Operational Semantics: Concepts and their Expression

Cliff B. Jones

School of Computing Science,
The University of Newcastle upon Tyne
NE1 7RU England.
`cliff.jones@ncl.ac.uk`

Keywords: Formal Semantics, Operational Semantics, Programming Languages

1 Introduction

One of the earliest approaches to giving formal semantics for programming languages was "operational semantics". Enthusiasm for this approach has waxed and waned (not least in my own mind). The main objective of this paper is to tease apart some concepts involved in writing such operational descriptions and (as separately as possible) to discuss useful notations. A subsidiary observation is that "formal methods" will only be used when their cost-benefit balance is positive. Here, learning mathematical ideas that are likely to be unfamiliar to educated software engineers must be considered as a cost; the benefit must be found in understanding, manipulating and recording ideas that are important in software.

At the ETAPS-WMT event in Warsaw, Niklaus Wirth reminded the audience of the benefits of the syntax notation known as "Extended Backus-Naur Form". This paper reviews some of the research in documenting the *semantics* of programming languages. It is argued that "operational semantics" can provide –for limited learning cost– considerable benefit in terms of understanding and experimenting with aspects of languages. The balance between identifying key concepts and choosing apposite notations is explored as are doubts about the *practical* value of "denotational semantics".

The main trends in the progress of language semantics can be summarised briefly. There is no intention here to attempt a complete history of research on formal semantics; rather, enough is recorded to set the context for what follows.

The publication of ALGOL-60 [BBG$^+$63] essentially solved the problem of documenting the syntax of programming languages. Of course, "BNF" was influenced by prior work on natural languages and there have been subsequent refinements and alternative presentations. But "(E)BNF" offers an adequate notation for defining the set of strings to be regarded as comprising a language. There are further bonuses in that (restricted forms of) grammars can be used to generate mechanically (efficient) *parsers* for their strings.

The most significant problem left outstanding with respect to the texts of programs (as distinct from their semantics) was how to define *context dependencies* such as the relationships required between the declaration and use of variable names. Chomsky's "context dependant" grammars did not offer perspicuous descriptions so a number of new approaches were worked out including attribute grammars [Knu68], the "two-level" grammars which evolved together with the ALGOL-68 proposal [vWMPK69] and the "dynamic syntax" idea in [HJ74]. Context dependencies are not the main concern here but a workable solution to recording them is mentioned in Section 3.

A far more important challenge was how to record the semantics of programming languages. A key early event was the conference organised by Heinz Zemanek at Baden-bei-Wien in 1964; the proceedings [Ste66] of this conference provide a wonderful snapshot not just from the papers but also because of the record of the discussions.[1] The most relevant (to the current paper) contribution was John McCarthy's formal description of "micro-ALGOL" [McC66]. Essentially he defined an "abstract interpreter" which took a *Program* and a starting state[2] and delivered the final state, thus:

$$\mathit{Program} \times \Sigma \to \Sigma$$

was defined in terms of recursive functions on statements and expressions

$$\mathit{exec}\colon \mathit{Stmt} \times \Sigma \to \Sigma \text{ and } \mathit{eval}\colon \mathit{Expr} \times \Sigma \to \mathit{Value}$$

Both *Program* and Σ were defined at an abstract level and [McC66] describes how to use an "abstract syntax" of a language (albeit with specific axioms to relate the constructor and selector functions for objects).

The IBM Vienna group list McCarthy (along with Cal Elgot and Peter Landin) as a major influence on their ambitious attempt to define PL/I using an abstract interpreter. It must be remembered that PL/I offered many intellectual extensions (as well as a deplorable number of gratuitous complexities) over any language that had been modelled at that time. Among the new features of relevance here are:

1. variable scope
2. exceptional statement sequencing (both the derided "goto" and a specific exception mechanism called "on units")
3. freedom of storage layout in terms of the way in which small data objects are packed into, say, structures (see [BW71]).
4. concurrency – which of course introduced non-determinacy

Each of Points 1–3 deserves more discussion than there is room for in the current paper but the principal interest below derives from the reaction to point 4 (cf. Section 2.2).

