possible to read off from any alleged meaning constituting property of a word, what is the correct use of that word. (Boghossian, 1989)

A semantic account must provide us with an account of what constitutes correct use. It seems generally recognized (Gluer, 2008) that this requirement on a theory of meaning has two components: a criterion of correctness and an obligation to do what is correct. We shall only be concerned with the first. Although aimed at theories of meaning for ordinary language, it is not hard to see that any semantic account of a programming language must equally distinguish *correct* from *incorrect* uses of program constructs. Indeed, in the case of programming languages, there are several central applications of semantic definitions that involve notions of *correctness.*

A semantic account must guide a compiler writer in implementing the language. It must enable a distinction to be drawn between the correct and incorrect implementation of a construct. In other words, it must facilitate a specification of compiler correctness. The compiler must correctly translate the source code into the target code, and correctness demands that the semantic definitions of the two languages must somehow agree under the translation.

From the user perspective, a semantic account must enable a distinction to be drawn between correct and incorrect use of programming

constructs - not just syntactically, but in the sense of meeting their intended specifications (formal or otherwise). For instance, assume the specification is a specification of the factorial function. Then a semantic account must determine whether or not the following program meets it. Syntax alone cannot do this.

$$x := 0;\ \ y := 1;$$
$$\textbf{\textit{while}}\ x < n\ \textbf{\textit{do}}\ (x := x + 1; y := x * y)$$

More generally, a semantic account must enable a distinction to be drawn between software that is intended for different ends i.e., meet different user requirements. For example, it must enable a distinction to be drawn between software intended to act as a web browser and software intended to aid in asset management of power generation. Presumably, a programmer who supplies one rather than the other will get told off.

Given these normative demands, how is a semantic definition of a language to be given? One not obviously implausible suggestion is via an interpretation into another programming language (or a subset of the source one). This is little more than a demand that a compiler provides the semantics. But a little reflection should be sufficient to convince the reader that such an approach does not satisfy our normative demands. Unless the semantics of the target language is given, and thus grounded, the semantics of the source language is not grounded: it just passes the burden of normativity from one language to another. We also need to have some semantic account of the language in which the translation is written. So, by itself, a translation cannot guide the implementer; it is an implementation, not an independent guide to one[1].

The Role of Machines

One way in which this picture might be grounded is in terms of a machine of some sort. This may be achieved stage by stage, one language getting its interpretation in the next, until a machine provides the final and actual mechanism of semantic interpretation. For instance, for our toy language, we require a machine with an underlying state whose role is to store numerical values in locations. Pictorially, this might take the following shape.

[1] But see (Rapaport, 2004).

x	y	z	w....
5	7	9	7..... .

The semantics of assignment is then unpacked by its impact on it. But what is the nature of this store? Is it physical or abstract? One common sense view is that, in order to block the potentially infinite regress of languages, it must be a physical device that grounds the meaning in the physical world. More explicitly, the intended meaning of the language is to be given by the actual effect on the state of a physical machine.

In particular, consider the following assignment instruction.

$$x := E$$

How is its semantics to be given on a physical machine? Apparently, the machine does what it does when the program is run - and what it does determines the meaning of assignment. But there are dissenters to such a view.

Actual machines can malfunction: through melting wires or slipping gears they may give the wrong answer. How is it determined when a malfunction occurs? By reference to the program of the machine, as intended by its designer, not simply by reference to the machine itself. Depending on the intent of the designer, any particular phenomenon may or may not count as a machine malfunction. A programmer with suitable intentions might even have intended to make use of the fact that wires melt or gears slip, so that a machine that is malfunctioning for me is behaving perfectly for him. Whether a machine ever malfunctions and, if so, when, is not a property of the machine itself as a physical object, but is well defined only in terms of its program, stipulated by its designer. Given the program, once again, the physical object is superfluous for the purpose of determining what function is meant. (Kripke, 1982)

There is no appeal to an independent specification; meaning is completely determined by what the machine does. It follows that there is no notion of malfunction, and no notion of correctness. So there is no sense to be made of the demand that the machine behave correctly. For this, some machine independent account is needed. This may be expressed in the following way.

> *When the state is updated by placing v in location x, and then the contents of x is retrieved, v will be returned. For any other location, the contents remain unchanged.*

Where ***Update*** changes the value in a given location and ***Lookup*** returns the value at a given location, we may rewrite this more symbolically as follows.

$$\boldsymbol{Lookup}(\boldsymbol{Update}(s, x, v), x) = v$$
$$\boldsymbol{Lookup}(\boldsymbol{Update}(s, x, v), y) = \boldsymbol{Lookup}(s, y) \quad where\ x \neq y$$

But these simple equations determine an operation on an abstract machine. And it is this that supplies the specification of the physical one, and makes the latter (semantically) superfluous. If the command *x:=10* places 28 in location *y*, this is not correct.

