A Correspondence Between

Denotational Semantics

and

Code Generation

Martin R. Raskovsky

A Correspondence Between Denotational Semantics and Code Generation

Martin R. Raskovsky

Thesis submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy

Department of Computing Science
University of Essex

August 8th 1982

To: my father - in memoriam.

my mother, Daniel,

my sisters and brothers.

Abstract

We describe a method for the systematic derivation, or automatic generation - from the formal denotational semantic specification - of an efficient compiler's code generation phase, producing efficient code for real machines. The method has been successfully implemented and tested with languages as complex as GEDANKEN!

The method has been used to implement a compiler-compiler which inputs the semantic specification of a programming language written in a standard denotational form, analyses it in the light of its semantic contents, decides upon certain predecided general implementation issues and outputs a program written in the systems programming language BCPL. This program constitutes the type.checking and code generation phases of a compiler for the given language. Its structure and operation is in effect essentially the same that we would have produced by hand. The only hand coding is the interface for the particular target machine, required for both the primitive functions of the original specification and those introduced by the compiler-compiler. For the latter, our system provides a library of routines to generate code for the DEC-10 system. So that in all the examples we tried, we only had to hand code the former. The parser is separately generated with an LL1 system which also generates BCPL procedures.

Acknowledgments

The original idea, on which this thesis is based, was implicitly suggested by the lectures given by: M.Brady, R.Bornat, P.Hayes and R.Turner, who combined the Scott-Strachey approach to programming language theory with programming language implementation techniques.

R.Turner, my supervisor, gave the necessary insight into denotational semantics, without which this thesis would not have been possible.

P.Collier contributed, during the first period, congruence proofs between the standard and implementation versions of denotational semantics. Also, his invaluable comments on early drafts of this dissertation are appreciated.

It was a great pleasure working with M.Henson, his clarity of thought, encouraging support throughout the years of research and final proof reading have been of immense value.

B.Sufrin suggested a production rule formalism, to describe abstractly the implementation; changing completely the mood of this text.

The insight into the DEC-10 operating system, given by N.Wilson, was of great assistance during the implementation stages.

Contents

List of Snapshots

CHAPTER 1

Introduction

The formal aspects of the syntax and semantics of programming languages, together with their underlying theories, have proved to be very useful for systematically writing, or mechanically generating, implementations for these languages.

For example, the style of description of ALGOL60 suggested to E. Irons [Iro61], that the formal syntactic description of the language could be used to control the compiler for the language directly. Shortly after, the term 'compiler-compiler' was already being used by R. Brooker [Bro62] [Bro63], however, [Bro60] describes an 'autocode to write autocodes'.

The micro-syntax of a language - how words are formed - is in general specified by a regular grammar (RG), whose associated implementation system is a finite state machine (FSM). The theory of regular languages and finite automata was developed in the early 1950's and is therefore one of the oldest branches of theoretical computing science. On the other hand, the syntax of a language - how words are put together into sentences - is in general specified by a context free grammar (CFG), whose associated implementation system is a push down automaton (PDA). The early 1960's witnessed a tremendous growth in language theory, the Chomsky hierarchy has been extensively studied by many people. Finally, the semantics of a language - what a sentence means - is for our purposes specified by abstract mathematical entities, a system of functions whose associated implementation system is the lambda calculus (LAM). This method, Denotational Semantics (DS), originated in the late 1960's. C. Strachey in his early paper 'Towards a formal Semantics' [Str66], showed that with the introduction of a few new basic concepts it was possible to describe not only the applicative, but also the imperative parts of a programming language in terms of applicative expressions [Lan65]. The result of Strachey's joint work with D. Scott, which started in 1969 [Sco70], was the construction of λ-calculus models and reflexive domains: The main point is the treatment of functions as the representation of the meaning of programs, rather than the syntactic or operational representation used thus far.

1.1 **Analogy**

The three implementation systems: FSM, PDA and LAM, can be regarded as input programs to a compiler generator system whose output can be interpreted by a direct simulation of each underlying machine. For example, from the definition of a particular FSM:

<S, I, s, O, M> **where**
S = finite non-empty set of states.
I = finite input alphabet.
s = the initial state in S.
O = set of final states ($O \subseteq S$).
M = the state transition function ($[S \times I] \rightarrow S$).

one can implement a lexical analyser by a simulation of M (also by adding a set of actions, an error state and a set of declarations global to every action). This approach to automatic generation of lexical analysers is extensively described in [Lew79].

From the definition of a PDA:

```
<S, I, P, s, p, O, M> where
S = finite set of states.
I = finite input alphabet.
P = finite pushdown alphabet.
s = the initial state in S.
p = the start symbol in P.
O = set of final states (O ⊆ S).
M = the state transition function
    ([S x [I + {empty}] x P] → finite subsets of [S x P*).
```

one can implement a parser, by a simulation of P and M, (also adding actions, error state, declarations, attributes values and attributed pushdown symbols). Syntax directed translation has been known for quite a long time, [Iro61], [Knu68], [Lew68], [AaU69], [AaU72] and [Lew79] are just a few references to the subject.

From the DS definition of the semantics of a programming language in LAM:

```
i:Ide.                          identifiers
e:Lam.                          lambda expressions

e ::= i | ee' | λi.e | (e)
```

one can directly implement the conversions of the λ-calculus:

a. If i is not free in e, then $\lambda i'.e \Rightarrow_a \lambda i.[i/i']e$.
b. $(\lambda i.e)e' \Rightarrow_b [e'/i]e$.
n. If i is not free in e, then $\lambda i.ei \Rightarrow_n e$.

Pioneer work in the area of compiler generation from denotational semantics, was carried out by P. Mosses [Mos75] [Mos76] [Mos78], whose system known as SIS (Semantic Implementation System), uniformly translates DS equations into LAM, and then runs an interpreter over this 'code'. Also, later work by [Jon80], [Gan80], [Ras80], [Pau81], [Hen82] and [Set82] has shown that semantic directed compiler/interpreter generation out of DS is a young and promising area of research.

Mosses's approach is simple and general; He treats a semantic specification as a program for a simulated LAM machine exactly in the same way as an RG or CFG is seen as program for respectively an FSM or a PDA. The result is, however, not always efficient and practical. In particular, the lack of efficiency of SIS reminds one of an analogy between the systems devised to efficiently implement an FSM or a PDA; they are not necessarily a simulation of the underlying automaton. In the same way, one can think of an implementation of LAM which does not necessarily call for a lambda interpreter. The particular problem with a semantic specification is that it refers to a run time activity, as opposed to a syntactic specification which refers to a recognise and parse activity. At run time efficiency matters become crucial.

An alternative approach for achieving an efficient and practical implementation is to use the formal specification to 'derive' or 'generate' a 'program', written in a systems programming language. From an RG one can generate a scanner, from a CFG a parser and from a DS a code generator. These three programs perform each a function which can be expressed as:

scanner:[CHA → SYM], parser:[SYM → TRE], translator:[TRE → COD] where
CHA = the representation of the source program as a character string.
SYM = the internal representation as a sequence of symbols.
TRE = the internal representation in the form of a tree.
COD = the target code.

In our research, the missing function checker:[TRE → TRE] (to type check and solve the context sensitive aspects of a programming language) is embedded in the code generation phase.

Consider as an example, the RG specification of IDENTIFIERS as an instance of a lexical specification:

```
t ::= i | ...                          terminals
i ::= u a*                             identifiers
a ::= u | l | d                        alphabetic
u ::= 'A' ... | 'Z'                    upper
l ::= 'a' ... | 'z'                    lower
d ::= '0' ... | '9'                    digit
```

The first step in a derivation of a scanner is to generate a recogniser:

```
let scann.next.symbol() be switchon current.character into
{ case 'A' ... 'Z':
    scann.next.character() repeatwhile  'A' <= current.character <= 'Z' \/
                                        'a' <= current.character <= 'z' \/
                                        '0' <= current.character <= '9'
    endcase

  case ...
}
```

Next, one can inject simple implementation techniques to transform it into a scanner:

```
let scann.next.symbol() be switchon current.character into
{ case 'A' ... 'Z':
    { let v = vec max.ident
      let i = 0
      { i   := i + 1
        v!i := current.character
        scann.next.character()
      } repeatwhile  'A' <= current.character <= 'Z' \/
                     'a' <= current.character <= 'z' \/
                     '0' <= current.character <= '9'
      v!0 := i
      current.symbol := look.up.ident(v)
    }
    endcase

  case ...
}
```

Note that this scanner, derived from the original RG specification, makes reference to a symbol structure via the call of the function look.up.ident whose specification was not given and is left open for an implementer to design according to his own implementation choice. Also the interface to an operating system or front-end, throughout the call of the procedure scann.next.character is also left unspecified, so that in a different environment, or different hardware configuration, an implementer can choose the appropriate implementation.

Now consider the CFG specification of a WHILE-LOOP as an instance of a command definition:

$$c ::= \text{While } b \text{ Do } c_1 \mid \ldots$$

Again, the first step is to derive a recogniser for this fragment, for example:

```
let parse.c() be switchon current.symbol into
{ case symbol.While:
    { scan.next.symbol()
      parse.b()
      check.for(symbol.Do)
      parse.c()
    }
    endcase

  case ...
}
```

Next, this recogniser can easily be made into a parser (a tree constructor) by adding appropriate action routines:

```
let parse.c() = valof switchon current.symbol into
{ case symbol.While:
    { let p1, p2 = nil, nil
      scan.next.symbol()
      p1 := parse.b()
      check.for(symbol.Do)
      p2 := parse.c()
      resultis make.node2(N2..While, p1, p2)
    }

  case ...
}
```

In this parser, one must note again, how the structure of the internal representation of the syntactic structure of a program is not specified. Again this leaves crucial implementation details open for an implementer to decide. So in this respect, we are arguing for a generation of structured and efficient compilers.

Finally, consider the semantic specification for the same fragment:

Syntactic Domains

b:Boo.	boolean expressions
c:Com.	commands
i:Ide.	identifiers

Syntax

c ::= **While** b **Do** c_1 | ..

Semantic Domains

D.	denoted values
S.	states
$c{:}C=[S \rightarrow S]$.	command continuations
$k{:}K=[T \rightarrow C]$.	expression continuations
$t{:}T=[\{\text{ True }\} + \{\text{ False }\}]$.	boolean values
$p{:}U=[[\text{Ide} \rightarrow D] \times C]$.	environments

Semantic Selector

$p\text{BRK}{=}{=}p{\downarrow}2$.

Semantic Functions

$\mathbf{B}{:}[\text{Boo} \rightarrow U \rightarrow K \rightarrow C]$.
$\mathbf{C}{:}[\text{Com} \rightarrow U \rightarrow C \rightarrow C]$.

$\mathbf{C}[\textbf{While}\ b\ \textbf{Do}\ c_1]pc=$
$\quad \text{Fix}\{\lambda c'.\mathbf{B}[b]p\{\lambda t.t \rightarrow \mathbf{C}[c_1](p[c/\text{BRK}])c',c\}\}.$

This definition is far more complex in its structure and information content than the syntactic one above. Though at first one may have the impression that the relationship between this semantic specification and a procedure to generate code for the same fragment is hopelessly unrecognisable, in fact, there are many systematic relationships between both. This relationship is important to anyone who wishes to study semantic or implementation structures.

As an example of the correspondence that we are proposing here is the first step; The derivation of the skeleton of a translator:

```
let trans.C(node, p, c) be switchon type^node into
{ case N2..While:
    Fix(λcl.trans.B(pl^node, p, λt.t→trans.C(p2^node, p([c/BRK]), cl),c))
    endcase

  case ...
}
```

The second step is to derive from this, the translator or code generator:

```
let trans.C(node).cont.(continue, jump) be switchon type^node into
{ case N2..While:
    {0 let restart.code = here(D..COD)
       let continuel = forward(D..COD)
       trans.B(pl^node).cont.(continuel, false.jump).dest.(first.reg)
       fix.here(continuel)
       trans.jump.if.false(first.reg, continue)
       { let old.env = this.env
         declare(D..COD, continue, BRK)
         trans.C(p2^node).cont.(restart.code, true.jump)
         reset(old.env)
    }0; endcase

  case ...
}
```

The functions and procedures <u>here</u>, <u>forward</u> and <u>fix.here</u> relate to open implementation issues with respect to a code structure; <u>declare</u>, <u>reset</u> and <u>this.env</u> refer to a descriptor structure; <u>first.reg</u> to a run-time structure; <u>trans.jump.if.false</u> to a code generator interface; <u>trans.B</u> is a procedure which is expected to be generated accordingly to the DS specification for boolean expressions; And finally <u>node</u>, <u>type</u>, <u>pl</u> and <u>p2</u> refer to a tree structure.

1.2 Our contribution

The generation of an efficient and practical program out of the formal specification of the syntax (or lexical) issues of a programming language is a solved problem. However, the generation of an efficient translator out of a semantic specification in the form of a denotational semantics is not. This thesis: A Correspondence Between the Denotational Semantics of Programming Languages and the Process of Code Generation, has grown out of experience in the engineering task of implementing a semantics directed compiler generator. As will be quite evident, the method, as with the analog syntactic problem, is to generate a program, written in a systems programming language, by transforming the semantic specification. The treatment of the relationships, between the source semantics and the target implementation indicates clearly a dependency upon the form and structure of both source mathematical metalanguage and target systems programming language. However, the manner in which the generation is formulated does not adhere rigidly to any particular semantic or implementation model. As a result, the method can be readily adapted to a variety of practical situations.

An essential feature of this thesis is the series of compiler generation examples presented, in general taken from J. Stoy's "Denotational Semantics: The Scott-Strachey Approach To Programming Language Theory" [Sto77]. This is designed to make the format more approachable to anybody familiar with this excellent account of denotational semantics.

This thesis will show how one can systematically derive, or automatically generate, the code generation phase of compilers for languages like those given as examples of semantic definitions in [Sto77] or as complex as GEDANKEN [Rey70].

The novel aspect of our contribution is our method of analysing a denotational specification. From a purely mathematical point of view the way that functions in some abstract space are described is irrelevant. But if one is to process this specification automatically, then the particular representation of these functions is important. We call this representation the concrete semantics. Concrete semantics can be interpreted algorithmically. This is what P. Mosses did by a direct interpretation of LAM. Our approach is characterised by the observation that under an appropriate algorithmic interpretation, functions and domains of a concrete semantics, can be regarded as procedures and data-types. Such interpretation amounts to establishing a correspondence between functions and procedures and therefore also between domains and data-types,

The structure and operation of the automatically generated (or systematically derived) code generator, which is written in the systems programming language BCPL, is in effect very similar to the one we might have produced by hand. The only hand coding is the interface for the particular target machine, required for both the primitive functions of the original specification and those introduced by the generation. For the latter, our system provides a library of routines to generate code for the DEC-10 system, so that in all the examples we tried, we only had to hand code the former. The parser is separately generated with an LL1 system [Suf78a] which generates procedures in the same systems programming language.

Our research is not directly concerned with the problem of correctness. We are mainly interested in the correspondence between a denotational specification and the process of code generation. We are also interested in showing that this correspondence can be used to produce a compiler-compiler, and that the code generators obtained by such a system produce efficient code.

In the following sequence of operations:

0. Given a transformational system T
   and a denotational specification S of a programming language L.
1. Using T, generate from S a code generator G for L.
2. Using G, compile a program P (in L) producing code C for the DEC-10.
3. Running C over some input I obtaining an output O.

we have limited our requirements for correctness to the empirical results of the input/output behaviour of T, G and C respectively in 1, 2 and 3.

However we do appreciate the need for correctness proofs. So among the lines given for future research, we indicate how we might proceed to prove the correctness of the automatically generated (or systematically derived) code generators.

1.3 The Form of this Thesis

In Chapter 2 we introduce a mechanism of transformation, a production rule system to transform semantic equations into procedures of the programming language BCPL [Ric79]. Chapters 3 to 5 each analyses a different programming language example (from [Sto77]) with increasing levels of difficulty. The main features are respectively: state, environment and continuations. Chapter 6 looks at the Lambda Calculus, defined using both direct and continuation semantics, both with call by value and name. Chapter 7 describes a different approach: instead of starting with a standard denotational semantics, we study the possibility of abstracting implementation ideas at a denotational level, thus starting with an implementation denotational semantics. The objective of this exercise is the generation of more efficient code generators and also to provide the grounds for correctness proofs, as we have shown in [Ras80]. Finally, Chapter 8 looks at some future directions of our research.

Appendix A briefly describes the implementation of a system called ISL (Implementation Semantic Language), which has automatically generated all the examples shown in this thesis ([Ras79] [Ras80] [Ras81] [Ras82]). In Appendix B, we have collected together all transformation rules. Appendix C defines the operators used in the source and target metalanguages. Appendices D and E show the two main examples, Stoy's example language and GEDANKEN.

CHAPTER 2

Production System

In this chapter we introduce the notation used to describe the transformation process which, starting from a DS description of a particular programming language, ends up with a Code Generation Process (CGP) for the same language. From the DS specification, we will deduce (and generate) procedures written in the systems programming language BCPL [Ric79]. These procedures are language dependent and by contrast, under certain design options machine independent. They are the non-primitive operations of a CGP, whose primitive operations and target machine are not specified. In other words, we are regarding the DS as a specification of a CGP. In order to deduce the CGP, we will apply a set of transformations which are formally defined by production rules, each one consisting of a conditioned conversion.

