# The Origins of Structural Operational Semantics

Gordon D. Plotkin

Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh, King's Buildings,
Edinburgh EH9 3JZ, Scotland

I am delighted to see my Aarhus notes [59] on SOS, Structural Operational Semantics, published as part of this special issue. The notes already contain some historical remarks, but the reader may be interested to know more of the personal intellectual context in which they arose. I must straightaway admit that at this distance in time I do not claim total accuracy or completeness: what I write should rather be considered as a reconstruction, based on (possibly faulty) memory, papers, old notes and consultations with colleagues.

As a postgraduate I learnt the untyped $\lambda$-calculus from Rod Burstall. I was further deeply impressed by the work of Peter Landin on the semantics of programming languages [34–37] which includes his abstract SECD machine. One should also single out John McCarthy's contributions [45–48], which include his 1962 introduction of abstract syntax, an essential tool, then and since, for all approaches to the semantics of programming languages. The IBM Vienna school [42, 41] were interested in specifying real programming languages, and, in particular, worked on an abstract interpreting machine for PL/I using VDL, their Vienna Definition Language; they were influenced by the ideas of McCarthy, Landin and Elgot [18].

I remember attending a seminar at Edinburgh where the intricacies of their PL/I abstract machine were explained. The states of these machines are tuples of various kinds of complex trees and there is also a stack of environments; the transition rules involve much tree traversal to access syntactical control points, handle jumps, and to manage concurrency. I recall not much liking this way of doing operational semantics. It seemed far too complex, burying essential semantical ideas in masses of detail; further, the machine states were too big. The lesson I took from this was that abstract interpreting machines do not scale up well when used as a human-oriented method of specification for real languages (but see below for further comment).

Rules play a central rôle in SOS. I recall two contributions in particular. The first is Smullyan's elegant work on formal systems [66]. These are systems of rules of the form:

$$\frac{A_1, \ldots, A_m}{B}$$

where the antecedents and consequents are atoms of the form $P(t_1, \ldots, t_n)$ and where, in turn, $P$ is a predicate symbol of a given arity $n$ and the $t_j$ are terms, taken to be strings of constants and variables. These systems enable the inductive specification of sets of strings; for example it will be clear how to regard Post production systems as formal systems with a single unary predicate symbol

in which the rules have a single antecedent. Smullyan showed how his formal systems enable very clear and natural specification of a variety of examples. The second contribution is Henk Barendregt's thesis [10] where reduction in the $\lambda$-calculus is axiomatised by rules such as:

$$\frac{N \geq N'}{MN \geq MN'}$$

This was a striking improvement on the usual tedious syntactic definition and it influenced my later work on programming aspects of the $\lambda$-calculus.

After completing my doctoral thesis in 1972, I went to the USA, visiting Syracuse and Stanford. One interest of mine there was John Reynolds' paper [61] on call-by-name, call-by-value and continuations; Michael Fischer at MIT had also written on this topic (see [19] for a recent version). This led to my work, reported in [54], where I wanted to deal systematically with the subject of the $\lambda$-calculus as a programming language. It was therefore important to give a thorough treatment of operational semantics and how the equational aspects of the $\lambda$-calculus relate to its programming language ones. Landin's SECD machine provides one kind of operational semantics and Reynolds had used evaluation functions to provide another. These were to be linked to the equational logic of the $\lambda$-calculus via the relevant call-by-value, or call-by-name, normal order reduction; normal order reduction itself was a well-known idea in the $\lambda$-calculus, though not for calling mechanisms.

Several reduction relations are considered in [54]. One is a 'left reduction' relation $\underset{V}{\longrightarrow}$. Iterated, this provides the required call-by-value normal order reduction. The relation $\underset{V}{\longrightarrow}$ is defined inductively in the paper, but that definition is immediately equivalent to one using the following set of rules, leaving out those for the constants:

$$\frac{M \to M'}{MN \to M'N}$$

$$\frac{N \to N'}{MN \to MN'} \quad (M \text{ a value})$$

$$(\lambda x.M)N \to [N/x]M \quad (N \text{ a value})$$

Immediately after the definition of $\underset{V}{\longrightarrow}$ an auxiliary reduction relation is defined by rules, rather than inductively, so perhaps I considered there to be little difference between the two modes of presentation. In more modern terms, the rules for $\underset{V}{\longrightarrow}$ provide a small-step call-by-value transition relation; a corresponding relation $\underset{N}{\longrightarrow}$ for call-by-name is also given, again defined inductively.

Mike Gordon's thesis [20, 21] on pure LISP, under the direction of Rod Burstall, appeared in 1973. It contains a pretty rule-based operational semantics,

with the environment needed to model dynamic binding incorporated in the configuration; this was the first treatment of part of a real programming language. Also relevant was work on recursive function definitions à la McCarthy [46] by Cadiou [15] and Vuillemin [70]; they considered various computation rules including ones for call-by-value and call-by-name. The Edinburgh group further increased in strength with the arrival of Robin Milner from Stanford in 1973.

