Forms of Semantic Specification

MS-CIS-91-61
Logic & Computation 37

Carl Gunter

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

September 1991

THE LOGIC IN COMPUTER SCIENCE COLUMN

by

Yuri GUREVICH

Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor, MI 48109-2122, USA

Forms of Semantic Specification

Carl A. Gunter
University of Pennsylvania

The way to specify a programming language has been a topic of heated debate for some decades and at present there is no consensus on how this is best done. Real languages are almost always specified informally; nevertheless, precision is often enough lacking that more formal approaches could benefit both programmers and language implementors. My purpose in this column is to look at a few of these formal approaches in hope of establishing some distinctions or at least stirring some discussion.

Perhaps the crudest form of semantics for a programming language could be given by providing a compiler for the language on a specific physical machine. If the machine architecture is easy to understand, then this could be viewed as a way of explaining the meaning of a program in terms of a translation into a simple model. This view has the advantage that every higher-level programming language of any use has a compiler for at least one machine and this specification is entirely formal. Moreover, the specification makes it known to the programmer how efficiency is best achieved when writing programs in the language and compiling on that machine. On the other hand, there are several problems with this technique, and I do not know of any programming language that is really specified in this way (although I'm told that such specifications do exist). First of all, the purpose of a higher-level programming language is usually to provide some mediation between the way a programmer would like to provide instructions and the way a machine needs to have them. If the programmer is forced to think of the meanings of his programs in terms of how they will be compiled, then most of the advantage of having a higher-level language is lost. Second, a description of how to compile the language on architecture X may not be convenient for understanding the best way to compile it for architecture Y. The temptation would be to simulate X on Y, but this may not be efficient and an attempt to implement it by some more direct means must cope with the question of what are the invariants that must be preserved. For this, the X compiler alone will provide no clue.

Table 1: ALGOL 60 semantics for assignments.

4.2.2. Examples

$$\begin{aligned} s &:= p[0] := n := n+1+s \\ n &:= n+1 \\ A &:= B/C-v-q\times S \\ S[v,k+2] &:= 3-\mathit{arctan}(s\times zeta) \\ V &:= Q>Y\wedge Z \end{aligned}$$

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 4.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1 Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2 The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

One approach to resolving this problem is to give a compiler for the language to an *abstract machine* rather than a physically existent piece of hardware. If this abstract machine is cleverly chosen, then it may be possible to simulate it on a wide range of concrete (actual) machines in an acceptably efficient manner. This will aid the portability of the language and, if the abstract machine is easy to understand, it will free implementers from the need to deal with a possibly more complex compilation for a machine they do not need to know about. Moreover, a programmer could safely think in terms of the abstract machine and still write efficient programs. Now, the notion of an 'abstract machine' could correspond to just about anything, but the nature of this objective suggests that an abstract machine must be fairly similar to the architectures on which it is to be implemented. In short, an abstract machine is a least common denominator for the class of machines over which it is an abstraction.

The problem with abstract machines, therefore, is that they may be abstract enough for some, but still too low-level for others. Every programmer who uses a higher-level language must understand its semantics at *some* level of abstraction. A compiler for an abstract machine is likely to be hard to understand in detail, even for the clearest language and simplest abstract machine. Hence programmers ordinarily understand the semantics of a language through examples, intuitions about the underlying machine model, and some kind of *informal* semantic description. By way of illustration, an excerpt from one such specification appears in Figure 1. The example is

Table 2: Syntax for a simple imperative programming language.

I	$\in$	**identifier**
N	$\in$	**numeral**
B	$::=$	**true** $\mid$ **false** $\mid$ B **and** B $\mid$ B **or** B $\mid$ **not** B $\mid$ $E < E \mid E = E$
E	$::=$	$N \mid I \mid E + E \mid E * E \mid E - E \mid -E$
C	$::=$	**skip** $\mid$ $C;C \mid I := E \mid$ **if** B **then** C **else** C **fi** $\mid$ **while** B **do** C **od**

taken from page 10 of the Algol 60 revised report [Nau63] and describes the semantics of the assignment statements in that language. The example is not chosen to point out any particular strength or failing of such informal semantic descriptions, but merely to give a good example of one. In the best circumstances, an experienced programmer's intuition about the meanings of the phrases fills in any omissions in the description, and it can be viewed as a simple, succinct, and readable account of the meanings of language constructs. At worst, however, the description can be fatally ambiguous or misleading and programming errors or compiler implementation errors could result.

