A DENOTATIONAL SEMANTICS FOR SHARED-MEMORY PARALLELISM AND NONDETERMINISM

Daniel M. Berry

February 1985
Report No. CSD-850004

A DENOTATIONAL SEMANTICS FOR SHARED-MEMORY PARALLELISM AND NONDETERMINISM*

Daniel M. Berry
Computer Science Department
University of California
Los Angeles, CA 90024 USA

Abstract

It is first shown how to construct a continuation from a deterministic Vienna Definition Language control tree. This construction is then applied to nondeterministic control trees. The result is a denotational but not quite continuation semantics for arbitrary shared-memory nondeterminism and parallelism. The implications of this result are discussed.

---

* This research was supported in part by the Department of Energy, Contract No. DE-AS03-76SF0034 P.A. No. DE-AT0376, ER70214, Mod. A006.

1. Introduction

1.1 What is done

This paper describes the development of a denotational semantics for nondeterminism and parallelism in a shared memory. The denotation of a program fragment is a tree representing a partial ordering of actions to be performed in the rest of the computation. Forking in the tree represents spawning of processes. Each action in the tree causes either an indivisible state change or a macro expansion to a subtree of other actions to be performed. Thus, this denotation is similar to Plotkin's resumptions [Plo76, Smy78]. All such trees of actions are finite even if the program gives rise to nonterminating processes, i.e., macro expansion to subtrees occurs only at run-time as needed.

It is convenient and advantageous to use VDL control trees as the tree denotation described above. It is convenient, because all the required mechanisms have already been set up [LLS70, LW69]. It is advantageous, because, as it is shown, given any VDL definition of a block structured language which is in a certain generally achievable normal form, it is possible to convert this interpreter definition into a denotational semantics of the same language giving the control trees as the meaning of program fragments.

In addition, it is shown that each control tree is an encoding of a function, called a P-continuation, which is on the value and storage returned by an instruction to the set of all answers obtained from all possible computations starting with the given instruction. It turns out the the function encoded by a control tree cannot be used in place of the control tree in the semantic equations, because information about possible interleavings has been lost.

Plotkin and later Smyth [Plo76, Smy78] present the power domain construction of resumptions R, where

$$R = S \rightarrow P[S+(S\times R)]$$

where S is a domain of states. These resumptions may be taken as the denotation of program fragments giving rise to processes which access shared memory. Plotkin and Smyth observe that functions on states to sets of states are not powerful enough to model arbitrary interleaving.

Schwarz presents a power domain semantics to handle an expression (without assignment) language with features that require parallelism [Sch79].

Francez, Hoare, Lehmann, and de Roever [FHLR79] present a semantics of CSP, which has its processes communicate via message passing rather than through shared memory. This restriction in means of interaction between processes allows use of domains simpler than power domains. They ascribe to each process description a potentially infinite history tree with potentially infinite branching which denotes all potential communications with other processes. The meaning of a collection of processes is obtained by binding their history trees, i.e., by matching in pairs of history trees potential communications that represent actual communications between the processes owning the trees. The result of the binding process is not in the same domain of history trees, so it cannot be further bound.

A later work by Francez, Lehmann, and Pnueli [FLP80] improves on this last work by offering as the semantics of a process description a relation between sets of attainable states and the communication sequence needed to attain these states. This sequence, being linear, is simpler than the potentially infinitely branching history trees of the previous work. They provide a binding operator which matches and merges two or more processes' communication sequences to obtain the semantics of the combined processes in the same domain. Thus the result of a binding can be further bound. They remark that to be able to handle arbitrary interleaved access to shared memory instead of just message passing would require the use of more complex power domain constructions.

Stoughton [Sto81] uses a mapping from states to sets of potentially infinite computation sequences as the meaning of a program or system. Interestingly, he programs the system's scheduler into his equations so that the scheduler can be varied from definition to definition.

The contribution of this paper is a denotational semantics which

1. is powerful enough to model arbitrary interleaved access to shared memory,
2. is finite,
3. is systematically constructible from a VDL interpreter definition of the same language, and
4. is an encoding for an easily constructed function on states to sets of states.

This last contribution is important because a definition is not considered abstract enough unless the meaning of a program is a function on states to states or sets of states.

1.2 History of this Paper

The history of this paper is instructive. In my graduate class on operational semantics, I teach the Vienna Definition Language (VDL) [LW69, LLS70, Weg72] as a language well-suited for writing operational definitions of programming languages. We cover several example definitions as well as how to convert a collection of instruction definitions into the state transition function they represent. The examples exhibit deterministic, nondeterministic, and parallel computations.

In the 1979 version of the course I decided to teach denotational semantics [Ten76, Ten78, Gor79] as well after having presented VDL. The lecture leading into the denotational semantics attempted to motivate the continuation idea by constructing a continuation from a deterministic control tree. Once the construction was understood, the functionalities of the continuation and of the semantic meaning function were obvious. Then it was easy to explain and understand denotational continuation semantics.

The construction had proceeded from deterministic control trees because I knew that nondeterminism had defied denotational, continuation semantics treatment [Gor79, ADA79]. In any case, after the class was over, just for the heck of it, I tried applying the construction also to nondeterministic control trees -- it seems to have worked.

1.3 Outline

This paper assumes at the outset familiarity with denotational, continuation semantics as described by Tennent or Gordon [Ten76, Ten78, Gor79]. The second Section, based on the lecture mentioned above, informally describes the construction of a continuation from a deterministic control tree. In order that a full knowledge of VDL is not needed to appreciate the basis of the construction, the discussion resorts to informal pictorial descriptions of VDL machine behavior. However, an acquaintance with the informal description of VDL and of the EPL machine given in [LLS70] or [Weg72] is helpful.

The third Section formalizes Section 1's construction of a continuation from a control tree. Because of the terseness of this Section, familiarity with the definition of VDL in [LLS70] or [LW69] helps. However, the formal development is self-contained.

The fourth Section returns to an informal discussion to apply a similar construction to nondeterministic control trees to obtain what is called a P-continuation (parallel continuation).

The fifth Section formalizes this new construction. This development assumes the construction of the third Section, and with that Section, is self-contained.

Finally, the sixth Section evaluates what has been done and attempts to clarify its relationship to the various existing kinds of semantics.

This paper uses the original, perhaps antiquated, VDL notation because it is talking about the original VDL and is showing that, in fact, the more modern presentations of semantics say exactly the same things. In addition, it is best to use the original VDL notation when talking about VDL so that yet another notation does not have to be developed.

2. From Deterministic Control Trees to Continuations (Informal Discussion)

2.1 Computations and Meanings

VDL is used to write information structure models [Weg70] (ISMs). An ISM is an abstract machine formally described as a triple*,

$$(\widehat{\text{is-state}},\widehat{\text{is-initial-state}},\Lambda),$$

the set of possible states, the set of initial states, and a state transition function Λ. In these first two Sections, all machines are deterministic so Λ applied to a state either yields the next state or is undefined indicating that the given state is a final state.

$$\Lambda:\widehat{\text{is-state}}\rightarrow\widehat{\text{is-state}}+\{\text{undefined}\}.$$

A computation is a possibly infinite sequence of states

$$<\xi_0,\xi_1,\ldots,\xi_{i-1},\xi_i,\ldots>$$

such that

a. ξ_0 is an initial state,

b. for each ξ_i with $i>0$,

$$\xi_i=\Lambda(\xi_{i-1}),$$

c. the sequence is not a proper initial subsequence of another sequence satisfying a) and b). (This condition guarantees that a finite sequence ends with a state ξ_n such that $\Lambda(\xi_n)=\text{undefined}$.)

If in the sequence, there is a state ξ_n such that $\Lambda(\xi_n)=\text{undefined}$, then the computation halts. Diagrammatically a finite computation may be shown as:

---

*Recall that in VDL, sets are defined by predicates. The predicate p is used to denote the set $\{x|p(x)\}$. Given a predicate p, $\hat{p}=\{x|p(x)\}$.

$$\begin{tikzcd}
\xi_0 \arrow[r, bend left, "\Lambda"] & \xi_1 \arrow[r, bend left, "\Lambda", "//" marking] & \xi_{i-1} \arrow[r, bend left, "\Lambda"] & \xi_i \arrow[r, bend left, "\Lambda", "//" marking] & \xi_{n-1} \arrow[r, bend left, "\Lambda"] & \xi_n
\end{tikzcd}$$

is-state(ξ_i), for all $i \geq 0$
is-initial-state(ξ_0)
$\Lambda(\xi_{i-1}) = \xi_i$, for all $i > 0$
$\Lambda(\xi_n)$=undefined

Denotational semantics has always tried to take as the meaning of a program the function it computes from the initial to the final state if it exists. That is, the meaning of a program p is a function f such that

$$f(\xi_0) = \begin{cases} \xi & \text{if the computation of } p \text{ from the initial state} \\ & \xi_0 \text{ halts at } \xi, \\ \perp & \text{if the computation of } p \text{ from the initial state} \\ & \xi_0 \text{ does not halt.} \end{cases}$$

One of the goals of this paper is to show the construction of this f from the state transition function Λ.

In addition, whatever the semantics, denotational semantics constructs meaning in syntax-directed manner; that is, the meaning of a construct is constructed from that of its direct syntactic components. It is typical that the meanings of all constructs, programs, assignment statements, single tokens, etc. are of the same type, e.g., function from start state to end state if it exists. Thus, viewing the abstract machine from a denotational framework, the meaning of a particular program construction, e.g., an assignment statement, is the composition of Λ across the states from the beginning to the end of the execution of the construct. Diagrammatically, the assignment $i := i + 1$ may be viewed in the context of a whole computation as shown in Figure 1.

Note the denotational idea of syntax-directed composition of Λ. The meaning of $i := i + 1$ is constructed from the meanings of i, $:=$, and $i + 1$. The meaning of $i := i + 1$ contributes to the construction of the meaning of the statement list containing $i := i + 1$. All of these meanings are of the same functionality, i.e., state to state.

The next Subsection begins the construction of these program and construct meaning functions in detail using an extension of the VDL EPL machine described in [LLS70] and also in [Weg72]. EPL is a simple block structured language with

1. integer and logical variables that must be declared in blocks,
2. potentially recursive procedures and functions with typeless, by-reference formal parameters, and
3. assignment, conditional, procedure call, and nested block statements.

The extension, called EPL+, adds label values, label variables, gotos, and while loops. The label and goto extensions appeared first in an unpublished report [BW72]. Appendix I contains the complete definition of EPL+.

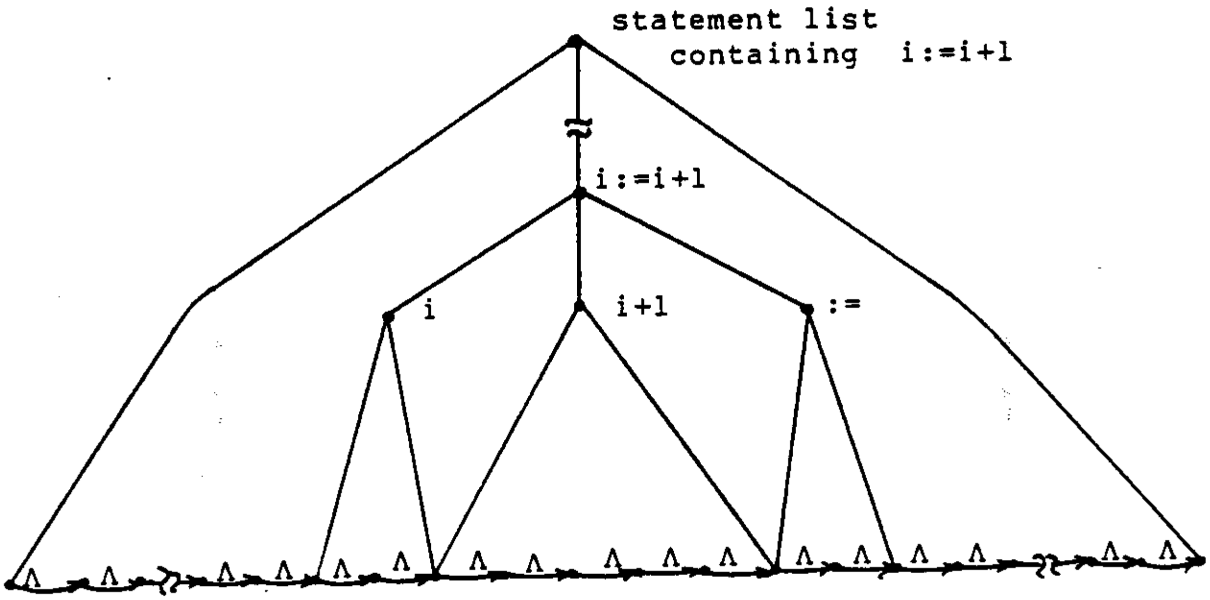


Figure 1

2.2 Modifications to EPL+ Machine

First, the EPL+ machine is given a deterministic control tree by getting rid of all forking in the control tree. This is done by

1. changing all commas in control tree representations into semicolons,

2. changing all macros which expand into sets of instructions, e.g.,

```
int-decl-part(t)=
        null;
           {int-decl(id(s-env(ξ),id(t))|id(t)≠Ω}
```

into sequentially recursing macros, e.g.,

```
int-decl-part(t)=
      is-<>(t)→null;
      T→int-decl-part(tail(t));
              int-decl(s-id(head(t))(e-env(ξ)),
                    s-attr(head(t)))
```

This change may, in some cases, necessitate a change in the syntax of the program. An object which is a set of things, e.g., an object satisfying the predicate is-decl-part, may have to be changed into a list of things conveying the same information, e.g., an object satisfying the predicate is-decl-list, in order to permit sequencing through the things.

Second, the state is reorganized so that it has only a storage, an environment and a control tree (which is deterministic) i.e.,

```
is-state=(<s-stg:is-stg>,<s-env:is-env>,<s-c:is-c>),
```

or to use more conventional denotational semantics notation

```
is-state=S×U×Cont.
```

It is necessary in this change to account for the information of the missing components, the unique name counter, the attribute and denotation directories, and the dump, in order to insure that correctness has been preserved.

The unique name counter is dispensed with by rewriting the **un-name** instruction to return any *n* such that `nos-stg(ξ)=Ω`. In order to distinguish unallocated locations for which `nos-stg(ξ)=Ω` from allocated but uninitialized locations, the latter will have the value UNINIT.

The information normally found in the attribute and denotation directories is moved into the storage.

```
is-stg=({<n:(<s-dn:is-dn>,<s-st:is-type>)>||is-n(n)})

is-dn=is-proc-dn∨is-funct-dn∨is-value∨is-label-dn∨is-UNINIT
```

Since the dump saves the environment and control of the calling block or procedure instance, to get rid of the dump it is necessary to reorganize the block, procedure, and function entry and exit:

1. Instead of dumping the current control tree and then overwriting the current control tree with a new one, the new one is macro-appended to the current one.

2. The environment which is saved in the dump to be restored by **exit** is now made an actual parameter of the **exit**, which restores it from its formal parameter.

Thus, for example,

int-block is rewritten as

```
int-block(t)=
          exit(s-env(ξ));
             int-st-list(s-st-list(t));
                int-decl-part(s-decl-part(t));
                   update-env(s-decl-part(t))
```

and **exit** is now

```
exit(env)=s-env:env
```

The result of this last change is to obtain one long linear control tree in place of the dump (stack) of control trees.

After the first change to make the machine deterministic, a typical state might appear as in Figure 2.

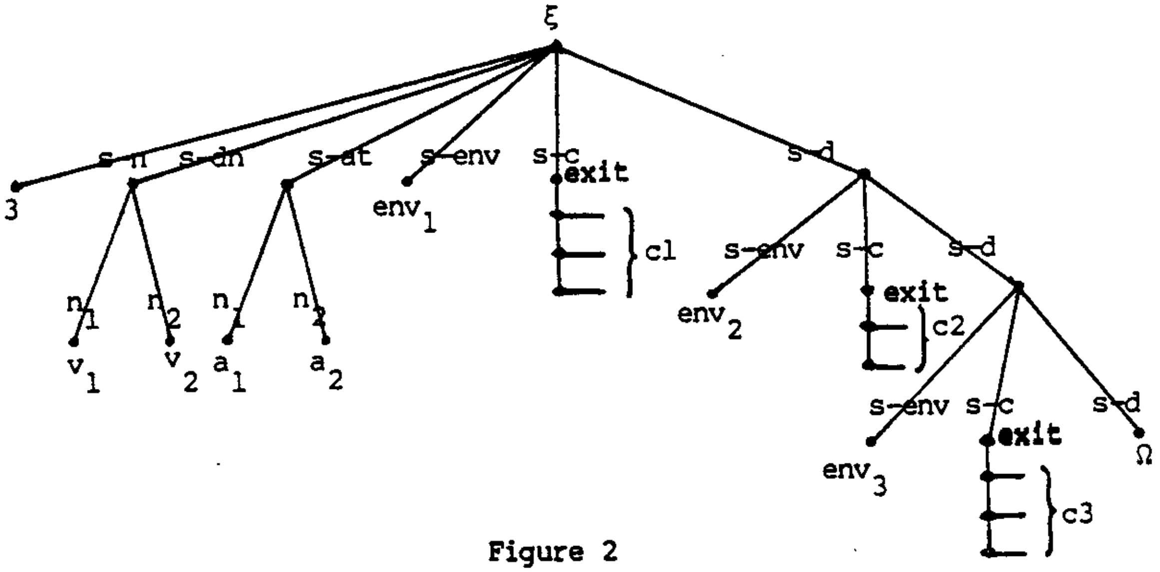


Figure 2

The effect of the second change is to change this state into that of Figure 3.

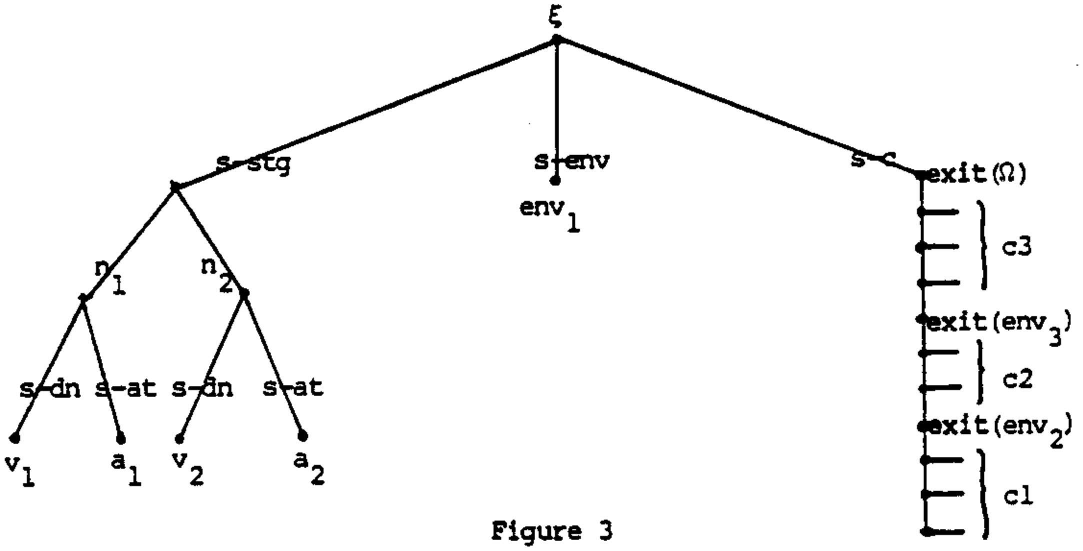


Figure 3

Appendix II contains the definition of EPL+ obtained by applying the above described modification to the definition found in Appendix I.

2.3 Key Observations

Observe what happens to label values by these modifications. In [BW72] it is shown that a label value in the EPL+ machine is a dump,

$$\text{is-label-dn}=(\langle \text{s-c}:\text{is-c}\rangle,\langle \text{s-env}:\text{is-env}\rangle,\langle \text{s-d}:\text{is-d}\rangle),$$

the control tree stating what is to be executed from arrival at the labeled statement until the end of the containing block, the environment being that in which these statements are to be executed, and the dump giving the environment, control, and dump to use after exiting this containing block. After these modifications, a label value ends up being simply a control tree paired with an environment,

$$\text{is-label-dn}=(\langle \text{s-c}:\text{is-c}\rangle,\langle \text{s-env}:\text{is-env}\rangle),$$

the control tree telling how to execute from the labeled statement until the end of the computation and the environment being that in which to execute at least the first instruction in the control tree.

Thus, it is clear that a control tree, after the above modifications, says exactly how to execute from the state owning it until the end of the computation if this end exists and from the state owning it on if the end does not exist. Thus control trees serve the same function as do Mazurkiewicz's tail functions [Maz71]. One can then view the construction of f from Λ as applying the control tree of the current state to the environment and storage of the same state to yield the final state if it exists or $\perp$ if the final state does not exist:

$$\text{if } \xi=\mu_0(\langle \text{s-stg}:stg\rangle\langle \text{s-env}:env\rangle,\langle \text{s-c}:cont\rangle) \text{ then}$$

$$f(\xi) =$$

$$cont(env,stg) = \begin{cases} \xi_f = \mu_0(\langle \text{s-stg}:stg_f\rangle,\langle \text{s-env}:env_f\rangle,\langle \text{s-c}:cont_f\rangle), \\ \qquad \text{if the computation from } \xi \text{ halts at } \xi_f, \\ \perp, \quad \text{otherwise.} \end{cases}$$

Here for the sake of generality f is considered to return a full state, but in the EPL+ machine, $env_f=\Omega$ and $cont_f=\Omega$. Thus, it would suffice to let the result of f be simply stg_f, the final storage, if it exists and $\perp$ otherwise.