My next paper in this line was going to be on the denotational semantics aspects of the untyped $\lambda$-calculus considered as a programming language. However, it seemed better to consider the simpler typed case first; here the relevant work was by Dana Scott: his famous, then underground, paper on LCF, the Logic of Computable Functions [65]. This resulted in my paper on PCF [55, 56], the Programming language for Computable Functions, where I gave an operational semantics for the term calculus of the logic, a typed $\lambda$-calculus with booleans, natural numbers and recursion at all types; later I worked on the untyped case, but that material never appeared in print. The operational semantics of PCF is given by a system of rules for a relation which one would now view as a small-step call-by-name transition relation; the connection to a variant call-by-name SECD machine is mentioned but not given. I viewed this operational semantics as making precise Scott's thoughts on symbolic calculation. I further claimed that it is only through having an operational semantics that the term calculus of the logic could be viewed as a programming language.

I recall Dana was sceptical regarding the latter point and, in that connection, he asked a good question: why call it operational semantics? What is operational about it? It would be interesting to know the origins of the term 'operational semantics'; an early use is in a paper of Dana's [63, 64] written in the context of discussions with Christopher Strachey where they came up with the denotational/operational distinction. The Vienna group did discuss operations in their publications, meaning the operations of the abstract interpreting machine, but do not seem to have used the term itself.

The above discussion of influence is not complete, but rather intended as a description of immediate influence. There was certainly other relevant work, for example, by Wadsworth [71, 72] and by Milne and Strachey [52]; other authors whose names come to mind as working on operational notions for the $\lambda$-calculus are Corrado Böhm, Clement McGowan, Jim Morris and Peter Wegner, and this list is surely not complete. Wadsworth started from a semantics and found corresponding $\lambda$-calculus mechanisms; these perform reductions inside $\lambda$-binders and so do not, I think, correspond to any programming language mechanisms. Milne and Strachey converted operational notions into the framework of denotational semantics and so were not concerned with finding direct symbolic presentations of operational semantics.

In the following years greater experience was gained with rule-based approaches to operational semantics. The work on PCF had yielded some increase in simplicity: from the untyped case to the typed one. In 1973 I taught a course to Edinburgh third year students and, since I felt the typed $\lambda$-calculus was still too far from their experience, I tried something simpler, namely McCarthy recursive

function definitions. By then rule-based ideas on operational semantics were current in the Edinburgh group and had come to seem quite natural. When Robin taught the third year course on the semantics of programming languages in 1975 he used them for the imperative case, publishing the rules in [50]; the language used was SIL, the Simple Imperative Language, which features in introductory texts on denotational semantics such as Mike Gordon's [22], itself the product of an Edinburgh undergraduate course. Outside Edinburgh, Matthew Hennessy and Ed Ashcroft and Egidio Astesiano and Gerardo Costa used the ideas when working on nondeterministic extensions of PCF, possibly with call-by-value [25, 26, 7, 8].

The ideas were next extended to handle parallel programming and concurrency. In 1979 Matthew Hennessy and I gave a structural operational semantics for an extension of SIL with a construct for parallel programming [29]; the transition relation is then nondeterministic. In the second half of the 1970s Robin was working on concurrency, developing what was to become CCS, his Calculus of Communicating Systems. In 1979 he found a beautiful and surprising structural operational semantics; the central new thought was to put transition information above the arrow, i.e., to use Keller's [33] labelled transition systems in the rules. This enabled one to give operational semantics for languages for communicating processes. Robin then left for six months in Aarhus, under their visiting lecturer program. That resulted in his CCS book [51] where the operational semantics was first published. As an aside, it is interesting to note his remark there that 'the original definition of ALGOL 68, though strongly verbal, is in essence a set of reduction rules.'

Some other papers by Matthew, Wei Li, Mark Millington and me exploited these ideas further: to study the model theory of CCS [30]; to give operational semantics for Hoare's (original) CSP [58, 60], inspired by the operational semantics in [1]; to give semantics for multitasking and exception handling in Ada and semantics for Edison [39, 40]; and to prove correct translations of some of these languages into each other [28, 27, 40, 49]. Some of these publications are dated after 1981, but they form a logical part of this group of papers.

Nondeterminism is a natural companion to concurrency and parallelism. Hennessy and Ashcroft's and Astesiano and Costa's work is mentioned above; I gave an operational semantics for Dijkstra's guarded command language in [57]; and operational semantics also proved useful in work with Krzysztof Apt on countable nondeterminism [4, 5].

