Rx for Semantics*

E. A. Ashcroft       W. W. Wadge

Abstract

A constructive framework of recent work in the semantics of programming languages is offered. The criticism is directed not so much at the techniques and results obtained as at the use to which they are put. The fact that denotational (or "mathematical") semantics plays on the whole a passive ("descriptive") rôle, while operational semantics plays on the whole an active ("prescriptive") rôle, is seen as the basic problem. It is suggested that these rôles be reversed.

1 Introduction

We would like to offer a constructive criticism of recent work in the semantics of programming languages, in particular, work in denotational ("mathematical") "Scott–Strachey" semantics. For the most part we are criticizing not the tools and techniques developed but rather the use made of these tools and the rôle which semantics plays in language design.

In semantics and in other branches of computer science there are two points of view concerning the rôle of mathematics. One point of view sees mathematics as playing primarily a passive rôle. According to this point of view, the entities considered by computer scientists are shaped mainly by forces outside their control; their job therefore is to develop the necessary tools to study computer science, that is, to describe, to model, to classify. We call this the descriptive approach.

The other point of view sees mathematics as playing primarily an active rôle. According to this point of view, machines, languages and systems are (or should be) the computer scientists’ own creations, so that they can freely choose to create them to conform to mathematically simple principles. The mathematics is directed toward design rather than study, and mathematics is used not so much to describe existing objects as to plan new ones. This we call the prescriptive approach.

In general, the descriptive approach aims for generality even at the expense of simplicity and elegance, while the prescriptive approach aims for simplicity and elegance even at the expense of generality.

Our criticism is that operational semantics is being used prescriptively and denotational semantics descriptively, whereas it would be better if the rôles were reversed.

2 The Scott–Strachey Method as a Descriptive Tool

It would not be correct to say that Scott, Strachey and their followers intended to develop a purely descriptive system. Strachey’s interest in mathematical semantics stemmed from his interest in language design, and in one of their first joint papers, Scott and Strachey at one point present what is in fact a good explanation of what it means to use denotational semantics prescriptively:

The authors have the peculiar idea that the domains of our concepts can be quite rigorously laid out before we make final the choice of language in which we are going

to describe these concepts.... This is not to deny that there may be some vague ideas of language which influence our choice of domains. What we suggest is that in order to sort out your ideas, you put your domains on the table first. Then we can start talking about them [24, p.40].

They in fact proceed in the same paper to define a small language along these lines. Furthermore, in the later works of Tennent [27], Milne and Strachey [21] and Stoy [26] (these are now the “standard” presentations of the subjects), the authors express the opinion that one (or even the best) use of the method is as a design tool. Tennent himself [28] used principles derived from the denotational approach to produce some valuable proposals (unfortunately neglected) for the improvement of PASCAL.

Nevertheless, despite these intentions and efforts, the Scott–Strachey method has in fact evolved as a descriptive tool. Researchers in the field began working through the language manuals, devising denotational characterizations for construct after construct. The three expositions of the method mentioned above and the more recent work by Gordon [13] all have the same form, namely, the presentation of a simple example language to illustrate the method, followed by a series of explanations of how the method can be extended to handle various complications. In Tennent’s survey paper alone the following features are discussed: assignment, conditional and repeat statements; side effects; coercions, pointer variables; dynamic allocation and deallocation; opening, closing and rewinding files; call by name, value, and reference; and goto statements, exits, error stops, backtracking, and coroutines. According to Milne, semantics have already been given for (among many others) Algol 60, SNOBOL, and Algol 68, and work was then “under way” on a description of PL/1.

(Ironically, at some institutions, many of the “nasty” features mentioned above have been banished from introductory programming courses, and the only places students come across some of them are courses in semantics!)

In the development of the Scott–Strachey method, generality has emerged as the main objective. In [21] Milne and Strachey list four properties that a specification method must possess, and the second of these is that “it must be applicable to all programming languages.” According to Milne and Strachey [21, p.10], “the method of mathematical semantics has the applications and properties just mentioned.” The summary on the flyleaf of the book repeats the claim that the methods “are applicable to any programming language.” Tennent states that the method of Scott and Strachey has proved to be adequate, “despite the complexity and variety exhibited by modern programming languages” [27, p.437].

