SPECIFYING THE UNDERLYING CONTROL STRUCTURES OF PROGRAMMING LANGUAGES IN THEIR DENOTATIONAL SEMANTICS

Masaki Nakagawa
Tokyo University of Agriculture and Technology
Koganei, Tokyo, Japan

The denotational semantics of a programming language specifies the meaning of programs in that language. Generally it is designed to be implementation free. However, current architectures require special techniques to implement block structured programming languages such as stacks and activation records, by which their underlying control structures are realized. A domain M such that $M \cong V \to N \times V \times M$, where N: names, V:values, is claimed to be suitable for specifying the required implementation techniques for the underlying control structures in a denotational semantics. We hope this kind of semantics, together with the syntax, provides sufficient specification of block structured programming languages to enable us to automate generation of their compilers.

1. INTRODUCTION

There are several motivations in providing a mathematical semantics of a programming language. We can design the semantics to provide implementers of the language with a precise description from which to produce a correct compiler; or to offer programmers in that language a reference standard on which to specify the behaviour of their programs; or to be composed of a small number of fundamental constructs for language designers to make a clean, mathematically neat programming language.

Although a great deal of progress has been made by D. Scott, C. Strachey, and other people in this discipline [1], [2], [3], there remains a considerable amount of work to be done in the development and application of their research towards the full achievement of the goals stated above. In this paper we are investigating the design of a semantics to help implementers of block structured languages.

As a starting point, we shall consider the semantics of an extremely simple programming language BREF (Block structured programming language with Recursive Functions). This language includes problems common to block structured programming languages such as ALGOL-60, -68, PASCAL, etc. The problem we will address is how to describe the underlying control structures of these languages in their denotational semantics. The semantics must specify stack mechanism but should not impose or imply much of the implementation detail for a particular computer, since the language could be implemented on various machines. Therefore, another goal of this sort of semantics is in revealing underlying control structures more rigorously than any other means without specifying too many fine details.

2. A BRIEF DESCRIPTION OF BREF

We now give a brief informal description of the BREF language.

2.1 Syntax

Syntactic domains

$i \in ID$ (identifiers)
$e \in Exp$ (expressions)
$n \in Nml$ (numerals)

The small letters i, e and n are to be used optionally with primes or subscripts as variables over the domains of identifiers, expressions and numerals, respectively.

Production

e::=n | i | lambda i. e | if e_1 then e_2 else e_3 |
e_1; e_2 | while e_1 do e_2 |
let i = e_1 in begin e_2 end |
let rec i = e_1 in begin e_2 end | (e) |
$e_1(e_2)$ | ref e | val i | i := e |
arithmetic predicates such as n > 1 etc. |
arithmetic expressions such as n - 1 etc.

Fig. 1. The Syntax of BREF

2.2 An Informal Description of the Semantics

BREF is a simple example of common block structured programming languages. It would be a subset of any such language, so the reader who is familiar with them may soon learn how to write programs in this language without much explanation. However, we shall point out some important semantic features of this language.

(i) BREF is an expression language. An expression may have side-effects, otherwise "e_1 ; e_2", "while e_1 do e_2" and "i :=e" lose their raisons d'être.

(ii) The usual scope rules for block structured languages apply in BREF.

(iii) The binding mechanism is static.
An expression is evaluated in the presence of an environment whose purpose is to bind each identifier to a value. The environment is the conceptualization of a symbol table constructed and referred to during a compilation.

(iv) BREF requires the notion of states, i.e., a mapping from locations into storable values.

(v) Only call-by-value is allowed for the function application.
The application of a function to an argument is performed by the expression: $e_1(e_2)$. It is evaluated in the same way as in PASCAL, etc.

(vi) BREF is a stack oriented language.
The underlying control structures of many block structured programming languages can be modelled by a system of activation records which are allocated on a stack (in ALGOL-60, -68, etc.) or in a heap (in SIMULA-67 [4], GEDANKEN [5], etc.). Compared with many low level languages and some high level languages, such as FORTRAN, the computational structures in the above languages are less transparent from a text of a program. It is extremely difficult to understand how recursion, coroutines or quasi-parallel processes are realized in a computer without knowledge of their underlying control structures. For example, the declarations of recursive functions (procedures) can be very brief and concise in a program, but their evaluation is realized by the rather complicated activation record allocation and deallocation mechanism on a stack. The concept of activation records is sufficient in order for BREF, ALGOL-60 and -68 to allow recursive functions (procedures) and for SIMULA-67, GEDANKEN to incorporate coroutines and quasi-parallel processes. We shall describe the underlying control structure of BREF algorithmically and illustrate the point by using an example. A fuller explanation is presented in [6], [7].

