MATHEMATICAL SEMANTICS

D.S. Scott

Rapporteurs: Dr. P. Henderson
Dr. P. Lauer
Mr. J. Rushby

1. Introduction

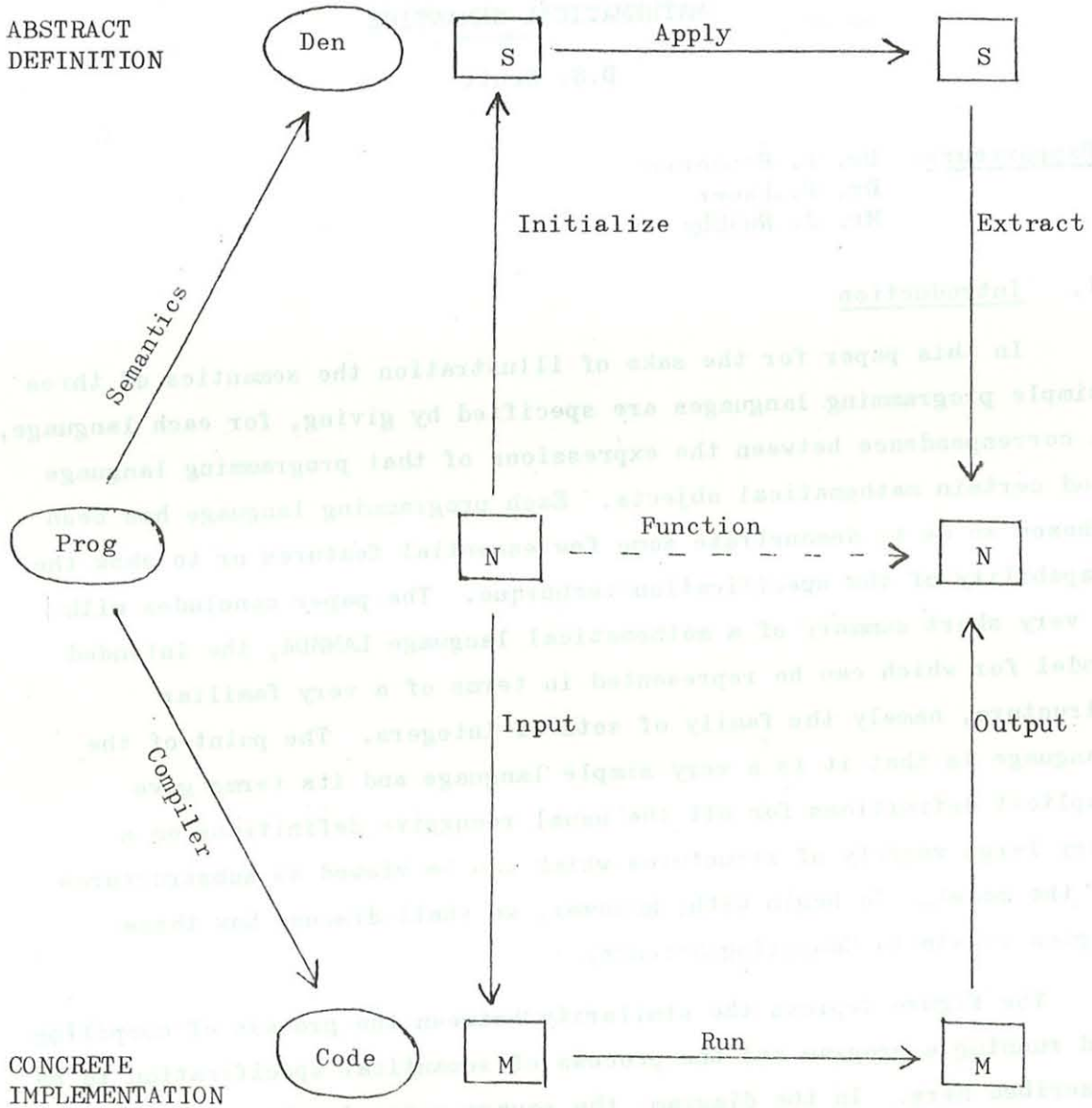
In this paper for the sake of illustration the semantics of three simple programming languages are specified by giving, for each language, a correspondence between the expressions of that programming language and certain mathematical objects. Each programming language has been chosen so as to demonstrate some few essential features or to show the capability of the specification technique. The paper concludes with a very short summary of a mathematical language LAMBDA, the intended model for which can be represented in terms of a very familiar structure, namely the family of sets of integers. The point of the language is that it is a very simple language and its terms give explicit definitions for all the usual recursive definitions on a very large variety of structures which can be viewed as substructures of the model. To begin with, however, we shall discuss how these topics relate to Computing Science.

The figure depicts the similarity between the process of compiling and running a program and the process of semantical specification to be described here. In the diagram, the square nodes denote sets as follows:

N = integers
M = run-time machine states
S = abstract mathematical states for semantical evaluation.