It would seem that any normative semantic account of our toy language must be given in terms of its impact upon such an abstract machine. Physical operations may conform to the specification given by the abstract ones, but they cannot provide a semantic correlate for a program.

Informal Semantics

But the nature of the machine is only part of the story. We still need to say how a whole programming language is to be interpreted. The most common approach employs natural language, where such accounts most often take the form of a reference manual for the language. And they can be big: the one for Java Language is almost 600 pages. The following is taken from The Java Language Specification, Third Edition - TOC

A while statement is executed by first evaluating the expression. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8). If execution of the expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, the while statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value: If the value is true, then the contained statement is executed. Then there is a choice: If execution of the statement completes normally, then the entire while statement is executed again, beginning by re-evaluating the expression. If execution of the statement completes abruptly, see §14.12.1 below. If the (possibly unboxed) value of the expression is false, no further action is taken and the while statement completes normally. If the (possibly unboxed) value of the expression is false the first time it is evaluated, then the statement is not executed.

This is the standard semantics of the *while* statement within the Java language. However, there are several complications that pertain to the special character of this language. For the time being, we shall ignore most of these and concentrate on the central issues. For this purpose we shall illustrate the semantic process with our toy language. Later we shall consider some of the complexities that arise with real languages.

As with the semantic conception of truth, our abstract notion of *execution* emerges from a recursive semantic description of the whole language.

1. If the execution of *E* in the state *s* returns the value *v*, then the execution of *x*:=*E* in a state *s*, returns the state that is the same as *s* except that the value *v* replaces the current value in location *x* i.e., ***Update***(*s*,*x*,*v*).

2. The execution of **skip** in a state *s*, returns *s*.
3. If the execution of *P* in *s* yields the state *s*' and the execution of *Q* in *s*' returns the state *s*", then the execution of *P;Q* in *s*, returns the state *s*"
4. If the execution of *B* in *s* returns **true** and the execution of *P* in *s* returns *s*', then the execution of **if** *B* **then** *P* **else** *Q* in *s*, evaluates to *s*'. If on the other hand, the execution of *B* in *s* returns **false** and the execution of *Q* in *s* returns *s*', then the execution of **if** *B* **then** *P* else *Q* in *s*, evaluates to *s*'.
5. If the execution of *B* in s returns **true**, the execution of *P* in *s* returns *s*', and the execution of **while** *B* **do** *P* in *s*' yields *s*", then the execution of **while** *B* **do** *P* in *s*, returns s". If the execution *B* in *s* returns false, then the execution of **while** *B* **do** *P* in *s*, return *s*.
6. The execution of a variable in state *s* returns the value obtained by looking it up in *s*.
7. If the execution of *E* in state *s* returns *v* and the execution of *E'* returns *v'* then the execution of the addition of *E* and *E'*, returns the addition of *v* and *v'*. We proceed similarly for multiplication.

This provides a natural language semantic account for our toy language. But being based upon an underlying abstract machine, it is an abstract account i.e., the semantics is given in terms of relations on the abstract machine.

Such an approach works well with simple languages, but with real ones matters are less clear. It is difficult to express essentially technical notions in natural language. For one thing, it does not always facilitate being clear about what we are talking about. Furthermore, the consequences of design decisions, articulated in natural language, may not be as sharp as they could be.

> In particular, Java has integrated multithreading to a far greater extent than most programming languages. It is also one of the only languages that specifies and requires safety guarantees for improperly synchronized programs. It turns out that understanding these issues is far more subtle and difficult than was previously thought. The existing specification makes guarantees that prohibit standard and proposed compiler optimizations; it also omits guarantees that are necessary for safe execution of much existing code (Pugh, 2000)

This indicates that there are deeper problems than ambiguity, the normal source of problems with natural language definitions. Lack of clarity cuts deeper than scope distinctions. In particular, there is a lack of

semantic clarity over the basic notions such as *threading* and *synchronization*. It is not a reformulation in a more formal language that is required, but a better conceptual understanding of these fundamental notions. Nor can we glean what they are supposed to do by running experiments on a machine. What they are supposed to do must be fixed by an abstract normative account.