2.1 Metalanguages

2.1.1 Source

Before we can define precisely what we mean by a 'transformation', we have to formalise the metalanguage in which the semantic specifications are written. It is an extension of the typed λ-calculus, the well-formed formulae WFF_s (source) of this metalanguage are:

S:Sou.	source expressions
a:Nam.	non-decorated names
d:Dom.	domain expressions
e:Exp.	lambda-expressions
f:Fun.	function operators
g:Dop.	domain operators
i:Ide.	semantic names
m:Mon.	monadic operators
n:Num.	numerals
o:Opr.	dyadic operators

p:Par. parameter lists
q:Equ. equations
s:Syn. syntactic strings
v:Val. semantic valuators

d ::= i | d* | $d_1 f d_2$ | [d]
e ::= S
f ::= + | x | →
g ::= ? | ?? | '|' | **In**
i ::= a | a_1 | ... | a_n | a* | v | **[s]**
m ::= #
o ::= $\underline{o}$ | $\underline{*}$ | $\underline{+}$ | $\not v$ | = | **Eq** | **Ne** | **Ls** | **Le** | **Gr** | **Ge**
p ::= pp_1 | i | $\langle i_1,\ldots,i_n \rangle$ | null
q ::= vp=e.
S ::= i | n | λp.e | λp. ...e | $e_1 e_2$ | e→e_1,e_2 | me_1 | $e_1 o e_2$ | egd |
[e_1/e_2] | $e_1 \ldots e_n$ | e_1 **where** p=e_2 | $\langle e_1,\ldots,e_n \rangle$ | (e)

The semantics of the different operators are formally defined in Appendix C. Projections ('|') and injections (In) are 'transparent', meaning that they do not intervene actively in the process of transformation; they are kept only as long as they are active carriers of information about functionality; as soon as they become unnecessary they are automatically deleted.

2.1.2 Target

We also have to formalise the final outcome well-formed formulae WFF_t (target), which is a subset of the language BCPL:

T:TAR. target
A:AUX. auxiliary parameters
C:COM. commands
D:DUM. dummy parameters
E:EXP. expressions
I:IDE. identifiers
N:MON. monadic operators
O:OPR. dyadic operators
P:PAR. parameter lists

A ::= **.dest.**(P)A | **.cont.**(P)A | null
C ::= T
D ::= I | I,D | null
E ::= I | E(P)A | $E_1 O E_2$ | N E | E->E_1,E_2 | (E)
N ::= ! | **not**
O ::= ^ | ! | + | - | = | **Eq** | **Ne** | **Ls** | **Le** | **Gr** | **Ge**
P ::= E | E,P | null

T ::= {C} | **let** I(D)A **be** C | **let** I(D)A=**valof** C | **let** I=E | **resultis** E |
E_1:=E_2 | E(P)A | **test** E **then** C_1 **or** C_2 | **if** E **then** C | **unless** E **do** C |
for I=E_1 **to** E_2 **do** C | **switchon** E **into** C | **case** I:C | **endcase** | C_1;C_2

The operators are also defined in Appendix C. In Essex-BCPL, the sequence "). and any character up to (", is equivalent to comma. As an aid to the eye and where appropriate, we will replace ';' by a new line. Also, when the length of a parameter list is not relevant, we will allow possible null parameters. For example: $E_0(P_1, E_1, P_2)$ may denote an instance of a function or procedure call with any P_i optionally null. Similarly, we will allow possible null commands, for example: { C_1; C; C_2 } may denote a block instance with any C_i optionally null.

2.2 Conversions

2.2.1 Rules

Conversions are defined by production rules of the form:

$$e_0 \quad => \quad e_1$$

meaning that e_0 is converted to e_1. Except for the first and last conversion of a given expression, where respectively e_0:WFF_s and e_1:WFF_t, e_0 and e_1 belong to WFF_m (the middle metalanguage), the combination of both source and target which results of the following redefinition of e:Exp and C:COM:

---

m:Mid. middle expressions

M ::= S | T
e ::= M
C ::= M

2.2.2 Conditions

Conversions are conditioned by boolean expressions, introduced by **when**, which 'trigger' the transformations and 'syntactically sugared' by **where** and

rename expressions. In general they are of the form:

$$\begin{array}{rl|c|l} & e_0 & \Rightarrow & e_1 \\ \textbf{where} & e_0 = e_2 & & \textbf{where } e_1 = e_3 \\ \textbf{when} & \langle\text{condition}\rangle & & \textbf{rename } i \Rightarrow I \end{array}$$

This conversion indicates that under both **where** definitions, if <condition> is satisfied, then e_0 is to be transformed to e_1. The **rename** construction helps to shorten the length of conversions, it indicates the substitution of I for i in e_1 (equivalent to $[I/i]e_1$). Expressions in **when**, **where** and **rename** clauses are defined by:

- Informal text.
- Usual boolean and arithmetic operators.
- Test for domain membership ':' and domain re-definition **In**.
- Test for sub-domain $\subseteq$.
- Textual equivalence $\equiv$ and non-equivalence $\not\equiv$.

For example, the rule to transform an expression involving the minimal fix point finder of a state to state abstraction is:

$$\begin{array}{ll|c|l} \textbf{when } i\!:\![S \to S] & \text{Fix}(\lambda i.e) & \Rightarrow & \{\ \textbf{let } i = \text{here(COD)};\ e\ \} \\ & & & \textbf{rename } i \Rightarrow \text{restart.code} \end{array}$$

This rule is specifying a transformation for every expression of the form indicated by the left hand side, if and only if, the bound variable 'i' belongs to the domain [S→S]. The result of such a rule is indicated by the form of the right hand side. The **rename** construction specifies that the name 'i' must be substituted by 'restart' in every sub-expression of the right hand side; this includes the **let** declaration and also any occurrence of 'i' inside 'e'.

Textual equivalence $\equiv$ and non-equivalence $\not\equiv$ are used to test the particular instance of a WFF_m expression. For example:

$$e \equiv e_1 e_2 \text{ and } e_1 \not\equiv \lambda p.e_3$$

is equivalent to

IsApplication(e) and not IsAbstraction(FunctionPart(e))

2.3 Transformations

A transformation is the process of actively filtering a WFF_s through all possible conversions, under the constraints indicated by the conditions. The productions are divided into different disjoint subsets. Among them we can find syntactic, semantic and optimising transformations. The following is a list of all subsets:

1 Normalisation
2 State Analysis
3 Syntactic Transformations
4 Splitting Continuations
5 Destination Analysis
6 Continuation Analysis
7 Environment Analysis
8 Optimising Continuations
9 Optimising Transformations
A BCPL

Each rule will be numbered as [Rn.i], where n is one of <1..9, A> as described above, and i is the rule number within n. To illustrate the nature of the transformations, we shall indicate the intermediate steps by 'snapshots'.

The information required to perform the transformations is obtained, firstly, from the concrete semantics, i.e: from the particular representation of the semantic functions, and secondly from certain domains that our system has to know about. To avoid having to write semantic equations with name dependency, we introduced the idea of a 'domain of interest', which for a given compiler-generation process must be given. The following is a list of all domains of interest known to our system:

ANS Answers	STA States
ENV Environments	REG Registered values
TEM Templates(functions and procedures)	THU Thunks(call by name)
BOO Compile-Time booleans	INT Compile-Time integers
LOC Locations	QUO Quotations

The importance of these domains, is that they characterise, from a code generation standpoint, the minimal information required to transform the programming languages that we will consider.

2.4 An Example

Snapshot 2.1: The Lambda Calculus. Original Specification

Syntactic Categories

```
i:Ide.                                   identifiers(undef)
e:Exp.                                   lambda-expressions
```

Syntax

$e ::= i \mid \textbf{Lam}\ i.e_1 \mid e_1e_2$

Semantic Domains

```
  N.                                     basic values
e:E=[N + F].                             values of expressions
  F=[E → E].                             function values
p:U=[Ide → E].                           environments
```

Semantic Domains of 'Interest'

```
  ENV=U.                                 environments
  REG=E.                                 registered values
  TEM=F.                                 templates
```

Semantic Equations

$E:[Exp \to U \to E].$ (2.1.1)

$E[i]p=$
$\quad p[i].$ (2.1.2)

$E[\textbf{Lam}\ i.e_1]p=$
$\quad Strict(\lambda e.E[e_1](p[e/[i]]))\ In\ E.$ (2.1.3)

$E[e_1e_2]p=$
$\quad (\lambda ee'.(e|F)e')(E[e_1]p)(E[e_2]p).$ (2.1.4)

We have introduced the mechanism of transformation. In the following chapters, we will follow [Sto77] examples; for every example language, we will define the appropriate conversion rules. To conclude this introductory chapter, we will follow the transformation process for the Lambda Calculus in its direct form with call by value. The original specification for this language is shown in Snapshot 2.1.

2.4.1 Syntactic Transformations

The first transformation set for this language is syntactic. It consists of rules which do not convey any implementation detail, only necessary to shape the CGP in a procedural form. It can be understood as a transformation set which just projects concrete semantic constructs of the original metalanguage to equivalent constructs of a different one. The difference is that the new procedural metalanguage is more suited to an algorithmic interpretation. In Snapshot 2.2 we show the result of such transformations. Note that we are 'tagging' transformations with their rule number even though these rules will be defined in the following chapters. However, to show the flavour of the transformation process, in this example we will display some transformations. Here is R1.1:

```
             __|      |__ if n>1
               |      |     let v node p be
               |      |     switchon type^node into
 v[s1]p=e1.    |      |     { case [s1]: e1; endcase
...            |  =>  |        ...
 v[sn]p=en.    |      |       case [sn]: en; endcase
               |      |     }
               |      |   if n=1
             __|      |__   let v node p be e1
```

Also note that at this stage in the analysis, E is declared (2.2.1) in a similar way as procedures are declared in BCPL, however, it is used as a

Snapshot 2.2: The Lambda Calculus. Syntactic Transformations

```
let E(node, p) be switchon type^node into              by R1.1, R3.1   (2.2.1)
{ case [i]:
   p([i]); endcase                                      by R1.1, R3.2   (2.2.2)

  case [Lam i.e1]:
   (λe.E([e1], p([e/[i]]))) In E; endcase
                                         by R1.1, R3.2/twice, R3.6      (2.2.3)

  case [e1e2]:
   { let e = E([e1], p); let e' = E([e2], p); e|F(e') }; endcase
                       by R1.1, R1.2, R3.2/3 times, R3.3, R3.4          (2.2.4)
}
```

function in (2.2.3) and (2.2.4). This is a symptom of showing an intermediate stage of the transformation process. If we were to provide a proof of correctness for each stage in the analysis, then we would have to define precisely the semantics of each intermediate metalanguage.

2.4.2 **Semantic Transformations**

Rules within this transformation set are driven, not only by the form of the concrete semantics, but also by those domains about which the transformation system knows, i.e: the domains of 'interest'. For example, the domains REG, TEM and ENV of Snapshot 2.1 are respectively but not exclusively driving the destination, continuation and environment analysis.

Destination Analysis: After analysing the use of temporary values the transformation process looks like Snapshot 2.3, where reg, first.reg and first.par can be understood as fast registers or as pointers to an activation record. Note that every semantic function giving a result in a summand of the definition of the domain of interest REG, gets a **.dest.**(reg) added to its parameter list. trans.load is a procedure introduced by the transformation process. Its first parameter indicates the type of object to be loaded; this information is used for compile or run-time type checking.

Snapshot 2.3: The Lambda Calculus. Destination Analysis

```
let E(node, p).dest.(reg) be switchon type^node into          by R5.1   (2.3.1)
{ case [i]:
   p([i]).dest.(reg); endcase                                  by R5.3   (2.3.2)

   case [Lam i.e1]:
     trans.load(F, E([e1], p([first.par/[i]])).dest.(first.reg)).dest.(reg)
      In E; endcase                  by R5.3/twice, R5.8, R5.9, R5.10   (2.3.3)

   case [e1e2]:
     E([e1], p).dest.(reg)
     E([e2], p).dest.(reg+1)
     reg|F(reg+1).dest.(first.reg); endcase    by R5.4/twice, R5.11   (2.3.4)
}
```

The second parameter is the object to be loaded into the destination indicated by the third parameter, the **.dest.(reg)** construction. This mechanism requires all regs to be carriers of type information. In this example, the object to be loaded is a function value and it is up to trans.load to plant the appropriate code to load a closure.

Recall the analogy between the lexical and syntax analysis problem, with the semantic one, as presented in the Chapter 1: Primitive procedures and functions like trans.load, introduced by the transformation process, are to the code generation process what primitives like scann.next.character and look.up.ident are to a scanner, or what scan.next.symbol and make.node are to a parser. These activities are not part of the initial specification, but are deduced and left semi-specified. In this respect, we regard a specification (lexical, syntactic or semantic) as a schema for machine translation with slots which have to be filled in.

Continuation Analysis: Next we correlate code fragments. This transformation set is called Continuation Analysis, but note that its effect is felt in languages whose specifications do not have continuations. This is because we

Snapshot 2.4: The Lambda Calculus. Continuation Analysis

```
let E(node, p).dest.(reg) be switchon type^node into                no change
{ case [i]:                                                         no change

  case [Lam i.e1]:
    { let ntry.domF = forward(F)
      let exit.code = forward(COD)
      let skip.code = forward(COD)
      trans.jump.to(skip.code)
      trans.entry(ntry.domF, node)
      E([e1], p([first.par/[i]])).dest.(first.reg)
      trans.exit(exit.code, node)
      fix.here(skip.code)
      trans.load(F, ntry.domF).dest.(reg)
    }; endcase                                               by R6.4   (2.4.3)

  case [e1e2]:
    E([e1], p).dest.(reg)
    E([e2], p).dest.(reg+1)
    trans.call(reg|F, reg+1).dest.(first.reg); endcase       by R6.6   (2.4.4)
}
```

associate continuations with pointers to code, and code is clearly produced regardless of the presence of continuations explicitly in the definition. In the Lambda Calculus, the two transformations at this stage are the specification of abstraction and of application, as can be seen in Snapshot 2.4. The transformation process, carrying considerable expert information, has isolated here 'crucial code fragments'. The places for entry to, exit from and call to a function have been recognised. The domain of interest TEM indicates which is the domain of functions and together with the particular form of the concrete semantics, provides the necessary information for the transformation process to insert appropriate procedure calls whose task is to generate code for these three crucial places. The fact that the code associated with the body of a function must not be executed at declaration time results in the 'skip' statements. Forward references are also handled by inserting appropriate procedure and function calls. To illustrate how this expert information is reflected by the transformation rules, here is R6.4:

```
                                       __       __
                                         |     |  { let ntry.code = forward(DOM(e_1))
                                         |     |    let exit.code = forward(COD)
                                         |     |    let skip.code = forward(COD)
                                         |     |    trans.jump.to(skip.code)
                                         |     |    trans.entry(ntry.code, node)
when e_1:TEM    e(P_0, e_1, P_1)A        | =>  |    e_1
                                         |     |    trans.exit(exit.code, node)
                                         |     |    fix.here(skip.code)
                                         |     |    e(P_0, ntry.code, P_1)A
                                       __|     |__}
```

Environment Analysis: Assuming a block structured use of the environment, we argue that a direct 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 (a symbol table) and primitives to declare search and undeclare denoted elements. Environments disappear from parameter lists and a structure is used to recover from declarations to the global environment (a stack or A-List). The main transformation rule of this analysis is R7.1, defined as follows:

```
                                 __       __
                                   |     |  { let old.env=this.env
                                   |     |    e
              e_0(P_0, e, P_1)A    | =>  |    e_0(P_0, P_1)A
                                   |     |    reset(old.env)
when e:ENV and e≠i               __|     |__}
```

After these transformations, our example looks like Snapshot 2.5. Note how this text looks almost like the text of a program written in BCPL. At this stage in this example, the only issues that still look denotational are the 'node references' enclosed between '[' and ']' and the projection '|'.

The primitives look.up and declare might or might not plant code, depending on the structure of the particular denoted value under scrutiny. In (2.5.2) look.up has to plant code to load a value into the destination reg. In

Snapshot 2.5: The Lambda Calculus. Environment Analysis

```
let E(node).dest.(reg) be switchon type^node into             by R7.5   (2.5.1)
{ case [i]:
   look.up([i]).dest.(reg); endcase                            by R7.4   (2.5.2)

 case [Lam i.e1]:
   { let ntry.domF = forward(F)
     let exit.code = forward(COD)
     let skip.code = forward(COD)
     trans.jump.to(skip.code)
     trans.entry(ntry.domF, node)
     { let old.env = this.env
       declare(domain.of(first.par), first.par, [i])
       E([e1]).dest.(first.reg)
       reset(old.env)
     }
     trans.exit(exit.code, node)
     fix.here(skip.code)
     trans.load(F, ntry.domF).dest.(reg)
   }; endcase                                           by R7.1, R7.2   (2.5.3)

 case [e1e2]:
   E([e1]).dest.(reg)
   E([e2]).dest.(reg+1)
   trans.call(reg|F, reg+1).dest.(first.reg); endcase
                                                        by R7.6/twice   (2.5.4)
}
```

(2.5.3), declare plants code to store the parameter into a temporary location updating the global symbol structure to reflect the binding to this temporary. look.up plants code because the Destination Analysis recognised denoted values as belonging to REG. In a different language with denoted values not in REG, the look up request will not plant any code, this would be signalled by the absence of the **.dest.()** construction. declare plants code if the object to be declared is contained in a run-time temporary place.

In 'dynamically' bound languages, the global symbol structure has to be maintained at run time, hence, all operations of 'statically' bound languages, which maintain a compile-time global symbol structure, have to plant code to maintain a similar one at run-time. These operations are those dictated by the same primitives declare, look.up, reset and by the statement: let old.env=this.env.

This mechanism to eliminate the environment is applicable, provided declarations are block structured and environments are used in such a way that it is not the case of two different environments accessible at the same moment. It can only be applicable when one single global symbol structure can represent the whole environment. If such a condition does not hold, then transformations will have to preserve the environment as one more parameter to every code generation procedure.

2.4.3 Optimising Transformations

Optimisations are not essential, but strengthen our main objective of generating a CGP which is efficient and usable. In Snapshot 2.6 we can see that the CGP relies on a 'tree-weighting' algorithm [Bor79] to allocate fast registers (if first.reg is seen as such, and not as an activation record pointer).

2.4.4 BCPL

Finally, an interface from a syntax analyser provides the syntactic information to rename node references with the appropriate names or selectors. Also at this stage we rename curly functions and domain names. The final version is shown in Snapshot 2.6. It can be successfully compiled in BCPL, and, if provided with a machine interface and syntax analyser it is a compiler for the Lambda Calculus.