Research in the semantics of programming languages has sought to find alternatives to low-level abstract machines and informal, unrigorous semantic descriptions. One of the earliest ideas for another approach was that of a *meta-circular evaluator* in which the meaning of a language construct of language L is explained in terms of an interpreter written in language L itself. Such a program can be illuminating, but it cannot explain all of the details about the semantics of the language and must therefore serve as an auxiliary form of semantic description. Another, somewhat idealistic, approach to the semantics of a programming language is to see it as a vocabulary for describing elements of a mathematical model. For example, a program might be viewed as a term in an algebra with an appropriate signature and modelled by a standard choice of algebra having that signature. Ideas such as this date back to the late 1960's, but, before discussing it in more detail, I would like to look at a more recent form of specification.

The syntax for a simple language appears in Table 2. The semantics of the language could no doubt be described quite clearly using computerese, but it is not much harder to give a formal semantics for this language. Our semantics will be defined in terms of an abstract representation of a machine memory as an assignment of integer values to identifiers of the language:

$$\textbf{mem} = \textbf{identifier} \rightarrow \mathbf{Z}.$$

Assume that semantic functions for arithmetic expressions and boolean expressions are already known:

$$\begin{array}{l} [\![B]\!] : \textbf{mem} \rightarrow \{\mathsf{true},\ \mathsf{false}\} \\ [\![E]\!] : \textbf{mem} \rightarrow \mathbf{Z} \end{array}$$

(It can be assumed—for the sake of uniformity—that these semantic functions were also defined by a means similar to that being used here for commands.) The meaning $[\![C]\!]$ of a command C will be defined as a partial function from **mem** (input) to **mem** (output). We will define the semantic function in terms of a concept of a machine configuration consisting of a (possibly empty) control part and a data part (representing the current memory). More precisely, a *configuration* is either

- a pair (C, s) consisting of a command C and a memory s, or
- a memory s.

Rules for evaluating a program of the simple imperative language are given in terms of a binary relation $\rightarrow$ on configurations; this is defined to be the least relation that satisfies the rules in Table 3. In the transition rules for assignment, the evaluation of the command results in a new memory in which the value associated with the identifier I is bound to the value of the expression E in memory s. The notation used there is defined as follows for a value $n \in \mathbf{Z}$:

$$s[n/I](J) = \begin{cases} n & \text{if } I \equiv J \\ s(J) & \text{otherwise} \end{cases}$$

Now, let $\rightarrow^+$ be the transitive closure of the relation $\rightarrow$ (that is, $\rightarrow^+$ is the least transitive relation that contains $\rightarrow$).

Lemma 1 *If $\gamma \rightarrow^+ s$ and $\gamma \rightarrow^+ s'$ for a configuration γ, then $s = s'$.*

The meaning of a program C can now be defined by:

$$[\![C]\!]s \simeq \begin{cases} s' & \text{if } (C, s) \rightarrow^+ s' \\ \text{undefined} & \text{otherwise} \end{cases}$$

where the symbol $\simeq$ is being used rather than $=$ to emphasize that $[\![C]\!]$ is a *partial* function which may be undefined on some memories s.

This semantic description now gives rise to a set of equations between programs if we take C to be equivalent to C' when $[\![C]\!] = [\![C']\!]$. It is a *virtue* of this equivalence that it ignores important

Table 3: Transition semantics for a simple imperative programming language.

$$(\mathbf{skip},\ s) \rightarrow s$$

$$\frac{(C_1, s) \rightarrow (C_1', s')}{(C_1; C_2, s) \rightarrow (C_1'; C_2, s')}$$

$$\frac{(C_1, s) \rightarrow s'}{(C_1; C_2, s) \rightarrow (C_2, s')}$$

$$(I := E, s) \rightarrow s[[\![E]\!]s/I]$$

$$\begin{array}{ll}
(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s) \rightarrow (C_1, s) & \text{if } [\![B]\!]s = \mathsf{true} \\
(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s) \rightarrow (C_2, s) & \text{if } [\![B]\!]s = \mathsf{false}
\end{array}$$

$$\begin{array}{ll}
(\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, s) \rightarrow s & \text{if } [\![B]\!]s = \mathsf{false} \\
(\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, s) \rightarrow (C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, s) & \text{if } [\![B]\!]s = \mathsf{true}
\end{array}$$

aspects of a program such as its efficiency. It is this separation of concerns that makes it possible to develop a theory of program transformations; we could not say when the replacement of one program by another is legitimate without saying what property is to be preserved under such a replacement.