A realisation struck me around then. I, and others, were writing papers on denotational semantics, proving adequacy relative to an operational semantics. But the rule-based operational semantics was both simple and given by elementary mathematical means. So why not consider dropping denotational semantics and, once again, take operational semantics seriously as a specification method for the semantics of programming languages? When Mogens Nielsen invited me to Aarhus to take up a six month visiting lectureship in 1981, I decided to pursue this idea by giving a course of lectures on the subject. These were as follows:

1. Transition Systems and Interpreting Automata

2. Simple Expressions and Commands
3. Definitions, Declarations and Type Checking
4. A First Subset of PASCAL
5. Functions and Procedures
6. Parallelism with Shared Resources
7. Communicating Processes
8. Hardware Description Languages
9. Types, Abstract Types and Polymorphism

with the SOS notes corresponding to the first three and the fifth.

It was by then natural to start with a justification of the rule-based approach, beginning with a discussion of what I took to be the main competitor, abstract machines, or interpreting automata. So in the first lecture a discussion of the SECD machine was given and used as an introduction to structural operational semantics. Two ideas were important for me there and bear repeating here.

The first idea, already clear from the above 'preliminary trials,' was that structural operational semantics was intended as being like an abstract machine but without all the complex machinery in the configurations, just the minimum needed to explain the semantical aspects of the programming language constructs. The extra machinery is avoided by the use of the rules, making the exploration of syntactical structure implicit rather than drearily explicit. Of course, abstract machines are important for the actual implementation of programming languages; indeed it would be good to have a general theory of such machines.

It is worth saying more about the relationship with VDL here. After working on PL/I, the group, now with the addition of Cliff Jones, worked on ALGOL 60 [6]. They too were sensitive to the problem of the 'grand state' of the PL/I machine, and sought a 'small state' alternative. They separated out the environment from the state; Cliff Jones had the idea of treating jumps by a suitable class of 'exit' configurations [24]; and, finally, not having to consider concurrency also simplified matters. However they did not have the idea of using rules, central to SOS. I have to confess that I was not aware of any of this previous development of VDL until Cliff Jones brought it to my attention when I was writing the present account!

The second idea was that the rules should be syntax-directed; this is reflected in the title of the Aarhus notes: the operational semantics is *structural*, not, as some took it, structured. In denotational semantics one follows an ideal of *compositionality*, where the meaning of a compound phrase is given as a function of the meaning of its parts. In the case of operational semantics one considers the behaviour of a program phrase, which is just the collection of the transitions it can make. This behaviour is, however, not compositional when thought of as a function of program phrases. However the rules do give it structurally, i.e., primitive recursively, in the syntax; the idea of structural recursion is due to Burstall [14].

Unfortunately I have very little memory of how I came to this view; it is not explicit in anything I am aware of before the Aarhus notes. It may be that I began

with the idea of the rules as syntax-directed and then, perhaps considering the comparison with denotational semantics, realised that that meant the behaviour function can be given by structural induction. In a recent paper aiming towards an integration of operational and denotational semantics, the structural view is expressed in a categorical form [69].

In fact ideas from denotational semantics pervade SOS. My plan was to follow Tennent's book [68] on programming languages. Tennent used the linguistic ideas developed by Dana Scott, Christopher Strachey and others in their seminal work [67] on denotational semantics, coupled with his own contributions. I wanted to illustrate those ideas by considering various language features for both functional and imperative programming languages. The idea of doing this via a series of 'toy' languages would have been natural given Ledgard's survey [38] and teaching and research experience: they make things clearer for students; they help focus on particular semantical points; and, as Robin demonstrated, considered as calculi they provide useful systems.

The tremendous computer science library at Aarhus was a great help for preparing both the lectures and the notes. It enabled me to find many examples for exercises and, much more importantly, by seeing that the method could handle a very wide range of language constructs, my confidence in its robustness was enhanced.

Not everything went well, particularly the treatment of recursion, where the dynamic semantics seemed to me to be quite clumsy and unnatural, although the idea used there of a recursion construct **rec** $d$ for definitions was itself attractive. Indeed, the rules as stated in the Aarhus notes are not quite right; they have been corrected in the version of the notes published in this special issue. Nowadays I would prefer to treat recursion by using a suitable $\mu$-construct. For example if one had a typed functional programming language, one could have an expression construct $\mu x{:}\sigma.e$ to be thought of as: $x$, recursively defined to be $e$; the variable $x$ will generally occur free in $e$. A suitable small-step transition relation rule is:

$$\frac{e[\mu x:\sigma.e/x] \rightarrow e'}{\mu x:\sigma.e \rightarrow e'}$$

where capture-avoiding substitution, familiar from the $\lambda$-calculus, is intended; an environment-based variation is also possible. Proceeding analogously for definitions, one would allow identifiers to range over definitions, as considered in the Aarhus notes in the section on modules and classes, and use the construct $\mu m:\alpha.d$ with the corresponding transition rule.

On the other hand the rule-based treatment of static semantics, the context-sensitive aspects of syntax, went very smoothly, and seemed to me to provide a useful alternative to attribute grammars, at least for specification purposes. It may be that these rules were based on ideas from the typed $\lambda$-calculus. Behaviour is now the type, or types, associated with a phrase and things do go compositionally; in fact researchers had already given compositional static semantics using the tools of denotational semantics. It would be interesting to formalise the rule-based approach in the compositional case and investigate the relationship with attribute grammars.

