
Higher-Order and Symbolic Computation, 13, 135–152, 2000

Continuations: A Mathematical Semantics for Handling Full Jumps

CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford

CHRISTOPHER P. WADSWORTH* cpw@inf.rl.ac.uk
Independent Consultant

Abstract. This paper describes a method of giving the mathematical semantics of programming languages which include the most general form of jumps.

1. Introduction

The purpose of this paper is to explain a method of giving the mathematical semantics of a programming language which includes a very general form of jump (goto statement). The general scheme for providing the semantics equations is that outlined in [13] although there are some considerable differences in detail. Furthermore, in order to avoid as far as possible a confusing proliferation in details, the actual language described in this paper is a very rudimentary one—almost the only significant feature it has is its ability to jump at inconvenient moments, and in particular to jump out of the evaluation of an expression.

The method used, which has come to be known as the method of "continuations", has been developed from the "tail functions" of Mazurkiewicz [5] and, independently, by F.L. Morris [7], extending the work of Landin [3, 4]. Although our use of continuations has become fairly widely known by word of mouth, it has not so far been published, though Fischer [2] uses essentially the same technique (for a somewhat different purpose) and Reynolds [9] shows how the use of continuations answers the order-of-application-dependence problem for definitional interpreters.

The rest of this paper will assume a familiarity with the general method of approach explained in [12, 13]. For the benefit of readers who are unfamiliar with this, a very condensed outline is given in Appendix A.1.

2. The problem of jumps

In the semantics given in [13] the value of a command is a function which transforms the store, so that, in symbolic terms

$$\mathcal{C}[\![\gamma]\!](\rho) = \theta$$

*Contact address: Pagoda, 73 Orchard Way, Wantage, Oxon OX12 8EB, UK

where $\mathcal{C}$ is the semantic function mapping commands to their meanings, γ is a command, ρ the environment which gives the denotations associated with the identifiers in γ and θ is a store transformation, i.e.

$$\theta \in C = [S \to S]$$

where S is the domain of machine states (or stores). We use the double brackets $[\![\ ]\!]$ as an aid to the eye to separate the program text γ from the value domain expressions which form the rest of the equation.

The normal sequencing of commands is then naturally interpreted as performing one store transformation after another, so that the overall effect is that of functional composition. Thus, if we have two commands γ_0 and γ_1 with store transformations given by

$$\theta_0 = \mathcal{C}[\![\gamma_0]\!](\rho)$$
$$\theta_1 = \mathcal{C}[\![\gamma_1]\!](\rho)$$

the effect of the sequence of commands $\gamma_0;\gamma_1$ on any initial store σ will be to produce a store

$$\sigma' = \theta_1(\theta_2(\sigma))$$

so that the semantic equations for sequencing takes the form

$$\mathcal{C}[\![\gamma_o;\gamma_1]\!](\rho) = (\mathcal{C}[\![\gamma_1]\!](\rho)) \circ (\mathcal{C}[\![\gamma_0]\!](\rho)).$$

This simple scheme breaks down if γ_0 contains a jump to some external label. The difficulty is that there is no meaning we can give to $\mathcal{C}[\![\gamma_0]\!](\rho)$ which will allow us to avoid performing γ_1 after γ_0, unless we abandon the simple explication of sequencing.

Various attempts to get round this difficulty have been made, but they only work satisfactory in simple cases. They all depend, ultimately, on being able to break up the program explicitly into segments which do not contain jumps. Only inside such segments can the sequencing equation be used safely. When combining the segments it is necessary to end all possible exits by a jump so that the successor can be given explicitly. This sort of scheme works quite well for flow charts, although the large number of extra labels required seem rather inelegant, but it fails in more complicated cases.

An important situation which resists this sort of treatment is an error exit from a function (Algol 60: Type-procedure). The reason for the difficulty is that a jump is called for in the middle of evaluating an expression, while there may be several partial results around. The fact that functions involve abstraction and subsequent application is an extraneous complication which only conceals the main problem of the jump and ensures that most investigators do not consider it. We can sharpen our ideas and avoid this unnecessary complication by using the **valof-resultis** construction. In this we have an *expression* of the form **valof** γ and a *command* of the form **resultis** ϵ. (Note that we make a distinction

between commands and expressions.) The value of an expression

valof § γ_0
γ_1
...
resultis ϵ'
... §

is found by obeying the commands γ_0,γ_1, ... in sequence until a command **resultis** ϵ' is obeyed. The expression ϵ' is then evaluated and this value is taken as the value of the whole expression. (There are several language-dependent decisions to be made about what happens if there is more than one **resultis** command in the **valof** body or if none is encountered dynamically, and various problems of scope and interaction with functional abstraction need to be settled. None of these is relevant to our present enquiry so they will not be discussed further.) All programming languages which allow functions to be defined have some construction which is semantically equivalent to the **resultis** command, but many languages restrict its use so that it gets confused with function definition.

The difficult type of jump is inside the body of a **valof** block to a label which is outside. In order to illustrate the semantics of these jumps, we shall use a small programming language which we introduce in the next section.

3. A small "continuation" language

The definition follows the style of the language definitions given in [13]. We start by defining the syntactic categories and metavariables. The syntax itself is given in a very idealised form which omits all problems of parsing and ambiguity. All that remain are the basic constructions which are semantically distinct.

3.1. Syntactic categories

$\xi \in Id$	Usual Identifiers
$\gamma \in Cmd$	Commands
$\epsilon \in Exp$	Expressions
$\phi \in Fn$	Some Primitive Commands

We use Greek letters, possibly decorated with subscripts and primes, as metavariables ranging over the category indicated.

3.2. Syntax

3.2.1. Commands.

$\gamma ::= \phi$ | **dummy** |
γ_0**;**γ_1 | $\epsilon \rightarrow \gamma_0, \gamma_1$ | **while** ϵ **do** γ |
goto ϵ | § γ_0**;**ξ_1**:**γ_1**;** ... ξ_{n-1}**:**γ_{n-1}**§** |
resultis ϵ

The definition uses a version of BNF. There is no presumption that variables in the different clauses are in any way related. Inside a single clause subscripts are used to identify the various components and these will be used below in the semantic equations.