As mentioned above, the computational structure of BREF is considerably different to the structure of a program text. Therefore, the compiler has to produce extra code by which the underlying control structure is realized at run-time. Consequently, our brief description of the underlying control structure is divided into two parts: one is about the action at compile-time, while the other is about that at run-time.

At compile-time

A template for an activation record of a block or a function (procedure) is created. It has a pointer to the compiled code; its binding slot for local variables is filled in with declared bindings for that block/function (but lambda parameters are not bound to any values); its static-link points to the template for the activation record of the block/function which textually includes this block/function #; its dynamic-link points to the above activation record if this activation record is for a block, but it is not filled in at compile-time if the activation record is for a function.

#The right side of a recursive function is evaluated in the environment created by itself as well as preceding bindings. This cyclic environment is conceptually realized by the cyclic structure created between the activation record of the recursive function and that of a block/function for which the recursive function is declared. I.e., the static-link in the former points to the latter, while in the local binding slot of the latter, the name of the recursive function points to the former. This is illustrated in fig. 2.

Dynamic-links of functions are filled in at run-time since the control must be returned to the point immediately following a function call in a program.

At run-time

The template for the activation record of the outermost block is copied onto the stack. The computation follows the compiled code.

When an inner block is about to be evaluated, the template for its activation record is copied onto the top of the stack, the computation with the activation record of the outer block is suspended and the code of the top activation record is executed. When execution of this code is completed, the activation record is relinquished from the stack, and the computation of the outer block is resumed. (This is what we call stack discipline.)

If a function (either recursive or nonrecursive) is about to be evaluated, the template for its activation record is copied onto the top of the stack and its dynamic-link is filled in by a pointer to the activation record to which the control must be returned after the function call. After the completion of the code, the activation record is relinquished, and the code of the activation record pointed to by the dynamic-link in the function is resumed. Evaluating a recursive function sequentially creates and sequentially relinquishes several activation records of the same template with their lambda parameters bound to separate arguments.

The following example illustrates the above process. Consider the program Pr:

```
let N=3
in begin                                          ┐
   let rec fact=lambda i. if i=1 then 1           │
                          else i*fact(i-1)        │
   in begin                                       │
      fact(N)   ┐  block B                        │ block A
      end       ┘                                 │
   end                                            ┘
```

At compile-time of Pr

The following three templates for the activation records of block A, the recursive function "fact" and block B are created:

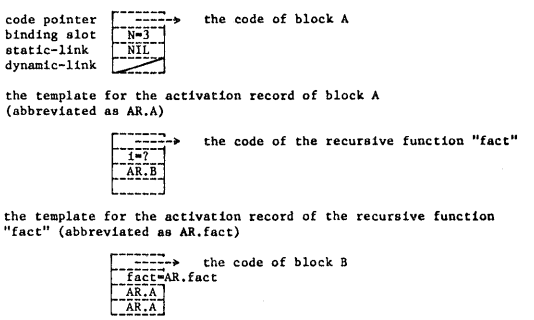

Fig. 2. The three Templates for the Activation Records

At run-time of Pr

(1) A copy of AR.A is placed on the stack.
(2) A copy of AR.B is stacked.
(3) The execution eventually starts from the code of AR.B.
(4) When the function application fact(N) is evaluated, AR.fact is copied onto the stack. The value of N is fetched from AR.A along its static-link and finally the code is executed with the lambda parameter i bound to 3.
(5) When fact(3-1) is evaluated, the computation with the above activation record is suspended, another copy of AR.fact is placed on top of the stack, and the code is executed with i bound to 2.
(6) Evaluating fact(2) entails the evaluation of fact(1). The similar step to (5) is taken. But, in this case, fact(1) yields the answer 1. This answer is returned to the activation record which is pointed to by the dynamic-link of the activation record for fact(1).
(7) The completed activation record is relinquished and the computation with the activation record pointed to by the former activation record is resumed. This action is repeated until the answer fact(3) is returned as the value of block A.

The denotational semantics of BREF presented in section 4 rigorously specifies all of the above features, but the main emphasis is placed on the specification of the feature (vi), i.e., the underlying control structure.