The lecture on PASCAL was, as will be evident, aimed at giving the semantics of a real programming language, though it did not get very far. The lecture covered some basic grammatical categories, up to blocks with constant and variable definitions. The lecture on parallelism covered parallelism in imperative languages, including synchronisation constructs: test and set, semaphores, critical sections and monitors. The lecture on communicating processes discussed CCS in a fairly standard way and then explored some alternative semantics. The first was a 'true concurrency' semantics involving multisets of actions; the second, following Lynch and Fischer [43], viewed communication as a disciplined use of shared variables; and in the last, influenced by Kahn and MacQueen [32], a dataflow view of lines as buffers of values was taken. Finally there was a brief exposition of a 'capability language,' in which channels could be passed as values (cf. Milner's pi-calculus [62]).

The lecture on hardware description languages was based on work of Mike Gordon on models of register transfer systems [23]. The language considered permitted both combinatorial and register transfer level circuit description and had facilities for parameterised recursive circuit specifications; there was an elaborate semantics involving micromoves for circuit stabilisation. The last lecture covered type variables, allowing constant type definitions and private ones, yielding something like ML abstract types without the facility for recursive definition; I do not know that anything was said, or thought, about polymorphism.

One topic not covered in the lectures was that of jumps. These could be treated via continuations but that seemed to me too complex for simple jumps and, I believe, I looked at the idea of adding suitable 'exit' configurations to the usual ones, possibly following the corresponding idea for the treatment of jumps in VDM [12, 31], the Vienna Development Method: the Vienna school went on to denotational semantics after their work on abstract machines, inheriting the treatment of jumps from VDL, giving us a curious cycle of influence of three treatments of jumps, from operational to denotational and back again! Flemming Nielson and Hanne Riis, then students at Aarhus, used this direct method to give a thorough treatment of several kinds of jumps in a student project [53]; the two or three main sources of their inspiration were the treatment of jumps in denotational semantics (Stoy's book [67]), perhaps a bit of influence from VDM, but mainly some ideas about complete labels which probably came from Bobrow and Wegbreit's [13].

Returning home from Aarhus I thought about revising the notes for publication, incorporating the material on concurrency. The other material was either too preliminary or seemed not to fit into the general flow. Unfortunately that was a project that never materialised. I sometimes wonder if (and hope that) the fun of obtaining 'underground' copies helped push the ideas more than conventional publication would have done!

Beyond that the story largely belongs to other people, and I just want to pick up one or two points. I had deliberately worked on small step operational semantics. Gilles Kahn and his coworkers showed with TYPOL [16], the specification language for their MENTOR system [17], what could be done with big

step semantics, which they called natural semantics because of an analogy with natural deduction. Their system made good use of the fact that rules can be viewed as Horn clauses and so can be executed in Prolog, thereby yielding an interpretation facility for the language. Interestingly this effort was initially driven by the desire to specify programming language type systems, specifically that of Ada. However it was soon realised that if one can have rules for the relation between a program phrase and its type, one can just as well have rules for the relation between a phrase and its value. An important question is how to execute the rules efficiently; work in this direction linking with attribute grammars can be found in [9].

Big step semantics also appears in the work of Per Martin-Löf. In [44] he gave an informal structural definition of an evaluation relation for closed type-theory terms; this evaluation relation is a big step semantics and so is the graph of the corresponding evaluation function. Per's paper was published in 1979, and therefore anticipates the Aarhus notes; he is very clear that his type theory can be thought of as a programming language. I think I was not aware of this work until after my visit to Aarhus.

Robin Milner and I had never written a paper together although our ideas had most certainly influenced each other's. One day I proposed to Robin that we do something. The SOS notes had deliberately not been written in a theoretical framework as I wanted not to be constrained but rather to work naturally with the various features; there was some idea that a theory could come later: once we had the data! Robin produced some notes on a general approach to operational semantics for a given algebraic signature, but unfortunately no publication resulted. There has been quite a bit of work on formats for operational semantics, see, e.g., Chapter 3 of [11]; what Robin had in mind was closest to De Simone format.

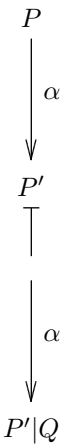
As mentioned above, SOS can be convenient as a basis for proving properties of programming languages; indeed in the Aarhus notes some proofs by structural induction are given and the possibility of proofs on the size of derivations is mentioned. A common current example is the proof of type safety properties, to the general effect that a phrase that can be typed cannot result in a dynamic type error. Another example is proving that the rules of a program logic are sound: this last theme first appears in the work of Apt, who used SOS for proof rules for CSP [1], in his survey article on Hoare logic [2], and, most recently, in his book with Olderog [3].