The intention of most clauses should be fairly plain. The language is largely uninterpreted (in the mathematician's sense) and most of the useful operations will presumably be primitive commands (ϕ). (Note that the assignment comes into this category.) The reason for this is that we are only interested in jumps and a small set of related commands. Leaving the others unspecified does not mean that we are unable to give semantic equations for them, merely that we are not at the moment interested.

dummy is the command which has no effect. γ_0**;**γ_1 is the way of indicating sequencing—it means γ_0 followed by γ_1. $\epsilon \rightarrow \gamma_0, \gamma_1$ is the conditional command, equivalent to Algol 60's **if** ϵ **then** γ_0 **else** γ_1. The next clause is a **while**-loop and is only included so that we can discuss its equivalence with a form involving a jump.

The next clause is the jump. If we have jumps, we clearly need some way of introducing labels, and this is the purpose of the next clause. The symbols § and **§** are statement brackets (equivalent to Algol 60's **begin** and **end**). The whole clause is to be thought as a block with $\xi_1, \ldots, \xi_{n-1}$ as labels and $\gamma_0, \ldots, \gamma_{n-1}$ as commands, generally compound, which may involve any of these labels. Any of these commands, of course, could themselves contain inner blocks. Normal rules apply about names which clash. In the language as given here, it might seem that identifiers can only stand for labels as this is the only identifier binding clause. Clearly any practical language would need some extension, so we shall allow identifiers also to denote truth values, but without specifying any explicit binding mechanism in the language. The last clause has already been discussed.

3.2.2. Expressions.

$$\begin{aligned}\epsilon ::= {} & \xi \mid \textbf{true} \mid \textbf{false} \mid \\ & \epsilon_0 \rightarrow \epsilon_1, \epsilon_2 \mid \textbf{valof}\ \gamma\end{aligned}$$

Expressions in this language form a very meagre set. They are either truth values (built up from the constants **true** and **false**) or they are label values. Either kind can be compounded by the conditional expression $\epsilon_0 \rightarrow \epsilon_1, \epsilon_2$ (where ϵ_0 needs to be a truth value though ϵ_1 and ϵ_2 can be both truth values or both label values). Similarly a **valof** expression can yield either a truth value or a label value.

4. Semantics

4.1. Value domains

As usual we start our account of the semantics by discussing the value domains for the language. The only data domain which is made explicit is T, which is the domain of truth values. We also need the domain S of machine states (or stores); we use σ (possibly decorated) as a variable denoting an individual state, and so we have $\sigma \in S$. State transformations are the functions from S to S; we shall call this domain C and use θ as its typical element.

We also need to know the denotations of the identifiers in the language. We shall call the domain of denotations D (with δ as the typical element) and discuss later what its detailed structure is. For the moment, we are only interested in the fact that such a domain exists, and that there is a function, known as an *environment* which gives the mapping from identifiers to their denotations. We shall use the letter ρ to stand for an environment, so that

$$\rho \in Env = [Id \to D].$$

4.2. Continuations

The difficulty of dealing with sequencing of commands which was mentioned in Section 2 springs ultimately from the way in which commands are combined to form a program. If, as in [13], we regard the meaning of a command γ as the store transformation $\mathcal{C}[\![\gamma]\!](\rho)$, the way in which this contributes to the meaning of a whole program depends crucially on whether it is *jump-free*—i.e. always goes on to execute the following command—or whether it is *jump-dependent*—i.e. may cause an escape exit or jump to an external label, in which case the following command is irrelevant. This means that we cannot find a satisfactory semantic equation for $\mathcal{C}[\![\gamma_0;\gamma_1]\!](\rho)$ because we cannot determine whether γ_0 is jump-free or not.

The solution to this problem is to abandon the idea of giving the state transformation for each command in isolation. We must define, instead, a semantic function which yields, for every command γ, in a program, the state transformation which would be produced from there to the end of the program. We shall use letter $\mathcal{P}$ for this function to distinguish it from the $\mathcal{C}$ of [13]. In order to deal with the effect of the program following γ, we need to supply an extra argument, θ, which is the state transformation corresponding to this part of the program. If γ is jump-free, we shall then have for the program including γ

$$\mathcal{P}[\![\gamma]\!](\rho)(\theta) = \theta \circ \mathcal{C}[\![\gamma]\!](\rho)$$

which can be interpreted as saying that the state transformation for the whole program, starting from the command γ, is that obtained by first performing the state transformation specified by γ (i.e. $\mathcal{C}[\![\gamma]\!](\rho)$) and then that specified by the rest of the program (i.e. θ). The argument θ is called a *continuation* (strictly a *command continuation*) and is of type $C = [S \to S]$. The semantic function $\mathcal{P}$ thus has the functionality

$$\mathcal{P} : [Cmd \to [Env \to [C \to [S \to S]]]].$$

The meaning of $\mathcal{P}[\![\gamma]\!](\rho)(\theta)$ when γ is jump-dependent will be discussed later in detail, but it is worth mentioning at once the crucial point that it need not depend on the argument θ. This means that it is possible to "throw away" the normal continuation for a command, and this is precisely what is needed for jumps.