The abstract machines used in the Vienna descriptions of PL/I[3] were "operational" descriptions of a major programming language. Other languages were

[1] This was the first of a series of IFIP "Working Conferences" that have done so much to advance Computing Science.

[2] In the simplest case the states (Σ) are mappings $\mathit{Id} \xrightarrow{m} \mathit{Value}$.

[3] The reports from the 1960s –known as "ULD-3" internally and referred to externally as the "Vienna Definition Language" (VDL)– are unlikely to be easily located; more accessible descriptions are [LW69,Luc81].

also defined using VDL. These definitions were not only impressive in their own right (and *did* result in cleaning up some aspects of PL/I); they were also used as a basis from which compiler designs could be justified (see [JL71] for itself and for more extensive references). Unfortunately VDL definitions tended to use huge states which included control trees that coped with both abnormal sequencing and concurrency. These "grand states" presented gratuitous difficulties in proofs about VDL descriptions. One of the measures considered here of the usefulness of semantic descriptions is the extent to which they make reasoning straightforward.

Some of the difficulties were overcome in the "functional semantics" used in [ACJ72]. But the main thrust of research was shifting to "denotational semantics" (see [Sto77]). The work on a compiler for ECMA/ANSI PL/I resulted in IBM Vienna shifting to denotational semantics [BBH$^+$74]. In contrast to the "continuations" used in Oxford to model abnormal jumps (e.g. goto statements), "VDM" (now "Vienna *Development* Method") used an "exit" mechanism. Once again, our objective was to use the definition as the basis for compiler development (cf. [BJ78] for itself and further references).

In 1981, Gordon Plotkin took a leave of absence in Aarhus and produced the technical report [Plo81] on "Structural Operational Semantics".[4] This widely copied contribution revived interest in operational semantics. *For the thrust of the current paper*, the most important contribution of [Plo81] was the step to using a "rule notation"; for example:[5]

$$\frac{(\mathit{rhs}, \sigma) \xrightarrow{e} v}{(\mathit{mk\text{-}Assn}(\mathit{lhs}, \mathit{rhs}), \sigma) \xrightarrow{s} \sigma \dagger \{\mathit{lhs} \mapsto v\}}$$

Where $\xrightarrow{e}$ can for now be viewed as a function $\mathit{Expr} \times \Sigma \to \mathit{Value}$ and

$\xrightarrow{s}: \mathit{Stmt} \times \Sigma \to \Sigma$

The advantage of the move to such a rule presentation is explored in Section 2.2.

2 Some key concepts

Poor notation can cloud important concepts but notation alone cannot rescue inadequate concepts. For this reason the current paper emphasises the concepts

[4] This material, together with a companion note on its origins, are finally to be published in a journal; it has been one of the pleasures of writing the current paper that I was able to compare recollections with Gordon while he was writing [Plo03].

[5] Where needed, specific notation from VDM is used in the current paper: even if unfamiliar, its intention should be obvious. For example, VDM defines objects like

$$\begin{array}{ll} \mathit{Assn} :: & \mathit{lhs}: \mathit{Id} \\ & \mathit{rhs}: \mathit{Expr} \end{array}$$

which gives rise to a constructor function yielding tagged values

$\mathit{mk\text{-}Assn}: \mathit{Id} \times \mathit{Expr} \to \mathit{Assn}$

which have been developed to model aspects of programming languages. The examples used below are illustrative of the sort of concepts which can make a description reveal the essence of the language being discussed.

2.1 Environments

In languages of the ALGOL family, the nesting of blocks and procedures introduces different scopes for identifiers thus permitting the same identifier to refer to different variables. Furthermore the ability to pass arguments "by location" ("by reference") makes it possible for different identifiers to refer to the same variable (or location). This gave rise to the idea of splitting the obvious map

$$Id \xrightarrow{m} Value$$

into two with an abstract set of locations (*Loc*) representing the equivalences over identifiers, thus

$$Env = Id \xrightarrow{m} Loc$$

$$\Sigma = Loc \xrightarrow{m} Value$$

The separation of such an environment from the state is an important aid to making properties of a language definition obvious: [JL71] contained a proof that a standard compiler method for referring to variables was correct (corresponded to the language description); one of the most tedious lemmas was that the environment after executing any statement was identical to that before such execution. This proof was only necessary because the environment and state mappings were bundled together. It really is the case that

$$Stmt \times Env \times \Sigma \to \Sigma$$

says more than

$$Stmt \times (Env \times \Sigma) \to (Env \times \Sigma)$$

This issue has been referred to as "small state vs. grand state". With hindsight, it is probable that one of the main attractions of denotational semantics was that it encouraged "small state" definitions.

