1976 Turing Award Lecture

Logic and Programming Languages

DANA S. SCOTT
University of Oxford

[Dana S. Scott was one of two recipients of the 1976 Turing Award presented at the ACM Annual Conference in Houston on October 20. M. O. Rabin's paper, Complexity of Computations, appears on page 319.]

Logic has been long interested in whether answers to certain questions are computable in principle, since the outcome puts bounds on the possibilities of formalization. More recently, precise comparisons in the efficiency of decision methods have become available through the developments in complexity theory. These, however, are applications to logic, and a big question is whether methods of logic have significance in the other direction for the more applied parts of computability theory.

Programming languages offer an obvious opportunity as their syntactic formalization is well advanced; however, the semantical theory can hardly be said to be complete. Though we have many examples, we have still to give wide-ranging mathematical answers to these queries: What is a machine? What is a computable process? How (or how well) does a machine simulate a process? Programs naturally enter in giving descriptions of processes. The definition of the precise meaning of a program then requires us to explain what are the objects of computation (in a way, the statics of the problem) and how they are to be transformed (the dynamics).

So far the theories of automata and of nets, though most interesting for dynamics, have formalized only a portion of the field, and there has been perhaps too much concentration of the finite-state and algebraic aspects. It would seem that the understanding of higher-level program features involves us with infinite objects and forces us to pass through several levels of explanation to go from the conceptual ideas to the final simulation on a real machine. These levels can be made mathematically exact if we can find the right abstractions to represent the necessary structures.

Author's present address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

The experience of many independent workers with the method of data types as lattices (or partial orderings) under an information content ordering, and with their continuous mappings, has demonstrated the flexibility of this approach in providing definitions and proofs, which are clean and without undue dependence on implementations. Nevertheless much remains to be done in showing how abstract conceptualizations can (or cannot) be actualized before we can say we have a unified theory.

As the eleven-and-one-half-th Turing lecturer, it gives me the greatest pleasure to share this prize and this podium with Michael Rabin. Alas, we have not had much chance to collaborate since the time of writing our 1959 paper, and that is for me a great loss. I work best in collaboration, but it is not easy to arrange the right conditions — especially in interdisciplinary subjects and where people are separated by international boundaries. But I have followed his career with deep interest and admiration. As you have heard today, Rabin has been able to *apply* ideas from logic having to do with decidability, computability, and complexity to questions of real mathematical and computational interest. He, and many others, are actively creating new methods of analysis for a wide class of algorithmic problems which has great promise for future development. These aspects of the theory of computation are, however, quite outside my competence, since over the years my interests have diverged from those of Rabin. From the late 1960's my own work has concentrated on seeing whether the ideas of logic can be used to give a better *conceptual* understanding of programming languages. I shall therefore not speak today in detail about my past joint work with Rabin but about my own development and some plans and hopes for the future.

The difficulty of obtaining a precise overall view of a language arose during the period when committees were constructing mammoth "universal" computer languages. We stand now, it seems, on the doorstep of yet another technological revolution during which our ideas of machines and software are going to be completely changed. (I have just noted that the ACM is campaigning again to eliminate the word 'machine' altogether.) The big, big languages may prove to be not very adaptable, but I think the problem of *semantics* will surely remain. I would like to think that the work — again done in collaboration with other people, most notably with the late Christopher Strachey — has made a basic contribution to the foundations of the semantic enterprise. Well, we shall see. I hope too that the research on semantics will not too much longer remain disjoint from investigations like Rabin's.

An Apology and a Nonapology

As a rule, I think, public speakers should not apologize: it only makes the audience uncomfortable. At such a meeting as this, however, one apology is necessary (along with a disclaimer).

Those of you who know my background may well be reminded of Sir Nicholas Gimcrack, hero of the play *The Virtuoso*. It was written

in 1676 by Thomas Shadwell to poke a little fun at the remarkable experiments then being done before the Royal Society of London. At one point in the play, Sir Nicholas is discovered lying on a table trying to learn to swim by imitating the motions of a frog in a bowl of water. When asked whether *he* had ever practiced swimming *in water,* he replies that he hates water and would never go near it! "I content myself," he said, "with the speculative part of swimming; I care not for the practical. I seldom bring anything to use .... Knowledge is the ultimate end."

Now though our ultimate aims are the same, I hasten to disassociate myself from the attitude of disdain for the practical. It is, however, the case that I have no practical experience in present-day programming; by necessity I have had to confine myself to speculative programming, gaining what knowledge I could at second hand by watching various frogs and other creatures. Luckily for me, some of the frogs could speak. With some of them I have had to learn an alien language, and perhaps I have not understood what they were about. But I have *tried* to read and to keep up with developments. I apologize for not being a professional in the programming field, and I certainly, therefore, will not try to sermonize: many of the past Turing lecturers were well equipped for that, and they have given us very good advice. What I try to do is to make some results from logic which seem to me to be relevant to computing *comprehensible* to those who could make use of them. I have also tried to add some results of my own, and I have to leave it to you to judge how successful my activities have been.

