Mathematical concepts in programming language semantics

by DANA SCOTT

Princeton University
Princeton, New Jersey

INTRODUCTION

In mathematics after some centuries of development the semantical situation is very clean. This may not be surprising, as the subject attracts people who enjoy clarity, generality, and neatness. On the one hand we have our concepts of mathematical objects (numbers, relations, functions, sets), and on the other we have various formal means of expression. The mathematical expressions are generated for the most part in a very regular manner, and every effort is made to supply all expressions with denotations. (This is not always so easy to do. The theory of distributions, for example, provided a non-obvious construction of denotations for expressions of an operational calculus. The derivative operator was well serviced, but one still cannot multiply two distributions.)

The point of such activity is that the formal rules of the calculus of expressions allow solutions of problems to be found and give us ways of checking correctness of proposed solutions. It is by no means the case that mathematical inspiration is thus reduced to automatic algebra, but the possibility of formal manipulation is a great help. Not only can we record useful lemmas, but a precise language is essential for teaching and communication.

It is of course possible to pick formalisms out of thin air and to ask people to accept their usefulness on mystical grounds. (The operational calculus was developed a little that way.) There is no denying that clever guesswork can be very successful, but ultimately it is fair to ask the mystic just what he is talking about. A way to counter that question is to show how the concepts being symbolized in the new language are explained in terms of familiar notions. To do this we can try either to correlate directly denotations definable in familiar terms with the expressions of the new language or to show how to translate the new expressions out of every context in which they might occur leaving only what is familiar remaining. If you can do it the first way, you can do it the second way, obviously; the converse is not obvious though. A contextual translation scheme may be very sensitive to the context; a transformation appropriate for one occurrence of an expression may not work elsewhere.

The negative integers can be used as a good example here. The rules of algebra are such that all minus signs can be eliminated from an equation by multiplying out the formal polynomials and then using transposition. (Different rules are used for different contexts, note.) If it is the question of an algebraic law with variables, we use this trick: Suppose we wish to assert that the equation \( f(x) = g(x) \) holds for all \( x \) at both positive and negative values. Of course \( f(x) = g(x) \) holds for all non-negative \( x \). But so does \( f(-x) = g(-x) \). In other words one equation has been equivalently translated into two, in which the variable can be restricted to the more familiar range. Now Descartes may have been willing to do algebra this way, but we would no longer consider it reasonable. We have developed our theory of numbers so as to include both positive and negative quantities, and we have gone on to imaginary and complex numbers with great gains in understanding. In the new algebra, expressions are directly meaningful and no translations are required. Nevertheless we prove that the new algebra is consistent by constructing a model out of familiar stuff. Complex numbers were fairly easy to handle; quaternions were harder. You all know examples and know that mathematicians are busy every day finding other weird structures. Whether the results deserve other than a mother's love is another question.

What does this discussion have to do with programming languages? Very much I feel. Programming languages have introduced a new dimension into the problem of semantics. In the first place there has been an explosion in the size and complexity of the expressions that must be considered. In view of the com-
lications, every effort is made to allow the writer of the program to keep tiresome details implicit: some controls can be written in, but generally it is the compiler that will take care of the boring parts. Now many, many compilers have been written—even students can do it these days. A compiler can be viewed as a translator from the source language (unfamiliar) into machine language (familiar). Why cannot we say, then, that the semantical problems of programming language are solved? The answer is, I believe, that every compiler (if it is meant for a real machine) is a compromise. There must be as many compilers as there are machines, each reflecting limitations and peculiarities of the given machine. Should not the ideal of language definition be machine independent?

Certainly the introduction of abstract machines does not settle this difficulty of a multiplicity of translation schemes. After all even for an abstract machine the compiler writer must choose specific ways of keeping track of the order or depth of nesting of computation initiations. There may be different ways of doing this, all of which lead to the same results. The variety of approaches even on an abstract machine may make it quite impossible to define in any meaningful way just what is a single computation step. Therefore, the idea of the state of the computation is not apparently at all well determined by the source language alone. When a specific compiler is fixed, however, we may very well want to investigate how it handles computation steps. But the question remains: do they have anything to do with the semantical definition of the original language? I think not.

Before pursuing the question of the proper level at which semantics enters, we should take note of a second dimension-expanding consequence of programming language study. Mathematical concepts generally have a rather static quality. True, mathematicians study flows on manifolds and many-body problems, but it seems to me that often the interest centers on global qualities or asymptotic behavior. What is peculiar in programming language—especially from a linguistic point of view—is that exactly the same symbols may possess a quite different import in different segments of a program. This is most apparent in the way variables (identifiers) are used. I need give no examples here; all of you understand why variables are not employed in the assignment statement in a “mathematical” way. The dynamic character of command execution requires a quite different treatment of variables, and this dynamic urge permeates the whole semantic analysis. The principal virtue of the programming language is that the compounding of the various operations need only be indicated in a rather schematic way, since the calculation is meant to be automatic in any case. But the calculation is not meant as a game or a joke. It is something that needs precision, so we must be able to “follow” what happens in some rather definite sense. What we are discussing is how definite this has to be in order to have a good semantical definition of a language.