I should like to conclude this account with an idea which occurred to me on returning to Edinburgh but that also never saw the light of day. There is a question of how best to present the rules. Generally one is content with a 'logical' format, such as the CCS left rule for the parallel operator:

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

I saw the rules as directly formalising the natural English description, for example, this rule says that the first step in executing $P|Q$ can be that of $P$. In

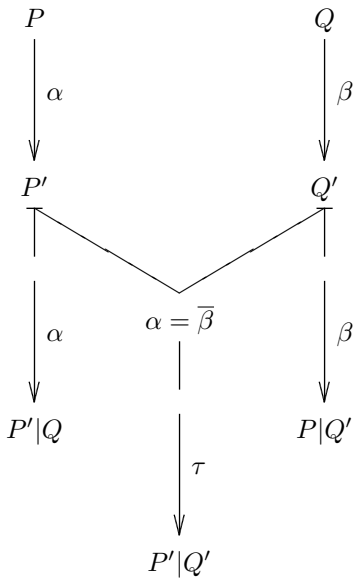
this sense one really wants to read the rules clockwise: I have $P|Q$ and I want to start executing it, so I can start executing $P$. One can organise this information diagrammatically, as follows:




Parallel Composition (Left): $P|Q$

where the arrows indicate transitions as usual and the horizontal lines of rules become long drawn out turnstiles, perhaps interspersed with some conditions, the side-conditions of the rules. The reader may prefer to insert $P|Q$, the 'subject' of the rule, in the gap.

This notation has the advantage of directly showing the flow of control, rather than having to compute it by looking at the rules. Another advantage is that it is quite compact and by combining several of these diagrams one can give a single diagram for all the rules for a given program construct. For example for the CCS parallel construct one could have:




Parallel Composition: $P|Q$

This can perhaps be viewed as a way of depicting flow of control via a kind of schematic Petri Net that also allows logical connections between conditions. One can go further. For example, for an imperative language one could also leave the state component implicit. So for the rule:

$$\frac{\langle S_1, \sigma\rangle \longrightarrow \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \longrightarrow \langle S_1'; S_2, \sigma'\rangle}$$

one could have the diagram:


Sequential Composition: $S_1; S_2$

where the idea is that the state changes along whole transitions but remains constant along inferred ones. One could again combine such diagrams to give a single diagram for each program construct.

Similar diagrammatic conventions allow one to give rules with environments but without explicitly mentioning them: one only indicates the local changes in information. For example suppose the imperative language had a local definition facility **let** $x = E$ **in** $S$, where $E$ ranges over arithmetic expressions. Then one would have the rule:

$$\frac{\rho[x = m] \vdash S \longrightarrow S'}{\rho \vdash \textbf{let } x = m \textbf{ in } S \longrightarrow \textbf{let } x = m \textbf{ in } S'}$$

which could be indicated diagrammatically by:

$$S \xrightarrow[[x = m]]{} S' \vdash \longrightarrow \textbf{let } x = m \textbf{ in } S'$$

Local definition: **let** $x = m$ **in** $S$

where the change in environment is indicated by decoration of the arrow and where, for variation's sake, we use a horizontal display instead of a vertical one.

What I find interesting in the above story of the origins of SOS is, on the one hand, how complex the various influences on ideas are and, on the other hand, even if the ideas themselves are simple, how much work one needs to do to show their power. I would expect that in this respect the story of the development of SOS is quite typical. Another interesting aspect is the mutual influence of teaching and research: things need to be simple so they can be taught to students who do not know strange calculi, and they need to be comprehensive to convince them; pleasingly, these qualities are also what are needed scientifically.

## Acknowledgements

First I would like to give a belated but very heartfelt thanks to Mogens Nielsen and everyone at Aarhus for the great visit there, which gave me the time and facilities for work on operational semantics. Special thanks go to Jette Milwertz, who typed up the notes, and to Karen Møller, Flemming Nielson and others, who kept them available for many years as an Aarhus technical report, including a reprinting in 1991. Having such a visiting lectureship available is a truly wonderful idea; there should be many more such positions in the universe!

My truly heartfelt thanks also go to Dr. Tetsuya Saito who, encouraged by Professor Tatsuya Hagino at Keio University, produced the original Latex version of the SOS notes in 1993; this was no mean task. Thanks also go to Luca Aceto and Wan Fokkink for inviting me to publish these notes in this special volume and for organising its proof reading and editing. I am further most grateful to the following people for the very substantial work involved in carrying out these latter editorial tasks: Luca Aceto, Patricia Bouyer, Ilaria Castellani, Emmanuel

Fleury, Wan Fokkink, Hans Hüttel, Francois Laroussinie, Bas Luttik and Paulien de Wind. Finally, thanks for comments and other help go to Krzysztof Apt, Henk Barendregt, Mike Gordon, Jan Friso Groote, Matthew Hennessy, Cliff Jones, Gilles Kahn, Robin Milner, Peter Mosses, Flemming Nielson, John Power, John Reynolds, Dana Scott, Colin Stirling and Daniele Turi.

