The University of Warwick

THEORY OF COMPUTATION

REPORT NO.30 (REVISED)

B FOR SEMANTICS

BY

E.A. ASHCROFT AND W.W. WADGE

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

DECEMBER 1979

R FOR SEMANTICS

E. A. Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

W. W. Wadge
Department of Computer Science
University of Warwick
Coventry, England

December 1979

$\mathbb{R}$ FOR SEMANTICS

Abstract

We would like in this note to offer a constructive criticism of current work in the semantics of programming languages, a criticism directed not so much at the techniques and results obtained as at the use to which they are put. The basic problem, in our view, is that denotational (or "mathematical") semantics plays on the whole a passive (we call it "descriptive") role, while operational semantics plays on the whole an active (we call it "prescriptive") role. Our suggestion is that these roles be reversed.

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

In semantics and in other branches of computer science there are two points of view concerning the role of mathematics. One point of view sees mathematics as playing primarily a passive role. 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, i.e. to describe, to model, to classify. We will call this the descriptive approach.

The other point of view sees mathematics as playing primarily an active role. 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 so that they conform to mathematically simple principles. The mathematics is directed towards design rather than study, and mathematics is used not so much to describe existing objects as to plan new ones. This we will 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 roles were reversed.

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 [25], 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 rigourously laid out before we make final the choice of the 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 all start talking about them."

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 [20] and Stoy [26], (they are now the 'standard' presentations of the subject), the authors express the opinion that one (or even the best) use of the method is as a design tool. Tennent himself (in [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 book 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, since "nasty" features like those mentioned above have been largely banished from programming courses, the only place students come across them is in courses in semantics!

In the development of the Scott/Strachey method, generality has emerged as the main objective. Milne in his book lists four properties that a semantic specification method must possess, and the second of these states that "it must be applicable to all programming languages". According to Milne, "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, in the first paragraph of his survey paper, states that the method of Scott and Strachey has proved to be adequate, despite "the complexity and variety exhibited by modern programming languages".

As is to be expected, this generality has been achieved at the expense of simplicity. The semantic descriptions become so large and complex that it is difficult to see how they can be useful or even how there can be any assurance of their correctness.

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, i.e. 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 a 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".

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 quite peripheral. They must act as assistants or advisers to the designer. When a design is proposed, the semanticist formalises 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 formally describe whatever the designer chooses as his or her final design. (We have first-hand 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 this was limited because the domains of the designer's concepts had been quite rigourously 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); they see 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 Ptolomy, which very successfully described the apparent motions of the planets in terms of an elaborate system of cycles and epicycles. Ptolomy'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 Scott and Strachey [25] 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 fairly easily makes it more likely, not less, that this feature (and worse ones) will be included in the language. Milne in his book presents a language, Sal, of his own design to illustrate the method. He deliberately includes in Sal facilities of "dubious merit"; for example, storable label values which permit "bizarre jumps back into blocks". He does this in order "to illustrate the ways in which mathematical semantics can handle both wise design decisions and foolish ones" (page 384). The

---

* Algol is in fact the second brightest star in the constellation Perseus!

method is so powerful it allows, or even encourages, language designers to ignore the advice of semanticists. It offers language designers a blank cheque and constitutes, to paraphrase Dijkstra [9], an open invitation to language designers to make a mess of their language.

Although we have accused denotational semantics of the Scott/Strachey variety of playing on the whole a descriptive role, it would not be correct to say that denotational semantics in general has played no role 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, Turner's SASL [29], the coroutine language of Kahn and McQueen [15], Dijkstra's "guarded command" language [10], PROLOG [17] and Lucid [2, 3, 4]. Although none of the designers used the Scott/Strachey method as such, the languages were denotationally prescribed in that the designers began with abstract, denoted objects (functions, rectangular multi-dimensional 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 EWD 614 [9] in March 1977. A good presentation of the prescriptive viewpoint can be found in Jack Dennis' opening remarks to the IFIP sponsored 1977 semantics conference* [7] where he offers the motto

> Let us not formalize what exists; rather, let us discover what should exist.

