From Standard to Implementation Denotational Semantics

Martin Raskovsky
and Phil Collier
January 1980

Under the Sun of Menorca

Computing Science
University of Essex
England

Abstract

We are developing a compiler compiler. It takes as input the formal definition of a programming language in Denotational Semantics and produces as output a fairly efficient compiler written in a systems programming language which in turn will produce code for a real machine. This work mainly deals with the code generation parts.

Contents

From Standard to Implementation Denotational Semantics

0. Introduction

According to the tradition of denotational/functional semantics(DS) of programming languages(PL), the meaning/referent of a program is a function from states to states of some underlying store. Compilers, on the other hand, generate code for some machine given the program as input. On the face of it these enterprises - writing semantic equations in the Scott-Strachey style and compiler-writing -seem very different. The former requires a certain degree of mathematical sophistication and the latter a certain competence as a system programmer. Moreover, the end products of the two enterprises appear to be different; on the one hand the result is a precise mathematical description of the programming language and on the other one ends up with a set of procedures which constitute a compiler for the language in question. In the original literature on functional semantics[Scott70] it was claimed that the semantic equations should serve as a guide for the compiler writer; any notion of correctness for the compiler would have to make reference to the semantic equations.

We believe that semantic equations can provide us with the information required to implement a compiler for the language in question. In other words, the DS of a particular PL at an appropriate level of abstraction could embody detailed information about control, environment and state serving as an input for a compiler-compiler.

Let's try and make this a little clearer. Given a DS equation, the process of code generation closely follows its text. For example the equation for the assignment command in a particular PL could be:

$$\mathbf{C}[\![e:=e']\!]\rho c = \mathbf{R}[\![e']\!]\rho(\lambda v.\mathbf{L}[\![e]\!]\rho(\lambda l.\text{Update } l v c)).$$

While reading this equation we can imagine a compiler that, in the presence of an environment(ρ), will generate code first to find a value, then to find a location, then to update the store and finally to jump if necessary to the given continuation. In fact, it is reasonable to argue that DS has formalised, at an appropriate level of abstraction, the behaviour of a program. A compiler needs to understand the behaviour called for by a program in order to plant code to execute it. Thus it is reasonable to conjecture that there may be a close relationship between semantic equations and compilers. If one could automate the production of a compiler from semantic equations then this could be viewed as a kind of compiler-compiler.

Indeed it is a part of our conjecture concerning the relationship between compilers and semantic equations that not only could the semantic equations dictate the structure of the compiler but

0. Introduction

conversely intuitions and experience of compiler writers should influence the DS equations themselves.

However, we do appreciate the need to have a 'standard' denotational semantics without any bias towards implementation ideas. So we propose to distinguish between two different forms of DS which for any particular language we shall have to prove congruent, namely:

Standard Denotational Semantics(SDS): A canonical definition free of bias towards any particular implementation.

Implementation Denotational Semantics(IDS): Embodying all implementation strategies desired.

We have developed a translation process which, starting from the IDS equations of a simple PL, generates a number of BCPL procedures[Richards69][BCPL77]. We have written the primitive functions and machine code interface to produce code for the DEC-10 system. To generate the parser an appropriate generator is interfaced[Sufrin77]

The final result is the following:

-An efficient compiler using standard compiling-techniques.
-Efficient code generated.
-Flexibility to implement different 'styles' of code.
-Flexibility to implement in different machines.

Research related to ours is the work by [Mosses74-5-6-8] and [Jones80] where semantic equations are uniformly translated to an intermediate code which in turn is interpreted. Both systems achieve generality by explicity separating the concept of a correct compiler from that of a useful one, and it is instructive to see why. While uniformly encoding all 'functions' and 'values', both methods throw away a great deal of the knowledge contained in the semantic equations. For example the [L ➤ V] function is simulated, instead of making 'natural' use of the machine store; and the familiar and useful concept of a pointer to the code is nonexistent. Standard semantic equations as developed by the Oxford school provide us with certain information about how to implement the language efficiently and this is lost in a uniform translation. These observations have been paramount in guiding our translation process.

The first step of our work was to consider simple language constructs and to implement a translator accepting languages whose characteristics were similar to those. In Section 1 of this paper we describe a language within this framework. We briefly motivate its Implementation Denotational Semantics and then we informally describe the transformation process to generate a compiler for it. Appendices B, C, D and E show respectively its Syntax, SDS, IDS and Compiler.

Currently we are enlarging the set of programming constructs accepted by our system. In Section 2 we consider recursion showing the impact that this feature has in the domains of its IDS specification. Finally we show how to prove the congruence between both SDS and IDS definitions.

1. A simple case study. TL: a Toy Language

The toy language(TL) chosen contains a number of basic commands like assignment, conditional, while, parameter-less procedure call, structured jumps (BREAK, LOOP, RETURN), input/output and blocks. It has arithmetic and boolean expressions, as well as data types: integers and procedures.

Recursion is not allowed, so that every storage location can be determined at compilation time (Static storage allocation). Moreover, a crucial attribute of such a language, which amounts to a simplification relative to ALGOL-like languages, LISP or SIMULA, is that for every procedure, a single unique data area[Gries71] (Analogous to an ALGOL stack frame or template[Bornat76-7]) can be set at compile time. In this respect TL resembles FORTRAN.

1.1 The Implementation Denotational Semantics of TL

In this section we will briefly motivate the IDS of TL. The three issues we discuss are the meaning of boolean expressions, the allocation of locations and efficiency in arithmetic expressions. The complete SDS and IDS specifications can be found in Appendices A, B, C and D.

1. A simple case study, TL: a Toy Language

1.1.1 Boolean Expressions

Consider a fragment of the syntax of boolean expressions in TL:

```
b ::= b' "and" b'' | b' "or" b'' | "true" | "false" | ...
```

and the corresponding semantic equations:

```
B:[Bex > U > S > T].

B[b' "and" b'']p=
  B[b']p @ Cond<B[b'']p,λs.FALSE>.

B[b' "or" b'']p=
  B[b']p @ Cond<λs.TRUE,B[b'']p>.

B["true"]ps=
  TRUE.

B["false"]ps=
  FALSE.
```

Boolean expressions viewed in this way are like any other expression with the exception that they evaluate to boolean values. But it happens that these can be evaluated in a completely different way. The evaluation of a boolean expression need not produce a value but can select the next path of the computation. This is exactly how Cond can be thought to behave: given two expressions, it picks one on the basis of a given boolean value.

To model this behaviour we redefine the function B, as a semantic valuator taking two continuations, one to be applied if the supplied boolean expression evaluates to true, and another if it evaluates to false.

```
B:[Bex > U > C > C > C].

B[b' "and" b'']pcc'=
  B[b']p(B[b'']pcc')c'.

B[b' "or" b'']pcc'=
  B[b']pc(B[b'']pcc').

B["true"]pcc'=
  c.

B["false"]pcc'=
  c'.
```

The only time that a truth value is really produced is when a general expression contains boolean subexpressions.

This model of boolean expressions with two continuations corresponds precisely to a way that efficient compilers implement them, namely as true and false chains.