The circular nodes denote objects as follows:

Prog = the program being executed or defined
Code = the machine code obtained from the compiler
Den = the mathematical object (function) obtained from the semantics.


ABSTRACT
DEFINITION
Den
S
Apply
S
Initialize
Extract
Semantics
Prog
N
Function
N
Input
Output
Compiler
Code
M
Run
M
CONCRETE
IMPLEMENTATION

We consider that (in this simple case) the program expects an integer as input and produces an integer as output. The program is intended to realise some function between integers, shown as a dotted line on the diagram.

Consider first the execution of the program by a conventional computing machine. First, the compiler is applied to the program to produce by syntactical manipulation the code. An initial machine state is determined by applying the function Input to an integer from the domain of the program function. The code and the initial machine state are then subjected to the Run function and a final state determined. The value of the program function for the given initial value is extracted from the final machine state by the function Output. The upper half of the figure depicts an entirely similar situation for semantical specification. The semantics of the programming language yields a denotation of the program as a mathematical function (from abstract state to abstract state). An initial argument for this function can be obtained using the Initialize function and the output value determined from the final state by the Extract function. The difference between the two halves of the diagram is that the top part is abstract and conceptual, while the lower half has to do with the representation of data and processes in concrete terms. It may very well be difficult to explain the correspondence between the two sides. The lower half may also change frequently, while the upper half is permanent.

This diagram is clearly a simplification of the real case but such a correspondence has been constructed for more realistic cases, for example, PAL, ALGOL60, PASCAL. One reason for studying the mathematical semantics of programming languages is that we are interested in proving things about programs. Executions of particular programs can only give demonstrations of their behaviour in particular cases. Using the upper half of the diagram we can give criteria for saying whether implementations are correct for all programs. Currently we can only do this and then carry out proofs for modest examples, but work continues to expand the scope of these proofs.

2. Some Notational Preliminaries

We shall insert here a few notational conventions for products, sums and function spaces of domains, which we shall need to explain the various semantical functions.

2.1 Products. The product of domains A, B is denoted by A x B, and that π is a member of A x B by π: A x B.
The components of π are denoted by π_0 : A and π_1 : B.
That is, the subscripts 0,1 select the left and right members respectively. Finally if we have α : A and β : B then we denote by $\langle\alpha,\beta\rangle$: AxB the member of AxB obtained by constructing a pair from these two values. All this we can diagram in rules as follows:

$$\frac{\pi: A\times B}{\pi_0 : A \qquad \pi_1 : B \qquad \pi = \langle\pi_0,\pi_1\rangle} \qquad \frac{\alpha:A \qquad \beta:B}{\langle\alpha,\beta\rangle:A\times B \qquad \langle\alpha,\beta\rangle_0 = \alpha \qquad \langle\alpha,\beta\rangle_1 = \beta}$$

In such a rule each of the statements below the line can be deduced from the statements above.

2.2 Functions. If we denote by $\lambda x{:}A.(\cdots x\cdots)$ the function from A to B obtained by abstracting for x in A on the expression $(\cdots x\cdots)$, which has values in B, and if we use the arrow notation $\varphi{:}A\to B$ to indicate a mapping from A to B, then we have these rules:

$$\frac{\varphi:A\to B}{\begin{cases}\alpha:A\\ \varphi(\alpha):B\end{cases}\qquad \varphi=\lambda x{:}A.\varphi(x)} \qquad \frac{x:A}{(\cdots x\cdots):B}{\lambda x{:}A.\cdots x\cdots : A\to B} \qquad \frac{\alpha:A}{(\lambda x{:}A.(\cdots x\cdots))(\alpha) = (\cdots\alpha\cdots)}$$

We can read the first rule as saying that if $\varphi{:}A\to B$, then α:A implies $\varphi(\alpha)$:B; furthermore the equation $\varphi = \lambda x{:}A.\varphi(x)$ holds. The second rule is the converse in that if x:A implies $(\cdots x\cdots)$:B for all x, then we can write $\lambda x{:}A.(\cdots x\cdots)$: A→B. Finally the last rule provides that if α:A, then $(\lambda x{:}A.\cdots x\cdots)(\alpha) = \cdots\alpha\cdots$. This means that an explicit definition of a function gives the expected function values for suitable arguments.

2.3 Sums The following rules for disjoint sums are not quite complete. They rely on tagging the elements with values 0 and 1 to

determine their ancestry.

$$\frac{\sigma : A+B}{\sigma_0 : T} \qquad \frac{\alpha : A}{<0,\alpha> : A+B} \qquad \frac{\beta : B}{<1,\beta> : A+B}$$

$$\sigma = <\sigma_0, \sigma_1>$$

$$\frac{\sigma_0 = 0}{\sigma_1 : A} \qquad \frac{\sigma_0 = 1}{\sigma_1 : B}$$

We assume $0,1:T$, but T may have other elements; thus additonal rules might be required.

3. The First Language

This programming language manipulates integer stacks. It is intended that programs represent mathematical functions from N to N, where N is the domain of integers. Evaluation starts with the initial value on top of the stack and finishes with the final value of top of the stack. For each expression ε in the programming language we shall give a function $[\![\varepsilon]\!]$ from state to state, where the state consists of just the stack.