Some years ago, Christopher Strachey began a project of trying to expose the semantics of a programming language as a series of mutually recursive functional equations. The particular complex of equations would be “syntax directed” in the sense that each clause of the recursive syntactical specification of the language would have a corresponding equation. The equations taken together would define the meaning of a program, because made to correspond to a program text would be a recursively defined function. This function was intended as a state-transformation function: the function that takes an initial state to the final state that results from executing the program. In many instances the notion of “state” used here could be taken to be “state of the memory” or some abstraction from that idea that make the definition machine independent.

The formalism Strachey used to state his equations was a modified version of the λ-calculus. There was an inherent difficulty in this approach: The λ-calculus at the time was just another symbolic calculus. The criteria for equivalence of expressions were rather weak, and it was not clear in which directions they should be expanded. The advantage of the λ-calculus was that it was “clean,” especially in its handling of variables, which was done along the usual “mathematical” lines. Thus it was felt that some progress had been made in explaining the unfamiliar in terms of the familiar. Still, the project was not quite complete.

One objection to the λ-calculus the present author had was that it possessed no mathematical semantics. It was another kind of operational calculus. True, the calculus had been proved consistent, but the consistency proof did not seem to give clear hints as to how to extend usefully the principles of equivalence of expressions. This uncertainty was particularly unsettling in view of the many inconsistent extensions that had been proposed. The main difficulty rested on the thoroughly type-free style of functional application and abstraction used in the λ-calculus, where, for example, a function could be applied to itself as an argument. And it was just this type-free character of the system that Strachey wanted to exploit.

During the fall of 1969 the author resided in Oxford in order to learn more about Strachey’s ideas on programming languages. After long, and sometimes rather heated, discussions on the role of mathematics in semantics, it slowly began to dawn on the author that
there might be an "objective" interpretation of the λ-calculus. As explained elsewhere,7,8 it was possible finally to combine ideas and techniques from many sources into a coherent theory of models for the λ-calculus. The key step was to find the right idea of an abstract domain (which turned out to be a kind of topological space) so that an appropriate notion of a function space would be a domain of the same kind. From that point it was only a matter of time until the question of whether a domain might exist that could be isomorphically identified with its own function space would arise. Fortunately the author was able to see how to effect the required mathematical construction, and a new approach to functional calculi was opened up.

This sounds most abstract and impractical, but the spaces constructed are actually no worse than the real numbers. The "irrational" elements can be explained as limits of infinite sequences in ways that are quite reminiscent of classical analysis.9 And what is more important proofs about the properties of the elements can be given inductively by "successive approximation" in a simple and natural way. Furthermore it is possible to discuss the effectiveness and computability of various elements and processes so as to tie in the abstractions with actual computation.

In this paper it will not be possible to discuss the mathematical details, though a few words will be inserted here and there to resolve puzzles about whether the method is reasonable.10 Instead it will be shown how the theory of these abstract domains can be applied to typical semantical problems, thus supplying the mathematical foundation needed for Strachey's equational approach. It is not claimed that the project for a mathematical semantics is at all completely finished, but the ideas have been found to be so flexible that we have gained full confidence that we can do serious model building incorporating features whenever the need for them is realized.11

LOCATIONS AND STORES

There are many levels of storage from core to disk to tape to card hopper to the mind of the user sitting at a console. Some of these are more addressable than others (and more reliable). Some require special handling, especially as regards allocation of space in preparation for future acts of storage. In the present discussion we shall not treat these subtleties, even though they are very important for good compiler operation. We shall simplify the presentation for the sake of illustration by making these assumptions:

- There are but two facets to storage: internal and external.
- The internal storage is addressed by elements of a homogeneous collection of locations.
- The contents stored at each location always may be read out (nondestructively), and at any time new contents may be read in updating (destroying) the old contents.
- The current state of the store allows a distinction between the area of active locations (with contents) and passive or free locations (without contents).
- At any time a new location may be selected from those free and placed in the active area.
- At any time a location may be lost or retired to the free list.
- One part of the external store allows reading in of fresh information (input).
- The other part of the external store allows writing of computed information (output).
- External reading and writing are independent operations; what is written cannot be read back unless it has been separately stored internally.