Most fortunately today I do *not* have to apologize for the lack of published material; if I had written this talk the day I received the invitation, I might have. But in the August number of *Communications* we have the excellent tutorial paper by Robert Tennent [14] on denotational semantics, and I very warmly recommend it as a starting place. Tennent not only provides serious examples going well beyond what Strachey and I ever published, but he also has a well-organized bibliography.

Only last month the very hefty book by Milne and Strachey [9] was published. Strachey's shockingly sudden and untimely death unfortunately prevented him from ever starting on the revision of the manuscript. We have lost much in style and insight (to say nothing of inspiration) by Strachey's passing, but Robert Milne has carried out their plan admirably. What is important about the book is that it pushes the discussion of a complex language through from the *beginning* to the *end.* Some may find the presentation too rigorous, but the point is that the semantics of the book is not mere speculation but the real thing. It is the product of serious and informed thought; thus, one has the detailed evidence to decide whether the approach is going to be fruitful. Milne has organized the exposition so one can grasp the language on many levels down to the final compiler. He has not tried to sidestep any difficulties. Though not lighthearted and biting, as Strachey often was in conversation, the book is a very fitting

memorial to the last phase of Strachey's work, and it contains any number of original contributions by Milne himself. (I can say these things because I had no hand in writing the book myself.)

Recently published also is the volume by Donahue [4]. This is a not too long and very readable work that discusses issues not covered, or not covered from the same point of view, by the previously mentioned references. Again, it was written quite independently of Strachey and me, and I was very glad to see its appearance.

Soon to come out is the textbook by Joe Stoy [13]. This will complement these other works and should be very useful for teaching, because Stoy has excellent experience in lecturing, both at Oxford University and at M.I.T.

On the foundational side, my own revised paper (Scott [12]) will be out any moment in the *SIAM Journal on Computing*. As it was written from the point of view of enumeration operators in more "classical" recursion theory, its relevance to practical computing may not be at all clear at first glance. Thus I am relieved that these other references explain the uses of the theory in the way I intended.

Fortunately all the above authors cite the literature extensively, and so I can neglect going into further historical detail today. May I only say that many other people have taken up various of the ideas of Strachey and myself, and you can find out about their work not only from these bibliographies but also, for example, from two recent conference proceedings, Manes [7] and Böhm [1]. If I tried to list names here, I would only leave some out—those that have had contact with me know how much I appreciate their interest and contributions.

Some Personal Notes

I was born in California and began my work in mathematical logic as an undergraduate at Berkeley in the early 1950's. The primary influence was, of course, Alfred Tarski together with his many colleagues and students at the University of California. Among many other things, I learned recursive function theory from Raphael and Julia Robinson, whom I want to thank for numerous insights. Also at the time through self-study I found out about the λ-calculus of Curry and Church (which, literally, gave me nightmares at first). Especially important for my later ideas was the study of Tarski's semantics and his definition of truth for formalized languages. These concepts are still being hotly debated today in the philosophy of natural language, as you know. I have tried to carry over the spirit of Tarski's approach to algorithmic languages, which at least have the advantage of being reasonably well formalized syntactically. Whether I have found the *right* denotations of terms as guided by the schemes of Strachey (and worked out by many hands) is what needs discussion. I am the first to say that not *all* problems are solved just by giving denotations to *some* languages. Languages like (the very pure) λ-calculus are well served but many programming concepts are still not covered.

My graduate work was completed in Princeton in 1958 under the direction of Alonzo Church, who also supervised Michael Rabin's thesis. Rabin and I met at that time, but it was during an IBM summer job in 1957 that we did our joint work on automata theory. It was hardly carried out in a vacuum, since many people were working in the area; but we did manage to throw some basic ideas into sharp relief. At the time I was certainly thinking of a project of giving a mathematical definition of a machine. I feel now that the finite-state approach is only partially successful and without much in the way of practical implication. True, many physical machines can be modelled as finite-state devices; but the *finiteness* is hardly the most important feature, and the automata point of view is often rather superficial.

Two later developments made automata seem to me more interesting, at least mathematically: the Chomsky hierarchy and the connections with semigroups. From the algebraic point of view (to my taste at least) Eilenberg, the Euclid of automata theory, in his books [5] has said pretty much the last word. I note too that he has avoided abstract category theory. Categories may lead to good things (cf. Manes [7]), but too early a use can only make things too difficult to understand. That is my personal opinion.

In some ways the Chomsky hierarchy is in the end disappointing. Context-free languages are very important and everyone has to learn about them, but it is not at all clear to me what comes next—if anything. There are so many other families of languages, but not much order has come out of the chaos. I do not think the last word has been said here. It was not knowing where to turn, and being displeased with what I thought was excessive complexity, that made me give up working in automata theory. I tried once in a certain way to connect automata and programming languages by suggesting a more systematic way of separating the machine from the program. Eilenberg heartily disliked the idea, but I was glad to see the recent book by Clark and Cowell [2] where, at the suggestion of Peter Landin, the idea is carried out very nicely. It is not algebra, I admit, but it seems to me to be (elementary, somewhat theoretical) programming. I would like to see the next step, which would fall somewhere in between Manna [8] and Milne–Strachey [9].

It was at Princeton that I had my first introduction to real machines—the now almost prehistoric von Neumann machine. I have to thank Forman Acton for that. Old fashioned as it seems now, it was still *real*; and Hale Trotter and I had great fun with it. How very sad I was indeed to see the totally dead corpse in the Smithsonian Museum with no indication at all what it was like when it was alive.