Furthermore, even the simple consequences of the semantics are not easy to articulate. For example, to ensure that it is coherent, we shall need to establish that expression execution does not change the state. This much we have assumed in our informal semantic account. Similarly, a compiler writer will need to argue, with some degree of precision, that the compiler is correct. This will involve an inductive argument that must take place during the construction not after it. Such arguments are not optional; at some level, and with some degree of precision, one cannot construct a compiler without undertaking such reasoning.

So despite its prevalence, there are non-trivial problems with natural language accounts.

Operational Semantics

However, a little notation will help with some of them. More specifically, we shall write

$$< P, s >\Downarrow s'$$

to indicate that evaluating P in state s terminates in s'. With this notation we can rewrite the whole semantic account of our simple language. It will be little more than a rewrite of the informal account with this notation replacing the words execute/execution.

1. Assignment

$$\frac{< E, s >\Downarrow v}{< x := E, s >\Downarrow \boldsymbol{Update}(s, x, v)}$$

2. Skip

$$< skip, s >\Downarrow s$$

3. Sequencing

$$\frac{< P, s >\Downarrow s' \qquad < Q, s' >\Downarrow s''}{< P; Q, s >\Downarrow s''}$$

4. Conditionals

$$\frac{< B, s >\Downarrow true \qquad < P, s >\Downarrow s'}{< \boldsymbol{if}\ B\ \boldsymbol{do}\ P\ \boldsymbol{else}\ Q, s >\Downarrow s'} \qquad \frac{< B, s >\Downarrow false \qquad < Q, s >\Downarrow s'}{< \boldsymbol{if}\ B\ \boldsymbol{do}\ P\ \boldsymbol{else}\ Q, s >\Downarrow s'}$$

5. While

$$\frac{< B, s >\Downarrow \textbf{true} \quad < P, s >\Downarrow s' \quad < \textbf{while } B \textbf{ do } P, s' >\Downarrow s''}{< \textbf{while } B \textbf{ do } P, s >\Downarrow s''} \qquad \frac{< B, s >\Downarrow \textbf{false}}{< \textbf{while } B \textbf{ do } P, s >\Downarrow s}$$

6. Variables

$$< x, s >\Downarrow < \textbf{\textit{Lookup}}(x, s), s >$$

7. Addition and Multiplication

$$\frac{< E, s >\Downarrow < v, s' > \quad < E', s' >\Downarrow <, v's'' >}{< E + E', s >\Downarrow < v + v', s'' >} \qquad \frac{< E, s >\Downarrow < v, s' > \quad < E', s' >\Downarrow <, v's'' >}{< E * E', s >\Downarrow < v * v', s'' >}$$

In addition to the use of our simple notation, we have replaced the conditional form of the informal semantics by rules. In particular, the antecedents of the informal rules e.g.

> If the execution of *B* in *s* returns **true** and the execution of *P* in *s* returns *s*', then…

are represented as the premises of the formal ones e.g.

$$< B, s >\Downarrow true \qquad < P, s >\Downarrow s'$$

So, apart from the fact that the inferential structure of the rules is now made explicit, these are minor changes.

But with this version of the semantics in place, we can more explicitly state a result that guarantees that the evaluation of expressions has no side effects.

> For all expressions *E* and states *s*
>
> $$if \ \ < E, s >\Downarrow < v, s' > \ \ then \ s = s'$$

The actual proof proceeds by induction on the expressions using the rules for the respective cases: we argue, by induction, that the execution of expressions does not yield side effects. For the base case, we observe that the execution of variables does not change the state. For the induction step, on the (inductive) assumption that the execution of E and E' do not, i.e.,

$$< E, s >\Downarrow < v, s > \qquad < E', s >\Downarrow < v, s >$$

it is clear that the execution of E+E' does not i.e.,

$$< E + E', s >\Downarrow < v + v', s >;$$

And the same result hold for multiplication.

Such arguments ensure that the informal semantics is *safe*. Without them, the semantic account for the execution of programs needs to be adjusted in order to take account of state change during expression execution.

So our simple notation enables a more transparent formulation of the results about the theory. It is not that far removed from the informal account, but it is more wholesome.

A Theory of Programs

But it is not just a semantic account; looked at more abstractly, our semantics constitutes *a theory of programs*. More exactly, we can read the above semantic account as a theory of operations determined by their evaluation rules. Here the relation ⇓ is taken to be sui-generis in the proposed theory and axiomatised by the rules.

