The Practical Importance of Formal Semantics

Pierre America
Philips Research Laboratories
Eindhoven, the Netherlands

Abstract

In this paper some possible directions are sketched along which formal semantics can be applied to practical problems. The most promising application areas are language implementation, language design, and program design. It is argued that in order to exploit these possibilities, a closer cooperation between semanticists and practitioners in the above areas is needed.

1 Introduction

The study of formal semantics of programming languages is often considered to be a field of little practical relevance. This point of view is taken by many people that are not familiar with these techniques. Sometimes they are scared off by the first mathematical formula in the text. Others read patiently through all the definitions and even come to understand them, but finally reach the conclusion that the semantics of a real program is a far too complex mathematical object for anyone to understand, and that therefore the whole field cannot lead to any useful result.

Even sadder is the viewpoint of many researchers in the field itself. Many of them have little or no interest for the practical relevance of what they are doing and are completely satisfied with published papers as the sole outcome of their research. Others have a wrong impression of what 'practical relevance' means because they have little or no contact with the people that actually build computer systems and for whom their results could be valuable.

In this paper, I shall try to demonstrate that formal semantic techniques can be useful in many fields of computer science. We shall see that it is not easy to apply these techniques, and that the only possibility is for semanticists and 'practical' people to work closely together, so that each gets a better understanding of the other's problems and capabilities.

2 Semantic formalisms

There are several broad categories in which formal semantic techniques can be divided: operational, denotational, and axiomatic semantics (for a more extensive overview, see

[Pag81]). Sometimes it is said that operational semantics is useful for the language implementor, denotational semantics for the language designer, and axiomatic semantics for the one who writes programs in the language. We shall see below that this is not the complete truth.

Broadly speaking, one can say that *operational semantics* involves some kind of abstract machine and it describes the meaning of the program in terms of the actions that this machine performs. The nature of this abstract machine differs considerably between the several operational formalisms. In VDL [Weg72], the machine and the instructions it executes both have the structure of a tree, where labels can be attached to nodes and edges. The operational semantics presented in [Bak80] uses states, which are functions from variables to values, and state transformations, functions from states to states, which describe the execution of a complete statement. The sequences of states resulting in this way come very close to what in other contexts is called denotational semantics.

A very popular approach is the use of *transition systems* in the 'Structured Operational Semantics' (SOS) style of [HP79,Plo81,Plo83]. Here the meaning of a program is expressed as a sequence of *transitions* between *configurations*. A configuration typically consists of a state, mapping variables to values, plus that part of the program that is still to be executed. A *transition relation* describes the collection of all possible transitions from one configuration to another. Typically this relation is defined inductively by using axioms and rules. Because of the fact that the program to be executed is contained in the configurations, the axioms and rules defining the transition relation can be closely connected to the syntactic structure of the programming language.

The essence of *denotational semantics* lies in the principle of *compositionality:* The semantics takes the form of a function that assigns a meaning, an element of some mathematical domain, to each individual language construct. The principle of compositionality says that this function should be defined in such a way that the meaning of a composite construct does not depend on the form of the constituent constructs, but only on their meaning.

There are several kinds of mathematical domains being used as the ranges of these meaning functions. In most instances some kind of limit construction is necessary to describe infinite behaviour and the structure of the domain should enable such a limit construction. Most forms of denotational semantics use some order-theoretic structure and among these complete partial orders (CPOs, see. e.g., [Bak80]) and several kinds of lattices [Sco76] are the most popular. A quite different approach is the use of complete metric spaces [BZ82].

Finally, instead of directly assigning a meaning to a program, *axiomatic semantics* gives a description of the constructs in a programming language by providing axioms that are satisfied by these constructs. The most popular formalism to express these axioms is Hoare logic [Hoa69]. Here a program or statement is described by two logical assertions: a precondition, describing the state of the system before executing the program, and a postcondition, describing the state after execution. Using such an axiomatic description of the programming language, it is possible, at least in principle, to prove the correctness of a program with respect to a specification.

3 Language implementation

Intuitively the most obvious relationship between formal semantics and practical applications is concerned with the implementation of programming languages. One would hope that having taken all the trouble to give a formal semantics to a programming language, it would be possible to arrive quickly at an implementation for this language. However, things are not so easy as that.