The statements of these rather natural assumptions have been given first in fairly precise yet intuitive language. The point of the present paper is to show how to "mathematize" such assumptions in terms of a model that can be incorporated into a full semantical definition. (No formal language has been discussed at this point, however.) To carry out the mathematical modelling, we postulate first the existence of several domains that will be interrelated by the model:

The basic domains

\[ L = \text{locations} \]
\[ V = \text{stored values} \]
\[ S = \text{states of the store} \]
\[ T = \text{truth values} \]

Possibly it would have been better to say "storable values" since not every element of \( V \) is actually stored. Just what these quantities are supposed to be will be left open until the next section. One has a variety of choices depending on the kind of language that is to be interpreted.

These domains alone do not specify the model because they have not yet been supplied with any structure. This can be effected by listing the operations and transformations that may be applied to the domains in

From the collection of the Computer History Museum (www.computerhistory.org)
various combinations:

The store operators

Contents: \( L \to [S \to V] \)
Update: \( LXV \to [S \to S] \)
Area: \( L \to [S \to T] \)
New: \( S \to L \)
Lose: \( L \to [S \to S] \)
Read: \( S \to V \times S \)
Write: \( V \to [S \to S] \)

Given any two domains \( D_0 \) and \( D_1 \), we write \( [D_0 \to D_1] \) for the space of all (admissible) functions from \( D_0 \) into \( D_1 \). (At this stage we do not need to worry just which functions are admissible.) We write \( f: D_0 \to D_1 \) to indicate that \( f \) is just such a function. What we have listed above are the functions that correspond to our intuitive ideas about the structure of the store. In order to explain how this works, the following variables will be used (with or without sub- and superscripts) restricted to the indicated domains:

\[ \alpha : L, \ \beta : V, \ \sigma : S, \ \tau : T \]

Suppose, then, \( \alpha \) is a particular location and \( \sigma \) is the current state of the store. The function value \( \text{Contents}(\alpha)(\sigma) \) is to be the \( V \)-value stored by \( \sigma \) at \( \alpha \). Conversely, let \( \beta \) be a given \( V \)-value. The pair \((\alpha, \beta)\) in \( LXV \) provides the desired information for updating the store. Therefore, when we write

\[ \text{Update}(\alpha, \beta)(\sigma) = \sigma' \]

we want \( \sigma' \), the transformed store, to be just like \( \sigma \) except in having \( \beta \) stored now in location \( \alpha \). Similarly \( \text{Area}(\alpha)(\sigma) \) is true or false according as \( \alpha \) is in the active area of \( \sigma \). The function \( \text{Lose}(\sigma) \) transforms \( \sigma \) into the store \( \sigma'' \) which is just like \( \sigma \) except that \( \alpha \) has been freed. Exactly what \( \text{Read} \) and \( \text{Write} \) do depends on the activities of the mad scientist at the console. Supposing, however, that he does not ruin the equipment, whenever \( \text{Read}(\sigma) \) is activated something will happen. This transformation can be recorded as a pair \((\beta', \sigma')\), where \( \beta' \) is a storable value, and \( \sigma' \) has the same internal state as \( \sigma \) did, but the external condition is made all eager and ready for the next input. In a similar way, the function \( \text{Write}(\beta') \) may be regarded as only affecting the external portion of the store.

The functional notation of mathematics has the gloss of precision. But so far the gloss is the only aspect introduced, since note that the above explanations of what the functions are supposed to do are just as informal as the original assumptions given in ordinary language. What must be avoided in this kind of work is the Fairthorne Hovercraft effect described as an "approach in which we use extremely sophisticated techniques and mathematical devices to skim at very high speed and great expense over the subject of concern without ever touching it."

Now in mathematical model building the way to touch ground is to find out how to formulate with precision the postulates that correspond to the intuitive assumptions. Notational precision is not enough—if there is no way of proving useful theorems. So far our postulates have been rather weak: nothing more than simply noting the functional characters (or logical types) of certain operators. The next step is to say exactly how these operators are connected.

What I wish to claim is that each of the intuitive assumptions previously informally indicated can be expressed rigorously by an equation. For example, after updating the store at a location, the contents of the location will be found to be what is expected and the location will be found to be active. This statement makes two equations as follows:

\[ \text{Contents}(\alpha)(\text{Update}(\alpha, \beta)(\sigma)) = \beta \]
\[ \text{Area}(\alpha)(\text{Update}(\alpha, \beta)(\sigma)) = \text{true} \]

These equations are meant to hold for all \( \alpha, \beta, \sigma \). What about locations \( \alpha' \) different from \( \alpha' \)? If \( \alpha' \neq \alpha \), we expect that:

\[ \text{Contents}(\alpha')(\text{Update}(\alpha, \beta)(\sigma)) = \text{Contents}(\alpha')(\sigma) \]

Similarly in the case of \( \text{Area} \). If we introduce the equality relation on \( L \):

\[ \text{Eq}: L \times L \to T \]

and conditional expressions, these four equations can be reduced to two in the obvious way.