Thus we have the following basic domains

Exp	=	expressions (for commands)
N	=	integers
S	=	states (or stacks)

We can define states as follows:

$$S = N \times S$$

We allow $\perp$ to denote the empty stack. Thus in general, if $\sigma:S$ then

$\sigma = <\sigma_0, \sigma_1>$, where $\sigma_0:N$ is the top of the stack, and $\sigma_1:S$ is the tail or remainder.

We can easily define the other necessary functions by:

$$\text{Initialise } (n) = <n, \perp>$$
$$\text{Extract } (\sigma) = \sigma_0$$

The semantical equations for the first language are as follows:

$$
\begin{aligned}
(1)\quad & [\![\underline{\text{pop}}]\!]\,\sigma && = \sigma_1 \\
(2)\quad & [\![\underline{\text{rpt}}]\!]\,\sigma && = \langle \sigma_0, \sigma \rangle \\
(3)\quad & [\![\underline{\text{inv}}]\!]\,\sigma && = \langle \sigma_{10}, \langle \sigma_0, \sigma_{11} \rangle \rangle \\
(4)\quad & [\![\uparrow \epsilon\,]\!]\,\sigma && = \langle \sigma_0, [\![\epsilon]\!]\sigma_1 \rangle \\
(5)\quad & [\![\epsilon ; \epsilon']\!]\sigma && = [\![\epsilon']\!]([\![\epsilon]\!]\sigma) \\
(6)\quad & [\![\underline{\text{dummy}}]\!]\,\sigma && = \sigma \\
(7)\quad & [\![\underline{\text{zero}}]\!]\,\sigma && = \langle 0, \sigma \rangle \\
(8)\quad & [\![\underline{\text{suc}}]\!]\,\sigma && = \langle \sigma_0 + 1, \sigma_1 \rangle \\
(9)\quad & [\![\underline{\text{pred}}]\!]\,\sigma && = \langle \sigma_0 - 1, \sigma_1 \rangle \\
(10)\quad & [\![\epsilon \vee \epsilon']\!]\,\sigma && = (\sigma_0 = 0) \supset \epsilon\ \sigma_1,\ [\![\epsilon']\!]\,\sigma_1 \\
(11)\quad & [\![*\epsilon]\!]\ \sigma && = (\sigma_0 = 0) \supset *\epsilon\ (\ [\![\epsilon]\!]\sigma), \sigma_1
\end{aligned}
$$

Taking these equations in order, we see that (1), (2) and (3) define obvious elementary stack manipulation operations (pop, repeat and invert). Equation (4) says that if an expression is prefixed by ↑ then the expression is actually to be applied to the topless stack, and the original top is to be subsequently replaced. Equation (5) gives the usual definition for sequential composition of strings of commands. Equations (7), (8) and (9) correspond to elementary arithmetic operations which are carried out on the top of the stack. Equation (10) gives a conditional command in the source language, where ϵ or ϵ' is chosen dependent upon the value at the top of the stack is zero or not. In this definition $p \supset e_1, e_2$ is the usual conditional which evaluates to either the value of e_1 or the value of e_2 dependent upon the truth of p. Of course only one of e_1 or e_2 is being evaluated. Finally Equation (11) gives a simple iteration of repeating ϵ zero or more times until the top of the stack is non zero. The tested element is removed whether repetition occurs or not. (Other conventions of what to do with the top of the stack are of course possible.)

Thus we have developed a system of equations which gives a denotation to each expression in the language. That is to say, for each expression in the programming language, the semantical equations determine a mathematical object (a function) which defines the mathematical meaning of that expression. Of course, we easily understand these equations as we read them, and we can imagine a naive implementation for realizing the computations. However, the semantical equations are independent of implementations and representations.

4. The second language

This language will be used to demonstrate how jumps and labels in the source language could be handled by an extension to the semantical equations. We are going to extend the first language by the addition of a block expression:

$$\underline{\text{begin}}\ \varepsilon_0 ; L_1 : \varepsilon_1 ; L_2 : \varepsilon_2 ; \ \ldots\ ; L_n : \varepsilon_n\ \underline{\text{end}}$$

and a goto statement: $\underline{\text{goto}}\ L_i$. To define the semantics it is necessary to extend the basic domains to include labels, environments and continuations:

Lab = labels
Env = environments
C = continuations

with the following definitions

$$\text{Env} = \text{Lab} \to [S \to S]$$
$$C = S \to S$$

We redefine $[\![\varepsilon]\!]$ so that now the logical type of the denotation is as follows:

$$[\![\varepsilon]\!]: \text{Env} \to [C \to [S \to S]]$$

that is if ε:Exp, ρ:Env, and θ:C, then $[\![\varepsilon]\!]\rho\ \theta : S \to S$.
Functional applications are written here in left associative form; that is, $fgh = (f(g))(h)$. It is intended that ρ contains definitions for all the labels occurring in ε, and that θ is a function to be applied after the evaluation of ε is finished (a <u>continuation</u>). Thus, in the "normal" evaluation of a statement (without jumps) we will have

$$[\![\varepsilon]\!]\rho\theta\sigma = \theta\sigma' = \sigma'',$$

where $\sigma' = [\![\varepsilon]\!]\rho I_s \sigma$ and I_s is the "harmless" continuation or identity transformation $I_s : S \to S$ with $I_s(\sigma) = \sigma$. The exact semantical equations for the second language are as follows.