Snapshot 2.6: The Lambda Calculus. BCPL

```
let trans.E(node).dest.(reg) be switchon type^node into   by RA.1   (2.6.1)
{ case T..Ident:
    look.up(node).dest.(reg); endcase               by RA.1/twice   (2.6.2)

  case N2..Abstraction:
    { let ntry.domF = forward(D..F)
      let exit.code = forward(D..COD)
      let skip.code = forward(D..COD)
      trans.jump.to(skip.code, true.jump)
      trans.entry(ntry.domF, node)
      { let old.env = this.env
        declare(domain.of(first.par), first.par, p1^node)
        trans.E(p2^node).dest.(first.reg)
        reset(old.env)
      }
      trans.exit(exit.code, node)
      fix.here(skip.code)
      trans.load(D..F, ntry.domF).dest.(reg)
    }; endcase              by R8.2, RA.1/3 times, RA.2/5 times   (2.6.3)

  case N2..Application:
    trans.E(p1^node).dest.(reg)
    test weight^p2^node=max.reg
    then { let old.env = this.env
           let dmp.loc = trans.dump(reg)
           trans.E(p2^node).dest.(reg)
           trans.call(dmp.loc, reg).dest.(first.reg)
           reset(old.env)
         }
    or   { let nxt = next(reg)
           trans.E(p2^node).dest.(nxt)
           trans.call(reg, nxt).dest.(first.reg)
         }; endcase         by R9.1, RA.1/5 times, RA.2/3 times   (2.6.4)
}
```

CHAPTER 3

State

In Chapter 2 we have introduced the production system that formalises the transformation process by which we will produce the code generation processes, but no transformation rules were described. In this chapter we start describing the correspondence that is the concern of this thesis. We will consider a simple language of flow diagrams based on [Sto77] Table 9.1, as reproduced in Snapshot 3.1 below.

The main topic of this chapter will be the State Analysis. However, in order to place in context such analysis we have to prelude it with the Normalisation analysis and postlude it with Syntactic and Semantic transformations. So in effect we will define all transformation rules that allow us to transform the original specification (in the source metalanguage WFF_s) to the final CGP (in WFF_t).

In the semantic specification of Snapshot 3.1, the state has been explicitly written everywhere; even where it can easily be eliminated. This can be achieved, for example, by use of the composition operator in (3.1.3) or by use of the 'star' operator in (3.1.4) to (3.1.6) and (3.1.10). This short hand version will be analysed shortly, once we have introduced the transformations to perform state elimination when explicitly used.

Note that although there are no commands producing side effects, the specification is designed as if there are. In the following sections we will enlarge the language with assignments. For the moment, to avoid clustering, we limit our analysis to the equations listed below.

Snapshot 3.1: Algebraic Language of Flow Diagrams. Original Specification

Syntactic Categories

c:Com. — commands
e:Exp. — expressions

Syntax

c ::= **Dummy** | **If** e **Then** c_1 **Else** c_2 | c_1;c_2 | **While** e **Do** c_1 |
c_1 **Repeatwhile** e
e ::= **True** | **False** | **If** e_1 **Then** e_2 **Else** e_3

Semantic Domains

t:T=[{ TRUE } + { FALSE }]. — truth values
s:S. — machine states
c:C=[S → S]. — command values
W=[S → T]. — expression values

Semantic Domains of 'Interest'

REG=W. — registered values
STA=S. — states

Semantic Equations

C:[Com → C]. (3.1.1)

C[Dummy]=
λs.s. (3.1.2)

C[c_1;c_2]=
λs.**C**[c_2](**C**[c_1]s). (3.1.3)

C[If e Then c_1 Else c_2]=
λs.**E**[e]s→**C**[c_1]s,**C**[c_2]s. (3.1.4)

C[While e Do c_1]=
Fix{λcs.**E**[e]s→c(**C**[c_1]s),s}. (3.1.5)

C[c_1 Repeatwhile e]=
Fix{λcs.{λs'.**E**[e]s'→cs',s'}(**C**[c_1]s)}. (3.1.6)

E:[Exp → W]. (3.1.7)

E[True]=
Strict(λs.TRUE). (3.1.8)

E[False]=
Strict(λs.FALSE). (3.1.9)

E[If e_1 Then e_2 Else e_3]=
λs.**E**[e_1]s→**E**[e_2]s,**E**[e_3]s. (3.1.10)

3.1 Normalisation

In Snapshot 3.1, we observe that **C** and **E** are recursively defined by cases over their different syntactic alternatives; This isolates, for each alternative, a semantic value. From the domains to which these values belong and from the concrete semantics, we wish to discover code generation actions. Because of the nature and structure of code generation, these actions will each be associated with a different syntactic construction; a different node of a parse tree. It seems then that the definition by cases over the different syntactic alternatives is an appropriate structure both for semantic definitions and code generation. But the form of a simultaneous set of mutually recursive equations can be algorithmically expressed by a set of mutually recursive procedures. In this case there are two semantic valuators, hence two procedures. Preserving this structure, we will transform the sequence of equations into two **let** declarations, each followed by a **switchon** statement selecting similarly by cases. To do this, we need to restrict equations to be homogeneous in their parameter lists, each case must be written with the same number of parameters. A pre-processor could be easily defined to automate this process, but this is not our main concern. We will assume that equations are homogeneous in this way and formalise this transformation by the following conversion:

```
                 __      __
                   |    |  if n>1
                   |    |     let v node p be
                   |    |     switchon type^node into
   v[s1]p=e1.      |    |     { case [s1]: e1; endcase
   ...             | => |        ...                          [R1.1]
   v[sn]p=en.      |    |       case [sn]: en; endcase
                   |    |     }
                   |    |  if n=1
                 __|    |__   let v node p be e1
```

Next, two transformations which are not more than syntactic 'de-sugaring'.

Snapshot 3.2: Algebraic Language of Flow Diagrams. Normalisation

```
let C node be switchon type^node into                  by Rl.1      (3.2.1)
{ case [Dummy]:
  λs.s; endcase                                        by Rl.1      (3.2.2)

  case [c1;c2]:
  λs.C[c2](C[c1]s); endcase                            by Rl.1      (3.2.3)

  case [If e Then c1 Else c2]:
   λs.E[e]s→C[c1]s,C[c2]s; endcase                     by Rl.1      (3.2.4)

  case [While e Do c1]:
   Fix(λc.λs.E[e]s→c(C[c1]s),s); endcase          by Rl.1, Rl.2     (3.2.5)

  case [c1 Repeatwhile e]:
   Fix(λc.λs.(λs'.E[e]s'→cs',s')(C[c1]s)); endcase  by Rl.1, Rl.2   (3.2.6)
}

let E node be switchon type^node into                  by Rl.1      (3.2.7)
{ case [True]:
   Strict(λs.TRUE); endcase                            by Rl.1      (3.2.8)

  case [False]:
   Strict(λs.FALSE); endcase                           by Rl.1      (3.2.9)

  case [If e1 Then e2 Else e3]:
  λs.E[e1]s→E[e2]s,E[e3]s; endcase                     by Rl.1      (3.2.10)
}
```

The first one is required for the example language of this chapter, however the second one is introduced now, but only required later. A cross reference of rule numbers and the pages where they are defined and used is given in Appendix B.

$$\lambda ip.e \quad \Rightarrow \quad \lambda i.\lambda p.e \qquad [Rl.2]$$

$$e_0 \text{ Where } p=e_1 \quad \Rightarrow \quad \{\ \textbf{let}\ p=e_1\ ;\ e_0\ \} \qquad [Rl.3]$$

Note that the Where construction above, is a WFF_s expression, and not a condition.

These transformations are the first step towards a translation into BCPL. Semantic equations are written in a mathematical metalanguage, and we have just projected an image, of a valuator's sequence of equations, into a procedure selecting by cases over one of its parameters. Snapshot 3.2 shows the shape of the semantic equations after these transformations, they are 'tagged' with the transforming rule number.

3.2 State Analysis

When analysing languages in a semantic world which includes a 'machine state' it is important to realise the distinction between 'compile-time' and 'run-time' activities. In an early paper, C. Strachey pointed out the distinction between:

$$L[e_1 \rightarrow e_2, e_3]s \quad = \quad L(\text{if}(R[e_1]s)\langle[e_2],[e_3]\rangle)s.$$

and

$$L[e_1 \rightarrow e_2, e_3]s \quad = \quad \text{if}(R[e_1]s)\langle L[e_2],L[e_3]\rangle s.$$

> "...the first expression, in which the choice is made between alternative [e]'s to translate and run, corresponds to an interpretive or "translate as you run" scheme, while the second, in which the choice is between already translated operators, corresponds to the more common scheme of separate compiling and running phases." [Str66]-p206

These two equations define precisely the same value, however, C. Strachey was aware that it is possible to give two different (algorithmic) interpretations to the concrete semantics, one corresponding to an interpreter, the other to a compiler.

Our primary objective is to analyse the relationship between a semantic specification and an associated code generation process, hence we are interested in the second form. Moreover, we are interested in completely splitting those actions that correspond to a compiler from those that correspond to the generated code. Consider:

$$\begin{array}{lll} M & : & [PRO \rightarrow [STA \rightarrow STA]] \\ C & : & [PRO \rightarrow COD] \\ H & : & [COD \rightarrow [STA \rightarrow STA]] \end{array}$$

Where M is the semantic function abstracting the meaning of a program p:PRO in a language with state s:STA. C is a compiler producing some code c:COD, which is in turn 'run' on the hardware of a particular machine H. We clearly require M=H o C. Hence, a compiler performs an action that ends up with a representation of the [STA → STA] function. The state analysis uses this observation, so that each equation provides a function in [STA → STA] for each syntactic alternative. The concrete semantics thus is a representation of a [STA → STA] function and successive refinements transform it into the code generator, which as we saw above produce, in turn, such a representation; namely the code.

3.2.1 No Copies of the State Allowed

Before starting with the analysis refered to above, we have to restrict the use of the state. It is very easy to write semantic equations that force copying the state, for example imagine a definition of sequencing like: $C[c_0;c_1]s=(\lambda s'.C[c_1]s)(C[c_0]s)$. This equation specifies that both $[c_0]$ and $[c_1]$ must be evaluated in the same state s. This implies copying the side effects of $[c_0]$ represented by s'. With the hardware configuration of todays machines, this is a very expensive operation (although there are experiments like the 'highly reliable' system of the University of Newcastle [Ran75]), and as J. Stoy points out:

> "...The semantics of this would involve more than the 'single thread' treatment of s that we practise in the present equations; it is of course expensive in memory." [Sto77]-p231

Hence, we will not consider languages whose implementation involves maintaining a copy of the state. This is a pre-condition of the correspondence that we are describing.

We are now ready to continue analysing the simple language of flow diagrams. We will develop, first, a sequence of transformations to eliminate the state. This transformation process works by 'brute force', it can be applied provided we ensure that the programming language under scrutiny does not require its implementation to maintain a copy of the state. Secondly, we will develop an alternative method: rewriting the semantic equations by 'structuring' the state with appropriate operators and primitive functions. The state will, then, never be explicitly used in any semantic valuator. The associated transformations will follow the structure of the new operators and primitive functions, and hence this second method will be shown preferable. However, both methods will result in BCPL programs which are, in effect, equivalent. The only differences are the primitive procedures introduced by the transformation process (P1), in place of the procedures (P2) which correspond to the primitive functions introduced in the semantic specification. Primitive functions in the original specification are preserved through the the transformation process and appear as primitive procedures in the target CGP. Under the same interpretation of P1 and P2, both methods produce equivalent code generators.

3.2.2 First Method - Elimination

We first observe that this language fulfils the pre-condition described above; there are no copies of the state; it is passed around as a 'single thread'. But note that to say so means that we have to explicitly look at every expression involving a state. There are five ways that it is used, namely: identity, abstraction, application, conditional and as a strict abstraction. We will consider each one in turn:

Identity: The identity function specifies that the state must be left exactly as it is, this is a trivial case in that it implies that no code is generated. We indicate this by an empty block:

when i:STA λi.i => {} [R2.1]

Abstraction and Application: Abstracting on the state, means that the body of the abstraction will perform an action in a new state, which will be produced by some code previously generated, it is not the concern of the compiler to look at the state, but simply to generate the appropriate code that will act in the new state, hence we directly eliminate such an abstraction:

when i:STA λi.e => e [R2.2]

States passed as parameters are either directly eliminated if they are simple variables; or associated with blocks of code if they are expressions:

when i:STA $e_0 i$ => e_0 [R2.3]

when e:STA $e_0 e$ => { e **In** COD; e_0 } [R2.4]

COD is an 'internal' domain of interest associated with code structures. It is defined in terms of the 'user defined' domains of interest STA and ANS:

COD = [[STA→STA] + [STA→ANS]] [D1]

In this case COD is used in a domain redefinition:

e In D = denotes a change of functionality [D2]

We will see later how **where** clauses look for this domain when a code structure is required.

Our discussion has and will continue to use the terms 'function' and 'functionality' with regard to the objects and properties of a denotational definition. Since our system of productions is manipulating the semantics concretely - that is as text - we see that it is committed to an algorithmic, rather than functional, reading of the specification. So, strictly, we should perhaps say 'procedure' for 'function' and 'data-type' for 'functionality'.

Why these transformations? Our interest is to discover the form and structure of a CGP. The semantic specification abstracts the machine configuration as a state to state function. For example:

```
s:S=[I* + O*].                                            States
i:I.                                                      Input Values
o:O.                                                      Output Values
Read:[S → [I x S]].
Write:[O → S → S].
```

Under the above definitions, a concrete semantic expression might involve sub-expressions like: Read(s) or Write(o)s. The corresponding statements in the associated code generation procedures will not need the state variables. Under the restriction of one single state thread, the code CGP does not require that reference whatsoever. The semantic specification is abstracting up to a run-time activity; the CGP, however, up to a code generation stage. The two sub-expressions above, after the application of the R2.3 will look respectively as: Read() and Write(o).

Conditional: If a state variable is used as one of the branches of a double arm conditional we can eliminate that branch turning it into a null block:

when i:STA $\quad e \rightarrow e_1, i \quad \Rightarrow \quad e \rightarrow e_1, \{\}$ [R2.5]

when i:STA $\quad e \rightarrow i, e_1 \quad \Rightarrow \quad e \rightarrow \{\}, e_1$ [R2.6]

Note that (apart from the trivial case of both branches selecting the same

Snapshot 3.3: Algebraic Language of Flow Diagrams. State Analysis

```
let C node be switchon type^node into                                   no change
{ case [Dummy]:
    {}; endcase                                                  by R2.1   (3.3.2)

  case [c1;c2]:
    C[c1]; C[c2]; endcase                        by R2.2, R2.3, R2.4     (3.3.3)

  case [If e Then c1 Else c2]:
    E[e]→C[c1],C[c2]; endcase                   by R2.2, R2.3/3 times     (3.3.4)

  case [While e Do c1]:
    Fix(λc.E[e]→{ C[c1]; c },{}); endcase
                                        by R2.2, R2.3/twice, R2.4, R2.5   (3.3.5)

  case [c1 Repeatwhile e]:
    Fix(λc.{ C[c1]; E[e]→c,{} }); endcase
                                 by R2.2/twice, R2.3/3 times, R2.4, R2.5  (3.3.6)
}

let E node be switchon type^node into                                   no change
{ case [True]:
    TRUE; endcase                                             by R2.2, R2.7   (3.3.8)

  case [False]:
    FALSE; endcase                                            by R2.2, R2.7   (3.3.9)

  case [If e1 Then e2 Else e3]:
    E[e1]→E[e2],E[e3]; endcase                  by R2.2, R2.3/3 times   (3.3.10)
}
```

state variable) it can not be the case that both branches select a variable as this means maintaining a copy of the state.

Strict: A strict abstraction on the state implies that if that function is applied to an improper state the result should also be improper. This means that nothing can be said later about it. This is, in general, what happens when a program fails due to an improper use of the state. For example, when a command fails to terminate, it is up to the hardware to tell whether or not the state can be examined (through a dump for example). It seems then natural to assume that such a strict function does not concern the compiler, it is a hardware activity. To assume the contrary, means that appropriate instructions will have to be planted to 'run-time check' whether the state is proper or not. Such a check is too expensive to be done by software, we prefer to rely on the hardware. This implies a condition on the form of the input semantics; they are always strict on the state. So when the predefined WFF_s identifier <u>Strict</u> is explicitly used in a state abstraction we can directly eliminate it, defining:

when i:STA Strict(λi.e) => λi.e [R2.7]

Having defined the transformations that are required to eliminate the state, we apply them to Snapshot 3.2 obtaining Snapshot 3.3.

3.2.3 Second Method - Structuring

The method outlined above is rather dangerous, in the sense that there is no check to detect that the 'single-thread' condition holds. Also as J. Stoy points out:

> "...The absence of the symbol s from the equations ... helps to emphasise that we must normally ensure that the 'same' state s is not used at two arbitrary separated points in a formula: that is to say, we must avoid defining the semantics so that implementation involves making a copy of the state of the machine, which is not economically feasible." [Sto77]-p231

Definitions: We first introduce two operators which were not discussed when describing the source metalanguage WFF_s.

$$
\begin{array}{rcl}
.\underline{o}. & : & [[[D \rightarrow D_1] \times [D_1 \rightarrow D_2]] \rightarrow [D \rightarrow D_2]] \\
f & : & [D \rightarrow D_1] \\
g & : & [D_1 \rightarrow D_2] \\
d & : & D \\
(f\ \underline{o}\ g)d & = & g(fd)
\end{array}
\qquad [D3]
$$

This operator is composition. We prefer this form, because we wish to read equations from left to right. The other composition operator (o), forces one to visually scan twice doing two passes over an equation. Some authors (see [Ten76]) denote this form of composition with a semicolon, hence there is an association with the sequencing operator of some programming languages (like most ALGOL60 offspring). We will interpret this operator as sequencing under certain domain configurations.

$\underline{o}$ is normally used for commands which are only involved in a COD transformation. For expressions which produce a value (without side effects) we define:

$$
\begin{array}{rcll}
.\underline{+}. & : & [[[D_1 \rightarrow D] \mathrm{x} [D \rightarrow [D_1 \rightarrow D_2]]] \rightarrow [D_1 \rightarrow D_2]] & \\
 & & \text{for any } D_1 \text{ and } D_2\text{. But not } D \subseteq STA & \\
f & : & [D_1 \rightarrow D] & \\
g & : & [D \rightarrow [D_1 \rightarrow D_2]] & \\
d_1 & : & D_1 & \\
(f \underline{+} g)d_1 & = & g(fd_1)d_1 & [D4]
\end{array}
$$

D may not be a state, to avoid any state saving. $\underline{o}$ and $\underline{+}$ eliminate the explicit use of the state of (3.1.3) to (3.1.6) and (3.1.10). For the rest we need two primitive functions, one is the identity function on the state which is a predefined WFF_s identifier, defined by:

$$
\begin{array}{rcll}
Is & : & [STA \rightarrow STA] & \\
Is & = & Strict(\lambda i.i) & [D5]
\end{array}
$$

The second named <u>Load</u> replaces the use of <u>Strict</u> and it is defined as a primitive function in the semantic specification of Snapshot 3.4.