The most obvious candidate for transformation into implementations is operational semantics: this type of semantics already takes the form of describing the actions performed by an abstract machine. Unfortunately, it is not so easy to relate the abstract machine model used by most versions of operational semantics to a concrete machine. Concrete machine architectures are always designed with the objective to implement them cheaply and efficiently in hardware. Moreover, their finiteness imposes many limits on their power, which of course do not apply to abstract machines.

At the moment, I am not aware of any attempt to implement the tree-like structures of the VDL abstract machines on ordinary hardware. For SOS-style transition systems [HP79,Plo81,Plo83], however, there is an interesting possibility: It is relatively easy to write the axioms and rules that describe the transition relation in the form of Horn clauses. In this way they could in principle be interpreted by a system such as Prolog. There are, of course, a few problems with this approach. The most obvious one pertains to the performance of this implementation, which of course can never be very good, but it may be sufficient for a first prototype implementation of a new language. The second one is more fundamental: the Prolog system itself is not complete, in the sense of being able to find all the solutions of the given goal to which the axioms give rise. Therefore the order and the form in which the axioms are written are very important for the execution of the program, and this effectively prohibits a fully automatic implementation along these lines. There exist complete theorem provers that are able to deal with Horn clauses, but these are orders of magnitude slower than the best Prolog implementations and they use much more memory.

The last problem is common to all semantics-based implementation techniques: how to deal with nondeterminism? A transition system may allow several different transitions from a given configuration. In principle, a single program may give rise to many possible transition sequences among which one should be chosen. Sometimes the semantics imposes requirements on the whole transition sequence, for example that it should be fair, or that it should not lead to a dead end. It is clear that this kind of extra requirements is extremely difficult to deal with in the above approach. Prolog traverses its search space in a depth-first way, which is not fair in most cases. Backtracking out of 'wrong' transition sequences is only possible if they are finite. The best solution to this problem would be if the original transition system were formulated in such a way that all its resulting transition sequences are acceptable.

Despite all appearances, denotational semantics may provide at least as good a basis for the automatic generation of language implementations as operational semantics. This may not be the case for the direct form of denotational semantics as described in [Bak80, chapters 1–9] and [Sto77, chapters 1–10], because the functions from states to

states resulting from these definitions are very cumbersome to compute. However, for semantic definitions using *continuations* opportunities appear to be better. The technique of continuations can best be summarized by saying that in defining the meaning of a statement or expression one considers the meaning of the following statements and expressions as a parameter. The output of the semantic function is then the meaning of the statement/expression under consideration plus all the following ones. This combined meaning need not be a function from states to states; it can equally well be a sequence of items to be written to the output file, for instance. For a nice introduction to continuation semantics, see [Gor79].

Once an abstract structure for the syntactic constructs in a language has been fixed (this could form the output of a parser), it is very easy to write the semantic definitions in a functional programming language (e.g., Miranda [Tur85]). If moreover the semantics makes clever use of continuations and the functional language is evaluated in a 'lazy' way, then the expression giving the semantics of a program can be executed directly, giving the output of the program as soon as it has been computed. This even works for nonterminating programs. I have done this experiment a few years ago with the continuation semantics for the language SMALL in [Gor79], using an interpreted SASL implementation [Tur79]. The resulting performance was acceptable for a prototype implementation and the recent advances in the implementation of functional languages [PJ87] would probably lead to an improvement by several orders of magnitude. Other approaches roughly along this line have been done before [Mos76].

Nevertheless, the performance that can be achieved by directly executing traditional denotational descriptions in this way is still much less than the performance exhibited by ordinary hand-written compilers. The most important reason for this is that in the abovementioned approach, operations that can be efficiently performed by computer hardware, such as reading or changing the contents of storage cells, are first mapped to relatively complicated mathematical notions (such as a state: a function from variable names to values), which must then be mapped back again to a concrete computer architecture. This detour can be avoided by splitting the semantic description in several levels. The lowest level provides those basic operations provided by the underlying architecture. It should comprise both a precise mathematical description and a direct translation to machine code. The higher levels can now make use of these basic operations. In this way the higher level description can be shorter, easier to understand, and easier to translate into an efficient implementation. This approach is being explored at several places [MW86b,LP87], with a resulting performance comparable with commercial, hand-written implementations.