---

* The corresponding IFIP Working Group is unfortunately entitled "Formal Description of Programming Concepts".

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 role. What requires explanation then is the fact that the more active role has been so slow in emerging. Some undoubtedly believe that the trouble is that denotional 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 [19], one of the first attempts to use denotational methods prescriptively. McCarthy was motivated by the early work of Church [6] 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, i.e. the least fixed point of the program.

This intended semantics of LISP, however, was never completely formalized by McCarthy, probably because the necessary mathematics was not well enough 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.

---

* It 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 [21].

Unfortunately it was later discovered that this was not the case - the interpreter used only a simple "pairlist" for binding values to variables, and 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 LISP language interpreter which is correct, completely general, and reasonably efficient, all at the same time.

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, that it 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. func-

tions) which are their meaning. In some simple languages, for example, the meaning of a program will be 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 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" or "giving semantics denotationally" rather than "giving operational semantics" and "giving denotational semantics".

To specify the semantics of a language denotationally 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 behaviour which results when a program is run on a machine. The meaning of the program is not the behaviour itself; but the meaning (e.g. the input/output relation) is specified in terms of the behaviour. 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, behaviour 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 means that they are mutually exclusive. On the contrary, the concepts are basically complementary; in general the semantics of a programming language should be specified both operationally and denotationally so that we know both what we want a program to compute as well as 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 another. 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 are complex operationally - at least if we are interested in operational concepts which are efficient.

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 description [12] involves a complex "knotted" environment. Call by value uses a simple-minded 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 techniques, such as closures and static binding.

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 operationally simple, e.g. whether to use recursively defined data types or pointer variables. The tendency will be for the choices to be made consistently and thus to design a language which is 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 denotationally 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.

This conclusion, that operational semantics is used prescriptively, might at first seem somewhat surprising because our criticism of denotational semantics, that it is used mainly descriptively, certainly applies as well to operational systems like the Vienna Definition Language [5]. These systems, however, are formal systems; we have already seen that the denotationally prescriptive method can be used informally, and the same is true of the operationally prescriptive method. Indeed the vast majority of modern languages were in fact designed using the informal operational method.

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 designing first a (more or less) abstract machine together with some idea of the kinds of behaviours allowed, and then formulating language features with which the desired behaviour is to be specified. Denotational considerations can play some role, 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.

The approach is usually 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", and the word is appropriate because this mathematical machine is usually closely modelled after the conventional Von Neumann architecture and is centred 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 communication (EPL [18], CSP [14]) or lazy evaluation (Friedman & Wise [11]). These latter languages are nevertheless still examples of operational prescription because the design is still based on dynamic concepts, on notions of behaviour 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.

The fact that conventional languages are designed operationally explains why their denotational semantics are so complex. It is often assumed that programming languages are inherently complicated because they are practical tools, but this is not the case. Their complexity, or at least the complexity of their denotational semantics, is (in our view) the sum total of a whole series of design decisions, the vast majority of which are made at the expense of the denotational view. Of course the fact that a particular kind of description is complicated does not necessarily mean that a language is poorly designed; but denotational semantics is, as we have indicated, modular semantics, and the fact that the denotational semantics is difficult may simply reflect the fact that the language is not very modular. The complexity of the denotational semantics of languages with 'nasty' features (such as jumps and side effects) is just another confirmation of the fact, already well understood, that these nasty features destroy the modularity of a language by enormously increasing the ways in which modules can interact.

It is certainly true that the methods of Scott and Strachey can give almost any language a semantics which is formally modular (denotational); but no amount of description can change the real nature of a language. The words "operational" and "denotational" are more usefully applied to describe the nature of a language itself, rather than the nature of its descriptions. The emphasis on denotational vs. operational descriptions can be very misleading and can even give rise to an almost mystical belief in the power of description; a belief that the problems with a language can somehow be solved by finding a new description, instead of by changing the language itself.