3. MODULES AND COMPUTATION BY MODULES

In the Scott-Strachey approach, the meaning of a program is, fundamentally, a function from states to states. So we are not given, nor should we expect, any explicit indication of how a language is to be implemented from this. We may be able to gain, implicitly, some ideas as to what an implementation might look like from the text denoting the function, but this would only be fortuitous.

For our purpose, then, functions do not seem to be adequate. Our model of underlying control structures is a system of activation records which obeys the stack discipline or unrestricted allocation mechanism in a heap. As described above, each activation record behaves as an automaton. It takes an input from some automaton and returns a value to another and changes its state into a renewal state. Its behaviour is not just the computation of a mathematical function of its input. Rather it has a body where input values are processed. Therefore we have to ask what are the suitable semantic constructs which correspond to functions in the Scott-Strachey approach.

3.1 Definition of Modules

Consider the following recursive domain equation:
$M \cong V \to N \times V \times M$
where N is a domain of names for modules and V is that of values. (This will be specified later.) Whether domains are complete lattices [8] or cpo's [9], [10], it can be proved that there exists a domain M which satisfies the above equation and that the domain M and that of automata are isomorphic [11], [12]. An element of the domain M is termed a module. A module $m \in M$ takes an input $v \in V$ and returns an output $v' \in V$ to a name $n \in N$ and also a renewal module $m' \in M$. This domain was first considered by R. Milner to provide a denotational semantics of a non-deterministic programming language [11], [13]. But, since it can describe the behaviour of automata denotationally, it enables us to specify the underlying control structure of BREF in its denotational semantics.

A value which a module receives is either a state, or a state with some information for the module. But, the former case can be considered as a special case of the latter such that the information received is just the control, i.e., the message "start your computation". We assume that the special symbol "cc" denoting the control is in the domain of basic values, which are not further specified.

The domain V is formally defined as follows:

B (basic values) with $cc \in B$
T (Boolean values)
$\mathbb{N}$ (integers)
$\mu \in N$ (names of modules)
$I = B + \mathbb{N} + T + M + (\mathbb{N} + T) \times N$ (information)
S (states)
$V = I \times S$ (values)

3.2 Computation by Modules

Computation by modules is realized by internal computation by the modules and external communications between them. The computation by a module m where $m = \lambda v.\langle n(v), v'(v), m'(v)\rangle$ is performed by the application $v'(v)$. If $m'(v) = \perp$ (bottom or undefined), the module is to be relinquished or garbage collected. If $m'(v) = m$ for $\forall v$, the module m is a pure reentrant module and it is not $\perp$ as in the Scott-Strachey approach.

The communications between modules have now to be formalized. Suppose there is a module m_1 such that $m_1 = \lambda v.\langle n(v), v'(v), m'(v)\rangle$. If the name $n(v)$ is that of a module m_2, the module m_2 must be applied to the output value $v'(v)$. As the module m_2 sends its output value to another module and so on, the communications "proceed". If there are many modules, to describe all at every snapshot of communications is troublesome and unnecessary, because only one module will be active at any one time. Therefore, we describe such a module and ignore all others. The proceeding of communications among modules is described by the notation:

$$
\begin{array}{l}
\quad\vdots \\
\Rightarrow \lambda v_1.\langle n_1(v_1), v_1'(v_1), m_1'(v_1)\rangle\ v \\
\Rightarrow \lambda v_2.\langle n_2(v_2), v_2'(v_2), m_2'(v_2)\rangle\ v_1'(v) \\
\Rightarrow \quad\vdots
\end{array}
$$

where $n_1(v)$ is the name of the second module.

When we wish to skip the finer detail such as c_2, c_3 in $c_1 \Rightarrow c_2$, $c_2 \Rightarrow c_3$, $c_3 \Rightarrow c_4$, we write $c_1 \,..\Rightarrow c_4$.

4. A DENOTATIONAL SEMANTICS OF BREF

To define a denotational semantics of BREF, the notion of states and environment must be formalized. The simplest idea is that a state is a mapping from locations (references) into storable values, while the environment is a mapping from identifiers into denotable values. Since it is not our aim to have their detailed models, we adopt the above idea. R. Milne and C. Strachey consider such models [2].