Automatic generation of implementations is not the only way in which formal semantics can help in language implementation. Also for hand-crafted implementations it is often very helpful to relate them to a formally defined semantics. Here the basic problem is ensuring the *correctness* of the implementation. A prerequisite for the formal approach is, of course, the existence of some formal semantic description of the programming language (see section 4).

To my knowledge, no complete implementation of a realistic programming language has yet been verified completely formally. But less ambitious goal are certainly

reachable. Among others, a formal semantic model can be used to justify a certain implementation or optimization technique. For example, in [Vaa86] it has been shown that in the implementation of a parallel object-oriented language, the use of message queues leads to a correct implementation with respect to the semantics, which imposes certain fairness requirements on the execution of a program. Moreover, it is shown that certain built-in data types, such as integers, that are conceptually modelled in the language as full-fledged objects, can in fact be implemented in the traditional way without any harm to the semantics. More sophisticated optimizations form the subject of current research. For another example, we refer to the techniques used for strictness analysis in functional programming languages, which are justified by a formal semantic study using abstract interpretation [HBPJ88].

4 Language design

Another activity where formal semantic methods can be applied fruitfully is the design of programming languages. The main purpose here is to provide a formalism in which the outcome of this language design process can be represented. Unfortunately, most descriptions of programming languages are informal, using natural language (e.g., [ANS83,BSI82]. It turns out that, whatever care is taken to make such a description precise and unambiguous, there always remain some points that are open for several different interpretations [Spe82,WSH77]. If we want to describe a certain aspect of a programming language precisely, without any possibility for misinterpretations, the only option is giving a *formal* description.

For describing the syntax of the language, the Backus-Naur form (BNF) has been in common use for many years, sometimes supplemented by much less popular formalisms, such as attribute grammars or two-level grammars, to capture the non-context-free aspects of the syntax. All these formalisms do is defining what is a legal program in the language and what is not; they do not tell us what a program means.

Complete formal semantic descriptions of commonly used programming languages are very rare (but see, e.g., [AH82,MW86a]). It is certainly not easy to give a formal description, be it operational, denotational, axiomatic, or otherwise, of programming languages like Pascal, C, or Ada. Sometimes it is said that the reason for this is that the formalisms for specifying the semantics of a language are not sufficiently developed. This is only partly true. Another reason, at least equally important, is that many of the languages that are currently in use have been designed by language implementors, with a clear view of the implementation of the language in mind, but with far less attention for an independent description of the exact meaning of programs. Some language designers even think that they have sufficiently defined the language when they have constructed one implementation for it.

The lack of appreciation of formal semantic techniques in circles of language designers is even more unfortunate because these techniques can do much more than just provide an unambiguous notation for expressing the meaning of programs. Trying to give a formal semantics for a language is a very good way of detecting weak points in the language design itself. 'Insecurities' as mentioned in papers like [Spe82,WSH77]

are pinpointed very quickly as soon as one tries to describe the meaning of programs in a formal way. At this point it would be possible to give a long range of examples of language design errors that could have been avoided. Of course, the presence of such problems can be demonstrated effectively by giving suitable example programs, but their absence can only be proved by a formal semantics.

The worst form of language design errors are the completely unintentional ones, where the language behaves in a way that is not expected and even less desired by its designer. For example, the Eiffel programming language was claimed to have a completely safe static type checking mechanism [Mey87], but closer analysis shows that this is not true [Coo89]. this kind of errors can be detected by defining a formal semantics for the language and rigorously proving the desired property. Sometimes it suffices to construct a simplified model, which concentrates on the particular aspect under concern (in the case of Eiffel's type system, the analysis in [Car88] clearly indicates some of the problems and possible solutions).

Another kind of errors is the result of conscious decisions by the language designer, where these decisions are nevertheless undesirable because they harm the readability, writability, or implementability of the language. It can be said that operational semantics, denotational semantics, and axiomatic semantics, in this order, are increasingly sensitive instruments for detecting this kind of problems in language design. A good rule of thumb says that there is such a problem whenever the description of a language feature that is considered to be unessential requires a special adaptation of the overall semantic model used to describe the language. For example, in a language like Pascal, the addition of a goto statement requires a substantial adaptation of the denotational description (see, e.g., [Bak80, chapter 10]). Another common phenomenon is the presence, in a language that is meant to be deterministic, of 'unspecified' behaviour in certain circumstances. A denotational description of such a language would have to be adapted to some form of nondeterminism, which indicates that there is a problem. In practice, such unspecified behaviour can form an important obstacle in porting programs from one implementation to another.