A deterministic control tree, such as *cont*, is comprised of one terminal or leaf node and the rest of the tree. In order to execute from ξ until the end, be it a final state or $\perp$, one must execute the current instruction found in the terminal node, obtaining a new state ξ' and then the rest of the control tree from ξ' until the end. As a consequence, application of *cont* to *env* and *stg* is the execution of **inst** at *cont*'s terminal node in *env* and *stg* to yield a new environment *env'*, a new storage *stg'*, and a new control tree *cont'*, followed by application of *cont'* to *env'* and *stg'*. That is,

$$cont(env,stg) = cont'(env',stg'),$$

where *cont'*, *env'*, and *stg'* are yielded by executing the instruction **inst**, at *cont*'s terminal node, in *env* and *stg*.

2.4 Outline of Construction

The full construction outlined here is found in Sections 2.5 through 2.7. These sections may be skipped if the construction is clear from this outline. Using the above described point of view, one can construct a denotational meaning function which gives the meaning of an instruction in terms of the current environment, the current storage, and what is to be done when the instruction is done. What is to be done when the instruction is done is denoted by the rest of the control tree, *rst*, and the return information, *return−info*, that results from removing the current instruction from the current control tree.

$$\mathcal{M}\ [\![\mathbf{inst}]\!]\ \mathit{env}\ <\mathit{rst},\mathit{return{-}info}>\ \mathit{stg} = \mathit{cont'}\ \mathit{env'}\ \mathit{stg'}$$

For each kind of instruction, there is a particular meaning equation scheme for it.

1. For

$$\begin{aligned}\mathbf{inst}{=}\ & \mathsf{PASS}{:}\mathit{val}\\ & \quad \mathbf{s{-}stg}{:}\mathit{stg'}\\ & \quad \mathbf{s{-}env}{:}\mathit{env'},\end{aligned}$$

the equation scheme is

$$\mathcal{M}\ [\![\mathbf{inst}]\!]\ \mathit{env}\ <\mathit{rst},\mathit{return{-}info}>\ \mathit{stg} = <\mathit{rst},\mathit{return{-}info}>\ \mathit{val}\ \mathit{env'}\ \mathit{stg'}.$$

2. For

$$\begin{aligned}\mathbf{inst}{=}\ & \mathbf{s{-}stg}{:}\mathit{stg'}\\ & \quad \mathbf{s{-}env}{:}\mathit{env'}\\ & \quad \mathbf{s{-}c}{:}\mathit{cont'},\end{aligned}$$

in which *cont'* is the label value for a goto, the equation scheme is

$$\mathcal{M}\ [\![\mathbf{inst}]\!]\ \mathit{env}\ <\mathit{rst},\mathit{return{-}info}>\ \mathit{stg} = \mathit{cont'}\ \Omega\ \mathit{env'}\ \mathit{stg'}.$$

3. For

$$\begin{aligned}\mathbf{inst}{=}\ & \mathbf{rest{-}of{-}inst}(\cdots \mathit{ri} \cdots);\\ & \qquad \mathit{ri}{:}\mathbf{first{-}part{-}of{-}inst},\end{aligned}$$

the equation scheme is

$$\begin{aligned}&\mathcal{M}\ [\![\mathbf{inst}]\!]\ \mathit{env}\ <\mathit{rst},\mathit{return{-}info}>\ \mathit{stg} =\\ &\qquad \mathcal{M}\ [\![\mathbf{first{-}part{-}of{-}inst}]\!]\ \mathit{env}\ \mathit{cont'}\ \mathit{stg}\end{aligned}$$

where

$$\begin{aligned}&\mathit{cont'}\ \mathit{val'}\ \mathit{env'}\ \mathit{stg'} =\\ &\qquad \mathcal{M}\ [\![\mathbf{rest{-}of{-}inst}(\cdots \mathit{val'} \cdots)]\!]\\ &\qquad\qquad \mathit{env'}\ <\mathit{rst},\mathit{return{-}info}>\ \mathit{stg'}.\end{aligned}$$

This informal development has been treating control trees and rests of control trees plus *return−info*s as functions. Actually, they are not, but each can certainly be considered as an encoding of a function which takes information about the state yielded by the execution of an instruction and produces the final result of the ensuing computation. These functions are nothing more than the *continuations* of continuation semantics. It is clear that a continuation requires a value, an environment, and a storage as its arguments. So, define a domain **Cont** with ק as a typical element as*

$$\text{ק} \in \mathbf{Cont} = \text{is-v}\hat{\text{a}}\text{lue} \times \text{is-}\hat{\text{e}}\text{nv} \times \text{is-}\hat{\text{s}}\text{tg} \rightarrow \text{is-s}\hat{\text{t}}\text{ate} + \{\perp\}.$$

However, in the case of EPL as mentioned, the only part of a final state which is not Ω is the storage. Thus, the above could be simplified as

$$\text{ק} \in \mathbf{Cont} = \text{is-v}\hat{\text{a}}\text{lue} \times \text{is-}\hat{\text{e}}\text{nv} \times \text{is-}\hat{\text{s}}\text{tg} \rightarrow \text{is-}\hat{\text{s}}\text{tg} + \{\perp\}.$$

By denotational semantics convention,

$\text{is-}\hat{\text{e}}\text{nv} = \mathbf{U},$
$\text{is-}\hat{\text{d}}\text{n} =$ the domain of storable values $= \mathbf{V},$
$\text{is-}\hat{\text{s}}\text{tg} = \mathbf{S},$
$\text{is-}\hat{\text{s}}\text{tg} + \{\perp\} =$ the domain of answers $= \mathbf{A} = \mathbf{S} + \{\perp\}.$

Thus,

$$\text{ק} \in \mathbf{Cont} = \mathbf{V} \times \mathbf{U} \times \mathbf{S} \rightarrow \mathbf{A}$$

or in curried form

$$\text{ק} \in \mathbf{Cont} = \mathbf{V} \rightarrow \mathbf{U} \rightarrow \mathbf{S} \rightarrow \mathbf{A}.$$

It is now possible to use these continuations in place of the <*rst*,*return−info*>s they represent in the above equation schemes. One obtains a meaning function

$$\mathcal{M} \in \mathbf{Inst} \times \mathbf{U} \times \mathbf{Cont} \times \mathbf{S} \rightarrow \mathbf{A},$$

or in curried form

$$\mathcal{M} \in \mathbf{Inst} \rightarrow \mathbf{U} \rightarrow \mathbf{Cont} \rightarrow \mathbf{S} \rightarrow \mathbf{A}.$$

Corresponding to the three equation schemes above one obtains

1. $\mathcal{M}\ \llbracket \mathbf{inst} \rrbracket\ env\ \text{ק}\ stg = \text{ק}\ val\ env'\ stg'$

2. $\mathcal{M}\ \llbracket \mathbf{inst} \rrbracket\ env\ \text{ק}\ stg = \text{ק}'\ \Omega\ env'\ stg'$

3. $\mathcal{M}\ \llbracket \mathbf{inst} \rrbracket\ env\ \text{ק}\ stg = \mathcal{M}\ \llbracket \mathbf{first{-}part{-}of{-}inst} \rrbracket\ env\ \text{ק}'\ stg$

   where $\text{ק}'\ val'\ env'\ stg' = \mathcal{M}\ \llbracket \mathbf{rest{-}of{-}inst(}\cdots val' \cdots\mathbf{)} \rrbracket\ env'\ \text{ק}\ stg'$.

---

*The Hebrew letter Quf, ק, is taken as the letter to denote instruction continuations here so as not to confuse it with the more specific command and expression continuations which have their own naming conventions.

In (2) above, p*′* is the label value of a goto.

Appendix IID contains the denotational continuation semantic definition of EPL+ constructed from the VDL definition of EPL+ of Appendix II.

The previous paragraphs have effectively shown how to systematically construct a denotational continuation semantic definition from a VDL semantic definition of the form described in Section 2.2. The constructed definition is such that

1. the parameters of the statement continuation are precisely the top-level state components other than the control tree, and

2. there is one equation for each instruction (action) of the three instruction forms mentioned above.

2.5 Value-Returning Instructions - Passing Value Back

There are two possibilities as to the nature of **inst** as it is executed. It may be either value-returning or macro. In the case of a value-returning instruction, one possibility is that it is of the form (as for expression evaluation with side effects):

inst=
PASS:*val*
s−stg:*stg′*
s−env:*env′*

Then *env′* and *stg′* are the result of executing **inst** in *env* and *stg* and are in general different from *env* and *stg*. Additionally, *cont′* is related to *cont* in the following way: Let *rst* be the rest of *cont*, i.e., *cont* after the terminal node is removed. Then *cont′* is *rst* as modified by the passed-up value, *val*, returned by the instruction. Recall that part of a node is information stating to which actual parameters of which instructions higher up the tree the PASSed value should be copied.

Thus, if *cont* is considered as composed of (*return−info*,**inst**);*rst*, i.e., *cont* appears as in figure A.

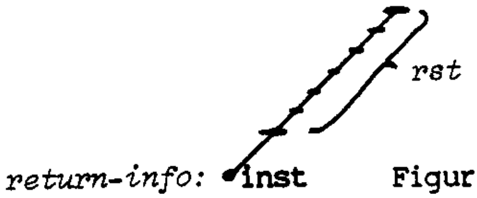


Figure A

and execution of **inst** in *env* and *stg* yields *env′*, *stg′*, and *val* as the new environment, new storage, and PASSed value respectively. Then,

cont(*env*,*stg*) = execute **inst** in *env* and
stg followed by doing
rst with the help of *return−info*

= *rst*−as−modified−by−*val*−according−
to−*return−info*(*env′*,*stg′*) (e1)

= *cont′*(*env′*,*stg′*).

All elements of the above equation yield either the same final state or ⊥ as the case may be.

The middle two elements of the above equation e1 may be used to define the meaning of an instruction **inst** as a function of the current environment, *env*, the current storage, *stg*, and what is to be done after the instruction, *rst*-as-assisted-by-*return−info*. That is,

meaning of **inst** in *env*, *stg*, *rst*,
and *return−info* =
rst–as–modified–by–*val*–according–to–
return−info(*env*′,*stg*′) (e2)

Examination of the parts of equation e2 shows that only *rst* and *return−info* appear on both sides. Thinking about these two items as a unit shows that it is *rst* and *return−info* combined that tell how to continue from the completion of **inst**. That is this pair, given the value, environment and storage returned by **inst**, tells how continue. Taking *rst* and *return−info* as a unit, e2 can be rewritten as:

meaning of **inst** in *env*, *stg* and
<*rst*,*return−info*> =
<*rst*,*return−info*>(*val*)(*env*′,*stg*′), (e3)

where execution of **inst** in *env* and *stg* yields *val*, *env*′, and *stg*′. Both sides of this equation yield the same final state or $\perp$ as the case may be.

This informal development has been treating control trees and rests of control trees plus *return−info*s as functions. Actually, they are not, but each can certainly be considered as an encoding of a function which takes information about the state yielded by the execution of an instruction and produces the final result of the ensuing computation. These functions are nothing more than the *continuations* of continuation semantics. As e3 is in terms of <rest–of–control–tree, *return−info*> pairs, the continuations encoded by them are considered first. From e3, it is clear that a continuation requires a value, an environment and a storage as its arguments. So, define a domain **Cont** with ρ as a typical element as

$$\rho \in \mathbf{Cont} = \text{is-}\hat{\text{value}} \times \text{is-}\hat{\text{env}} \times \text{is-}\hat{\text{stg}} \rightarrow \text{is-}\hat{\text{state}} + \{\perp\}.$$

However, in the case of EPL as mentioned, the only part of a final state which is not Ω is the storage. Thus, the above could be simplified as

$$\rho \in \mathbf{Cont} = \text{is-}\hat{\text{value}} \times \text{is-}\hat{\text{env}} \times \text{is-}\hat{\text{stg}} \rightarrow \text{is-}\hat{\text{stg}} + \{\perp\}.$$

By denotational semantics convention,

$\text{is-}\hat{\text{env}} = \mathbf{U}$,
$\text{is-}\hat{\text{dn}} =$ the domain of storeable values $= \mathbf{V}$,
$\text{is-}\hat{\text{stg}} = \mathbf{S}$,
$is\text{-}\hat{stg} + \{\perp\} =$ the domain of answers $= \mathbf{A} = \mathbf{S} + \{\perp\}$.

Thus,

$$\rho \in \mathbf{Cont} = \mathbf{V} \times \mathbf{U} \times \mathbf{S} \rightarrow A$$

or in curried form

$$\rho \in \mathbf{Cont} = \mathbf{V} \rightarrow \mathbf{U} \rightarrow \mathbf{S} \rightarrow \mathbf{A}.$$

It is now possible to formalize the notion of a meaning function on instructions, environments, storages and continuations (representable by <rest–of–control–tree,*return−info*> pairs) to answers. Historically, the order of the parameters of the meaning function is instructions, environments, continuations, and storages because it gives a more useful currying. Letting **Inst** be the domain of control tree instructions,

$$\mathit{Mean} \in \mathbf{Inst}\times\mathbf{U}\times\mathbf{Cont}\times\mathbf{S}\to\mathbf{A},$$

or in curried form,

$$\mathit{Mean} \in \mathbf{Inst}\to\mathbf{U}\to\mathbf{Cont}\to\mathbf{S}\to\mathbf{A}.$$

Following e3, the equation defining the meaning of an instruction is:

$$\mathit{Mean}[\![\mathbf{Inst}]\!]\,\mathit{env}\ \rho\ \mathit{stg}=\mathit{ans}=\rho\ \mathit{val}\ \mathit{env}'\ \mathit{stg}' \tag{e3a}$$

where

$\mathbf{Inst} \in \mathbf{Inst}$,
$\mathit{env}, \mathit{env}' \in \mathbf{U}$,
$\rho \in \mathbf{Cont}$,
$\mathit{stg}, \mathit{stg}' \in \mathbf{S}$,
$\mathit{ans} \in \mathbf{A}$, and
$\mathit{val} \in \mathbf{V}$,

and where execution of **Inst** in *env* and *stg* yields *val*, *env'* and *stg'*. The equation may be read, "The meaning of **Inst** in *env* and *stg* in the presence of the continuation ρ is simply the answer obtained by applying the continuation to the *val*, *env'*, and *stg'* yielded by the instruction in *env* and *stg*."

If the instruction of a particular group of instructions corresponding to some syntactic domain always leave the environment or storage unchanged or do not pass up a value, a special continuation for that syntactic domain can be designed in which the unchanged component or the returned value is left out as a parameter of the continuation. For example, in many languages, commands (statements) do not modify environments and return no value, so one might use a command continuation $\theta \in \mathbf{CCont}$, which is on storage only:

$$\theta \in \mathbf{CCont}=\mathbf{S}\to\mathbf{A}.$$

Also in many languages, expressions have no side effects on either the environment or storage. Thus, it is reasonable to designate an expression continuation, $\kappa \in \mathbf{ECont}$, as being on values only:

$$\kappa \in \mathbf{ECont}=\mathbf{V}\to\mathbf{A}.$$

Thus, the command continuation needs only the new storage that a command generates to go to the final answer; it assumes that the environment is unchanged and no value is passed up by the command. Likewise, the expression continuation needs only the returned value to go to the final answer; it assumes that the environment and storage are unmodified by the expression.

It is now possible to identify the function encoded by *cont* and *cont'* in the expressions

$$\mathit{cont}(\mathit{env},\mathit{stg})$$

and

$$\mathit{cont}'(\mathit{env}',\mathit{stg}').$$

The function they encode is a statement continuation, that is a continuation for a construct which modifies the environment and storage but does not return a value. Such a continuation has the functionality

$$\mathbf{U}\to\mathbf{S}\to\mathbf{A}.$$

Define the domain **SCont** with typical element כ:*

כ∈**SCont=U→S→A**.

Note that *cont'* is produced by modifying *rst* by *val* according to *return−info*. That is, applying a general continuation ק∈**Cont** to a value yields a statement continuation, כ. This statement continuation represents the execution after completion of the statement containing the expression returning the value. So to speak, the general continuation swallows the value returned by the expression to make the continuation for the containing statement.

1.6 Value-Returning Instructions - Assigning Control Tree

Consider again the application of a control tree *cont* to *env* and *stg*. Suppose that the instruction **inst** of the terminal node of *cont* is a value-returning instruction of the form (e.g., as for a goto or a back-tracking):

inst=

s−stg:*stg'*
s−env:*env'*
s−c:*cont'*

The rules of VDL are such that if the control tree is replaced in a value-returning instruction then no node containing it can have *return−info*.** In this case *env'* , *stg'* and *cont'* are the result of executing **inst** in *env* and *stg* and are in general different from *env*, *stg*, and *cont*. In this case,

cont(*env*,*stg*) = execute **inst** in *env* and
stg followed by doing
rst with the help of
return−info

= execute **inst** in *env* and
stg thus ignoring
rst and *return−info*
entirely (e4)

= *cont'*(*env'*,*stg'*).

That is, execution of the instruction, **inst**, causes the rest of the control tree, *cont*, on which it was found to be ignored entirely and for the *cont'* to be used to dictate the rest of the computation based on the *env'* and *stg'* returned by the instruction.

The meaning of such an instruction can be exhibited by an equation such as

meaning of **inst** in *env*, *stg*, and
<*rst*,*return−info*> = (e5)
cont'(*env'*,*stg'*).

This may be formalized using the functions introduced earlier. Suppose <*rst*,*return−info*> encodes ק and *cont'* encodes כ. Then,

---

* כ is the Hebrew letter Kaf.

**A replaced control tree combined with the presence of *return−info* gives rise to a μ with two dependent composite selectors.

$$Mean[\![\textbf{inst}]\!]env\ \rho\ stg = \supset env'\ stg' \qquad \text{(e6)}$$

where execution of **inst** in *env* and *stg* yields ⊃, *env'*, and *stg'*.

1.7 Macro Instruction

Recall that *cont*=(*return−info*,**inst**);*rst*, i.e., as shown in figure A. Consider now the application of *cont* to *env* and *stg* when the instruction, **inst**, of the terminal node of *cont* is a macro instruction of the form:

inst=
  rest−of−inst(· · · *ri* · · · **);**
    ri:**first−part−of−inst**

where *ri* may be empty. That is, the application of *cont* to *env* and *stg* is the execution of **inst** which yields the same environment, *env*, the same storage, *stg*, and the control tree, *cont'*=(*ri*,**first−part−of−inst**);(*return−info*,**rest−of−inst(** · · · *ri* · · · **)**);*rst*, as shown in figure B,

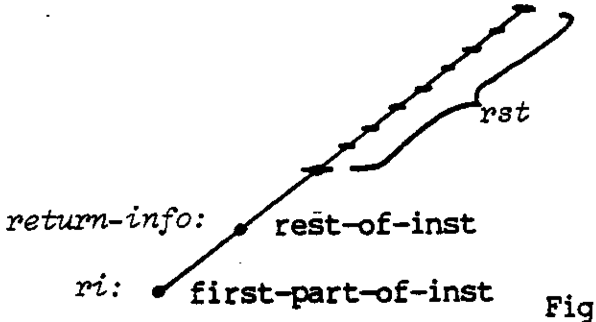


Figure B

followed by application of *cont'* to *env* and *stg*. Equationally,

cont(*env*,*stg*) = execute **inst** in *env* and
  stg followed by doing
  rst with the help of
  return−info (e7)

  = *cont'*(*env'*,*stg'*).

However, application of *cont'* to *env* and *stg* is the execution of **first−part−of−inst** in *env* and *stg* followed by doing **rest−of−inst(** · · · *ri* · · · **)** with the help of *ri* followed by doing *rst* with the help of *return−info*. Thus,

execute **inst** in *env* and *stg* followed by doing
  rst with the help of *return−info*
 = *cont'*(*env*,*stg*)
 = execute **first−part−of−inst** in *env* and *stg* (e8)
  followed by doing **rest−of−inst(** · · · *ri* · · · **)** with
  the help of *ri* followed by doing
  rst with the help of *return−info*.

In other words, when execution of **inst** requires execution of **first−part−of−inst** followed by doing **rest−of−inst(** · · · *ri* · · · **)** with the help of *ri*, then execution of **inst** followed by doing *rst* with the help of *return−info* is execution of **first−part−of−inst** followed by doing **rest−of−inst(** · · · *ri* · · · **)** with the help of *ri* followed by doing *rst* with the help of *return−info*. In such a case, the meaning of **inst** is defined in terms of the meaning of **first−part−of−inst**.

meaning of **inst** in *env*, *stg*, and
<*rst*,*return*−*info*> = (e9)
meaning of **first−part−of−inst** in *env*, *stg*,
and <**rest−of−inst**(· · · *ri* · · ·);<*rst*,*return*−*info*>,*ri*>

Letting <*rst*,*return*−*info*> be an encoding of ρ, e9 can be formalized as

Mean⟦**inst**⟧*env* ρ *stg* = (e10)
Mean⟦**first−part−of−inst**⟧*env* ρ' *stg*

where

ρ'(*val'*,*env'*,*stg'*)=
Let *cont''*=**rest−of−inst**(· · · *ri* · · ·)−modified−by−
val'−according−to−*ri* **in**
apply *cont''* to *env'* and *stg'* to yield
val'', *env''*, and *stg''* which are
passed as parameters to ρ

That is, ρ',

1. first, given the *val'*, *env'*, and *stg'* returned by **first−part−of−inst**, does the effect of **rest−of−inst**(· · · *ri* · · ·) to obtain *val''*, *env''*, and *stg''*,

2. second, passes *val''*, *env''* and *stg''* to ρ to obtain the final answer.

Note that

rest−of−inst(· · · *ri* · · ·)−modified-by-*val'*-according-to-*ri*

is nothing more than **rest−of−inst**(· · · *val'* · · ·).

The meaning equation, e10, for **inst** reflects **inst**'s macro nature in that the macro arranges for the execution of an expanded list of instructions, starting with **first−part−of−inst**, to achieve its effect rather than directly modifying the environment and storage by itself.