L (locations)
Vs = $\mathbb{N}$ + T (storable values)
Vd = $\mathbb{N}$ + T + L + M (denotable values)
$\sigma \in S = L \to Vs$ (states)
$\rho \in Env = ID \to Vd$ (environment)

The extension of the environment ρ with x bound to v is expressed as $\rho[v/x]$. The Greek letters σ, ρ and μ, optionally with primes or subscripts, are variables over S, Env, and N respectively.

We assume all domains are complete lattices and define continuous functions on them. As mentioned before, however, we may assume them to be cpo's. In that case, we only have to ignore the augmented top, $\top$, of each domain which is added to make it a complete lattice.

(i) Useful functions and operators

(1) injection X in I
injection from X (B, $\mathbb{N}$, T, M, Vs x N) into I

(2) projection $|B$, $|\mathbb{N}$, $|T$, $|M$, $|(Vs \times N)$
e.g.

$$v|B = \begin{cases} \top_B & \text{if } v = \top_I \\ b & \text{if } v = b \text{ in I for } b \in B \\ \bot_B & \text{otherwise} \end{cases}$$

However, we are not rigorous in our use of injection and projection functions.

(3) Cartesian product functions < ,..., >

(4) selection x @ i
selects the ith element from a tuple x

(5) conditional functions
$Cond_A$: (A x A) → (T → A)

$$Cond_A(a,a')t = \begin{cases} a & \text{if } t = \text{true} \in T \\ a' & \text{if } t = \text{false} \in T \\ \bot_A & \text{if } t = \bot_T \\ \top_A & \text{if } t = \top_T \end{cases}$$

(6) equality functions $=_A$: A x A → T

$$a =_A a' = \begin{cases} \top_T & \text{if } a = \top_A \text{ or } a' = \top_A \\ \bot_T & \text{if } a = \bot_A \text{ or } a' = \bot_A \\ \text{true} & \text{if a and a' are identical} \\ \text{false} & \text{otherwise} \end{cases}$$

(7) fixed point operators
Y_{An}: $(A^n \to A^n) \to A^n$

For the last three, we omit the subscript A when it is clear from the context.

(ii) Auxiliary functions

Before proceeding to the semantic definition of BREF, auxiliary functions are introduced.

(1) new: S → L "new" returns a new location given a state σ.
(2) name: M → N
For a newly created module m, name returns a new name. Or, less abstractly, name(m) may be considered as a (base) address of the new module.

(iii) Semantics of BREF

The semantic function for BREF ξ has functionality:
Exp → (Env → (N → M)). The third argument N is similar to the continuation in the Scott-Strachey approach. Because, $\xi[[\ e\ ]]\rho\mu$ which is a module has to know the module to which it sends its output value. The brackets [[ ]] are used to emphasize that the argument enclosed by [[ ]] is a syntactic object, and is not part of the metalanguage for the semantic specification.

$$\xi[[n]]_{\rho\mu} = \lambda v.<\mu, <\alpha[[n]], v@2>, \bot>$$

We shall not specify the interpretation of numerals. A semantic function α: Nml → $\mathbb{N}$ is assumed.

$$\xi[[i]]_{\rho\mu} = \lambda v.<\mu, <\rho[[i]], v@2>, \bot>$$

$$\xi[[\text{lambda i. e}]]_{\rho\mu} =$$

$$\lambda v.<\mu,<\lambda v'.\xi[[e]]_{\rho',\mu'},<cc \text{ in I}, v'@2> \text{in V}, v@2>, \bot>$$

where $\rho' = \rho[\{v'@1|(Vs \times N)\}@1/i]$,
$\mu' = \{v'@1|(Vs \times N)\}@2$

Given a pair consisting of information and a state, it returns to the module named μ, together with a state, the module which is to receive the value for "i" and the name of a module (dynamic-link).

$$\xi[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]]_{\rho\mu} = m$$

where $m = \lambda v.<name(m_1), <cc, v@2>,$
$\lambda v'.<Cond(name(m_2), (name(m_3))v'@1|T,$
$<cc, v'@2>,$
$\lambda v''.<\mu, v''>, \bot>>>$

$m_1 = \xi[[e_1]]_{\rho name(m)}$
$m_2 = \xi[[e_2]]_{\rho name(m)}$
$m_3 = \xi[[e_3]]_{\rho name(m)}$

Or more formally:

$$m = (Y_4 \lambda <m, m_1, m_2, m_3>.<\text{the above definition of m},$$
$$\xi[[e_1]]_{\rho name(m)},$$
$$\xi[[e_2]]_{\rho name(m)},$$
$$\xi[[e_3]]_{\rho name(m)}>)@1$$

But, we prefer to use the former.