The concept of environments and the abstract set of locations make it delightfully easy to illustrate the distinctions between different parameter passing modes ("call by value", "call by reference", "call by value/return", "call by name"). Furthermore, careful modelling of –say– *ArrayLoc* and *StructLoc* can result in a collection of semantic objects which convey a lot of information about a language without even looking at the semantic rules.

2.2 Modelling non-determinacy

Many features of programming languages give rise to non-determinacy in the sense that more than one state can result from a given (program and) starting state. Examples include specific non-deterministic constructs such as guarded commands, freedom of order of evaluation of expressions (in a language with side-effects) and –most importantly– concurrency. Whatever the origin of the non-determinacy, it is clear that *exec*: *Stmt* $\times \Sigma \rightarrow \Sigma$ does not capture the semantic intent. A move to producing a set of states, as in *exec*: *Stmt* $\times \Sigma \rightarrow \mathcal{P}(\Sigma)$ is not convenient because of the need to ramify the combinations. The key advantage of a rule format such as

$$\frac{(s_1, \sigma) \xrightarrow{s} \sigma'}{(s_1 \ \mathbf{or} \ s_2, \sigma) \xrightarrow{s} \sigma'}$$

$$\frac{(s_2, \sigma) \xrightarrow{s} \sigma'}{(s_1 \ \mathbf{or} \ s_2, \sigma) \xrightarrow{s} \sigma'}$$

is that it provides a natural way of expressing the relation

$$\xrightarrow{s}: \mathcal{P}((\mathit{Stmt} \times \Sigma) \times \Sigma)$$

With some slight penalty (see Section 4.1), this natural expression extends well to concurrent languages. The advantage of the rule format appears to be that the non-determinacy has been factored out to a "meta-level" at which the choice of order of rule application has been separated from the link between text and states. For this reason, the complications of writing a function which directly defines the set of possible final states are avoided. Here is a case where the notation used to express the concept of relations (on states) is crucial.

It is however possible to express relations in other ways (e.g. direct use of union and relational composition or some sugaring thereof) and Section 2.4 discusses the use of "combinators".

2.3 More on environments

The importance of environments was emphasised in Section 2.1. An obvious way of incorporating them into the rules is to write (with ρ: *Id* $\rightarrow$ *Loc*) something like

$$\frac{(\mathit{rhs}, \rho, \sigma) \xrightarrow{e} v}{(\mathit{mk\text{-}Assn}(\mathit{lhs}, \mathit{rhs}), \rho, \sigma) \xrightarrow{s} \sigma \dagger \{\rho(\mathit{lhs}) \mapsto v\}}$$

There are however advantages in considering alternatives and various authors have chosen to use environments as decorations of the arrow marking the relation or to place the environment on the left of a turnstile. This last solution has the advantage of reducing the number of times that the (mostly constant) environment need be repeated.

There is also an interesting link between environments and non-determinacy. VDM definitions have always shown non-deterministic choice of locations at block entry. This apparently pedantic point is important in justifying compiling algorithms. It is of course an annoyance to anyone trying either to stay within the world of functions or who wishes to mechanise the execution of a definition.

2.4 Modelling abnormal sequencing

The issue of how to model programming language constructs which cause the sequence of execution to cut across the phrase structure is vexed. Even if one is omnipotent enough to cause the banning of the "goto" statement, there are other places that abnormal sequencing arises in most programming languages (e.g. returning a value from within a function (procedure), forms of exception handling that are put into the hands of the programmer).