3. Formalization of Construction of Continuation from Deterministic Control Tree

All of the development of Section 2 may be formalized using the concepts developed in the formal definition of the VDL programming language as given in [LLS70] and in [LW69]. Using the notation of the former, this Section shows the construction from a deterministic control tree to the continuation it is an encoding of.

First it is necessary to outline the formal definition of the VDL programming language, adapting it to fit the restricted states and control trees used in the informal development of the previous Section.

3.1 Control Trees in General

First recall that for any state ξ ,

is−c(s−c(ξ))

and that

$$\text{is-c}=\text{is-ct}\vee\text{is-}\Omega.$$

That is, the s-c component of any state either is a control tree or is Ω . A state ξ such that $\text{s-c}(\xi)=\Omega$ is a final state.

A control tree in general is described by

$$\begin{array}{ll}
\text{is-ct} = & (\langle\text{s-in:is-in}\rangle, \\
 & \langle\text{s-al:}(\{\langle\text{elem(i):is-ob}\rangle \,\|\, \\
 & \qquad \text{is-intg(i)}\})\rangle, \\
 & \langle\text{s-ri:}(\langle\text{s-comp:is-sel}\rangle, \\
 & \qquad \langle\text{s-ap:is-ip-set}\rangle)\rangle, \\
 & \{\langle\text{r:is-ct}\rangle \,\|\, \text{r} \in \text{R}\}
\end{array}$$

$$\begin{array}{ll}
\text{where:} & \text{is-}\widehat{\text{intg}}=\{1,2,\cdots\} \\
 & \text{is-}\widehat{\text{in}} = \text{set of instruction names} \\
 & \text{is-}\widehat{\text{sel}} = \text{set of all selectors, i.e., } \text{is-}\widehat{\text{sel}} = S^{*} \\
 & \text{is-}\widehat{\text{ob}} = \text{set of all objects} \\
 & \text{is-ip}=(\langle\text{elem(1):is-intg}\rangle,\langle\text{elem(2):is-intg}\rangle) \\
 & \text{R is an infinite set of simple selectors such that} \\
 & \qquad \text{R}\cap\{\text{s-in},\text{s-al},\text{s-ri},\text{s-comp},\text{s-ap}\}=\varnothing
\end{array}$$

Note that the s-al (argument list) component of a control tree may not be a proper list, because some of the indices less than the maximum may select Ω .

For convenience, the predicate is-ri is introduced to denote the set of *return-infos*

$$\text{is-ri}=(\langle\text{s-comp:is-sel}\rangle,\langle\text{s-ap:is-ip-set}\rangle)$$

A few useful functions are nd, yielding for a control tree the set of composite selectors selecting the nodes of the tree,

$$\text{nd}(ct) = \{\chi \mid \chi\in \text{R}^{*} \;\&\; \chi(ct)\neq\Omega\},$$

pred, yielding for a composite selector selecting a node, the composite selector selecting one node higher up the tree,

$$\text{pred}(\chi)=(\iota\chi_1)((\exists s\in\mathcal{S})(s\circ\chi_1=\chi)),$$

and pred^{n} , the obvious extension of pred:

$$\begin{array}{ll}
\text{pred}^{n}(\chi) = & \\
 & n=0\rightarrow\chi \\
 & \text{T}\rightarrow\text{pred}^{n-1}(\text{pred}(\chi))
\end{array}$$

3.2 Deterministic Control Trees

Now as a result of the restrictions to the deterministic case, it is known that if $\text{is-ct}(ct)$ then for at most one selector $r\in\text{R}$, $r(ct)\neq\Omega$.

3.2.1 Control Tree Representations

What appears in the VDL programming language wherever a control tree or sub-control tree is needed is a two-dimensional (i.e. indentation as well as the text itself counts) representation of the control tree object as defined above. Because of the determinism assumption, a control tree representation, *ct−rep*, has one of the following forms:

1. *instr*

2. *instr*;
   succ.

An *instr* has one of the forms

1. **In**

2. $\mathbf{In}(expr_1, \ldots, expr_n)$,

and *succ* has one of the forms

1. *ct−rep*

2. *dum*:*ct−rep*

3. *expr*(*dum*):*ct−rep*.

To obtain from a control-tree representation the control tree it represents; first the dummy names, *dum*, are eliminated by the following two rules:

1. Each occurrence of a dummy name, *dum*, as the prefix of an instruction (as in the 2nd and 3rd forms of a *succ*) is replaced by a set of integer pairs: $<i,j>$ is in this set if and only if the same *dum* appears as the jth actual parameter of the ith predecessor node.

2. All uses of dummy names in actual parameter positions are replaced by Ω.

Then the control tree represented by a given control tree representation is determined recursively by Table 1 [LLS70] given below.

3.2.2 Instruction Schemata

The instruction schemata (i.e., instruction definition in the VDL programming language) associated with an instruction name **In** has, in general, the following form:

$$
\begin{aligned}
&\mathbf{In}(x_1, \ldots, x_n) = \\
&\qquad p_1(x_1,\ldots,x_n,\xi) \rightarrow group_1(x_1,\ldots,x_n,\xi) \\
&\qquad \cdot \\
&\qquad \cdot \\
&\qquad \cdot \\
&\qquad p_m(x_1,\ldots,x_n,\xi) \rightarrow group_m(x_1,\ldots,x_n,\xi)
\end{aligned}
$$

where each p_i is a meta expression denoting a truth value and each $group_i$ has one of the two forms:

Category	Form	Represented Control Tree
ct−rep	*instr*	*instr*
	instr; *succ*	μ(*instr*;{<r:*succ*>}>) where r ∈ R
instr	in	μ_o(<s–in:in>)
	in($expr_1, \ldots, expr_n$)	μ_o(<s–in:in>,<s–al: μ_o(<elem(1):$expr_1$>, . . . , <elem(n):$expr_n$>)>)
succ	*ct−rep*	μ(*ct−rep*;<s–ri: μ_o(<s–comp:*I*>,<s–ap:{}>)>)
	*dum**:*ct−rep*	μ(*ct−rep*;<s–ri: μ_o(<s–comp:*I*>,<s–ap:*dum**>)>)
	expr(*dum**):*ct−rep*	μ(*ct−rep*;<s–ri: μ_o(<s–comp:*expr*>,<s–ap:*dum**>)>)

where: *dum** is the set of ordered pairs which has replaced *dum*.

Table 1

1. value returning:

   PASS:$\epsilon_v(x_1, \ldots, x_n, \xi)$
   s−env:$\epsilon_\rho(x_1, \ldots, x_n, \xi)$
   s−stg:$\epsilon_\sigma(x_1, \ldots, x_n, \xi)$
   s−c:$\epsilon_p(x_1, \ldots, x_n, \xi)$

   where each ϵ_i is an arbitrary expression; any of the lines may be missing; a missing first (PASS) line is taken as equivalent to PASS:Ω .

2. macro:

   ct−rep$(x_1, \ldots, x_n, \xi)$

   a control tree representation.

It is now possible to define a function cont which extracts the continuation encoded by a control tree and return-information pair.

Each instruction schema can be considered a shorthand for a function,

$$\Delta_{in}\text{:is-}\hat{\text{ob}}^n \times \text{is-}\hat{\text{state}} \times \text{is-}\hat{\text{sel}} \times \text{is-}\hat{\text{ri}} \rightarrow \text{is-}\hat{\text{state}} \times \text{is-}\hat{\text{ob}},$$

$$\Delta_{in}(x_1, \ldots, x_n, \xi, \tau, ri) =$$
$$p_1(x_1, \ldots, x_n, \xi) \rightarrow group_1^+(x_1, \ldots, x_n, \xi, \tau, ri)$$
$$\vdots$$

$$p_m(x_1,\ldots,x_n,\xi) \to group^{+}_{m}(x_1,\ldots,x_n,\xi,\tau,ri)$$

where $group^{+}_{i}$ is obtained from $group_i$ according to the latter's form: If $group_i$ is

1. value returning then $group^{+}_{i}$ is

$$\begin{aligned}
&\langle \mu(\mu(\xi;\{<(\mathrm{s\text{-}comp}(ri)) \circ elem(j)) \circ \mathrm{s\text{-}al}(\mathrm{pred}^{i}(\tau)) \circ \mathrm{s\text{-}c}:\\
&\qquad \epsilon_v(x_1,\;.\;.\;.\;,x_n,\xi)> \mid <i,j> \in \mathrm{s\text{-}ap}(ri)\});\\
&\quad <\mathrm{s\text{-}env}:\epsilon_\rho(x_1,\;.\;.\;.\;,x_n,\xi)>,\\
&\quad <\mathrm{s\text{-}stg}:\epsilon_\sigma(x_1,\;.\;.\;.\;,x_n,\xi)>,\\
&\quad <\mathrm{s\text{-}c}:\epsilon_{p}(x_1,\;.\;.\;.\;,x_n,\xi)>),\\
&\epsilon_v(x_1,\;.\;.\;.\;,x_n,\xi)\rangle
\end{aligned}$$

2. macro, then $group^{+}_{i}$ is

$$\begin{aligned}
&\langle \mu(\xi;<\tau \circ \mathrm{s\text{-}c}:\mu(ct\text{-}rep(x_1,\;.\;.\;.\;,x_n,\xi);<\mathrm{s\text{-}ri}:ri>)>),\\
&\Omega\rangle
\end{aligned}$$

That is Δ_{in} returns both the next state, *à la* Φ_{in} defined in [LW69], and the passed up value.

Using this Δ_{in}, it is possible to write a recursive function, cont, which given a deterministic control tree *ct* (with therefore a unique terminal node) and return information *ri*, stating where in *ct* a returned value is to be passed, constructs the continuation encoded by them. That is:

$$\mathrm{cont{:}is\text{-}\hat{c}t \times is\text{-}\hat{r}i \to \mathbf{Cont}}$$

The continuation yielded by $\mathrm{cont}(ct,ri)$ is of type

$$\mathbf{V \times E \times S \to A}\ .$$

Figure 4 is useful for understanding the definition of cont.

Definition 1. Suppose $\mathrm{is\text{-}c}(ct)$ and $\mathrm{is\text{-}ri}(ri)$.
Then,

$$\begin{aligned}
\mathrm{cont}(ct,ri) = {} & \lambda(\epsilon,\rho,\sigma).\\
& (is-\Omega(ct) \to \sigma,\\
& T \to \mathrm{cont}(ct',nri)(\epsilon',\rho',\sigma'))
\end{aligned}$$

where

$$\begin{aligned}
&\mathrm{tn}(ct) = (\iota\tau)(\tau \in \mathrm{nd}(ct)\ \&\ (\forall\chi)(\chi \in \mathrm{nd}(ct) \supset \mathrm{pred}(\chi) \neq \tau)),\\
&\tau = \mathrm{tn}(ct),\\
&ct\dagger = \mu(ct;\{<(\mathrm{s\text{-}comp}(ri)) \circ elem(j) \circ \mathrm{s\text{-}al}(\mathrm{pred}^{i-1}(\tau)):\epsilon> \mid <i,j> \in \mathrm{s\text{-}ap}(ri)\}),
\end{aligned}$$

$node = \tau(ct\uparrow)$,
in = s–in($node$),
al = s–al($node$),
nri = s–ri($node$),
n = number of formal parameters in schema for s–in($node$), and
$\langle\mu_o(<\text{s–env}:\rho'>,<\text{s–stg}:\sigma'>,<\text{s–c}:ct'>),\epsilon'\rangle =$
$\quad\Delta_{\text{in}}(elem(1,al),\ ...\ ,elem(n,al),$
$\qquad\mu_o(<\text{s–env}:\rho>,<\text{s–stg}:\sigma>,<\text{s–c}:\delta(ct\uparrow,\tau)>),\tau,nri)$

In the above, tn identifies the selector τ selecting the terminal node of ct. Also, recall that $\delta(O,\chi)$ deletes the object at $\chi(O)$ from O.

4. From Nondeterministic Control Trees to Parallel Continuations (Informal Discussion)

The ultimate goal of this Section is to obtain a parallel continuation which can be used in equations to define the meanings of syntactic constructs in a denotational, syntax-directed manner. Section 3 started with deterministic control trees and obtained by a particular construction continuations and equations such that the control trees and continuations could be used interchangeably in the equations. Thus, this Section starts with nondeterministic control trees and follows the same construction in an attempt to obtain parallel continuations and equations such that the control trees and the continuations can be used interchangeably in these equations. That is, this Section takes the paradigm of applying the nondeterministic control tree to the rest of the state. The results are a parallel continuation and a rather messy equation. In the hopes that the control trees and the continuations can be used in the equations, the messiness of the equations is tolerated. It turns out that only the control trees may be used in the equations. Examination of the equations shows why continuations cannot be used in them and suggests a cleaner formulation of the equations suitable for use with the control trees only. It does however turn out that from any control tree used in either of the equation forms, one can obtain the parallel continuation that it encodes.

4.1 Unmodifications and New Modifications to EPL+ Machine

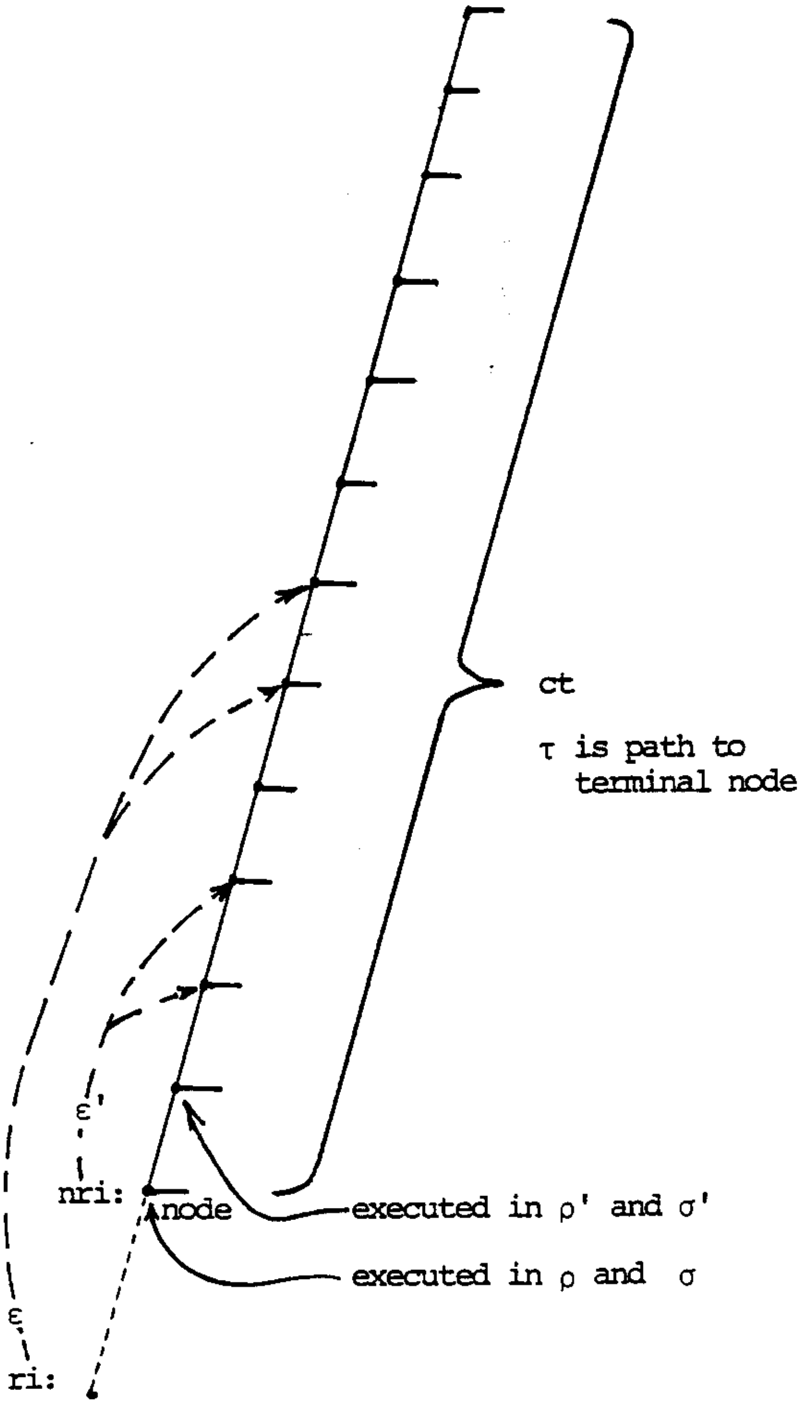
First undo the changes to the EPL+ machine of Section 2.2 which made the control tree strictly deterministic. That is, the commas are re-introduced and the macros which had originally expanded into sets of instructions are put back into that form. At this point, a typical state looks as shown in Figure 5. The control tree may have several terminal nodes. At any given state, any terminal node may be selected for execution -- giving rise to nondeterminism in the computation sequence.

If EPL+ were now extended to permit parallel blocks, e.g.,

par s_1; · · · ;s_n **end**,

a sequence of statements which are executed asynchronously (i.e., possibly completely interleaved parallelism) some additional changes to the state and the instruction definitions must be made. Since the execution of each of these statements may independently enter blocks, procedures, and functions and may do gotos, each must be able to operated in its own environment rather than from a single global shared environment. They must, however, all share the same global storage, otherwise the nondeterminism will not make itself felt through possible nondeterminate results in a shared storage.

Thus, the environment must be removed from the top level of the state and made a parameter of each instruction which itself can access identifiers or which expands to at least one instruction which can access identifiers. As a result, a typical state ends up appearing as in Figure 6. Observe that it is no longer necessary for exit to have an environment as a parameter since there is no global environment to restore. Observe also that a label value is simply a control tree, as its environment is found directly in the instructions to execute after the goto.



$\rho\ \mathcal{G}\ \text{is-}\hat{\text{e}}\text{nv} = \mathbf{U}$
$\rho'\mathcal{G}\ \text{is-}\hat{\text{e}}\text{nv} = \mathbf{U}$
$\sigma\ \mathcal{G}\ \text{is-}\hat{\text{s}}\text{tg} = \mathbf{S}$
$\sigma'\mathcal{G}\ \text{is-}\hat{\text{s}}\text{tg} = \mathbf{S}$
$\varepsilon\ \mathcal{G}\ \text{is-}\hat{\text{d}}\text{en} = \mathbf{V}$
$\varepsilon'\mathcal{G}\ \text{is-}\hat{\text{d}}\text{en} = \mathbf{V}$

Figure 4

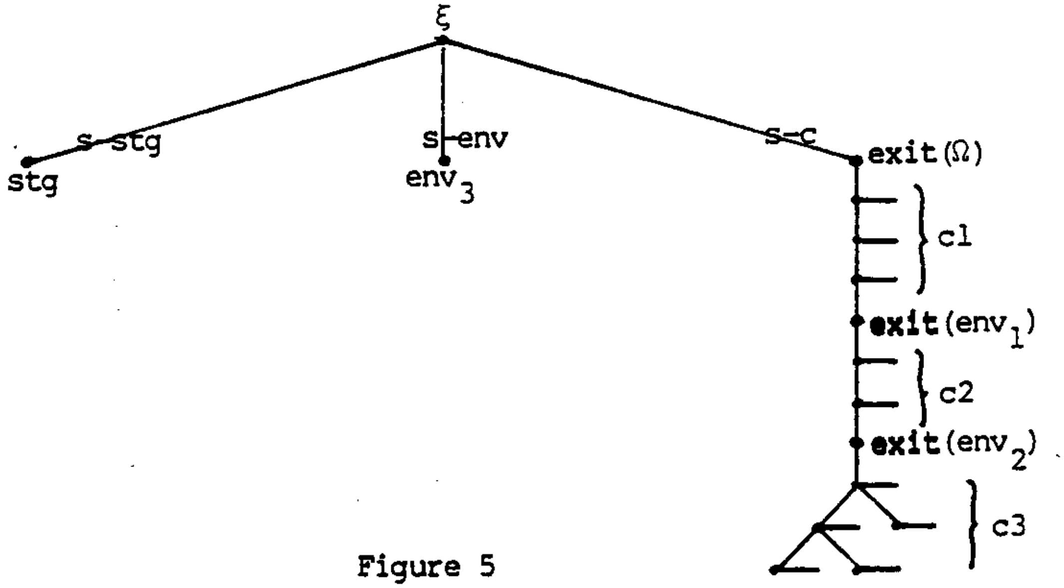


Figure 5

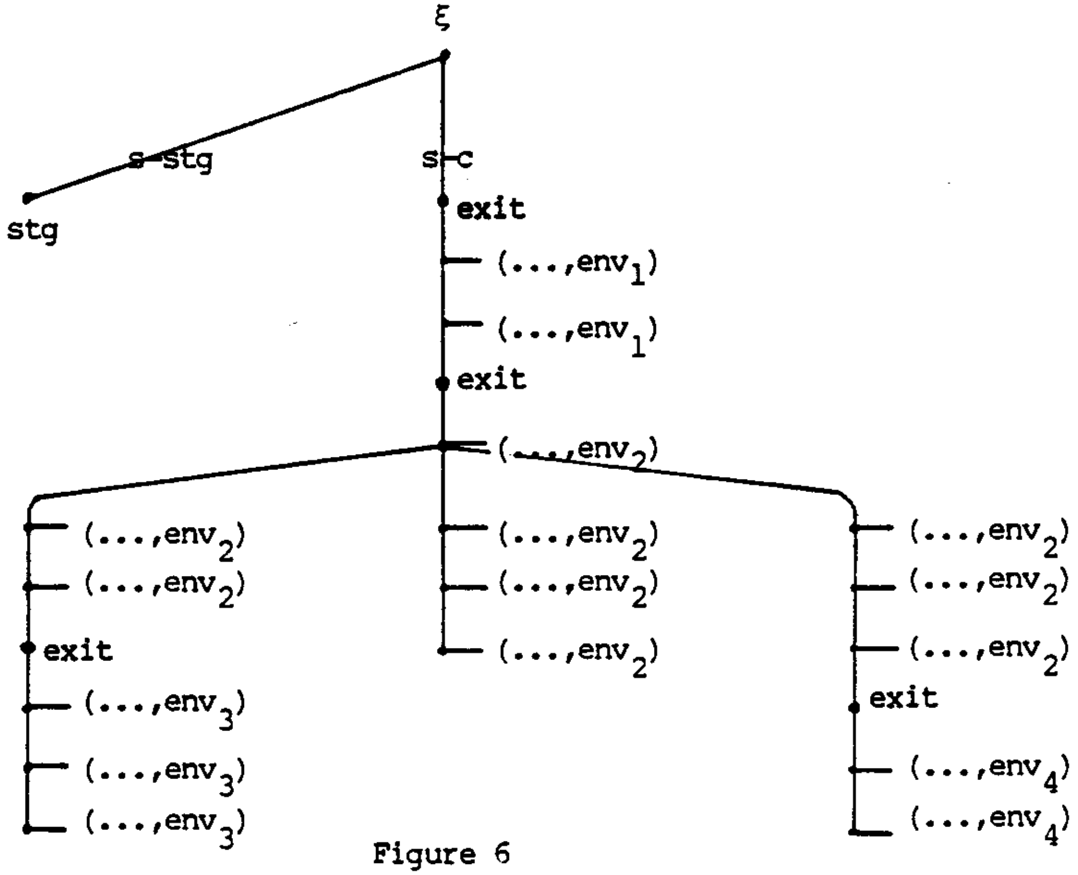


Figure 6

Appendix III contains a definition of EPL+ extended with a parallel block.