$$
\begin{aligned}
&(1)\quad [\![\underline{\text{pop}}]\!]\,\rho\theta\sigma &&= \theta(\sigma_1)\\
&(2)\quad [\![\underline{\text{rpt}}]\!]\,\rho\theta\sigma &&= \theta(\langle\sigma_0,\sigma\rangle)\\
&(3)\quad [\![\underline{\text{inv}}]\!]\,\rho\theta\sigma &&= \theta(\langle\sigma_{10},\langle\sigma_0,\sigma_{11}\rangle\rangle)\\
&(4)\quad [\![\uparrow\epsilon]\!]\,\rho\theta\sigma &&= [\![\epsilon]\!]\rho(\lambda\sigma'.\theta(\langle\sigma_0,\sigma'\rangle))\sigma_1\\
&(5)\quad [\![\epsilon;\epsilon']\!]\,\rho\theta\sigma &&= [\![\epsilon]\!]\rho([\![\epsilon']\!]\rho\theta)\sigma\\
&(6)\quad [\![\underline{\text{dummy}}]\!]\,\rho\theta\sigma &&= \theta(\sigma)\\
&(7)\quad [\![\underline{\text{zero}}]\!]\,\rho\theta\sigma &&= \theta(\langle 0,\sigma\rangle)\\
&(8)\quad [\![\underline{\text{suc}}]\!]\,\rho\theta\sigma &&= \theta(\langle\sigma_0+1,\sigma_1\rangle)\\
&(9)\quad [\![\underline{\text{pred}}]\!]\,\rho\theta\sigma &&= \theta(\langle\sigma_0-1,\sigma_1\rangle)\\
&(10)\quad [\![\epsilon\vee\epsilon']\!]\,\rho\theta\sigma &&= (\sigma_0=0)\supset[\![\epsilon]\!]\rho\theta\sigma_1,[\![\epsilon']\!]\rho\theta\sigma_1\\
&(11)\quad [\![*\epsilon]\!]\,\rho\theta\sigma &&= (\sigma_0=0)\supset[\![\epsilon]\!]\rho(\lceil *\epsilon]\!]\rho\theta)\sigma,\sigma_1\\
&(12)\quad [\![\underline{\text{halt}}]\!]\,\rho\theta\sigma &&= \sigma\\
&(13)\quad [\![\underline{\text{goto}}\ L]\!]\,\rho\theta\sigma &&= \rho(L)\sigma\\
&(14)\quad [\![\underline{\text{begin}}\ \epsilon_0;L_1:\epsilon_1;L_2:\epsilon_2;\ldots L_n:\epsilon_n\ \underline{\text{end}}]\!]\rho\theta\sigma &&= \theta_0(\sigma)
\end{aligned}
$$

where $\theta_0 = [\![\epsilon_0]\!]\rho'\theta_1$

$\theta_1 = [\![\epsilon_1]\!]\rho'\theta_2$

$\theta_2 = [\![\epsilon_2]\!]\rho'\theta_3$

$\vdots$

$\theta_n = [\![\epsilon_n]\!]\rho'\theta$

and $\rho' = \rho[\![\theta_1,\theta_2,\ldots,\theta_n/L_1,L_2,\ldots,L_n]\!]$, where

in Equation (14) ρ' is the same as ρ except the values of $L_1,\ldots,L_n$ are changed to $\theta_1,\ldots,\theta_n$, respectively.

Equations (1), (2), (3), and (6), (7), (8), (9) differ from before only in that consideration has been taken of the need to apply the continuation. Equation (4) is more subtle: we first have to prepare the new continuation $\lambda\sigma'.\ \theta(\langle\sigma_0,\sigma'\rangle)$ in order to save the top (σ_0). With this in hand we then evaluate ϵ with the tail of the original stack (σ_1). The result is the same as before if we restrict ourselves to the first language as a subset of the second. Similarly, Equation (5) also gives the same meaning as before, but note the change in order. This time using the continuation $[\![\epsilon']\!]\rho\theta$ implies that ϵ' is evaluated <u>before</u> the original continuation θ is applied but it comes <u>after</u> the evaluation of ϵ. Equation (10) is upgraded to take account of the new forms for $[\![\epsilon]\!]$, $[\![\epsilon']\!]$. Equation

(11) makes use of the continuation in exactly the same way as (5). The last three equations are new and account for the extended power of the expanded language.