Operationally based languages are still operationally based no matter how they are described, and giving them a denotational semantics does not make them any more denotational. Quite the contrary, the nasty features are very resistant to any form of description, and their inherently operational nature shows through very clearly even in their 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, of the blurring of the distinction between the two description methods, has been noticed and clearly described by Anderson, Belz and Blum [1]. In their words

> "In early "low-level" programming languages, a program was directly related to a machine and, in particular, to the memory (or store) of the machine. Both the addresses (locations) and the contents of memory cells were objects of computations. In higher level languages, this idea also proved useful for various computational purposes; e.g. for "assignment" of values to variables and for creating and manipulating data structures. However, the idea is also present in the meaning [our emphasis] of programs because language designers sometimes include in their languages not only the conventional types which the average applications programmer wishes to use in his computations, but also types which reflect the ways and means to implement the language. Thus, in attempting to give a meaning to a program, we come face to face with machine or computational types such as locations (or "references", direct and indirect) and with functions associating "identifiers" with locations (i.e. "environments") and functions associating locations with their contents ("states").
>
> Certainly Semanol [the authors' semantic system], which is operational, must include such types in its conception of semantics. However, these types are fundamental to the ODM [Oxford Definition Method, i.e. Scott/Strachey method], and meanings in ODM turn out to be not purely functions from input data types to output data types, but rather functions from states to states or from environment-state pairs to states ..... it has the consequence that ODM definitions have an underlying machine-like aspect, abeit an abstract one, since they refer to, or are understood in terms of, states and state transitions. Indeed considerable attention is paid to the store, updating it, allocating and deallocating loca-

> tions etc. These machine-like constructs are basic to the ODM conception of semantics and they are used in all but the very simplest ODM descriptions. The fact that in ODM a state assumes a somewhat abstract shape, namely, as a function from locations to values, may make it more mathematical, but it does not in any way detract from its machine-like character."

Examples of this phenomenon can be found in the denotational description of almost any imperative language. Tennent [28], for example, gives an expression which denotes the meaning of an assignment statement. The expression involves the three variables σ, σ', and σ''. The first is the state of the store before the assignment is executed, the second the state of the store after the expression on the right of the assignment is evaluated, and the third the state of the store after the expression on the left is evaluated. The denotation of the assignment statement is the function which yields (given σ) the fourth and final state of the store.

It might seem at this point that we are contradicting what was said earlier by arguing now that the denotational and operational descriptions are essentially the same, but in fact there is no real contradiction. The two points of view need not be equivalent because the denotational view is potentially more abstract. In Scott's own words [24]

> "functions are independent of their means of computation and are hence "simpler" than the explicitly generated, step-by-step evolved sequences of operations on representations. In giving precise definitions of operational semantics there are always to be made more or less arbitrary choices of schemes for cataloguing partial results and the links between phases of the calculation ... and to a great extent these choices are irrelevant for a true "understanding" of a program.

Denotational semantics can be simpler and more abstract because it refers to the ends themselves, and not to complicated and arbitrary means to these ends.

This potential is not realized, however, when the method is applied to describe operationally based languages. The denotational semantics of a language is able to 'abstract out' operational considerations only to the extent that these considerations are a means to an end, e.g. the means to compute a function. The operationally based languages with their nasty features give programmers more direct control over the machine and allow them to bring about behavior which is not necessarily a means to a simple end. The cataloguing of results and the linking of computations become ends in themselves, and must therefore appear in the semantic description (in the form, for example, of stores and continuations). An assignment statement with side effects may be used by a programmer in the course of computing a simple object but by itself it is simply a complicated command which results in three changes in the state of a system.

This 'operational' tendency in the Scott/Strachey approach can already be seen in the original paper of Scott and Strachey where the authors use their techniques to design a simple language. Unfortunately, the language is imperative: it has assignment statements, expressions and procedures (both with side effects), and commands and locations as assignable values. The semantic equations for some of the simpler features are given in full, but with the assignment statement "... the sequence of events is more complicated. In this paper we shall not try to write the equation for [the meaning of assignment] but we can say in words more or less what happens". What happens, of course, is the usual complicated series of change of the state of the store, depending on the values (commands, locations) involved.