The meaning of a continuation may perhaps become clearer if we consider the machine states explicitly. Suppose we have a state σ_0; a continuation θ would mean that the final

state of the machine at the end of the program would be

$$\sigma' = \theta(\sigma_0).$$

If we now want to find the effect of performing the command γ with θ as its continuation, we should use the state transformation $\theta' = \mathcal{P}[\![\gamma]\!](\rho)(\theta)$ so that the final state would be:

$$\sigma_1' = \mathcal{P}[\![\gamma]\!](\rho)(\theta)(\sigma_0).$$

This sort of expression may be unfamiliar to some readers as it involves the use of *higher order functions*—that is, functions whose arguments and results are also functions. When we use functions of this kind we often use *repeated application* (as in the last equation) with the convention that application associates to the left. Thus, if we were to insert all brackets, we should write

$$\sigma_1' = (((\mathcal{P}[\![\gamma]\!])(\rho))(\theta))(\sigma_0).$$

We nearly always leave out the association brackets and sometimes also the brackets round an argument which are normally used to indicate functional application. In this minimally bracketed form, we should write

$$\sigma_1' = \mathcal{P}[\![\gamma]\!]\rho\theta\sigma_0$$

(note that we shall always retain the text brackets $[\![\ ]\!]$).

For the purposes of this paper, any command can be considered to be a complete program and we take the meaning of a program to be the state transformation it produces when its constituent command γ is executed in the standard environment ρ_0. This is given by

$$\mathcal{P}[\![\gamma]\!]\rho_0\theta_0$$

where θ_0 is the identity function on states.

This approach to the meaning of a whole program is deliberately rather simple-minded. The reader will notice that both the initial environment ρ_0 and the initial continuation θ_0 are generally provided by the operating system. As we are only considering single programs we have used the identity function as the continuation since "no more" remains to be done when the program is completed.[1]

We are now in a position to consider the semantic equations for some clauses of the syntax. Taking sequencing first, we are looking for the value of $\mathcal{P}[\![\gamma_0;\gamma_1]\!]\rho\theta$. The intention is that we carry out γ_0 (in the environment ρ), and follow it with the continuation which arises from carrying out γ_1 (in ρ) with the continuation θ. In symbols this is

$$\mathcal{P}[\![\gamma_0;\gamma_1]\!]\rho\theta\sigma = \mathcal{P}[\![\gamma_0]\!]\rho\,\{\mathcal{P}[\![\gamma_1]\!]\rho\theta\}\sigma$$

(Here, and in future, we use braces { } as an aid to the eye to delimit continuations.) $\mathcal{P}[\![\gamma_1]\!]\rho\theta$ is the state transformation obtained by carrying out γ_1 (in ρ) before θ and this expression is therefore the correct continuation to provide for γ_0.

We can now see in outline how to deal with labels and jumps. The value of a label will be the state transformation from the labelled-point to the end of the program. (The reader can notice clearly here the development of continuations from the "tail functions" of Mazurkiewicz [5].) The semantic equation for $\mathcal{P}[\![\textbf{goto } \epsilon]\!]\rho\theta$ will simply ignore θ and use as continuation the value of ϵ. Before we can describe this in detail, however, we need to look into the use of continuations with expressions.

4.3. Expressions

In contrast to a command, whose execution yields only a machine state, an expression when successfully evaluated yields both a (possibly altered) machine state and a result. Exactly what is to be done with the result, however, is not determined by the expression itself, but by its context. It is appropriate to take account of this by using a new type of continuations known as an *expression continuation*. This has as its arguments the expression result and a state and produces in turn a final state. For technical reasons it is convenient to supply these arguments singly so that the functional nature of an expression continuation is

$$K = [E \to [S \to S]]$$

where E is the domain of expression results. In the language we are considering here, E is the same as D (the domain of values which can be denoted by identifiers) but this is not a necessary feature of all languages (see ref. [14] for a discussion of the domain structure of programming languages). We shall use δ for the typical element of both E and D and $\kappa \in K$ for an individual expression continuation. The semantic function for an expression will be denoted by $\mathcal{E}$ and has the functionality

$$\mathcal{E} : [Exp \to [Env \to [K \to [S \to S]]]].$$

The semantic equations for some expression clauses are now obvious. Thus

$$\mathcal{E}[\![\textbf{true}]\!]\rho\kappa\sigma = \kappa(tt)\sigma$$
$$\mathcal{E}[\![\textbf{false}]\!]\rho\kappa\sigma = \kappa(ff)\sigma$$

where *tt* and *ff* are the truth vales corresponding to the program text **true** and **false** (we use different symbols as the two domains are quite distinct). The clause for an identifier is almost equally simple:

$$\mathcal{E}[\![\xi]\!]\rho\kappa\sigma = \kappa(\rho[\![\xi]\!])\sigma$$

which says that we find the value denoted by ξ in the environment ρ and pass this as an argument to the continuation κ.

The fact that the same store σ occurs on both sides of these equations indicates that it has not been altered by the evaluation of the expression—in other words that **true**, **false** and identifiers ξ can be evaluated without side-effects. This is not true of expressions in general (and of **valof** expressions in particular). If the evaluation of ϵ in the environment

ρ and with a machine state σ terminates normally producing a result δ and an altered state σ', we should get

$$\mathcal{E}[\![\epsilon]\!]\rho\kappa\sigma = \kappa(\delta)(\sigma').$$

The clause for the conditional expression $\epsilon_0 \to \epsilon_1, \epsilon_2$ needs a little more discussion. We start by evaluating ϵ_0 in the environment ρ, but what continuation should we give it? If the value produced by ϵ_0 is *tt* the continuation should be to evaluate ϵ_1 (also in ρ) and pass its value to κ; if the value produced by ϵ_0 is *ff* then ϵ_2 should be chosen in place of ϵ_1. This yields the following clause:

$$\mathcal{E}[\![\epsilon_0 \to \epsilon_1, \epsilon_2]\!]\rho\kappa\sigma = \mathcal{E}[\![\epsilon_0]\!]\rho\,\{Cond(\mathcal{E}[\![\epsilon_1]\!]\rho\kappa, \mathcal{E}[\![\epsilon_2]\!]\rho\kappa)\}\,\sigma$$

where *Cond* is a general function of type

$$[V \times V] \to [T \to V]$$

(where V can be any suitable domain) such that if $p, q \in V$

$$Cond(p, q)(tt) = p$$
$$Cond(p, q)(ff) = q.$$

In our particular case V is C and, in accordance with our usual convention, we have enclosed the expression continuation in braces.