As is to be expected, this generality has been achieved at the expense of simplicity. All but the simplest conventional languages require large and sophisticated definitions. (This is acknowledged by Gordon in [13, p.148], where he decides not to attempt to put together a definition of a full language because of the forbidding size of the result. It is also apparent, for example, in the definition of SAL in [21].) It is true that only a small number of basic concepts (abstractions, function spaces, continuations) are involved, but these concepts are so powerful that even small definitions can specify exceedingly intricate objects or denotations. The Scott–Strachey semantic descriptions are complicated in the sense that they are very difficult to use. For example, it is hard to see how there can be any assurance that these involved expressions correctly reflect the intentions of the language designers.

What lies behind this quest for generality is the idea that the Scott–Strachey method is a method for giving semantics to already existing languages, that is, a method for recording design decisions already made. This attitude can be perceived clearly in almost all writings on the subject, despite the remarks about the method’s use as a design tool. Milne refers to the Scott–Strachey method as method for formalizing semantics; presumably, this means giving precise descriptions of a semantics already prescribed but formulated informally. Tennent calls the Scott–Strachey method a method for giving “formal models” of the meanings of languages. Gordon simply entitles his book The Denotational Description of Programming Languages (emphasis ours).

It is certainly true that some Scott–Strachey semanticists see themselves as playing a part in the language design process; but the parts they are offered are usually quite peripheral. They
must act as assistants or advisers to the designer. When a design is proposed, the semanticist formalizes it and is allowed to offer suggestions, some of which (if they are not too radical) might possibly be accepted. In any case, the semanticist is required (and able) to describe formally whatever the designer chooses as his or her final design. (We have firsthand knowledge of one such situation in which a colleague was called in by the designer of one of the Ironman language candidates. He did have some influence on the language, but it was limited because the domains of the designer’s concepts had been quite rigorously laid out by the Ironman specification.) For all intents and purposes, the semanticist is used as a describer. In fact, Stoy concedes that the time when language design will be the main use of the Scott–Strachey method lies “in the future”; at present, it is being used for description.

This attitude (at least in its extreme form) sees programming languages as naturally occurring objects (like heavenly bodies); it sees the semanticist as being in essentially the same position as an astronomer gazing through his telescope at some distant star or galaxy. Indeed, we might continue the analogy and liken the Scott–Strachey method to the cosmological theories of the ancient astronomer Ptolemy, which very successfully described the apparent motions of the planets in terms of an elaborate system of cycles and epicycles. Ptolemy’s technique was completely general in the sense that any planetary motion could be described by adding enough epicycles.

We suggest that this preoccupation with generality and description could have an effect exactly opposite to that originally intended (as stated in the passage in [24] quoted earlier). Language designers worry a lot about their languages. For example, they worry about whether or not they will be powerful, efficient, well structured, and amenable to verification. The fact that the Scott–Strachey method can deal with goto statements makes it more likely, not less, that this feature (and worse ones) will be included in the language (assuming the designers pay any attention at all to formal semantics). Milne and Strachey present a language, SAL, of their own design to illustrate the method. They deliberately include in SAL facilities of “dubious merit,” for example, storable label values which permit “bizarre jumps back into blocks.” They do this in order “to illustrate the ways in which mathematical semantics can handle both wise design decisions and foolish ones” [21, p.384]. The method is so powerful that it allows, or even encourages, language designers to ignore the advice of semanticists. It offers language designers a blank check and constitutes, to paraphrase Dijkstra [7], an open invitation to language designers to make a mess of their language.