## References

1. Krzysztof R. Apt, Formal Justification of a Proof System for Communicating Sequential Processes, *JACM,* Vol. 30, No. 1, pp. 197–216, 1983.
2. Krzysztof R. Apt, Ten Years of Hoare's Logic: a Survey, Part II: Nondeterminism, *TCS,* Vol. 28, pp. 83–109, 1984.
3. Krzysztof R. Apt and Ernst-Rüdiger Olderog, *Verification of Sequential and Concurrent Programs,* Graduate Texts in Computer Science, Berlin: Springer-Verlag, 2nd. edition, 1997.
4. Krzysztof R. Apt and Gordon D. Plotkin, A Cook's Tour of Countable Nondeterminism, *Proc. 8th. ICALP* (eds. S. Even and O. Kariv), LNCS, Vol. 115, pp. 479–494, Berlin: Springer-Verlag, 1981.
5. Krzysztof R. Apt and Gordon D. Plotkin, Countable Nondeterminism and Random Assignment, *JACM,* Vol. 33, No. 4, pp. 724–767, 1986.
6. C. Dave Allen, Dave N. Chapman and Cliff B. Jones, *A Formal Definition of ALGOL 60,* Technical Report 12.105, IBM Laboratory, Hursley, 1972.
7. Egidio Astesiano and Gerardo Costa, Sharing in Nondeterminism, *Proc. 6th. Coll. on Automata, Languages and Programming* (ed. H. A. Maurer), LNCS, Vol. 71, pp. 1–15, Berlin: Springer-Verlag, 1979.
8. Egidio Astesiano and Gerardo Costa, Nondeterminism and Fully Abstract Models, *Informatique Théorique et Applications,* Vol. 14, No. 4, pp. 323–347, 1980.
9. Isabelle Attali, *Sémantique Naturelle: Evaluation et Expressivité,* Mémoire d'Habilitation à Diriger des Recherches, Université de Nice, Sophia Antipolis, 1996.
10. Henk Barendregt, *Some Extensional Term Models for Combinatory Logics and $\lambda$-Calculi,* Ph.D. Thesis, Department of Mathematics, Utrecht, 1971.
11. Jan A. Bergstra, Alban Ponse and Scott A. Smolka (eds.), *Handbook of Process Algebra,* Amsterdam: Elsevier, 2001.
12. Dines Bjørner and Cliff B. Jones (eds.), *The Vienna Development Method: the Meta-Language,* LNCS, Vol. 61, Berlin: Springer-Verlag, 1978.
13. Daniel G. Bobrow and Ben Wegbreit, A Model for Control Structures for Artificial Intelligence Programming Languages, *IEEE Transactions on Computers,* Vol. 25, No. 4, pp. 347–353, 1976.
14. Rod M. Burstall, Proving Properties of Programs by Structural Induction, *The Computer Journal,* Vol. 12, No. 1, pp. 41–48, 1969.
15. Jean-Marie Cadiou, *Recursive Definitions of Partial Functions and their Computations,* Ph.D. thesis, Stanford University, 1972.
16. Thierry Despeyroux, Executable Specification of Static Semantics, *Semantics of Data Types* (eds. G. Kahn, D. B. MacQueen and G. Plotkin), LNCS, Vol. 173, pp. 215–233, Berlin: Springer-Verlag, 1984.
17. Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn and Bernard Lang, Programming Environments Based on Structured Editors: The MENTOR experience, *Interactive Programming Environments* (eds. D. Barstow, E. Sandewall and H. Shrobe), pp. 128–140, New York: McGraw-Hill, 1984.