1.1.2 Marking locations 'in use'

Consider now the allocation of locations in the SDS of TL:

```
s:S=[[L > V] x V* x R* x [L > T]].        States
p:U=[[Ide > D] x C x C x C].              environments
New:[S > [L x S]].

C["begin" d;c' "end"]pcs=
  (D[d]p * λp'.C[c']p'(λs'.c<s'↓1,s'↓2,s'↓3,s↓4>))s.

D["integer" i]p=
  New * λls.<p[[i]/l],s>.
```

The function 'New', which obtains unused locations when necessary, seems to be abstracting a 'free storage' mechanism which is not the one dictated by a block structured discipline. Also the location deallocation mechanism, where the area function [L > T] indicates which locations are in use, is not satisfactory from an implementation point of view. (The area function is stored on entry to each block, so that it can be restored on exit.) It would seem reasonable that locations be marked 'in use' in the environment allowing 'automatic' deallocation of locations at the end of a block, as environments, and therefore details of the amount of storage in use are bound into the continuation following the end of the block.

Accordingly, we rewrite in IDS the SDS definitions as follows:

```
a:A=[L > T].                          Area function
s:S=[[L > V] x V* x R*].              States
p:U=[[Ide > D] x C x C x C x A].      environments
New:[U > [L x U]].

C[["begin" d;c' "end"]]pc=
  C[[c']](D[[d]]p)c.

D[["integer" i]]p=
  New p=>λ<l,p'>.p'[[[i]]/l].
```

1.1.3 Crucial code fragments: Expressions

In order to claim to be producing an efficient compiler, we must ensure that expressions are compiled into efficient code. For example the semantic equation for arithmetic expressions in SDS is:

```
R[[e'ae'']]p=
  R[[e']]p @ λv.(R[[e'']]p o A[[a]]v).
```

If we leave this as it stands the corresponding fragment of the generated compiler will be:

```
CASE Exp.Aop:
  RR(First OF Node, Reg)
  RR(Third OF Node, NextReg(Reg))
  AA(Second OF Node, Reg, NextReg(Reg))
  ENDCASE
```

where RR and AA are the generated procedures to plant code respectively for expressions and operators. If we forget about restrictions on the maximun number of registers available, this procedure will work but will produce very inefficient code. For example

```
a + b * 10
```

will generate the following DEC-10 code:

```
MOVE    1,a
MOVE    2,b
MOVEI   3,10
IMUL    2,3
ADD     1,2
```

but we can do better:

```
MOVE    1,b
IMULI   1,10
ADD     1,a
```

To generate this code a better algorithm can easily be implemented; we will follow the one given in [Bornat77]:

```
R[e'ae'']p=
  IfNeedToReverse[e'ae'']>R(Reverse[e'ae'']|Exp)p,
  (R[e']p @
   λv.IsLeaf[e'']>RLeaf[e''][a]pv,
      IfNeedToDump[e'']>
      Dump pv * λ<l,p'>.(R[e'']p' @ λv.A(Reverse[a]|Aop)vl),
      (R[e'']p @ A[a]v)).
```

In fact, this equation abstracts the 'register-allocation' technique of 'tree weighting' and 'dumping'.

1.2 The translation

We now describe the translation that takes as input the IDS equations and produces BCPL procedures which are the code generation part of a compiler for the language defined. (These procedures can be found in Appendix E.)

1.2.1 Removal of the State

In the first stage of the translation process we remove all references to the state. This is in keeping with the fact that a state to state transformation is a function performed by the code generated together with the hardware of a particular machine. The compiler is performing a translation that ends one step behind the state to state function. As an example, consider the semantic equation for assignment:

```
C[i:=e]pc=
  R[e]p @ Assign(p[i]|L) o c.
```

For a detailed specification of the operators refer to Apendix A. After the analysis of the operator @ we end up with the following procedural text:

```
LET C[i:=e]pc BE
{ Assign(p[i]|L)(R[e]p) o c
}
```

To emphasize that this is not a mathematical equation we enclose the new procedural-text within curly brackets.

The analysis of the operator o will in fact produce two statements, and after uncurrying the assignment example will now look like:

```
LET C([i:=e], p, c) BE
{ Assign(p([i])|L, R([e], p))
  c
}
```

1.2.2 Stored values

Next we introduce hardware registers to replace direct references to stored values. The introduction of fast registers transforms the procedural text for the compilation of the assignment command into:

```
LET C([i:=e], p, c) BE
{ R([e], p, FirstReg)
  Assign(p([i])|L, FirstReg)
  c
}
```

so that R will know where to store the result of evaluating the expression and Assign where to get it from.

1.2.3 Continuations

Consider the semantic equation for a WHILE loop:

```
C["while" b "do" c']pc=
  Fix(λc'.B[b]p(C[c'](p[BRK/c][LOO/c'])c')c).
```

Before the analysis of continuations it will be translated into:

```
LET C(["while" b "do" c'], p, c) BE
{ Fix(λc'.B([b], p, C([c'], p[BRK/c][LOO/c'], c'), c))
}
```

The translator, knowing what a continuation is, and being able to analyse the context in which it appears will be able to translate this text into:

```
LET C([["while" b "do" c']], p, c, t) BE
{ LET c' = ThisContinuation()
  LET c'' = ForwardContinuation()
  B([[b]], p, c'', c, FALSE)
  FixContinuation(c'')
  C([[c']], p[BRK/c][LOO/c'], c', TRUE)
}
```

Where the three procedures This, Forward and Fix-Continuation are used in such a way that they leave to the compiler writer the final choice of implementation. For example they could respectively be -CurrentProgramCounter, NewChain and FixChain- in a 'chaining mechanism' or -PlantNewLabel, ForwardLabel and PlantLabel- relying on the activity of a loader.

1.2.4 Environments

We argue that a simulation of the mathematical environment function is not feasible if efficiency is desired. Thus we translate in a way to have only one global environment around at a time, for which we provide a data structure and primitives to declare and undeclare denoted elements. Environments disappear from parameter lists. The inverse of some functions are defined in order to undo any alteration to the global environment, so that we normally end up with a 'sandwich' of the form:

```
Update environment(...something...)
Call to some procedure
Undo environment(...same something...)
```

Thus, after these transformations the procedural text corresponding to the WHILE becomes:

```
LET C([["while" b "do" c']], c, t) BE
{ LET c' = ThisContinuation()
  LET c'' = ForwardContinuation()
  B([[b]], c'', c, False)
  FixContinuation(c'')
  Declare(BRK, c)
  Declare(LOOP, c')
  C([[c']], c', True)
  UnDeclare(LOOP)
  UnDeclare(BRK)
}
```

Note that the procedure Declare and UnDeclare (and also This, Forward and Fix Continuation) do not generate code. They are part of the 'compile time' activity.