4.2 Nondeterministic Control Trees

A nondeterministic control tree, *cont*, has a set of terminal nodes. At any step in a computation, some one of these nodes is selected. Once a node is selected, the rest of the tree, *rst*, is determined, i.e., the original tree with the selected node removed. In order to know how to pass values up or to where to attach macro subtrees it is necessary to remember along with *rst* the composite selector χ selecting the removed node as well as the return information. This is illustrated in Figure 7.

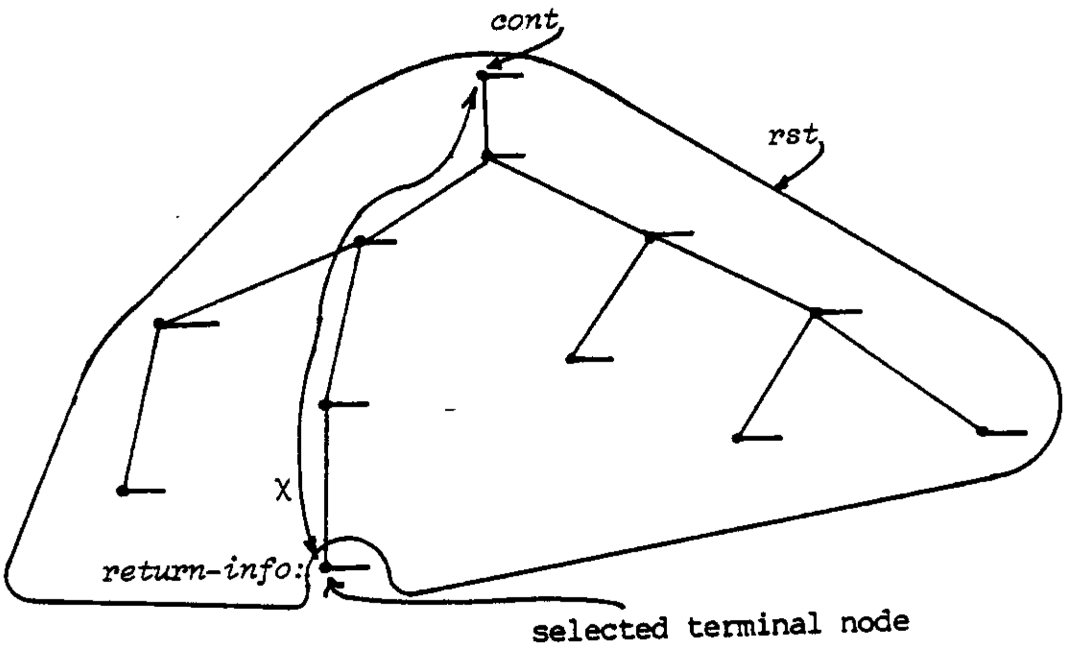


Figure 7

Just as in the deterministic case *rst* and *return−info* were identified as a unit, so are *rst*, χ, and *return−info* identified as a unit in the nondeterministic case. It is convenient to call the *rst*, χ and *return−info* associated with a node η of *cont* the *p−rest* of *cont* with respect to η and to call triples of *rsts*, χs, and *return−infos* *p−rests**. If one observes that in the deterministic case, the χ of a terminal node is uniquely determined then it is clear that, in fact, the $<rst, return-info>$ pair for a deterministic control tree conveys the same information as the $<rst, \chi, return-info>$ triple for a nondeterministic control tree.

4.3 Computations and Λ

To execute from a state ξ with control tree *cont* until the end, starting with this particular selection of terminal node, it is necessary to execute the instruction at the selected terminal node. When it or its expansion is completed then *rst*, possibly modified by a returned value according to *return−info* is executed until the end.

---

*"p" for parallel.

The *rst* itself in general has a set of terminal nodes, some of which may be terminal nodes in *cont* which were not selected for execution. As a consequence, for each terminal node of *cont*, there is a different beginning of the remainder of the computation, and for each such remainder, for each terminal node of the possibly modified *rst* there is a different beginning of the remainder of the remainder of the computation. Thus, from a given control tree *cont*, there is a tree of possible computations, as illustrated in Figure 8.

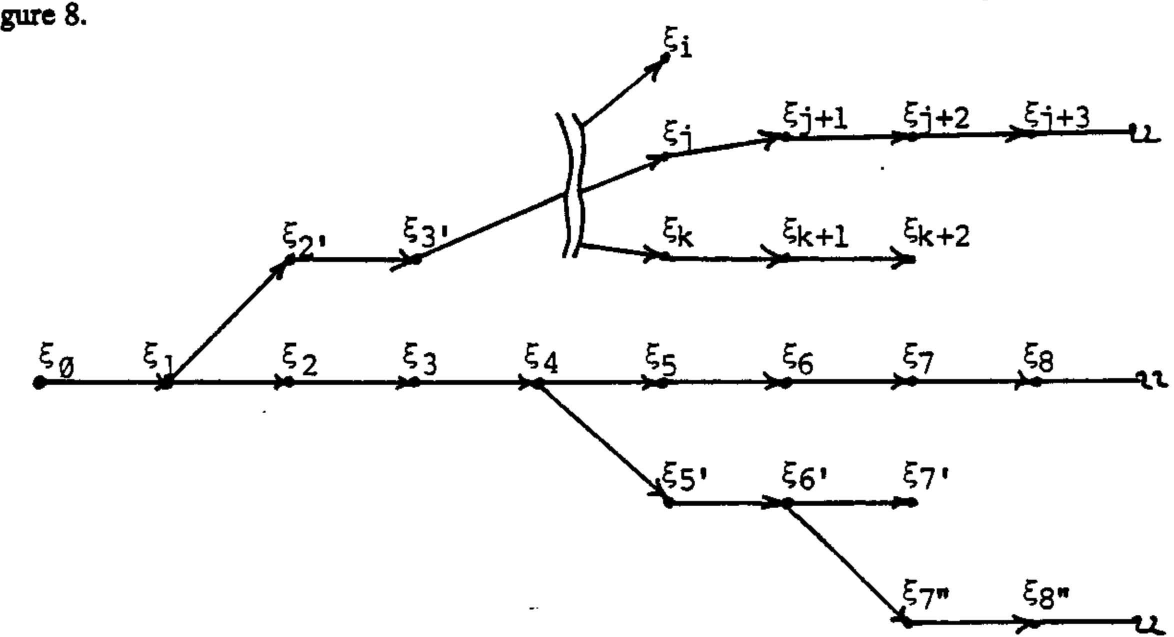


Figure 8

Each node of the tree is a state and each path, possibly infinite, is the computation comprised, in sequence, of the states lying on it.

The usual way to fit nondeterministic computations into the information structure model framework is to let the Λ of the triple,

(is-state,is-initial-state,Λ),

return for a state a set of states.

Λ:is-state→*P*(is-state).

A transition from state ξ is accomplished by selecting any element of Λ(ξ). If Λ(ξ) returns the empty set, the computation is said to halt at ξ.

By analogy to the discussion of Section 2, the construction of *f* from Λ may be viewed as giving for a particular initial state the set of answers obtainable from all computations of that initial state, i.e., the set of all final states and ⊥ if any computation is nonterminating:

f:is-state→*P*(is-state+{⊥})

From this discussion it is clear that application of a control tree to the remainder of the state, i.e., the storage, should be viewed as yielding a set of answers and that a nondeterministic control tree is an encoding either of a set of classical continuations each yielding an answer or of a *parallel continuation* yielding a set of answers.

4.4 Application of Nondeterministic Control Trees

Let *cont* be a nondeterministic control tree and let *stg* be storage. Then application of *cont* to *stg* gives rise to a set of answers. A subset of these answers is obtained by selecting one of the terminal nodes of *cont*, say η, executing its instruction **inst** in *stg* and its $p-rest$ to yield new storage stg' and a new control tree $cont'$, and then applying $cont'$ to stg'. The application of $cont'$ to stg' itself yields the required set of answers.

To be more precise, suppose $\{\eta_1, \ldots, \eta_n\}$ is the set of terminal nodes of *cont*. Suppose that the $p-rest$ of *cont* with respect to η_i is $<rst_i,\chi_i,return-info_i>$. Suppose that the instruction of η_i is $\mathbf{inst}_i$. Then,

$$\begin{aligned}cont(stg)=\bigcup\{&\text{ execute } \mathbf{inst}_i \text{ in } stg \text{ and } <rst_i,\chi_i,return-info_i> \text{ followed}\\ &\text{by doing } rst_i \text{ with the help of } \chi_i \text{ and } return-info_i | 1\le i\le n\}.\end{aligned} \tag{E1}$$

Note, that each of the "execute $\mathbf{inst}_i$ ..."s itself returns a set of answers so that it is necessary to take the union of these sets to get the desired set of answers for $cont(stg)$.

Suppose, now, that execution of $\mathbf{inst}_i$ in *stg* and $<rst_i,\chi_i,return-info_i>$ yields stg_i' and $cont_i'$. There are three ways in which the instruction may operate:

1. If $\mathbf{inst}_i$ is value-returning and $return-info_i$ is non empty, then stg_i' is that explicitly yielded by the instruction and $cont_i'$ is rst_i as modified by passing up a value from the arc selected by χ_i according to $return-info_i$.

2. If $\mathbf{inst}_i$ is value-returning and it calculates a new control tree, then $cont_i'$ that control tree and stg_i' is that explicitly yielded by the instruction.

3. If $\mathbf{inst}_i$ is macro then stg_i' is *stg* and $cont_i'$ is obtained by hanging the macro subtree off the χ_i arc of rst_i leaving $return-info_i$ at the node selected by χ_i.

However $\mathbf{inst}_i$ operates, the set of answers yielded by "execution of $\mathbf{inst}_i$ ..." is that yielded by $cont_i'(stg_i')$. Stated equationally,

$$\begin{aligned}cont(stg)=\bigcup\{&\text{ execute } \mathbf{inst}_i \text{ in } stg \text{ and } <rst_i,\chi_i,return-info_i> \text{ followed by}\\ &\qquad\text{doing } rst_i \text{ with the help of } \chi_i \text{ and } return-info_i | 1\le i\le n\}\\ =\bigcup\{&cont_i'(stg_i') | 1\le i\le n\}.\end{aligned} \tag{E2}$$

Following the pattern of the deterministic case, a meaning equation is built out of the last two elements of E2 rewritten as:

$$\begin{aligned}\bigcup\{&\text{ execute } \mathbf{inst}_i \text{ in } stg \text{ and } <rst_i,\chi_i,return-info_i> \text{ followed by doing}\\ &\quad rst_i \text{ with the help of } return-info_i \text{ and } \chi_i\\ &\qquad | \exists \text{ a terminal node } \eta_i \text{ of } cont \text{ such that } \mathbf{inst}_i \text{ is the instruction of } \eta_i\\ &\qquad \text{and } <rst_i,\chi_i,return-info_i> \text{ is the } p-rest \text{ of } cont \text{ w.r.t. } \eta_i\}\\ = \bigcup\{&cont_i'(stg_i') | \exists \text{ a terminal node } \eta_i \text{ of } cont \text{ such that } \mathbf{inst}_i \text{ is the}\\ &\qquad \text{instruction of } \eta_i, <rst_i,\chi_i,return-info_i> \text{ is the } p-rest \text{ of } cont\\ &\qquad \text{w.r.t. } \eta_i, \text{ and execution of } \mathbf{inst}_i \text{ in } stg \text{ and } <rst_i,\chi_i,return-info_i>\\ &\qquad \text{yields } cont_i' \text{ and } stg_i'\}.\end{aligned} \tag{E3}$$

4.5 Meaning

The concern, then, is with the meaning of a *set* of instructions (from the terminal nodes of a control tree *cont*) in *stg* and a corresponding *set* of *p−rests* (of *cont* with respect to the terminal nodes). Parts of the previous sentences are parenthesized because a meaning should be defined for any set of instructions and any set of the same size of *p−rests*. However, the meaning makes sense only when the instructions are all from nodes of a single control tree, and the *p−rests* are all of that control tree with respect to the same nodes.

To keep the correspondence between elements of the two sets, it is convenient to make the two sets sequences of the same length; the ith instruction is executed in *stg* and the ith *p−rest*.

On this basis, it is possible to introduce a meaning function with functionality.*

$$\mathcal{M} \in \mathbf{Inst}^* \times (\text{is-}\hat{\text{ct}} \times \text{is-}\hat{\text{sel}} \times \text{is-}\hat{\text{ri}})^* \times \text{is-}\hat{\text{stg}} \rightarrow \mathcal{P}(\mathrm{A})$$

defined by

$$\begin{aligned}
&\mathcal{M}[\![<\mathbf{Inst}_1, \ldots, \mathbf{Inst}_n>]\!] << rst_1, \chi_1, return-info_1>, \\
&\qquad \ldots, < rst_n, \chi_n, return-info_n>> stg \\
\text{(E4)}\quad &= \bigcup\{cont_i'(stg_i') \mid \text{execution of } \mathbf{Inst}_i \text{ in } stg \text{ and } < rst_i, \chi_i, return-info_i> \\
&\qquad \text{yields } stg_i' \text{ and } cont_i'\}
\end{aligned}$$

where it is assumed that there exists a control tree, *cont*, such that

1. $\{\eta_1, \ldots, \eta_n\}$ = the set of terminal nodes of *cont*, and

2. for $1 \leq i \leq n$, $\mathbf{inst}_i$ is the instruction of η_i, and $< rst_i, \chi_i, return-info_i>$ is the *p−rest* of *cont* with respect to η_i.

The consistency assumption attached to E4 is to insure that the various pieces making up an instance of the equation could appear in a control tree together.

Each element of the set to which the generalized union operator is applied in E4 is a set of answers. Thus,

$$\begin{aligned}
&\mathcal{M}[\![<\mathbf{inst}_1, \ldots, \mathbf{inst}_n>]\!] << rst_1, \chi_1, return-info_1>, \\
\text{(E5)}\quad &\qquad \ldots, < rst_n, \chi_n, return-info_n>> stg \\
&= \bigcup\{anset_i \mid 1 \leq i \leq n\}
\end{aligned}$$

where $anset_i$ is determined as follows:

$$\begin{aligned}
anset_i = \ &\mathbf{if} \text{ execution of } \mathbf{inst}_i \text{ in } stg \text{ and } < rst_i, \chi_i, return-info_i> \text{ gives rise} \\
&\quad \text{to a value-returning group yielding } stg_i' \text{ and } val_i' \\
&\mathbf{then} \\
&\quad \mathcal{M}[\![<\mathbf{inst}_1', \ldots, \mathbf{inst}_h'>]\!] << rst_1', \chi_1', return-info_1'>, \ldots, \\
&\qquad < rst_h', \chi_h', return-info_h'>> stg_i' \\
&\qquad \mathbf{where}\ cont' \text{ is } rst_i \text{ with } val_i' \text{ passed up from } \chi_i \text{ according to } return-info_i,
\end{aligned}$$

---

*$\text{Is-}\hat{\text{ct}}$ is the set of control trees, $\text{is-}\hat{\text{sel}}$ is the set of compound selectors, and $\text{is-}\hat{\text{ri}}$ is the set of *return−info*s.

```
        $\{\eta_1', \ldots, \eta_h'\}$= terminal nodes of $cont'$,
        for $1 \le j \le h$, $\mathbf{inst}_j'$ is the instruction of $\eta_j'$, and
        $<rst_j', \chi_j', return-info_j'>$ is the $p-rest$ of $cont'$ w.r.t. $\eta_j'$
elif execution of $\mathbf{inst}_i$ in $stg$ and $<rst_i, \chi_i, return-info_i>$ gives rise
    to a value-returning group yielding $stg_i'$ and $cont_i'$, and $return-info_i$ is $\Omega$
then
    $\mathcal{M}[\![<\mathbf{inst}_1', \ldots, \mathbf{inst}_k'>]\!]<<rst_1', \chi_1', return-info_1'>, \ldots,$
        $<rst_k', \chi_k', return-info_k'>> stg_i'$
    where $\{\eta_1', \ldots, \eta_k'\}$= terminal nodes of $cont'$,
        for $1 \le j \le k$, $\mathbf{inst}_j'$ is the instruction of $\eta_j'$, and
        $<rst_j', \chi_j', return-info_j'>$ is the $p-rest$ of $cont'$ w.r.t. $\eta_j'$
elif execution of $\mathbf{inst}_i$ in $stg$ and $<rst_i, \chi_i, return-info_i>$ gives rise
    to a macro group yielding a control subtree $st$
then
    $\mathcal{M}[\![<\mathbf{inst}_1', \ldots, \mathbf{inst}_m'>]\!]<<rst_1', \chi_1', return-info_1'>, \ldots,$
        $<rst_m', \chi_m', return-info_m'>> stg$
    where $cont'$ is $rst_i$ with $st$ made its $\chi_i$th component,
        $\{\eta_1', \ldots, \eta_m'\}$= terminal nodes of $cont'$,
        for $1 \le j \le m$, $\mathbf{inst}_j'$ is the instruction of $\eta_j'$, and
        $<rst_j', \chi_j', return-info_j'>$ is the $p-rest$ of $cont'$ w.r.t. $\eta_j'$
fi
co Note the lack of prime after the last "stg" just before the last "where" co
```

The major reason for the complexity of E5 is to insure that interleaving of value-returning instructions is carried out to the fullest degree implied by the nondeterminism and parallelism. E5 has taken care that if a macro instruction being interleaved with $A_1, \ldots, A_n$ expands into an interleaving of $B_1, \ldots, B_k$, then all of $A_1, \ldots, A_n, B_1, \ldots, B_k$ are interleaved together. If care is not taken, it is easy to end up with only the $B_1, \ldots, B_k$ being interleaved and then when they are all done, the interleaving of $A_1, \ldots, A_n$ continuing. The former is a correct model of the nondeterminism and parallelism while the latter is not.

For a particular VDL definition of the correct form with, say, k different instructions, $INST_1, \ldots, INST_k$, E5 would be rewritten as E6 below:

(E6)

$$\begin{aligned}&\mathcal{M}[\![<\mathbf{inst}_1, \ldots, \mathbf{inst}_n>]\!]<<rst_1, \chi_1, return-info_1>,\\ &\qquad \ldots, <rst_n, \chi_n, return-info_n>> stg =\\ &\bigcup\{anset_i \mid 1 \le i \le n\}\end{aligned}$$

where $anset_i$=

```
case $\mathbf{inst}_i$ in
    $INST_1$ then $\mathcal{M}[\![si_1]\!] spr_1\ stg_1'$,
    .
    .
    .
    $INST_k$ then $\mathcal{M}[\![si_k]\!] spr_k\ stg_k'$
esac
```

where each si_j is a sequence of instructions, and each spr_j is a like-lengthed sequence of $p-rests$. $\mathcal{M}[\![si_j]\!] spr_j\ stg_j'$ is the expression obtained by carrying out the conditional expression in E5 for $anset_i$ with

$\mathbf{inst}_i = INST_j$.

It is necessary to observe that the formula E6 is an equation *scheme* rather than an equation as is normally given in a denotational semantic definition. It is an equation scheme and not an equation because of its dependence on the *n* particular instructions that happen to be at the terminal nodes of the current control tree and the *n* particular *p−rests* that happen to result when each of the *n* terminal nodes is removed from the tree, i.e., because of its dependence on the current control tree. There is one instance of this equation scheme, i.e., one equation, for each possible control tree.

In essence, the formula is an equation scheme representing a definition with an unbounded number of equations, which are finitely specifiable by the scheme.

It is interesting to note that there is yet another finite specification for this unbounded set of equations, namely the original VDL definition from which the scheme is constructed. The VDL definition, being in fact a program, is probably less hairy to the reader than the scheme, which is in a non-direct* form to permit the use of continuations or continuation-like objects.

The above equation scheme is messy, but it is in a form in which the meaning of a construct is given in terms of pieces of the control trees, i.e., *p−rests*, which state what is to be done after the execution of the current construct is finished. The next Section obtains the functions encoded by these control tree pieces.