18. Calvin C. Elgot and Abraham Robinson, Random-Access Stored Program Machines, an Approach to Programming Languages, *JACM,* Vol. 11, No. 4, pp. 365–399, 1964.
19. Michael J. Fischer, Lambda-Calculus Schemata, *Lisp and Symbolic Computation,* Vol. 6, Nos. 3 & 4, pp. 259–288, 1993.
20. Michael J. C. Gordon, *Models of Pure LISP,* Ph.D. Thesis, Experimental Programming Reports: No. 31, School of Artificial Intelligence, University of Edinburgh, 1973.
21. Michael J. C. Gordon, Operational Reasoning and Denotational Semantics, *Construction, Amélioration et Vérification de Programmes,* pp. 83–98, Colloques IRIA, Arc et Senans, 1975.
22. Michael J. C. Gordon, *The Denotational Description of Programming Languages,* Berlin: Springer Verlag, 1979.
23. Michael J. C. Gordon, Register Transfer Systems and their Behaviour, *Computer Hardware Description Languages and Their Applications* (eds. M. Breuer and R. Hartenstein), pp. 23–36, Amsterdam: North Holland, 1981.
24. Wolfgang Henhapl and Cliff B. Jones, *On the Interpretation of GOTO Statements in the ULD,* Technical Report LR 25.3.065, IBM Laboratory, Vienna, 1970.
25. Matthew C. B. Hennessy, The Semantics of Call-by-Value and Call-by-Name in a Nondeterministic Environment, *SIAM J. on Comp.,* Vol. 9, No. 1, pp. 67–84, 1980.
26. Matthew C. B. Hennessy and Edward A. Ashcroft, A Mathematical Semantics for a Nondeterministic Typed Lambda-Calculus, *TCS*, Vol. 11, No. 3, pp. 227–225, 1980.
27. Matthew C. B. Hennessy and Wei Li, Translating Ada Tasking into CCS, *Proc. IFIP TC-2 Work. Conf. on Formal Description of Programming Concepts (II)* (ed. D. Bjørner), pp. 227–247, Amsterdam: North-Holland, 1982.
28. Matthew C. B. Hennessy, Wei Li and Gordon D. Plotkin, A First Attempt at Translating CSP into CCS, *Proc. 2nd. Int. Conf. on Distributed Computing Systems* (ed. E. Gelenbe), pp. 105–115, New York: IEEE Computer Society Press, 1981.
29. Matthew C. B. Hennessy and Gordon D. Plotkin, Full Abstraction for a Simple Parallel Programming Language, *Proc. 8th. MFCS* (eds. G. Goos and J. Hartmanis), LNCS, Vol. 74, pp. 108–120, Berlin: Springer-Verlag, 1979.
30. Matthew C. B. Hennessy and Gordon D. Plotkin, A Term Model for CCS, *Proc. 9th. MFCS* (ed. P. Dembinski), LNCS, Vol. 88, pp. 261–274, Berlin: Springer-Verlag, 1980.
31. Cliff B. Jones, Scientific Decisions which Characterize VDM, *FM'99 - Formal Methods* (eds. J. M. Wing, J. Woodcock and J. Davies), LNCS, Vol. 1708, pp. 28–47, Berlin: Springer-Verlag, 1999.
32. Gilles Kahn and David MacQueen, Coroutines and Networks of Parallel Processes, *Proc. IFIP'77* (ed. B. Gilchrist), pp. 993–998, Amsterdam: North-Holland, 1977.
33. Robert M. Keller, Formal Verification of Parallel Programs, *CACM*, Vol. 19, No. 7, pp. 371–384, 1976.
34. Peter J. Landin, The Mechanical Evaluation of Expressions, *Computer Journal*, Vol. 6, No. 4, pp. 308–320, 1964.
35. Peter J. Landin, A Correspondence between ALGOL-60 and Church's Lambda Notation: Parts I and II, *CACM*, Vol. 8, pp. 89–101 & 158–165, 1965.
36. Peter J. Landin, The Next 700 Programming Languages, *CACM*, Vol. 9, No. 3, pp. 157–166, 1966.

37. Peter J. Landin, A Lambda Calculus Approach, *Advances in Programming and Non-Numerical Computation* (ed. L. Fox), Symposium Publications Division, Chapter 5, pp. 97–141, Oxford: Pergamon Press, 1966.
38. Henry F. Ledgard, Ten Mini-Languages: A Study of Topical Issues in Programming Languages, *ACM Computing Surveys*, Vol. 3, No. 3, pp. 115–146, 1971.
39. Wei Li, An Operational Semantics of Multitasking and Exception Handling in Ada, *Proc. AdaTEC Conf. on Ada*, pp. 138–151, New York: ACM Press, 1982.
40. Wei Li, *An Operational Approach to Semantics and Translation for Concurrent Programming Languages*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh, CST-20-83, 1983.
41. Peter Lucas, Formal Semantics of Programming Languages: VDL, *IBM J. of Res. and Dev.*, Vol. 25, No. 5, pp. 549–561, 1981.
42. Peter Lucas and Kurt Walk, On The Formal Description of PL/I, *Annual Review in Automatic Programming*, Part 3, Vol. 6, pp. 105–182, Oxford: Pergamon Press, 1969.
43. Nancy A. Lynch and Michael J. Fischer, On Describing the Behavior and Implementation of Distributed Systems, *TCS*, Vol. 13, No. 1, pp. 17–43, 1981.
44. Per Martin-Löf, Constructive Mathematics and Computer Programming, *Proc. 6th. International Congress for Logic, Method, and Philosophy of Science* (eds. L. J. Cohen et al), Studies in Logic and the Foundations of Mathematics, Vol. 104, pp. 153–175, Amsterdam: Elsevier North-Holland, 1982.
45. John McCarthy, Towards a Mathematical Theory of Computation, *Proc. IFIP Congress '62* (ed. C. M. Popplewell), pp. 21–28, Amsterdam: North Holland, 1963.
46. John McCarthy, A Basis for a Mathematical Theory of Computation, *Computer Programming and Formal Systems* (eds. P. Braffort and D. Hirschberg), pp. 33–70, North-Holland, Amsterdam, 1963.
47. John McCarthy, A Formal Description of a Subset of ALGOL, *Formal Language Description Languages for Computer Programming*, Proceedings of an IFIP Working Conf. (ed. T. B. Steel, Jr), pp. 1–12, Amsterdam: North Holland, 1966.
48. John McCarthy and James A. Painter, Correctness of a Compiler for Arithmetic Expressions, *Mathematical Aspects of Computer Science* (ed. J. T. Schwartz), Proc. Symp. in Applied Mathematics, Vol. 19, pp. 33–41, Providence, RI: American Mathematical Society, 1967.
49. Mark Millington, *Theories of Translation Correctness for Concurrent Programming Languages*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh, CST-46-87, 1987.
50. Robin Milner, Program Semantics and Mechanized Proofs, *Foundations of Computer Science II* (eds. K. R. Apt and J. W. de Bakker), Mathematical Centre Tracts, No. 82, Amsterdam, 1976.
51. Robin Milner, *A Calculus of Communicating Systems*, LNCS, Vol. 93, Berlin: Springer-Verlag, 1980.
52. Robert Milne and Christopher Strachey, *A Theory of Programming Language Semantics*, London: Chapman & Hall, 1976.
53. Flemming Nielson and Hanne Riis, *A Treatment of Goto and Jump*, unpublished manuscript, Aarhus, 1981.
54. Gordon D. Plotkin, Call-by-Name, Call-by-Value and the Lambda-Calculus, *TCS*, Vol. 1, No. 2, pp. 125–159, 1975.
55. Gordon D. Plotkin, LCF Considered as a Programming Language, *Construction, Amélioration et Vérification de Programmes*, pp. 243–261, Colloques IRIA, Arc et Senans, 1975.