While for many programming languages the goal of a clean denotational semantics has not been reached (in many cases it has not even be considered!), the actual goal should be a satisfactory *axiomatic* semantics, which is even more difficult to attain. As we shall in the next section, the usability of a programming language for the systematic construction of reliable programs will depend increasingly on the possibility of applying formal techniques during program design, and for this a good formal proof system for the language is a necessary prerequisite.

5 Program design

Finally, an important area where formal semantic techniques might be applied is in the construction of programs. It is often said that formal semantics, in particular of the operational or denotational kind, is useless because the meaning of a realistic program is such a complex mathematical object that nobody can understand it. The latter statement is certainly true in most cases, but nevertheless the conclusion is not

justified.

A formal definition of a programming language, including both syntax and semantics, could be a concise and unambiguous reference document that can answer all questions about the language, in the same way as a BNF description is a generally accepted and appreciated form of describing the context-free syntax of a language. The question is whether this document is readable by a programmer without several years of studying complicated mathematics. For many forms of denotational semantics this is indeed a problem, but not so severe as one would think: Often the 'complicated mathematics' is only needed to justify the definitions, e.g., by demonstrating that a certain equation indeed has a unique (or otherwise distinguished) solution. In reading the semantics it is mostly possible to ignore these mathematical issues completely (as is demonstrated, for example, in [Gor79]). However, it must be admitted that even then denotational descriptions of programming languages are relatively hard to understand.

This is not necessarily the case for other forms of formal semantics, however. For example, the SOS style of operational semantics [HP79,Plo81,Plo83] employs a set of axioms and rules for defining the set of possible transitions between configurations. Broadly speaking, an axiom describes the basic functionality of a single kind of statement or expression, whereas a rule describes in which way the constituents of a composite construct are evaluated. Because of the strong connection with the syntactic structure and the fact that the mathematics involved is usually not very complicated, this form of semantics leads to descriptions that are very easy to read, even for non-specialists. For example, the operational semantics for the language POOL [ABKR86] turned out to be relatively easy to understand to prospective programmer and implementors of the language.

The form of semantics that is directed most specifically towards the programmer, axiomatic semantics, has not yet become very successful, unfortunately. The reason for this is probably that it is even more difficult to give a clear axiomatic semantics to a language that was not designed with this in mind, than to provide it with a denotational semantics. As most commonly used languages do not even lend themselves to a reasonable denotational semantics, a complete axiomatic semantics is completely out of the question. For some realistic programming languages an attempt has been made to describe them axiomatically, but either the correctness and completeness of the description was an open question [LGH*78] or only a part of the language was described [HW73]. (For the programming language POOL an axiomatic description has been given [Boe89], but it must be admitted that this is not easy to read for an average programmer.)

Nevertheless, formal techniques are playing an increasingly important role in the design of software systems, especially in situations where the reliability of the software is vital. Automatic support for many of the manipulations of the formal objects involved is gradually becoming available. It is clear that at the basis of these formal techniques there should be a formal axiomatic description of the programming language, or in other words, a formal proof system. It is not absolutely necessary that this proof system is *complete*, in the sense that it can prove all correct assertions about any program, but it should, of course, deal with all aspects of the programming

language, and it absolutely must be *sound*, which means that everything that can be proved in the proof system is actually true.

In the mean time, even if a complete axiomatic semantics of the programming language in use is not available, a certain knowledge of the techniques of formal program verification can give the programmer excellent guidelines in the design of his program. Reasoning quasi-formally in terms of assertions describing the state at certain points of the program can provide the support necessary to get subtle pieces of code correct.

6 Conclusions

We have seen that formal semantic techniques can be usefully applied in language implementation, language design, and program design. It turned out that the traditional opinion of operational semantics being mostly useful for language implementors, denotational semantics for language designers, and axiomatic semantics for programmers, does no longer correctly represent the state of affairs: Denotational semantics provides a good starting point for automatic language implementations, SOS-style operational semantics can provide the programmer with a concise and accurate description of what the language constructs do, and a clean axiomatic description should be one of the foremost goals in language design.