In writing these equations, one puzzle is what to write in the cases where the functions are possibly undefined. A solution is to introduce a special value to correspond to this case. The symbol for this value adopted here is \( \bot \). We write \( f(x) = \bot \) to indicate that the function \( x \) is undefined at \( x \). (Other symbols in use are \( O, \omega, *, \) and \( U \).) This convention is useful in the main equation relating \( \text{Contents} \) to \( \text{Area} \):

\[ \text{Contents}(\alpha)(\sigma) = \text{if } \text{Area}(\alpha)(\sigma) \text{ then } \text{Contents}(\alpha)(\sigma) \text{ else } \bot \]
There is not enough room here to formulate all the necessary equations; thus, my claim must remain just that. But it does not seem to me that the claim is too implausible. Note, however, that these postulates on the character of the store would not completely specify everything. The exact nature of V is left open at this stage of the model building. Concerning L, all we would know is that there are infinitely many locations, because new ones can be produced indefinitely. Actually that is probably enough to know about L until one gets into problems of allocation. The input/output phase has also been left a little vague, but that is not necessarily bad either. If we can prove any theorems at all, it is better to prove them from fewer assumptions. Some assumptions are needed, nevertheless; and it requires experimentation to find just the right balance. Part of the thesis of this paper is that the mathematical style advocated allows for this experimentation to take place in a convenient way.

A LANGUAGE

There were no surprises in the discussion of storage in the previous section, and there are no surprises planned for this section either. All we shall do here is to indulge in some very modest language design. Any real innovation is saved for the section on semantics. And this is how it should be, otherwise we would not be giving solutions to natural problems. It is quite possible, though, that after the semantical approach has been appreciated language design practice will be altered. Strachey has argued for this, but the effects remain to be seen. In any case we hope to develop a semantical style within which the consequences of different design choices can be studied.

To start with our language will be divided into the usual syntactical categories:

The syntactical domains

- \( Id = \text{identifiers} \)
- \( Num = \text{numerals} \)
- \( Op = \text{(binary numerical) operations} \)
- \( Cmd = \text{commands} \)
- \( Exp = \text{expressions} \)

A feature of our method is that we treat these domains on a par with the basic and semantical domains. They are different, since they are constructed for syntactical purposes; but they are domains nevertheless. The following variables are used in restricted senses:

- \( \xi : Id, \nu : Num, \omega : Op, \gamma : Cmd, \varepsilon : Exp \)

The syntax for \( Id, Num, \text{and } Op \) will not be given since it is standard. The remaining two categories are defined as follows:

Commands

\[
\gamma ::= \varepsilon | \nu \cdot \omega | \nu \rightarrow \gamma_0, \gamma_1 | \nu \cdot \omega \]

\[
dummy | \gamma_0, \gamma_1 | \text{while } \nu \text{ do } \gamma |
\]

\[
\text{let } \xi = \varepsilon \text{ in } \gamma | \text{let } \xi = \varepsilon \text{ in } \gamma | \text{letrec } \xi \text{ be } \varepsilon \text{ in } \gamma |
\]

Expressions

\[
\varepsilon ::= \xi | \perp | \text{read} | \gamma \text{ result is } \varepsilon | \varepsilon_0 \rightarrow \varepsilon_1, \varepsilon_2 |
\]

\[
\varepsilon : T | \text{true} | \text{false} | \varepsilon_0 = \varepsilon_1 |
\]

\[
\varepsilon : N | \nu | \varepsilon_0 = \varepsilon_1 |
\]

\[
\varepsilon : C | \gamma |
\]

\[
\varepsilon : P | \lambda \xi. \varepsilon | \varepsilon_0 = \varepsilon_1 |
\]

\[
\varepsilon : R | \downarrow \varepsilon | \uparrow \varepsilon |
\]

Some comments are necessary: (i) There is an unpleasant conflict in this syntax between the use of words versus the use of symbols. The trouble is that there are not enough symbols—especially on keyboards—otherwise all constructs could be symbolized. But then one has to remember what the symbols are for, so it is better to use words—especially English words. But it takes so long to write out all those words. The situation is hopeless. (ii) There is an unpleasant ambiguity in this syntax. The same string could be parsed in several different ways. Necessarily, then, the meaning of a command or an expression will have to depend on the parse. But we will pretend in writing equations that everything can be determined from the string alone. If that worries you, please write in the parentheses (you are allowed to by the last clause of the definition). But no one can stand to write in all those parentheses. The situation is hopeless. But for the semantical theory it does not really matter. The style of language definition was chosen to give a few hints as to the meanings. But I hope there are some constructs that seem quite mysterious on a first reading.
Some explanation is in order. Take commands first. The first clause gives us assignment statements. Everyone knows what they mean—except what does it mean to have a compound expression on the left? Well, never mind. The next command is obviously intended to invoke output. That was easy. For the next clause it would have been better to write do ε. In this language commands can be set up and (their closures) stored away for later execution. This command means that ε should be evaluated out; and if the value is a command closure, then it should be done. If not, what then? Never mind. Next we have the conditional command (Boolean-valued expressions are possible). To execute dummy, do nothing—except waste time. The semicolon provides the means for sequencing, or composition, of commands. Next we have the while statement. Its effect is well-known. Finally there are three kinds of initialization in this language. But which is which?