Equation (12) forces us to ignore the continuation while making no change of state. (13) selects the state transformation stored in the environment as the value of the label; it ignores the continuation because this is a broken jump and $\rho(L)$ is the new continuation.

In Equation (14) the θ_i are defined together recursively since the new environment ρ' involves all the others. If there are no jumps, this is just a composition. If there are jumps, the ρ' will select the proper meaning. Of course we could even jump outside the block as given by a continuation already specified in ρ. We see in this way how the scope of label definitions is kept straight. If we get to the end of the block, we fall through using θ, the original continuation specified upon entry.

5. The third language

In this third programming language we try to include something of the concept of a stored program - together with procedure calls which involve the usual idea of a variable as the parameter of a procedure. This language can be taken as an extension of the second, except for brevity the ability to compute with integers is dropped. The necessary equations can easily be restored as they involve no special problems. Instead of integers the elementary items stored on the stack are trees of functions, here called denotations.

For the basic domains we have:

Exp	=	expressions
Idf	=	identifiers
Env	=	environments
C	=	continuations
S	=	states
D	=	denotations

The last four are defined by:

$$\begin{aligned} Env &= Idf \to D \\ C &= S \to S \\ S &= D \times S \\ D &= [S \to S] + [D \times D] \end{aligned}$$

A denotation is therefore either a function or a pair of denotations; that is it is a binary tree with functions (continuations) at the ends of branches. (It is a good question why such a domain as D exists.)

In the following, typical elements of the domains will be denoted thus:

ε : Exp	θ : C
ξ : Idf	σ : S
ρ : Env	δ : D

The disjoint union (as described in section 2.3) enables us to separate elements of D into two kinds according to the following scheme:

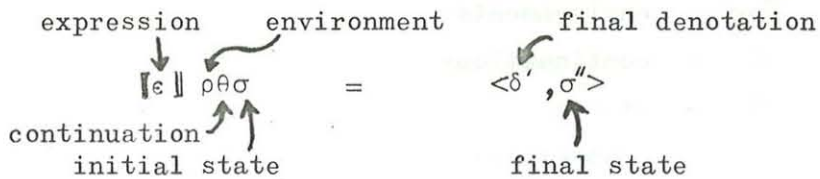
$$\begin{aligned} &\langle 0, \theta \rangle : D && \text{if } \theta : [S \to S] \\ &\langle 1, \langle \delta_1, \delta_2 \rangle \rangle : D && \text{if } \langle \delta_1, \delta_2 \rangle : [D \times D]. \end{aligned}$$

The two elements of D, $\langle 0, \perp \rangle$ and $\langle 1, \perp \rangle$, can be used for <u>true</u> and <u>false</u>, respectively, and we shall construct conditionals in terms of them. In case we wanted integers we would have to add +N to the domain equation for D, but it would complicate the subscripts.

As before, we have the rules of semantical evaluation expressed using

$$[\![-]\!] : Exp \to Env \to C \to S \to S,$$

where the types of the various components of each equation is given by:



The semantical equations are as follows:

$$
\begin{aligned}
(1)\quad & [\![\underline{\text{halt}}]\!]\,\rho\theta\sigma &&= \sigma \\
(2)\quad & [\![\underline{\text{cont}}]\!]\,\rho\theta\sigma &&= \theta(\sigma) \\
(3)\quad & [\![\underline{\text{pop}}\ ]\!]\,\rho\theta\sigma &&= \theta(\sigma_1) \\
(4)\quad & [\![\xi]\!]\,\rho\theta\sigma &&= \theta(<\rho(\xi),\sigma>) \\
(5)\quad & [\![\underline{\text{dbl}}\ ]\!]\,\rho\theta\sigma &&= \theta(<\sigma_0,\sigma>) \\
(6)\quad & [\![\underline{\text{inv}}\ ]\!]\,\rho\theta\sigma &&= \theta(\sigma_{10},<\sigma_0,\sigma_{11}>>) \\
(7)\quad & [\![\underline{\text{true}}]\!]\,\rho\theta\sigma &&= \theta(<<0,\ \perp>,\sigma>) \\
(8)\quad & [\![\underline{\text{false}}]\!]\,\rho\theta\sigma &&= \theta(<<1,\ \perp>,\sigma>) \\
(9)\quad & [\![\underline{\text{pair}}]\!]\,\rho\theta\sigma &&= \theta(<<1,<\sigma_0,\sigma_{10}>>,\sigma_{11}>) \\
(10)\quad & [\![\underline{\text{split}}]\!]\,\rho\theta\sigma &&= \theta(\sigma_{00}\supset \perp,<\sigma_{010},<\sigma_{011},\sigma_1>>) \\
(11)\quad & [\![\underline{\text{apply}}]\!]\,\rho\theta\sigma &&= \theta(\sigma_{00}\supset \sigma_{01}(\sigma_1),\ \perp) \\
(12)\quad & [\![\underline{\text{save}}]\!]\,\rho\theta\sigma &&= \theta(<<0,\theta>,\sigma>) \\
(13)\quad & [\![\lambda\xi.\varepsilon]\!]\,\rho\theta\sigma &&= \theta(<<0,\theta'>,\sigma>) \text{ where} \\
& \qquad\qquad \theta' &&= \lambda\delta,\sigma'.[\![\varepsilon]\!](\rho[\delta/\xi])(\lambda\sigma''.\sigma'')\sigma' \\
(14)\quad & [\![\varepsilon;\varepsilon']\!]\,\rho\theta\sigma &&= [\![\varepsilon]\!]\rho([\![\varepsilon']\!]\rho\theta)\sigma \\
(15)\quad & [\![\varepsilon\vee\varepsilon']\!]\,\rho\theta\sigma &&= \sigma_{00}\supset[\![\varepsilon]\!]\rho\theta\sigma_1,[\![\varepsilon']\!]\rho\theta\sigma_1 \\
(16)\quad & [\![\uparrow\varepsilon]\!]\,\rho\theta\sigma &&= [\![\varepsilon]\!]\rho(\lambda\sigma'.\theta(<\sigma_0,\sigma'>))\sigma_1 \\
(17)\quad & [\![\underline{\text{begin}}\ \varepsilon_0;\xi_1:\varepsilon_1;\xi_2:\varepsilon_2;\ldots;\xi_n:\varepsilon_n\ \underline{\text{end}}]\!]\rho\theta\sigma &&= \theta_0(\sigma),
\end{aligned}
$$