To emphasize this mathematical nature, we shall mathematically explore matters a little. For example, we may define

$$< P, s >\Downarrow \triangleq \exists s' \cdot < P, s >\Downarrow s'$$

This provides a notion of *terminating* program. We may also define a notion of equivalence for programs.

$$P \simeq Q \triangleq \forall s \cdot \forall s' \cdot <P,s>\Downarrow s' \leftrightarrow <Q,s>\Downarrow s'$$

i.e., we cannot tell them apart in terms of their extensional behaviour. Technically, this is an equivalence relation. Moreover, we have the provability of the following three propositions that govern the partial equality of our programming constructs.

1. **if true then** P **else** $Q \simeq P$
2. **if false then** P **else** $Q \simeq Q$
3. **while** B **do** $P \simeq$ **if** B **then** (P; **while** B **do** P) **else skip**

So we have the beginnings of a theory of programs. It certainly captures ones intuitions about the evaluation mechanism that is implicit in the standard informal understanding of these constructs. While not a deep and exciting one, it is still a mathematical theory. Consequently, it would appear that a programming language (i.e., the bundle that is its syntax and semantics) is a mathematical theory i.e., we appear to have arrived at **MT**.

Unfortunately, this claim may be challenged at every step.

Empirical Semantics

We can attempt to block matters at the outset i.e., we may attack the practical necessity for any kind of semantics, even of the informal variety, i.e., one might claim that semantics is irrelevant in practice. Whatever, the intention of the original designer, it is how the language

functions in the working environment that determines the activity of programming. And for this, any pre-determined normative semantic description is largely irrelevant. This would block **SF**; indeed it seems to deny any role for semantics. So is it plausible? Here is one set of considerations in its favour.

A programmer attempting to learn a programming language does not study the manual, the semantic definition. Instead, she explores the implementation on a particular machine. She carries out some experimentation, runs test programs, compiles fragments etc. until she figures out what the constructs of the language do. Learning a language in this way is a practical affair. Moreover, this what programmers require in practice. Indeed, in order to program a user needs to know what will actually happen on a given physical machine. And this is exactly what such a practical investigation yields.

In other words, a programming language is treated as an artefact that is subject to experimental investigation. The programmer still needs to construct her own theories about the semantic content of the language. But presumably, through testing and experimentation, together with her previous knowledge of programming languages and their constructs, she could systematically uncover the evaluation mechanism of the language. Indeed, she might be able to piece together something like our operational semantics[2]. But such theories are constructed as scientific theories about the language and its implementation, and as such they are subject to falsification. On this scenario, it is this experimental method that enables us to discover the actual meaning of the language. This is a very different methodological picture to that supplied by the normative one.

Of course, we might doubt whether such theory construction is practically feasible: can one from scratch unpack matters to the point where one has enough information to use the language? But even assuming that we find such methodology persuasive, and that we can write down the evaluation mechanism, there is a more significant

[2]This might be seen as similar in spirit to Quine's field linguist engaged in what he refers to as *radical translation* (Quine, 1960). In so far as a user could by some form of experimentation fix the interpretation of the language, it is. However, this form of empirical uncovering of semantics is not an argument against its normative function. It is merely a route to finding out what it means. Once the translation manual has been constructed, it provides a means of fixing correct use. Indeed, this provision is built into Davidson's' perspective (Davidson, 1984) where the role of the field linguist is radical interpretation not translation. Here the goal is the construction of a *theory of meaning* that is compositional. But these issues require more careful analysis than is possible here.

problem with this empirical approach. Empirical theories are subject to falsification and so, by their very nature, cannot be normative. So it would seem to follow that the advocate of this empirical picture must believe that no normative account is necessary, and that matters are always up for revision. But, this cannot be right. As we originally argued, without some normative account, there can be no criterion of correctness and malfunction, and no standard by which to measure progress. Programming involves reasoning, and this requires a distinction between the correct and incorrect use of expressions of the language. And this can only take place against a semantic account of the language that fixes the correct use of its constructs. Although the activity of programming will almost always involve some form of experimentation and testing, this must take place against the backdrop of some normative account.

To square this demand with the present empirical picture we might amend matters slightly in order to make room for a normative role for the extracted theory. We might begin with the empirical approach. But what may have been first formulated as a scientific theory of the language, in the activity of programming, must assume normative status i.e., once formulated, this initial scientific theory of the language must act as (a reverse engineered) semantic specification of the language.