An odd fact about many programming languages is that identifiers are used for many different purposes even though they form just one syntactical category. The context is expected to give the clue as to use. It is not such a bad idea. When one starts to write a program, the number of identifiers to be used is probably not known. To have to divide them up into various subcategories would complicate the language a great deal, and restrict the ways they could be chosen, by tiresome rules. They have only a local significance in any case, so some rough, informal conventions of programming style are sufficient to assure readability of programs.

The three types of initialization in this language illustrate three quite different uses of identifiers. In the first, we begin by evaluating ε. It will turn out to be either an L-value or a V-value. Whatever it is, we associate the same value with ξ and go on to execute γ under this assignment (keeping assignments to other identifiers the same as before, of course). In the second, we evaluate ε. If the value is a V-value, we are happy. If the value is an L-value, we find the contents of the current state of the store in that location—a V-value. In either case we now have a V-value. Next we find a new L-value outside the active area of the store and attach it to ξ. Finally we update this location with the computed V-value. Thus prepared, we can now execute γ. (The only reason for being so complicated is that in γ we would like to be able to use ξ on the left of assignment statements.) The first two cases are not so very different, but the third presents a fresh aspect.

At the time of making a recursive definition, we want to associate the "import" of ε with ξ, but we do not want to evaluate ε. Also ε may contain ξ (a free occurrence of the identifier), and we want ξ to mean the same as we just said. Having made this tie-up without actually evaluating anything, we begin to execute γ. Every time we find a (free) ξ, in effect, we want to replace it be ε and continue the evaluation. That is to say, ε will be evaluated only when it is called for by an occurrence of ξ. Since ε contains ξ, this may happen repeatedly. This sequence of events is quite unlike the non-recursive initializations where the ξ's in ε are not at the same as the ξ's in γ.

Turning now to expressions, the explanations can be rather short. The read expression invokes input and has a V-value. The result is combination allows for the command γ to be executed before ε is evaluated. Next we see the conditional expression. The ε: X expressions are all Boolean-valued and are tests on the nature of the value of ε. In lines two and three of the definition we have obvious Boolean-valued or numerical-valued expressions. In line four we have to do with (closures of) commands, and : γ forms just such a closure making a command into an expression. Line five has to do with procedures. First we find λ-abstraction (functional abstraction) which forms a (closure of) a procedure from an expression with respect to a formal parameter. Next we have application of a procedure to an argument. Line six, finally, has to do with references. We can make a reference (to the value of ε), or we can look up a reference. References bring in the problem of sharing, but we will not treat it in this paper. Nor shall we discuss arrays. Note too that labels and jumps have not been included.

The explanations just given have not been very full; but the programming concepts mentioned are familiar, and most of us would now have a fairly good idea of what a program written in this language would do. Such intuitive understanding is not rigorous semantics, however, and as everyone knows sticky points can arise. Just as we could turn the intuitive assumptions about the store into precise equations, so must we formalize our semantical understanding, as will be outlined in the next section.

THE SEMANTICS

The plan of the semantical definition is that every command and every expression shall denote something. It is the construction of the domains in which the values lie that requires the mathematics. We have spoken already about the basic domains L, V, S, and T. Of these L and T can be given independently in advance, but V and S can only be formed in conjunction with the over-all semantical project. To do this we have to look at the kinds of expressions there are to be found in the
language and to provide appropriate domains. The
uses of the identifiers also have to be taken into account.
This analysis suggests these domains in addition to the
four already isolated:

**The semantical domains**

- **N** = numerical values
- **E** = expression values
- **W** = dynamic values
- **D** = denoted values
- **C** = command values
- **P** = procedure values
- **R** = reference values

The denoted values accumulate all the possible values
that might be attached to an identifier. A scheme of
such attachments is called an **environment**. Mathematically it is just a function from identifiers to denoted
values. All our evaluations will have to take place
within (or relative to) such an environment. The
following variable is restricted to environments:

\[ \rho : \text{Id} \rightarrow \text{D} \]

Environments can change, but the meanings of the
numerals and the numerical operations are fixed. The
way to fix them is to define in the standard way these
two functions:

- **Numval**: \( \text{Num} \rightarrow \text{N} \)
- **Opval**: \( \text{Op} \rightarrow \text{N} \times \text{N} \)

Explicit definitions need not be given here.