Scott and Strachey have given an example of the prescriptive

use of denotational techniques, but their language is nevertheless operationally inspired. Their semantics is (or could be) formally denotational but the approach is (has to be) operational in spirit - the meaning of constructs such as assignment is specified in terms of changes of "the internal states of our hypothetical machine". The mathematical tools developed by Scott and Strachey, in particular the theory of domain equations, are in essence being used to further the operational approach. The authors in the paper under consideration use a domain equation to specify (the domain of states of) an abstract machine whose memory locations are capable of storing commands and procedures as well as labels and values.

The basically operational nature of Scott and Strachey's example language is by no means accidental; rather it is a direct outcome of the authors' belief, expressed clearly in the first part of their paper, that programming languages are somehow, as a general rule, inherently operational. In their words

> "We begin by postulating that the interpretation of the language depends on the states of "the system". That is to say, computer-oriented languages differ from the mathematical counterparts by virtue of their dynamic character."

The evolution of the Scott/Strachey method has on the whole followed the directions set by its founders. In extending the method to handle more and more of the features of imperative languages the mathematical techniques have been used, on the whole, to develop more and more elaborate models of "the system" underlying these languages. It is certainly true that the formal Scott/Strachey description of, say, Algol 60, can be considerably simpler than a conventional operational one; in part this is because Algol does have a denotational element which can

be abstracted; but mainly it is because the theory of domains permits the specification of very abstract machines.

The 'Oxford method', when applied to imperative languages, is superior to (say), the Vienna method only because the Oxford Definition Language (our term) has a superior data type specification facility. Our objection is not to the theory of domains *per se* but to its use as the basis of a very abstract theory of automata.

At this point it is necessary for us, having said so much 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 own work on Lucid [2, 3, 4] allows us to make some contribution.

With the denotational approach the design of the language begins with the specification of the domains of semantic objects. The obvious question is, which domains? We have already seen that some domains are essentially machines, i.e. the elements of such domains are the states of a machine. If we base our design on such a domain, we will very probably find outselves taking the operational path. From our limited experience, it seems that the best strategy is to choose domains of simple conventional objects, e.g. 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 semantic 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 a 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 role of trusted advisor.

In the development of Lucid, our initial concern was with capturing the idea of iteration in a mathematical way, and 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 doesn't specify the operational behaviour, it is possible to use other iterative methods, such as data flow, 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 role. 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, by 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 of a distributed implementation. By contrast, the addition of "parallism" 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.

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, namely "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.

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, the 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. 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.

The denotational approach may at last offer some hope of solving what Reynolds [22] called "a serious unsolved problem" in language design, namely "the simultaneous achievement of simplicity and generality". It is possible that modern general purpose languages fail to be simple because they are operationally designed. It seems that no general language can be simple for the purpose of writing programs and

and also have a simple implementation. Operationally designed languages are by nature close to the machine and so place much of the burden of efficiency on the shoulders of the programmer. As a result, designers are forced to include a great number of extra features, constructs, options and so on, in order to give the programmer the desired control over the behaviour. As a result the languages (and programs) are anything but simple. The situation may well be different with the denotationally defined languages. 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".

It is not impossible for such languages (e.g. Lucid) to be both general and simple, because programmers are not required to specify behaviour. For these languages it is the implementation which must be complex, and may involve sophisticated analysis and transformation techniques. Fortunately these languages are mathematically so simple that these techniques could possibly be very effective - complicated or sophisticated implementations are not necessarily inefficient.

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, i.e. to implement, them. This is our prescription for semantics.

REFERENCES

1. Anderson, E.R., Belz, F.C. and Blum, E.K., "Issues in the Formal Specification of Programming Languages", Formal Description of Programming Concepts, E.J. Neuhold (ed.), North Holland Publishing Company, 1-30 (1978).

2. Ashcroft, E.A. and Wadge, W.W., "Lucid - A Formal System for Writing and Proving Programs", SIAM J. Comput., 5, No. 3 (1976).