56. Gordon D. Plotkin, LCF Considered as a Programming Language, *TCS*, Vol. 5, No. 3, pp. 225–255, 1977.
57. Gordon D. Plotkin, Dijkstra's Predicate Transformers and Smyth's Power Domains, *Abstract Software Specifications: 1979 Copenhagen Winter School Proceedings* (ed. D. Bjørner), LNCS, Vol. 86, pp. 527–553, Berlin: Springer-Verlag, 1980.
58. Gordon D. Plotkin, An Operational Semantics for CSP, *Logics of Programs and their Applications* (ed. Salwicki, A.), LNCS, Vol. 148, pp. 250–252, Berlin: Springer-Verlag, 1980.
59. Gordon D. Plotkin, *A Structural Approach to Operational Semantics*, DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
60. Gordon D. Plotkin, An Operational Semantics for Hoare's CSP, *Proc. IFIP TC-2 Work. Conf. on Formal Description of Programming Concepts (II)* (ed. D. Bjørner), pp. 199–226, Amsterdam: North-Holland, 1982.
61. John Reynolds, Definitional Interpreters for Higher-Order Programming Languages, *Proc. 25th. ACM National Conf.* , pp. 717–740, New York: ACM, 1972.
62. Davide Sangiorgi and David Walker, *The $\pi$-calculus: a Theory of Mobile Processes*, Cambridge: Cambridge University Press, 2001.
63. Dana S. Scott, Outline of a Mathematical Theory of Computation, *Proc. 4th. Annual Princeton Conf. on Information Sciences and Systems*, pp. 169–176, 1970.
64. Dana S. Scott, *Outline of a Mathematical Theory of Computation*, Programming Research Group, Technical Monograph PRG–2, Oxford University, 1970.
65. Dana S. Scott, A Type-Theoretical Alternative to ISWIM, CUCH, OWHY, *TCS*, Vol. 121, Nos. 1 & 2, pp. 411–440, 1993.
66. Raymond M. Smullyan, *Theory of Formal Systems*, Annals of Mathematics Studies No. 47, Princeton: Princeton University Press, 1961.
67. Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Cambridge: MIT Press, 1977.
68. Robert D. Tennent, *Principles of Programming Languages*, Prentice Hall Series in Computer Science (ed. C. A. R. Hoare), London: Prentice Hall, 1981.
69. Daniele Turi and Gordon D. Plotkin, Towards a Mathematical Operational Semantics, *Proc. 12th. LICS*, pp. 280–291, Los Alamitos: IEEE Computer Society Press, 1997.
70. Jean Vuillemin, Correct and Optimal Implementations of Recursion in a Simple Programming Language, *JCSS*, Vol. 9, No. 3, pp. 322–354, 1974.
71. Christopher Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, Ph.D. Thesis, Oxford University, 1971.
72. Christopher Wadsworth, The Relation between Computational and Denotational Properties for Scott's $D_\infty$-models of the $\lambda$-calculus, *SIAM J. on Comp.* , Vol. 5, No. 3, pp. 488–522, 1976.