In order to verify that this is correct, let us suppose that the evaluation of ϵ in the environment ρ and with the machine state σ yield the result δ (which is *tt* or *ff*) and an altered state σ'. Then, as above,

$$\begin{aligned}\mathcal{E}[\![\epsilon_0 \to \epsilon_1, \epsilon_2]\!]\rho\kappa\sigma &= \{Cond\ (\mathcal{E}[\![\epsilon_1]\!]\rho\kappa, \mathcal{E}[\![\epsilon_2]\!]\rho\kappa)\}(\delta)(\sigma') \\ &= \begin{cases} \mathcal{E}[\![\epsilon_1]\!]\rho\kappa\sigma' & \text{if } \delta = tt \\ \mathcal{E}[\![\epsilon_2]\!]\rho\kappa\sigma' & \text{if } \delta = ff \end{cases}\end{aligned}$$

which gives the desired effect of choosing the correct expression (from ϵ_1 and ϵ_2) and compounding the side effect of evaluating ϵ_0.

5. The remaining commands

5.1. Jumps and labels

We can now discuss the semantic equation for a simple jump:

$$\mathcal{P}[\![\textbf{goto}\,\xi]\!]\rho\theta\sigma = (\rho[\![\xi]\!]|C)\sigma$$

The expression $\rho[\![\xi]\!] \mid C$ is the element of D denoted by the identifier ξ (which should be a label). This element is then *projected* into C (which implies that the domain C is a part

of D); this projection merely performs a sort of dynamic type checking. The result is then a state transformation which is applied to σ whatever the original continuation θ.

We can deal with a *computed jump* (i.e. with a general expression in place of a label identifier) almost equally easily. The equation is

$$\begin{aligned}\mathcal{P}[\![\textbf{goto}\ \epsilon]\!]\rho\theta &= \mathcal{E}[\![\epsilon]\!]\rho\{\mathit{Jump}\}\\ &\quad\text{where}\quad \mathit{Jump} \in K = E \to C\\ &\qquad\quad \mathit{Jump}(\delta) = \delta \mid C.\end{aligned}$$

Thus, if the value of ϵ in the environment ρ and with a machine state σ is a continuation θ' and if the evaluation alters the state to σ', the effect of **goto** ϵ will be

$$\begin{aligned}\mathcal{P}[\![\textbf{goto}\ \epsilon]\!]\rho\theta\sigma &= \mathit{Jump}(\theta')(\sigma')\\ &= \theta'(\sigma')\end{aligned}$$

whatever the original continuation θ.

This clearly has the right effect and manages to lose the continuation completely. Thus the command sequence **goto** ϵ**;**γ with a continuation θ will give

$$\begin{aligned}\mathcal{P}[\![\textbf{goto}\ \epsilon\textbf{;}\gamma]\!]\rho\theta\sigma &= \mathcal{P}[\![\textbf{goto}\ \epsilon]\!]\rho\{\mathcal{P}[\![\gamma]\!]\rho\theta\}\sigma\\ &= \mathcal{E}[\![\epsilon]\!]\rho\{\mathit{Jump}\}\sigma\\ &= \mathcal{P}[\![\textbf{goto}\ \epsilon]\!]\rho\theta\sigma\end{aligned}$$

for all ρ, θ and σ. This means that the two commands are strictly equivalent, which we can write, using the programming language itself, as

$$\ulcorner\textbf{goto}\ \epsilon\textbf{;}\gamma\urcorner \equiv \ulcorner\textbf{goto}\ \epsilon\urcorner$$

(The symbols $\ulcorner\ \urcorner$ are a form of quotation marks separating the programming language text from the meta-symbol $\equiv$.)

If our jumps are to be of any use, we must have some way of introducing labels into the language. This is the purpose of the next clause whose semantic equation is rather more elaborate:

$$\begin{aligned}\mathcal{P}[\![\S\ \gamma_0\textbf{;}\xi_1\textbf{:}\gamma_1\textbf{;}\ldots\xi_{n-1}\textbf{:}\gamma_{n-1}\ \S]\!]\rho\theta\sigma &= \theta_0\sigma\\ \text{where}\quad \theta_0 &= \mathcal{P}[\![\gamma_0]\!]\rho'\theta_1\\ \theta_1 &= \mathcal{P}[\![\gamma_1]\!]\rho'\theta_2\\ &\vdots\\ \theta_{n-1} &= \mathcal{P}[\![\gamma_{n-1}]\!]\rho'\theta\\ \text{and}\quad \rho' &= \rho[\theta_1,\ldots,\theta_{n-1}/\xi_1,\ldots,\xi_{n-1}].\end{aligned}$$

The meaning of this is that we first set up an environment ρ' which is the same as ρ except that the identifiers $\xi_1,\ldots,\xi_{n-1}$ here denote $\theta_1,\ldots,\theta_{n-1}$ (thus masking any former

denotation they may have had). We then set up the equations given for $\theta_0, \theta_1, \ldots, \theta_{n-1}$ which make use of the new environment ρ'. (Note that the continuation used in the last equation is the original θ, since the continuation appropriate to the last command in a block is just the continuation of the whole block.) The state transformation for the block (with continuation θ) is then θ_0.

It should be clear that the γ_i may involve any of the ξ_i, so that this set of equations is non-trivial: any θ_i may involve any or all of the θ_i (including itself) through the corresponding ξ_i and ρ'.

The solution of this set of equations may seem a formidable problem. It is, of course, the most complicated (but also the most interesting) part of semantic theory. Fortunately the mathematical theory which underlies this method [11, 12, 13] not only guarantees the existence of a solution to equations of this sort, but also gives a general method for finding it. This involves using the *minimal fixed point operator* which we write as Y; the theory proves the existence of Y and gives a formula for it.