1.2.5 BCPL

Finally we translate into BCPL. This involves only syntactic transformations, i.e. renaming curly functions and making them procedures selecting by cases via a SWITCHON statement, renaming decorated variables, making syntactic references into node references via selectors, and a translation for those procedures returning a tuple. The fragment of the resultant procedure to generate code for commands corresponding to the assignment and WHILE commands is:

```
LET CC(Node, c, t) BE SWITCHON Type OF Node INTO
{
  CASE Com.assignment:
    RR(Second OF Node, FirstReg)
    Update(UJ(First OF Node), FirstReg)
    JumpContinuation(c, t)
    ENDCASE

  CASE Com.while:
    ( LET c1 = ThisContinuation()
      LET c2 = ForwardContinuation()
      BB(First OF Node, c2, c, FALSE)
      FixContinuation(c2)
      Declare(BRK, c)
      Declare(LOO, c1)
      CC(Second OF Node, c1, TRUE)
      UnDeclare(LOO)
      UnDeclare(BRK)
    )
    ENDCASE
    ...
}
```

1.2.6 Example of code generation

Consider a fragment of a program in TL which is a procedure to compute the function factorial by iteration :

```
begin integer N;
      integer F;
      procedure Fact;
      begin F := 1;
            while N > 0 do begin F := F * N; N := N -1 end
      end;
      ...
      call Fact;
      ...
end
```

The corresponding code for the DEC-10, planted by our generated compiler will be

```
---------------------------------------------------------------------------
        JRST    0,L1                ; integer N
                                    ; New(W1)
                                    ; Declare(N, W1)
                                    ; integer F
                                    ; New(W2)
                                    ; Declare(F, W2)
                                    ; procedure Fact; ...
L2:     MOVEM   16,W3               ; New(W3)
                                    ; Declare(RET , L3)
                                    ; Declare(BRK , L4)
                                    ; Declare(LOO , L4)
        MOVEI   1,^D1               ; F := 1
        MOVEM   1,W2
L5:     MOVE    1,W1                ; while N>0 do ...
        JUMPLE  1,L3                ; Declare(BRK , L3)
                                    ; Declare(LOO , L5)
        MOVE    1,W2                ; F := F*N
        IMUL    1,W1
        MOVEM   1,W2
        MOVE    1,W1                ; N := N-1
        SUBI    1,^D1
        MOVEM   1,W1
        JRST    0,L5                ; UnDeclare(LOO , L5)
                                    ; UnDeclare(BRK , L3)
                                    ; UnDeclare(LOO , L4)
                                    ; UnDeclare(BRK , L4)
                                    ; UnDeclare(RET , L3)
L3:     JRST    0,@W3               ; Declare(Fact, L2)

L1:     ...
        JSP     16,L2               ; call Fact
        ...
        JRST    0,L6                ; UnDeclare(Fact, L2)
                                    ; Free(W2)
                                    ; UnDeclare(F, W2)
                                    ; Free(W1)
                                    ; UnDeclare(N, W1)

L4:     PRINTs  0,[ASCIZ/
Wrong/]

L6:     ...

W1:     XWD     0,
W2:     XWD     0,
W3:     XWD     0,
---------------------------------------------------------------------------
```

2. Further developments. RTL: a Recursive Toy Language

In this section, we introduce recursion in our toy language, so that now we talk about a language RTL(Recursive TL). We develop the IDS equations and then we indicate how to carry out a proof that the SDS and IDS are congruent.

2.1 Declaration and Invocation Environment

If we simply add recursion to the IDS of TL, we obtain an equation like the following (we also add one call-by-value parameter):

```
D⟦"procedure" i(i');c⟧p=
  Fix(λp'.p[⟦i⟧/λavc.(New(p'[ARE/a])=>
                      λ<l,p''>.C⟦c⟧
                                (p''[⟦i'⟧/l][RET/c][BRK/Wrong]
                                 [LOO/Wrong])c}]).
```

Again we are interested in an equation which will indicate how to plant efficient code, but it seems that this equation does not help us. If we consider the virtual machine behaviour at the different times of declaration, invocation and execution of a procedure, we can isolate five different objects, which are manipulated in a way that characterises most of the flavour of different programming languages. Namely, associated with every procedure there is:

<u>(I) Local binding</u>
A function to map everything which is bound within the procedure.

<u>(II) External binding</u>
A similar (but not equal) function to map everything which is free.

<u>(III) Local workspace</u>
A function to keep track of those locations defined within the procedure which follow a block structured discipline as opposed to those following a heap discipline.

<u>(IV) Return continuation</u>
The function mapping what remains to be done when the procedure activation terminates.

<u>(V) Current continuation</u>
The function mapping what remains to be done within the procedure.

Some of these are defined at declaration time. For example part of (I), (II), part of (III) and (V) are defined at this time in languages with static binding like Algol.

At invocation time, a copy of what was created at declaration time is made and some other functions are defined, for example (IV) and in dynamically bound languages (II).

At execution time, some functions may be updated. For example (I) and (III) may be extended by new declarations. For a full description of this model, the reader is refered to [Hayes78].

If we now look at our domain definitions and equations we can see that there is no clear mathematical machinery to abstract our model at the different times of declaration and invocation. The environment appears to be abstracting most of the objects above, but they are not structured in the same way:

---

p:U=[[Ide > D] x C x C x C x A]. environments

Secondly, there is no distinction whatsoever between free and bound identifiers. From a (purely) mathematical point of view, it is not necessary to distinguish between them. However, from an implementation standpoint, we have to be able to tell whether a variable has been declared within the current procedure or in an external one, leading to a completely different behaviour of the look up function. For example it might be necessary to walk down a link chain in a stack.

Finally the domain of locations is not absracted at an appropriate level. In the implementation of block structured languages it is reasonable to associate variables to 'offsets' within the workspace of a procedure or block at compilation time. Locations are only allocated at execution time when a 'base' is calculated for all the offsets of the local variables.

To overcome these problems we are going to modify the environment so that it precisely abstracts the model described above. The first four functions are going to be members of the environment while (V), the current continuation is still going to be passed as an explicit parameter to the valuations.

2. Further developments, RTL: a Recursive Toy Language

```
l:L.                                    block structured Locations
b:B.                                    Bases
o:O.                                    Offsets

f:F=[M x U x O x P].                    Function closures
p:U=[M x U x [B x O] x C].              environments
     I    II    III      IV
```

In relation to [Hayes78], F is an Invocation Record Frame and U is an Invocation Record, or in terms of [Bornat77] a Process State Descriptor.

We now describe the parts of the environment, or invocation record in detail:

(I) Local binding
The binding map:

```
m:M=[[Ide > D] x C x C].                binding Map
```

---

is quite similar to the original environment domain. It binds identifiers to their denoted values and the structured jumps BREAK and LOOP to their respective continuations. The empty binding map is defined to be:

```
Nilm:M.

Nilm=
  <λ[i].Nild,Wrong,Wrong>.
```

(II) External binding or an Environment link
This is a reference to the environment of the textually enclosing procedure where the denotation of free identifiers can be found. The function

---

```
LookUp:[Ide > U > E].

LookUp[i]p=
  p[i]=>
  λd.d=Nild>LookUp[i](pEXT),
           d?O>Loc(Nloc<pBAS,d|O>)||E,d?F>d|F||E,Te.
```

---

defined recursively, implies a behaviour which searches down this chain of environments when the denotation of a free identifier is required. Bound identifiers are found in the binding map. LookUp also converts offsets in D to their corresponding locations by reference to the Base in the local workspace component of U.

The semantic equation for a single identifier inside an expression then becomes:

R[i]p=
  Load(LookUp[i]p¦L).

(III) Local workspace
In a function closure, or declaration record frame, the local workspace is an offset. It indicates which is the first free offset at declaration time, whereas in an environment in IDS it is a pair <b, o> indicating where the workspace starts and ends, respectively: <pBAS, FirstO> and <pBAS, pTOP>.