From Princeton I went to the University of Chicago to teach in the Mathematics Department for two years. Though I met Bob Ashenhurst and Nick Metropolis at that time, my stay was too short to learn from them; and as usual there is always too great a distance between departments. (Of course, since I am only writing about connections with computing, I am not trying to explain my other activities in mathematics and logic.)

From Chicago I went to Berkeley for three years. There I met many computer people through Harry Huskey and René de Vogelaere, the latter of whom introduced me to the details of Algol 60. There was, however, no Computer Science Department as such in Berkeley at that time. For personal reasons I decided soon to move to Stanford. Thus, though I taught a course in Theory of Computation at Berkeley for one semester, my work did not amount to anything. One thing I shall always regret about Berkeley and Computing is that I never learned the details of the work of Dick and Emma Lehmer, because I very much admire the way they get *results* in number theory by machine. Now that we have the Four-Color Problem solved by machine, we are going to see great activity in large-scale, special-purpose theorem proving. I am very sorry not to have any hand in it.

Stanford had from the early 1960's one of the best Computer Science departments in the country, as everyone agrees. You will wonder why I ever left. The answer may be that my appointment was a mixed one between the departments of Philosophy and Mathematics. I suppose my *personal* difficulty is knowing where I should be and what I want to do. But personal failings aside, I had excellent contacts in Forsythe's remarkable department and very good relations with the graduates, and we had many lively courses and seminars. John McCarthy and Pat Suppes, and people from their groups, had much influence on me and my views of computing. In Logic, with my colleagues Sol Feferman and Georg Kreisel, we had a very active group. Among the many Ph.D. students in Logic, the work of Richard Platek had a few years later, when I saw how to use some of his ideas, much influence on me.

At this point I had a year's leave in Amsterdam which proved unexpectedly to be a turning point in my intellectual development. I shall not go into detail, since the story is complicated; but the academic year 1968/69 was one of deep crisis for me, and it is still very painful for me to think back on it. As luck would have it, however, Pat Suppes had proposed my name for the IFIP Working Group 2.2 (now called Formal Description of Programming Concepts). At that time Tom Steel was Chairman, and it was at the Vienna meeting that I first met Christopher Strachey. If the violence of the arguments in this group is any indication, I am really glad I was not involved with anything important like the Algol Committee. But I suppose fighting is therapeutic: it brings out the best and the worst in people. And in any case it is good to learn to defend oneself. Among the various combatants I liked the style and ideas of Strachey best, though I think he often overstated his case; but what he said convinced me I should learn more.

It was only at the end of my year in Amsterdam that I began to talk with Jaco de Bakker, and it was only through correspondence over that summer that our ideas took definite shape. The Vienna IBM Group that I met through WG 2.2 influenced me at this stage

also. In the meantime I had decided to leave Stanford for the Princeton Philosophy Department; but since I was in Europe with my family, I requested an extra term's leave so I could visit Strachey in Oxford in the fall of 1969. That term was one of feverish activity for me; indeed, for several days, I felt as though I had some kind of real brain fever. The collaboration with Strachey in those few weeks was one of the best experiences in my professional life. We were able to repeat it once more the next summer in Princeton, though at a different level of excitement. Sadly, by the time I came to Oxford permanently in 1972, we were both so involved in teaching and administrative duties that real collaboration was nearly impossible. Strachey also became very discouraged over the continuing lack of research funds and help in teaching, and he essentially withdrew himself to write his book with Milne. (It was a great effort and I do not think it did his health any good; how I wish he could have seen it published.)

Returning to 1969, what I started to do was to show Strachey that he was *all wrong* and that he ought to do things in quite another way. He had originally had his attention drawn to the λ-calculus by Roger Penrose and had developed a handy style of using this notation for functional abstraction in explaining programming concepts. It was a *formal* device, however, and I tried to argue that it had *no* mathematical basis. I have told this story before, so to make it short, let me only say that in the first place I had actually convinced him by "superior logic" to give up the type-free λ-calculus. But then, as one consequence of my suggestions followed the other, I began to see that computable functions could be defined on a great variety of spaces. The real step was to see that function-spaces *were* good spaces, and I remember quite clearly that the logician Andrzej Mostowski, who was also visiting Oxford at the time, simply did not believe that the kind of function spaces I defined had a constructive description. But when I saw they actually did, I began to suspect that the possibilities of using function spaces might just be more surprising than we had supposed. Once the doubt about the enforced rigidity of logical types that I had tried to push onto Strachey was there, it was not long before I had found one of the spaces isomorphic with its own function space, which provides a model of the "type-free" λ-calculus. The rest of the story is in the literature.

(An interesting sidelight on the λ-calculus is the rôle of Alan Turing. He studied at Princeton with Church and connected computability with the (formal) λ-calculus around 1936/37. Illuminating details of how his work (and the further influence of λ-calculus) was viewed by Steve Kleene can be found in Crossley [3]. (Of course Turing's later ideas about computers very much influenced Strachey, but this is not the time for a complete historical analysis.) Though I never met Turing (he died in 1954), the second-hand connections through Church and Strachey and my present Oxford colleagues, Les Fox and Robin Gandy, are rather close, though by the time I was a graduate

at Princeton, Church was no longer working on the λ-calculus, and we never discussed his experiences with Turing.)