5.2. *While-loop*

The only explicit use we shall make of Y is in the equation for a **while**-loop.

$$\mathcal{P}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!]\rho\theta = Y(\lambda\theta' \cdot \mathcal{E}[\![\epsilon]\!]\rho\{\mathit{Cond}\mathcal{P}[\![\gamma]\!]\rho\theta', \theta)\})$$

To see how this works, let us consider the function $H : C \to C$ such that

$$H(\theta') = \mathcal{E}[\![\epsilon]\!]\rho\{\mathit{Cond}(\mathcal{P}[\![\gamma]\!]\rho\theta', \theta)\}.$$

For any specific θ', $H(\theta')$ is a state transformation; comparison with the equation for a conditional expression shows that the meaning of $H(\theta')\sigma$ is:

Evaluate ϵ in environment ρ with state σ; let the result be δ with an altered state σ'.
If $\delta = tt$ perform the command γ in the environment ρ with a continuation θ' and an initial state σ'.
If $\delta = ff$, perform only the continuation θ on the state σ'.

If we now identify θ' with $H(\theta')$ we can see that the result will be a satisfactory interpretation of the **while**-loop. The argument of Y in the semantic equation is just the function H written in the λ-notation and $Y(H)$ gives us the minimal fixed point of H—i.e. the minimal solution of the equation $\theta' = H(\theta')$. (In this context the minimal solution is the solution which is defined over the smallest possible domain of arguments. This turns out to be exactly what we need for our semantic equations.)

Let us introduce the command **dummy**, which has no effect and hence has the semantic equation

$$\mathcal{P}[\![\textbf{dummy}]\!]\rho\theta = \theta$$

and the conditional command $\epsilon \to \gamma_0, \gamma_1$, which, by analogy with the conditional expression has the equation

$$\mathcal{P}[\![\epsilon \to \gamma_0, \gamma_1]\!]\rho\theta = \mathcal{E}[\![\epsilon]\!]\{\mathit{Cond}(\mathcal{P}[\![\gamma_0]\!]\rho\theta, \mathcal{P}[\![\gamma_1]\!]\rho\theta)\}.$$

We can now write a block in the programming language which should be equivalent to the loop **while** ϵ **do** γ. One form of this is

$$\S\ \textbf{dummy;}\ \xi\textbf{:}\epsilon \to (\gamma\textbf{;goto}\ \xi), \textbf{dummy}\ \S,$$

where the label identifier ξ does not occur in ϵ or in γ. It is a relatively simple matter to prove this equivalence. We have:

$$\mathcal{P}[\![\S\ \textbf{dummy;}\ \xi\textbf{:}\epsilon \to (\gamma\textbf{;goto}\ \xi), \textbf{dummy}\ \S]\!]\rho\theta = \theta_0$$
$$\text{where}\quad \theta_0 = \mathcal{P}[\![\textbf{dummy}]\!]\rho'\theta_1$$
$$\theta_1 = \mathcal{P}[\![\epsilon \to (\gamma\textbf{;goto}\ \xi), \textbf{dummy}]\!]\rho'\theta$$
$$\text{and}\quad \rho' = \rho[\theta/\xi].$$

Calculating, we get

$$\begin{aligned}
\theta_0 &= \theta_1\\
\theta_1 &= \mathcal{E}[\![\epsilon]\!]\rho'\{\mathit{Cond}(\mathcal{P}[\![\gamma\textbf{;goto}\ \xi]\!]\rho'\theta, \mathcal{P}[\![\textbf{dummy}]\!]\rho'\theta)\}\\
&= \mathcal{E}[\![\epsilon]\!]\rho'\{\mathit{Cond}(\mathcal{P}[\![\gamma]\!]\rho'\{\mathcal{P}[\![\textbf{goto}\ \xi]\!]\rho'\theta\}, \theta)\}\\
&= \mathcal{E}[\![\epsilon]\!]\rho'\{\mathit{Cond}(\mathcal{P}[\![\gamma]\!]\rho'\{\mathcal{E}[\![\xi]\!]\rho'\{\mathit{Jump}\}\}, \theta)\}\\
&= \mathcal{E}[\![\epsilon]\!]\rho'\{\mathit{Cond}(\mathcal{P}[\![\gamma]\!]\rho'\theta_1, \theta)\}
\end{aligned}$$

where we can write ρ in place of ρ' in the last equation because ξ does not occur in ϵ or γ.

Remembering our definition of H above we get

$$\theta_0 = \theta_1 = H(\theta_1).$$

Thus θ_1 is also a fixed point of H. A more formal statement of the semantic equations for a block would ensure that θ_1 is the minimal fixed point of H and therefore identical to $Y(H)$, the semantic expression for the **while**-loop.

Similar methods can be used to prove several general equivalences between programs such as:

$$\begin{aligned}
\ulcorner\textbf{while}\ \epsilon\ \textbf{do}\ \gamma\urcorner &\equiv \ulcorner\epsilon \to (\gamma\textbf{;while}\ \epsilon\ \textbf{do}\ \gamma), \textbf{dummy}\urcorner\\
\ulcorner\epsilon_0 \to \textbf{goto}\ \epsilon_1,\ \textbf{goto}\ \epsilon_2\urcorner &\equiv \ulcorner\textbf{goto}\ \epsilon_0\ \to \epsilon_1, \epsilon_2\urcorner\\
\ulcorner\epsilon \to \gamma_0, \gamma_1\urcorner &\equiv \ulcorner\S\ \textbf{goto}\ \epsilon_0 \to \xi_0, \xi_1\\
&\qquad \xi_0\textbf{:}\ \gamma_0\textbf{;goto}\ \xi_2\textbf{;}\\
&\qquad \xi_1\textbf{:}\ \gamma_1\textbf{;goto}\ \xi_2\textbf{;}\\
&\qquad \xi_2\textbf{:}\ \textbf{dummy}\ \S\urcorner.
\end{aligned}$$

The last is subject to identifiers ξ_0, ξ_1 and ξ_2 not occurring in ϵ, γ_0 or γ_1.