From this informal development, it seems clear that a *p−rest* of a control tree, that is a $<rst,\chi,return-info>$ triple plays the same role for a nondeterministic computation as does an $<rst,return-info>$ pair or a continuation for a deterministic computation in that the *p−rest* tells how to proceed from the current storage and returned value to the final answers. Whereas the $<rst,return-info>$ pair or the continuation yields a unique answer for any given value, environment, and storage, the *p−rest* in general does not yield a unique answer; it yields for any given value and storage a set of answers.

This argument suggests that whatever function a *p−rest* encodes, it should be on $\mathbf{V}\times\mathbf{S}$ to $\mathcal{P}(\mathbf{A})$. The next Section shows that the formal construction of such a function from a given *p−rest*. The domain of such functions will be called **P−Cont** for for Parallel Continuation, and a typical element of this domain is called פ*.

5. Formal development of parallel continuations

The development of this Section follows that of Section 3 and in fact assumes the definitions of that Section except as modified herein. In fact, this Section effectively begins at the end of Subsection 3.1 after which the restriction to deterministic control trees is introduced. Additionally, the present development takes into account that the environment has been moved out of the state and into the instruction argument lists.

---

* in the sense of direct as opposed to continuation semantics

*פ is the Hebrew letter Peh.

5.1 Control Tree Representations

Since a node may have a set of successor nodes, a control tree representation may now be either an

1. *instr* or an
2. *instr*;
   succ−set.

with the object represented by *instr* being as before. The object represented by *instr*; *succ−set* is

$$\mu(instr;\{<\mathbf{sel}(x,succ-set):x> | x \in succ-set\})$$

where *instr* and *succ* represent objects as before, and $\mathbf{sel}(x,set)$ defines a one-to-one mapping from objects $x \in set$ to selectors in R.

A group e.g.,

$$group_i(x_1, \ldots, x_n, \xi),$$

may be in one of two forms

1. value returning:

   $$\begin{array}{l} \mathsf{PASS}:\epsilon_v(x_1, \ldots, x_n, \xi) \\ \mathbf{s-stg}:\epsilon_\sigma(x_1, \ldots, x_n, \xi) \\ \mathbf{s-c}:\epsilon_D(x_1, \ldots, x_n, \xi) \end{array}$$

   where each ϵ_i is an arbitrary expression; any of the lines may be missing; and a missing first (PASS) line is taken as equivalent to PASS:Ω

2. macro:

   $$ct-rep(x_1, \ldots, x_n, \xi)$$

   a control tree representation.

Δ_{in} can now be defined as

$$\begin{array}{l} \Delta_{in}(x_1, \ldots, x_m, \tau, ri) = \\ \qquad p_1(x_1, \ldots, x_n, \xi) \rightarrow group_1^+(x_1, \ldots, x_n, \xi, \tau, ri) \\ \qquad \cdot \\ \qquad \cdot \\ \qquad \cdot \\ \qquad p_m(x_1, \ldots, x_n, \xi) \rightarrow group_m^+(x_1, \ldots, x_n, \xi, \tau, ri) \end{array}$$

where $group_i^{+}$ is obtained from $group_i$ according to the latter's form:

1. value returning:

$$\begin{array}{l}
\langle \mu(\mu(\xi;\{\langle(\text{s-comp}(ri)\circ elem(j))\circ\text{s-al}(\text{pred}^{i}(\tau))\circ\text{s-c}:\\
\qquad \epsilon_v(x_1,\ \ldots\ ,x_n,\xi)\rangle \mid \langle i,j\rangle\in\text{s-ap}(\text{ri})\}),\\
\quad \langle \text{s-stg}:\epsilon_\sigma(x_1,\ \ldots\ ,x_n,\xi)\rangle,\\
\quad \langle \text{s-c}:\epsilon_{\mathfrak{D}}(x_1,\ \ldots\ ,x_n,\xi)\rangle),\\
\epsilon_v(x_1,\ \ldots\ ,x_n,\xi)\rangle
\end{array}$$

2. macro:

$$\begin{array}{l}
\langle \mu(\xi;\langle\tau\circ\text{s-c}:\mu(ct-rep(x_1,\ \ldots\ ,x_n,\xi);\langle\text{s-ri}:ri\rangle)\rangle),\\
\Omega\rangle
\end{array}$$

With these definitions it is possible to define a function p-cont on $p-rest$s to **P−Conts**,

$$\text{p-cont}:\text{is-}\hat{\ }\text{ct}\times\text{is-}\hat{\ }\text{sel}\times\text{is-}\hat{\ }\text{ri}\rightarrow\mathbf{P-Cont}$$

which given the components of a $p-rest$, yields a **P−Cont**, $\mathfrak{D}$, such that

$$\mathfrak{D}:\mathbf{V}\times\mathbf{S}\rightarrow\mathcal{P}(\mathbf{A}).$$

Figure 9 is useful for seeing where the pieces of the definition of p-cont come from.

In the definition the formal parameters of p-cont, ct, τ, and ri together constitute a $p-rest$. The result is a function with formal parameters $\epsilon\in\mathbf{V}$ and $\sigma\in\mathbf{S}$ which, as can be seen, returns a set of answers.

Definition 2. Suppose $\text{is-c}(ct)$ and $\text{is-ri}(ri)$. Suppose $\text{tn}(ct)$ yields a set, i.e., ct is nondeterministic. Suppose $\tau\in\text{tn}(ct)$.
Then,

$$\begin{array}{l}
\text{p-cont}(ct,\tau,ri) = \lambda(\epsilon,\sigma).\\
\qquad (is-\Omega(ct)\rightarrow\{\sigma\},\\
\qquad T\rightarrow\bigcup\{\text{p-cont}(ct_i{}',\text{pred}(\tau_i,nri_i)(\epsilon_i{}',\sigma_i{}'))\mid 1\leq i\leq m\})
\end{array}$$

where

$$\begin{array}{l}
ct\uparrow = \mu(ct;\{\langle(\text{s-comp}(\text{ri}))\circ elem(j)\circ\text{s-al}(\text{pred}^{i-1}(\tau)):\epsilon\rangle\mid\langle i,j\rangle\in\text{s-ap}(ri)\})\\
\{\tau_1,\ \ldots\ ,\tau_m\} = tn(ct\uparrow)\\
\text{for each } i\in\{j\mid 1\leq j\leq m\},\\
\quad node_i = \tau_i(ct\uparrow)\\
\quad \mathbf{nin}_i = \text{s-in}(node_i),\\
\quad nal_i = \text{s-al}(node_i),\\
\quad nri_i = \text{s-ri}(node_i),
\end{array}$$

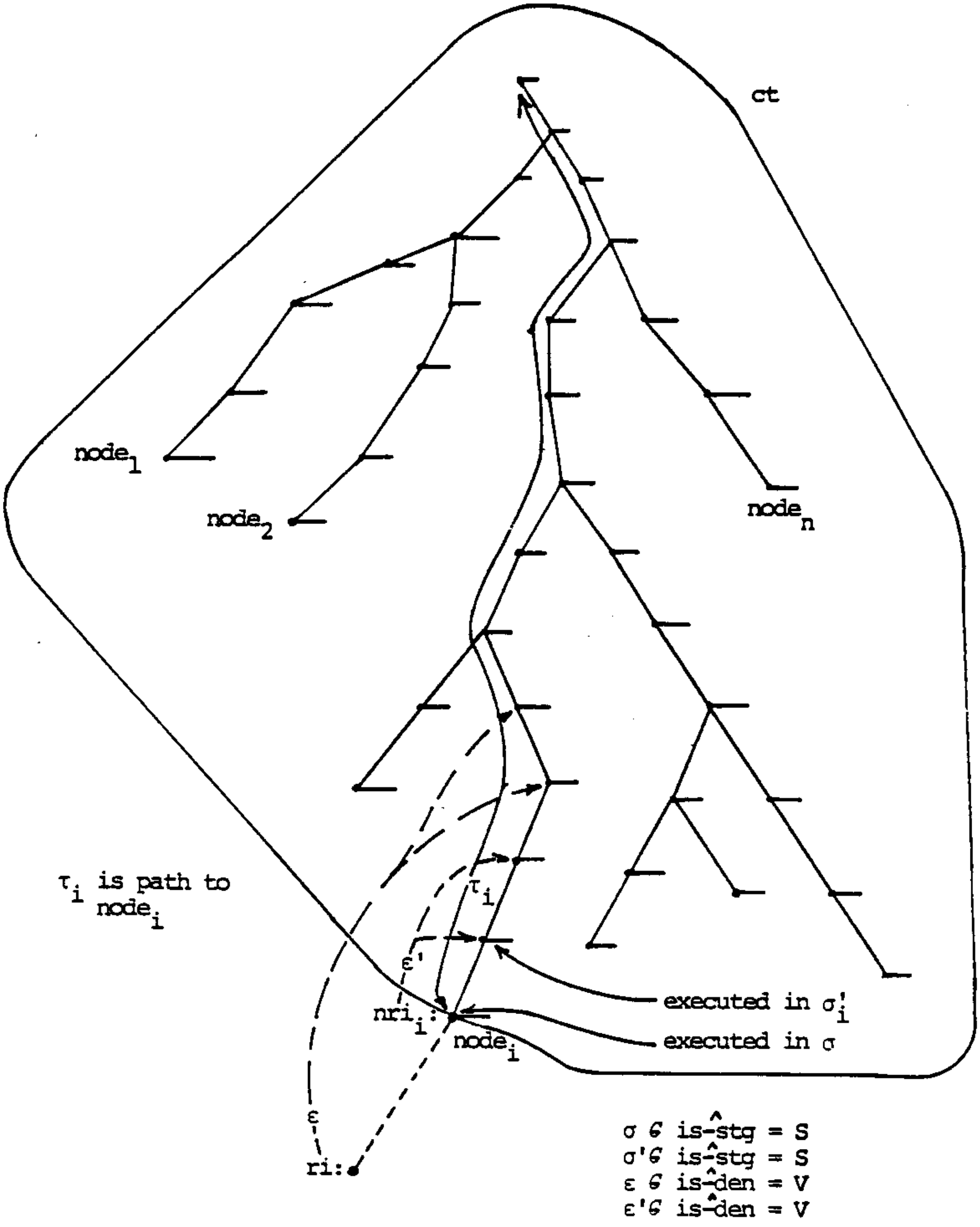

ct
node$_1$
node$_2$
node$_n$
τ_i is path to node$_i$
τ_i
ε'
nri$_i$:
node$_i$
executed in σ'_i
executed in σ
ε
ri:
$\sigma \in$ is-$\hat{}$stg = S
$\sigma' \in$ is-$\hat{}$stg = S
$\varepsilon \in$ is-$\hat{}$den = V
$\varepsilon' \in$ is-$\hat{}$den = V

Figure 9

n_i = number of formal parameters in schema for s–in($node_i$)

$$
\begin{aligned}
&\langle \mu_o(<\text{s–stg}:\sigma_i'>,<\text{s–c}:ct_i'>),\epsilon_i'\rangle = \\
&\qquad \Delta_{nin_i}(elem(1,al_i),\ .\ .\ .\ ,elem(n_i,al_i), \\
&\qquad\quad \mu_o(<\text{s–stg}:\sigma>,<\text{s–c}:\delta(ct\uparrow,\tau_i)>),\tau_i,nri_i)
\end{aligned}
$$

6. Attempt to Use P-continuations in Equations

Thus, it is possible given a *p−rest* to construct the parallel continuation it encodes. The question now arises as to whether the nondeterministic meaning equation E5 of Section 4.5 can be recast into a form in which the corresponding parallel continuation can be used in place of each *p−rest*. After all, denotational continuation semantics for deterministic computations does use a continuation wherever the equations of this paper use a *<rst,return−info>* pair.

The answer seems to be "No!". The control tree of a *p−rest* carries information which is lost when its parallel continuation is extracted. Both convey the set of all possible final answers starting from a particular (single) storage and a particular (single) value. However, only the control tree also exhibits what will be the set of terminal nodes to choose from following the selection and execution of one of its terminal nodes. This information is needed to construct the *p−rest* of the *rst* of the current *p−rest* with respect to *each* of the resulting terminal nodes. *Only with this information can macro expansion result in an interleaving of the nodes of the appended subtree with those of the original rst.*

Therefore, it seems that the meaning equation must be stated in terms of *p−rests* which explicitly encode the nondeterministic choices and that a parallel condition cannot be used in the equation and maintain completely interleaved nonindependent parallelism. It is interesting though, that from any such *p−rest*, the parallel continuation it encodes is obtainable.

This loss of information in going from control tree to continuation is not critical in the deterministic case because there is only one choice at any time, and thus, it is not necessary to encode choices.

As a consequence, the semantics for parallelism proposed in this paper is a continuation semantics only in an indirect manner.

Is the semantics denotational in the sense that the meaning of a syntactic construct is constructible from those of its direct syntactic components? If one extends the notion of syntactic construct to include a set of them, then the answer is "Yes!" The meaning equation says that the meaning of a sequence (really, a set) of syntactic entities is constructed from the meanings of the direct components of the individual elements of the sequence (really, the set).

The reader should observe the similarity between the *p−rests* and Plotkin's resumptions. In both cases, the transition from a state yields a set of possible next states plus *p−rests* or resumptions, as the case may be. Both carry enough information to continue in the same manner from any of the possible next states. Plotkin and Smyth also observe that the more abstract functions on states to sets of states, called P-continuations in the present paper, are not powerful enough to model full interleaving. The present paper, however, shows how to obtain the more abstract functions from the less abstract *p−rests*.

6.1 Alternative Meaning and Interpreter

It is thus recognized that the proposed semantic definition for nondeterminism and parallelism is only denotational and cannot be made also continuation. An opportunity then arises to abandon the non-direct form of the meaning equation scheme, which has various nonindependent pieces of one control tree on both sides and for which there must be integrity constraints guaranteeing that the pieces could come from one single tree. This non-direct form was used from the beginning to permit eventual replacement of the $p-rest$s by parallel continuations. With the possibility of this replacement scuttled, perhaps a more direct form of a meaning equation scheme can be used in which the meaning function has a single whole control tree as a parameter:

$$\mathcal{M}' \in \text{is-}\hat{\text{c}}\text{t} \times \text{is-}\hat{\text{s}}\text{tg} \rightarrow \mathcal{P}(\text{A}).$$

Equation E6 defining

$$\mathcal{M}[\![<\mathbf{inst}_1, \ldots, \mathbf{inst}_n>]\!]<<rst_1,\chi_1,return-info_1>, \\ \ldots,<rst_n,\chi_n,return-info_n>>stg$$

can be revised to define

$$\mathcal{M}'[\![cont]\!]stg$$

where $cont$ is the control tree with terminal nodes $\{\eta_1, \ldots, \eta_n\}$ such that for $1 \leq i \leq n$, $\mathbf{inst}_i$ is the instruction of η_i and $<rst_i,\chi_i,return-info_i>$ is the $p-rest$ of $cont$ with respect to η_i. Specifically

(E7) $$\mathcal{M}'[\![cont]\!]stg = \bigcup\{anset_i \mid 1 \leq i \leq n\}$$

where $anset_i$ is determined as follows:

```
$anset_i$ = if execution of $\mathbf{inst}_i$ in $stg$ and $<rst_i,\chi_i,return-info_i>$ gives rise
        to a value-returning group yielding $stg_i'$ and $val_i'$
    then
        $\mathcal{M}'[\![cont_i']\!]stg_i'$
        where $cont'$ is $rst_i$ with $val_i'$ passed up from $\chi_i$ according to $return-info_i$,
    elif execution of $\mathbf{inst}_i$ in $stg$ and $<rst_i,\chi_i,return-info_i>$ gives rise
        to a value-returning group yielding $stg_i'$ and $cont_i'$, and $return-info_i$ is $\Omega$
    then
        $\mathcal{M}'[\![cont_i']\!]stg_i'$
    elif execution of $\mathbf{inst}_i$ in $stg$ and $<rst_i,\chi_i,return-info_i>$ gives rise
        to a macro group yielding a control subtree $st$
    then
        $\mathcal{M}'[\![cont_i']\!]stg$
        where $cont'$ is $rst_i$ with $st$ made its $\chi_i$th component
    fi
```

This new meaning function is cleaner that the original but is still only an equation scheme. This meaning function is essentially the interpreter, and its most direct finite specification is the VDL definition itself!

6.2 Denotation of Parallel Programs

If one chooses to write a denotational definition of the language using the equation scheme suggested by E6, it is legitimate to ask, "Just what is the denotation of a program?" Because of the existence of the function `p-cont` yielding the element of **P−Cont** encoded by any given $p-rest$, there two possible answers to this question. The denotation of a program can be taken either as a $p-rest$ or as a parallel continuation.

Given a program p, the $p-rest$ denoting p is[*]

<**interpret−program**(p),I,Ω>,

and the parallel continuation denoting p is

`p-cont`(**interpret−program**(p),I,Ω).

Each of these applied (in its own way) to an initial storage stg_0 and an empty value yields the set of all answers resulting from executing p with an initial storage stg_0.

If one insists that whatever is taken as the denotation be useable in an equation scheme of the form suggested by E6 and that the denotation of a construct be constructible from the *same kind* of denotation of its direct syntactic components, then only the $p-rest$ denotation can be used.

In either case, the denotation of a program and its set of final answers depends of the choice of which operations are indivisible, i.e., value-returning in VDL. It is these indivisible operations that get interleaved to produce a computation sequence. A different choice of indivisible operations yields a different set of possible interleavings and thus a different set of final answers.

Note that the denotation of $p{:=}p+1;p{:=}p+1$,

<**interpret−st−list**(<$p{:=}p+1,p{:=}p+1$>),I,Ω>,

is different from that of $p{:=}p+2$,

<**interpret−st**($p{:=}p+2$),I,Ω>.

However, this is as it should be, because in the presence of other processes accessing the same memory, the results of the two program fragments could very well be different.

There might be objections to tying the definition of a programming language to such an implementation dependent concept as the set of indivisible operations. However,

1. ultimately, in any computational system permitting shared access of a common storage medium, there is a smallest, indivisible operation that cannot be interrupted -- usually the assignment to a single word or byte. (Without this shared access to a storage medium, parallelism is uninteresting and poses no problems; the processes are completely independent and can be defined completely separately.)

2. all programming languages known to this author assume that certain operations, especially assignment to a single variable, are indivisible. Even Algol 68, whose definition [vWn75] says explicitly

---

[*] I is the identity selector.

that which actions are inseparable (indivisible) is left undefined and thus up to the implementation, ends up specifying *effectively indivisible* operations [Sch78]. There are other axioms in the definition that allow deducing that even if assignation of a single variable is not indivisible in fact in an implementing machine, it must be implemented as if it were.

3. at the language level, synchronization is used to make long sequences of operations effectively indivisible. This fact carries the implication of the existence of some indivisible operations, e.g., incrementation and test-and-set, with which the synchronization primitives can be implemented.

The final technical question is whether the recursive function definitions given in Sections 3 and 5 of `cont` and `p-cont` have fixed points. This author is not in a position to answer this question and welcomes anyone to consider the question. He feels however that the definitions do define functions simply because he knows that the VDL functions on which they are based work and the notion of a computation as a sequence of states generalized by Λ works. These new functions capture a whole computation as a recursive rendition of the loop:

```
while ∃ an instruction to execute do
    select one instruction inst;
    determine set of possible next instructions;
    execute inst
od.
```

7. Conclusion

This paper has presented a denotational semantics for nondeterminism and parallelism which is at once

1. powerful enough to deal with arbitrary interleaved access to shared memory,
2. systematically constructible from an operational semantics of the same, and
3. such that a more abstract functional meaning on states to sets of states is obtainable from the given meaning.

This paper has also dealt with the relation between VDL and denotational semantics. The conclusion is that from a technical point of view, it does not matter whether a VDL or a denotational semantic definition is given of a programming language. Section 3 shows how to construct a denotational continuation semantic definition from a deterministic VDL semantic definition. Appendix IV gives a VDL definition constructed from the example denotational continuation semantic definition found in Chapter 13 of [Ten81]. Because each can be constructed from each other, both are based on the same firm mathematical grounds. It becomes strictly a matter of taste as to which style is used. A language definer should take into account which is easier for him or her to write and which is easier for the intended audience to use in the intended manner.

One supposed advantage of denotational semantics is that it is easily used in proofs of properties about the defined language. However, the fact is that operational and especially VDL definitions have been used in proofs also [HJ70, JL71, McG70, McG72, Bry72]. A favorite exercise is to demonstrate the correctness of an implementation of a language or a feature by showing the implementation operationally equivalent to the definition. In general, any property may be proved by a computational induction that shows it true in all states of a computation.

It is clear that the mutual constructibility extends also to any formal presentation of operational semantics. Consider the most recent (to this author's knowledge) presentation of operational semantics developed by Hennessy, Li, and Plotkin (See e.g., [Plo83]). A specification in their structured operational semantics can be systematically converted to denotational form either directly or via VDL. The rules

$$\langle \mathbf{skip},\sigma\rangle \to \sigma$$

$$\frac{\langle y,\sigma\rangle \to v}{\langle x{:=}y,\sigma\rangle \to \sigma^{x}_{v}}$$

$$\frac{\langle c_0,\sigma\rangle \to \sigma'}{\langle c_0;c_1\rangle \to \langle c_1,\sigma'\rangle}$$

can be expressed in the denotational equations as they suggest,

$$\mathcal{C}[\![\mathbf{skip}]\!]\sigma=\sigma$$

$$\mathcal{C}[\![x{:=}y]\!]\sigma=\sigma^{x}_{v}$$
where $\mathcal{C}[\![y]\!]\sigma=v$

$$\mathcal{C}[\![c_0;c_1]\!]\sigma=\mathcal{C}[\![c_0]\!]\circ\mathcal{C}[\![c_1]\!]\sigma\ .$$

Alternatively, the rules may be converted into a VDL instruction for a machine with only a control tree,*

```
interpret($c,\sigma$)=
   $c=[\![\mathbf{skip}]\!]\to$
         pass($\sigma$)
   $c=[\![x{:=}y]\!]\to$
         pass($\mu(\sigma;\langle x{:}v\rangle)$);
            $v$:interpret($y,\sigma$)
   $c=[\![c_0;c_1]\!]\to$
         interpret($c_1,\sigma'$);
            $\sigma'$:interpret($c_0,\sigma$)
```

and then to almost the same denotational equations.