It would be nice to identify locations with the product of bases and offsets in the following manner:

L = [B x O].

However, if we do this we cannot achieve a realistic implementation semantics. As it stands identifying L with B x O (assuming B and O are countably infinite domains, so that for any B and O that might occur in a program the corresponding location exists) means we have an infinite number of locations - which is certainly not required in an implementation semantics. However, if we restrict B and O to being finite domains, we then imply an arbitrary limit to the number of blocks that can appear in a program, and an arbitrary number of locations that can be used in each. Neither of these two possibilities matches up with the standard semantics of the language.

So we are forced to postulate that there are a finite number of locations and a function:

Loc:[N > L]. Undefined

---

which gives a proper location when given an integer in {i¦ 1 <= i <= n}, where n is the number of locations, and otherwise indicates an error. Also we need a function:

2. Further developments. RTL: a Recursive Toy Language

```
Nloc:[[B x O] > N].                              Undefined
```

---

to indirectly find the location corresponding to each B x O. (We do not make Nloc:[[B x O] > L] as we may want to store a <b, o> pair without assuming that the corresponding location exists.)

As we have already indicated, the existence of <b, o>, for some b and o does not guarentee the existence of the corresponding location, and we therefore need the function 'New' again, this time with functionality:

```
New:[[B x O] > L].

New<b,o>=
  Loc(Nloc<b,o>).
```

We must, of course, insist that the locations are used in ascending numeric order, with Nloc<FirstB, FirstO> = 1, and in fact B and O could be identified with N, but we prefer not to do this. Instead we define two primitive functions to obtain new bases and offsets, which we assume satisfy the above two conditions:

```
NewB:[[B x O] > B].                              Undefined
NextO:[O > O].                                   Undefined
```

and two constants which are the first base and first offset:

```
FirstB:B.                                        Undefined
FirstO:O.                                        Undefined
```

To increase the size of the workspace at invocation time we use the post-fix operator:

```
p[TOP / NextO(pTOP)] = p', where p'TOP = NextO(pTOP),
                             and p'X   = pX otherwise
```

Getting a block structured location and binding it to an identifier is now a single activity modelled by the primitive functions BindF at declaration time, and by BindP at invocation time:

```
BindF:[Ide > M > O > [M x O]].
BindF[i]mo=
  <m[[i]/o],NextO o>.

BindP:[Ide > U > U].
BindP[i]p=
  New(pLOC)=>λl.l=Tl>Tp,p[TOP/NextO(pTOP)][[i]/pTOP].
```

(IV) The fourth element in a function closure is a member of P, the domain of procedure values:

```
P=[U > V > C].                              Procedure values
```

It models the meaning of the procedure which is expecting an environment and an actual value for its formal parameter. While in an environment, it is a member of C, the domain of (return) continuations. In relation to [Hayes78], (IV) can be seen as a reference to the current continuation field of the calling invocation record (environment).

The action of activating a function closure (creating a new invocation environment), is modelled by:

```
Activate:[F > [B x O] > C > V > C].

Activate f<b,o>cv=
  (f↓4)<f↓1,f↓2,<NewB<b,o>,f↓3>,c>v.
```

Assuming contiguity of caller and callee, activating means pushing the callee's base on top of the workspace of the caller's invocation environment.

After incorporating the new environment structure and their associated primitive functions the IDS definition of procedure declaration in RTL is:

```
D["procedure" i(i');c]p=
 Fix(λp'.(BindF[i']Nilm FirstO=>
         λ<m,o>.p[[i]/<m,p',o,λp''v.(New<p''BAS,p''[i']|O>=>
                                        λl.(Assign lv o
                                           C[c]p''(p''RET)))>])).
```

2. Further developments, RTL: a Recursive Toy Language

The equation shows how the binding map (m) is formed from the empty one (Nilm) with an additional binding of the parameter [i′] to the first free offset, an external binding (p′) which is the newly created fixed point environment, an indication of how many offsets have already been claimed (one in this case) and finally the procedure value in P.

The IDS equation for a procedure call in RTL is:

```
C[["call" i(e)]]pc=
  R[[e]]p @ Activate(LookUp[[i]]p|F)(pLOC)c.
```

2.2 Relationship between the definitions

We indicate here how a proof of congruence between IDS and SDS of TL can be obtained. This is based on the proof of congruence between IDS and SDS of the recursive version of TL, which is similar in many respects.

There are two substantial changes between the two semantics given for TL: the structure of the environment is altered and the semantic function for boolean expressions has different functionality. As these are entirely separate issues we propose to split the proof into two parts so they don't become confused (which they could do as the environment is a parameter to the valuation (semantic function) for boolean expressions). The disadvantage of splitting the proof into two parts is that we need an intermediate semantics between SDS and IDS which has one of the changes referred to above, but not the other. This is a little unfortunate especially as later on we need two further intermediate semantics for the environment part of the proof, but we persist with the method in the belief that it is the easier to follow.

As the proof of congruence between the valuations for boolean expressions is considerably easier than that between the two environment domains we consider that first by defining a semantics SDS(B) differing from SDS by having the boolean expression valuation from IDS. Later we consider the congruence between SDS(B) and IDS which will establish that the new environment domain does not significantly alter the semantics of the language.

3.1 The Congruence between SDS and SDS(B)

Definition of SDS(B).

As SDS except: B:[Bex > U > C > C > C], and of course all the clauses in the definition of B are altered to look like those in IDS, where p refers to the environment in SDS rather than IDS. Also change the following clauses:

C[["if" b "then" c′ "else" c′′]]pc=
  B[[b]]p(C[[c′]]pc)(C[[c′′]]pc).

C[["while" b "do" c′]]pc=
  Fix(λc′.B[[b]]p(C[[c′]](p[BRK/c][LOO/c′])c′)c).

Theorem : SDS is congruent with SDS(B).

Proof:

We assume here that all the functions starting If... and Is... in the IDS valuation for B[ere′] give false for any argument. So

---

B[[ere′]]pcc′=
  R[[e]]p @ λv.(R[[e′]]p @ λv′.O[[r]]vv′cc′).

We need the following lemma (where B1 refers to B in SDS and B2 refers to B in IDS):

  B1[[b]]p @ Cond<c,c′> = B2[[b]]p.

---

which is easily proved by induction over the structure of b. The result follows immediately from this lemma.

3.2 The congruence between SDS(B) and IDS

Many of the equations in the SDS(B) and IDS of TL now look alike, and although the state and environment domains in the two semantics are different, the proofs of their congruence are trivial. The interest, therefore, lies in the equations in the two definitions which look different, and in particular in the semantic function D. Although we only have a handful of cases to consider the task is more difficult than appears at first sight for reasons we now outline.

The alterations to the state and environment domains appear to be minor, but they are very fundamental. We are taking information out of the state underlying SDS(B) and putting it into the environment in IDS. For these two semantics to be congruent we have to insist that this information corresponds at all times, otherwise they could be using the locations in different ways.