As Plotkin observes in [Plo03], the history of concepts for modelling abnormal sequencing is surprising. As a reaction against the complicated control trees used in the VDL operational semantics descriptions of PL/I, [HJ70] proposed adding an explicit abnormal return value to all interpreting functions. This idea was pushed through the "functional semantics" in [ACJ72] but without notational finesse. The abnormal exit idea was then absorbed into the Vienna research on denotational semantics[6] with a way of hiding the fact that functions of type

$$\begin{array}{l} \mathit{Stmt} \to \mathit{Env} \to \Sigma \to (\Sigma \times [\mathit{Abn}]) \\ \text{where } [\mathit{Abn}] = \mathit{Abn} \mid \mathbf{nil} \end{array}$$

were being composed. These "combinators" have (as both Peter Mosses and Gordon Plotkin have observed) similar objectives to the "monads" proposed by Eugenio Moggi or indeed to Mosses' aim of separating descriptions of different language aspects [Mos92].

It is easy to express the abnormal exit idea in Ploktin-style rules (with $sl \in \mathit{Stmt}^*$ and $\mathit{Abn} = \mathit{Label}$)

$$\frac{\begin{array}{l}(\mathbf{hd}\, sl, \sigma) \xrightarrow{s} (\sigma', \mathbf{nil}) \\ (\mathbf{tl}\, sl, \sigma') \xrightarrow{s} (\sigma'', x)\end{array}}{(sl, \sigma) \xrightarrow{s} (\sigma'', x)}$$

$$\frac{(\mathbf{hd}\, sl, \sigma) \xrightarrow{s} (\sigma', \mathit{lab})}{(sl, \sigma) \xrightarrow{s} (\sigma', \mathit{lab})}$$

But this reverts to the notational messiness of [ACJ72] where the abnormal handling permeates the whole definition. There is however no reason why one could not adopt the idea of combinators in the operational world and define them over $\mathcal{P}(\Sigma \times (\Sigma \times \mathit{Abn}))$.

3 A propaedeutic view

The title of this section is in honour of [Tur85].

My enthusiasm for the new VDM approach led me to inflict –in the 1980s– programming language courses based on denotational semantics on several cohorts of students at Manchester University. Now, in Newcastle, I prefer to teach operational semantics and it might be interesting to review some of the content and the reasons for moving away from denotational semantics.

Specific technical choices for my current courses are

[6] For a fuller discussion of this see [Jon01].

– Use of *Abstract Syntax* in preference to concrete syntax: although concrete syntax is convenient for small definitions, it really does not scale up to larger languages; furthermore pattern matching with abstract objects gives a nice way of defining functions (or rules) by cases.
– Context Conditions are defined as recursive predicates over abstract programs (and static environments); this appears to be an intuitive way of separating issues like strong typing.
– Plotkin-style rules are used (rather than some form of combinators over relations as discussed in Section 2.2).
– We separate as far as possible different aspects from programming languages so that the students never see a complete definition of a language with –say– concurrency *and* exception mechanisms.

The objective is to show the students how to read a description of a large language and equip them to be able to experiment with ideas about languages before writing compilers (or inflicting languages on users).

We have had interesting debates about tool support for such definitions. The IFAD VDM Toolset enables students to run functional descriptions. Some students benefit from this; rather more seem to feel that the level of detailed fiddling distracts them from the main ideas. I am reluctant to push for the use of a tool which can only (easily) support the deterministic case. Much more interesting is the work of Tobias Nipkow and his colleagues in Munich that can be thought of as using the rules of the language as an extended logical frame which adds the ability to make deductions about the state relation of a program (see [KNvO$^+$02]). An interesting challenge here is whether it is easy to prove results about the limit of behaviour of a non-deterministic program.

4 Research challenges

4.1 The need for configurations

The argument in Section 2.1 for separating environment information (which only changes at block or procedure boundaries) from state (which can change on any assignment statement) is compelling. It is therefore a cause of some distress that concurrent execution forces a handling of program text which fails to make clear the limitations of how it can be changed. Consider the execution of a concurrent construct $sl_1 \mid\mid sl_2$ in which the first statement of either sl_i sequence can be executed. One is forced to write the rules in terms of some sort of "configuration" which shows which statements remain to executed – for example

$$\frac{(\mathbf{hd}\, sl_1, \sigma) \xrightarrow{s} ([\,], \sigma')}{((sl_1 \mid\mid sl_2), \sigma) \xrightarrow{s} ((\mathbf{tl}\, sl_1 \mid\mid sl_2), \sigma')}$$

If the language in question allows a finer level of interleaving, its description is forced to define the finer granularity in ever more detailed manipulation of the text.[7] Remembering that a key argument for separating environments from

[7] The terms "small step" and "big step" are used by many authors.

states is to make properties of the language manifest, this text manipulation is unfortunate (there is, in principle, no reason why the parallel execution of sl_1 could not reverse the order of the statements in sl_2; in order to check that no such stupidities occur, one has to read the whole definition). The challenge here would be to have some explicit way of marking monotonic reduction in the text components.