Ultimately a command means a store transformation.
In mathematics expressions have "static" expression
values, but in programming it turns out that commands
are invoked in evaluating a command. Thus an
expression has an effect as well as a value. We also have
to remember that expressions are sometimes used for
their L-values and sometimes for their V-values. All of
this can be summarized in several semantical functions
which have these logical types:

**The semantical functions**

\[ \epsilon : \text{Cmd} \rightarrow \text{[Id} \rightarrow \text{D}] \rightarrow \text{[S} \rightarrow \text{S}] \]
\[ \xi : \text{Exp} \rightarrow \text{[Id} \rightarrow \text{D}] \rightarrow \text{[S} \rightarrow \text{E} \times \text{S}] \]
\[ \eta : \text{Exp} \rightarrow \text{[Id} \rightarrow \text{D}] \rightarrow \text{[S} \rightarrow \text{L} \times \text{S}] \]
\[ \upsilon : \text{Exp} \rightarrow \text{[Id} \rightarrow \text{D}] \rightarrow \text{[S} \rightarrow \text{V} \times \text{S}] \]

To understand how these functional types are
arrived at, consider a command \( \gamma \). Given an environ-
ment \( \rho \) and a state of the store \( \sigma \), we can write the
equation

\[ \epsilon(\gamma)(\rho)(\sigma) = \sigma' \]

to indicate that \( \sigma' \) will be the state achieved after
execution of \( \gamma \). Similarly for expressions the equation

\[ \xi(\epsilon)(\rho)(\sigma) = (\eta, \sigma') \]

indicates that \( \eta \) is the value found for \( \epsilon \), while \( \sigma' \) is the
state achieved after the act of evaluation.

An immediate question is why the \( \rho \) and \( \sigma \) are
separated as arguments. The answer is that they enter
into the evaluations in different ways. For one thing,
they change at quite different rates with \( \rho \) staying fixed
much longer than \( \sigma \). For another, we will be tempted to
copy \( \rho \), but we will generally never feel free to ask for a
copy of the whole store—there just is no room for that.
Possibly in simulating a small machine on a large
machine we could conceive of programming the copying
of the store, but this is not yet a standard concept. In a
way \( \rho \) enters at a more conscious level, while \( \sigma \) flows
along as the unconscious. Much of the action on \( \sigma \) is
meant to be automatic, though there is communication
between the levels. Besides these reasons, it is often the
case that we want to discuss \( \epsilon(\gamma)(\rho) \) without reference
to a particular \( \sigma \); thus it is convenient to be able to drop
it off as the last (deepest) argument.

For stylistic reasons and as an aid to the eye, we shall
use **emphatic brackets** with the semantical functions:

\[ \xi(\epsilon)(\rho)(\sigma) \]

These are helpful because the expressions tend to get
along and to pile up their own parentheses. Before we
can discuss the semantical equations, however, we must
return to the definitions of the various domains.

The domains \( L \), \( T \) and \( N \) are given. The rest are
constructed. The constructions can be summarized as follows:

**The domain equations**

\[ V = T + N + C + P + R \]
\[ E = V + L \]
\[ W = S \rightarrow E \times S \]
\[ D = E + W \]
\[ C = S \rightarrow S \]
\[ P = E \rightarrow W \]
\[ R = L \]
We shall not need variables over all of these domains. The following restricted variables are used:

\[ \eta: E, \quad \delta: W, \quad \theta: C \]

One domain is missing from the list: namely, S. The construction of S is a bit complicated and needs more discussion of L than we have time for here. Roughly

\[ S = A \times [L \rightarrow V] \times I/O, \]

where the A-coordinate represents the area; the middle coordinate, the contents-function; and the last, the input/output features. But this is a little too rough, though we must leave it at that.\(^{14}\)

What is essential to note here is the highly recursive character of these domain equations; in particular, V involves S and S involves V. The involvement comes about through the use of function domains such as \([S \rightarrow S]\) and \([L \rightarrow V]\). Since V-values are storable values, something seems wrong here. We said earlier that we would not store (i.e., copy) stores, yet we are asking to store a store function \(\theta:S \rightarrow S\). There is actually no conflict here. Functions involve their arguments in a potential, not actual, manner. In implementing such a language, functions would be stored as finite descriptions (i.e., texts of definitions) not as infinitary abstract objects. From the point of view of mathematical conception, however, we want to imagine in theory that it is the functions that are stored. If we did not want to do this, we should explicitly program the syntax of function definition and the necessary interpretive operations. In any case, storing a function is done without storing the store.