Finally, this paper has observed that VDL control trees really encode continuations. One may note that the control tree idea is even more prevalent than just in VDL definitions. Examination of Section 2.1.4, *Actions*, of the Revised Algol 68 Report [vWn75] shows that the interpreter, described in English, is really a control tree with actions at each node. An action is the elaboration (execution) of a construct (piece of program text) in an environ (environment). This elaboration may both have an effect on the state and yield a value. Furthermore, examination of Sections 1.2.3 through 1.2.5, on operations, instructions, and the mechanization of the meta-language, of the ECMA/ANSI PL/I BASIS/I definition [ANSI74] shows that the PL/I interpretation machine also has a control tree, namely a parse tree of instructions in the process of being executed. By a suitable systematic construction, these two definitions may be converted to denotational semantic definitions.

---

*Lest the reader draw the conclusion that VDL produces longer definitions, observe that short instruction names e.g., "C" or "<" could have been used as well.

Acknowledgements

The author thanks Orna Berry, Ed Blum, Jean-Marie Cadiou, Michael Gordon, Daniel Lehmann, Peter Lucas, Peter Lauer, David Jefferson, Dave MacQueen, Dave Martin, Shahrzade Mazaher, Richard Schwartz, Peter Wegner, and Maria Zamfir for their helpful comments during oral presentations of this work. Their comments helped to improve this paper. The author also thanks the referees of previous versions of this paper for their helpful comments.

Bibliography

[ADA79] "Preliminary Draft: Green Language Formal Definition", CII Honeywell Bull (April, 1979).

[ANSI74] ECMA-ANSI, *PL/I-BASIS/I*, ECMA/TC10, ANSI.X3J1 (July,1974).

[Bry72] Berry, D.M., "The Equivalence of Models of Tasking", *Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices* 7:1 (January, 1972).

[BW72] Berry, D.M. and P. Wegner, "Adding Labels to EPL", Faculty of Mathematics, Hebrew University, Jerusalem, Israel (1972).

[FHLR79] Francez, N., C.A.R. Hoare, D.J. Lehmann, and W.P. de Roever, "Semantics of Nondeterminism, Concurrency, and Communication", *JCSS* 19, 290-308 (1979).

[FLP80] Francez, N., D.J. Lehmann, and A. Pnueli, "A Linear History Semantics for Distributed Languages", *21st Annual Symposium on Foundations of Computer Science* (1980).

[Gor79] Gordon, M.S.C., *The Denotational Description of Programming Languages: An Introduction*, Berlin: Springer-Verlag (1979).

[HJ70] Henhapl, W. and C.B. Jones, "The Block Concept and Some Possible Implementations, with Proofs of Equivalence", IBM Laboratory Vienna, Tech. Rep. TR 25.10 (1970).

[JL71] Jones, C.B. and P. Lucas, "Proving Correctness of Implementation Techniques", in Engeler, E. (ed.) *Symposium on Semantics of Algorithmic Languages*, Berlin: Springer-Verlag (1971).

[LLS70] Lucas, P., P.E. Lauer and H. Stigleitner, "Method and Notation For the formal Definition of Programming Languages", rev. ed., IBM Lab, Vienna, Tech. Rep. TR 25.087 (1970).

[LW69] Lucas, P. and K. Walk, "On the formal description of PL/I", *Annual Review in Automatic Programming* 6:3 (1969).

[Maz71] Mazurkiewicz, A.W., "Proving Algorithms by Tail Functions" *Information and Control* 18, 220-226 (1971).

[McG70] McGowan, C.L., "The Correctness of a Modified SECD Machine", *Second ACM Symposium on Theory of Computing* (1970).

[McG72] McGowan, C.L., "An Inductive Proof Technique for Interpreter Correctness", in Rustin, R. (ed.) *Formal Semantics of Computer Languages*, Englewood Cliffs: Prentice-Hall (1972).

[Plo76] Plotkin, G.D., "A Power Domain Construction", *SIAM J. Comput.* 5,3 (1976).

[Plo83] Plotkin, G.D., "An Operational Semantics for CSP", in Bjørner (Ed.), *Formal Description of Programming Concepts II*, North-Holland, Amsterdam (1983).

[Sch78] Schwartz, R.L., "An Axiomatic Semantic Definition of Algol 68", Tech. Rep. UCLA-ENG-7838, Computer Science Dept., UCLA (1978).

[Sch79] Schwarz, J.S., "Denotational Semantics of Parallelism", *Semantics of Concurrent Computation*, Berlin: Springer-Verlag (1979).

[Smy78] Smyth, M.B., "Power Domains", *JCSS* 16, 23-36 (1978).

[Sto81] Stoughton, A., "Access Flow: A Protection Model which Integrates Access Control and Information Flow", Computer Science Dept., UCLA (1981).

[Ten76] Tennent, R.D., "The Denotational Semantics of Programming Languages", *CACM* 19 (1976).

[Ten78] Tennent, R.D., "A Practical Guide to Denotational Semantic Definitions", Univ. of Oxford, Programming Research Group (April 1978).

[Ten81] Tennent, R.D., *Principles of Programming Languages*, Englewood Cliffs: Prentice-Hall (1981).

[vWn75] van Wijngaarden, A. *et al* (eds.), "Revised Report on the Algorithmic Language Algol 68", *Acta Informatica* 5 (1975).

[Weg70] Wegner, P. "Three Computer Cultures: Computer Technology, Computer Mathematics, and Computer Science", *Advances in Computers* 10 (1970).

[Weg72] Wegner, P., "The Vienna Definition Language", *Computing Surveys* 4:1 (March, 1972).

Appendix I

Abstract Syntax of Program:

(A1) is-progr = is-block
(A2) is-block = (<s-decl-part:is-decl-part>,<s-st-list:is-st-list>)
(A3) is-decl-part = ({<id:is-attr> ‖ is-id(id)})
(A4) is-attr = is-var-attr ∨ is-proc-attr ∨ is-funct-attr ∨ is-label-attr
(A5) is–var̂–attr = {INT,LOG,LABVAR}
(A6) is-proc-attr = (<s-param-list:is-id-list>,<s-st:is-st>)
(A7) is-funct-attr = (<s-param-list:is-id-list>,<s-st:is-st>,<s-expr:is-expr>)
(A7.5) is-label-attr = is-st-list
(A8) is-st = is-assign-st ∨ is-cond-st ∨ is-proc-call ∨ is-block ∨
    is-goto-st ∨ is-while-st
(A9) is-assign-st = (<s-left-part:is-var>,<s-right-part:is-expr>)
(A10) is-expr = is-cont ∨ is-var ∨ is-funct-des ∨ is-bin ∨ is-unary
(A11) is-const = is-log ∨ is-int
(A12) is-var = is-id
(A13) is-funct-des = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A14) is-bin = (<s-rd1:is-expr>,<s-rd2:is-expr>,<s-op:is-bin-rt>)
(A15) is-unary = (<s-rd:is-expr>,<s-op:is-unary-rt>)
(A16) is-cond-st = (<s-expr:is-expr>,<s-then-st:is-st>,<s-else-st:is-st>)
(A17) is-proc-call = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A18) is-goto-st = id-id
(A19) is-while-st = (<s-cond:is-expr>,<s-body:is-st>)

where:

is–ĩd	an infinite set of identifiers
is–l̂og	a set of constants denoting the truth values
is–înt	an infinite set of constants denoting the integer values
is–unâry–rt	a set of unary (one-place) operators
is–binâry–rt	a set of binary (two-place) operators
{INT,LOG}	two attributes used to distinguish integer variables from logical variables.

Abstract Syntax of State:

(S1) is-state = (<s-env:is-env>,<s-c:is-c>,<s-at:is-at>,<s-dn:is-dn>,
    <s-d:is-d>,<s-n:is-integer-value>)
(S2) is-env = ({<id:is-n> ‖ is-id(id)})
(S3) is-c = ... (standard control trees as defined in LLS70)
(S4) is-at = ({<n:is-type> ‖ is-n(n)})
(S5) is–t̂ype = {INT,LOG,PROC,FUNCT,LABVAR,LABCONST}
(S6) is-dn = ({<n:is-value ∨ is-proc-den ∨ is-funct-den ∨ is-label-den>
    ‖ is-n(n)})
(S6.1) is-proc-den = (<s-env:is-env>,<s-attr:is-proc-attr>)

(S6.2) is-funct-den = (<s-env:is-env>,<s-attr:is-funct-attr>)
(S6.3) is-label-den = (<s-env:is-env>,<s-c:is-c>,<s-d:is-d>)
(S7) is-d = (<s-env:is-env>,<s-c:is-c>,<s-d:is-d>) $\vee$ is-Ω

where:

is-$\hat{n}$	infinite set of names (used for the generation of unique names)
{PROC,	two attributes used to distinguish function names and procedure names
{s-env,s-c, s-at,s-dn,s-n}	selectors for the components of the interpreting machine
is–vâlue	infinite set of values, the integers and the truth values

The *initial state* for any given program t $\in$is–p$\hat{r}$ogr is:

μ_0(<s-c:int–progr(t)>,<s-n:1>)

States ξ whose control part s-c(ξ) is Ω are *end states*.

Abbreviations used in Instruction Schemata:

ENV	s-env(ξ)
C	s-c(ξ)
AT	s-at(ξ)
DN	s-dn(ξ)
D	s-d(ξ)

Instruction Schemata:

```
(I1)    int–progr(t) = int–block(t)
        for: is-progr(t)

(I2)    int–block(t) =
                s–d:$\mu_0$(<s–env:ENV>,<s–c:C>,<s–d:D>)
                s–c:exit;
                        int–st–list(s–st–list(t));
                            int–decl–part(s–decl–part(t));
                                update–env(s–decl–part(t))
        for: is-block(t)

(I3)    update–env(t) =
                null;
                    {update–id(id,n);n:un–name|id(t)$\neq\Omega$}
        for: is-decl-part(t)
```

```
(I4)    update-id(id,n) =
                s-env:μ(ENV;<id:n>)
        for: is-id(id), is-n(n)

(I5)    int-decl-part(t) =
                null;
                    {int-decl(id(ENV),id(t))|id(t)≠Ω}
        for: is-decl-part(t)

(I6)    int-decl(n,attr) =
                is-var-attr(attr)→
                        s-at:μ(AT;<n:attr>)
                is-proc-attr(attr)→
                        s-at:μ(AT;<n:PROC>)
                        s-dn:μ(DN;<n:μ_0(<s-attr:attr>,<s-env:ENV>)>)
                is-funct-attr(attr)→
                        s-at:μ(AT;<n:FUNCT>)
                        s-dn:μ(DN;<n:μ_0(<s-attr:attr>,<s-env:ENV>)>)
                is-label-attr(attr)→
                        s-at:μ(AT;<n:LABCONST>)
                        s-dn:μ(DN;<n:μ_0(<s-env:ENV>,<s-d:D>,
                                                <s-c:exit;
                                                        int-st-list(attr);>)>)
        for: is-n(n), is-attr(attr)

(I7)    int-st-list(t) =
                is-<>(t)→null
                T→
                        int-st-list(tail(t));
                            int-st(head(t))
        for: is-st-list(t)

(I8)    int-st(t) =
                is-assign-st(t)→int-assign-st(t)
                is-cond-st(t)→int-cond-st(t)
                is-proc-call(t)&(at_t = PROC)→int-proc-call(t)
                is-block(t)→int-block(t)
                is-goto-st(t)&(at_l = LABCONST∨at_l = LABVAR)→int-goto-st(t)
                is-while-st(t)→int-while-st(t)
        where: at_t = ((s-id(t))(ENV))(AT),
                  at_l = (t(ENV))(AT)
        for: is-st(t)

(I9)    int-assign-st(t) =
                is-var-attr(n_t(AT))→
                                assign(n_t,v);
                                    v:int-expr(s-right-part(t))
                T→error
        where: n_t = (s-left-part(t))(ENV)
        for: is-assign-st(t)

(I10)   assign(n,v) =
```

```
              s−dn:μ(DN;n<:convert(v,n(AT))>)
        for: is-n(n), is-value(v)

(I11)   int−cond−st(t) =
              branch(v,s–then–st(t),s–else–st(t));
                 v:int−expr(s–expr(t))
        for: is-cond-st(t)

(I12)   branch(v,st1,st2) =
              convert(v,LOG)→int−st(st1)
              ¬convert(v,LOG)→int−st(st2)
        for: is-value(v), is-st(st1), is-st(st2)

(I13)   int−proc−call(t) =
              (length(arg–list$_t$) = length(p–list$_t$))→
                       s−env:μ(env$_t$;{<elem(i,p–list$_t$):elem(i,arg–list$_t$)(ENV)> |
                                $1 \le i \le$length(p–list$_t$)}
                       s−d:$\mu_0$(<s–env:ENV>,<s–c:C>,<s–d:D>)
                       s−c:exit;
                                int−st(st$_t$)
              T→error
        where: n$_t$ = (s–id(t))(ENV),
                 p–list$_t$ = s–param–list • s–attr • n$_t$(DN),
                 env$_t$ = s–env • n$_t$(DN),
                 arg–list$_t$ = s–arg–list(t),
                 st$_t$ = s–st • s–attr • n$_t$(DN)
        for: is-proc-call(t))

(I14)   exit =
                 s−env:s–env(D)
                 s−c:s–c(D)
                 s−d:s–d(D)

(I15)   int−expr(t) =
              is–bin(t)→
                       int−bin−op(s–op(t),a,b);
                          a:int−expr(s–rd1(t)),
                          b:int−expr(s–rd2(t))
              is–unary(t)→
                       int−un−op(s–op(t),a);
                          a:int−expr(s–rd(t))
              is–funct–des(t)&(at$_t$ = FUNCT)→
                       pass−value(n);
                          int−funct−call(t,n);
                             n:un−name
              is–var(t)&is–var–attr(n$_t$(AT))→
                       PASS:n$_t$(DN)
              is–const(t)→
                       PASS:value(t)
              T→error
        where: n$_t$ = t(ENV),
                 at$_t$ = ((s–id(t))(ENV))(AT)
```

```
        for: is-expr(t)

(I16)   pass−value(n) =
                PASS:n(DN)

(I17)   int−funct−call(t,n) =
                (length($arg–list_t$) = length($p–list_t$))→
                        s−env:$\mu$($env_t$;{<elem(i,$p–list_t$):elem(i,$arg–list_t$)(ENV)> |
                                $1 \leq i \leq$ length($p–list_t$)}
                        s−d:$\mu_0$(<s–env:ENV>,<s–c:C>,<s–d:D>)
                        s−c:exit;
                                assign(n,v);
                                    v:int−expr($expr_t$);
                                        int−st($st_t$)
                T→error
        where: $n_t$ = (s–id(t))(ENV),
                $p–list_t$ = s–param–list ∘ s–attr ∘ $n_t$(DN),
                $env_t$ = s–env ∘ $n_t$(DN),
                $arg–list_t$ = s–arg–list(t),
                $st_t$ = s–st ∘ s–attr ∘ $n_t$(DN),
                $expr_t$ = s–expr ∘ s–attr ∘ $n_t$(DN)
        for: is-funct-des(t), is-n(n)

(I18)   int−goto−st(t) =
                s−env:$env_t$
                s−d:$d_t$
                s−c:$c_t$
        where:$n_t$ = t(ENV),
                $env_t$ = s–env ∘ $n_t$(DN),
                $d_t$ = s–d ∘ $n_t$(DN),
                $c_t$ = s–c ∘ $n_t$(DN)
        for: is-goto-st(t)

(I19)   int−while−st(t) =
                loop−or−exit(v,t);
                    v:int−expr(s–cond(t))
        for: is-while-st(t)

(I20)   loop−or−exit(v,t) =
                convert(v,LOG)→
                        int−while−st(t);
                            int−st(s–body(t))
                ¬convert(v,LOG)→null
        for: is-value(v), is-while-st(t)

where (defined):
        un−name =
                PASS:$n_{s-n(\xi)}$
                s−n:s–n($\xi$)+1

        null =
                PASS:$\Omega$
```

where:

The following functions and instructions are not further specified:

convert(v,attr)	function which yields v converted (if necessary) to the type specified by attr which may either be INT or LOG.
int−bin−op(op,a,b)	Instruction which returns the result of applying the operator op to a and b. It is left open whether there is a conversion performed in case the operator is not applicable to operands of type a and b.
int−un−op(op,a)	Instruction which returns the result of applying the operator op to a (for the problem of conversion see above).
value(a)	Function which yields the value given a constant a.

Appendix II

Abstract Syntax of Program:

(A1) is-progr = is-block
(A2) is-block = (<s-decl-part:is-decl-list>,<s-st-list:is-st-list>)
(A3) is-decl = (<s-id:is-id>,<s-attr:is-attr>)
(A4) is-attr = is-var-attr ∨ is-proc-attr ∨ is-funct-attr ∨ is-label-attr
(A5) is-var-attr = {INT,LOG,LABVAR}
(A6) is-proc-attr = (<s-param-list:is-id-list>,<s-st:is-st>)
(A7) is-funct-attr = (<s-param-list:is-id-list>,<s-st:is-st>,<s-expr:is-expr>)
(A7.5) is-label-attr = is-st-list
(A8) is-st = is-assign-st ∨ is-cond-st ∨ is-proc-call ∨ is-block ∨
is-goto-st ∨ is-while-st
(A9) is-assign-st = (<s-left-part:is-var>,<s-right-part:is-expr>)
(A10) is-expr = is-cont ∨ is-var ∨ is-funct-des ∨ is-bin ∨ is-unary
(A11) is-const = is-log ∨ is-int
(A12) is-var = is-id
(A13) is-funct-des = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A14) is-bin = (<s-rd1:is-expr>,<s-rd2:is-expr>,<s-op:is-bin-rt>)
(A15) is-unary = (<s-rd:is-expr>,<s-op:is-unary-rt>)
(A16) is-cond-st = (<s-expr:is-expr>,<s-then-st:is-st>,<s-else-st:is-st>)
(A17) is-proc-call = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A18) is-goto-st = id-id
(A19) is-while-st = (<s-cond:is-expr>,<s-body:is-st>)

where:

is-id	an infinite set of identifiers
is-log	a set of constants denoting the truth values
is-int	an infinite set of constants denoting the integer values
is-unary-rt	a set of unary (one-place) operators
is-binary-rt	a set of binary (two-place) operators
{INT,LOG}	two attributes used to distinguish integer variables from logical variables.

Abstract Syntax of State:

(S1) is-state = (<s-env:is-env>,<s-c:is-c>,<s-stg:is-stg>)
(S2) is-env = ({<id:is-n> ‖ is-id(id)})
(S3) is-c = ... (standard control trees as defined in LLS70)
(S4) is-stg = ({<n:(<s-dn:is-den>,<s-at:is-type>) ‖ is-n(n)>})
(S5) is-type = {INT,LOG,PROC,FUNCT,LABVAR,LABCONST}
(S6) is-den = is-proc-den ∨ is-funct-den ∨ is-value ∨
is-label-den ∨ is-UNINIT
(S6.1) is-proc-den = (<s-env:is-env>,<s-attr:is-proc-attr>)
(S6.2) is-funct-den = (<s-env:is-env>,<s-attr:is-funct-attr>)

(S6.3) is-label-den = (<s-env:is-env>,<s-c:is-c>)

where:

is-n̂	infinite set of names (used for the generation of unique names)
{PROC, FUNCT}	two attributes used to distinguish function names and procedure names
{s-env,s-c, s-at,s-dn, s-stg}	selectors for the components of the interpreting machine
is-vâlue	infinite set of values, the integers and the truth values

The *initial state* for any given program t ∈is-prôgr is:
μ_0(<s-c:**int-progr**(t)>,<s-n:1>)

States ξ whose control part s-c(ξ) is Ω are *end states*.