Our treatment of predefined functions (like <u>Is</u>) and defined primitives (like <u>Load</u>) leaves their algorithmic interpretation to the machine interface. This is a way to 'hide' implementation details and strategies, avoiding becoming too involved in crucial decisions. Compare this to the explicit use of the

Snapshot 3.4: Flow Diagrams State-Structured. Original Specification

Modifications to Snapshot 3.1

Semantic Primitive

Load:[T $\to$ W].

Load=

λt.Strict(λs.t).

Semantic Equations

C:[Com $\to$ C]. (3.4.1)

C[Dummy]=
Is. (3.4.2)

C[c_1;c_2]=
C[c_1] $\underline{o}$ C[c_2]. (3.4.3)

C[If e **Then** c_1 Else c_2]=
E[e] $\underline{+}$ λt.t$\to$C[c_1],C[c_2]. (3.4.4)

C[While e **Do** c_1]=
Fix{λc.{E[e] $\underline{+}$ λt.t$\to$C[c_1] $\underline{o}$ c,Is}}. (3.4.5)

C[c_1 **Repeatwhile** e]=
Fix{λc.{C[c_1] $\underline{o}$ E[e] $\underline{+}$ λt.t$\to$c,Is}}. (3.4.6)

E:[Exp $\to$ W]. (3.4.7)

E[True]=
Load TRUE. (3.4.8)

E[False]=
Load FALSE. (3.4.9)

E[If e_1 **Then** e_2 Else e_3]=
E[e_1] $\underline{+}$ λt.t$\to$E[e_2],E[e_3]. (3.4.10)

---

function Strict in the previous method. Hence, structuring with primitive functions is a way of preserving, throughout the transformation process, the meaning specified by the original semantic definition.

In Snapshot 3.4 we show the semantic equations for the same simple algebraic language of flow diagrams using the new operators and primitive functions. Note that the state is never explicitly used in any semantic equation. It is only used in the primitive functions or implicitly in the operators. This helps to emphasise, but does not guarantee, the absence of expressions involving a copy of the state. In a language like this one a construction potentially producing side effects on the state, used in a construction which does not produce a side effect, may require copying the state. For example: suppose we embed a command, which has side effects, inside an expression, which has not, and define:

e ::= c "In" e | ...
E[c "In" e]=C[c] $\underline{o}$ E[e].

Discovering this, and any other pathological case, means proving by induction that the state is consistently used in every possible subexpression. We are assuming, as a pre-condition on the input semantics, that such mixed expressions do not occur. We do not pursue this matter further. We now consider the necessary transformations for the structured version.

Identity: Firstly, a transformation for the explicit use of the identity function on the state.

Is => {} [R2.8]

Reversed Composition for Commands: Secondly, the association of $\underline{o}$ with sequencing is made only in the case that the first expression produces a side effect (a COD function). However, if the first expression only reads the state ([STA→D]) then we associate $\underline{o}$ with application:

when e_0:[STA→D_1] e_0 $\underline{o}$ e_1 => C [R2.9]
where C = { e_0; e_1 } if D_1=STA or D_1=ANS (i.e: e_0:COD)
C = e_1(e_0 In D_1) otherwise

$\underline{o}$ is a generic operator. In the particular instance of .$\underline{o}$.:[[CODx[STA→D_2]]→[STA→D_2]] it can be understood as specifying the link between two blocks of code. This becomes even more evident when D_2=STA or

D_2=ANS, in which case .$\underline{o}$.:[[CODxCOD]→COD].

Composition for Expressions: Finally, $\underline{+}$, which differs from $\underline{o}$ in the value that is passed across the boundary of the two blocks of code, hence it is associated with application:

$$e_0 \underline{+} e_1 \quad \Rightarrow \quad (e_1 \text{ In } [D \to D_2])(e_0 \text{ In } D) \qquad [R2.10]$$

when for any domain D and D_2 e_0:[STA→D] and e_1:[D→[STA→D_2]]

$\underline{+}$ is a generic operator, in the particular instance of .$\underline{+}$.:[[[STA→D]x[D→[STA→D_2]]]→[STA→D_2]] it links two blocks of code; The first block produces a value which is in turn passed to the second.

As an example, here is how we analyse the while-loop:

State Elimination:

Fix(λcs.E[e]s→c(C[c_1]s),s)	From (3.1.5)
Fix(λc.λs.E[e]s→c(C[c_1]s),s)	By R1.2-(3.2.5)
Fix(λc.λs.E[e]s→{ C[c_1]s; c },s)	By R2.4
Fix(λc.λs.E[e]→{ C[c_1]; c },s)	By R2.3/twice
Fix(λc.λs.E[e]→{ C[c_1]; c },{})	By R2.5
Fix(λc.E[e]→{ C[c_1]; c },{})	By R2.2-(3.3.5)

Structuring the State:

Fix(λc.(E[e] $\underline{+}$ λt.t→C[c_1] $\underline{o}$ c,Is))	From (3.4.5)
Fix(λc.(E[e] $\underline{+}$ λt.t→C[c_1] $\underline{o}$ c,{}))	By R2.8
Fix(λc.(E[e] $\underline{+}$ λt.t→{ C[c_1]; c },{}))	By R2.9
Fix(λc.(λt.t→{ C[c_1]; c },{})(E[e]))	By R2.10-(3.5.5)

An implicit domain transformation, which is not shown in our snapshots, is the change of functionality of all domains (data-types) involving a STA or ANS. For example: W=[S→T] changes to W′=T. In effect the state 'disappears' or is replaced by COD, like in: C=[S→S] changed to C′=COD. This can only by done under the pre-condition which rules out semantics involving copies of the state. Note that such changes are necessary to allow the transformation process to continue discovering code generation actions. For example, what is the functionality of the sub-expressions Read() and Write(o), defined in section 3.2.2? We can but not choose to say that Read:[S ≻ [I x S]] and Write:[O ≻ S ≻ S] any longer. To be consistent we have to associate them with the new domains (data types): Read:[[] ≻ [I x COD]] and Write:[O ≻ COD], which are redefining <u>Read</u> as the code generation procedure with no parameters which should plant code to read a value, and <u>Write</u> as another one to plant code to write one. In fact <u>Read</u> is not in that domain, our system will redefine it as Read:I, and the Destination Analysis (to be described below) will transform Read() to Read(reg) with Read:[REG ≻ COD] indicating that <u>Read</u> is a procedure which generates code to read a value into the destination indicated by its parameter.

In Snapshot 3.5 we show the result of applying to the structured version of our current example language all transformations corresponding to the Normalisation and State Analysis. Note that the only differences between the unstructured Snapshot 3.3 and the structured Snapshot 3.5 are the absence of the function <u>Load</u> in the former and a number of extra λ-abstractions in the latter. We could define a transformation to perform beta-reduction; in this case it is possible. But in some other cases we can not do such a reduction. In the Destination Analysis below, we will explain why we can not perform beta-reduction, and we also comment on the significance of the differences.

Snapshot 3.5: Flow Diagrams State-Structured. State Analysis

```
let C node be switchon type^node into                      by R1.1   (3.5.1)
{ case [Dummy]:
    {}; endcase
                                                    by R1.1, R2.8    (3.5.2)
  case [c1;c2]:
    C[c1]; C[c2]; endcase
                                                    by R1.1, R2.9    (3.5.3)
  case [If e Then c1 Else c2]:
    (λt.t→C[c1],C[c2])(E[e]); endcase
                                                    by R1.1, R2.10   (3.5.4)
  case [While e Do c1]:
    Fix(λc.(λt.t→{ C[c1]; c },{})(E[e])); endcase
                                     by R1.1, R2.8, R2.9, R2.10      (3.5.5)
  case [c1 Repeatwhile e]:
    Fix(λc.(λt.t→c,{}){ C[c1]; E[e] }); endcase
                                     by R1.1, R2.8, R2.9, R2.10      (3.5.6)
}

let E node be switchon type^node into                      by R1.1   (3.5.7)
{ case [True]:
    Load TRUE; endcase
                                                           by R1.1   (3.5.8)
  case [False]:
    Load FALSE; endcase
                                                           by R1.1   (3.5.9)
  case [If e1 Then e2 Else e3]:
    (λt.t→E[e2],E[e3])(E[e1]); endcase              by R1.1, R2.10   (3.5.10)
}
```

3.3 Syntactic Transformations

3.3.1 Proceduring

Firstly, the functions are curried, but procedures are parametric. According to the Oxford school's tradition, functions are defined as curried as possible. There is no reason why they could not be defined by equivalent un-curried functions. In fact, the DS of the programming language [ADA80] is, from the start in this form.

Why do we have to un-curry? If we wish to give an algorithmic interpretation to the concrete representation of semantic functions, then it seems natural to use a form, which is in line with those defined in our target programming language. For this reason we introduce the following conversions:

$$\text{let } vi_1 \ldots i_n \text{ be } C \quad => \quad \text{let } v(i_1, \ldots, i_n) \text{ be } C \qquad [R3.1]$$

$$e_0 e_1 \ldots e_n \quad => \quad e_0(e_1, \ldots, e_n) \qquad [R3.2]$$

3.3.2 **Applied Occurrence of Abstractions**

Secondly, the equivalence between the application of lambda abstractions and BCPL's **let** declarations, is formalised as follows:

$$(\lambda i.e)(e_1) \quad => \quad \{ \text{ let } i{=}e_1;\ e \ \} \qquad [R3.3]$$

$$(\lambda i.e)(e_1)(e_2) \quad => \quad \{ \text{ let } i{=}e_1;\ e(e_2) \ \} \qquad [R3.4]$$

$$\text{when not } i{:}COD\ (\lambda i.e)\{C;\ e_1\} \quad => \quad C;\ \{ \text{ let } i{=}e_1;\ e \ \} \qquad [R3.5]$$

Note that if the condition of R3.5 fails, then either R3.3 or R3.4 is applied. The following shows the syntactic transformations for the while-loop as left by the State Analysis:

State Elimination:

$Fix(\lambda c.E[e]\rightarrow\{\ C[c_1];\ c\ \},\{\})$	From (3.3.5)
$Fix(\lambda c.E([e])\rightarrow\{\ C([c_1]);\ c\ \},\{\})$	By R3.2/3 times

Structuring the State:

$Fix(\lambda c.(\lambda t.t\rightarrow\{\ C[c_1];\ c\ \},\{\})(E[e]))$	From (3.5.5)
$Fix(\lambda c.(\lambda t.t\rightarrow\{\ C([c_1]);\ c\ \},\{\})(E([e])))$	By R3.2/3 times
$Fix(\lambda c.\{\text{ let } t{=}E([e])\ ;\ t\rightarrow\{\ C([c_1])\ ;\ c\ \},\{\}\})$	By R3.3-(3.6.5)

Note that R3.2 is applied regardless of the brackets which are printed because of precedence reasons. So a $e_0 e_1$ construct of WFF_s like Fix(e), is transformed by R3.2 to the same Fix(e), but now a E(P)A of WFF_t. Also note that ';' has lower precedence that anything else.

Before embarking on the semantic transformations, let us observe that the result of transformations R3.1 to R3.5, as described in Snapshot 3.6, is still a Standard Denotational Specification, written in an un-curried form with (more or less) the original domains. Alternatively, it can be regarded

Snapshot 3.6: Flow Diagrams State-Structured. Syntactic Transformations

```
let C(node) be switchon type^node into                        by R3.1     (3.6.1)
{ case [Dummy]:                                                        no change

  case [c1;c2]:
    C([c1]); C([c2]); endcase                                by R3.2/twice    (3.6.3)

  case [If e Then c1 Else c2]:
    { let t = E([e]); t→C([c1]),C([c2]) }; endcase
                                                      by R3.2/3 times, R3.3   (3.6.4)

  case [While e Do c1]:
    Fix(λc.{ let t = E([e]); t→C([c1]); c,{} }); endcase
                                                      by R3.2/3 times, R3.3   (3.6.5)

  case [c1 Repeatwhile e]:
    Fix(λc.{0 C([c1]); { let t = E([e]); t→c,{} }0); endcase
                                                      by R3.2/3 times, R3.5   (3.6.6)

}

let E(node) be switchon type^node into                        by R3.1     (3.6.7)
{ case [True]:
    Load(TRUE); endcase                                       by R3.2     (3.6.8)

  case [False]:
    Load(FALSE); endcase                                      by R3.2     (3.6.9)

  case [If e1 Then e2 Else e3]:
    { let t = E([e1]); t→E([e2]),E([e3]) }; endcase
                                                      by R3.2/3 times, R3.3   (3.6.10)
}
```

as something akin to store semantics [MaS76]. We could have started with a version that looked similar to this one, like P.Mosses's DSL [Mos79], and then derived the store semantics applying the mechanism developed by [MaS76] as described in [Sto77]. To avoid the need of presenting yet another mathematical metalanguage, we preferred not to do so, starting with the version in [Sto77].

Recently, R. Sethi introduced PLUMB programs [Set82], which are also something akin to store semantics. In PLUMB the state is never explicitly written in any semantic equation, this is done by means of a mechanism called a 'pipe', which extends function composition. R.Sethi has shown that 'pipes' are suitable to express the control flow aspects of sequential languages. In this sense, if one starts directly with a store semantics then our State Analysis is not required.

The State Analysis has been applied prior to the Syntactic Transformations only for convenience, because it is easier to eliminate the application of a curried function than to eliminate the parameters of an un-curried procedure. In this respect, the State Analysis is part of the semantic transformations which we describe now.

3.4 Semantic Transformations

What can we say now about this specification (Snapshot 3.6)? It looks very much like an interpreter. In fact we can regard it as an definitional interpreter (in the sense of [Rey72]) written in a metalanguage which looks as much like a programming language as a mathematical system of equations. If we are able to discover, from the specification, something more concrete about the way that semantic objects are handled, then we will be able to say something about how we can implement a compiler for the language in question. The difference between such an interpreter and a compiler, is that the latter is characterised by a process of translation, a generation of an intermediate or final representation, i.e: the target code. It actually does not matter, whether this code is later, either run on the hardware of a particular machine, or if it is interpreted by software. The CGP is the primary object of our analysis. So, what are the characteristics of this code?

3.4.1 Destination Analysis

Let us consider the form of the specification that manages intermediate values, obtained while evaluating sub-expressions. The interpreter seems to hold these values in variables, but the CGP will not see the values, it is only at run time when values will be produced, and handled by the code. The compiler's activity is to define where at run time, these values will be kept. Hence, here is our first semantic transformation:

```
                  let v(D) be C  | =>  | let v(D).dest.(reg) In COD      [R5.1]
when v(D):REG                   _|     |_be C
```

REG is a domain of interest. In the example language that we are considering in this section REG=W and W=[S>T]. After the State Analysis W'=T so that now REG=W'=T. The transformation above indicates that every function, producing a registered value (in REG) as a result, will now have an indication for the destination of the resultant value. The variable reg is used to keep a description of the destination but there is no indication of what this destination is. It might be either a register descriptor or a level+offset, describing a position in an activation record. Note that expressions that were in [D>REG] (any D), now are in [[D x REG] > COD].

We also need the counterpart to R5.1:

```
                  let v(D) be C    =>    let v(D) be [first.reg/reg]C    [R5.2]
when not v(D):REG
```

Where first.reg is either a free constant or a free variable, containing a description of a place for run time temporary values (respectively the first available register or the start of the corresponding activation record's workspace).

Function calls are converted in two ways depending on the context in which they appear. If the result of a function is immediately bound to a variable then we make a register name out of the variable name. Otherwise, the destination is the same variable which was defined in R5.1:

```
when e(P):REG            e(P)     =>    e(P).dest.(reg) In COD              [R5.3]

           { let i=e(P); C }  =>  { e(P).dest.(i) In COD; C }             [R5.4]
when e(P):REG                        rename i=>(i=a_k)→reg+k, reg
```

Note that in R5.4, the destination reg+k is made out of the decoration 'k' of the identifiers name. The effect of this is that the register allocation depends on the decoration of names (with digits) of the original specification. It might be argued that this is not desirable; that the transformation rules should find out which registers must be allocated, instead of forcing such issues to depend on the textual form of the semantic specification. However, this method allows the control of register allocation at the level of the semantic specification. We have opted for this general approach, allowing 'user control' of register allocation. This is why, our semantic equations contain some explicit abstractions which seem not to be necessary. In the introductory example (section 2.4), the equation for an application in the Lambda Calculus (2.1.4) is expressed as:

$E[e_1e_2]p=(\lambda ee'.(e|F)e')(E[e_1]p)(E[e_2]p)$.

Instead of the equivalent form: $E[e_1e_2]p=((E[e_1]p)|F)(E[e_2]p)$.

Because of this 'user controlled' register allocation technique, we can not apply beta-reduction (see section 3.2.3 above). Another reason for not applying beta-reduction, is that the applied occurrence of abstractions, transformed as **let** declarations, helps to avoid re-evaluation. In general, the argument to an applied occurrence of an abstraction will be an expression with corresponding code generation process. We do not wish to substitute such an expression since this might result in a duplication of the same code generation text, or worst even, a duplication of the same generated code. The reader must remember that we are generating the text of a code generator expressed as procedures written in a programming language, we are not 'implementing' the lambda-calculus by its reduction rules. The only reason for beta-reduction would be to reduce the length of expressions; but this reduction must not be to the detriment of the target CGP.

Both these issues, variable naming to allow register allocation and explicit abstractions, are implementation issues which have been pushed up to the level of a Standard Denotational Specification. We believe that further analysis, could show that these two features could be automated, and hence not necessarily explicitly expressed at that level, and that the relevant information could be extracted from a truly Standard Specification without writing semantic equations with this sort of implementation detail. However in Chapter 7, we explore the possibility of abstracting more important issues at the level of a Semantic Specification, through a technique which we call Implementation Denotational Semantics.