However, there are serious objections to even this picture. In particular, there must still be an initial normative account that underpinned the original compiler. Even the compiler writer, who just happens also to be the language designer, has semantic intentions. So this experimental picture cannot gain any purchase without some initial normative foundation. Moreover, assuming a normative status for any empirically derived theory faces the very same problem that made the construction of the scientific theory seem necessary in the first place: in the future, the whole system may malfunction in new ways not predicted by the theory. In this empirical setting, the user requirement that initiated the scientific perspective (i.e., the user needs to know what actually happens) will lead to the development of a new theory. And so on. Indeed, it would seem that this user requirement is unobtainable: continual revision is required to feed this desire to know what actually happens. This is not to say that some experimentation of the sort described, may not occur in practice. All sorts of things may occur in

practice. But it is to say that one cannot dispense with a normative role for theories of the language, however they are come by.

Indeed, this whole approach to the semantics of a language seems confused. There is a clear difference between what the language is taken to mean and how we discover its meaning. Any attempt to discover the meaning of the language by testing and experimentation, presupposes that there is some pre-determined notion of meaning to discover.

So there seems little possibility of undermining **MT** by this route i.e., arguing away the need for a normative semantics. However, we might challenge the second step i.e., the move from the informal to the formal semantics.

Informal Mathematics

Have we not assumed the conclusion of **MT** in moving from the informal to the formal account i.e., by providing a rule based account using the more formal notation, have we not pre-judged the issue? Indeed, the objector might agree that the formal account is mathematical, but argue that we do not need it for practice, thereby undermining **MT**.

The arguments given for the formal account were essentially pragmatic in nature; they insist that precise accounts enable us to more carefully articulate the ontology and express and prove the properties of the language. But such arguments are not arguments that show the necessity of such a formal semantics. The informal ones, carefully formulated, might still be sufficient to define and explore the language.

However, even if we doubt the need for the more formal account, it is not clear that we need to give up **MT**: if we stick to informal semantics and informal argumentation, does it follow that we lose mathematical status for our theories? Not obviously. Actually, it seems that not much hangs on the formalization step.

In our brief account of the nature of mathematical theories we alluded to the distinction between being *formal* and being *mathematical*. Although formal logic and set theory have influenced the style and presentation of proofs, ordinary mathematical proofs are not articulated in any formal language. Most mathematicians do not work inside formal theories expressed in some variant of predicate logic; most mathematics is articulated in ordinary language with a sprinkling of notation to pick out the underlying concepts. Moreover, the use of the formal notation

does not transform a non-mathematical account into a mathematical one. The mathematical status of the theory does not depend upon such formal presentation: its mathematical nature is not brought into existence by it. In fact, the move from the informal to the formal is common place in mathematics. Informal theories often get rigorously axiomatised later e.g., Hilbert's Geometry. But the informal accounts are still mathematical. Euclid's geometry, despite its informality, is still taken to be a mathematical theory. It did not suddenly get mathematical status in the 20th century with Hilbert's axiomatisation.

In the case of our toy language, apart from the fact that one is expressed in English and the other with some abbreviational notation, and in the formal version the rule based structure has been made explicit, there is a no difference between the two versions of the semantics. Surely such cosmetic changes cannot have such a significant conceptual consequence.

Consequently, the argument that semantic accounts are mathematical does not depend upon the semantics and underlying theory being formally articulated. And this is consistent with the standard development of axiomatic mathematical theories. In our case, there seems to be an underlying theory of operations that forms part of the thing that is a programming language. Consequently, at this point, at least for our toy language, we have no compelling reason to give up **MT** in its present form. In particular, the thing that is our programming language is a theory of programs, formally presented or not.

Conservative Extensions

However, although we might allow that simple theories such as our theory of programs are worthy of mathematical status, we might still insist that this is not so for actual programming languages; what might hold for simple toy languages does not scale up. In particular, theories acceptable to the mathematical community must have some aesthetic qualities: they must have qualities such as elegance and ease of application in their intended domain of application. Moreover, part of being elegant involves the ability to be mathematically explored. If they cannot, for whatever reason (e.g. their complexity), they will not be given the mathematical communities stamp of approval. And while it is possible to provide semantic definitions of the kind given for our toy language for large fragments, and even whole languages (for example, (Wikibooks, 2009) provides a semantic definition of Haskell), in general,

such definitions are not tractable theories. They are hard, if not impossible, to mathematically explore. They are often a complex mixture of notions and ideas that do not form any kind of tractable mathematical entity. Consequently, when provided, such semantic definitions are often complicated and unwieldy, and therefore of limited mathematical value. Often, the best one can do with some of these is to marvel at the persistence and ingenuity of the person who has written the semantic description. Given this, it is harder to argue that actual programming languages are genuine mathematical theories.