Abbreviations used in Instruction Schemata:

ENV	s-env(ξ)
C	s-c(ξ)
STG	s-stg(ξ)

Instruction Schemata

```
(I1)    int-progr(t) = int-block(t)
        for: is-progr(t)

(I2)    int-block(t) =
              exit(ENV);
                 int-st-list(s-st-list(t));
                    int-decl-part(s-decl-part(t),C,ENV);
                       update-env(s-decl-part(t))
        for: is-block(t)

(I3)    update-env(t) =
              is-<>(t)→null
              T→
                       update-env(tail(t));
                          update-id(s-id(head(t)),n);
                             n:un-name
        for: is-decl-part(t)

(I4)    update-id(id,n) = s-env:μ(ENV;<id:n>)
```

for: is-id(id), is-n(n)

```
(I5)    int-decl-part(t,outerct,outerenv) =
            is-<>(t)→null
            T→
                    int-decl-part(tail(t),outerct,outerenv);
                       int-decl(s-id(head(t))(ENV),s-attr(head(t)),outerct,outerenv);
        for: is-decl-part(t), is-c(outerct), is-env(outerenv)
```

```
(I6)    int-decl(n,attr,outerct,outerenv) =
            is-var-attr(attr)→
                    s-stg:$\mu$(STG;
                            <n:$\mu_0$(<s-at:attr>,
                                <s-dn:UNINIT>)>)
            is-proc-attr(attr)→
                    s-stg:$\mu$(STG;
                            <n:$\mu_0$(<s-at:PROC>,
                                <s-dn:$\mu_0$(<s-attr:attr>,
                                              <s-env:ENV>)>)>)
            is-funct-attr(attr)→
                    s-stg:$\mu$(STG;
                            <n:$\mu_0$(<s-at:FUNCT>,
                                <s-dn:$\mu_0$(<s-attr:attr>,
                                              <s-env:ENV>)>)>)
            is-label-attr(attr)→
                    s-stg:$\mu$(STG;
                            <n:$\mu_0$(<s-at:LABCONST>,
                                <s-dn:$\mu_0$(<s-env:ENV>,
                                              <s-c:
                                                  $\mu$(outerct;<$SUCC_1 \circ$ tn(outerct):
                                                                exit(outerenv);
                                                                  int-st-list(attr)>)>)>)
        for: is-n(n), is-attr(attr), is-c(outerct), is-env(outerenv)
```

Note: tn(control-tree) is the composite selector selecting, in this case, the unique terminal node of control-tree. Thus, the mutation of outerct above has the effect of appending the sub control tree

```
exit(outerenv);
   int-st-list(attr)
```

to the terminal node end of outerct, thus creating a control tree which executes from the labelled statement on. Tn is defined formally in section 2.

```
(I7)    int-st-list(t) =
            is-<>(t)→null
            T→
                    int-st-list(tail(t));
                       int-st(head(t))
        for: is-st-list(t)
```

```
(I8)    int-st(t) =
            is-assign-st(t)→int-assign-st(t)
            is-cond-st(t)→int-cond-st(t)
            is-proc-call(t)&($at_t$ = PROC)→int-proc-call(t)
```

```
          is–block(t)→int–block(t)
          is–goto–st(t)&($at_l$ = LABCONST∨$at_l$ = LABVAR)→int–goto–st(t)
          is–while–st(t)→int–while–st(t)
      where: $at_t$ = s–at((s–id(t)(ENV)(STG),
               $at_l$ = s–at((t(ENV)(STG)
      for: is-st(t)

(I9)  int–assign–st(t) =
          is–var–attr(s–at·$n_t$(STG))→
                       assign($n_t$,v);
                          v:int–expr(s–right–part(t))
          T→error
      where: $n_t$ = (s–left–part(t))(ENV)
      for: is-assign-st(t)

(I10) assign(n,v) =
          s–stg:μ(STG;<s–dn·n:convert(v,s–at·n(STG))>)
      for: is-n(n), is-value(v)

(I11) int–cond–st(t) =
          branch(v,s–then–st(t),s–else–st(t));
             v:int–expr(s–expr(t))
      for: is-cond-st(t)

(I12) branch(v,st1,st2) =
          convert(v,LOG)→int–st(st1)
          ¬convert(v,LOG)→int–st(st2)
      for: is-value(v), is-st(st1), is-st(st2)

(I13) int–proc–call(t) =
          (length($arg–list_t$) = length($p–list_t$))→
                    exit(ENV);
                       int–st($st_t$);
                          establish–env($env_t$,$p–list_t$,$arg–list_t$)
      where: $n_t$ = (s–id(t))(ENV),
               $p–list_t$ = s–param–list·s–attr·s–dn·$n_t$(STG),
               $env_t$ = s–env·s–dn·$n_t$(STG),
               $arg–list_t$ = s–arg–list(t),
               $st_t$ = s–st·s–attr·s–dn·$n_t$(STG)
      for: is-proc-call(t)

(I13') establish–env(env,p–list,arg–list) =
          s–env:μ(env;{<elem(i,p–list):elem(i,arg–list)(ENV)> |
                    1≤i≤length(p–list)}
      for: is-env(env), is-id-list(p-list), is-id-list(arg-list)

(I14) exit(env) =
          s–env:env
      for: is-env(env)

(I15) int–expr(t) =
          is–bin(t)→
```

```
                int-bin-op(s-op(t),a,b);
                   a:int-expr(s-rd2(t));
                      b:int-expr(s-rd1(t))
        is-unary(t)→
                int-un-op(s-op(t),a);
                   a:int-expr(s-rd(t))
        is-funct-des(t)&(at_t = FUNCT)→
                int-funct-call(t)
        is-var(t)&is-var-attr(s-at • n_t(STG))→
                PASS:s-dn • n_t(STG)
        is-const(t)→
                PASS:value(t)
        T→error
    where: n_t = t(ENV),
           at_t = s-at • (s-id(t)(ENV))(STG)
    for: is-expr(t)

(I16)   pass-back(v) = PASS:v
        for: is-value(v)

(I17)   int-funct-call(t) =
            (length(arg-list_t) = length(p-list_t))→
                pass-back(v);
                   exit(ENV);
                      v:int-expr(expr_t);
                         int-st(st_t);
                            establish-env(env_t,p-list_t,arg-list_t)
        where: n_t = (s-id(t))(ENV),
               p-list_t = s-param-list • s-attr • s-dn • n_t(STG),
               env_t = s-env • s-dn • n_t(STG),
               arg-list_t = s-arg-list(t),
               st_t = s-st • s-attr • s-dn • n_t(STG),
               expr_t = s-expr • s-attr • s-dn • n_t(STG)
        for: is-function-des(t)

(I18)   int-goto-st(t) =
            s-env:env_t
            s-c:c_t
        where: dn_t = s-dn • (t(ENV))(STG),
               env_t = s-env(dn_t),
               c_t = s-c(dn_t)
        for: is-goto-st(t)

(I19)   int-while-st(t) =
            loop-or-exit(v,t);
               v:int-expr(s-cond(t))
        for: is-while-st(t)

(I20)   loop-or-exit(v,t) =
            convert(v,LOG)→
                int-while-st(t);
                   int-st(s-body(t))
```

```
        T→null
    for: is-value(v), is-while-st(t)

where (defined):
    un-name =
        PASS:n
    where: n(STG) = Ω

    null =
        PASS:Ω
```

where:

The following functions and instructions are not further specified:

convert(v,attr)	function which yields v converted (if necessary) to the type specified by attr which may either be INT or LOG.
int-bin-op(op,a,b)	Instruction which returns the result of applying the operator op to a and b. It is left open whether there is a conversion performed in case the operator is not applicable to operands of type a and b.
int-un-op(op,a)	Instruction which returns the result of applying the operator op to a (for the problem of conversion see above).
value(a)	Function which yields the value given a constant a.

Appendix III

Abstract Syntax of Program:

(A1) is-progr = is-block
(A2) is-block = (<s-decl-part:is-decl-part>,<s-st-list:is-st-list>)
(A3) is-decl-part = ({<id:is-attr> ‖ is-id(id)})
(A4) is-attr = is-var-attr ∨ is-proc-attr ∨ is-funct-attr ∨ is-label-attr
(A5) is-var-attr = {INT,LOG,LABVAR}
(A6) is-proc-attr = (<s-param-list:is-id-list>,<s-st:is-st>)
(A7) is-funct-attr = (<s-param-list:is-id-list>,<s-st:is-st>,<s-expr:is-expr>)
(A7.5) is-label-attr = is-st-list
(A8) is-st = is-assign-st ∨ is-cond-st ∨ is-proc-call ∨ is-block ∨
    is-goto-st ∨ is-while-st ∨ is-par-block
(A9) is-assign-st = (<s-left-part:is-var>,<s-right-part:is-expr>)
(A10) is-expr = is-cont ∨ is-var ∨ is-funct-des ∨ is-bin ∨ is-unary
(A11) is-const = is-log ∨ is-int
(A12) is-var = is-id
(A13) is-funct-des = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A14) is-bin = (<s-rd1:is-expr>,<s-rd2:is-expr>,<s-op:is-bin-rt>)
(A15) is-unary = (<s-rd:is-expr>,<s-op:is-unary-rt>)
(A16) is-cond-st = (<s-expr:is-expr>,<s-then-st:is-st>,<s-else-st:is-st>)
(A17) is-proc-call = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A18) is-goto-st = id-id
(A19) is-while-st = (<s-cond:is-expr>,<s-body:is-st>)
(A20) is-par-block = (<s-par:is-st-list>)

where:

is-id	an infinite set of identifiers
is-log	a set of constants denoting the truth values
is-int	an infinite set of constants denoting the integer values
is-unary-rt	a set of unary (one-place) operators
is-binary-rt	a set of binary (two-place) operators
{INT,LOG}	two attributes used to distinguish integer variables from logical variables.

Abstract Syntax of State:

(S1) is-state = (<s-c:is-c>,<s-stg:is-stg>)
(S2) is-env = ({<id:is-n> ‖ is-id(id)})
(S3) is-c = ... (standard control trees as defined in LLS70)
(S4) is-stg = ({<n:(<s-dn:is-den>,<s-at:is-type>) ‖ is-n(n)>})
(S5) is-type = {INT,LOG,PROC,FUNCT,LABVAR,LABCONST}
(S6) is-den = is-proc-den ∨ is-funct-den ∨ is-value ∨
    is-label-den ∨ is-UNINIT
(S6.1) is-proc-den = (<s-env:is-env>,<s-attr:is-proc-attr>)

(S6.2) is-funct-den = (<s-env:is-env>,<s-attr:is-funct-attr>)
(S6.3) is-label-den = (<s-c:is-c>)

where:

is-n̂	infinite set of names (used for the generation of unique names)
{PROC, FUNCT}	two attributes used to distinguish function names and procedure names
{s-env,s-c, s-at,s-dn, s-stg}	selectors for the components of the interpreting machine

is-vâlue infinite set of values, the integers and the truth values

The *initial state* for any given program t ∈is-prôgr is:
μ_0(<s-c:**int-progr**(t)>,<s-n:1>)

States ξ whose control part s-c(ξ) is Ω are *end states*.

Abbreviations used in Instruction Schemata:

C	s-c(ξ)
STG	s-stg(ξ)

Instruction Schemata:

```
(I1)    int-progr(t) = int-block(t,Ω)
        for: is-progr(t)

(I2)    int-block(t,env) =
                exit;
                    int-st-list(s-st-list(t),env');
                        int-decl-part(s-decl-part(t),env');
                            env':update-env(s-decl-part(t),env)
        for: is-block(t), is-env(env)

(I3)    update-env(t,env) =
                pass-env(e,env);
                    {id(e):un-name|id(t)!=Ω}
        for: is-decl-part(t), is-env(env)

(I4)    pass-env(e,env) =
                PASS:μ(env;{<id:n>|id(e)≠Ω&id(e)=n}
        for: is-env(e), is-env(env)
```

(I5)

```
int-decl-part(t,env) =
        null;
            {int-decl(id(env),id(t),env)|id(t)≠Ω}
for: is-decl-part(t), is-env(env)
```

(I6)

```
int-decl(n,attr,env) =
        is-var-attr(attr)→
                s-stg:μ(STG;
                        <n:μ_0(<s-at:attr>,
                            <s-dn:UNINIT>)>)
        is-proc-attr(attr)→
                s-stg:μ(STG;
                        <n:μ_0(<s-at:PROC>,
                            <s-dn:μ_0(<s-attr:attr>,
                                      <s-env:env>)>)>)
        is-funct-attr(attr)→
                s-stg:μ(STG;
                        <n:μ_0(<s-at:FUNCT>,
                            <s-dn:μ_0(<s-attr:attr>,
                                      <s-env:env>)>)>)
        is-label-attr(attr)→
                s-stg:μ(STG;
                        <n:μ_0(<s-at:LABCONST>,
                            <s-dn:μ_0
                              <s-c:
                                  μ(C;<pred^2(τ(C):
                                              int-st-list(attr)>)>)>)>)
for: is-n(n), is-attr(attr), is-env(env)
```

Note: τ is the formal parameter of the function $\Phi_{int-decl}$, which is a composite selector selecting the currently executed instruction in s-c(ξ); when the instruction schema for **int-decl** is converted into the definition of $\Phi_{int-decl}$ the τ in the schema ends up being bound by the formal parameter τ in the definition of $\Phi_{int-decl}$. Pred^2(χ) is the composite selector selecting two nodes up from the leaf end of χ. These are defined formally in section 4.

(I7)

```
int-st-list(t,env) =
        is-<>(t)→null
        T→
                int-st-list(tail(t),env);
                   int-st(head(t),env)
for: is-st-list(t)
```

(I8)

```
int-st(t,env) =
        is-assign-st(t)→int-assign-st(t,env)
        is-cond-st(t)→int-cond-st(t,env)
        is-proc-call(t)&(at_s = PROC)→int-proc-call(t,env)
        is-block(t)→int-block(t,env)
        is-goto-st(t)&(at_l = LABCONSTvat_l = LABVAR)→int-goto-st(t,env)
        is-while-st(t)→int-while-st(t,env)
where: at_s = s-at((s-id(t)(env)(STG),
            at_l = s-at((t(env)(STG)
for: is-st(t)
```

```
(I9)    int−assign−st(t,env) =
            is−var−attr(s−at • n_t(STG)→
                            assign(n_t,v);
                                v:int−expr(s−right−part(t))
            T→error
        where: n_t = (s−left−part(t))(env)
        for: is-assign-st(t)

(I10)   assign(n,v) =
            s−stg:μ(STG;<s−dn • n:convert(v,s−at • n(STG))>)
        for: is-n(n), is-value(v)
(I11)   int−cond−st(t,env) =
            branch(v,s−then−st(t),s−else−st(t),env);
                v:int−expr(s−expr(t))
        for: is-cond-st(t), is-env(env)

(I12)   branch(v,st1,st2,env) =
            convert(v,LOG)→int−st(st1,env)
            ¬convert(v,LOG)→int−st(st2,env)
        for: is-value(v), is-st(st1), is-st(st2), is-env(env)

(I13)   int−proc−call(t,env) =
            (length(arg−list_t)=length(p−list_t)→
                        exit;
                            int−st(st_t,env');
                                env':establish−env(env_t,p−list_t,arg−list_t,env)
        where: n_t = (s−id(t))(env),
                p−list_t = s−param−list • s−attr • s−dn • n_t(STG),
                env_t = s−env • s−dn • n_t(STG),
                st_t = s−st • s−attr • s−dn • n_t(STG),
                arg−list_t = s−arg−list(t)
        for: is-proc-call(t)

(I13)   establish−env(env_t,p−list_t,arg−list_t,env) =
            PASS:μ(env_t;{<elem(i,p−list_t):elem(i,arg−list_t)(env)> |
                        1≤i≤length(p−list_t)}
        for: is−env(env_t),is−id−list(p−list_t),is−id−list(arg−list_t),is−env(env)

(I14)   exit = null

(I15)   int−expr(t,env) =
            is−bin(t)→
                        int−bin−op(s−op(t),a,b);
                            a:int−expr(s−rd1(t),env),
                            b:int−expr(s−rd2(t),env)
            is−unary(t)→
                        int−un−op(s−op(t),a);
                            a:int−expr(s−rd(t),env)
            is−funct−des(t)&(at_t = FUNCT)→
                        int−funct−call(t,env)
            is−var(t)&is−var−attr(s−at • n_t(STG))→
                        PASS:s−dn • n_t(STG)
```

```
          is-const(t)→PASS:value(t)
          T→error
     where: $n_t$ = t(env),
               $at_t$ = s-at•(s-id(t)(env))(STG)
     for: is-expr(t), is-env(env)

(I16)  pass-back(v) = PASS:v
     for: is-value(v)

(I17)  int-funct-call(t,env) =
          (length($arg\text{-}list_t$) = length($p\text{-}list_t$))→
                    pass-back(v);
                       exit;
                          v:int-expr($expr_t$,env');
                             int-st($st_t$,env');
                                env:establish-env($env_t$,$p\text{-}list_t$,$arg\text{-}list_t$,env)
     where: $n_t$ = (s-id(t))(env),
               $p\text{-}list_t$ = s-param-list•s-attr•s-dn•$n_t$(STG),
               $env_t$ = s-env•s-dn•$n_t$(STG),
               $arg\text{-}list_t$ = s-arg-list(t),
               $st_t$ = s-st•s-attr•$n_t$(STG),
               $expr_t$ = s-expr•s-attr•s-dn•$n_t$(STG)
     for: is-function-des(t), is-env(env)

(I18)  int-goto-st(t,env) =
          s-c:s-c•s-dn•(t(env))(STG)
     for: is-goto-st(t), is-env(env)

(I19)  int-while-st(t,env) =
          loop-or-exit(v,t,env);
             v:int-expr(s-cond(t),env)
     for: is-while-st(t), is-env(env)

(I20)  loop-or-exit(v,t,env) =
          convert(v,LOG)→
                    int-while-st(t,env);
                       int-st(s-body(t),env)
          T→null
     for: is-value(v), is-while-st(t), is-env(env)

(I21)  int-par-block(t,env) =
          null;
             {int-st(elem(i,s-par(t)),env)|1≤i≤length(s-par(t))}
     for: is-par-block(t)

where (defined):
     un-name =
          PASS:n
     where: n(STG) = Ω

     null =
          PASS:Ω
```

where:

The following functions and instructions are not further specified:

convert(v,attr)	function which yields v converted (if necessary) to the type specified by attr which may either be INT or LOG.
int−bin−op(op,a,b)	Instruction which returns the result of applying the operator op to a and b. It is left open whether there is a conversion performed in case the operator is not applicable to operands of type a and b.
int−un−op(op,a)	Instruction which returns the result of applying the operator op to a (for the problem of conversion see above).
value(a)	Function which yields the value given a constant a.

Appendix IID

Abstract Syntax of Program:

(A1) is-progr = is-block
(A2) is-block = (<s-decl-part:is-decl-list>,<s-st-list:is-st-list>)
(A3) is-decl = (<s-id:is-id>,<s-attr:is-attr>)
(A4) is-attr = is-var-attr ∨ is-proc-attr ∨ is-funct-attr ∨ is-label-attr
(A5) is-var-attr = {INT,LOG,LABVAR}
(A6) is-proc-attr = (<s-param-list:is-id-list>,<s-st:is-st>)
(A7) is-funct-attr = (<s-param-list:is-id-list>,<s-st:is-st>,<s-expr:is-expr>)
(A7.5) is-label-attr = is-st-list
(A8) is-st = is-assign-st ∨ is-cond-st ∨ is-proc-call ∨ is-block ∨
is-goto-st ∨ is-while-st
(A9) is-assign-st = (<s-left-part:is-var>,<s-right-part:is-expr>)
(A10) is-expr = is-cont ∨ is-var ∨ is-funct-des ∨ is-bin ∨ is-unary
(A11) is-const = is-log ∨ is-int
(A12) is-var = is-id
(A13) is-funct-des = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A14) is-bin = (<s-rd1:is-expr>,<s-rd2:is-expr>,<s-op:is-bin-rt>)
(A15) is-unary = (<s-rd:is-expr>,<s-op:is-unary-rt>)
(A16) is-cond-st = (<s-expr:is-expr>,<s-then-st:is-st>,<s-else-st:is-st>)
(A17) is-proc-call = (<s-id:is-id>,<s-arg-list:is-id-list>)
(A18) is-goto-st = id-id
(A19) is-while-st = (<s-cond:is-expr>,<s-body:is-st>)

where:

is-id	an infinite set of identifiers
is-log	a set of constants denoting the truth values
is-int	an infinite set of constants denoting the integer values
is-unary-rt	a set of unary (one-place) operators
is-binary-rt	a set of binary (two-place) operators
{INT,LOG}	two attributes used to distinguish integer variables from logical variables.

Abstract Syntax of State Components:

(S2) is-env = ({<id:is-n> ‖ is-id(id)})
(S4) is-stg = ({<n:(<s-dn:is-den>,<s-at:is-type>) ‖ is-n(n)>})
(S5) is-type = {INT,LOG,PROC,FUNCT,LABVAR,LABCONST}
(S6) is-den = is-proc-den ∨ is-funct-den ∨ is-value ∨
is-label-den ∨ is-UNINIT
(S6.1) is-proc-den = (<s-env:is-env>,<s-attr:is-proc-attr>)
(S6.2) is-funct-den = (<s-env:is-env>,<s-attr:is-funct-attr>)
(S6.3) is-label-den = (<s-env:is-env>,<s-c:is-in-Cont>)
is-in-Cont(p) = p ∈ Cont

where:

is-n̂	infinite set of names (used for the generation of unique names)
{PROC, FUNCT}	two attributes used to distinguish function names and procedure names
{s-env,s-c, s-at,s-dn, s-stg}	selectors for the components of the interpreting machine
is-v̂alue	infinite set of values, the integers and the truth values

Note that predicates of the form

is-t = {<d:is-r>‖is-d(d)}

describe tables, i.e., functions on is-d̂ to is-r̂. In fact, For all t such that is-t(t), There exists a unique f ∈ [is-d̂→is-r̂] such that for all d such that is-d(d),

d(t) = f(d);

f is said to be the function represented by the table t (f is unique since for all d' not appearing explicitly in t, d(t') is taken as Ω). In addition,

μ(t;<d:r>)

represents the function

f[d/r].

Thus, the elements of

is-ênv

represent the elements of

[is-îd→is-n̂],

and the elements of

is-ŝtg

represent the elements of

[is-n̂→(<s-dn:is-den>,̂<s-at:is-type>)].

In denotational semantics, function application is denoted by juxtaposition of the function name with its arguments. In VDL, function application is denoted by the function name followed by a parenthesized argument list. When calculating values of arguments to the denotational meaning function and to a continuation, the VDL convention is used, but when applying the denotational meaning function or a continuation, the denotational semantics convention is used.

Semantic Equations:

In the following,
is-env(**ENV**), is-in-Cont(ρ), is-stg(**STG**),
is-env(*env*), is-stg(*stg*), is-value(v), and likewise
for any primed version of these symbols.