It is very strange that my λ-calculus models were not discovered earlier by someone else; but I am most encouraged that new kinds of models with new properties are now being discovered, such as the "powerdomains" of Gordon Plotkin [10]. I am personally convinced that the field is well established, both on the theoretical and on the applied side. John Reynolds and Robert Milne have independently introduced a new inductive method of proving equivalences, and the interesting work of Robin Milner on LCF and its proof techniques continues at Edinburgh. This direction of proving things about models was started off by David Park's theorem on relating the fixed-point operator and the so-called paradoxical combinator of λ-calculus, and it opened up a study of the infinitary, yet computable operators which continues now along many lines. Another direction of work goes on in Novosibirsk under Yu. L. Ershov, and quite surprising connections with topological algebra have been pointed out to me by Karl H. Hofmann and his group. There is no space here even to begin to list the many contributors.

In looking forward to the next few years, I am particularly happy to report at this meeting that Tony Hoare has recently accepted the Chair of Computation at Oxford, now made permanent since Strachey's passing. This opens up all sorts of new possibilities for collaboration, both with Hoare and with the many students he will attract after he takes up the post next year. And, as you know, the practical aspects of use and design of computer languages and of programming methodology will certainly be stressed at Oxford (as Strachey did too, I hasten to add), and this is all to the good; but there is also excellent hope for theoretical investigations.

Some Semantic Structures

Turning now to technical details, I should like to give a brief indication of how my construction goes, and how it is open to considerable variation. It will not be possible to argue here that these are the "right" abstractions, and that is why it is a relief to have those references mentioned earlier so easily available.

Perhaps the quickest indication of what I am getting at is provided by two domains: $\mathscr{B}$, the domain of *Boolean values*, and $\mathcal{S} = \mathscr{B}^{\infty}$, the domain of *infinite sequences* of Boolean values. The first main point is that we are going to accept the idea of *partial* functions represented mathematically by giving the functions from time to time *partial values*. As far as $\mathscr{B}$ goes the idea is very trivial: we write

$$\mathscr{B} = \{\mathit{true}, \mathit{false}, \bot\}$$

where $\bot$ is an extra element called "the undefined." In order to keep $\bot$ in its place we impose a partial ordering $\sqsubseteq$ on the domain $\mathscr{B}$, where

$$x \sqsubseteq y \text{ iff either } x = \bot \text{ or } x = y,$$

for all x, y $\mathscr{B}$. It will not mean all that much here in $\mathscr{B}$, but we can read "$\sqsubseteq$" as saying that the information content of x is *contained in* the information content of y. The element $\perp$ has, therefore, empty information content. The scheme is illustrated in Figure 1.

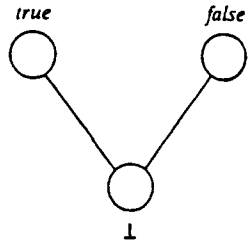

FIGURE 1. *The Boolean values.*

(An aside: in many publications I have advocated using *lattices*, which as partial orderings have a "top" element $\top$ as well as a "bottom" element $\perp$, so that we can assert $\perp \sqsubseteq x \sqsubseteq \top$ for all elements of the domain. This suggestion has not been well received for many reasons I cannot go into here. Some discussion of its reasonableness is to be found in Scott [12], but of course the structure studied there is special. Probably it is best neither to exclude or include a $\perp$; and, for simplicity, I shall not mention it further today.)

Looking now at $\mathscr{S}$, the domain of sequences, we shall employ a shorthand notation where subscripts indicate the coordinates; thus,

$$x = \langle x_n \rangle_{n=0}^{\infty}$$

for all $x \in \mathscr{S}$. Each term is such that $x_n \in \mathscr{B}$, because $\mathscr{S} = \mathscr{B}^{\infty}$. *Technically*, a "direct product" of structures is intended, so we define $\sqsubseteq$ on $\mathscr{S}$ by

$$x \sqsubseteq y \text{ iff } x_n \sqsubseteq y_n, \text{ for all } n.$$

Intuitively, a sequence y is "better" in information than a sequence x iff some of the coordinates of x which were "undefined" have passed over into "being defined" when we go from x to y. For example, each of the following sequences stands in the relation $\sqsubseteq$ to the following ones:

$\langle \perp, \perp, \perp, \perp, \ldots \rangle$,
$\langle true, \perp, \perp, \perp, \ldots \rangle$,
$\langle true, false, \perp, \perp, \ldots \rangle$,
$\langle true, false, true, \perp, \ldots \rangle$.

Clearly this list could be expanded infinitely, and there is also no need to treat the coordinates in the strict order $n = 0, 1, 2 \ldots$. Thus the $\sqsubseteq$ relation on $\mathscr{S}$ is far more complex than the original $\sqsubseteq$ on $\mathscr{B}$.