An establisned method[Milne76][Stoy77-9] for relating two domains in semantics which are to be proved congruent, is by imposing inclusive predicates on them. In particular here we have to relate the information in the L ➤ T component of the SDS(B) state and the L ➤ T component of U in IDS, which contain information about the locations in use in either semantics. As this information is kept in different domains in SDS(B) and IDS, predicates defined on corresponding domains cannot insist that it is the same.

One way to overcome this problem might seem to be to define a composite predicate on pairs of states and environments, so we can relate the locations in use. Unfortunately this does not work for at least one reason: environments are bound into continuations in IDS (as well as in SDS), and when we are supplying a state to a continuation in IDS there is no way of checking that the environment bound into that continuation contains the same 'location in use' information as the state supplied in SDS(B). Infact we cannot find out anything about the environment bound into a continuation. A possible solution to this might seem to be to split a continuation so that it is a member of the domain [U ➤ S ➤ S] x U, leaving the environment explicit, but this involves changing the semantics in such a way that it is not implementation oriented. In any case we are trying to find a proof that SDS(B) and IDS are congruent, not find a proof and then make up IDS.

Unfortunately we are led to the conclusion that two intermdiate semantics are required to prove the congruence between SDS(B) and IDS. These are SDS(M), which is SDS(B) modified by having a copy of the L ➤ T component of the state in the environment, and IDS(M), as IDS except that the state has a copy of the L ➤ T component of the environment in it. The details of the semantics have to be altered a little to keep the new parts of the domains in step (ie containing the same information) as the originals. This still does not solve all the problems, though, as we still have environments bound into continuations, and even though the state now also contains the 'location in use' information we must ensure that it is the same as that in the bound in environment when a state is applied to a continuation. To take care of this problem we propose 'continuation transforming' functions which take as arguments a continuation and the environment to be bound into it, and only allow the continuation to be applied to a supplied state, when the 'location in use' information agrees with that in the bound in environment. These functions appear in the semantics everywhere where a new continuation is being created as argument to a semantic function (and a few other places where they help in the proof - don't forget we are now creating a semantics for this purpose). The net result is that every continuation in the semantics contains a check that the supplied state contains the same 'location in use' information as the bound in environment before it is applied; this is because every time a continuation is created the check is incorporated, and all continuations have to be created somewhere in the semantics.

What then have we achieved after all this effort, and how is the proof to proceed? Well we now have four semantics:

SDS(B) <-> SDS(M) <-> IDS(M) <-> IDS

(where the two (M) semantics contain the 'continuation transforming' functions referred to above) which are all congruent. For SDS(B) and SDS(M) to be congruent we have to show that the added component of the environment does not affect the semantics of any program in any significant way, and that the checks for identity of location information in each continuation have no effect. Similarly for the congruence between IDS(M) and IDS. When we have established these results, and they are intuitively fairly clear, all we have to do is show the congruence between SDS(M) and IDS(M) to finish the whole proof.

Acknowledgement

We acknowledge the help and support from Ray Turner and the SRC. Mike Brady has always given encouraging support.

A. Notation

Operators

d:D=Any Domain

01:

$. \rightarrow . : [[\ T \times D \times D\ ] \rightarrow D\ ]$

This is the conditional function. An expression t -> d, d' will take the value d when t is True and the value d' when t is False.

02:

$.\underline{o}. : [[[\ D \rightarrow D'] \times [\ D' \rightarrow D''\ ]] \rightarrow [\ D \rightarrow D''\ ]]$

$f : [\ D \rightarrow D'\ ]$
$g : [\ D' \rightarrow D''\ ]$

$(\ f\ \underline{o}\ g\ )\ d = g(\ f\ d\ )$

This is the reversed form of the composition operator.

03:

$.\underline{\star}. : [[[\ D' \rightarrow [\ D \times D''\ ]] \times [\ D \rightarrow [\ D'' \rightarrow D'''\ ]]] \rightarrow [\ D' \rightarrow D'''\ ]]$

$f : [\ D' \rightarrow [\ D \times D''\ ]]$
$g : [\ D \rightarrow [\ D'' \rightarrow D'''\ ]]$

$(\ f\ \underline{\star}\ g\ )\ d' = g\ d\ d''$ Where $f\ d' = \langle d,\ d'' \rangle$

Reversed form of the Star operator used by C.Strachey in the semantic equation for the While-loop.

04:

$.\underline{@}. : [[[\ D' \rightarrow D\ ] \times [\ D \rightarrow [\ D' \rightarrow D''\ ]]] \rightarrow [\ D' \rightarrow D''\ ]]$

$f : [\ D' \rightarrow D\ ]$
$g : [\ D \rightarrow [\ D' \rightarrow D''\ ]]$

$(\ f\ \underline{@}\ g\ )\ d' = g\ (\ f\ d')\ d'$

This operator will normally be used for expressions without side effects.

```
05:
.=>.:[[[D x [D → D']] → D']

f:[D → D']
x:D

d => λ x.fx is the same as (λx.fx)(d)
```

This operaror, wnich reads as "produce" is the reverse of application, so that we can read equations from left to right.

```
06:
.|.
d:[ D1 + ... + Dn ]
i:N and 1 <= i <= n
```

d|Di is the projection of d into the subdomain Di of [D1 + ... + Dn]

```
07:
.||.
d:Di
i:N and 1 <= i <= n
```

d||[[D1 + ... + Dn] is the injection of d into [D1 + ... + Dn]

```
08:
.↓.:[[[ D1 x ... x Dn ] x N ] → D ]

d=<d1, ... , di, ... , dn >:[ D1 x ... x Di x ... x Dn ]
i:N    And 1 <= i <= n

d ↓ i=di
```

So that ↓ is used to extract individual components of tuples.

```
09:
.†.

d=< d1, ... , di, d(i+1), ... , dn >:[ D1 x ... x Dn ]
i:N   And 1 <= i < n

d†i=< d(i+1), ... , Dn >
```

Operator used to remove elements from tuples.

A. Notation

010:
.^.

d=< d1, ... , di >:[D1 x ... x Di]
d'= < dj, ... , dn >:[Dj x ... x Dn]
i,j:N   and j=i+1 <= n

d^d'=< d1, ... , dn >

Operator used to concatenate tuples.

011:
.?.:

d:[D1 + ... + Dn]
i:N and 1 <= i <= n

d?Di Is True if d is in the Di subdomain of [D1 + ... + Dn], otherwise if False

---

012:
[/]:[[U x D x D'] > U]

x:D

$$p[d/d'] = \begin{cases} <(\lambda x.\ x=d \rightarrow d',\ (p\downarrow 1)x),\ p\downarrow 2,\ \ldots > & \text{if } d\text{:Ide} \\ <p\downarrow 1,\ \ldots\ ,\ p\downarrow(i-1),\ d',\ p\downarrow(i+1),\ \ldots > & \text{if } d \text{ is a selector and } pd = p\downarrow i \end{cases}$$

This is the postfix operator to create new environments. (The notation pSEL, where SEL has been defined as a semantic selector, is equivalent to SELp.)