3 The Prescriptive Approach

Although we have accused denotational semantics of the Scott–Strachey variety of playing on the whole a descriptive rôle, it would not be correct to say that denotational semantics in general has played no rôle at all in language design. In fact, one of the first and yet most successful and influential of all programming languages, LISP, was the result of an early attempt (not completely successful, it is true) to use denotational ideas prescriptively. Other examples include APL, Landin’s ISWIM [18], Scott’s LAMBDA [25], Turner’s SASL [29], the coroutine language of Kahn and MacQueen [15], Dijkstra’s “guarded command” language [8], Prolog [17], Lucid [2, 4, 3], and ML of Gordon, Milner, Morris, Newey, and Wadsworth [12]. Although some of the designers did not use the theory of domains as such, the languages were denotationally prescribed in that the designers began with abstract, denoted objects (functions, rectangular multidimensional arrays, streams, infinite sequences) and then proceeded to specify the language and investigate implementations.

We are not the first to perceive the difference between the prescriptive and descriptive approaches, nor are we the first to recommend the prescriptive approach in semantics. We are not even the first to use the word in this sense; Dijkstra used it in [9] in March 1977. A good presentation of the prescriptive viewpoint can be found in Jack Dennis’s opening remarks to the

---

1 Algol is in fact the second brightest star in the constellation Perseus!
IFIP-sponsored 1977 semantics conference [6], where he offers the motto, “Let us not formalize what exists; rather, let us discover what should exist.”

Dennis was talking about semantics in general, but we have also seen that in denotational semantics in particular there is understanding of the issue and a desire to play a more active rôle. What requires explanation, then, is the fact that the more active rôle has been so slow in emerging. Some undoubtedly believe that the trouble is that denotational semantics is still not general enough and that the answer is to extend the method even further to handle such things as nondeterminism and fairness. These technical questions may be worth pursuing, but from what we have seen earlier it is very unlikely that even a completely successful resolution of these problems would result in a real change of direction.

We can obtain some indication of the real nature of the problem by looking a little more closely at the early history of LISP [20], one of the first attempts to use denotational methods prescriptively. McCarthy was motivated by the early work of Church [5] on the λ-calculus and of Kleene [16] on recursion. His original intention was to produce a language in which a program was simply a set of recursive function definitions, the meaning of which was one of the functions so defined, that is, the least fixed point of the program.

This intended semantics of pure LISP, however, was never completely formalized by McCarthy, probably because the necessary mathematics was not well understood at that time. Instead, the semantics was specified by giving an interpreter program (written in LISP itself), which, it was believed, correctly computed the results of the functions defined.

Unfortunately, it was later discovered that this was not the case; the interpreter used only a simple “pairlist” for binding values to variables, and it did not always select the appropriate value. This situation was never corrected. The problem was not just that programmers were consciously and eagerly using “dynamic binding”; the real difficulty was probably the fact that a correct implementation of static binding is not easy to construct. Some form of elaborate Algol-like stack is needed, together with (for handling higher-order functions) a system for carrying around closures (expression-environment pairs). If a completely correct implementation of recursive definitions is required, interpreting functions and functional applications in the usual sense, some form of call by name is also needed [30], and this complicates the interpreter even more. This is not to say that these implementation problems are unsolvable; but there is no simple interpreter, for the pure LISP originally intended, which is correct, completely general, and reasonably efficient, all at the same time.

4 Denotational versus Operational Semantics

We could summarize the history of LISP by saying that its developers discovered that a simple denotational semantics of LISP was inconsistent with a simple operational semantics (implementation) and that they chose the latter as its basis. It is our thesis that this situation is not peculiar to LISP but rather is only a particular instance of a general phenomenon, namely, the existence of a significant degree of conflict or incompatibility in the relationship between denotational and operational considerations.

At this point it is worth saying a few words about the terms “denotational” and “operational” and the difference between denotational and operational semantics. Semantics in general is the study of the association between programs and the mathematical objects (e.g., functions) which are their meaning. In some simple languages, for example, the meaning of a program is the functional relation between its input and output, so that correctness (but not efficiency) refers to the meaning alone. The phrases “denotational semantics” and “operational semantics” are somewhat misleading because they seem to imply that the different approaches give different results.

2The corresponding IFIP Working Group is unfortunately entitled “Formal Description of Programming Concepts”.