However, there is an observation that, on the face of it, might be taken to soften this objection. And this involves the logical idea of a *conservative extension*. Suppose that we have constructed a theory T_1 of a language L_1. Suppose also that, in the sense of mathematical logic, we have shown that T_1 is a conservative extension of a smaller theory T_2, a theory of a language L_2, a subset of L_1. Further suppose that T_2 meets our criteria for being a mathematical theory. Can we then claim that T_2 is also a mathematically acceptable theory? In other words, is a theory that is a conservative extension of a mathematical theory, also a mathematical theory? A positive answer fits mathematical practice where mathematical exploration results in the construction of conservative extensions. Indeed, the construction of these extensions is itself part of the exploration process of the core theory.

Programming languages admit of a similar distinction. While the whole language/theory may not have sufficient simplicity and elegance to be mathematically explored, it may nevertheless possess a conceptual core that may be. Such a core should support the whole language in the sense that the theory of the latter is a conservative extension of the theory of its core. This offers a slightly different interpretation of **MT**. But it is one in line with mathematical practice.

Unfortunately, there are further problems to overcome. No doubt there are some simple economies of syntax and theory that may be made for almost all languages. But it will generally be a non-trivial task to locate such mathematically acceptable cores for existing languages. Many languages have been designed with a meagre amount of mathematical input, and it would be somewhat miraculous if such languages/theories could post-hoc be transformed into elegant cores.

MT and SF

But there is another route. And one that brings **SF** back to the fore. The nature of existing languages does not dictate how new languages might be designed. It does not logically prevent elegant computational theories from being used as an aid to the design of new languages; languages that come closer to achieving mathematical status.

And this brings in the role of theoretical computer science. One of its goals has been to isolate pure computational theories of various kinds. Some of these notions were already embedded in actual programming languages, and, in many cases, formed the source of the underlying intuitions that were sharpened and moulded into an axiomatic theory. Mostly they have not been devised to be used, but to provide careful axiomatic articulations of informal, yet significant, computational concepts. Such theories include axiomatic theories of the following notions.

- Operations
- Types and Polymorphism
- Concurrency and Interaction
- Objects and Classes

Theories of operations mostly emanate from the Lambda Calculus (Church, 1941). This was invented as a formalism to provide a formal account of computability. But from a computer science perspective (Landin P. , 1965; Landin P. , 1964), it provides a mathematical account that underlies the notions of *function/procedure definition* and *function/procedure call* as they occur in actual programming languages. Landin (Landin P. , 1966) actually advocated that the calculus be used as the design core for future languages. Other variations on the calculus take seriously the fact that expressions in the language of the lambda calculus may fail to terminate under the standard rules of reduction. This leads to the Partial Lambda Calculus (Moggi.A., 1988).

However, most programming languages admit some notion of *type*, and so these pure untyped theories of operations do not reflect the operational content of existing languages. Consequently, logicians and theoretical computer scientists have developed variations on the calculus that incorporate types (Barandregt, 1992). While the elementary theories have *monomorphic* type systems, most languages now admit some notion of *polymorphism*. Theories of the impredicative notion (e.g.

System **F**) were invented independently by the logician Girard (Girard, 1989) and the theoretical computer scientist Reynolds (Reynolds, 1974). This is an impredicative theory in that the polymorphic types are included in the range of the type variables. Less powerful theories, in particular predicative ones restrict the range to exclude these types from the range. Others carve out various subsets of the type system and restrict the range to these. These theories and their mathematically established properties provide us with hard information for the activity of design.

The π-calculus (Milner R. , 2006) belongs to the family of *process* calculi: mathematical formalisms for describing and analyzing properties of *concurrent computation* and *interaction*. It was originally developed as a continuation of the Calculus of Communicating Systems. Whereas the λ-calculus is a pure theory of operations, the π-calculus is a pure theory of processes. It is itself Turing complete, but is has also inspired a rich source of extensions that get closer to being useable programming languages e.g. (Barnes, 2006).

Our final example concerns *objects*, *classes* and *inheritance*. (Abadi, 1996) contains an extensive source for such calculi (e.g. ς – *calculus*), including some with type structure. The authors also consider the interaction of such theories with other notions such as polymorphism.

One would be hard pushed to argue that such theories are not mathematical ones. They not only reflect clear computational intuitions, often derived from existing languages, but they are capable of being mathematically explored. Indeed, the pure lambda calculus is now a branch of mathematical logic/theoretical computer science with its own literature and mathematical goals (Barendregt, 1984).