where

$$
\begin{aligned}
\theta_0 &= [\![\varepsilon_0]\!]\,\rho'\theta_1 \\
\theta_1 &= [\![\varepsilon_1]\!]\,\rho'\theta_2 \\
\theta_2 &= [\![\varepsilon_2]\!]\,\rho'\theta_3 \\
&\ \ \vdots \\
\theta_n &= [\varepsilon_n]\,\rho'\theta
\end{aligned}
$$

and

$$\rho' = \rho[<0,\theta_1>,<0,\theta_2>,\ldots<0,\theta_n>/\xi_1,\xi_2,\ldots,\xi_n]$$

These rules are sufficiently different that it is worth going through them one at a time:

(1) halt ignores the continuation and yields the current state as the result.

(2) continue is the same as dummy in the earlier languages.

(3) pop removes the top value of the stack.

(4) Execution of an identifier results in the corresponding denotation $\rho(\xi)$ being placed on top of the stack.

(5) dbl duplicates the top stack element

(6) inv exchanges the top two stack elements

(7)(8) true and false push their denotations onto the top of the stack.

(9) pair combines the top two elements of the stack (denotations), into a tree of denotations.

(10) split undoes pair. (Here the conditional is used in a slightly different way than before: $t \supset \sigma_0, \sigma_1$ means that σ_0 is selected if $t = 0$, and σ_1 if $t = 1$. Note that if the top element is elementary and can not be split, then we continue with an empty stack. It would be easy to make other conventions if desired.)

(11) apply expects the top stack element to be a denotation, which is then applied to the remainder of the stack.

(12) save records the continuation on top of the stack as a repeatable action and continues with the same continuation (self application!).

(13) With λ we have a defined function which requires an argument before it can be applied. The immediate effect is simply to push the denotation θ' of this function definition onto the stack. This denotation is such, however, that if applied (say by (11)), it will consider the top stack element (δ) as the argument corresponding to ξ (so $\rho[\delta/\xi]$). (The continuation part of the defined ρ' is taken simply as the identity function, since no other continuation seems to be especially called for.)

(14) sequential composition is the same as before.

(15) the conditional form transfers according to the test of the top of the stack.

(16) With $\uparrow$ we temporarily ignore the top of the stack as before.

(17) blocks are treated as before, except now labels and identifiers are the same. (Note that no equation is given for goto since this can be programmed as

goto $\xi \equiv \xi$; apply; halt.)

6. The language LAMBDA

Only the briefest summary of this model for pure lambda calculus is given here since it has been the subject of a series of lectures at the 1974 Kiel Logic Summer School and a long paper will appear in those proceedings, which also will contain a long bibiography of related papers.

The plan is to represent a very large variety of structures in terms of a very familiar structure, namely the family of sets of integers. A set of integers will be construed here as an infinite object being obtained by enumeration. We can think of a set of integers as a multi-valued integer the elements of which a processor is outputting one by one, in some (unimportant) order. The singled-valued integers correspond to the sets with just one element. For simplicity of notation, we identify sets of a single integer with the integer itself (e.g. 3 for $\{3\}$). We use ω (omega) to denote the set of (non negative) integers: $\omega = \{0,1,2,\ldots\}$, and $P\omega$ to represent the powerset of ω: $P\omega = \{x \mid x \subseteq \omega\}$. The domain $P\omega$ is the "state-space" for the language LAMBDA. The functions we use on this space are novel in that they are not only multi-valued, but multi-argmented as well. They are certain of the mappings from $P\omega$ into $P\omega$. If values are multi-valued, arguments should be too, for don't you want to compose these functions? The question is: why are multi-valued functions interesting or useful?