An obvious difference between $\mathscr{B}$ and $\mathscr{S}$ is that $\mathscr{B}$ is finite while $\mathscr{S}$ has infinitely many elements. In $\mathscr{S}$, also, certain elements have *infinite information content*, whereas this is not so in $\mathscr{B}$. However, we can employ the partial ordering in $\mathscr{S}$ to explain abstractly what we mean by "finite approximation" and "limits." The sequences listed above are *finite* in $\mathscr{S}$ because they have only finitely many coordinates distinct from $\perp$. Given any $x \in \mathscr{S}$ we can cut it down to a finite element by defining

$$(x \restriction m)_n = \begin{cases} x_n, & \text{if } n < m; \\ \perp, & \text{if not.} \end{cases}$$

It is easy to see from our definitions that

$$x \restriction m \sqsubseteq x \restriction (m+1) \sqsubseteq x,$$

so that the $x \restriction m$ are "building up" to a limit; and, in fact, that limit is the original x. We write this as

$$x = \bigsqcup_{m=0}^{\infty} (x \restriction m),$$

where $\bigsqcup$ is the sup or least-upper-bound operation in the partially ordered set $\mathscr{S}$. The point is that $\mathscr{S}$ has many sups; and, whenever we have elements $y^{(m)} \sqsubset y^{(m+1)}$ in $\mathscr{S}$ (regardless of whether they are finite or not), we can define the "limit" z, where

$$z = \bigsqcup_{m=0}^{\infty} y^{(m)}.$$

(Hint: ask yourself what the coordinates of z will have to be.) We cannot rehash the details here, but $\mathscr{S}$ really is a topological space, and z really is a limit. Thus, though $\mathscr{S}$ is infinitary, there is a good chance that we can let manipulations fall back on finitary operations and be able to discuss *computable* operations on $\mathscr{S}$ and on more complex domains.

Aside from the sequence and partial-order structure on $\mathscr{S}$, we can define many kinds of algebraic structure. That is why $\mathscr{S}$ is a good example. For instance, up to isomorphism the space satisfies

$$\mathscr{S} = \mathscr{S} \times \mathscr{S},$$

where on the right-hand side the usual binary direct product is intended. Abstractly, the domain $\mathscr{S} \times \mathscr{S}$ consists of all ordered pairs $\langle x, y \rangle$ with $x, y \in \mathscr{S}$, where we define $\sqsubseteq$ on $\mathscr{S} \times \mathscr{S}$ by

$$\langle x, y \rangle \sqsubseteq \langle x', y' \rangle \text{ iff } x \sqsubseteq x' \text{ and } y \sqsubseteq y'.$$

But for all practical purposes there is no harm in identifying $\langle x, y \rangle$ with a sequence already in $\mathscr{S}$; indeed coordinatewise we can define

$$\langle x, y \rangle_n = \begin{cases} x_k, & \text{if } n = 2k; \\ y_k, & \text{if } n = 2k+1. \end{cases}$$

The above criterion for $\sqsubseteq$ between pairs will be verified, and we can say that $\mathcal{S}$ has a (bi-unique) pairing function.

The pairing function $\langle \cdot, \cdot \rangle$ on $\mathcal{S}$ has many interesting properties. In effect we have already noted that it is *monotonic* (intuitively: as you increase the information contents of x and y, you increase the information content of $\langle x, y \rangle$. More importantly, $\langle \cdot, \cdot \rangle$ is *continuous* in the following precise sense:

$$\langle x, y \rangle = \bigsqcup_{m=0}^{\infty} \langle x \downarrow m, y \downarrow m \rangle,$$

which means that $\langle \cdot, \cdot \rangle$ behaves well under taking finite approximations. And this is only one example; the whole *theory* of monotone and continuous functions is very important to this approach.

Even with the small amount of structure we have put on $\mathcal{S}$, a *language* suggests itself. For the sake of illustration, we concentrate on the two isomorphisms satisfied by $\mathcal{S}$; namely, $\mathcal{S} = \mathcal{B} \times \mathcal{S}$ and $\mathcal{S} = \mathcal{S} \times \mathcal{S}$. The first identified $\mathcal{S}$ as having to do with (infinite) sequences of Boolean values; while the second reminds us of the above discussion of the pairing function. In Figure 2 we set down a quick BNF definition of a language with two kinds of expressions: *Boolean* (the β's) and *sequential* (the σ's).

$$\begin{aligned}
\beta &::= \textbf{true} \mid \textbf{false} \mid \textbf{head}\ \sigma \\
\sigma &::= \beta^* \mid \beta\sigma \mid \textbf{tail}\ \sigma \mid \\
&\qquad \textbf{if}\ \beta\ \textbf{then}\ \sigma'\ \textbf{else}\ \sigma'' \mid \\
&\qquad \textbf{even}\ \sigma \mid \textbf{odd}\ \sigma \mid \textbf{merge}\ \sigma'\ \sigma''
\end{aligned}$$

FIGURE 2. *A brief language.*

This language is very brief indeed: no variables, no declarations, no assignments, only a miniature selection of constant terms. Note that the notation chosen was meant to make the meanings of these expressions obvious. Thus, if σ denotes a sequence x, then **head** σ has got to denote the first term x_0 of the sequence x. As $x_0 \in \mathcal{B}$ and $x \in \mathcal{S}$, we are keeping our types straight.