$\xi[[e_1; e_2]]_{\rho\mu} = m$

where $m=\lambda v.<name(m_1), <cc,v@2>,$

$\lambda v'.<name(m_2), <cc,v'@2>,$

$\lambda v".<\mu,v", \perp>>>$

$m_1=\xi[[e_1]]_{\rho\, name(m)}$

$m_2=\xi[[e_2]]_{\rho\, name(m)}$

$\xi[[while\ e_1\ do\ e_2]]_{\rho\mu} = m$

where $m=\lambda v.<name(m_1), <cc,v@2>,$

$\lambda v'.Cond(<name(m_2), <cc,V'@2>, m>,$

$<\mu, <cc,v'@2>, \perp>)v'@1|T>$

$m_1=\xi[[e_1]]_{\rho\, name(m)}$

$m_2=\xi[[e_2]]_{\rho\, name(m)}$

$\xi[[let\ i = e_1\ in\ begin\ e_2\ end]]_{\rho\mu} = m$

where $m=\lambda v.<name(m_1), <cc,v@2>,$

$\lambda v'.<name(m_2), v',$

$\lambda v".<\mu,v", \perp>>>$

$m_1=\xi[[e_1]]_{\rho\, name(m)}$

$m_2=\lambda v'''.\xi[[e_2]]_{\rho[v'''@1|Vd/i]name(m)}<cc,v'''@2>$

$\xi[[let\ rec\ i = e_1\ in\ begin\ e_2\ end]]_{\rho\mu} = m$

where $m=\lambda v.<name(m_1), <cc,v@2>,$

$\lambda v'.<name(m_2), v',$

$\lambda v".<\mu,v", \perp>>>$

$m_1=\xi[[e_1]]_{\rho'\, name(m)}$

$m_2=\lambda v'''.\xi[[e_2]]_{\rho'\, name(m)}<cc,\ v'''@2>$

where $\rho'=\rho[v'''@1|Vd/i]$

Note that "e_1" is evaluated in the same environment as "e_2", hence the module m_1 feels the environment which the module m is creating.

$\xi[[(e)]]_{\rho\mu} = \xi[[e]]_{\rho\mu}$

$\xi[[e_1(e_2)]]_{\rho\mu} = m$

where $m=\lambda v.<name(m_1), <cc,v@2>,$

$\lambda v'.<name(m_2), <cc,v'@2>,$

$\lambda v".<name(v'@1|M),$

$<<v"@1|Vs,\mu>in\ I,v"@2>,\perp>>>$

$m_1=\xi[[e_1]]_{\rho\, name(m)}$

$m_2=\xi[[e_2]]_{\rho\, name(m)}$

The module m_1 should return a module as its output information, and the module m_2 should return an argument. The former is applied to the latter by the renewal of the module m.

$\xi[[ref\ e]]_{\rho\mu} = m$

where $m=\lambda v.<name(m'), <cc,v@2>,$

$\lambda v'.<\mu, <new(v'@2),\sigma'>, \perp>>$

where $\sigma'=\lambda l.Cond(v'@1,v'@2(l))(l=new(v'@2))$

$\xi[[i := e]]_{\rho\mu} = m$

where $m=\lambda v.<name(m'), <cc,v@2>,$

$\lambda v'.<\mu,<v'@1, Ns>, \perp>>>$

where $Ns=\lambda l.Cond(v'@1,v'@2(l))(l=\rho[[i]])$

For the above two, $m' = \xi[[e]]_{\rho\, name(m)}$

$\xi[[val\ i]]_{\rho\mu} = \lambda v.<\mu, <v@2(\rho[[i]]|L), v@2>, \perp>$

We do not specify the semantics for all the arithmetic predicates and expressions. They are all similar to the examples given:

$\xi[[e_1 - e_2]]_{\rho\mu} = m$

where $m=\lambda v.<name(m_1), <cc,v@2>,$

$\lambda v'.<name(m_2), <cc,v'@2>,$

$\lambda v".<\mu,<v'@1-v"@1,v"@2>, \perp>>>$

$\xi[[e_1 > e_2]]_{\rho\mu} = m$

where $m=\lambda v.<name(m_1), <cc,v@2>,$

$\lambda v'.<name(m_2), <cc,v'@2>,$

$\lambda v".<\mu,<(v'@1>v"@1),v"@2>,\perp>>>$

For the above two,

$m_1=\xi[[e_1]]_{\rho\, name(m)}$, $m_2=\xi[[e_2]]_{\rho\, name(m)}$

5. A SPECIMEN EVALUATION

Consider the semantics of the program Pr, i.e., $\xi[[Pr]]_{\rho_0\mu_0}<cc,\sigma_0>$ where σ_0, ρ_0 and μ_0 denote an initial state, an initial environment and the name of an initial module respectively. μ_0 can be considered to be the name of the module $\lambda v.<ee, v, \perp>$, where the symbol "ee" denotes the name of a display (terminal) module.