3. Ashcroft, E.A. and Wadge W.W., "Lucid, a Nonprocedural Language for Iteration", CACM, 20, No. 7 (1977).

4. Ashcroft, E.A. and Wadge, W.W., "Structured Lucid", CS-79-21, Computer Science Department, University of Waterloo (1979).

5. Bjorner, D. and Jones, C.V. (eds.), "The Vienna Development Method", Lecture Notes in Computer Science, No. 61, Springer Verlag (1978).

6. Church, A., "The Calculi of Lambda-Conversion", Am. Math. Studies, 6, Princeton (1943).

7. Dennis, J., "Opening Remarks", Formal Description of Programming Concepts, E.J. Neuhold (ed.), North Holland Publishing Company, xi-xvii (1978).

8. Dijkstra, E.W., "GOTO Statement Considered Harmful", CACM 11, No. 3, 147-148 (1968).

9. Dijkstra, E.W., "A Somewhat Open Letter to EAA", EWD 614, Burroughs, The Netherlands (1977).

10. Dijkstra, E.W., "A Discipline of Programming", Prentice-Hall (1976).

11. Friedman, D. and Wise, D., "Applicative Multiprogramming", Technical Report 72, Indiana University, Computer Science Department (1979).

12. Gordon, M., "Operational Reasoning and Denotational Semantics", IRIA Colloquium on Proving and Improving Programs, 83-98, Arc et Senans (1975).

13. Gordon, M., "The Denotational Description of Programming Languages", Springer Verlag (1979).

14. Hoare, C.A.R., "Communicating Sequential Processes", CACM, 21, pp. 666-678 (1978).

15. Kahn, G. and McQueen, D.B., "Coroutines and Networks of Parallel Processes", IFIP 77, pp. 993-998 (1977).

16. Kleene, S.C., "Introduction to Meta Mathematics", Van Nostrand, Princeton (1952).

17. Kowalski, R.A., "Logic for Problem Solving", North Holland Elsevier (1979).

18. May, M.D., Taylor, R.J.B. and Whitby-Strevens, C., "EPL: an Experimental Language for Distributed Computing", Proceedings of "Trends and Applications: Distributed Processing", National Bureau of Standards, pp. 69-71 (May 1978).

19. McCarthy, J. et al, "LISP 1.5 Programmer's Manual", M.I.T. Press (1962).

20. Milne, R. and Strachey, C., "A Theory of Programming Language Semantics", Chapman and Hall (1976).

21. Plotkin, G.D., "A Power Domain Construction", pp. 452-487 of the SIAM Journal on Computing, 5 (1976).

22. Reynolds, J., "GEDANKEN - A Simple Typeless Language Based on the Principles of Completeness and the Reference Concept", CACM 13, No. 5, pp. 308-319 (1970).

23. Richards, M., "The Portability of the BCPL Compiler", Software Practice and Experience, 1, No. 2 (1971).

24. Scott, D., "Outline of a Mathematical Theory of Computation", Proc. 4th Am. Princeton Conf. on Information Sciences and Systems, pp. 169-176 (1970).

25. Scott, D.S., and Strachey, C., "Toward a Mathematical Semantics for Computer Languages", pp. 19-46 of Proceedings of the Symposium on Computers and Automata (ed. Fox. J.), Polytechnic Institute of Brooklyn Press, New York (1971); and Technical Monograph PRG-6, Programming Research Group, University of Oxford (1971).

26. Stoy, J.E., "Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory", M.I.T. Press (1977).

27. Tennent, R.D., "The Denotational Semantics of Programming Languages", pp. 437-453 of Communications of the ACM, 19 (1976).

28. Tennent, R.D., "Language Design Methods Based on Semantic Principles", Acta Informatica, 8, pp. 97-112 (1977).

29. Turner, D.A., "A New Implementation Technique for Applicative Languages", Software Practice and Experience, 9, pp. 31-49 (1979).

30. Vuillemin, J., "Correct and Optimal Implementations of Recursion in a Simple Programming Language", Proc. 5th ACM Symp. Theory of Computing, pp. 224-239 (1973).