The form of semantics I have just described is usually called a *structural operational semantics* or SOS.[1] It is sometimes also called a 'Plotkin style' operational semantics because of an influential DAIMI technical report of Gordon Plotkin [Plo81] and several papers in which he used this form of specification. I prefer to use the term *transition semantics* for the kind of operational semantics given in Table 3 since 'transition' is more descriptive in this context than 'structural' or 'Plotkin-style'. The terminology is also familiar from the use of similar systems in the study of process algebras (see [Hen88] for example). On the other hand, this more general term has the drawback of including a range of operational semantics—such as those for automata and Petri nets—that seem quite different from the semantics in Table 3.

In looking at the transition semantics it is natural to wonder if it is possible to describe the relation $(C, s) \rightarrow^+ s'$ more directly. Rather than using rules to describe transition steps $\rightarrow$, why not use rules to describe $\rightarrow^+$ itself? This is indeed possible and one way of doing it appears in Table 4. The desired connection can be described precisely:

[1] I have sometimes also seen the term struct*ured* operational semantics. Is this terminology meant to suggest that other forms of operational semantics are unstructured?

Table 4: Natural semantics for a simple imperative programming language.

$$(\mathbf{skip},\ s) \Downarrow s$$

$$\frac{(C_1, s) \Downarrow s' \qquad (C_2, s') \Downarrow s''}{(C_1; C_2, s) \Downarrow s''}$$

$$(I := E, s) \Downarrow s[[\![E]\!]s/I]$$

$$\frac{[\![B]\!]s = \mathsf{true} \qquad (C_1, s) \Downarrow s'}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s) \Downarrow s'}$$

$$\frac{[\![B]\!]s = \mathsf{false} \qquad (C_2, s) \Downarrow s'}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s) \Downarrow s'}$$

$$\frac{[\![B]\!]s = \mathsf{false}}{(\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, s) \Downarrow s}$$

$$\frac{[\![B]\!]s = \mathsf{true} \qquad (C, s) \Downarrow s' \qquad (\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, s') \Downarrow s''}{(\mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}, s) \Downarrow s''}$$

Lemma 2 *For any program C and memory s, $(C, s) \to^{+} s'$ if, and only if, $(C, s) \Downarrow s'$.*

This approach to operational semantic description is very popular now and has many appealing characteristics which I will discuss briefly below. It is sometimes called a 'structural operational semantics' since the rules follow the structure of the terms of the language, just as for the transition semantics. This clash of nomenclature can be confusing though, since the binary relations $\to$ and $\Downarrow$ are very different from one another. I do not know full details of the history of the use of the $\Downarrow$ form of semantic specification. It is largely avoided by Plotkin in [Plo81]. Here is a short passage from that report discussing the tradeoff involved in directly establishing rules for whole derivations as opposed to axiomatizing via sequences of steps:

> It could well be argued that our formalization is not really that direct. A more direct approach would be to give rules for the transition sequences themselves (the evaluations). For the intuitive specification refers to these evaluations rather than any hypothetical atomic actions from which they are composed. However, axiomatizing a step is intuitively simpler, and we prefer to follow a simple approach until it leads us into such difficulties that it is better to consider whole derivations.

The $\Downarrow$ form of semantic description does appear in an article of Martin-Löf (see page 547 of [Mar79]):

> To execute $c(a)$, first execute c. If you get $(\lambda x)b$ as a result, then continue by executing $b(a/x)$. Thus $c(a)$ has value d if c has value $(\lambda x)b$ and $b(a/x)$ has value d.

This is the call-by-name rule with 'has value' for the $\Downarrow$ relation. The rule for call-by-value in this form appears in [Plo75] as part of the discussion there of Landin's SECD machine. Such a semantics is sometimes termed a *natural semantics* and has been carefully studied and used by Gilles Kahn and others (see [DDDK86] for example). There are those who take exception to this term (is it really the most natural form of semantic description?) but, in its favor, there is also a similarity between derivations of $\Downarrow$ relations and proofs in natural deduction. When the binary relation $M \Downarrow V$ is instead the relation $M : t$ between a term and its type, then the connection with natural deduction has some technical basis (the Curry-Howard correspondence). Those who have attempted to find another terminology seem to leap from the frying pan into the fire by using vague and misleading terms like 'relational semantics'. For practical applications, the use of natural semantics is so robust and easy to work with that it has been used to provide a *formal* semantic specification for Standard ML [MTH90, MT91].