4.2 A practical doubt about denotational semantics

The *Leitmotiv* of my semantics course is a call for abstraction. Trivialising, the case for denotational semantics is the extra lambda abstraction (from $exec\colon Program \times \Sigma \to \Sigma$ to $meaning\colon Program \to (\Sigma \to \Sigma)$). The search for a homomorphic semantic function and the attempts to characterize and (sometimes) achieve “full abstraction” all make it sound as though denotational semantics ought be the climax of such a course. The problem is the (mathematical) cost of achieving –say– abstract procedure denotations. The early struggle (cf. [Sto77]) to provide models of domains which made sense of functions which could take themselves as arguments presents a considerable learning overhead; how much more daunting is the need for Plotkin’s Power Domains (cf. [Plo76])?

Acknowledgements

This paper was written for the 2003 ETAPS event organised in honour of Wlad Turski: my best wishes go to him in the year of his sixty-fifth birthday. The meeting in Warsaw in April 2003 provided a wonderful opportunity to meet and talk with many old friends and I am grateful for the invitation.

Turning to the paper itself, it has benefited from the interaction with Gordon Plotkin and from discussions with Martin Ellis and Nigel Jefferson all of whom provided valuable comments on drafts of this paper. For funding to support my research, I am grateful to the (UK) EPSRC funding for the Dependability IRC (see www.dirc.org.uk) and to the EU-FP5 for funding DSoS.

References

[ACJ72] C. D. Allen, D. N. Chapman, and C. B. Jones. A formal definition of ALGOL 60. Technical Report 12.105, IBM Laboratory Hursley, August 1972.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[BBH+74] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.

[BJ78] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[BW71] H. Bekič and K. Walk. Formalization of storage properties. In *[Eng71]*, pages 28–61. 1971.

[Eng71] E. Engeler. *Symposium on Semantics of Algorithmic Languages*. Number 188 in Lecture Notes in Mathematics. Springer-Verlag, 1971.

[HJ70] W. Henhapl and C. B. Jones. On the interpretation of GOTO statements in the ULD. Technical Report LN 25.3.065, IBM Laboratory, Vienna, March 1970.

[HJ74] K. V. Hanford and C. B. Jones. *Dynamic Syntax: A Concept for the Definition of the Syntax of Programming Languages*, volume 7 of *Annual Review in Automatic Programming*. Pergamon Press, 1974.

[JL71] C. B. Jones and P. Lucas. Proving correctness of implementation techniques. In *[Eng71]*, pages 178–211. 1971.

[Jon01] C. B. Jones. The transition from VDL to VDM. *JUCS*, 7(8):631–640, 2001.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[KNvO$^+$02] Gerwin Klein, Tobias Nipkow, David von Oheimb, Leonor Prensa Nieto, Norbert Schirmer, and Martin Strecker. Java source and bytecode formalisations in Isabelle: Bali, 2002.

[Luc81] P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Research and Development*, 25(5):549–561, September 1981.

[LW69] P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6 of *Annual Review in Automatic Programming Part 3*. Pergamon Press, 1969.

[McC66] J. McCarthy. A formal description of a subset of ALGOL. In *[Ste66]*, pages 1–12, 1966.

[Mos92] P. D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science, 26. Cambridge University Press, 1992.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452–487, September 1976.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Plo03] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Functional and Logic Programming*, 2003. forthcoming.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.

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

[Tur85] W. M. Turski. *Informatics: A Propaedemic View*. North-Holland, 1985.

[vWMPK69] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. *Report on the Algorithmic Language ALGOL 68*. Mathematisch Centrum, Amsterdam, October 1969. Second printing , MR 101.