Let us return now to the Destination Analysis. If a function call involves other calls amongst its parameters which produce values (in REG), then as a result of R5.3 or R5.4, these values are indicated by a .dest.(E) construct. In this case we make a sequence of statements, replacing the old parameter by its destination register.

$$\left\lceil e_2(P_0,\ e_1,\ P_1) \right\rfloor \Rightarrow \left\lfloor \begin{array}{l} e_1 \\ e_2(P_0,\ E,\ P_1) \end{array} \right. \quad \text{when } e_1 = e(P)A.\mathrm{dest}.(E) \qquad [R5.5]$$

By symmetry we also define:

$$\left.\begin{array}{r} e(P,\ i) \\ \text{when } i\text{:REG and } P \text{ not null} \end{array}\right| \Rightarrow \left| \begin{array}{l} e(P)\mathbf{.dest.}(i)\ \mathbf{In}\ \text{COD} \\ \ \end{array}\right. \qquad \text{[R5.6]}$$

Next, if we consider the language with the state written explicitly, as shown in Snapshot 3.3, we need to load variables that appear either as statements like in (3.3.8) and (3.3.9) or in a conditional's branch (not in this example):

$$\left.\begin{array}{r} \{C_0;\ i;\ C_1\} \\ \text{or} \\ e_0 \rightarrow i,\ e_2 \\ \text{or} \\ e_0 \rightarrow e_1,\ i \\ \text{when } i\text{:REG and not } i\text{:COD} \end{array}\right| \begin{array}{c} \Rightarrow \\ \\ \Rightarrow \\ \\ \Rightarrow \\ \ \end{array} \left| \begin{array}{l} \{C_0;\ C;\ C_1\} \\ \text{or} \\ e_0 \rightarrow C,\ e_2 \\ \text{or} \\ e_0 \rightarrow e_1,\ C \\ \text{where } C = \text{trans.load}(\text{DOM}(i),\ i)\ \mathbf{In}\ \text{REG} \end{array}\right. \qquad \text{[R5.7]}$$

$$\begin{array}{rcl} \text{DOM} & : & \text{Exp} \rightarrow \text{EXP} \\ \text{DOM}(e\text{:}d) & = & E\text{:EXP} \end{array} \qquad \text{[D6]}$$

$$\begin{array}{ll} \text{where} & E = d \text{ if not } d=[D_1 + \ldots + D_n] \text{ for any } D_i \text{ and not } e\text{:LOC} \\ & E = \text{domain.of}(e) \text{ otherwise} \end{array}$$

The primitive operation <u>trans.load</u>, will be used to indicate that a particular value must be loaded into its destination. The reason for the domain definition indicated by **In**, is to allow R5.3 or R5.4 to supply a destination. For type checking purposes <u>trans.load</u> associates a type with the destination. This type must be given as the first parameter. If the type associated with the load is known at the moment of generation of the CGP then the function <u>DOM</u> returns an identifier, otherwise it returns the expression <u>domain.of(e)</u> so that the CGP would be able to determine the particular instance of a type while generating code. Types are unknown when they belong to an union domain or when the operation is to load the contents of a location. Section 4.3.3 explores other type checking issues.

The actual code planted to load a value is not defined, it is up to the interpretation of trans.load to define the precise code. For example, in this simple example language, the only values that might be loaded are the constants TRUE and FALSE. Using DEC-10 instructions and associating to these constants respectively the values minus one and zero, trans.load(D..T, TRUE).dest.(reg) might plant:

```
                       interpreting reg as
    a fast register (reg=AC)          an invocation record word (reg=#off)
    SETO    AC,0                      SETOM    0,#off(BAS)
```

and trans.load(D..T, FALSE).dest.(reg) will plant similar code using the instruction SETZ.

Order of Application: In R5.2, the expression [first.reg/reg]C does not make sense unless we wait until all regs have been introduced by other transformation rules. We solve this by applying the substitution once all rules have been applied. Alternatively a substitution can be interpreted as a call by need, i.e: whenever a rule inserts a reg within C, it is immediately substituted by first.reg.

The order of application of rules is not determined. So the transformation process can be understood as a non-deterministic process. However, in a particular implementation, one can apply a specific order. For example, we have opted for a syntax directed transformation, where the order of analysis is directed by the structure of the abstract syntax in a left to right manner.

Applying the Destination Analysis rules results in Snapshot 3.7.

Snapshot 3.7: Flow Diagrams State-Structured. Destination Analysis

```
let C(node) be switchon type^node into                              no change
{ case [Dummy]:                                                     no change
  case [c1;c2]:                                                     no change

  case [If e Then c1 Else c2]:
    E([e]).dest.(first.reg); first.reg>C([c1]),C([c2]); endcase
                                                       by R5.2, R5.4   (3.7.4)

  case [While e Do c1]:
    Fix(λc.{ E([e]).dest.(first.reg); first.reg>C([c1]); c,{} }); endcase
                                                       by R5.2, R5.4   (3.7.5)

  case [c1 Repeatwhile e]:
    Fix(λc.{ C([c1]); E([e]).dest.(first.reg); first.reg>c,{} }); endcase
                                                       by R5.2, R5.4   (3.7.6)
}

let E(node).dest.(reg) be switchon type^node into              by R5.1   (3.7.7)
{ case [True]:
    Load(TRUE).dest.(reg); endcase                              by R5.3   (3.7.8)

  case [False]:
    Load(FALSE).dest.(reg); endcase                             by R5.3   (3.7.9)

  case [If e1 Then e2 Else e3]:
    E([e1]).dest.(reg); reg>E([e2]).dest.(reg),E([e3]).dest.(reg); endcase
                                                   by R5.3/twice, R5.4  (3.7.10)
}
```

3.4.2 **Continuation Analysis**

In this example language, there are no continuations, nevertheless in terms of code generation, there are certain parts where jumps, to and from different parts of the code will be produced. We will analyse the following areas: variables as statements, conditionals, and the minimal fix point finder.

Variables as statements: A variable denoting a code function might stand in a block as a statement. This happened for example in the equation for a while-loop (3.5.5), where R2.9 transformed the composition operator o into a sequence of statements. Also, a variable denoting a code function might stand in a conditional's branch, like in (3.7.6). These code variables are interpreted as a need to jump:

$$\begin{array}{lcccl} & \{C_0;\ i;\ C_1\} & \Rightarrow & \{C_0;\ C;\ C_1\} & \\ & \text{or} & & \text{or} & \\ \textbf{when } i\text{:COD} & e_0 \rightarrow i,\ e_2 & \Rightarrow & e_0 \rightarrow C,\ e_2 & [R6.1] \\ & \text{or} & & \text{or} & \\ & e_0 \rightarrow e_1,\ i & \Rightarrow & e_0 \rightarrow e_1,\ C & \\ & & & \textbf{where } C = \text{trans.jump.to}(i) & \end{array}$$

Conditionals: We wish to evaluate the boolean part in a particular destination (if it is not already a destination variable), and then check the result, planting appropriate instructions to select the corresponding path.

$$\begin{array}{lcl} & & \{\ \textbf{let } \text{econd.code=forward(COD)} \\ & & \quad \textbf{let } \text{fcond.code=forward(COD)} \\ & & \quad C \\ & & \quad \text{trans.jump.if.false(i, fcond.code)} \\ e_0 \rightarrow e_1, e_2 & \Rightarrow & \quad e_1 \\ & & \quad \text{trans.jump.to(econd.code)} \\ \textbf{where } e_0 = e(P)\textbf{.dest.}(i)A & & \quad \text{fix.here(fcond.code)} \\ \quad \text{or } e_0 = i & & \quad e_2 \\ \textbf{when } i\text{:REG} & & \quad \text{fix.here(econd.code)} \\ & & \}\ \textbf{where } C = (e_0 = i) \rightarrow \text{null},\ e_0 \end{array}$$

What is the meaning of forward and fix.here? Every time a forward reference is made a CGP will have to take some actions. Different techniques are possible, but at this stage, one does not wish to be committed to any particular one. The primitives forward and fix.here, like all primitive procedures and function introduced by the transformation process, can have different interpretations, one can choose any of the well known techniques to achieve the desired effect. For example: if one wishes to use a chaining mechanism, these operations can be interpreted respectively as new.chain and fix.chain. Alternatively, if one wishes to rely on the activity of a loader, they can be interpreted as new.label and trans.label. The parameter COD to forward is supplied for type checking (compile or run-time) purposes.

This transformation rule is the most general way of transforming the conditional, but by looking at its particular form, one can optimise the CGP. For example, in (3.7.5) and (3.7.6), the false part is a null block {}, in this case some of the right hand side statements are unnecessary. Also, in (3.7.6) the true part was a simple variable and a continuation, and as a result of R6.1 it would now be an unconditional jump trans.jump.to. Instead of jumping to fcond.code when the boolean part evaluates to false, we can reverse the test-and-jump replacing the original trans.jump.if.false by trans.jump.if.true. When the false part is an unconditional jump, we can replace the fcond.code of trans.jump.if.false by the parameter of the unconditional. Finally, if the true branch consists of an expression involving continuations (like the examples of Chapter 5), then there is no need for the forward-fix and jump to econd.code constructions. These, and other similar observations, are formalised as follows:

$e_0 \rightarrow e_1, e_2 \quad \Rightarrow \quad C$ [R6.2]

when (E0IsDes or E0IsIde) and i:REG
where $C = \{ C_1; C_2; C_3; C_4; C_5; C_6; C_7; C_8; C_9 \}$
C_1 = NoEndCo → null, **let** econd.code = forward(COD)
C_2 = NoFalse → null, **let** fcond.code = forward(COD)
C_3 = E0IsIde → null, e_0
C_4 = JumpRut(i, FalseCo)
C_5 = Reverse → null, e_1
C_6 = NoEndCo → null, trans.jump.to(econd.code)
C_7 = NoFalse → null, fix.here(fcond.code)
C_8 = E2IsJmp → null, e_2
C_9 = NoEndCo → null, fix.here(econd.code)
E0IsDes = e_0=e(P).**dest.**(i)A
E0IsIde = e_0=i
E1IsJmp = e_1=trans.jump.to(i_1)
E2IsJmp = e_2=trans.jump.to(i_2)
E2IsNul = e_2={}
Reverse = E2IsNul and E1IsJmp
NoFalse = Reverse or E2IsJmp
NoEndCo = E2IsNul or E2IsJmp or WillJump(e_1)
JumpRut = Reverse → trans.jump.if.true, trans.jump.if.false
FalseCo = Reverse → i_1, E2IsJmp → i_2, fcond.code
HasCont(e)=TRUE if e contains continuations which will jump
HasCont(e)=FALSE otherwise

There is still more room for improvement. For example we have not considered what happens when the true part is a null block {}, but this case never happens in our examples so we will not pursue this further. Other improvements relate to those forms of tests which can be translated with skip instructions. We will wait until the examples require this feature.

Fix: The paradoxical combinator Y was introduced by J. Curry [Cur58] and was discussed in relation to an operational semantics by P. Landin [Lan65]. In relation to a DS, C. Strachey, in an early paper pointed out:

> "...it does the same job as LABEL in LISP. Thus (LABEL,c,(LAMBDA, (X), (S))) is equivalent to Fix(λc.λx.S).
> [Str66]-p213

The denotation of Y is the function $\mathrm{Fix}=\lambda F.\bigsqcup_n F^n(\mathbf{Bot})$, <u>Fix</u>, a predefined WFF_s identifier, it is used both in (3.7.5) and (3.7.6). It is a generic function whose interpretation depends on its functionality. In this case it is used in expressions of the form: $\mathrm{Fix}(\lambda c.c_1)$, hence its functionality belongs to [COD$\to$COD]. $\mathrm{Fix}(\lambda c.c_1)$ denotes the COD function denoted by c_1, with c bound to the same. In implementation terms this means that $\mathrm{Fix}(\lambda c.c_1)$ is denoting the instruction sequence forming the code of c_1, with c bound to the same. The following transformation will implement this observation:

```
when i:COD          Fix(λi.e)‾‾| => |‾‾{ let i = here(COD); e }        [R6.3]
                           __|    |__rename i=>restart.code
```

In a similar fashion to <u>forward</u> and <u>fix.here</u>, the interpretation of <u>here</u> is left open to implementation choice. If one wishes to use either a chaining mechanism, or rely on the activity of a loader, then it can be interpreted respectively as <u>this.program.counter</u> or <u>trans.label</u>

Applying these transformations we obtain Snapshot 3.8.

Snapshot 3.8: Flow Diagrams State-Structured. Continuation Analysis

```
let C(node) be switchon type^node into                                  no change
{ case [Dummy]:                                                          no change
  case [c1;c2]:                                                          no change

  case [If e Then c1 Else c2]:
    E([e]).dest.(first.reg)
    { let econd.code = forward(COD)
      let fcond.code = forward(COD)
      trans.jump.if.false(first.reg, fcond.code)
      C([c1])
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      C([c2])
      fix.here(econd.code)
    }; endcase                                                  by R6.2   (3.8.4)

  case [While e Do c1]:
    {0 let restart.code = here(COD)
       E([e]).dest.(first.reg)
       { let fcond.code = forward(COD)
         trans.jump.if.false(first.reg, fcond.code)
         C([c1])
         trans.jump.to(restart.code)
         fix.here(fcond.code)
    }0; endcase                                    by R6.1, R6.2, R6.3   (3.8.5)

  case [c1 Repeatwhile e]:
    { let restart.code = here(COD)
      C([c1])
      E([e]).dest.(first.reg)
      trans.jump.if.true(first.reg, restart.code)
    }; endcase                                     by R6.1, R6.2, R6.3   (3.8.6)
}

let E(node).dest.(reg) be switchon type^node into                        no change
{ case [True]:                                                           no change
  case [False]:                                                          no change

  case [If e1 Then e2 Else e3]:
    E([e1]).dest.(reg)
    { let econd.code = forward(COD)
      let fcond.code = forward(COD)
      trans.jump.if.false(reg, fcond.code)
      E([e2]).dest.(reg)
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      E([e3]).dest.(reg)
      fix.here(econd.code)
    }; endcase                                                  by R6.2   (3.8.10)
}
```

Snapshot 3.9: Flow Diagrams with Side Effects. Original Specification

Modifications to Snapshot 3.4

Semantic Domain

w:W=[S → [T x S]]. expression values

Semantic Primitive

Load:[T → W].
Load=
λt.Strict(λs.<t,s>).

Semantic Equations

$$C[\text{If e Then } c_1 \text{ Else } c_2]= E[e] \underline{*} \lambda t.t \rightarrow C[c_1],C[c_2]. \quad (3.9.1)$$

$$C[\text{While e Do } c_1]= Fix\{\lambda c.\{E[e] \underline{*} \lambda t.t \rightarrow C[c_1] \underline{o} c,Is\}\}. \quad (3.9.2)$$

$$C[c_1 \text{ Repeatwhile e}]= Fix\{\lambda c.\{C[c_1] \underline{o} E[e] \underline{*} \lambda t.t \rightarrow c,Is\}\}. \quad (3.9.3)$$

$$E[\text{If } e_1 \text{ Then } e_2 \text{ Else } e_3]= E[e_1] \underline{*} \lambda t.t \rightarrow E[e_2],E[e_3]. \quad (3.9.4)$$

3.5 Side effects

Consider now a similar language with side effects. It is based on [Sto77], table 9.3. In Snapshot 3.9, we reproduce only those parts that differ from the specification of Snapshot 3.4. Note that the redefinition of W, results in every $\underline{+}$ being replaced by $\underline{*}$. Except for the new equation for Load, these replacements are the only modifications to the original semantic equations, confirming once more that structuring the state with appropriate operators leads to structured equations. We have to analyse this new operator at the level of the State Analysis.

3.5.1 Reversed Star

The $\underline{*}$ operator, is the reverse of the star operator used by C. Strachey [Str73] to abstract the meaning of a conditional:

$$
\begin{array}{lcl}
\underline{.*.} & : & [[D_1 \rightarrow [DxD_2]]x[D \rightarrow [D_2 \rightarrow D_3]] \rightarrow [D_1 \rightarrow D_3]] \\
 & & \text{for any } D_1,\ D_2 \text{ and } D_3.\ \text{But not } D \underline{C} STA \\
f & : & [D_1 \rightarrow [DxD_2]] \\
g & : & [D \rightarrow [D_2 \rightarrow D_3]] \\
d & : & D \\
d_1 & : & D_1 \\
d_2 & : & D_2 \\
(f \underline{*} g)d_1 & = & gdd_2 \text{ where } \langle d,d_2 \rangle = fd_1 \qquad [D7]
\end{array}
$$

D may not be a state, to avoid any state saving. As for the composition operator, we prefer the reversed form that allows us to read equation from left to right. This operator is used for expressions with side effects. Expressions produce, in general, a value and the side affect is a modification of the state. As already explained in section 3.2.1, we are only allowing one copy of the state at any given time. This means that in $e\underline{*}e'$, $\underline{*}$ is carrying information which is not relevant for the process of code generation. It is the code generated by e and e′ that will carry out this activity, and it will follow the same sequence of actions specified by the appropriate transformation of e and e′.