5.3. Valof and resultis

In order to deal with a **valof** expression we need to set aside temporarily the expression continuation and provide some other, command-type continuation for the body. Since **resultis** commands are interpreted as being bound to the smallest textually surrounding **valof** block (in the same way that ordinary identifiers are bound to their denotations), it is appropriate to save the expression continuation in the environment by creating a special element **res** to denote it. To do this we extend the environment domain to consist of pairs, a mapping from *Id* to *D* as before, together with an expression continuation associated with **res**:

$$\rho \in Env = [[Id \to D] \times K].$$

By a convenient abuse of notation, we shall continue to write $\rho[\![\xi]\!]$ and $\rho[\delta/\xi]$ to refer to the $[Id \to D]$ component of ρ, and we shall write $\rho[\![\mathbf{res}]\!]$ for the second component of ρ, and $\rho[\kappa/\mathbf{res}]$ for the environment ρ' which is the same as ρ except that the K-component of ρ' is κ.[2]

The semantic equation for a **valof** expression can now be given.

$$\mathcal{E}[\![\mathbf{valof}\ \gamma]\!]\rho\kappa\sigma = \mathcal{P}[\![\gamma]\!](\rho[\kappa/\mathbf{res}])\{Fail\}$$

We expect γ, the body of the **valof** expression, to be terminated by a **resultis** command. If, however, the program is wrongly constructed and the end of the body, γ, is reached by normal sequencing, we need to signal an error. This is done by providing special command continuation *Fail* which will only be used if execution of the body is completed without a **resultis**-command being obeyed.

When a **resultis** ϵ command is encountered we whish to reinstate the original expression continuation (which is now the environment) and provide it with the value of ϵ as an argument. So we get:

$$\mathcal{P}[\![\mathbf{resultis}\ \epsilon]\!]\rho\theta = \mathcal{E}[\![\epsilon]\!]\rho\{\rho[\![\mathbf{res}]\!]\}.$$

In this equation $\rho[\![\mathbf{res}]\!]$ is the original κ which was inserted by $\rho[\kappa/\mathbf{res}]$ in the equation for **valof**.

We note in passing that escapes such as the **break**-command, which causes an immediate exit from the smallest enclosing loop, can be treated in an analogous fashion.

6. Discussion

We have now completed the explication of the semantics of our language. The results are collected together on two pages as Appendix A.2. One point which was left unsettled has now been resolved—the nature of the domain *D*. It must contain *C* (for label valued variables), and, in order to give some point to the language, *T* (although the language makes no provisions for creating identifiers denoting truth values).

Appendix A.2 also gives the functionality of the semantic functions, the syntactic categories (or domains) and the clauses of the idealised syntax. It thus effectively defines the complete language, apart from the details of the concrete syntax.

The discussion on the semantic equations in this paper has been deliberately informal. The whole method of giving the mathematical semantics is generally unfamiliar and, as it involves a rather large amount of mathematical notation, it takes some time to get used to. Again, familiarity is the cure: those of us who have worked with continuations for some time have soon learned to think of them as natural and in fact often simpler than the earlier methods.

One very important consequence of the informality of our discussion is that no proper consideration has been given to questions of termination. If a continuation is the state transformation to the end of the program, how do we deal with programs which fail to terminate? The short answer to this is that such a continuation is wholly undefined and so has the value $\perp$ (bottom). All our domains are constructed in such a way that there is a partial ordering on them which can be thought of as being based on the amount of information contained in an element. The element with no information in each domain is denoted by the symbol $\perp$, with its domain indicated by a subscript, though this is often omitted. An unending loop will have a continuation value of $\perp_C$ and it will be this value which is given by the minimal fixed-point operator Y. (The treatment of non-terminating programs raises several questions which we do not attempt to deal with here. Reynolds [9, 10] has discussed some of the factors involved.)

The method of continuations introduced here can be extended to the whole of a programming language with no great difficulty. In particular, it can be used with block structure, abstraction and application (definition and use of procedures) both recursive and non-recursive. It will also handle a jump out of recursively nested calls of recursively defined procedures (if the language allows it). Even more extreme jumps are possible. In a language such as PAL [1] which permits assignments it may be possible to jump out of an expression and then later jump back into it again and resume the process of evaluation. Continuations are sufficiently powerful to deal with such a situation. (This could not be taken to imply approval of jumps back into expressions as a language design feature—but if a language can specify something, however odd, the method used to give its formal semantics must be powerful enough to describe it.)

In contrast to [13] our main semantic functions now take three arguments: an environment, a continuation and a store. (For some languages further arguments are needed; in Algol 60 and Algol 68, for instance, the coercion context for expressions is supplied to their semantic functions [6, 8].) Together these arguments may be regarded as the *semantic context* in which each part of the program must be interpreted before its contribution to the meaning of the whole program can be determined. The environment and the store provide all necessary information about the history of the computation preceeding the part under consideration; the continuation indicates how the computation will proceed to the end of the program unless the current control causes a jump. Notice, however, that the semantic context does not implicitly contain any of the "house-keeping" information introduced when carrying out the computation on a machine. In Section 4 of our exposition, there is no mention, for example, of how implementations keep track of the current execution point or

catalogue the partial results of an expression evaluation. Language descriptions which keep the details of run-time organization below the surface reveal the semantics more clearly and allow an interpreter to choose strategies suited to his machine.

This paper is part of the outcome of a collaboration with Dana Scott started in the autumn of 1969. We have since been joined by the staff and the students at the Programming Research Group. The aim has been to lay a firm foundation for a method of specifying the mathematical semantics of programming languages which is sufficiently general to deal with many different ones. The need to describe already existing languages has acted as a useful corrective to the mathematician's tendency to simplify and generalise. Actual languages are not only peppered with gritty little problems which greatly lengthen and complicate their formal description, but they also contain genuine difficulties, such as error exits from functions. These are often overlooked because their importance is not appreciated.