$\xi[[Pr]]_{\rho_0\mu_0}<cc,\sigma_0>$

:

$=> \xi[[inside\ of\ block\ A]]_{\rho_0[3/N]\ name(m)}<cc,\sigma_0>$

where $m=\lambda v.<\mu_0,v, \perp>$

$\rho_1=\rho_0[3/N]$

$=m_1<cc,\sigma_0>$

where $m_1=\lambda v.<name(m_2),<cc,v@2>,$

$\lambda v'.<name(m_3), v',$

$\lambda v".<name(m), v", \perp>>>$

$m_2=\xi[[factbody]]_{\rho_2 name(m_1)}$

$m_3=(\lambda v"'.\xi[[inside\ of\ block\ B]]_{\rho_2 name(m_1)})$

$<cc,v"'@2>$

$\rho_2=\rho_1[v"'@1|M/fact]$

$factbody=lambda\ i.\ if\ i=1\ then\ 1$

$else\ i*fact(i-1)$

:

$=> (\lambda v.\xi[[fact(N)]]_{\rho_2 name(m_1)}<cc,v@2>)<Vm,\sigma_0>$

where

$Vm=\lambda v'.\xi[[factbody]]_{\rho_2[v'@1@1/i]v'@1@2}<cc,v'@2>$

$\rho_2=\rho_1[Vm/fact]$

$=\xi[[fact(N)]]_{\rho_2 name(m_1)}<cc,\sigma_0>$

Note that $m_1=\lambda v".<name(m), v", \perp>$ now.

:

$=>(\lambda v'.\xi[[factbody]]_{\rho_2[v'@1@1/i]v'@1@2}{}^{<cc,v'@2>}$

$<<3,name(m_1)>,\sigma_0>$

$=\xi[[factbody]]_{\rho_2[3/i]name(m_1)}<cc,\sigma_0>$

$=m_4\langle cc,\sigma_0\rangle$

where $m_4=\lambda v.\langle name(m_5),\langle cc,v@2\rangle,$

$\lambda v'.\langle Cond(name(m_6),name(m_7))v'@1|T,$

$\langle cc,v'@2\rangle,$

$\lambda v''.\langle name(m_1),v'',\perp\rangle\rangle\rangle$

$m_5=\xi[[i=1]]_{\rho_3 name(m_4)}$

$m_6=\xi[[1]]_{\rho_3 name(m_4)}$

$m_7=\xi[[i*fact(i-1)]]_{\rho_3 name(m_4)}$

$\rho_3=\rho_2[3/i]$

$\Rightarrow m_5\langle cc,\sigma_0\rangle$

:

$\Rightarrow m_4\langle false \in T,\sigma_0\rangle$ where m_4 is the first renewal of the above

$\Rightarrow m_7\langle cc,\sigma_0\rangle$

Note that $m_4=\lambda v''.\langle name(m_1),v'',\perp\rangle$ now.

$=m_8\langle cc,\sigma_0\rangle$

where $m_8=\lambda v.\langle name(m_9),\langle cc,v@2\rangle,$

$\lambda v'.\langle name(m_{10}),\langle cc,v'@2\rangle,$

$\lambda v''.\langle name(m_4),$

$\langle v'@1*v''@1,v''@2\rangle,\perp\rangle\rangle\rangle$

$m_9=\xi[[i]]_{\rho_3 name(m_8)}$

$m_{10}=\xi[[fact(i-1)]]_{\rho_3 name(m_8)}$

:

$\Rightarrow m_9\langle cc,\sigma_0\rangle$

$\Rightarrow m_8\langle 3,\sigma_0\rangle$ where m_8 is the first renewal of the above

$\Rightarrow m_{10}\langle cc,\sigma_0\rangle$

Note that $m_8=\lambda v''.\langle name(m_4),\langle 3*v''@1,v''@2\rangle,\perp\rangle$ now.