More precisely, for each expression we can define its (constant) *value* $[\![\cdot]\!]$; so that $[\![\beta]\!] \in \mathcal{B}$ for Boolean expressions β, and $[\![\sigma]\!] \in \mathcal{S}$ for sequential expressions. Since there are ten clauses in the BNF language definition, we would have to set down ten equations to completely specify the semantics of this example; we shall content ourselves with selected equations here. To carry on with the remark in the last paragraph:

$$[\![\textbf{head}\ \sigma]\!] = [\![\sigma]\!]_0.$$

On the other side, the expression β^* creates an infinite sequence of Boolean values:

$$[\![\beta^*]\!] = \langle [\![\beta]\!], [\![\beta]\!], [\![\beta]\!], [\![\beta]\!], \ldots \rangle.$$

(This notation, though rough, is clear.) In the same vein:

$$[\![\beta\sigma]\!] = \langle [\![\beta]\!], [\![\sigma]\!]_0, [\![\sigma]\!]_1, [\![\sigma]\!]_2, \ldots\rangle;$$

while we have

$$[\![\textbf{tail}\ \sigma]\!] = \langle [\![\sigma]\!]_1, [\![\sigma]\!]_2, [\![\sigma]\!]_3, [\![\sigma]\!]_4, \ldots\rangle.$$

Further along:

$$[\![\textbf{even}\ \sigma]\!] = \langle [\![\sigma]\!]_0, [\![\sigma]\!]_2, [\![\sigma]\!]_4, [\![\sigma]\!]_6, \ldots\rangle;$$

and

$$[\![\textbf{merge}\ \sigma'\sigma'']\!] = \langle [\![\sigma']\!], [\![\sigma'']\!]\rangle.$$

These should be enough to give the idea. It should also be clear that what we have is really only a selection, because $\mathcal{S}$ satisfies many more isomorphisms (e.g., $\mathcal{S} = \mathcal{S} \times \mathcal{S} \times \mathcal{S}$), and there are many, many more ways of tearing apart and recombining sequences of Boolean values—all in quite computable ways.

The Function Space

It should not be concluded that the previous section contains the whole of my idea: this would leave us on the elementary level of program schemes (e.g., van Emden–Kowalski [6] or Manna [8] (last chapter)). What some people call "Fixpoint Semantics" (I myself do not like the abbreviated word "fixpoint") is only a *first* chapter. The *second* chapter already includes procedures that take procedures as arguments—higher type procedures—and we are well beyond program schemes. True, fixed-point techniques can be applied to these higher-type procedures, but that is not the only thing to say in their favor. The semantic structure needed to make this definite is the *function space.* I have tried to stress this from the start in 1969, but many people have not understood me well enough.

Suppose $\mathcal{D}'$ and $\mathcal{D}''$ are two domains of the kind we have been discussing (say, $\mathcal{B}$ or $\mathcal{B} \times \mathcal{B}$ or $\mathcal{S}$ or something worse). By $[\mathcal{D}' \to \mathcal{D}'']$ let us understand the domain of *all* monotone and continuous functions f mapping $\mathcal{D}'$ into $\mathcal{D}''$. This is what I mean by a *function space.* It is not all that difficult mathematically, but it is not all that obvious either that $[\mathcal{D}' \to \mathcal{D}'']$ is again a domain "of the same kind," though admittedly of a more complicated structure. I cannot prove it here, but at least I can define the $\sqsubseteq$ relation on the function space:

$$f \sqsubseteq g \text{ iff } f(x) \sqsubseteq g(x) \text{ for all } x \in \mathcal{D}'.$$

Treating functions as abstract *objects* is nothing new; what has to be checked is that they are also quite reasonable *objects of computation.* The relation $\sqsubset$ on $[\mathcal{D}' \to \mathcal{D}'']$ is the first step in checking this, and it leads to a well-behaved notion of a finite approximation to a function. (Sorry! there is no time to be more precise here.) And when that is seen,

the way is open to *iteration* of function spaces; as in $[[\mathscr{D}' \to \mathscr{D}''] \to \mathscr{D}''']$. This is not as crazy as it might seem at first, since our theory identifies $f(x)$ as a computable binary function of *variable* f and *variable* x. Thus, as an operation, it can be seen as an *element* of a function space:

$$[[[\mathscr{D}' \to \mathscr{D}''] \times \mathscr{D}'] \to \mathscr{D}''].$$

This is only the start of a theory of these operators (or *combinators*, as Curry and Church call them).

Swallowing all this, let us attempt an infinite iteration of function spaces beginning with $\mathscr{S}$. We define $\mathscr{F}_0 = \mathscr{S}$ and $\mathscr{F}_{n+1} = [\mathscr{F}_n \to \mathscr{S}]$. Thus $\mathscr{F}_1 = [\mathscr{S} \to \mathscr{S}]$ and

$$\mathscr{F}_4 = [[[[\mathscr{S} \to \mathscr{S}] \to \mathscr{S}] \to \mathscr{S}] \to \mathscr{S}].$$

You just have to believe me that this is all highly constructive (*because* we employ only the continuous functions).