Granting the reasonableness of the connections between V and S, it is still fair to ask whether such domains exist. This is where the real mathematics enters. The idea mentioned in the Introduction of constructing \(\lambda\)-calculus models can be employed to construct these more involved domains. It requires a combination of lattice theory and topology to achieve the proper kind of function theory.\(^{17}\) On set-theoretical grounds it is clear that when we write \([S \rightarrow S]\) we cannot mean to take arbitrary functions, otherwise the cardinality of S (and of V) would become unbounded. The point is that the functions defined by programs are of a very special sort. Looked at in the right way they are continuous functions, and there are many fewer continuous functions than there are arbitrary functions. And this turned out to be the secret: by restricting the function spaces to the continuous functions, the required domains can indeed be constructed. Expensive as this device is, the Hovercraft effect is avoided by showing that such a theory of domains fits in very well with the theory of computable functions (recursive functions) and that all the functions required by the semantics of the language are indeed computable. In particular the theory provides a basis for justifying the existence of functions defined by involved recursions (like the semantical functions) and for proving properties of such functions.\(^{18,19}\)

It should be stressed that the semantical domains make no explicit reference to the syntax of the language. True, we gave the language first, and then cooked up the domains to match. It could have easily been the other way round. Take V for example. There were five main sections to the definition of an expression corresponding to five sorts of values an expression could have. (Strictly speaking this is not true. The expressions \(\varepsilon\eta\delta\theta\) and \(\uparrow\xi\varepsilon\) could have arbitrary values. It might have been more logical to put these clauses on the first line of the definition.) Therefore in the construction of V, we had to provide for five sorts of elements, T, N, C, P, and R. The summation operation (+) on domains has to be defined as a disjoint union (and a little care is needed with the topology of the resulting space). If we had wanted, we could have included other sorts of summands. They would not have been reflected in the language, however. It is an interesting question whether the language reflects enough of the structure of the domain V, but one that we cannot pause to consider here. The point is that language design and domain construction are parallel activities, each can (and should) affect the other.

Finally it remains to illustrate some of the equations needed to define the semantical functions:

\[ \text{Some semantical equations} \]

\[
\begin{align*}
\varepsilon[\varepsilon_0 := \varepsilon_1](\rho) &= \text{UpdatePair}(\varepsilon_0[\varepsilon_1](\rho)) (\forall[\varepsilon_1](\rho)) \\
\varepsilon[\text{write } \varepsilon](\rho) &= \text{Write} \varepsilon[\varepsilon](\rho) \\
\varepsilon[\gamma_0; \gamma_1](\rho) &= \varepsilon[\gamma_1](\rho) \cdot \varepsilon[\gamma_0](\rho) \\
\varepsilon[\eta](\rho) &= \text{Do } \varepsilon[\varepsilon](\rho) \\
\varepsilon[\text{let } \xi := \varepsilon \text{ in } \gamma](\rho) &= (\lambda \eta \cdot \varepsilon[\gamma](\rho[\eta/\xi])) \cdot \varepsilon[\varepsilon](\rho) \\
\varepsilon[\text{letrec } \xi \text{ be } \varepsilon \text{ in } \gamma](\rho) &= \varepsilon[\gamma](\rho[\text{Fixpoint}(\lambda \delta \cdot \varepsilon[\varepsilon](\rho[\delta/\xi]))/\xi]) \\
\end{align*}
\]

For the \(\varepsilon\)-function there should be 11 equations in all; and for the \(\varepsilon\)-function, 21 equations. Then the semantical definition would be complete. Note how the equations are mutually recursive functional equations: the \(\varepsilon\)- and \(\varepsilon\)-functions recur on both sides of the
Mathematical Concepts in Programming Language Semantics

equations in functional contexts. Note too that there is much unexplained notation.

In order to be able to combine functions, we need at least two kinds of composition; because some functions are of type \([S \rightarrow S]\), and some are of type \([S \rightarrow X \times S]\). We can see how this works in one example. Let:

\[ f = \mathcal{E}_0(\rho) : S \rightarrow L \times S \]
\[ g = \mathcal{V}_1(\rho) : S \rightarrow V \times S \]

We want:

\[ \text{Pair}(f)(g) : S \rightarrow [L \times V] \times S. \]

Suppose \(\sigma\) is a given state. Then \(f(\sigma) = (\alpha, \sigma')\). Carrying over \(\sigma'\) to the next step, \(g(\sigma') = (\beta, \sigma'')\). So we can define:

\[ \text{Pair}(f)(g)(\sigma) = ((\alpha, \beta), \sigma''). \]

(The reason for doing it this way is to get the effect of two separate evaluations, because there were two expressions \(\mathcal{E}_0\) and \(\mathcal{V}_1\) needing evaluation.) Then by definition:

\[ (\text{Update} \circ \text{Pair}(f)(g))(\sigma) = \text{Update}(\alpha, \beta)(\sigma'') \]

This shows how \(\circ\) is a form of functional composition. In the third equation we wrote \(\circ\), since this was just the ordinary composition of two \([S \rightarrow S]\) functions.