Now, thinking as a logician, let me note that the function $[\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!]$ was *not* explicitly defined from the functions $[\![B]\!]$ and $[\![C]\!]$ using a semantic function $\mathcal{W}$ and an equation such as:

$$[\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!] = \mathcal{W}([\![B]\!],\ [\![C]\!]).$$

This is inconsistent with the way a semantics is ordinarily given for an object language in logic. Consider, for example, the way Tarski's semantics of first order logic is given. If ϕ and ψ are first order formulae and ρ is an assignment of meanings (in the universe of the model) then $[\![\phi \wedge \psi]\!]\rho$ is true if, and only if, $[\![\phi]\!]\rho$ and $[\![\psi]\!]\rho$ are both true. For a variable x, $[\![\forall x.\ \phi]\!]\rho$ is true if, and only if, for every element a of the model, $[\![\phi]\!]\rho[a/x]$ is true. Other meanings are defined similarly.

Indeed, in research on natural language, this form of semantic description is also stressed. The idea that the meaning of the whole should be described in terms of the meanings of the parts using semantic functions is called *compositionality* and the idea dates back at least to the writings of Frege at the turn of the century. From a computer science engineering perspective, the goal is, at least in principle, a very desirable one. Compositionality can be an excellent property for controlling complexity (although attempts to achieve compositionality can also *introduce* complexity). So, let us see how far we can go in getting a compositional semantics from what we have done so far in Tables 3 and 4.

As before, let us treat the meaning of a command C as a partial function $[\![C]\!]$ on memories. Clearly $[\![\textbf{skip}]\!]s = s$. Sequencing is interpreted by composition of partial functions, $[\![C_1; C_2]\!]s =$

$[\![C_2]\!]([\![C_1]\!]s)$ and assignment is 'updating', $[\![I := E]\!]s = s[[\![E]\!]s/I]$. We interpret branching by

$$[\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}]\!]s = \begin{cases} [\![C_1]\!]s & \text{if } [\![B]\!]s = \mathsf{true} \\ [\![C_2]\!]s & \text{if } [\![B]\!]s = \mathsf{false} \end{cases}$$

So far, so good. However, when we write down the intuitive explanation of the meaning of the while loop a problem appears:

$$[\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!]s = \begin{cases} s & \text{if } [\![B]\!]s = \mathsf{false} \\ [\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!]([\![C]\!]s) & \text{if } [\![B]\!]s = \mathsf{true} \end{cases}$$

In particular, the expression **while** B **do** C **od**, which we are trying to define, appears on *both* sides of the equation. What we mean is that $[\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!]$ is some form of *canonical fixed point* of this equational characterization. To put it another way, it is a fixed point of the functional $F : [\textbf{mem} \rightharpoonup \textbf{mem}] \to [\textbf{mem} \rightharpoonup \textbf{mem}]$ where

$$F(f)(s) = \begin{cases} s & \text{if } [\![B]\!]s = \mathsf{false} \\ f([\![C]\!]s) & \text{if } [\![B]\!]s = \mathsf{true} \end{cases}$$

Now $[\![\textbf{while } B \textbf{ do } C \textbf{ od}]\!] = \mathsf{fix}(F)$ really *is* a compositional description because the definition of F was made in terms of $[\![C]\!]$ and $[\![B]\!]$ only.

But how do we know that such a fixed point even exists? If it does, is there a canonical choice of such fixed point that will match the meaning that we gave with our operational semantics above? These are the central questions that the theory of semantic domains and its associated techniques for relating operational and fixed-point semantics are meant to address. There is an elegant and simple mathematical theory that provides the necessary fixed-points as elements of certain kinds of spaces (called domains). The spaces typically used may be viewed as partially ordered sets or (sometimes) as metric spaces or topological spaces. This form of semantic description has sometimes been called 'mathematical' semantics. It has its origins in the work of Scott and Strachey in the late 1960's [SS71, Sto77]. I dislike the defining term 'mathematical', since the operational descriptions above (transition semantics and natural semantics) are themselves mathematically rigorous and abstract. It has also been called 'compositional' semantics for the reasons I outlined above. I prefer to call it a 'fixed-point' semantics since that provides a helpful description of the primary tool that underlies the method.