It is fairly clear that there is a natural sense in which this is *cumulative*. In the first place $\mathscr{S}$ is "contained in" $[\mathscr{S} \to \mathscr{S}]$ as a subspace: identify each $x \in \mathscr{S}$ with the corresponding constant function in $[\mathscr{S} \to \mathscr{S}]$. Clearly by our definitions this is an order-preserving correspondence. Also each $f \in [\mathscr{S} \to \mathscr{S}]$ is (crudely) approximated by a constant, namely $f(\perp)$ (this is the "best" element $\sqsubseteq$ all the values $f(x)$). This relationship of subspace and approximation between *spaces* will be denoted by $\mathscr{S} \lhd [\mathscr{S} \to \mathscr{S}]$.

Pushing higher we can say

$$[\mathscr{S} \to \mathscr{S}] \lhd [[\mathscr{S} \to \mathscr{S}] \to \mathscr{S}],$$

but now for a *different* reason. Once we fix the reason why $\mathscr{S} \lhd [\mathscr{S} \to \mathscr{S}]$, we have to respect the function space structure of the higher $\mathscr{F}_n$. In the special case, suppose $f \in [\mathscr{S} \to \mathscr{S}]$. We want to inject f into the next space, so call it $i(f) \in [[\mathscr{S} \to \mathscr{S}] \to \mathscr{S}]$. If g is any element in $[\mathscr{S} \to \mathscr{S}]$ we are being required to define $i(f)(g) \in \mathscr{S}$. Now, since $g \in [\mathscr{S} \to \mathscr{S}]$, we have the original *projection* backwards $j(g) = g(\perp) \in \mathscr{S}$. So, as this is the best approximation to g we can get in $\mathscr{S}$, we are stuck with defining

$$i(f)(g) = f(j(g)).$$

This gives the next map i: $\mathscr{F}_1 \to \mathscr{F}_2$. To define the corresponding projection j: $\mathscr{F}_2 \to \mathscr{F}_1$, we argue in a similar way and define

$$j(\phi)(x) = \phi(i(x)),$$

where we have $\phi \in [\mathscr{S} \to \mathscr{S}] \to \mathscr{S}$, and $i(x) \in [\mathscr{S} \to \mathscr{S}]$ is the constant function with value x. With this progression in mind there is no

trouble in using an exactly similar plan in defining i: $\mathscr{F}_2 \to \mathscr{F}_3$ and j: $\mathscr{F}_3 \to \mathscr{F}_2$. And so on, giving the exact sense to the cumulation:

$$\mathscr{F}_0 \lhd \mathscr{F}_1 \lhd \mathscr{F}_2 \lhd \ldots \lhd \mathscr{F}_n \lhd \mathscr{F}_{n+1} \lhd \ldots .$$

Having all this, it would be a pity not to pass to the limit (this time with *spaces*), and this is just what I want you to accept. What is obtained by decreeing that there is a space

$$\mathscr{F}_\infty = \lim_{n\to\infty} \mathscr{F}_n?$$

Since the separate stages interact thus:

$$\mathscr{F}_{n+1} = [\, \mathscr{F}_n \to \mathscr{S} \,],$$

it is not so queer to guess that

$$\mathscr{F}_\infty = [\, \mathscr{F}_\infty \to \mathscr{S} \,]$$

holds (at least up to isomorphism). It does, but I can only indicate the bare bones of the reason (and reasonableness) of this isomorphism. In the first place the separate spaces $\mathscr{F}_n$ have been placed one inside another, which not only makes a tower of spaces but also respects the combination $f(x)$ as an algebraic operation of *two* variables. $\mathscr{F}_\infty$ in a precise sense is the completion of the union of the $\mathscr{F}_n$; that is *within* these spaces we can think of towers of *functions* each approximating the next (by the use of the i and j mappings), so that in $\mathscr{F}_\infty$ these towers are given limits. If the towers are truncated, then we can argue that each space $\mathscr{F}_n \lhd \mathscr{F}_\infty$.

Now why the isomorphism of $\mathscr{F}_\infty$? Take a function (continuous!) in $[\mathscr{F}_\infty \to \mathscr{S}]$. By its very continuity it will be determined by what it does to the *finite* levels $\mathscr{F}_n$. That is, it will have better and better approximations in $[\mathscr{F}_n \to \mathscr{S}] = \mathscr{F}_{n+1}$; thus, the approximations "live" in the finite levels of $\mathscr{F}_\infty$. Their limit ought to just give us back the function $[\mathscr{F}_\infty \to \mathscr{S}]$ we started with. In the same way *any* element in $\mathscr{F}_\infty$ can be regarded as a limit of approximate functions in the spaces $[\mathscr{F}_n \to \mathscr{S}]$. Admittedly there are details to check; but, in the limit, there is no real difference between $\mathscr{F}_\infty$ and $[\mathscr{F}_\infty \to \mathscr{S}]$: the infinite level of higher type functions is its *own* function space. (As always: this is a consequence of continuity.)