B. Syntax of IL

Syntactic domains (common to both SDS and IDS versions)

a:Aop.	Arithmetic operators
b:Bex.	Boolean expressions
c:Com.	Commands
d:Dec.	Declarations
e:Exp.	non boolean Expressions
i:Ide.	Identifiers (Undefined)
j:Jmp.	structured Jumps
n:Num.	Numbers (Undefined)
q:Quo.	Quotations (Undefined)
r:Rop.	Relational operators
w:Wri.	Writable expressions

Syntax (common to both SDS and IDS versions)

```
a ::= + | - | * | /
b ::= b' "and" b'' | b' "or" b'' | "true" | "false" | (b') |
      ere'
c ::= c';c'' | i:=e | "if" b "then" c' "else" c'' |
      "while" b "do" c' | "call" i | "dummy" | j | "read" i |
      "write" w | "begin" c' "end" | "begin" d;c' "end"
d ::= "procedure" i;c | "integer" i | d';d''
e ::= i | e'ae'' | n | (e')
j ::= "break" | "loop" | "return"
r ::= > | < | = | >= | <= | <>
w ::= e | q
```

C. Standard Denotational Semantics of TL

C. Standard Denotational Semantics of TL

ISL: SDS of TL

Semantic domains

c:C=[S > S].	Command cont.
D=[P + L].	Denoted values
l:L.	Locations
N.	iNtegers
P=[C > C].	Procedure values
Q.	Quotations
r:R=[N + Q].	pRintable values
s:S=[[L > V] x V* x R* x [L > T]].	States
T=[{ TRUE } + { FALSE }].	Truth values
p:U=[[Ide > D] x C x C x C].	environments
v:V=[N].	storable Values

Semantic selectors

```
BRK==λp.p↓2.
LOO==λp.p↓3.
RET==λp.p↓4.
```

Semantic functions

```
B:[Bex > U > S > T].
C:[Com > U > P].
D:[Dec > U > S > [U x S]].
J:[Jmp > U > C].
R:[Exp > U > S > V].
W:[Wri > U > P].
```

Semantic primitives

```
A:[Aop > V > V > V].                          Undefined
N:[Num > N].                                  Undefined
Q:[Quo > Q].                                  Undefined
O:[Rop > V > V > T].                          Undefined
Wrong:C.                                      Undefined
Assign:[L > V > C].
Assign lvs=
  <λl'.l=l'>v,(s↓1)l',s↓2,s↓3,s↓4>.

Load:[L > S > V].
Load ls=
  (s↓1)l.

New:[S > [L x S]].
Read:[S > [V x S]].
Read s=
  #(s↓2)=0><Tv,s>,<s↓2↓1,<s↓1,s↓2†1,s↓3,s↓4>>.

Write:[R > C].
Write rs=
  <s↓1,s↓2,s↓3^r,s↓4>.
```

C. Standard Denotational Semantics of TL

Semantic valuator for expressions

R⟦i⟧p=
  Load(p⟦i⟧↓L).

R⟦e′ae′′⟧p=
  R⟦e′⟧p @ λv.(R⟦e′′⟧p o A⟦a⟧v).

R⟦n⟧ps=
  N⟦n⟧.

R⟦(e′)⟧p=
  R⟦e′⟧p.

Semantic valuator for boolean expressions

B⟦b′ "and" b′′⟧p=
  B⟦b′⟧p @ Cond<B⟦b′′⟧p,λs.FALSE>.

B⟦b′ "or" b′′⟧p=
  B⟦b′⟧p @ Cond<λs.TRUE,B⟦b′′⟧p>.

B⟦"true"⟧ps=
  TRUE.

B⟦"false"⟧ps=
  FALSE.

B⟦(b′)⟧p=
  B⟦b′⟧p.

B⟦ere′⟧p=
  R⟦e⟧p @ λv.(R⟦e′⟧p o O⟦r⟧v).

Semantic valuator for commands

```
C⟦c';c''⟧ρc=
  C⟦c'⟧ρ(C⟦c''⟧ρc).

C⟦i:=e⟧ρc=
  R⟦e⟧ρ @ Assign(ρ⟦i⟧|L) o c.

C⟦"if" b "then" c' "else" c''⟧ρc=
  B⟦b⟧ρ @ Cond<C⟦c'⟧ρc,C⟦c''⟧ρc>.

C⟦"while" b "do" c'⟧ρc=
  Fix(λc'.(B⟦b⟧ρ @ Cond<C⟦c'⟧(ρ[BRK/c][LOO/c'])c',c>)).

C⟦"call" i⟧ρc=
  (ρ⟦i⟧|P)c.

C⟦"dummy"⟧ρc=
  c.

C⟦j⟧ρc=
  J⟦j⟧ρ.

C⟦"read" i⟧ρc=
  Read * Assign(ρ⟦i⟧|L) o c.

C⟦"write" w⟧ρc=
  W⟦w⟧ρc.

C⟦"begin" c' "end"⟧ρc=
  C⟦c'⟧ρc.

C⟦"begin" d;c' "end"⟧ρcs=
  (D⟦d⟧ρ * λρ'.C⟦c'⟧ρ'(λs'.c<s'↓1,s'↓2,s'↓3,s↓4>))s.
```

C. Standard Denotational Semantics of TL

Semantic valuator for declarations

D⟦"procedure" i;c⟧ps=
  <p[⟦i⟧/λc.C⟦c⟧(p[RET/c][BRK/Wrong][LOO/Wrong])c],s>.

D⟦"integer" i⟧p=
  New ⋆ λls.<p[⟦i⟧/l],s>.

D⟦d';d''⟧p=
  D⟦d'⟧p ⋆ D⟦d''⟧.

Semantic valuator for structured jumps and writable values.

J⟦"break"⟧p=
  pBRK.

J⟦"loop"⟧p=
  pLOO.

J⟦"return"⟧p=
  pRET.

W⟦e⟧pc=
  R⟦e⟧p @ Write ⊙ c.

W⟦q⟧pc=
  Write(Q⟦q⟧) ⊙ c.

D. Implementation Denotational Semantics of TL

ISL: IDS of TL

Semantic domains

a:A=[L > T].	Area function
c:C=[S > S].	Command cont.
D=[P + L].	Denoted values
l:L.	Locations
N.	iNtegers
P=[A > C > C].	Procedure values
Q.	Quotations
r:R=[N + Q].	pRintable values
s:S=[[L > V] x V* x R*].	States
T=[{ TRUE } + { FALSE }].	Truth values
p:U=[[Ide > D] x C x C x C x A].	environments
v:V=[N].	storable Values
W=[V + L].	dumped Walues
Y=[Aop + Bex + Exp + Rop].	reversed sYntax

Semantic selectors

BRK==λp.p↓2.
LOO==λp.p↓3.
RET==λp.p↓4.
ARE==λp.p↓5.