One distinction we have striven to maintain throughout our work is that between semantics—given a program, what function does it compute—and implementation—how is a machine to be organised to execute the actions specified by programs? This distinction takes many forms; it is found, for example, in the need to distinguish between a data object and its machine representation, or, perhaps, even more fundamentally, between functions and algorithms. The role of mathematical semantics is to give a precise, unambiguous definition of *what* programs mean, sufficient to determine their outcome, while remaining uncommitted as to the details of *how* this outcome is to be achieved on a (real or abstract) computing machine. In this way we hope to focus on the "essential meanings" of language constructs and thus explain the equivalence of programs and prove correctness of their implementations. Some theories about implementations have already been established [6]. The proofs of most involve complex structural inductions on the clauses of the semantic functions to verify that certain properties of the semantic context are invariant under all computations; for a large language this can be a tedious and error-prone activity.

Much work remains to be done, but we have now reached the stage where the methods are sufficient for conventional programming languages. Full descriptions of PAL, Algol 68 and Algol 60 have been prepared. The last of these will be published shortly [8]; considerations of time may rule out publication of the other two. Work continues and from time to time we hope to publish small papers such as this giving progress reports. The time for a unified presentation is still some way off.

Appendix A.1: Mathematical semantics

The method aims to produce a *mathematical* rather than *operational* semantics by specifying the equivalence between constructs in the programming language and certain mathematical entities rather than discussing in any way how the program is to be implemented. Thus the imperative and dynamic features of the language are interpreted as "change of state" functions whose domain and range are both machine states.

The means adopted to display this equivalence is to define a number of *semantic functions* whose domain lie in the programming language text and whose ranges are the machine states

and various higher order functional objects associated with them. (These include, of course, the data types on which the program is operating.)

In order to specify these semantic functions it is necessary to examine the syntax of the programming language, but only the barest minimum of attention is paid to it. Unlike some other approaches, we are not concerned with symbol manipulation starting from the original text, but with a far more abstract view of the syntax. The syntax is given in an abbreviated version of BNF, but the number of clauses is reduced to a minimum; all questions of parsing and some of admissibility are left out of consideration. In any actual implementation of a real language, there would need to be a further stage of specifying a concrete, unambiguous syntax which incorporated the syntactic restrictions placed on the language.

The semantic functions are then defined by a set of mutually recursive equations, one for each clause in the idealised syntax. The semantic functions will occur on both sides of these equations, but those occurring on the right hand side will have arguments which are component parts of the clause which is the left hand argument. This method of definition allows us to concentrate our attention on the semantic structure of the language—of how the value (= meaning) of a clause is built up from the values of its components.

One particular semantic function plays an important part throughout. This is the *environment* (for which we reserve the letter ρ) which gives the values denoted by the identifiers in the language. The environment can only be altered by a variable binding operation (such as a definition or procedure call). Other commands may alter the machine state (for which we use the letter σ) but they will leave the environment unaltered. Another important difference between ρ and σ is that changes to σ are irreversible,—i.e. only one σ at the time can be kept in the machine. New environments on the other hand are usually thought of as being modifications of an old one, and it is normal to keep them both. Thus on exit from a block the old environment (but not the old σ) is restored.

It is important to keep the notation used under very strict control. We have found that a careful choice of letters, type faces and brackets aids the eye and makes the necessarily rather long formulae easier to read. In particular, we use upper case script letters for semantic functions (with the exception of ρ), lower case Greek letters for variables whose nature is determined by the letter (e.g. ϵ for expressions, θ for state transformations, etc.) italic type is used for basic functions in the value domains (e.g. *Conds*, *Y*). Programming language text is enclosed in double brackets and its reserved words are printed in bold face (e.g. **while**, **goto**).

Appendix A.2: A Small "continuation" language

Syntactic categories

$\xi \in Id$	Usual Identifiers
$\gamma \in Cmd$	Commands
$\epsilon \in Exp$	Expressions
$\phi \in Fn$	Some Primitive Commands

Syntax

$$
\begin{aligned}
\gamma ::= {} & \phi \mid \textbf{dummy} \mid \\
& \gamma_0;\gamma_1 \mid \epsilon \to \gamma_0, \gamma_1 \mid \textbf{while } \epsilon \textbf{ do } \gamma \mid \\
& \textbf{goto } \epsilon \mid \S\ \gamma_0;\xi_1 : \gamma_1; \ldots \xi_{n-1} : \gamma_{n-1} \S \mid \\
& \textbf{resultis } \epsilon \\
\epsilon ::= {} & \xi \mid \textbf{true} \mid \textbf{false} \mid \\
& \epsilon_0 \to \epsilon_1 \epsilon_2 \mid \textbf{valof } \gamma
\end{aligned}
$$

Value domains

T	Truth Values
S	Machine States (Stores)
$\theta \in C = [S \to S]$	Command Continuations
$\delta \in D = [T + C]$	Denotations
$\delta \in E = [T + C]$	Expression Results
$\kappa \in K = [D \to C]$	Expression Continuations

Semantic functions

$\rho : [[Id \to D] \times K] = Env$	Environments
$\mathcal{P} : [Cmd \to [Env \to [C \to C]]]$	
$\mathcal{E} : [Exp \to [Env \to [K \to C]]]$	

Semantic equations

C1. $\mathcal{P}[\![\phi]\!]\rho =$ (Some given function $[C \to C]$ associated with ϕ)

C2. $\mathcal{P}[\![\textbf{dummy}]\!]\rho\theta = \theta$

C3. $\mathcal{P}[\![\gamma_0;\gamma_1]\!]\rho\theta = \mathcal{P}[\![\gamma_0]\!]\rho\,\{\mathcal{P}[\![\gamma_1]\!]\rho\theta\}$

C4. $\mathcal{P}[\![\epsilon \to \gamma_0, \gamma_1]\!]\rho\theta = \mathcal{E}[\![\epsilon]\!]\{Cond(\mathcal{P}[\![\gamma_0]\!]\rho\theta, \mathcal{P}[\![\gamma_1]\!]\rho\theta)\}$