If $f(x)$ is the empty set, this means that the process initiated by applying f to x gives no output. We use the symbol $\perp$ for this low information value: $f(x) = \perp$, and we say that the function value is undefined. If $f(x) = \omega$, this corresponds to the process with a "short" in it - it just puts out all the integers, one after another. We use $\top$ for this high-information constant: $f(x) = \top$, and we say that $f(x)$ is overdefined. These are the extreme cases, and we are really interested in intermediate cases, say

$$f(x) = 3 \cup 10 \cup 29.$$

We think, intuitively, of a function as a processor performing a transformation which, as it sees more and more of the input, it gives

more and more of the output. Such transformations could be called enumeration operators, since they compute by enumeration. Now, if there is a way of (theoretically) computing these functions, then the functions should be continuous - this is the essential point. Let e_n denote the n^{th} finite subset of ω. (Note that if $x \in P\omega$ then $x = \bigcup \{e_n \mid e_n \subseteq x\}$.) If f is continuous, then (by definition) we have for all x in $P\omega$:

$f(x) = \bigcup \{f(e_n) \mid e_n \subseteq x\}$. This means that for every finite subset contained within the value of the function, there is a finite subset of the argument which determines that subset. As a consequence these functions are monotone - the more you give of the input the more you get of the output.

If you consider only the continous functions, then you can identify them with the sets of integers. We do this by defining the graph of function:

$$\text{Graph } (f) = \{(n,m) \mid m \in f\ (e_n)\}.$$

This set of pairs of integers (where a pair can be identified with an integer by pairing functions) completely defines the functions by virtue of the definition of continuity. Once you have reduced a function to a set of integers in this way, this set is available as the argument of another function, since a continous function can accept any set of integers as its argument. In this way we obtain a coherent mathematical interpretation of self-application.

Many of these ideas appear in H. Roger's book on recursive function theory. He and Friedberg wrote a paper on enumeration operators, and Myhill and Shepherdson had thought about them in an early paper. Rogers only considers the ones with recursively enumerable graphs, and he doesn't really introduce a theory of enumeration operators. Myhill and Shepherdson went a bit further - they said "look how many interesting enumeration operators there are!" and gave a page listing various operators. No one seems to have read this page since. The author's first λ-calculus model was much more complicated to explain, and the idea of using graphs is due to G. Plotkin, but he did not think of making a reduction to $P\omega$.

The point about a λ-calculus model is that there's a theory of enumeration operators and a simple notation for them embodied in the language LAMBDA. In this language, λ-abstraction is simply the idea explained above of taking the graph of the function:

$$\lambda\, x.\tau = \{(n,m) \mid m \in \tau[e_n/x]\}.$$

Every expression in LAMBDA denotes a set. So the graph of a function is the denotation of the λ expression. The inverse of abstraction is application. Suppose u is the graph of a function. Suppose we want to decode the information that is condensed into u, to u-evaluate the function at the argument x. We have:

$$u(x) = \{m \mid \exists e_n \subseteq x, \quad (n,m) \in u\}.$$

You start the enumeration of the finite subset e_n of x and look at the numbers n (the Gödel numbers of the finite sets) to see which ordered pairs (n,m) appear in u. The numbers m enumerated in this way form the output. If u is the graph of a continuous function, you will just have obtained the desired value of that function. That's the whole idea of the model. The other primitives of the language (besides functional abstraction and application) are the arithmetic notions:

$$\begin{aligned} 0 &= \{0\} \\ x+1 &= \{n+1 \mid n \in x\} \\ x-1 &= \{n \mid n+1 \in x\} \\ z \supset x,y &= \{n \in x \mid 0 \in z\} \cup \{m \in y \mid \exists k.\, k+1 \in z\} \end{aligned}$$

The idea of the first three is obvious, while the last is a multi-valued version of McCarthy's conditional expression. The test is on whether z is zero or positive; in case it is both, the answer is x ∪ y. All LAMBDA - definable functions are continuous and computable (in the precise sense of having recursively enumerable graphs).

The following laws of λ-calculus are valid in the model:

$$\begin{aligned} &(\alpha) \quad \lambda x.\tau = \lambda y.\ \tau[y/x] \\ &(\beta) \quad (\lambda x.\tau)\ (\sigma) = \tau[\sigma/x] \\ &(\mathcal{F}) \quad \lambda x.\tau = \lambda x.\ \sigma \text{ iff } \forall x.\tau = \sigma \end{aligned}$$

In the above, $[\sigma/x]$ is the notation for the substitution of a term for a variable. The first law (α) just means we can rewrite bound variables. The second law (β) means that application is the inverse of abstraction (continuity is needed for the proof of this law!) And the last law (ξ) means that two functions have the same graph if and only if they take on the same values.

The following is <u>not</u> true in LAMBDA:

$$(\eta) \qquad \lambda x.u(x) = u$$

because not every set of integers is the graph of some function. The failure of this law is not very important.