The design and exploration of such theories might well be used, as one tool among many, to aid the process of language design. Actual programming languages might then be designed around such cores with actual implemented programming languages and their theories as conservative extensions. Some languages have been designed using this broad strategy. For example, *the logic of computable functions* of (Scott, 1993) is an extension of the simple typed lambda calculus that includes a fixpoint/recursion operator. A predicative polymorphic version of this (with type variables ranging over types with decidable equality) forms the logical spine of ML (Milner R. T., 1999). But one would need to do a fair amount of work to even articulate the theory of the whole language,

let alone investigate whether or not it is a conservative extension of this core. Still, it is within the spirit of the present proposal.

Moreover, programming languages are rarely based upon a single core notion. In reality we require languages that support quite complex mixtures of such. For example, we might form a theory made up from the $\pi - calculus$, the $\varsigma - calculus$ and some predicative version of system **F**. This should enable us to explore combinations of polymorphism, concurrency and objects i.e., we may subject such a theory to mathematical analysis. We might for example show that type membership is decidable. This informs the language design process. Indeed, we would be able to investigate and prove *safety guarantees for improperly synchronized programs* (Pugh, 2000). While putting such theories together in coherent ways is no easy task, there are theoretical frameworks that support such merging activity (Goguen, 1992; Turner R. , 2009).

Strachey's plan was that such fundamental notions should be first clarified and languages designed with this knowledge to hand. This idea has actually furnished a whole industry of language design. More specifically, the last forty years have seen the employment of denotational and operational semantics as tools in programming language design (Tennent, 1977; Schmidt, 1986).

Our approach is slightly different but still in line with the **SF** principle. In our case it is our core theories that supply the material from which actual languages may be constructed. Of course, Strachey never put it in these terms; such theories were largely not around at the time of his pronouncement. His original idea alluded to some underlying structures that were left unspecified. The interpretation that resulted in denotational semantics came later. Nevertheless, the spirit of what we are suggesting is much the same. It is a version of Strachey's idea with his informal ideas being fleshed out with foundational axiomatic theories.

This is a very clean picture, but it must represent the ideal situation. In practice, there is more to design than devising and exploring such core theories and their combinations. One also needs also to take pragmatic issues, into account. Central here are issues of programming practice and implementation (Wirth, 1974). Indeed, the whole enterprise of language design is a two-way street with theory and practice informing

each other. In order to build pure computational theories, one must have some practice to reflect upon. Practice plus some theory leads to actual languages, which in turn generates new theories that feed back into language design. The various activities bootstrap each other. This finds the appropriate place for theory: it advocates a *theory first principle,* for each new generation of programming languages. This endorses both a more realistic interpretation of the semantics first principle, and increases the chances that the resulting theory will be mathematically kosher.

Conclusion

This is just one topic in the conceptual analysis of the nature of programming languages. Such work should form a significant part of a philosophy of computer science. In particular, the status of programming languages, as mathematical theories, raises issues that impinge upon some of the central and contemporary questions in the philosophies of language, mathematics, science and engineering. In particular, in examining Strachey's claims, we are as much engaged in clarifying the nature of mathematical theories as we are in examining the nature of programming languages.

Bibliography

Abadi, M. a. (1996). *A Theory of Objects.* New York: Springer-Verlag, Monographs in Computer Science.

Barandregt, H. (1992). Lambda Calculi with Types. In D. M. S. Abramsky, *Handbook of Logic for Computer Science Vol 2* (pp. 117-309). Oxford University Press.

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (Vols. Studies in Logic and the Foundations of Mathematics, 103 (Revised edition ed.). Amsterdam: North Holland.

Barnes, F. a. (2006). Retrieved from Occam-pi: blending the best of CSP and the Pi-calculus: http://www.cs.kent.ac.uk/projects/ofa/kroc/

Boghossian, P. (1989). The Rule-following Considerations. *Mind* , 507-549.

Church, A. (1941). *The Calculi of Lambda Conversion.* Prineton: Princeton University Press.

Colburn, T. (2007). Methodology of Computer Science. In L. Floridi, *The Blackwell Guide to the Philosophy of Computing and Information* (pp. 318--326). Blakwell, Oxford.

Colburn, T. R. (2000). *Philosophy and Computer Science.* New York: Explorations in Philosophy. Series. M.E. Sharpe.

Crole, R. (1993). *Categories for Types.* Cambridge: Cambridge University Press.

Davidson, D. (1984). Radiical Interpretation. In D. Davidson, *Inquiries into Truth and Interpretation* (pp. 125-140). Oxford: Oxford University Press.

Fernandez, M. (2004). *Programming Languages and Operational Semantics: An Introduction.* London: King's College Publications.