There are many opportunities for applying formal semantics, but it is clear that in order to use them, much more cooperation is needed between researchers in the field of semantics and the people active in the areas of language implementation, language design, and programming. As long as semantics is being considered as a purely academic discipline with no relevance to the real world, it will be just that, and it takes more than just proclaiming the practical importance of semantics in order to change this situation.

References

[ABKR86[Pierre America, Jaco de Bakker, Joost N. Kok, and Jan Rutten. Operational semantics of a parallel object-oriented language. In *Conference Record of the 13th Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 13–15, 1986, 194–208.

[AH82] Derek Andrews and Wolfgang Henhapl. Pascal. In Dines Bjørner and Cliff B. Jones, editors, *Formal Specification and Software Development*, 175–251, Prentice-Hall International, Englewood Cliffs, New Jersey, 1982.

[ANS83] ANSI. *The Programming Language Ada Reference Manual*, ANSI/MIL-STD-1815A-1983, approved 17 February 1983. Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[Bak80] J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1980.

[Boe89] Frank S. de Boer. *A proof system for POOL.* Technical Report, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, 1989. To appear.

[BSI82] BSI. *Specification for the computer programming language Pascal.* Standard BS 6192, British Standards Institution, Herts, United Kingdom, 1982.

[BZ82] J. W. de Bakker and J. I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Coo89] William Cook. A proposal for making Eiffel type-safe. In *ECOOP'89: European Conference on Object-Oriented Programming*, Nottingham, England, July 10–14, 1989. To appear.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[HBPJ88] Chris Hankin, Geoffrey Burn, and Simon Peyton Jones. A safe approach to parallel combinator reduction. *Theoretical Computer Science*, 56(1):17–36, January 1988.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[HP79] Matthew Hennessy and Gordon Plotkin. Full abstraction for a simple parallel programming language. In J. Bečvář, editor, *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science*, 1979, 108–120, Lecture Notes in Computer Science 74, Springer-Verlag.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[LGH*78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10:1–26, 1978.

[LP87] Peter Lee and Uwe Pleban. A realistic compiler generator based on high-level semantics. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, München, West Germany, January 21–23, 1987, 284–295, ACM.

[Mey87] Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2):85–99, February 1987.

[Mos76] Peter D. Mosses. Compiler generation using denotational semantics. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science*, 1976, 436–441, Lecture Notes in Computer Science 45, Springer-Verlag.

[MW86a] Peter D. Mosses and David A. Watt. *Pascal: action semantics — towards a denotational description of ISO Standard Pascal using abstract semantic algebras.* Draft Version 0.3, Aarhus University, Computer Science Department, Aarhus, Denmark, September 5, 1986.

[MW86b] Peter D. Mosses and David A. Watt. *The use of action semantics.* Report DAIMI PB-217, Aarhus University, Computer Science Department, Aarhus, Denmark, August 1986. Appeared in [Wir86].

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages.* Prentice-Hall, 1981.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Plo81] Gordon D. Plotkin. *A structural approach to operational semantics.* Report DAIMI FN-19, Aarhus University, Computer Science Department, Aarhus, Denmark, September 1981.

[Plo83] Gordon D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II*, 1983, 199–223, North-Holland.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, September 1976.

[Spe82] David Spector. Ambiguities and insecurities in Modula-2. *ACM SIGPLAN Notices*, 17(8):43–51, August 1982.

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

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Tur85] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, 1985, 1–16, Lecture Notes in Computer Science 201, Springer-Verlag.

[Vaa86] Frits W. Vaandrager. *Process algebra semantics for POOL.* Report CS-R8629, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, August 1986.

[Weg72] Peter Wegner. The Vienna Definition Language. *ACM Computing Surveys*, 4(1):5–63, 1972.

[Wir86] M. Wirsing, editor. *Formal Description of Programming Concepts III: Proceedings of the Third IFIP WG 2.2 Working Conference*, Gl. Avernæs, Ebberup, Denmark, August 25–28, 1986, North-Holland.

[WSH77] J. Welsh, W. J. Sneeringer, and C. A. R. Hoare. Ambiguities and insecurities in Pascal. *Software—Practice and Experience*, 7, 1977.