As usual, we formalise a transformation with a conversion rule:

$$e_0 \underset{D_1}{\underline{*}} e_1 \quad \Rightarrow \quad (e_0 \text{ In } [D_1 \rightarrow D])\underline{o}(e_1 \text{ In } [D \rightarrow D_3]) \qquad [R2.11]$$

when for any domains D, D_1 and D_3 $e_0:[D_1 \rightarrow [DxSTA]]$ and $e_1:[D \rightarrow [STA \rightarrow D_3]]$

$\underline{*}$ is a generic operator, in the particular instance of $.\underline{*}.:[[D_1 \rightarrow [DxSTA]]x[D \rightarrow [STA \rightarrow D_3]] \rightarrow [D_1 \rightarrow D_3]]$ it links two blocks of code. The first block produces a side effect and a value which is passed to the second block. In most cases D_1=STA and D_3=STA or D_3=ANS, in which case it will be trapped by R2.9. The rest of the transformations are the same as those for the same language without side effects, and the final version does not differ from the corresponding version for that language. This is to be expected since, so far, the language has no explicit expressions with side effects. For example, this is how we transform the while loop:

While loop with side effects:	
$Fix(\lambda c.(E[e] \; \underline{*} \; \lambda t.t \rightarrow C[c_1] \; \underline{o} \; c, Is))$	From (3.9.2)
$Fix(\lambda c.(E[e] \; \underline{*} \; \lambda t.t \rightarrow C[c_1] \; \underline{o} \; c, \{\}))$	By R2.8
$Fix(\lambda c.(E[e] \; \underline{*} \; \lambda t.t \rightarrow \{ \; C[c_1]; \; c \; \}, \{\}))$	By R2.9
$Fix(\lambda c.(E[e] \; \underline{o} \; \lambda t.t \rightarrow \{ \; C[c_1]; \; c \; \}, \{\}))$	By R2.11
$Fix(\lambda c.(\lambda t.t \rightarrow \{ \; C[c_1]; \; c \; \}, \{\})(E[e]))$	By R2.9

3.6 The Store

In order to analyse the relation between semantic equations producing side effects and the corresponding procedures to generate code, we now define the store as a function from identifiers to values. We add identifiers to the syntactic categories of expressions, and assignments for generality both in commands and expressions. Locations are introduced in chapter 7 where the example language with environments provides the appropriate block structure. Firstly, we consider the semantic equations of Snapshot 3.10 where the state is explicitly used. Secondly, we will rewrite them by structuring the state. We require new transformations only for the former and at the level of the State Analysis.

3.6.1 Updating

A modification of the state occurs in both equations for assignment. The store is modified in such a way that after the assignment, identifiers denote the result of the right hand side evaluation. In the semantic equations this indicated by the [/] construction, the code generator requires a procedure <u>trans.update</u> which will generate a move to memory:

when i:STA $\quad i[e_1/e_2] \quad => \quad trans.update(e_1, e_2) \quad$ [R2.12]

Snapshot 3.10: The Store State-Unstructured. Original Specification

Extensions to Snapshot 3.9

Syntax

```
i:Ide.                                                      identifiers
c ::= i:=e | ...
e ::= i | i:=e1 | ...
```

Semantics

$s:S=[Ide \to T]$. machine states

$C:[Com \to C]$. (3.10.1)

$C[i:=e]=$
$E[e] \underline{*} \lambda ts.s[t/[i]]$. (3.10.2)

$E:[Exp \to W]$. (3.10.3)

$E[i]=$
$Strict(\lambda s.\langle s[i],s\rangle)$. (3.10.4)

$E[i:=e_1]=$
$E[e_1] \underline{*} \lambda ts.\langle t,s[t/[i]]\rangle$. (3.10.5)

3.6.2 **Loading**

Loading a value from the store is indicated by an application of the state. This requires again the primitive procedure trans.load:

when i:STA i(e) => trans.load(DOM(e), e) In REG [R2.13]

3.6.3 **Tuples**

In all our examples, tuples have length of two. To simplify the analysis, we will consider only tuples of this length. In this unstructured version, the state is explicitly used in tuples, both as a single variable in (3.10.4) and as an expression in (3.10.5). In the former case, we simple eliminate such a variable, since, like the identity function of the state, it does not convey any code generation information. In the latter, since only one copy of the state is allowed, side effects can not occur in both tuple expressions, so we can impose a particular (left to right) order of evaluation, transforming the tuple to a sequence of two statements:

when i:STA $\langle e_0, i\rangle$ => e_0 [R2.14]

Snapshot 3.11: The Store State-Unstructured. Destination Analysis

```
let C(node) be switchon type^node into                by R1.1, R3.1  (3.11.1)
{ case [i:=e]:
    E([e]).dest.(first.reg); trans.update([i], first.reg); endcase
    by R1.1, R1.2, R2.2, R2.9, R2.11, R2.12, R3.2/twice, R3.3, R5.2, R5.4
    R5.6                                                          (3.11.2)

  case ...
}

let E(node).dest.(reg) be switchon type^node into
                                                by R1.1, R3.1, R5.1  (3.11.3)
{ case [i]:
    trans.load(Ide, [i]).dest.(reg); endcase
                            by R1.1, R2.2, R2.7, R2.13, R2.14, R5.3  (3.11.4)

  case [i:=e1]:
    E([e1]).dest.(reg)
    trans.load(domain.of(reg), reg).dest.(reg)
    trans.update([i], reg); endcase
    by R1.1, R1.2, R2.2, R2.9, R2.11, R2.12, R2.15, R3.2/twice, R3.3, R5.3
    R5.4, R5.6, R5.7                                              (3.11.5)

  case ...
}
```

when e_1:STA $\langle e_0, e_1 \rangle$ => { e_0 ; e_1 } [R2.15]

Applying these transformations and all those required to bring it to the level of the Destination Analysis we obtain Snapshot 3.11.

3.7 Structuring

If we now express the semantic equations without explicit use of the state, rewriting the equations from Snapshot 3.10 to those in Snapshot 3.12, we find that this time, there is no need to define any new transformation rule. The problem is that, now, one has to supply the code for the primitives Update and Conts, which can easily be done with the equivalent procedures trans.update and trans.load supplied by our system. The corresponding generation, also at the level of the Destination Analysis, is quite similar to the one of the unstructured version, as can be seen in Snapshot 3.13.

Snapshot 3.12: The Store State-Structured. Original Specification

Modifications to Snapshot 3.10

Semantic Primitives

```
Update:[Ide → T → C].
Update[i]ts=
  s[t/[i]].

Conts:[Ide → W].
Conts[i]s=
  <s[i],s>.
```

Semantic Equations

```
C:[Com → C].                                                 (3.12.1)

C[i:=e]=
  E[e] * Update[i].                                          (3.12.2)

E:[Exp → W].                                                 (3.12.3)

E[i]=
  Conts[i].                                                  (3.12.4)

E[i:=e1]=
  E[e1] * λt.(Update[i]t o Load t).                          (3.12.5)
```

Snapshot 3.13: The Store State-Structured. Destination Analysis

```
let C(node) be switchon type^node into               by R1.1, R3.1  (3.13.1)
{ case [i:=e]:
    E([e]).dest.(first.reg); Update([i]).dest.(first.reg); endcase
              by R1.1, R2.9, R2.11, R3.2/twice, R5.2, R5.3, R5.5  (3.13.2)

  case ...
}

let E(node).dest.(reg) be switchon type^node into
                                                by R1.1, R3.1, R5.1  (3.13.3)
{ case [i]:
    Conts([i]).dest.(reg); endcase               by R1.1, R3.2, R5.3  (3.13.4)

  case [i:=e1]:
    E([e1]).dest.(reg); Update([i]).dest.(reg); Load(reg).dest.(reg)
    endcase
    by R1.1, R2.9/twice, R2.11, R3.2/3 times, R3.3, R5.3, R5.4, R5.6
                                                                     (3.13.5)

  case ...
}
```

3.8 BCPL

In order to test and run our code generation phase, we also have to generate a lexical analyser and parser. For these, we use two systems which also generate BCPL procedures. For the former, we use LEXGEN [Suf78a], and for the latter LL1 [Suf78b]. This syntactic phase builds up an internal representation of the source programs in the form of a tree. From this automatic generation, and with the help of a text editor FORM [Suf77], we provide an interface to the code generation phase which defines the names of each tree node. With this automatically generated interface, we associate a 'tag' or 'type' to every [s] of a command of the form **case** [s]:C, and a 'selector' for every sub-expression in C of the same command.

Syntactic Alternative	Tag	Selectors p1	p2	p3
Dummy	T..Dummy			
If e **Then** c_1 **Else** c_2	N3..ConditionalCom	e	c_1	c_2
c_1;c_2	N2..Sequence	c_1	c_2	
While e **Do** c_1	N2..While	e	c_1	
c_1 **Repeatwhile** e	N2..RepeatWhile	c_1	e	
i:=e	N2..Assignment	i	e	
True	T..True			
False	T..False			
If e_1 **Then** e_2 **Else** e_3	N3..ConditionalExp	e_1	e_2	e_3
i	T..Ident			
i:=e'	N2..AssignmentExp	i	e'	

So that:

Every [s] => replaced by its appropriate 'tag' or 'selector' [RA.1]

Every 'curly' valuator v and every domain d => respectively replaced by trans.v and D..d [RA.2]

With this, we have completed all transformations. In Snapshot 3.14 we show the final version of the unstructured version with side effects and locations. It can be compiled successfully in BCPL and if provided with a syntax analyser and machine interface, it is a CGP for this example language. The only differences between the final version of the unstructured version, against the structured one, are the primitives procedures. In the former, the transformation process has inserted calls to the primitive procedures <u>trans.load</u> and <u>trans.update</u>. In the structured version, the primitive functions of the original semantic specification <u>Load</u>, <u>Conts</u> and <u>Update</u> have been carried over to the final CGP, transformed to BCPL procedure calls. Both <u>Load</u> and <u>Conts</u> correspond to <u>trans.load</u> since in our 'machine-configuration' loading and looking up memory are equivalent operations. The equivalence between the two versions is then obvious, we have to choose one to show the final version, we select the unstructured version, to avoid having to include in the machine interface those three primitives. All procedures inserted by the transformation process are supplied by our system. but primitive functions of the original semantic specification, corresponding to procedures in the final CGP, have to be supplied, in the machine interface, by the user.

Snapshot 3.14: Flow Diagrams State-Unstructured. BCPL

```
let trans.C(node) be switchon type^node into    by R1.1, R3.1, RA.1   (3.14.1)
{ case T..Dummy:
    {}; endcase                                  by R1.1, R2.8, RA.1   (3.14.2)

  case N2..Sequence:
    trans.C(p1^node); trans.C(p2^node); endcase
          by R1.1, R2.9, R3.2/twice, RA.1/3 times, RA.2/twice  (3.14.3)

  case N3..ConditionalCom:
    trans.E(p1^node).dest.(first.reg)
    { let econd.code = forward(D..COD)
      let fcond.code = forward(D..COD)
      trans.jump.if.false(first.reg, fcond.code)
      trans.C(p2^node)
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      trans.C(p3^node)
      fix.here(econd.code)
    }; endcase
    by R1.1, R2.9, R2.11, R3.2/3 times, R3.3, R5.2, R5.4, R6.2
    RA.1/4 times, RA.2/5 times                                  (3.14.4)

  case N2..While:
    {0 let restart.code = here(D..COD)
       trans.E(p1^node).dest.(first.reg)
       { let fcond.code = forward(D..COD)
         trans.jump.if.false(first.reg, fcond.code)
         trans.C(p2^node)
         trans.jump.to(restart.code)
         fix.here(fcond.code)
    }0; endcase
    by R1.1, R2.8, R2.9/twice, R2.11, R3.2/3 times, R3.3, R5.2, R5.4, R6.1
    R6.2, R6.3, RA.1/3 times, RA.2/4 times                      (3.14.5)

  case N2..RepeatWhile:
    { let restart.code = here(D..COD)
      trans.C(p1^node)
      trans.E(p2^node).dest.(first.reg)
      trans.jump.if.true(first.reg, restart.code)
    }; endcase
    by R1.1, R2.8, R2.9/twice, R2.11, R3.2/3 times, R3.5, R5.2, R5.4, R6.1
    R6.2, R6.3, RA.1/3 times, RA.2/3 times                      (3.14.6)

  case N2..Assignment:
    trans.E(p2^node).dest.(first.reg); trans.update(p1^node, first.reg)
    endcase
    by R1.1, R1.2, R2.2, R2.9, R2.11, R2.12, R3.2/twice, R3.3, R5.2, R5.4
    R5.6, RA.1/3 times, RA.2                                    (3.14.7)
}
```

Snapshot 3.14 (continued)

```
let trans.E(node).dest.(reg) be switchon type^node into
                                    by R1.1, R3.1, R5.1, RA.1  (3.14.8)
{ case T..True:
    trans.load(D..T, TRUE).dest.(reg); endcase
            by R1.1, R2.2, R2.7, R2.14, R5.3, R5.7, RA.1, RA.2  (3.14.9)

  case T..False:
    trans.load(D..T, FALSE).dest.(reg); endcase
            by R1.1, R2.2, R2.7, R2.14, R5.3, R5.7, RA.1, RA.2 (3.14.10)

  case N3..ConditionalExp:
    trans.E(p1^node).dest.(reg)
    { let econd.code = forward(D..COD)
      let fcond.code = forward(D..COD)
      trans.jump.if.false(reg, fcond.code)
      trans.E(p2^node).dest.(reg)
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      trans.E(p3^node).dest.(reg)
      fix.here(econd.code)
    }; endcase
    by R1.1, R2.9, R2.11, R3.2/3 times, R3.3, R5.3/twice, R5.4, R6.2
    RA.1/4 times, RA.2/5 times                                 (3.14.11)

  case T..Ident:
    trans.load(D..Ide, node).dest.(reg); endcase
         by R1.1, R2.2, R2.7, R2.13, R2.14, R5.3, RA.1/twice, RA.2 (3.14.12)

  case N2..AssignmentExp:
    trans.E(p2^node).dest.(reg)
    trans.load(domain.of(reg), reg).dest.(reg)
    trans.update(p1^node, reg); endcase
    by R1.1, R1.2, R2.2, R2.9, R2.11, R2.12, R2.15, R3.2/twice, R3.3, R5.3
    R5.4, R5.6, R5.7, RA.1/3 times, RA.2                       (3.14.13)
}
```

CHAPTER 4

Environment

In this chapter we study the impact on the transformation process that results from incorporating an environment. We consider a flow diagram language with environments based on Table 10.1 of [Sto77]. We also extend that table with functions and procedures; both with one call-by-value parameter.

Snapshot 4.1: Flow Diagrams with Environments. Original Specification

Syntax

i:Ide.	identifiers
c:Com.	commands
e:Exp.	expressions

c ::= **Dummy** | **If** e **Then** c_1 **Else** c_2 | c_1;c_2 | **While** e **Do** c_1 |
    Let i=e **In** c_1 | **Call** e(e_1)

e ::= i | **If** e_1 **Then** e_2 **Else** e_3 | **Let** i=e_1 **In** e_2 | **Fn** i.e_1 | **Fn** i. **Is** c |
    e_1(e_2)

Semantic Domains

T=[{ TRUE } + { FALSE }].	truth values
F=[E > W].	function values
P=[E > C].	procedures values
q:Q.	quotations
O=[T + Q].	output values
s:S=O*.	machine states
c:C=[S > S].	state transformations
e:E=[T + F + P + { ErrorE }].	expression results
W=[S > [E x S]].	expression evaluations
D=E.	denotations
p:U=[Ide > D].	environments

Semantic Domains of 'Interest'

ENV=U.	environments
REG=E.	registered values
STA=S.	states
QUO=Q.	quotations
TEM=[F + P].	templates

Semantic Primitives

Cwrong:[Q > C].
Cwrong=
  λq.Strict{λs.s%q}.

Ewrong:[Q > W].
Ewrong=
  λq.Strict(λs.<ErrorE,s%q>).

Snapshot 4.1 (continued)

Semantic Equations

C:[Com → U → C]. (4.1.1)

C[**Dummy**]p=
Is. (4.1.2)

C[c_1;c_2]p=
C[c_1]p o **C**[c_2]p. (4.1.3)

C[**Let** i=e **In** c_1]p=
E[e]p * λe.**C**[c_1](p[e/[i]]). (4.1.4)

C[**If** e **Then** c_1 **Else** c_2]p=
E[e]p * λe.e?T→e|T→**C**[c_1]p,**C**[c_2]p,Cwrong "condition in <If> not <Boolean>"
. (4.1.5)

C[**While** e **Do** c_1]p=
Fix
{λc.{**E**[e]p *
λe.e?T→e|T→**C**[c_1]p o c,Is,
Cwrong "condition in <While> not <Boolean>"}}. (4.1.6)

C[**Call** e(e_1)]p=
E[e]p *
λe.e?P→**E**[e_1]p * λe'.Strict{e|P}e',
Cwrong "expression in <Call> not <Procedure>". (4.1.7)

E:[Exp → U → W]. (4.1.8)

E[i]p=
Strict(λs.<p[i] In E,s>). (4.1.9)

E[**Let** i=e_1 **In** e_2]p=
E[e_1]p * λe.**E**[e_2](p[e/[i]]). (4.1.10)

E[**If** e_1 **Then** e_2 **Else** e_3]p=
E[e_1]p *
λe.e?T→e|T→**E**[e_2]p,**E**[e_3]p,Ewrong "condition in <If> not <Boolean>"
. (4.1.11)

E[**Fn** i.e_1]p=
Strict(λs.<(λe.**E**[e_1](p[e/[i]])) In E,s>). (4.1.12)

E[**Fn** i. **Is** c]p=
Strict(λs.<{λe.**C**[c](p[e/[i]])} In E,s>). (4.1.13)

E[e_1(e_2)]p=
E[e_1]p *
λe.e?F→**E**[e_2]p * λe'.Strict(e|F)e',
Ewrong "expression in <Call> not <Function>". (4.1.14)

In the specification shown in Snapshot 4.1 the domain of expression results, E=[T + F + P + { ErrorE }], consists of the union of the three basic domains of truth values, functions, procedures, and the error element ErrorE (denoted by $?_E$ in [Sto77]). This is why, unlike Table 10.1 of [Sto77], some equations include a domain check of the form e?D. For a definition of '?' see Appendix C. To check and run the code produced by the generated CGP, we wish to include a pre-declared procedure named [Write]. Once we start pre-declaring names, instead of including the constants [True] and [False] among the syntactic category of expressions, as it is done in [Sto77], we can also pre-declare them as identifiers. The difference with [Sto77] is that our example language 'runs' in a pre-declared environment. This can be specified with the aid of a new valuator P, giving the semantic value of a 'program':

Snapshot 4.2: Pre-declarations. Original Specification

Extensions to Snapshot 4.1

Syntax: Predeclared Identifiers

```
True:Ide.
False:Ide.
Write:Ide.
```

Semantic Equations

```
P:[Com → C].

P[c]=
  C[c](Tu[PWRITE/[Write]][TRUE/[True]][FALSE/[False]]).

PWRITE:P.
PWRITE=
  λe.e?T→Strict{λs.s%(e|T)},Cwrong "expression in <Write> not <Boolean>".
```

The 'top' and 'bottom' element of all single letter domains are predefined identifiers in WFF_s. Their names are made out by appending to the letters 'T' and 'B' (respectively associated with 'top' and 'bottom'), the lower case letter corresponding to the domain in question. In the equation for P above, 'Tu' is the top of 'U'. In the equation for Ewrong in Snapshot 4.1 we used the error element ErrorE. This shows the different possibilities provided in WFF_s; we could have used 'Te' instead, the top element of 'E'.