C5. $\mathcal{P}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!]\rho\theta = Y(\lambda\theta' \cdot \mathcal{E}[\![\epsilon]\!]\rho\{Cond(\mathcal{P}[\![\gamma]\!]\rho\theta', \theta)\})$

C6. $\mathcal{P}[\![\textbf{goto } \epsilon]\!]\rho\theta = \mathcal{E}[\![\epsilon]\!]\rho\,\{Jump\}$
where $Jump(\delta) = \delta \mid C$

C7. $\mathcal{P}[\![\S\ \gamma_0;\xi_1:\gamma_1; \ldots \xi_{n-1}:\gamma_{n-1}\ \S]\!]\rho\theta = \theta_0$

$$
\begin{aligned}
\text{where} \quad \theta_0 &= \mathcal{P}[\![\gamma_0]\!]\rho'\theta_1 \\
\theta_1 &= \mathcal{P}[\![\gamma_1]\!]\rho'\theta_2 \\
&\vdots \\
\theta_{n-1} &= \mathcal{P}[\![\gamma_{n-1}]\!]\rho'\theta \\
\text{and } \rho' &= \rho[\theta_1, \ldots, \theta_{n-1}/\xi_1, \ldots, \xi_{n-1}]
\end{aligned}
$$

C8. $\mathcal{P}[\![\textbf{resultis } \epsilon]\!]\rho\theta = \mathcal{E}[\![\epsilon]\!]\rho\ \{\rho[\![\textbf{res}]\!]\}$

E1. $\mathcal{E}[\![\xi]\!]\rho\kappa = \kappa(\rho[\![\xi]\!])$

E2. $\mathcal{E}[\![\textbf{true}]\!]\rho\kappa = \kappa(tt)$

E3. $\mathcal{E}[\![\textbf{false}]\!]\rho\kappa = \kappa(ff)$

E4. $\mathcal{E}[\![\epsilon_0 \rightarrow \epsilon_1, \epsilon_2]\!]\rho\kappa = \mathcal{E}[\![\epsilon_0]\!]\rho\{Cond(\mathcal{E}[\![\epsilon_1]\!]\rho\kappa, \mathcal{E}[\![\epsilon_2]\!]\rho\kappa)\}$

E5. $\mathcal{E}[\![\textbf{valof } \gamma]\!]\rho\kappa = \mathcal{P}[\![\gamma]\!](\rho[\kappa/\textbf{res}])\ \{Fail\}$

Notes

1. The reader may ask if it is any more justifiable to take a single program in isolation than it is to take a single command. The answer, of course, is that it is not, and that in the same way as command continuations are needed to explain jumps inside programs, further hierarchical levels of continuations, such as process continuations, job continuations and operating system continuations, will be needed to give the semantics of the operating system. The outer-most level (or possibly levels) are not inside the machine at all and are implemented by operator intervention.

   We do not discuss the use of continuations in the semantics of operating systems any further in this paper as to do so would require a fuller understanding of the differences between operating systems and programs that is yet at our disposal. It would also make the paper much too long . . .
2. An alternative approach would be to include **res** in *Id* as a special identifier, not accessible to the programmer, and consequently to include its range K in the domain D so that *Env* remains $[Id \rightarrow D]$. In the language considered here, however, no mechanism is provided for binding ordinary identifiers to expression continuations. To include K in D would therefore not be compatible with [13], where D is reserved for the domain of values denotable by program identifiers. Richer languages might, of course, include further constructs which provide a suitable binding mechanism, e.g. Landin's J-operator [4].

References

1. Evans, A., Jr. PAL—a language for teaching programming linguistics. In *Proc. 23rd ACM National Conference*, Brandon Systems, Princeton, N.J., 1968, pp. 395–403.
2. Fischer, M.J. Lambda-calculus schemata. *LISP and Symbolic Computation* **6**(3/4) (1993) 259–288. An earlier version appeared in an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.
3. Landin, P.J. The mechanical evaluation of expressions. *Computer Journal* **6**(4) (1964) 308–320.
4. Landin, P.J. The next 700 programming languages. *Communications of the ACM* **9**(3) (1966) 157–164.
5. Mazurkiewicz, A. Proving algorithms by tail functions. *Information and Control* **18**(3) (1971) 220–226.
6. Milne, R.E. The formal semantics of computer languages and their implementations. Ph.D. Thesis, Cambridge University, 1974. Also as Technical Monograph PRG-13, Oxford University Computing Laboratory, Programming Research Group.
7. Morris, F.L. The next 700 formal language descriptions. *LISP and Symbolic Computation* **6**(3/4) (1993) 249–258.
8. Mosses, P.D. The mathematical semantics of Algol 60. Technical Monograph PRG-12, Oxford University Computing Laboratory, Programming Research Group, 1974.
9. Reynolds, J.C. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, Boston, 1972.
10. Reynolds, J.C. On the interpretation of Scott's domains. *Informatica Teorica*, Vol. 15 of *Symposia Mathematica*, Instituto Nazionale di Alta Matematica Roma, 1975, pp. 123–135. Distributed by Academic Press, London.

11. Scott, D. Continuous lattices. In *Proc. of the 1971 Dalhousie Conference*. Lecture Notes in Mathematics, Vol. 274, pp. 97–136, Springer-Verlag, 1972. Also as Technical Monograph PRG-7, Oxford University Computing Laboratory, Programming Research Group.
12. Scott, D. Outline of a mathematical theory of computation. In *Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems*, 1970, pp. 169–176. Also as Technical Monograph PRG-2, Oxford University Computing Laboratory, Programming Research Group.
13. Scott, D. and Strachey, C. Toward a mathematical semantics for computer languages. In *Proc. of the Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn, 1971. Also as Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group.
14. Strachey, C. Varieties of programming language. In *Proc. of the International Computing Symposium*, Cini Foundation, 1972, pp. 222–233. Also as Technical Monograph PRG-10, Oxford University Computing Laboratory, Programming Research Group.