3It is in fact true that there is still no general and completely satisfactory domain-theoretic characterization of nondeterminism. This is the famous “powerdomain” problem discussed, for example, in [22]. In spite of the claims to generality discussed earlier, the Scott–Strachey method is still not applicable to all programming languages.
meanings to the same program; instead, the proper distinction is between different methods of specifying the same meanings. It would be better to talk of “giving semantics operationally” and “giving semantics denotationally” rather than “giving operational semantics” and “giving denotational semantics.”

To specify the semantics of a language means to specify a group of functions which assign mathematical objects to the programs and to parts of programs (modules) in such a way that the semantics of a module depends only on the semantics (i.e., not on the form) of the submodules.

The approach is inherently modular, and, indeed, “modular semantics” is a possible alternative name. For complex languages the semantic objects assigned to modules may be required to have very unusual properties, and Scott, Strachey, and their followers developed the theory of domains and domain equations to establish the existence of such objects.

To specify the semantics of a language operationally means to specify an abstract machine together with the machine behavior which results when a program is run on the machine. The meaning of the program is not the behavior itself; but the meaning (e.g., the input/output relation) is specified in terms of the behavior. Operational methods are not necessarily modular.

The words “denotational” and “operational” are also used in a wider context, with the first referring to concepts related to denotations of modules and static semantic objects in general and the second referring to concepts related to machines, behavior, and dynamic considerations in general. The essential difference is that denotational ideas refer to what a program or module is computing, whereas operational concepts refer to how it is computed. One refers to ends, the other to means.

It should be clear, then, that when we say that there is an element of incompatibility between operational and denotational viewpoints, we do not mean that they are mutually exclusive. On the contrary, the concepts are basically complementary; in general, the semantics of a programming language can be specified both operationally and denotationally so that we know both what we want a program to compute and how we can compute it.

When we say that there is an element of conflict between the two points of view, we mean to say that they are not completely symmetrical, not simply mirror images of each other, and that things favored by one are not necessarily favored by the other. It seems to be a general rule that programming language features and concepts which are simple operationally tend to be complex denotationally, whereas those which are simple denotationally tend to be complex operationally—at least if we are interested in operational concepts which are efficient. As elsewhere in this paper, the word “simple” means, roughly speaking, easy to use and understand, whereas “complicated” means the opposite.

One can give very many examples of this phenomenon. Goto statements seem so simple and natural in terms of machines, but their denotational descriptions require the elaborate methods of continuation semantics. Dynamic binding uses a very simple single pairlist evaluation technique, but its denotational description [11] involves a complex “knotted” environment. Call by value uses a simple evaluation algorithm but complicates the denotational notion of function involved. On the other hand, we have already remarked that a truly functional language, though mathematically simple, can require sophisticated, or at least complicated, implementation devices, such as closures and displays. Stoy has noticed these two examples of the conflict. On dynamic binding he says,

As is obvious and well-known, in the implementation of languages with dynamic scope rules unevaluated subexpressions may be represented simply by their text. In our particular case [i.e., giving denotational semantics] this actually makes life more complicated rather than less, as the alternative scope rules [i.e., static binding] would not require named subexpressions to remain unevaluated, so that names would denote only simple values. But it does simplify the implementations of languages involving values such as functions, which would otherwise have to be represented by closures or something equivalent [26, p.337].

On call by value rather than call by name he says,

These extra complications have prompted some theoreticians to call for the replacement
of the call by value rule in our languages—even though the hardware to implement
call by name efficiently would be a little more complicated [26, p.179].

Given this situation, we can see that language designers are faced with an endless series of
choices to make: decisions about whether or not a feature will be mathematically simple or oper-
ationally simple, for example, whether to use recursively defined data types or pointer variables.
The tendency is for the choices to be made consistently and thus for the language being designed
to be either mathematically or operationally straightforward. The first approach means designing
a language based on simple denotational concepts; we have already seen that this is the denota-
tionally prescriptive method. The second approach means designing a language based on simple
operational concepts; it should be clear by now that this can only be the operationally prescriptive
method. Most conventional programming languages are designed this way.