The most common term for descriptions in the style of Scott and Strachey (which I've rarely heard called a 'Scott-Strachey-style semantics') is *denotational semantics*. With the examples we have at hand, it is hard to see why this term distinguishes fixed-point semantics from operational semantics. It appears to me that the equation

$$[\![C]\!]s \simeq \begin{cases} s' & \text{if } (C, s) \to^{+} s' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Table 5: Call-by-name evaluation for **PCF**.

$$\mathbf{0} \Downarrow \mathbf{0} \qquad \mathbf{true} \Downarrow \mathbf{true} \qquad \mathbf{false} \Downarrow \mathbf{false}$$

$$\frac{M \Downarrow \mathbf{0}}{\mathbf{pred}(M) \Downarrow \mathbf{0}} \qquad \frac{M \Downarrow \mathbf{succ}(V)}{\mathbf{pred}(M) \Downarrow V} \qquad \frac{M \Downarrow V}{\mathbf{succ}(M) \Downarrow \mathbf{succ}(V)}$$

$$\frac{M \Downarrow \mathbf{0}}{\mathbf{zero?}(M) \Downarrow \mathbf{true}} \qquad \frac{M \Downarrow \mathbf{succ}(V)}{\mathbf{zero?}(M) \Downarrow \mathbf{false}}$$

$$\lambda x : t.\ M \Downarrow \lambda x : t.\ M \qquad \frac{M \Downarrow \lambda x : t.\ M' \qquad [N/x]M' \Downarrow V}{M(N) \Downarrow V}$$

$$\frac{M_1 \Downarrow \mathbf{true} \qquad M_2 \Downarrow V}{\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \Downarrow V} \qquad \frac{M_1 \Downarrow \mathbf{false} \qquad M_3 \Downarrow V}{\mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 \Downarrow V}$$

$$\frac{[\mu x : t.\ M/x]M \Downarrow V}{\mu x : t.\ M \Downarrow V}$$

indicates that C is the denotation for a particular function from memories to memories. In the case of a natural operational semantics, I have sometimes heard it claimed that the relation $M \Downarrow V$ indicates that V is the *meaning* or *value* of M and hence M is a denotation for V. Thus natural operational semantics is denotational. While this intuition is an attractive one, I believe that it breaks down in non-trivial cases. To illustrate the point, let me introduce a slightly more substantial toy programming language.

The syntax for **PCF** is given as follows:

$$\begin{array}{rcl} t & ::= & \mathbf{num} \mid \mathbf{bool} \mid t \to t \\ M & ::= & \mathbf{0} \mid \mathbf{succ}(M) \mid \mathbf{pred}(M) \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{zero?}(M) \mid \\ & & x : t \mid \lambda x : t.\ M \mid M(M) \mid \mu x : t.\ M \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M \end{array}$$

where x ranges over a primitive syntax class of variables. This language is essentially the one introduced by Gordon Plotkin in [Plo77] modulo a recasting of numerals and a binding construct $\mu x : t.\ M$ in place of a constant Y. A natural operational semantics for the terms of the language can be found in Table 5. The expression $[M/x]N$ denotes the result of substituting term M for free occurrences of variable x in term N with the usual conventions insuring that free variables of M are not captured by bindings of N. If we read $M \Downarrow V$ as an assertion that V is the value of M, then the rules tell us the meaning of M in terms of its structure. This assignment of meaning is not compositional though, since the rule for recursion does not define the meaning for $\mu x : t.\ M$ in terms of the meaning of a subexpression but instead in terms of the possibly more complex expression $[\mu x : t.\ M/x]M$. On the other hand, the fixed-point semantics of **PCF** *does* define the meanings of **PCF** programs compositionally using a fixed-point operator to define the meaning of $\mu x : t.\ M$ in terms of the meaning of M.

My primary interest in introducing this example was not to remake the point about compositionality—the point here concerns *denotation.* If $M \Downarrow V$ is supposed to indicate the meaning of M as V, then the semantics in Table 5 is a very disappointing one. Note, in particular, that the value of $\lambda x : t.\ M$ is simply $\lambda x : t.\ M$. This form of 'denotation' in which a term denotes itself might be termed 'fully concrete' (by contrast with the notion of a fully abstract fixed-point semantics) and does not directly provide any useful idea of identification between programs of higher type. This notion must come through some other means in which the meaning of a function is given by the rule for application with the idea that $\lambda x : t.\ M$ defines a function from input values to output values. If t happens to be itself a higher type, this view is problematic since a value of type t needs to be a function rather than an expression if we are to get a reasonable theory.