We have defined the state as a list of output values. These are either produced by PWRITE, which expects a boolean value as its parameter, or by one of the type checking primitives Cwrong or Ewrong, which expect a quotation. These three primitives append their parameter to the output stream. The symbol '%', used in the equations for PWRITE, Cwrong and Ewrong, denotes the list concatenation operator. It is defined in Appendix C and it has not been included among the WFF_s operators because, in all our examples, '%' is used only in equations for primitive functions, which do not intervene in the process of transformation.

4.1 Syntactic Transformations

Recall that in the previous chapter we argued that a strict function on the state corresponds to a hardware activity, hence we eliminated such a function. Now we are presented with strict abstractions in the semantic equations for both procedure and function call, respectively (4.1.7) and (4.1.14). These strict abstractions ensure call by value, an interesting case to which we devote a whole section. For the moment let us assume that we always use call by value. This means that we temporarily define a rule to eliminate the occurrence of Strict. We will see later in Chapter 6 (when analysing the semantic specification of the Lambda Calculus with both call-by-value and call-by-name) how we can 'discover' the particular form of a call by looking at every possible strict or non-strict function. Hence for the moment we define a simple temporary rule which directly eliminates the function Strict:

when e:TEM Strict(e) => e [R3.6]

TEM is a 'Domain of Interest'. Its name derives from the implementation concept of a 'template'; a data structure to implement procedures and functions (see Chapter 7). TEM indicates, in a name independent way, which domains are associated with procedures and functions. In what follows whenever we refer to a template, we mean either a procedure or a function.

4.2 Destination Analysis

4.2.1 Template Declaration

The declaration of a template is specified by the abstractions in (4.1.12) and (4.1.13). Assuming a block structured universe, each template will demand isolation. This means that a new, fresh, data area to keep all values must be defined for every abstraction definition. One of the areas, used to keep temporary values, is the area of destinations. In Chapter 2 we introduced first.reg as either a free constant or a free variable containing a description of a place for run-time temporary values. first.reg then indicates where this area starts. If it is a constant it indicates the first fast register available to contain temporaries, otherwise it is a variable and contains a description, probably as a level and offset, of the start of the corresponding run-time activation record's workspace. Hence, if we wish to isolate the use of destinations within each applied occurrence of a template, then we must ensure that all code, generated within the templates body, makes use of destinations within this new area. The problem is that if the template is declared in a semantic function which produces a value in REG, then R5.1 applies (and R5.2 does not, see section 3.4.1) and all references are made to the parameter reg, which might not be first.reg. Therefore, we must substitute first.reg for reg in the body of the template:

when $e \neq i$ and e:TEM   e   =>   [first.reg/reg]e   [R5.8]

A parameter will also be expected in a particular area, indicated by a free constant or free variable named <u>first.par</u>:

when e:TEM and i:REG e => [first.par/i]e_1 In DOM(e) [R5.9]
where e $\equiv \lambda$i.e_1

Note that, like <u>first.reg</u>, the interpretation of <u>first.par</u> is open to various implementation choices. It contains a description of a place for parameters, if it is a free constant it indicates a fast register. Alternatively if it is a free variable it indicates the start of the corresponding activation record's parameter area. Also, note the change of functionality of [first.par/i]e_1; we still need to remember, for later analysis, the original domain.

The declaration of a template also requires a load operation which is performed by the same primitive procedure <u>trans.load</u>:

when e:TEM e => trans.load(DOM(e), e) In REG [R5.10]

Again, it is up to this primitive to decide what code is in effect necessary. In (4.3.12) and (4.3.13) it will be necessary to load a closure, a label value which, after the Continuation Analysis, will replace the expressions appearing as the second parameter of those two calls of <u>trans.load</u>. For example, according to the machine interface that we have implemented to test our generated CGP, trans.load(D, label).dest.(reg) with an environment link technique [Bor79], will generate the following DEC-10 instructions:

interpreting reg as a fast register (reg=AC)		an invocation record word (reg=#off)	
HRR	AC,label	HRR	AC,label
HRL	AC,BAS	HRRM	AC,#off(BAS)
		HRLM	BAS,#off(BAS)

Snapshot 4.3: Flow Diagrams with Environments. Destination Analysis
A Fragment

```
let C(node, p) be switchon type^node into                by R1.1, R3.1   (4.3.1)
{ case [Call e(e1)]:
    E([e], p).dest.(first.reg)
    first.reg?P->
    E([e1], p).dest.(first.reg+1)
    first.reg|P(first.reg+1).dest.(first.reg),
    Cwrong("expression in <Call> not <Procedure>"); endcase
    by R1.1, R2.9/twice, R2.11/twice, R3.2/4 times, R3.3/twice, R3.6, R5.2
    R5.4/twice, R5.11                                                (4.3.7)
}

let E(node, p).dest.(reg) be switchon type^node into
                                              by R1.1, R3.1, R5.1   (4.3.8)
{ case [Fn i.e1]:
    trans.load(F, E([e1], p([first.par/[i]])).dest.(first.reg)).dest.(reg)
     In E; endcase
    by R1.1, R2.2, R2.7, R2.14, R3.2/twice, R5.3/twice, R5.8, R5.9, R5.10
                                                                    (4.3.12)

  case [Fn i. Is c]:
    trans.load(P, C([c], p([first.par/[i]]))).dest.(reg) In E; endcase
    by R1.1, R2.2, R2.7, R2.14, R3.2/twice, R5.3, R5.8, R5.9, R5.10 (4.3.13)

  case [e1(e2)]:
    E([e1], p).dest.(reg)
    reg?F->E([e2], p).dest.(reg+1); reg|F(reg+1).dest.(first.reg),
    Ewrong("expression in <Call> not <Function>").dest.(reg); endcase
    by R1.1, R2.9/twice, R2.11/twice, R3.2/4 times, R3.3/twice, R3.6, R5.3
    R5.4/twice, R5.11                                               (4.3.14)
}
```

4.2.2 Template Invocation

For a function call we must also ensure that the resultant value has an appropriate destination.

when e:TEM   e(P) => e(P).**dest**.(first.reg)   [R5.11]

Note that, by symmetry, we provide a destination both for procedures and functions, even though for procedures it is not always necessary (it is required for example in NEW of a CLASS in SIMULA67 [Sim68]).

Applying these and all other transformations to bring Snapshot 4.1 to the level of the Destination Analysis results in Snapshot 4.3 where we show only those parts which have been transformed by conversions defined in this chapter.

4.3 Continuation Analysis

We still have not encountered continuations in our example language. Nevertheless, in a similar fashion to the analysis of section 3.4.2, we have to look at those parts where jumps, to and from different parts of the code, need to be produced. The areas to analyse are the specifications of abstraction and application. In these areas the code to be planted obviously has to be related and linked.

4.3.1 Template Declaration

The code associated with the declaration of a template (4.3.12) and (4.3.13), relates to the crucial code fragment, usually refered to as the areas of entry to and exit from a procedure or function. The request for such code can be expressed as:

$$\textbf{when } e_1\text{:TEM} \qquad \left. e(P_0,\ e_1,\ P_1)A \right] \Rightarrow \left[\begin{array}{l} \{\ \textbf{let}\ \text{ntry.code} = \text{trans.entry(node)} \\ e_1 \\ \text{trans.exit(node)} \\ e(P_0,\ \text{ntry.code},\ P_1)A \\ \} \end{array} \right.$$

The parameter node to trans.entry and trans.exit is a reference to the parse-tree node under scrutiny. It is supplied to help the machine interface. For example, it might be used to trace procedure or function entry and exit, with reference to the source statement; or to establish a link between entry and exit for purely code generation purposes; or to set the type of a parameter on entry.

In some cases, we might need to refer to the code planted on exit (for example if the language includes a **resultis**), so we rewrite the rule above as:

```
                                   __         __
                                     |       |  { let ntry.code = forward(DOM(e1))
                                     |       |    let exit.code = forward(COD)
                                     |       |    trans.entry(ntry.code, node)
when e1:TEM    e(P0, e1, P1)A        |  =>   |    e1
                                     |       |    trans.exit(exit.code, node)
                                     |       |    e(P0, ntry.code, P1)A
                                   __|       |__}
```

But because code is planted in a sequential manner, it will be necessary, at the moment of abstraction, to plant appropriate instructions to skip at declaration time over the abstraction body. So we redefine this rule once more:

```
                                   __         __
                                     |       |  { let ntry.code = forward(DOM(e1))
                                     |       |    let exit.code = forward(COD)
                                     |       |    let skip.code = forward(COD)
                                     |       |    trans.jump.to(skip.code)
                                     |       |    trans.entry(ntry.code, node)
when e1:TEM    e(P0, e1, P1)A        |  =>   |    e1                                   [R6.4]
                                     |       |    trans.exit(exit.code, node)
                                     |       |    fix.here(skip.code)
                                     |       |    e(P0, ntry.code, P1)A
                                   __|       |__}
```

To avoid clashes of names, here and in all similar cases, declared variables, like any <u>xxxx.code</u> above will, if required, have a digit appended to its name. Also when the domain associated with one of these variables (the parameter to <u>forward</u>) is not COD, then the postfix <u>code</u> of any <u>xxxx.code</u> is replaced by <u>domD</u>, where D is the associated domain. This is done only as an aid to the eye, the lexical structure of WFF_t names convey no semantic value.

In the example language of this section, there are no recursive templates.

However, in J. Stoy's final example language (Appendix D) there are both recursive functions (D.1.34) and procedures (D.1.35). The characteristic of such semantic specification is the presence of the minimal fix point finder <u>Fix</u> in an expression of the form Fix(λi.e). The instance of i:COD resulted (see R6.3 in section 3.4.2) in the association of the variable 'i' with the first instruction of 'e'. Now we are presented with the case of i:TEM, According to R6.4, the first instruction of a template is indicated by the variable <u>ntry.code</u>, so we simply require:

when i:TEM   Fix(λi.e)   =>   [ntry.code/i]e   [R6.5]

The effect is that while planting code for 'e', if there is a reference to 'i', there will be a reference to the entry code. This is precisely the required effect for a recursive structure. Note that the variable <u>ntry.code</u> is in effect inherited from the transformation of 'e', and follows the same naming conventions mentioned above. The result of applying this rule can be found in (D.2.34) and (D.2.35).

4.3.2 Template Invocation

At the moment of invocation of a template (4.3.7) and (4.3.14), everything seems to be ready. The abstractions values are kept respectively in <u>first.reg</u> and <u>reg</u>, the arguments in <u>first.reg+l</u> and <u>reg+l</u> and the destination for the results is in both cases <u>first.reg</u>. There is only need to plant code to call the procedure or function:

when e:TEM   e(P)A   =>   trans.call(e, P)A   [R6.6]

The actual code planted for call, entry and exit, is not fixed. This depends on the definition of <u>trans.call</u>, <u>trans.entry</u> and <u>trans.exit</u>. One can choose any of the well known machine or language dependent techniques to achieve the desired effect. In the machine interface used to try our generated CGP, we use an environment link technique [Bor79], the DEC-10 code for entry, exit and call is shown below:

```
        garbage collected frames          stack discipline

code for entry:
        NTRY    0,#size                   MOVEM   LNK,0(TOP)
        MOVEM   LNK,0(TOP)                MOVEM   BAS,1(TOP)
        MOVEM   BAS,1(TOP)                MOVE    BAS,TOP
        MOVEM   ENV,2(TOP)                ADDI    TOP,#size
        MOVE    BAS,TOP

code for exit:
        MOVE    LNK,BAS                   MOVE    TOP,BAS
        MOVE    BAS,1(LNK)                MOVE    BAS,1(TOP)
        JRST    0,@0(LNK)                 JRST    0,@0(TOP)

code for call:
                        MOVE    AC,template
                        HLRZ    ENV,AC
                        JSP     LNK,0(AC)
```

The difference between the left hand side 'garbage collected frames' and the right hand side 'stack discipline' is that in the former, space for invocation frames is obtained through the pseudo-op NTRY, which returns in TOP a pointer to a new frame #size words long (this area must be garbage collected). In the latter, #size words are obtained from the stack, and are released on exit. The example language of this chapter accepts functions and procedures both as parameters and as function results, hence we must use the former in this language.

4.3.3 Type Checking

The process of type checking can be regarded as a [TRE → TRE] transformation which is applied prior to the process of code generation. The corresponding parts of a semantic specification would be abstracted at the level of a 'static' semantics (in the sense of [ADA80]). This phase is of no interest to us, our primary objective is the analysis of code generation. However, type-checking can be regarded as a parallel phase to code generation, either because there is run-time type checking or because the language under scrutiny embeds in its 'dynamic' semantics (also in the sense of [ADA80]) some sort of 'domain-check', which might call for either a compile or run-time type check. In all previous examples, this matter has been avoided by considering a very simple domain of expression results E=T. But in the current example, the domain of expression results is E=[T + F + P + { ErrorE }]. This is why, each equation requiring one particular value in E, has to check, using the WFF_s operator '?', if the given value is in the expected summand. From a code generation standpoint, this domain check is associated with a type-checking process. To decide whether this type-checking should be done at compile or at run-time, we simple look for 'registered' values associated with the check. Assuming a domain check is used only in the boolean part of a conditional, we define firstly:

when i:REG   i?d   =>   trans.skip.if.in(i, d)   [R6.7]

And secondly, we extend the **when** and **where** clauses of R6.2 as follows:

when (E0IsDes or E0IsIde or E0IsInt) and i:REG
where C_3 = E0IsIde → null,
        Reverse and E0IsInt → trans.skip.if.not.in(P),
        e_0
    C_4 = E0IsSkp → trans.jump.to(FalseCo), JumpRut(i, FalseCo)
    E0IsInt = e_0=trans.skip.if.in(P)

The primitive procedure <u>trans.skip.if.in</u> will have to plant appropriate instructions to 'run-time' type check the domain associated with a destination. For example, using two words, one for a value and another for a type; the generated CGP will generate in turn:

code for i?d -> c1, c2:

```
        DMOVE   AC2,i           ; as a result of R6.7
        SKIPE   AC3,d
        JRST    0,F             ; R6.2
        c1
        JRST    0,E
F:      c2
E:
```

The transformation rule above, is triggered by a condition on REG. This suggests its counterpart, the compile-time type checking rule:

when not i:REG   i?d   =>   check.if.in(i, d)

check.if.in does not plant any code, it compile-time checks the description of a value (i) with respect to a type (d). This method works well provided all expressions involving a compile time check are not in REG. To see why this is not enough consider the example language of this chapter. This language clearly requires run time type checking, because:

- There are no explicit types provided by the syntax
- Expressions can result in booleans, functions or procedures.
- These values can be passed as parameters or returned as the value of functions.
- The double arm conditional expression does not guarantee 'balancing' in the sense of ALGOL68 [Wij75].

But suppose that we impose certain syntactic restrictions, which guarantee that run-time type checking is not necessary. Suppose we restrict the use of the conditional and the kind of values passed to and from functions. These restrictions, whatever their nature, do not necessarily require a different semantic specification, but our transformational system will still generate a CGP with run-time type checking. The problem is that the two rules above are triggered by registered values in REG, which are independent of any syntactic restriction. To overcome this problem, we introduce an implementation issue at the level of a semantic specification. When we require a compile-time type check we shall use the static domain check '??', instead of the dynamic '?'. This is a similar operator which can be trapped during the transformation process to impose our requirement. We rewrite the rule above as:

e => check.if.in(i, d) [R6.8]
when (e $\underline{=}$ i?d and not i:REG) or e $\underline{=}$ i??d

This mechanism of type checking, both at compile and at run-time, requires destinations to be carriers of type information. This is why, the primitive <u>trans.load</u> takes a domain name as one of its parameters. It is up to the definition of <u>trans.load</u> to either associate a type to its destination descriptor, for compile time type checking, or to plant appropriate instructions to load, at run time, a type to be associated with a fast register or activation record location.

Cond and Scond: In some examples of [Sto77], the conditional is avoided by explicit use of the function <u>Cond</u>. This is a generic function, and it is a predefined WFF_s identifier. Its counterpart <u>Scond</u> (Static Cond) indicates compile-time type checking.

Cond	:	[[A x A] → B → A]	
Cond<e_1, e_2>e_0	=	e_0?T → (e|T → e_1, e_2), Wrong	[D8]
Scond	:	[[A x A] → B → A]	
Scond<e_1, e_2>e_0	=	e_0??T → (e|T → e_1, e_2), Wrong	[D9]

These definitions lead naturally to the following Normalisation rules:

Cond<e_1, e_2>e_0 => e_0?T → (e_0|T → e_1, e_2), Wrong [R1.4]

Scond<e_1, e_2>e_0 => e_0??T → (e_0|T → e_1, e_2),Wrong [R1.5]

So that R6.7 or R6.8 (the corresponding run-time or compile-time rules) can be applied accordingly.