To design a language operationally means to begin with operational (not denotational) concepts
and then proceed to develop the language itself on this basis. It means first designing a (more
or less) abstract machine together with some idea of the kinds of behaviors allowed and then
formulating language features with which the desired behavior is to be specified. Denotational
considerations can play some rôle, but only insofar as they do not interfere with the general
approach; and, usually, denotationally motivated features are generalized on an operational basis,
so that, for example, pure functions become functions with side effects and a variety of calling
conventions.

Usually, the approach is implicit and the machine is specified very informally, but occasionally
the technique is more explicit and formal. The language BCPL [23], for example, is specified
or at least implemented using a simple abstract machine with a stack, some registers, and a
modest instruction set. The entire BCPL compiler (except for the code generator) is available
in the abstract machine language, and implementing BCPL on a new computer involves mainly
implementing the abstract machine.

We said that operational design begins with the design of an abstract “machine”; the word
“machine” is appropriate because the mathematical machine is usually closely modeled after the
conventional von Neumann architecture and is centered around some form of storage. Sometimes,
however, the machine has a very different structure, say with locations or registers capable of
holding strings of arbitrary length. Sometimes the language is based not so much on a particular
machine as on some general concept or model of computation, for example, interprocess commu-
nication (EPL [19], CSP [14]) or lazy evaluation [10]. These latter languages are nevertheless still
examples of operational prescription because the design is still based on dynamic concepts, on
notions of behavior, and on the change of state of some system. The operationally-prescriptive
approach is bottom up in that design starts with the machine and works upward.

Operationally based languages remain operationally based no matter how they are described,
and giving them a denotational semantics does not make them any more denotationally based.
Quite the contrary, the nasty features are very resistant to any form of description, and their
inherently operational nature shows through very clearly even if someone describes them using
Scott–Strachey semantics. In fact, the attempts of this school to extend their methods to “handle”
such features has merely resulted in supposedly denotational descriptions which are suspiciously
operational. This phenomenon, the blurring of the distinction between the two description meth-
ods, has been noticed and clearly described by Anderson, Belz, and Blum [1].

It must be emphasized that the operationally-based languages are not simply the imperative
languages. Imperative features in a language are clearly machine-oriented and dynamic in char-
acter, but they are not the only such features. We have seen, for example, that dynamic binding
and call by value are also operationally inspired; yet neither feature is in any sense imperative.
Similarly, the term “denotationally based” is not simply equivalent to the term “applicative” or
“nonprocedural.” If we take “applicative” to mean “pure functional” or “referentially transparent”,
then it is certainly true that applicative languages are denotationally based. On the other hand,
if we consider applicative languages to be those whose only feature is function application, then
this may not be the case. The “applicative” language of Friedman and Wise, for example, is based
not on the denotational concept of function but on the operational concept of lazy evaluation.
One of the most important advantages of the denotational approach is the fact that in general it produces languages with simple inference and manipulation rules. Our discussion so far has been almost entirely in terms of operational and denotational semantics, and it might seem that a third party, named “axiomatic semantics,” has been ignored. Verification considerations are of course extremely important in design; fortunately, however, it seems that the axiomatic and denotational requirements are essentially the same. We have seen that the “nasty” operationally motivated features complicate the denotational semantics because they reduce the modularity in the language; the same features complicate the rules of inference, and for the same reason.

5 Lucid: A Case Study of the Denotationally Prescriptive Approach

At this point it is necessary for us, having talked about the operational approach, to try to indicate more precisely what constitutes a genuine denotationally-prescriptive approach. This is not easy to do, for the simple reason that there is very little experience to draw on; the overwhelming majority of programming languages were (and are) designed on an operational basis. Nevertheless, we feel that our work on Lucid [2, 4, 3] allows us to make some contribution.