Fetzer, J. (1988). Program Verification: The Very Idea. *Communications of the ACM 31(9)* , 1048--1063.

Girard, L. a. (1989). *Proofs and Types.* Cambridge: Cambridge University Press.
Gluer, K. W. (2008). *The Normativity of Meaning and Content.* Retrieved from Stanford Encyclopedia of Philosophy: http://plato.stanford.edu/entries/meaning-normativity/
Goguen, J. a. (1992). Institutions: Abstract Model Theory for Specification and Programming. *J. ACM 39(1)* , 95-146.
Hoare, A. (1969). An Axiomatic Basis For Computer Programming. *Communications of the ACM, Volume 12 / Number 10* , 576-583.
Jech, T. (1971). *Lecture Notes in Set Theory.* New York: Springer.
Kripke, S. (1982). *Wittgenstein on Rules and Private Language.* Boston: Harvard University Press.
Landin, P. (1965). A Correspondence Between ALGOL 60 and Church's Lambda-Notation. *Communications of the ACM, vol. 8, no. 2.* , 89-101.
Landin, P. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal 6(4).* , 308-320.
Landin, P. (1966). The next 700 Programming Languages. *Communications of the ACM* , 157-166.
Milne, R. a. (1976). *A Theory of Programming Language Semantics.* Chapman and Hall.
Milner, R. T. (1999). *The Definition of Standard ML.* MIT Press.
Milner, R. (2006). *The Polyadic π-Calculus.* Berlin: Springer.
Moggi.A. (1988). *http://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-63/.*
Oles, F. J. (1982). *A category-theoretic approach to the semantics of programming languages.* Syracuse, NY, US: Syracuse University.
Plotkin, G. (2004). A structural approach to operational semantics. *J. Log. Algebr. Program, vol. 60-61* , 17-139.
Pugh, W. (2000). The Java Memory Model is Fatally Flawed. *Concurrency: Practice and Experience 12(6).* , 445-455.
Quine. (1960). *Word and Object.* . Cambridge, Mass: MIT Press.
Rapaport, W. (2004). Implementation is Semantic Interpretation. *Monist, volume 82.* , 109--130.
Reynolds, J. (1974). Towards a theory of type structure. In *Lecture Notes in Computer Science.* (pp. 408-425). Berlin: Springer.
S. Abramsky, D. M. (1992). *Handbook of Logic in Computer Science. Vol 2.* Oxford: Oxford University Press.
Schmidt, D. (1986). *Denotational Semantics: A Methodology for Language Development.* Boston: Allyn and Bacon.
Scott, D. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science* , 411-440.
Shapiro, S. (2004). *Philosophy of Mathematics: Structure and Ontology.* Oxford: Oxford University Press.
Stoy, J. (1977). *The Scott-Strachey Approach to Programming Language Semantics.* Boston: MIT Press.
Strachey, C. (2000). Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation.* , 11-49.
Strachey, C. (1965). Towards a formal semantics. In T. B. Steel, *Formal Language Description Languages for Computer Programming.* Amsterdam, North Holland (1965). .
Tennent, R. (1977). Language design methods based on semantic principles. *Acta Informatica, 8* , 97–112.
Tennent, R. (1981). *Principles of Programming Languages.* Oxford: Prentice-Hall International.

Turing, A. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, 2 42.* , 230--65.
Turner, R. (2009). *Computable Models.* New York: Springer.
Turner, R. (2007). Understanding Programming Languages. *Minds and Machines 17(2)* , 129-133.
Wang, H. (1974.). *From Mathematics to Philosophy.* London: London: Routledge & Kegan Paul.
Wikibooks. (2009). *Haskell/Denotational Semantics*. Retrieved from Wikibooks: http://en.wikibooks.org/wiki/Haskell/Denotational_semantics
Wirth, N. (1974). On the Design of Programming Languages. *IEEE Trans. Software Eng.* , 386-393.

Key Terms and Their Definitions

1. *Axiomatic Theories*: Theories constituted by groups of axioms/rules. These are not necessarily cast within a formal language i.e., they may be informally presented.
2. *Computational Theories*: Theories that are axiomatisations of computational notions. Examples include the λ and π calculi.
3. *Informal Mathematics*: Mathematics as practised; not as formalised in standard formal systems.
4. *Operational semantics*: A method of defining programming languages in terms of their underlying abstract machines.
5. *Mathematical Theories*. In this paper these are interpreted as axiomatic theories in the logical sense.
6. *Theoretical Computer Science*: the mathematical theory of computer science. In particular, it includes the development and study of mathematical theories of computational notions.