Here are some examples of the denotations of some specific expressions:

$$\begin{aligned}
\lambda x.\bot &= \{(n,m) \mid m \in \bot\} = \bot \\
\lambda x.\top &= \{(n,m) \mid m \in \top\} = \top \\
\lambda x.x &= \{(n,m) \mid m \in e_n\} \\
K = \lambda x.\ \lambda y.\ x &= \{(n,m) \mid m \in \lambda y.\ e_n\} \\
&= \{(n,m) \mid k,l.\ m \in (k,l), l \in e_n\} \\
&= \{(n,(k,l)) \mid l \in e_n\}.
\end{aligned}$$

Given any argument x, the combinator K forms the function of y, say, that is constantly equal to x. There are many other combinators all of which can be interpreted in the model in this way. If you like, you can think of LAMBDA as a language of procedures - procedures that accept procedures as arguments and give procedures as results. λ- abstraction and functional application correspond to procedure declaration and call.

Now I wish to briefly review the situation with respect to <u>fixed points</u>. Continuous functions always have fixed points.

We define: $$Y(F) = \bigcup_{n=0}^{\infty} F^n(\bot)$$

This iteration operator gives you the <u>least</u> fixed points of a function F. We can think of Y as a function of F; in fact, Y is itself a <u>continuous</u> function.

Transferring a result of David Park to this model, the so-called Paradoxical combinator:

$$Y = \lambda u.(\lambda x.u(x(x)))(\lambda x.u(x(x)))$$

is the least fixed point operator. Hence, it gives us the possibility of solving any system of equations in LAMBDA even for mutual fixed points. Further it turns out that by use of fixed-point equations a set α is recursively enumerable (r.e.) iff $\alpha = \tau$ for some constant LAMBDA expression τ. This gives us a direct connection between LAMBDA and its model and ordinary recursion theory (the theory of single-valued partial recursive functions on integers).

Next we want to get structured values out of this model (we have already got the r.e. sets), so let us consider for example lists. One way to represent lists in this language is to think of them as functions of integer subscripts. We define:

$$<> = \lambda x.\perp = \perp.$$

$$<x_0, x_1, \ldots x_n> = \lambda z.z \supset x_0, <x_1, \ldots x_n>(z-1)$$

A list of n elements is that function which takes its argument, and if that argument is zero, delivers the element x_0, and which otherwise decreases its argument by one and tests to see if you should choose one of the remaining components. So as long as $i < n$, we have $<x_0, x_1, \ldots, x_{n-1}>(i) = x_i$. Note that

$$\perp = <> \subseteq <x_0> \subseteq <x_0, x,> \subseteq \ldots \subseteq <x_0, x_1, \ldots x_n> \subseteq \ldots$$

Ordered pairs of sets are represented as special cases of a list. We will use the usual subscripts for the selectors of ordered pairs, so that $<u_0, u_1> = <u(0), u(1)>$. Now the combinator

$$\pi = \lambda u.\ <u_0, u_1>$$

"expresses" the property of being a pair in the sense that the pairs are exactly the fixed points of π. The object π represents the space of all pairs; both the range and fixed-points of π consist of exactly the pairs. In a way π is a coercion operator, since $\pi(a)$ always coerces a into the closest ordered pair. For example

$$\pi\ (<x,y,z>) = <x,y>.$$

In a more technical manner of speaking π is a retract. Another useful fact is that the lattice corresponding to the range of π is the natural lattice structure on $P\omega \times P\omega$. This follows from our definition of ordered pair, because we have:

$$\langle x_0, x_1\rangle \subseteq \langle y_0, y_1\rangle \text{ iff } x_0 \subseteq y_0 \text{ and } x_1 \subseteq y_1$$

Another example of a fixed-point equation is $f = \lambda x.f(x)$. Now not every set is the graph of a continuous function (because the graphs of continuous functions have a certain amount of regularity to them), but those that satisfy this equation are graphs. Therefore the combinator

$$\varphi = \lambda f.\ \lambda x.f(x)$$

represents the continuous function space from $P\omega$ to $P\omega$. The fixed points of φ form a lattice which is exactly the lattice structures we would want for this function space. This is another example of a retract. In general a retract is a function a such that:

$$a = a \cdot a = \lambda x.\ a(a(x))$$

Each retract represents a substructure of $P\omega$, namely its range, which is the same as its fixed-point set. If a and b are two retracts, we can define:

$$a \times b = \lambda u.\ \langle a(u_0),\ b(u_1)\rangle$$
$$a \to b = \lambda u.\ \ b \cdot u \cdot a$$

These operations correspond exactly to the product and (continuous) function space of the spaces represented by a and b. But these operations on retracts are continuous. Thus we can apply the same fixed-point construction to the formation of spaces. This then gives us the theoretical basis for forming the domains needed for the semantics of languages like those illustrated above in sections 3 - 5 as well as providing the fixed-point methods for showing that the semantical equations themselves have solutions.

This discussion has been merely "definition theory". Much work is being done - and much more needs to be done - in giving a proof theory and in developing applications to real languages. There are many indications that the method is both sound and flexible.