I *am not* arguing that one cannot use natural operational semantics to reason about programs; indeed, it can be very useful for proving certain properties. I *am* arguing that, for higher types, the operational rules in Table 5 do not really say that M denotes V if $M \Downarrow V$ (unless denotation is here being taken in a trival sense) and I see no immediate way of repairing this short-coming by, say, a reshuffling of the form of the rules. To get a semantics in which a program is associated via a semantic function with an element of an abstract realm of mathematical meanings requires some further elaboration of the significance of the operational rules.

Despite these equivocations on what is denotational, it is certainly the case that natural semantics and fixed-point semantics have great similarity when viewed as interpreters. It is often remarked that the descriptive component of a denotational semantics, as one finds in references such as [Gor79], is like a λ-calculus evaluator for the language being interpreted. Indeed, the functional fragment of a language like ML is excellent for writing interpreters for other languages. On the other hand, the natural semantics for a language is like an interpreter written with Horn clauses and can be evaluated by a form of proof search familiar from logic programming. In my experience, such semantic descriptions do not use unification beyond pattern matching or do any backtracking, so the interpreter involved is simpler than that required for prolog.

The traditional view of fixed-point semantics as compositional semantics has been challenged by recent developments in the semantics of programming languages. In languages such as **PCF,** as described above, there is a close relationship between the syntactic structure of a term and the typing rule that applies to the term. For example, there is exactly one typing rule for application, which has the following form:

$$\frac{H \vdash M : s \to t \qquad H \vdash N : s}{H \vdash M(N) : t}$$

where H is a type assignment associating types with free variables in $M(N)$. If the types of M and N are uniquely determined by the terms, then so too is the type t of the application. Since the meaning of $M(N)$ is given by a fixed-point semantics in terms of the meanings of M and N, we may think of the semantics as following the structure of the *typing rules* just as well as thinking of them as following the structure of the term. In the case of **PCF**, this simply provides two ways of thinking of the same inductive definition. But for most of the calculi that involve *subtyping*, this coincidence breaks down. The problem can usually be traced to the *subsumption* rule:

$$\frac{M : s \qquad s < t}{M : t}$$

where $s < t$ means that s is a subtype of t. If subtyping is non-trivial, then some terms will have more than one type. But, more significantly, in most calculi with such a rule, a given typing judgement $M : t$ may have *more than one proof.* In particular, there is a real difference between defining the meaning of a term inductively in its structure and defining its meaning inductively in terms of a proof of a typing judgement for it. Moreover, if an assignment of meaning follows the structure of a proof, then it is necessary to show that the meaning attached to a term is the same regardless of the choice of the proof. This is called the *coherence problem* for the semantic function. The reader can find some discussion of this in [BCGS91]; there has been substantial progress on understanding the role of coherence results in calculi without subtypes as well.

Discussions of the kinds of formal specification techniques for programming languages often divide these into three categories: operational (including structural), denotational (fixed-point), and *axiomatic.* The term 'axiomatic semantics' sounds like an oxymoron to a logician of course, but it sometimes has a certain legitimacy. Hoare triples, for example, embed the programming language in a logic whose semantics is understood and thereby indirectly impart meaning to programs. It is worth noting that such an axiomatic semantics is usually given *compositionally* using rules that explain properties of programs in terms of properties enjoyed by component parts. On the other hand, there are cases where the rules are not compositional, especially for logics of concurrent programs.

Let me turn now to a discussion of abstract machines and how they are related to other forms of operational semantics. The most familiar example of an abstract machine in the context of functional programming is Landin's SECD machine [Lan64, Lan65] for call-by-value evaluation of the untyped or typed λ-calculus.[2] As I discussed earlier for abstract machines in general, a characteristic of the SECD machine is the similarity of its basic instructions to those of standard

[2] It is also possible to adapt the call-by-value machine to call-by-name execution, thereby more closely matching the evaluation for **PCF** in Table 5.

computer architectures. The SECD machine is described in dozens of sources, so let just very briefly recall a variant of it here. A state of the machine is a four tuple (S, E, C, D) where S is the *Stack*, E is the *Environment*, C is the *Code* and D is the *Dump*. The action of the machine is described by a collection of transitions driven by the next operation in the code list C. Instructions in the code list can be formed from the λ-terms together with a primitive instruction *ap* for application. The stack component S is simply a stack of closures, *i.e.* terms paired with environments. An environment E is a partial function from variables to closures. The dump is either *nil* or it is an (S, E, C, D) tuple. The following transition rules cover all of the cases that can arise starting from a tuple $(\mathit{nil}, \emptyset, M, \mathit{nil})$ where M is a closed λ-term:

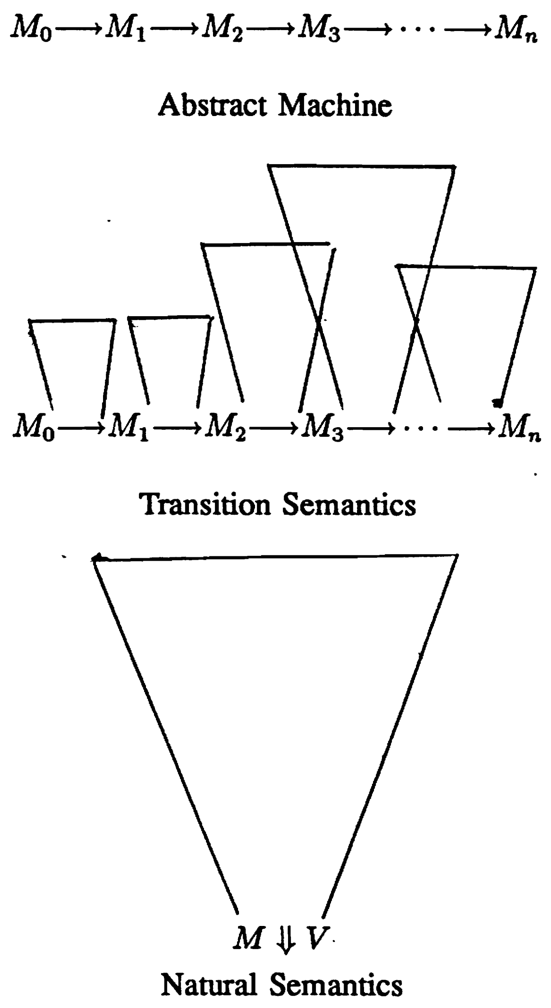
$$\begin{array}{l}
(Cl :: S, E, \mathit{nil}, (S', E', C', D')) \rightarrow (Cl : S', E', C', D') \\
(S, E, x :: C, D) \rightarrow (E(x) :: S, E, C, D) \\
(S, E, (\lambda x.\ M) :: C, D) \rightarrow (((\lambda x.\ M), E) :: S, E, C, D) \\
(((\lambda x.\ M), E') :: Cl :: S, E, \mathit{ap} :: C, D) \rightarrow (\mathit{nil}, E'[Cl/x], M, (S, E, C, D)) \\
(S, E, (M(N)) :: C, D) \rightarrow (S, E, N :: M :: \mathit{ap} :: C, D)
\end{array}$$

The double colon :: denotes cons for lists here; both S and C are being represented as lists.

The thing to notice about this SECD semantics is that there are no rules with hypotheses, just transitions. The representation of a term includes its own stacks for holding intermediate values during computation. By contrast, at the opposite extreme, a natural operational semantics suppresses as much detail about this aspect of computation as possible. Instead of using rewriting as the driving force of computation, it uses search for a proof. The transition semantics provides a sort of half-way house in which computation is a combination of rewriting and proof search. If we think of rewriting as a 'horizontal' kind of computation (represented as a sequence of arrows from left to right) and search as a 'vertical' kind of computation (represented with a tree), then the difference between abstract machines, transition semantics, and natural semantics can be graphically illustrated as in Figure 1.

A detailed introductory exposition of many of the relationships I have discussed here for the Simple Imperative Language in Table 2 can be found in the new book of Hanne and Flemming Nielson [NN91]; similar languages are examined in [Ten91]. My own book [Gun92] focuses on the language **PCF** and its extensions. There is a growing literature on the relationships between various forms of operational semantics. Research of John Hannan [Han91, HM92] has helped to clarify some of the distinctions between these different forms of operational semantics and I refer the reader to his work for a further treatment. There is also an approach to operational semantics that focuses on evaluation contexts; it has been used to study the relationships between different forms of semantic specification in languages that include control constructs [FFKD87] and the approach seems to work well for proofs that type systems exclude certain kinds of errors [WF91].