D. Implementation Denotational Semantics of TL

Semantic functions

```
B:[Bex → U → C → C → C].
C:[Com → U → C → C].
D:[Dec → U → U].
J:[Jmp → U → C].
R:[Exp → U → S → V].
W:[Wri → U → C → C].
```

Semantic primitives

```
A:[Aop → V → W → S → V].                       Undefined
N:[Num → N].                                   Undefined
Q:[Quo → Q].                                   Undefined
O:[Rop → V → W → C → C → C].                   Undefined
BJump:[Rop → V → C → C → C].                   Undefined
BLeaf:[Exp → Rop → U → V → C → C → C].         Undefined
Dump:[U → V → S → [L x U x S]].                Undefined
IfNeedToDump:[Exp → T].                        Undefined
IfNeedToReverse:[[Bex + Exp] → T].             Undefined
IfZero:[Exp → T].                              Undefined
IsLeaf:[Exp → T].                              Undefined
Reverse:[Y → Y].                               Undefined
RLeaf:[Exp → Aop → U → V → S → V].             Undefined
Wrong:C.                                       Undefined
Assign:[L → V → C].
Assign lvs=
  <λl′.l=l′→v,(s↓1)l′,s↓2,s↓3>.

Load:[L → S → V].
Load ls=
  (s↓1)l.

New:[U → [L x U]].
Read:[S → [V x S]].
Read s=
  #(s↓2)=0→<Tv,s>,<s↓2↓1,<s↓1,s↓2†1,s↓3>>.

Write:[R → C].
Write rs=
  <s↓1,s↓2,s↓3^r>.
```

Semantic valuator for expressions

```
R[i]p=
  Load(p[i]|L).

R[e'ae'']p=
  IfNeedToReverse[e'ae'']>R(Reverse[e'ae'']|Exp)p,
  (R[e']p @
   λv.IsLeaf[e'']>RLeaf[e''][a]pv,
      IfNeedToDump[e'']>
      Dump pv * λ<l,p'>.(R[e'']p' @ λv.A(Reverse[a]|Aop)vl),
      (R[e'']p @ A[a]v)).

R[n]ps=
  N[n].

R[(e')]p=
  R[e']p.
```

Semantic valuator for boolean expressions

```
B[b' "and" b'']pcc'=
  B[b']p(B[b'']pcc')c'.

B[b' "or" b'']pcc'=
  B[b']pc(B[b'']pcc').

B["true"]pcc'=
  c.

B["false"]pcc'=
  c'.

B[(b')]pcc'=
  B[b']pcc'.

B[ere']pcc'=
  IfZero[e]>R[e']p @ λv.BJump(Reverse[r]|Rop)vcc',
  IfZero[e']>R[e]p @ λv.BJump[r]vcc',
  IfNeedToReverse[ere']>B(Reverse[ere']|Bex)pcc',
  (R[e]p @
   λv.IsLeaf[e']>BLeaf[e'][r]pvcc',
      IfNeedToDump[e']>
      Dump pv * λ<l,p'>.(R[e']p' @ λv.O(Reverse[r]|Rop)vlcc'),
      (R[e']p @ λv'.O[r]vv'cc')).
```

D. Implementation Denotational Semantics of T

Semantic valuator for commands

```
C[c';c'']pc=
  C[c']p(C[c'']pc).

C[i:=e]pc=
  R[e]p @ Assign(p[i]|L) o c.

C["if" b "then" c' "else" c'']pc=
  B[b]p(C[c']pc)(C[c'']pc).

C["while" b "do" c']pc=
  Fix(λc'.B[b]p(C[c'](p[BRK/c][LOO/c'])c')c).

C["call" i]pc=
  (p[i]|P)(pARE)c.

C["dummy"]pc=
  c.

C[j]pc=
  J[j]p.

C["read" i]pc=
  Read * Assign(p[i]|L) o c.

C["write" w]pc=
  W[w]pc.

C["begin" c' "end"]pc=
  C[c']pc.

C["begin" d;c' "end"]pc=
  C[c'](D[d]p)c.
```

Semantic valuator for declarations

```
D[["procedure" i;c]]p=
  p[[[i]]/λac.C[[c]](p[ARE/a][RET/c][BRK/Wrong][LOO/Wrong])c].

D[["integer" i]]p=
  New p=>λ<l,p'>.p'[[[i]]/l].

D[[d';d'']]p=
  D[[d']]p=>D[[d'']].
```

Semantic valuator for structured jumps and writable values.

```
J[["break"]]p=
  pBRK.

J[["loop"]]p=
  pLOO.

J[["return"]]p=
  pRET.

W[[e]]pc=
  R[[e]]p @ Write o c.

W[[q]]pc=
  Write(Q[[q]]) o c.
```

E. The Generated Compiler

```
//      File DSK:TLBCL.MS
//      Compiled by ISL 1A(23) at 10:32 21/2/80
//      Output of phase 6


LET RR(Node, Reg) BE SWITCHON Type OF Node INTO
{ CASE S..i:
    Load(UU(Node), Reg)
    ENDCASE

  CASE Exp.Aop:
    TEST IfNeedToReverse(Node, Reg) THEN RR(Reverse(Node), Reg)
    OR
    { RR(First OF Node, Reg)
      TEST IsLeaf(Third OF Node)
      THEN RLeaf(Third OF Node, Second OF Node, Reg)
      OR
      TEST IfNeedToDump(Third OF Node)
      THEN
      { LET l = Dump(Reg)
        RR(Third OF Node, Reg)
        AA(Reverse(Second OF Node), Reg, l)
        Free(l)
      }
      OR
      { RR(Third OF Node, NextReg(Reg))
        AA(Second OF Node, Reg, NextReg(Reg))
      }
    }
    ENDCASE

  CASE S..n:
    NN(Node, Reg)
    ENDCASE

  CASE Exp.brackets:
    RR(First OF Node, Reg)
    ENDCASE
```

```
LET BB(Node, c, cl, t) BE SWITCHON Type OF Node INTO
{ CASE Bex.and:
    { LET c2 = ForwardContinuation()
      BB(First OF Node, c2, cl, FALSE)
      FixContinuation(c2)
      BB(Second OF Node, c, cl, t)
    }
    ENDCASE

  CASE Bex.or:
    { LET c2 = ForwardContinuation()
      BB(First OF Node, c, c2, TRUE)
      FixContinuation(c2)
      BB(Second OF Node, c, cl, t)
    }
    ENDCASE

  CASE Bex.true:
    JumpContinuation(c, t)
    ENDCASE

  CASE Bex.false:
    JumpContinuation(cl, NOT t)
    ENDCASE

  CASE Bex.brackets:
    BB(First OF Node, c, cl, t)
    ENDCASE
```