With the denotational approach the design of the language begins with the specification of the domains of semantics objects. The obvious question is, which domains? We have already seen that some domains are essentially machines; that is, the elements of such domains are the states of a machine. If we base our design on such a domain, we will probably find ourselves taking the operational path. From our limited experience, it seems that the best strategy is to choose domains of simple conventional objects, that is, numbers, sequences, sets, and functions. There are several simple domain-building equations that are useful: for example, if $A$ is a domain of interest and

$$L = A + L \times L$$

then $L$ is the domain of LISP “S-expressions” built up with elements of $A$ as atoms (this is in fact McCarthy’s original domain equation). The work of many semanticists who concentrate on developing domains of new semantics objects and their corresponding theories is relevant here. The language Lucid is based on a very simple domain consisting of infinite sequences of elements of the base domain, and on the derived function domains.

It should be emphasized very strongly that operational considerations can play an important part in the denotationally prescriptive approach. If a language is based on domains of lists, the designer can reasonably expect that programs in the language may be implemented or even understood in terms of pointer-linked structures. Operational concepts can be valuable design and programming aids provided they are kept in the proper perspective and do not come to dominate the thinking. In the denotational approach it is the implementer who must play the rôle of trusted advisor.

In the development of Lucid, our initial concern was with capturing the idea of iteration in a mathematical way: at first it seemed very difficult because of the normal requirement that loops terminate, which seemed to imply that computation traces should be finite. We finally realized that, although only a finite amount of activity is necessary to yield the output of a finite computation, it is much simpler mathematically to specify this output by extracting it from infinite histories of notionally unending computations. This puts more of a burden onto implementers, because the implementation has to decide when and how to stop following these infinite computations. On the other hand, because the semantics suggests but does not specify the operational behavior, it is possible to use other iterative methods, such as dataflow, or even methods which are not iterative at all.

The full Lucid language did not suddenly appear in its completed form; Lucid developed over a period of time, as do all programming languages. The two different approaches (descriptive and prescriptive) are actually approaches to the development of languages. As languages (and
families of languages) develop, both aspects of semantics (denotational and operational) must develop together; but the question is, which aspect plays the leading rôle? The operationally-based languages develop by generalizing existing operational notions and devising new ones (e.g., procedures, then procedures with elaborate calling conventions, then coroutines and actors, and so on). Denotational languages, on the other hand, develop by generalizing denotational ideas, adding new functions, domains, domain operations, and so on.

With Lucid, for example, the next step was to develop the mathematical side, in almost the simplest possible way, but considering functions from histories to histories. Although it was never our intention to develop a “coroutine” language, it emerged that modules specifying such functions can be understood operationally as coroutines with persistent memory; and whole programs can be understood as producer-consumer (or dataflow) networks of such “actors” computing in parallel. Furthermore, this operational interpretation can be used as the basis for a distributed implementation. By contrast, the addition of “parallelism” and “message-passing” features to a conventional operationally-based language results in enormous complications and makes a denotational description nearly impossible. For example, the (aforementioned) colleagues working on the Ironman description wisely declined to handle any of the language’s parallel features.

6 Conclusion

In a sense the goal of the denotationally-prescriptive method is to restore the proper relationship between theoretical and practical matters in the field of language design. At present, languages are operationally based, and their definitions more or less specify the means of implementation. The real interest seems more in finding ways of describing them, for which mathematical theoreticians devise elaborate techniques. Theoretical computer scientists are to a large extent outside observers. In the future, languages will (we hope) be designed on denotational principles, and mathematics will be applied actively to the problems of design and implementation. The denotational approach paradoxically offers both implementers and semanticists more freedom. To quote Dijkstra [9]: “The semantics no longer needs to capture the properties of mechanisms given in some other way, the postulated semantics is to be regarded as the specifications that a proper implementation should meet.” Implementers will have more freedom because they will be free to investigate a variety of genuinely different implementations. Semanticists will no longer have to worry about describing strange constructs and will be free to investigate genuinely interesting domains.

In summary, our argument is as follows: at present, languages are prescribed on the basis of simple operational concepts, and advanced denotational techniques are developed to describe them. A far better idea is that languages be prescribed on the basis of simple denotational principles and that sophisticated operational techniques be developed to describe, that is, to implement, them. This is our prescription for semantics.

References