Before closing, let me make a comment on a methodology implicit in the discussion above: the use of 'toy' languages such as the simple imperative programming language. Can we really


$M_0 \longrightarrow M_1 \longrightarrow M_2 \longrightarrow M_3 \longrightarrow \cdots \longrightarrow M_n$
$M_0 \longrightarrow M_1 \longrightarrow M_2 \longrightarrow M_3 \longrightarrow \cdots \longrightarrow M_n$
$M \Downarrow V$

Abstract Machine

Transition Semantics

Natural Semantics

Figure 1: Three forms of operational semantics.

learn anything about the semantics of 'real' languages by studying rarefied theoretical ones? The question is a serious one, because the control of complexity is a key element of a clear description of a language. Two methods that look equally reasonable for a language with a grammar less than a half-page long may not be so for a language 20 times as complex. Do all of the semantics discussed above scale up equally well? Opinions seem to vary on the matter, but I think most researchers in semantics will agree that complexity is a serious problem for the formal semantics of a substantial language—no matter what approach is used.

Nevertheless, there is a strong case for the study of 'toy' languages aside from the obvious fact that the complete semantics for a 'real' language will not fit in a journal-length article. When there is a focus on conceptual issues, a simple language can illustrate the basic question in isolation where it can be studied in some depth. In a real language, one has trouble seeing the forest because of all the trees. I believe that programmers implicitly use what they understand about 'toy' sublanguages as a way of reasoning about their very real programs. If a component of a program has been written in a small fragment of a much larger language, then one can employ reasoning principles that apply to that small fragment. For example, if a procedure in a larger program is written in a purely functional fragment of the programming language, then certain reasoning principles will be available for that procedure. The full language specification is likely to hide these principles under the bulk of state and control features usually included in a 'real' language. In this way one trys to proceed by conceptual layers in reasoning from the simple to the complex.

In writing on the topic of forms of semantic specification, one is forced into a certain immodesty. Since there are so many different approaches, it seems that the ones mentioned are distinguished unfairly from those omitted. For example, there are such techniques as the Vienna Definition Language (VDM) [Weg72], the action semantics of Peter Mosses (see the references in [Mos90]), the evolving algebras of Yuri Gurevich [Gur88] and many others. In the end, I believe that no single approach to semantic specification is best for all tasks. The practical application of formal specifications relies on a variety of specification techniques, an understanding of which purposes are best served by each, and a foundational theory of the relationships between them. I hope that the discussion in this column can provide some impetus for further comparison and analysis of the varieties of semantic specification techniques now available.

I would like to express my appreciation to the people who made many helpful comments and suggestions based on an earlier draft of this note: Peter Baumann, Bard Bloom, Matthias Felleisen, Yuri Gurevich, James Huggins, Gilles Kahn, Gordon Plotkin, Jon Riecke, and Robert Tennent. Not everyone agreed with me on every point (or agreed with one another, for that matter!), but their views contributed substantially to the perspective of my exposition.

References

[BCGS91] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation.*, 93:172–221, 1991.

[DDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: mini-ML. In *Symposium on LISP and Functional Programming*, pages 13–27, ACM, 1986.

[FFKD87] M. Felleisen, D. P. Friedman, E. E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[Gor79] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer, 1979.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* To be published by the MIT Press, 1992.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Böger, editor, *Trends in Theoretical Computer Science*, pages 1–57, Computer Science Press, 1988.

[Han91] J. Hannan. Making abstract machines less abstract. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, Springer-Verlag LNCS, 1991.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HM92] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 1992. To appear.

[Lan64] P. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[Lan65] J. Landin. An abstract machine for designers of computing languages. In *IFIP Congress*, pages 438–439, North-Holland, 1965.

[Mar79] Per Martin-Löf. Constructive mathematics and computer programming. In *International Congress for Logic, Methodology and Philosophy of Science*, pages 538–571, North-Holland, 1979.

[Mos90] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 577–632, North Holland, 1990.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Nau63] P. Naur. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6:1–7, 1963.

[NN91] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction for Computer Science.* To be published by the Wiley Press, 1991.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo81] G. D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report FN-19, Computer Science Department, Aarhus University, Ny Munkegade—DK 8000 Aarhus C—Denmark, 1981.

[SS71] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Computers and Automata*, pages 19–46, Polytechnic Institute of Brooklyn Press, 1971.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, 1977.

[Ten91] R. D. Tennent. *Semantics of Programming Languages.* Prentice-Hall, 1991.

[Weg72] P. Wegner. The Vienna Definition Language. *ACM Computing Surveys*, 1:5–63, 1972.

[WF91] A. K. Wright and M. Felleisen. *A syntactic approach to type soundness.* Technical Report COMP TR91-160, Department of Computer, Rice University, 1991.