E. The Generated Compiler

```
CASE Bex.Rop:
  TEST IfZero(First OF Node)
  THEN
  { RR(Third OF Node, FirstReg)
    BJump(Reverse(Second OF Node), FirstReg, c, cl, t)
  }
  OR
  TEST IfZero(Third OF Node)
  THEN
  { RR(First OF Node, FirstReg)
    BJump(Second OF Node, FirstReg, c, cl, t)
  }
  OR
  TEST IfNeedToReverse(Node) THEN BB(Reverse(Node), c, cl, t)
  OR
  { RR(First OF Node, FirstReg)
    TEST IsLeaf(Third OF Node)
    THEN BLeaf(Third OF Node, Second OF Node, FirstReg, c, cl, t)
    OR
    TEST IfNeedToDump(Third OF Node)
    THEN
    { LET l = Dump(FirstReg)
      RR(Third OF Node, FirstReg)
      OO(Reverse(Second OF Node), FirstReg, l, c, cl, t)
      Free(l)
    }
    OR
    { RR(Third OF Node, NextReg(FirstReg))
      OO(Second OF Node, FirstReg, NextReg(FirstReg), c, cl, t)
    }
  }
  ENDCASE
```

```
LET CC(Node, c, t) BE SWITCHON Type OF Node INTO
{ CASE Com.semicolon:
    { LET c1 = ForwardContinuation()
      CC(First OF Node, c1, FALSE)
      FixContinuation(c1)
      CC(Second OF Node, c, t)
    }
    ENDCASE

  CASE Com.assignment:
    RR(Second OF Node, FirstReg)
    Assign(UU(First OF Node), FirstReg)
    JumpContinuation(c, t)
    ENDCASE

  CASE Com.ifthenelse:
    { LET c1 = ForwardContinuation()
      LET c2 = ForwardContinuation()
      BB(First OF Node, c1, c2, FALSE)
      FixContinuation(c1)
      CC(Second OF Node, c, TRUE)
      FixContinuation(c2)
      CC(Third OF Node, c, t)
    }
    ENDCASE

  CASE Com.whiledo:
    { LET c1 = ThisContinuation()
      LET c2 = ForwardContinuation()
      BB(First OF Node, c2, c, FALSE)
      FixContinuation(c2)
      Declare(BRK, c)
      Declare(LOO, c1)
      CC(Second OF Node, c1, TRUE)
      UnDeclare(LOO)
      UnDeclare(BRK)
    }
    ENDCASE

  CASE Com.call:
    CallContinuation(UU(First OF Node))
    JumpContinuation(c, t)
    ENDCASE

  CASE Com.dummy:
    JumpContinuation(c, t)
    ENDCASE
```

E. The Generated Compiler

```
  CASE Jmp.break: CASE Jmp.loop: CASE Jmp.return:
    JJ(Node)
    ENDCASE

  CASE Com.read:
    Read(FirstReg)
    Assign(UU(First OF Node), FirstReg)
    JumpContinuation(c, t)
    ENDCASE

  CASE Com.write:
    WW(First OF Node, c, t)
    ENDCASE

  CASE Com.beginend:
    CC(First OF Node, c, t)
    ENDCASE

  CASE Com.beginsemicolonend:
    { LET cl = ForwardContinuation()
      JumpContinuation(cl, DeclareingContinuation(First OF Node))
      DD(First OF Node)
      FixContinuation(cl)
      CC(Second OF Node, c, t)
      UnDD(First OF Node)
    }
    ENDCASE
}


LET WW(Node, c, t) BE SWITCHON Type OF Node INTO
{ CASE S..i: CASE S..n: CASE Exp.Aop:
  CASE Exp.brackets:
    RR(Node, FirstReg)
    Write(FirstReg)
    JumpContinuation(c, t)
    ENDCASE

  CASE S..q:
    Write(QO(Node))
    JumpContinuation(c, t)
    ENDCASE
```

```
LET DD(Node) BE SWITCHON Type OF Node INTO
{ CASE Dec.procedure:
    { LET cl = ThisContinuation()
      LET c = EntryContinuation()
      Declare(RET, c)
      Declare(BRK, Wrong)
      Declare(LOO, Wrong)
      CC(Second OF Node, c, FALSE)
      UnDeclare(LOO)
      UnDeclare(BRK)
      UnDeclare(RET)
      ExitContinuation(c)
      Declare(First OF Node, cl)
    }
    ENDCASE

  CASE Dec.integer:
    { LET l = New()
      Declare(First OF Node, l)
    }
    ENDCASE

  CASE Dec.semicolon:
    DD(First OF Node)
    DD(Second OF Node)
    ENDCASE
}


LET JJ(Node) BE SWITCHON Type OF Node INTO
{ CASE Jmp.break:
    JumpContinuation(UU(BRK), TRUE)
    ENDCASE

  CASE Jmp.loop:
    JumpContinuation(UU(LOO), TRUE)
    ENDCASE

  CASE Jmp.return:
    JumpContinuation(UU(RET), TRUE)
    ENDCASE
```

F. References

E. References

[BCPL77] Reference manual, Department Of Computer Science. Essex University. 1977.

[Bornat76]R.Bornat. Notes for Comparative Study of Programming Languages, Department Of Computer Science. Essex University. 1976.

[Bornat77]R.Bornat. Understanding and Writing Compilers. MacMillan 1977.

[Gries71]D.G.Gries. Compiler Construction for Digal Computers, J.Wiley and Sons. 1971.

[Hayes78]P.J.Hayes. Invocation Records: A conceptual Framework for Evaluating Program Text. Department Of Computer Science. Essex University. 1979.

[Jones80]N.D.Jones and D.A.Schmidt. Compiler Generation from Denotaional Semantics (Preliminary Report) Workshop on Semantics-Directed Compiler Generation. Department Of Computer Science. Aarhus University. 1980.

[Milne76]R.Milne and C.Strachey. A Theory of programming language semantics, Chaoman and Hall. 1976.

[Mosses74]P.D.Mosses. The Semantics of Semantic Equations. Mathematical Foundations of Computer Science. Lecture Notes in Computer Science 28. Springer-Verlag. Proc. 3rd MFCS Symposium. Warsaw. 1974. pp.409-422

[Mosses75]P.D.Mosses. Mathematical Semantics and Compiler Generation, PhD. thesis. University of Oxford. 1975.

[Mosses76]P.D.Mosses. Compiler Generation using Denotational Semantics. Mathematical Foundations of Computer Science. Lecture Notes in Computer Science 45. Springer-Verlag. Proc. 5th MFCS Symposium. Gdansk Poland. 1976. pp.436-441

[Mosses78]P.D.Mosses. SIS: A Compiler Generator System using Denotational Semantics, Reference Manual. University of Aarhus. 1978.

[Raskovsky79]M.R.Raskovsky and R.Turner. Compiler Generation and Denotational Semantics. Fundamentals of Computation Theory, 1979.

[Raskovsky80]M.R.Raskovsky. ISL (In preparation) Department Of Computer Science. Essex University. 1980.

[Richards69]M.Richards. BCPL: A tool for compiler writing and system programming. Proceedings of the 1969 Spring Joint Computer

Conference. Boston AFIPS Montvale 1969 pp.557-566.

[Scott70]D.Scott. Outline of a Mathematical Theory of Computation, PRG-2. Oxford University Computing Laboratory. 1970.

[Scott71]D.Scott and C.Strachey. Toward a Mathematical Semantics for Computer Languages, PRG-6. Oxford University Computing Laboratory. 1971.

[Scott76]D.Scott. Data Types as Lattices, Procedings of the 1974 Colloquium in Mathematical Logic. Kiel. Springer-Verlag. Berlin 1976. pp.579-650.

[Stoy77]J.E.Stoy. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press. 1977.

[Stoy77]J.E.Stoy. The congruence of Two Programming Language Definitions. (manuscript). 1979.

[Strachey66]C.Strachey. Towards a formal semantics, Formal Language Description Languages for Computer Programming. (edited by T.B.Steel). North-Holland. Amsterdam 1966. pp.198-220.

[Strachey67]C.Strachey. Fundamental Concepts in Programming Languages, International Summer School in Computer Programming, 1967 (Typescript).

[Strachey74]C.Strachey and C.P.Wadsworth. Continuations, A Mathematical Semantics for handling full jumps, PRG-11. Oxford University Computing Laboratory. 1974.

[Sufrin77]B.Sufrin. LL1: A Parser Generator. Department Of Computer Science. Essex University. 1978.