$=\xi[[fact(i-1)]]_{\rho_3 name(m_8)}\langle cc,\sigma_0\rangle$

:

$\Rightarrow \xi[[factbody]]_{\rho_3[2/i]name(m_8)}\langle cc,\sigma_0\rangle$

$\rho_4=\rho_3[2/i]$

:

$\Rightarrow m_{12}\langle cc,\sigma_0\rangle$

where $m_{12}=\xi[[i*fact(i-1)]]_{\rho_4 name(m_{11})}$

$m_{11}=\lambda v''.\langle name(m_8),v'',\perp\rangle$

:

$\Rightarrow \xi[[i]]_{\rho_4 name(m_{13})}\langle cc,\sigma_0\rangle$

where $m_{13}=\lambda v''.\langle name(m_{11}),\langle 2*v''@1,v''@2\rangle,\perp\rangle$

$\Rightarrow \xi[[fact(i-1)]]_{\rho_4 name(m_{13})}\langle cc,\sigma_0\rangle$

:

$\Rightarrow \xi[[factbody]]_{\rho_4[1/i]name(m_{13})}\langle cc,\sigma_0\rangle$

:

$\Rightarrow m_{13}\langle 1,\sigma_0\rangle$

$\Rightarrow m_{11}\langle 2,\sigma_0\rangle$

$\Rightarrow m_8\ \langle 2,\sigma_0\rangle$

$\Rightarrow m_4\ \langle 6,\sigma_0\rangle$

$\Rightarrow m_1\ \langle 6,\sigma_0\rangle$

$\Rightarrow m\ \langle 6,\sigma_0\rangle$

$\Rightarrow \mu_0\ \langle 6,\sigma_0\rangle$

$\Rightarrow \langle ee,\langle 6,\sigma_0\rangle,\perp\rangle$

The above series of communications can be regarded as a rigorous specification of the underlying control structure when the program Pr is evaluated.

ACKNOWLEDGEMENT

The author would like to thank Mr. Phil Collier, Dr. R. Turner, Dr. P. Hayes at the University of Essex, Prof. E. Goto, Prof. N. Yoneda at the University of Tokyo and Prof. N.E. Takahashi at Tokyo University of Agriculture and Technology for their valuable comments and encouragement. Thanks are also due to the editor, reviewers and Miss Kristine K. Olson whose comments, criticisms and suggestions improved the presentation and content of this paper.

REFERENCES

[1] D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, Proc. Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, 1971, 19-46.

[2] R. Milne and C. Strachey, A theory of programming language semantics, Chapman and Hall, London, 1976.

[3] J. Stoy, The Scott-Strachey approach to the mathematical semantics of programming languages, MIT Press, Cambridge, MIT, Dec. 1977.

[4] G. Birtwistle, L. Enderin, M. Ohlin and J. Palme, DEC. SYSTEM-10 SIMULA language handbook, part 1, Swedish National Defence Research Institute and the Norwegian Computing Center, Stockholm

[5] J.C. Reynolds, GEDANKEN-A simple typeless language based on the principle of completeness and the reference concept, Communications of the ACM, Vol. 13, No. 5, May 1970, 308-319.

[6] R. Bornat and B.J. Wielinga, Does AI programming really have to be like knitting with spaghetti, Proc. AISB AI Summer School, 1976.

[7] C. Hewitt, Viewing control structures as patterns of passing messages, Artificial Intelligence, Vol. 8, No. 3, June 1977, 323-364.

[8] D. Scott, Continuous lattices, Proc. Dalhousie Conference, Springer Lecture Notes Series, No. 274 Springer-Verlag, Heidelberg, 1972.

[9] G. Plotkin, A powerdomain construction, SIAM J. on Computing, Vol. 5, No. 3, Sep. 1976, 452-487.

[10] M. Smyth, Powerdomains, J. of Computer and System Science, Vol. 16, 1978, 23-36.

[11] R. Milner, Processes: a mathematical model of computing agents, Logic Colloguium '73, North-Holland, Amsterdam, 1975, 157-174.

[12] M. Nakagawa, Mathematical semantics for parallel computation, M. Sc. dissertation, Univ. of Essex, Dec. 1978.

[13] R. Milner, An approach to the semantics of parallel programs, Proc. Convegno di Informatica Teorica, Istituto di Elaborazione della Informazione, Pisa, 1973.