(I1) *Mean* ⟦**int−progr(t)**⟧ **ENV** ρ **STG** =
Mean ⟦**int−block(t)**⟧ **ENV** ρ **STG**
for: is-progr(t)

(I2) *Mean* ⟦**int−block(t)**⟧ **ENV** ρ **STG** =
Mean ⟦**update−env(s−decl−part(t))**⟧ **ENV** ρ' **STG**
where ρ' v *env* *stg* =

```
                Mean ⟦dsle(s-decl-part(t),s-st-list(t),ρ,ENV)⟧ env ρ stg
        for: is-block(t)

(I2')   Mean ⟦dsle(dp,sl,outerquf,outerenv)⟧ ENV ρ STG =
            Mean ⟦int-decl-part(dp,outerquf,outerenv)⟧ ENV ρ' STG
        whereρ' v env stg =
                Mean ⟦sle(sl,outerenv)⟧ env ρ stg
        for:is-decl-part(dp), is-st-list(sl), is-in-Cont(outerquf), is-env(outerenv)

(I2")   Mean ⟦sle(sl,outerenv)⟧ ENV ρ STG =
            Mean ⟦int-st-list(sl)⟧ ENV ρ' STG
        whereρ' v env stg =
                Mean ⟦exit(outerenv)⟧ env ρ stg
        for:is-st-list(sl), is-env(outerenv)

(I3)    Mean ⟦update-env(t)⟧ ENV ρ STG =
            is-<>(t)→Mean ⟦null⟧ ENV ρ STG
            T→Mean ⟦un-name⟧ ENV ρ' STG
        whereρ' v env stg =
                Mean ⟦ie(s-id(head(t)),v,tail(t))⟧ env ρ stg
        for: is-decl-part(t)

(I3')   Mean ⟦ie(is,n,dp)⟧ ENV ρ STG =
            Mean ⟦update-id(id,n)⟧ ENV ρ' STG
        whereρ' v env stg =
                Mean ⟦update-env(dp)⟧ env ρ stg
        for: is-id(id), is-n(n), is-decl-part(dp)

(I4)    Mean ⟦update-id(id,n)⟧ ENV ρ STG =
            ρ Ω μ(ENV;<id:n>) STG
        for: is-id(id), is-n(n)

(I5)    Mean ⟦int-decl-part(t,outerquf,outerenv)⟧ ENV ρ STG =
            is-<>(t)→Mean ⟦null⟧ ENV ρ STG
            T→
                   Mean ⟦int-decl(s-id(head(t))(ENV),s-attr(head(t)),outerquf,
                             outerenv)⟧ ENV ρ' STG
        where: ρ' v env stg =
                Mean ⟦int-decl-part(tail(t),outerquf,outerenv)⟧ env ρ stg
        for: is-decl-part(t), is-in-Cont(outerquf), is-env(outerenv)

(I6)    Mean ⟦int-decl(n,attr,outerquf,outerenv)⟧ ENV ρ STG =
            is-var-attr(attr)→
                   ρ Ω ENV μ(STG;
                                  <n:μ₀(<s-at:attr>,
                                        <s-dn:UNINIT>)>)
            is-proc-attr(attr)→
                   ρ Ω ENV μ(STG;
                                  <n:μ₀(<s-at:PROC>,
                                        <s-dn:μ₀(<s-attr:attr>,
                                                      <s-env:ENV>)>)>)
            is-funct-attr(attr)→
```

```
                ρ Ω ENV μ(STG;
                                <n:$\mu_0$(<s–at:FUNCT>,
                                    <s–dn:$\mu_0$(<s–attr:attr>,
                                                   <s–env:ENV>)>)>)
        is–label–attr(attr)→
                ρ Ω ENV μ(STG;
                                <n:$\mu_0$(<s–at:LABCONST>,
                                    <s–dn:$\mu_0$(<s–env:ENV>,
                                                   <s–c:ρ'>)>)>)
    where: ρ' v env stg =
                Mean ⟦sle(attr,outerenv)⟧ env outerquf stg
    for: is-n(n), is-attr(attr), is-in-Cont(outerquf), is-env(outerenv)
```

```
(I7)    Mean ⟦int–st–list(t)⟧ ENV ρ STG =
            is–<>(t)→Mean ⟦null⟧ ENV ρ STG
            T→
                Mean ⟦int–st(head(t))⟧ ENV ρ' STG
        where: ρ' v env stg =
                Mean ⟦int–st–list(tail(t))⟧ env ρ stg
        for: is-st-list(t)
```

```
(I8)    Mean ⟦int–st(t)⟧ ENV ρ STG =
            is–assign–st(t)→Mean ⟦int–assign–st(t)⟧ ENV ρ STG
            is–cond–st(t)→Mean ⟦int–cond–st(t)⟧ ENV ρ STG
            is–proc–call(t)&($at_t$ = PROC)→
                Mean ⟦int–proc–call(t)⟧ ENV ρ STG
            is–block(t)→Mean ⟦int–block(t)⟧ ENV ρ STG
            is–goto–st(t)&($at_1$ = LABCONST∨$at_1$ = LABVAR)→
                Mean ⟦int–goto–st(t)⟧ ENV ρ STG
            is–while–st(t)→Mean ⟦int–while–st(t)⟧ ENV ρ STG
        where: $at_t$ = s–at((s–id(t)(ENV)(STG),
                  $at_1$ = s–at((t(ENV)(STG)
        for: is-st(t)
```

```
(I9)    Mean ⟦int–assign–st(t)⟧ ENV ρ STG =
            is–var–attr(s–at • $n_t$(STG))→
                        Mean ⟦int–expr(s–right–part(t))⟧ ENV ρ' STG
            T→error
        where: ρ' v env stg =
                Mean ⟦assign($n_t$,v)⟧ env ρ Stg,
             $n_t$ = (s–left–part(t))(ENV)
        for: is-assign-st(t)
```

```
(I10)   Mean ⟦assign(n,v)⟧ ENV ρ STG =
            ρ Ω ENV μ(STG;<s–dn • n:convert(v,s–at • n(STG))>)
        for: is-n(n), is-value(v)
```

```
(I11)   Mean ⟦int–cond–st(t)⟧ ENV ρ STG =
            Mean ⟦int–expr(s–expr(t))⟧ ENV ρ' STG
        where: ρ' v env stg =
                Mean ⟦branch(v,s–then–st(t),s–else–st(t))⟧ env ρ stg
        for: is-cond-st(t)
```

(I12) *Mean* ⟦**branch(v,st1,st2)**⟧ **ENV** ρ **STG** =
convert(v,LOG)→*Mean* ⟦**int−st(st1)**⟧ **ENV** ρ **STG**
¬**convert(v,LOG)**→*Mean* ⟦**int−st(st2)**⟧ **ENV** ρ **STG**
for: is-value(v), is-st(st1), is-st(st2)

(I13) *Mean* ⟦**int−proc−call(t)**⟧ **ENV** ρ **STG** =
(**length**(**arg−list**$_t$) = **length**(**p−list**$_t$))→
Mean ⟦**establish−env**(**env**$_t$,**p−list**$_t$,**arg−list**$_t$)⟧ **ENV** ρ′ **STG**
where: ρ′ v *env stg* =
Mean ⟦**se**(**st**$_t$,**ENV**)⟧ *env* ρ′ **Stg**,
n$_t$ = (**s−id(t)**)(**ENV**),
p−list$_t$ = **s−param−list** • **s−attr** • **s−dn** • **n**$_t$(**STG**),
env$_t$ = **s−env** • **s−dn** • **n**$_t$(**STG**),
arg−list$_t$ = **s−arg−list(t)**,
st$_t$ = **s−st** • **s−attr** • **s−dn** • **n**$_t$(**STG**)
for: is-proc-call(t)

(I13’) *Mean* ⟦**establish−env(env,p−list,arg−list)**⟧ **ENV** ρ **STG** =
ρ Ω μ(**env**;{<**elem(i,p−list)**:**elem(i,arg−list)(ENV)**> |
1≤i≤**length(p−list)**} **STG**
for: is-env(env), is-id-list(p-list), is-id-list(arg-list)

(I13") *Mean* ⟦**se(st,outerenv)**⟧ **ENV** ρ **STG** =
Mean ⟦**int−st(st)**⟧ **ENV** ρ′ **STG**
where: ρ′ v *env stg* =
Mean ⟦**exit(outerenv)**⟧ *env* ρ *stg*
for: is-st(st), is-env(outerenv)

(I14) *Mean* ⟦**exit(env)**⟧ **ENV** ρ **STG** =
ρ Ω **env STG**
for: is-env(env)

(I15) *Mean* ⟦**int−expr(t)**⟧ **ENV** ρ **STG** =
is−bin(t)→
Mean ⟦**int−expr(s−rd1(t))**⟧ **ENV** ρ″ **STG**
is−unary(t)→
Mean ⟦**int−expr(s−rd(t))**⟧ **ENV** ρ′ **STG**
is−funct−des(t)&(**at**$_t$ = **FUNCT**)→
Mean ⟦**int−funct−call(t)**⟧ **ENV** ρ **STG**
is−var(t)&is−var−attr(s−at • **n**$_t$(**STG**))→
ρ **s−dn** • **n**$_t$(**STG**) **ENV STG**
is−const(t)→
ρ **value(t) ENV STG**
T→error
where: ρ″ **a** *env stg* =
Mean ⟦**eb(s−rd2(t),s−op(t),a)**⟧ *env* ρ **Stg**,
ρ′ **a** *env stg* =
Mean ⟦**int−un−op(s−op(t),a)**⟧ *env* ρ **Stg**,
n$_t$ = **t(ENV)**,
at$_t$ = **s−at** • (**s−id(t)(ENV)**)(**STG**)
for: is-expr(t)

```
(I15')  Mean ⟦eb(rd2,op,a)⟧ ENV ρ STG =
            Mean ⟦int−expr(rd2)⟧ ENV ρ' STG
        where: ρ' b env stg =
                    Mean ⟦int−bin−op(op,a,b)⟧ env ρ stg
        for: is-expr(rd2), is-bin-op(op), is-value(a)

(I16)   Mean ⟦pass−back(v)⟧ ENV ρ STG =
            ρ v ENV STG
        for: is-value(v)

(I17)   Mean ⟦int−funct−call(t)⟧ ENV ρ STG =
            (length(arg−list_t) = length(p−list_t))→
                        Mean ⟦establish−env(env_t,p−list_t,arg−list_t)⟧ ENV ρ' STG
        where: ρ' v env stg =
                    Mean ⟦seep(st_t,expr_t,ENV)⟧ env ρ Stg,
                n_t = (s−id(t))(ENV),
                p−list_t = s−param−list ∘ s−attr ∘ s−dn ∘ n_t(STG),
                env_t = s−env ∘ s−dn ∘ n_t(STG),
                arg−list_t = s−arg−list(t),
                st_t = s−st ∘ s−attr ∘ s−dn ∘ n_t(STG),
                expr_t = s−expr ∘ s−attr ∘ s−dn ∘ n_t(STG)
        for: is-function-des(t)

(I17')  Mean ⟦seep(st,expr,outerenv)⟧ ENV ρ STG =
            Mean ⟦int−st(st)⟧ ENV ρ' STG
        where: ρ' v env stg =
                    Mean ⟦eep(expr,outerenv)⟧ env ρ stg
        for: is-st(st), is-expr(expr), is-env(outerenv)

(I17")  Mean ⟦eep(expr,outerenv)⟧ ENV ρ STG =
            Mean ⟦int−expr(expr)⟧ ENV ρ' STG
        where: ρ' v env stg =
                    Mean ⟦ep(outerenv,v)⟧ env ρ stg
        for: is-expr(expr), is-env(outerenv)

(I17"') Mean ⟦ep(outerenv,val)⟧ ENV ρ STG =
            Mean ⟦exit(outerenv)⟧ ENV ρ' STG
        where: ρ' v env stg =
                    Mean ⟦pass−back(val)⟧ env ρ stg
        for: is-env(outerenv), is-value(val)

(I18)   Mean ⟦int−goto−st(t)⟧ ENV ρ STG =
            c_t Ω env_t STG
        where: dn_t = s−dn ∘ (t(ENV))(STG),
               env_t = s−env(dn_t),
               c_t = s−c(dn_t)
        for: is-goto-st(t)

(I19)   Mean ⟦int−while−st(t)⟧ ENV ρ STG =
            Mean ⟦int−expr(s−cond(t))⟧ ENV ρ' STG
        where: ρ' v env stg =
                    Mean ⟦loop−or−exit(v,t)⟧ env ρ stg
```

for: is-while-st(t)
(I20) *Mean* ⟦loop−or−exit(v,t)⟧ ENV ρ STG =
convert(v,LOG)→
Mean ⟦int−st(s−body(t))⟧ ENV ρ' STG
T→*Mean* ⟦null⟧ ENV ρ STG
where: ρ' val *env stg* =
Mean ⟦int−while−st(t)⟧ *env* ρ *stg*
for: is-value(v), is-while-st(t)

where (defined):
Mean ⟦un−name⟧ ENV ρ STG =
ρ n ENV STG
where: n(STG) = Ω

Mean ⟦null⟧ ENV ρ STG =
ρ Ω ENV STG

Mean ⟦int−bin−op(op,a,b)⟧ ENV ρ STG =
ρ (a op b) ENV STG
where: (a op b) is the result of applying the operator op to a and b.
It is left open whether there is a conversion performed in case the operator is not applicable to operands of type a and b.
for: is-binary-rt(op), is-value(a), is-value(b)

Mean ⟦int−un−op(op,a)⟧ ENV ρ STG =
ρ (op a) ENV STG
where: (op a) is the result of applying the operator op to a.
It is left open whether there is a conversion performed in case the operator is not applicable to an operand of type a.
for: is-unary-rt(op), is-value(a)

where:
The following functions and instructions are not further specified:

convert(v,attr)	function which yields v converted (if necessary) to the type specified by attr which may either be INT or LOG.
value(a)	Function which yields the value given a constant a.

Appendix IV

Syntactic Domains:

(A1) is-Exp = is-0 ∨ is-1 ∨ is-neg ∨ is-not ∨ is-add ∨ is-equal ∨
is-Ide ∨ is-procedure
(A2) is-neg = (<s-neg:is-Exp>)
(A3) is-not = (<s-not:is-Exp>)
(A4) is-add = (<s-add1:is-Exp>,<s-add2:is-Exp>)
(A5) is-equal = (<s-equal1:is-Exp>,<s-equal2:is-Exp>)
(A6) is-Ide = · · ·
(A7) is-procedure = (<s-procedure:is-Com>)

(A8) is-Def = is-var-def ∨ is-const-def
(A9) is-var-def = (<s-new:is-Ide>,<s-init:is-Exp>)
(A10) is-const-def = (<s-val:is-Ide>,<is-init:is-Exp>)

(A11) is-Com = is-NULL ∨ is-assign ∨ is-call ∨ is-semic ∨ is-if ∨
is-while ∨ is-with ∨ is-labeled-Com ∨ is-Seq
(A12) is-assign = (<s-lp:is-Ide>,<s-rp:is-Exp>)
(A13) is-call = (<s-call:is-Exp>)
(A14) is-semic = (<s-first:is-Com>,<is-second:is-Com>)
(A15) is-if = (<s-if:is-Exp>,<s-then:is-Com>,<s-else:is-Com>)
(A16) is-while = (<s-while:is-Exp>,<s-do:is-Com>)
(A17) is-with = (<s-with:is-Def>,<s-do:is-Com>)
(A18) is-labeled-Com = (<s-label:is-Ide>,<s-com:is-Com>)

(A19) is-Seq = (<s-goto:is-Ide>)

(A20) is-Pro = (<s-output:is-Ide>,<s-program:is-Com>)

Semantic Domains:
(S1) is-ˆT = {TRUE,FALSE}
(S2) is-ˆZ = { · · · ,-2,-1,0,1,2, · · · }
(S3) is-B = is-T ∨ is-B
(S4) is-R = is-B ∨ is-P
(S5) is-L = · · · (locations)
(S6) is-S = ({<l:is-R ∨ is-UNUSED>||is-L(l)})
(S7) is-D = is-L ∨ is-R ∨ is-c ∨ is-UNDEFINED
(S8) is-U = ({<I:is-D>||is-Ide(I)})
(S9) is-P = is-c

(S10) is-state = (s-U:is-U>,<s-S:is-S>,<s-c:is-c>)
(S11) is-c = · · · (control trees)

(S12) is-initial-state = (<s-U:is-initial-U>,<s-S:is-initial-S>,
<s-c:**int−Pro**(P,B)>
where: is-Pro(P), is-B(B)

(S13) is-initial-U({<I:is-UNDEFINED>|is-Ide(I)})

(S14) is-initial-S({<l:is-UNUSED>|is-L(l)})

Abbreviations used in Instruction Schemata:

u	s-U(ξ)
s	s-S(ξ)
c	s-c(ξ)

Instruction Schemata:

```
(I1)    int−Exp(E) =
            is−0(E)→
                    PASS:0
            is−1(E)→
                    PASS:1
            is−neg(E)→
                    test−r−is−Z−and−negate(r);
                       r:int−Exp(s−neg(E))
            is−not(E)→
                    test−r−is−T−and−not(r);
                       r:int−Exp(s−not(E))
            is−add(E)→
                    test−r2−is−Z−and−add($r_1$,$r_2$);
                       $r_2$:test−r1−is−Z−and−int−Exp($r_1$,s−add2(E));
                          $r_1$:int−Exp(s−add1(E))
            is−equal(E)→
                    test−r2−is−T−and−equal($r_1$,$r_2$);
                       $r_2$:test−r1−is−T−and−int−Exp($r_1$,s−equal2(E));
                          $r_1$:int−Exp(s−equal1(E))
            is−Ide(E)→
               is−L(E(u))→PASS:(E(u))(s)
               is−R(E(u))→PASS:E(u)
            is−procedure(E)→
                    PASS:int−Com(s−procedure(E));
                                establish−env(u)
        for: is-Exp(E)
```

```
(I2)    test−r−is−Z−and−negate(r) =
            is−Z(r)→PASS: ¬r
            T→error
        for: is-R(r)
```

```
(I3)    test−r−is−T−and−not(r) =
            is−T(r)→PASS: −r
            T→error
        for: is-R(r)
```

```
(I4)    test−r2−is−Z−and−add($r_1$,$r_2$) =
            is−Z($r_2$)→PASS:$r_1$+$r_2$
            T→error
```

```
        for: is-R($r_1$), is-R($r_2$)

(I5)    test−r1−is−Z−and−int−Exp($r_1$,E) =
                is−Z($r_1$)→int−Exp(E)
                T→error
        for: is-R($r_1$), is-Exp(E)

(I6)    test−r2−is−T−and−equal($r_1$,$r_2$) =
                is−T($r_2$)→PASS:$r_1$=$r_2$
                T→error
        for: is-R($r_1$), is-R($r_2$)

(I7)    test−r1−is−T−and−int−Exp($r_1$,E) =
                is−T($r_1$)→int−Exp(E)
                T→error
        for: is-R($r_1$), is-Exp(E)

(I8)    int−Def(D) =
                is−var−def(D)→
                                alloc−assoc(s−new(E),r);
                                    r:int−Exp(s−init(E))
                is−const−def(D)→
                                assoc(s−val(E),r);
                                    r:int−Exp(s−init(E))
        for: is-Def(D)

(I9)    alloc−assoc(I,r) =
                (∃l′)(is−UNUSED(l′(u)))→
                                s−S:μ(s;<l:r>)
                                s−U:μ(u;<I:l>)
                T→error
        where: is−UNUSED(l(u))
        for: is-Ide(I), is-R(r)

(I10)   assoc(I,r) =
                s−U:μ(u;<I,r>)
        for: is-Ide(I), is-R(r)

(I11)   int−Com(C) =
                is−NULL(C)→
                                null
                is−assign(C)→
                                assign(s−lp(C),r);
                                    r:int−Exp(s−rp(C))
                is−call(C)→
                                call(r);
                                    r:int−Exp(s−call(C))
                is−semic(C)→
                                int−Com(s−second(C));
                                    int−Com(s−first(C))
                is−if(C)→
                                branch(r,s−then(C),s−else(C));
```

```
                    r:int-Exp(s-if(C))
        is-while(C)→
                loop-or-exit(r,C);
                    r:int-Exp(s-while(c))
        is-with(C)→
                establish-env(u);
                    int-Com(s-do(C));
                        int-Def(s-with(C))
        is-labeled-Com(C)→
                establish-env(u);
                    int-Com(s-com(C));
                        assoc(s-label(C),c')
    where: c' =
                c;
                    int-Com(C)
    for: is-Com(C)
```

```
(I12)   null = PASS:Ω
```

```
(I13)   assign(I,r) =
                is-L(I(u))→s-S:μ(s;<I(u):r>)
                T→error
        for: is-Ide(I), is-R(r)
```

```
(I14)   call(r) =
                is-P(r)→
                        establish-env(u);
                            r
                T→error
        for: is-R(r)
```

```
(I15)   branch(r,$C_1$,$C_2$) =
                is-T(r)→
                    r→int-Com($C_1$)
                    T→int-Com($C_2$)
                T→error
        for: is-R(r), is-Com($C_1$), is-Com($C_2$)
```

Note that the above instruction and the next instruction deviate slightly from Tennent's semantics in that they both check that r is in the domain is-T. This checking is consistent with the rest of the instructions and the rest of Tennent's semantics.

```
(I16)   loop-or-exit(r,C) =
                is-T(r)→
                    r→
                            int-Com(C);
                                int-Com(s-do(C))
                    T→
                            null
                T→error
        for: is-R(r), is-while(C)
```

```
(I17)   establish-env(U) =
                s-U:U
        for: is-U(U)

(I18)   int-Seq(S) =
                is-c((s-goto(S))(u))-s-c:(s-goto(S))(u)
                T-error
        for: is-Seq(S)

(I19)   int-Pro(P,B) =
                establish-env(u);
                    int-Com(s-program(P));
                        alloc-assoc(s-output(P),B)
        for: is-Pro(P), is-B(B)
```