Much structure is lurking under the surface here; in fact more than I thought at first. In Figure 3, I illustrate a chain of isomorphisms that shows that $\mathscr{F}$ gets much of the character of $\mathscr{S}$ with which we are already familiar. The reasons why these are valid are as follows. First, we treat $\mathscr{F}_\infty$ as a function space. Now *pairs of functions* can be isomorphically put into correspondence with functions taking on

pairs of values. But $\mathcal{S} \times \mathcal{S} = \mathcal{S}$ as we already know. The final step just puts functions on $\mathcal{F}_\infty$ back to elements of $\mathcal{F}_\infty$.

$$\begin{aligned} \mathcal{F}_\infty \times \mathcal{F}_\infty &= [\mathcal{F}_\infty \to \mathcal{S}] \times [\mathcal{F}_\infty \to \mathcal{S}] \\ &= [\mathcal{F}_\infty \to \mathcal{S} \times \mathcal{S}] \\ &= [\mathcal{F}_\infty \to \mathcal{S}] \\ &= \mathcal{F}_\infty \end{aligned}$$

FIGURE 3. *The first chain of isomorphisms.*

Using the isomorphsim of Figure 3, we can gain the further result illustrated in Figure 4. The reasons are fairly clear. Take a function from $\mathcal{F}_\infty$ to $\mathcal{F}_\infty$. The values of this function can be construed as functions.

$$\begin{aligned} \mathcal{F}_\infty \to \mathcal{F}_\infty = &[\mathcal{F}_\infty \to [\mathcal{F}_\infty \to \mathcal{S}]] \\ &[[\mathcal{F}_\infty \times \mathcal{F}_\infty] \to \mathcal{S}] \\ &[\mathcal{F}_\infty \to \mathcal{S}] \\ &\mathcal{F}_\infty \end{aligned}$$

FIGURE 4. *The second chain of isomorphisms.*

But consider that a function whose values are functions is just (up to isomorphism of spaces) a *function of two arguments*. As we have seen in Figure 3, $\mathcal{F}_\infty \times \mathcal{F}_\infty = \mathcal{F}_\infty$, so we obtain the final simplification (up to isomorphism).

What we have done is to sketch why $\mathcal{F}_\infty$, the space of functions of infinite type, is a model of the λ-calculus. The λ-calculus is a language (*not* illustrated here), where every term can be regarded as denoting *both* an argument (or value) *and* a function at the same time. The *formal* details are pretty simple, but the *semantical* details are what we have been looking at: every element of the space $\mathcal{F}_\infty$ can be taken at the same time as being an element of the space $[\mathcal{F}_\infty \to \mathcal{F}_\infty]$; thus, $\mathcal{F}_\infty$ provides a model, but it is just one of many.

Without being able to be explicit, a denotational (or mathematical) semantics was outlined for a pure language of procedures (also *pairs* and all the other stuff in Figure 2). In the references cited on real programming languages, all the other features (of assignments, sequencing, declarations, etc., etc.) are added. What has been established in these references is that the method of semantical definition does in fact work. I hope you will look into it.

References

1. Bohm, C., Ed. *λ-Calculus and Computer Science Theory. Lecture Notes in Computer Science, Vol. 37*. Springer-Verlag, New York, 1975.
2. Clark, K. L., and Cowell, D. F. *Programs, Machines, and Computation*. McGraw-Hill, New York, 1976.

3. Crossley, J. N., Ed. *Algebra and Logic* Papers from the 1974 Summer Res. Inst. Australian Math. Soc., Monash U. Clayton, Victoria, Australia. *Lecture Notes in Mathematics, Vol. 450,* Springer-Verlag, 1976.
4. Donahue, J. E. *Complementary Definitions of Programming Language Semantics. Lecture Notes in Computer Science, Vol. 42,* Springer-Verlag, 1976.
5. Eilenberg, S. *Automata, Languages, and Machines.* Academic Press, New York, 1974.
6. van Emden, M. H., and Kowalski, R. A. The semantics of predicate logic as a programming language. *J. ACM 23,* 4 (Oct. 1976), 733–742.
7. Manes, E. G., Ed. *Category Theory Applied to Computation and Control.* First Int. Symp. *Lecture Notes in Computer Science, Vol. 25,* Springer-Verlag, New York, 1976.
8. Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.
9. Milne, R., and Strachey, C. A. *Theory of Programming Language Semantics.* Chapman and Hall, London, and Wiley, New York, 2 Vols., 1976.
10. Plotkin, G. D. A powerdomain construction. *SIAM J. Comptng. 5* (1976), 452–487.
11. Rabin, M. O., and Scott, D. S. Finite automata and their decision problems. *IBM J. Res. and Develop. 3* (1959), 114–125.
12. Scott, D. S. Data types as lattices. *SIAM J. Comptng. 5* (1976), 522–587.
13. Stoy, J. E. *Denotational Semantics—The Scott-Strachey Approach to Programming Language Theory.* M.I.T. Press, Cambridge, Mass.
14. Tennent, R. D. The denotational semantics of programming languages. *Comm. ACM 19, 8* (Aug. 1976), 437–453.

Categories and Subject Descriptors:

F.3.2 **[Logics and Meanings of Programs]**: Semantics of Programming Languages—*algebraic approaches to semantics; denotational semantics;* F.4.1 **[Mathematical Logic and Formal Languages]**: Mathematical Logic—*lambda calculus and related systems; logic programming*

General Terms:

Languages, Theory

Additional Key Words and Phrases:

Automata, context-free languages