The notation \(\rho[\eta/\xi]\) means that the original environment \(\rho\) has been altered to assign the value \(\eta\) to the argument \(\xi\). (Remember that the arguments of environments are the formal identifiers as syntactic objects.) The import of the fifth equation then is to evaluate \(\varepsilon\) first, pass on the value to the environment, and then execute the command starting in the state achieved after evaluation of \(\varepsilon\). Note that \(\varepsilon\) is evaluated in the original environment.

The \text{Fixpoint} function in the last equation is applied to a function \(F : W \rightarrow W\). All our domains are constructed so that functions defined by expressions always have fixed points—if they have the same domain of definition and range of values, of course. The domains are lattices as well as topological spaces, so we can speak of the \text{least} solution to the equation:

\[ \delta_0 = F(\delta_0). \]

This \(\delta_0 = \text{Fixpoint}(F)\). Fixed points of functional equations solve recursions. So in the letrec command, we first set up the solution \(\delta_0\) and then execute the command in the environment \(\rho[\delta_0/\xi]\). This explains our mathematical concept of recursion.\(^{20}\) Note that there are no stacks, no pointers, no bookkeeping. None of those devices of implementation are relevant to the concept. That is why this is a mathematical semantics.

CONCLUSION

The presentation in this paper has been but a sketch. For a complete semantics even of this small language, many more details would be required. But it is hoped that enough of the project has been exposed to convince the reader that it is possible to carry out such a plan. And it is not even too painful to do so: not so many mathematical constructs are required, and the equations need not be so very complicated. Of course, judgment of the worth of doing so should be withheld until the semantical equations for a really serious language are fully written out.

REFERENCES

1 C STRACHEY
Towards a formal semantics
In T B Steel (ed) Formal Language Description Languages (North-Holland 1966) pp 198-220. Cf. also Ref 15 for another description of the present program

2 H B CURRY ET AL
Combinatory logic
Vol 1 (North-Holland 1958) Vol 2 (North-Holland 1972 in press) This is a basic reference with a very large bibliography

3 P J LANDON
The mechanical evaluation of expressions
Comp J 6 pp 308-320 1964

4 A EVANS JR
\text{PAL—a language for teaching programming linguists}
In Proc ACM 23rd Nat Conf Brandon Systems Princeton NJ The PAL language carries out many of the ideas introduced by Landon

5 Cf. Ref 2 p 78

6 The first proof of inconsistency was given by Kleene and Rosser for a system of Church. The proof was later much simplified. Cf. Ref 2 pp 258 ff

7 D SCOTT
Outline of a mathematical theory of computation

8 D SCOTT
Continuous lattices
Proc Dalhousie Conf Springer Lecture Notes in Math 1972 in press

9 D SCOTT
The lattice of flow diagrams
In E Engeler (ed) Semantics of Algorithmic Languages Springer Lecture Notes in Math Vol 188 1971 pp 311-366. This paper provides many examples of the limit concept. Cf. also Ref 10

10 D SCOTT
Lattice theory, data types and semantics
NYU Symp on Formal Semantics Prentice-Hall 1972 in press. This is a general introduction more detailed in some respects than Ref 7

From the collection of the Computer History Museum (www.computerhistory.org)
11 C STRACHEY
Varieties of programming language
Proc ACM Conf Venice 1972 to appear. This paper contains more examples
12 R A FAIRTHORNE
Remarks in P Atherton (ed) Classification Research
Munksgaard Copenhagen 1965 p 210
13 C STRACHEY
An abstract model of storage
In preparation. This report will give complete details
14 Cf. Ref 11
15 D SCOTT, C STRACHEY
Towards a mathematical semantics for computer languages
Proc of Symp on Computers and Automata Microwave Research Institute Symp Ser Vol 21 Polytechnic Institute of Brooklyn 1972 in press. A nearly identical language and its semantics is outlined in this paper together with some more of the mathematical background
16 Cf. Ref 13
17 Cf. Refs 7 and 8
18 R MILNER
Implementation and applications of Scott's logic for computable functions
SIGPLAN Notices Vol 7 No 1 January 1972 pp 1-6. (=SIGACT News No 14)
19 J W De BAKKER
Recursive procedures
Math Centre Tracts 24 Mathematisch Centrum Amsterdam 1971. Other useful references are contained in these two papers
20 Cf. Refs 9, 15 and 19 for further discussion of recursion