In Snapshot 4.4 we show the result of the Continuation Analysis. Again, to

Snapshot 4.4: Flow Diagrams with Environments. Continuation Analysis

A Fragment

```
let C(node, p) be switchon type^node into                          no change
{ case [Call e(e1)]:
    E([e], p).dest.(first.reg)
    { let econd.code = forward(COD)
      let fcond.code = forward(COD)
      trans.skip.if.in(first.reg, P)
      trans.jump.to(fcond.code)
      E([e1], p).dest.(first.reg+1)
      trans.call(first.reg|P, first.reg+1).dest.(first.reg)
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      Cwrong("expression in <Call> not <Procedure>")
      fix.here(econd.code)
    }; endcase
}                                              by R6.2, R6.6, R6.7   (4.4.7)
let E(node, p).dest.(reg) be switchon type^node into                no change
{ case [Fn i.e1]:
    { let ntry.domF = forward(F)
      let exit.code = forward(COD)
      let skip.code = forward(COD)
      trans.jump.to(skip.code)
      trans.entry(ntry.domF, node)
      E([e1], p([first.par/[i]])).dest.(first.reg)
      trans.exit(exit.code, node)
      fix.here(skip.code)
      trans.load(F, ntry.domF).dest.(reg)
    }; endcase
  case [Fn i. Is c]:                                      by R6.4   (4.4.12)
    { let ntry.domP = forward(P)
      let exit.code = forward(COD)
      let skip.code = forward(COD)
      trans.jump.to(skip.code)
      trans.entry(ntry.domP, node)
      C([c], p([first.par/[i]]))
      trans.exit(exit.code, node)
      fix.here(skip.code)
      trans.load(P, ntry.domP).dest.(reg)
    }; endcase
  case [e1(e2)]:                                          by R6.4   (4.4.13)
    E([e1], p).dest.(reg)
    { let econd.code = forward(COD)
      let fcond.code = forward(COD)
      trans.skip.if.in(reg, F)
      trans.jump.to(fcond.code)
      E([e2], p).dest.(reg+1)
      trans.call(reg|F, reg+1).dest.(first.reg)
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      Ewrong("expression in <Call> not <Function>").dest.(reg)
      fix.here(econd.code)
    }; endcase
}                                              by R6.2, R6.6, R6.7   (4.4.14)
```

avoid clustering, only those parts affected by the transformations of this section are shown. Moreover, the conditional command and the while loop, have been removed, because the effect of R6.7 is similar to the one shown in (4.4.7) and (4.4.14).

4.4 Environment Analysis

We wish to maintain one single compile (or run) time symbol structure, global for any procedure requiring access to it. This assumes a block structured use of the environment. As with any data structure, we need to insert, delete and find elements (descriptors). In the current example, the environment is passed around and it gets updated in both types of definitions by denotation (4.1.4) and (4.1.10). Both updates occur within the context of recursive procedures. To maintain such a symbol structure - global to a set of mutually recursive procedures - we have to insert and delete locally to every recursive activation. The CGP will have to remember - within each recursive activation - which objects are declared, so that before exiting that particular activation, the same objects can be undeclared in turn. The transformations that follow will eliminate all environments from parameter lists, and will 'sandwich' declarations with the corresponding reset action.

```
                                    __             __
                                      |           |  { let x = e2
                                      |           |    declare(e1, x)
   e0(P0, i([e1/e2]), P1)A            |    =>     |    e0(P0, P1)A
                                      |           |    undeclare(x)
when i:ENV                          __|           |__}
```

This has the desired effect, but we wish to be more general. In a different language we might have more than one declaration, and the undeclaring activity might be too expensive. We therefore, rewrite the rule above, adding also a similar one for a **let** declaration:

```
                                         __      __
                                           |    |  { let old.env=this.env
                                           |    |    e
                       e0(P0, e, P1)A      | => |    e0(P0, P1)A                        [R7.1]
                                           |    |    reset(old.env)
when e:ENV and e≠i                       __|    |__}

when i:ENV             i([e1/e2])         =>       declare(DOM(e1), e1, e2)           [R7.2]
                                         __      __
                                           |    |  { let old.env = this.env
                       { let i = e         |    |    e
when i:ENV               C                 | => |    C                                  [R7.3]
                       }                   |    |    reset(old.env)
                                         __|    |__}
```

The variable <u>this.env</u> is a reference to the current environment (a symbol structure plus a stack, an A-List or whatever implementation choice has been made). The assignment to <u>old.env</u> remembers, locally to each recursive activation, the state of the current environment. <u>declare</u> updates it and <u>reset</u> puts it back to the original state. The first parameter to <u>declare</u>, like the one supplied to <u>trans.load</u>, is given for type checking purposes. <u>declare</u> might or might not produce code. In particular, if the declared object is not known at compile time, (because, say, it is a destination in REG, which will be associated with a particular value at run-time) then <u>declare</u> will have to associate a temporary location with the declared name. In our machine interface, the following equivalence holds:

```
                           __      __
                             |    |  { LET I = new.loc()
     declare(P0, reg, P1)    | => |    trans.update(I, reg)
                             |    |    declare(P0, I, P1)
                           __|    |__}
```

We are not including this equivalence as rule of our transformational system. It is a procedural action within the machine interface. Transformations like this, could be added at every level, but such transformations are in effect macro expansions and we do not pursue further in this direction.

R7.1, R7.2 and R7.3 fulfil two requirements, i.e: insertion and deletion. To search for an element in the global symbol structure we need:

when i:ENV   i(P)A   =>   look.up(P)A   [R7.4]

We are in effect presented here with a choice of 'styles' of target CGP. Suppose we extend R5.3 and R5.7, both defined in section 3.4.1, with a further condition acting on the domain of interest ENV, as follows:

e(P)   =>   e(P).**dest.**(reg) **In** COD
when e(P):REG and not (e=i and i:ENV)

$\{C_0;\ e;\ C_1\}$ => $\{C_0;\ C;\ C_1\}$
or
$e_0 \rightarrow e,\ e_2$ => $e_0 \rightarrow C,\ e_2$
or
$e_0 \rightarrow e_1,\ e$ => $e_0 \rightarrow e_1,\ C$
where C = trans.load(DOM(e), e) **In** REG
when e:REG and not e:COD and (e=i or (e=i(P) and i:ENV))

Under the transformations dictated by the these two rules, the procedural text corresponding to an identifier within an expression would be:

trans.load(domain.of(look.up([i])), look.up([i])).**dest.**(reg)

This corresponds to a view which associates, to the process of looking up a value, the activity which involves only a read of a symbol table description. We have experimented with this version. It was attractive because it structured the CGP, splitting a look-up from a load activity. In more complex languages, however, like the one considered in Chapter 6, (the Lambda Calculus with both call-by-value and call-by-name) the structure of look.up requires the following process: looking up a symbol table; loading a value; call of a 'thunk' to find the value associated with a 'name' expression; and jump if necessary to the appropriate continuation. Also, we wish to prepare the shape of our transformational system to handle not only the 'static binding' mechanism that we practise in the present examples, but also 'dynamic binding'. In this case, all primitive procedures which maintain a symbol structure at 'compile-time', should instead plant appropriate code to do the same, at 'run-time'. In this case we also require look.up to be a separate process. We do not proceed with the proposed extensions to R5.3 and R5.7. The corresponding procedural text under the original definition of these two rules is: look.up([i]).**dest**.(reg) (4.5.9).

Now that the environment is global to all the CGP's procedures, we can eliminate it from parameter lists:

when i:ENV **let** $v(P_0, i, P_1)A$ => **let** $v(P_0, P_1)A$ [R7.5]

when i:ENV $e(P_0, i, P_1)A$ => $e(P_0, P_1)A$ [R7.6]

In Snapshot 4.5 we show the effect of these transformation rules. We display only the cases corresponding to both **let** declarations and the identifier among expressions. The other cases transformed by the Environment Analysis are similar to those shown.

4.5 Optimising Transformations

4.5.1 Dumping

If we now consider the interpretation of first.reg as a fast register then we cannot leave the occurrence of the expressions first.reg+1 and reg+1 of Snapshot 4.4 as they stand: On the one hand it is not always true that every intermediate result will occupy one register's word. On the other hand in a recursive procedure (of the CGP), such an expression assumes the existence of an infinite supply of fast registers. What we require is to check whether or not the first.reg+1 or reg+1 is available. This can be done, if we assume a weighted tree [Bor79]. In this case weight selects the number of registers

Snapshot 4.5: Flow Diagrams with Environments. Environment Analysis
A Fragment

```
let C(node) be switchon type^node into                    by R7.5    (4.5.1)
{ case [Let i=e In c1]:
    E([e]).dest.(first.reg)
    { let old.env = this.env
      declare(domain.of(first.reg), first.reg, [i])
      C([c1])
      reset(old.env)
    }; endcase                                   by R7.1, R7.2, R7.6  (4.5.4)
}

let E(node).dest.(reg) be switchon type^node into          by R7.5    (4.5.8)
{ case [i]:
    look.up([i]).dest.(reg) In E; endcase                  by R7.4    (4.5.9)

  case [Let i=e1 In e2]:
    E([e1]).dest.(reg)
    { let old.env = this.env
      declare(domain.of(reg), reg, [i])
      E([e2]).dest.(reg)
      reset(old.env)
    }; endcase                                   by R7.1, R7.2, R7.6  (4.5.10)
}
```

required by a particular node and max.reg indicates the last available register. The CGP can then check if the request can be granted. If it can, then a new destination is obtained through a call of the function next, so that the appropriate offset can be evaluated. If it can not be granted, then a dump operation takes place. We formalise this by:

```
                 __     __
                   |   |  test E=max.reg
                   |   |  then
                   |   |  { let old.env = this.env
                   |   |    let D = trans.dump(R)
              C    | =>|    [R/R+1]{D/R}C                          [R9.1]
                   |   |    reset(old.env)
                   |   |  } or
                   |   |  { let nxt = next(R)
                   |   |    [nxt/R+1]C
                 __|   |__} rename D=>dmp.loc
when   C ≡ { C1; C2; C3 } and C2≡ e(P)A.dest.(R+1)
       R ≡ reg or R ≡ first.reg
where  E = weight^[s] if C2≡ e(P0, [s], P1)A.dest.(R+1) (P contains an [s])
       E = R otherwise
```

Note that we are using the special substitution rule { / }, whose definition is similar to the normal substitution rule [/], except when entering an auxiliary parameter list AUX, where it does not substitute **.dest.**(P). We have to substitute only non-destination registers, because a destination is an implicit declaration, introduced by rules like R5.4, among others. The trimmed form (to show only the relevant detail) is:

```
{ let i=e(P); C }    =>    { e(P).dest.(i) ; C }
```

Also this requires a definition of 'free', which 'feels' destinations as declarations.

<u>trans.dump</u> gets a new temporary destination (a location) and plants appropriate instructions to store the contents of <u>R</u>. The temporary destination needs to remain 'in scope' only while code for [R/R+1]{D/R}C is planted. This is why the dump activity is surrounded by the 'brackets':
let old.env = this.env ; C ; reset(old.env)

4.5.2 Multiple Declarations

More than one declaration, in the same equation, results in multiple declarations which can easily be optimised as follows:

```
{ let old.env = this.env      |   |
   C1                         |   |  { let old.env = this.env
  {let old.env = this.env     |   |     C1
    C2                        | =>|     C2
    reset(old.env)            |   |     reset(old.env)
  }                           |   |  }
  reset(old.env)              |   |
}                             |   |
```

[R9.2]

4.5.3 Loading

It is possible for the generation process to request loads from a register into itself. This can be easily trapped, but we have to be cautious: A load operation also assigns a type to a register for compile or run-time type checking purposes. Hence, we can eliminate the load operation but not the type definition, except when there is no type alteration.

```
   trans.load(E, I).dest.(I)  | => |  make.type(I, E)          [R9.3]
when E ≠ domain.of(I)         |    |

   trans.load(E, I).dest.(I)  | => |  {}                       [R9.4]
when E = domain.of(I)         |    |
```

<u>domain.of</u> was defined in section 3.4.1, as part of the definition of <u>DOM</u>. <u>make.type</u> is a primitive operation that associates a type to a destination.

We also take the oportunity to eliminate possible expensive duplicate sub-expressions produced by the introduction of <u>domain.of</u>. This will happen in general in the parameter list of <u>trans.load</u> or <u>declare</u>.

```
      E0(domain.of(E), E, P)A  | => |  E0(domain.of(xx), xx)A   [R9.5]
when E≠I                       |    |  where xx = E
```

Note that the **where** statement defining xx is BCPL syntax. Is is not included as a WFF_t expression since it is trivially equivalent to a BCPL **let**.

4.6 BCPL

When a conditional does not denote a run time activity and appears in a block, we have to transform it into one of the different types of BCPL conditional commands, depending on the form of each branch:

```
{ C0                 |    |  { C0
  e0->e1,e2          | => |    test e0 then e1 or e2      [RA.3]
  C1                 |    |    C1
}                    |    |  }

{ C0                 |    |  { C0
  e0->e1,{}          | => |    if e0 then e1              [RA.4]
  C1                 |    |    C1
}                    |    |  }
```

Snapshot 4.6: Compile-Time Type Checking. BCPL

```
case N3..ConditionalCom:
    trans.E(p1^node).dest.(first.reg)
    test check.if.in(first.reg, D..T)
    then { let econd.code = forward(D..COD)
           let fcond.code = forward(D..COD)
           trans.jump.if.false(first.reg, fcond.code)
           trans.C(p2^node)
           trans.jump.to(econd.code)
           fix.here(fcond.code)
           trans.C(p3^node)
           fix.here(econd.code)
         }
    or  Cwrong("condition in <If> not <Boolean>"); endcase
    by R1.1, R2.9, R2.11, R3.2/4 times, R3.3, R5.2, R5.4, R6.2, R6.8
    R7.6/3 times, RA.1/4 times, RA.2/6 times, RA.3                    (4.6.5)
```

$$\left[\begin{array}{l}\{\ C_0 \\ \ \ e_0 \to \{\}, e_2 \\ \ \ C_1 \\ \}\end{array}\right] \Rightarrow \left[\begin{array}{l}\{\ C_0 \\ \ \ \textbf{unless}\ e_0\ \textbf{do}\ e_2 \\ \ \ C_1 \\ \}\end{array}\right] \qquad [RA.5]$$

Among other applications, these rules are applied in a compile-time type checking process. Suppose that the syntactic restrictions refered to in section 4.3.3 apply to our current example language. The equation for a conditional will be rewritten exactly as in (4.1.5) except for the static check '??' instead of the dynamic '?'. The resultant procedural case (4.6.5) shown in Snapshot 4.6 should be compared with the corresponding (4.7.5) in Snapshot 4.7.

We have completed all conversions required for the transformation processes of this chapter. In Snapshot 4.7, we display the final version in BCPL; this time we include all syntactic categories. The example language, used at this point, is now powerful enough to allow us to show the kind of code that we are able to generate. The reader must be aware, that the topic of this thesis, merges both the theoretical issues of a semantic universe, with implementation issues of code generation techniques. It is therefore imperative to show, at certain point, the sort of code that we can produce.

Consider the following input program:

```
Let f=Fn i.
     Fn j.i
In Let g=f(True)
   In Let x=g(False)
      In Call Write(x)
```

This trivial program involves the high level concept of a function returning a function and it serves to emphasise the mechanism and treatment of the environment presented in this chapter. The above program, compiled with the CGP shown in Snapshot 4.7 produced the following DEC-10 code:

```
code for: Let f=Fn .... etc
code for: Fn i. etc
        JRST    0,L5              ; trans.jump(L5, TRUE)
L6:     NTRY    0,0               ; trans.entry(L6, Fn i. etc)
        MOVEM   LNK,0(TOP)
        MOVEM   BAS,1(TOP)
        MOVEM   ENV,2(TOP)
        MOVE    BAS,TOP
        DMOVEM  AC4,3(BAS)        ; declare({}, AC4, i)
code for: Fn j.i
        JRST    0,L7              ; trans.jump(L7, TRUE)
L8:     NTRY    0,0               ; trans.entry(L8, Fn j.i)
        MOVEM   LNK,0(TOP)
        MOVEM   BAS,1(TOP)
        MOVEM   ENV,2(TOP)
        MOVE    BAS,TOP
        DMOVEM  AC4,3(BAS)        ; declare({}, AC4, j)
code for: i
        HRR     ENV,2(BAS)
        DMOVE   AC2,3(ENV)        ; trans.load({}, [L1,#003]).dest.(AC2)
        MOVE    LNK,BAS           ; trans.exit(forward, Fn j.i)
        MOVE    BAS,1(LNK)
        JRST    0,@0(LNK)         ; 5 to entry
L7:     HRRI    AC2,L8
        HRL     AC2,BAS           ; trans.load(F, L8).dest.(AC2)
        MOVEI   AC3,F             ; make.type(AC2, F)
        MOVE    LNK,BAS           ; trans.exit(forward, Fn i. etc)
        MOVE    BAS,1(LNK)
        JRST    0,@0(LNK)         ; 8 to entry
```

```
L5:     HRRI    AC2,L6
        HRL     AC2,BAS          ; trans.load(F, L6).dest.(AC2)
        MOVEI   AC3,F            ; make.type(AC2, F)
        DMOVEM  AC2,10+Base      ; declare(F, AC2, f)
 code for: Let g=f(True) etc
 code for: f(True)
 code for: f
        DMOVE   AC2,10+Base      ; trans.load(F, [L0,#010]).dest.(AC2)
        CAIE    AC3,F            ; trans.skip.if.in(AC2, F)
        JRST    0,L9             ; trans.jump(L9, TRUE)
 code for: True
        DMOVE   AC4,4+Base       ; trans.load(T, [L0,#004]).dest.(AC4)
        HLRZ    ENV,AC2
        JSP     LNK,0(AC2)       ; trans.call(AC2, AC4)
        JRST    0,L10            ; trans.jump(L10, TRUE)
L9:     SETO    AC2,0            ; ErrorE (null value)
        SETO    AC3,0            ; ErrorE (null type)
        OUs     0,[ASCIZ/*C*L?Ewrong expression in <Call> not <Function>/]
L10:    DMOVEM  AC2,12+Base      ; declare(F, AC2, g)
 code for: Let x=g(False) etc
 code for: g(False)
 code for: g
        DMOVE   AC2,12+Base      ; trans.load(F, [L0,#012]).dest.(AC2)
        CAIE    AC3,F            ; trans.skip.if.in(AC2, F)
        JRST    0,L11            ; trans.jump(L11, TRUE)
 code for: False
        DMOVE   AC4,6+Base       ; trans.load(T, [L0,#006]).dest.(AC4)
        HLRZ    ENV,AC2
        JSP     LNK,0(AC2)       ; trans.call(AC2, AC4)
        JRST    0,L12            ; trans.jump(L12, TRUE)
L11:    SETO    AC2,0            ; ErrorE (null value)
        SETO    AC3,0            ; ErrorE (null type)
        OUs     0,[ASCIZ/*C*L?Ewrong expression in <Call> not <Function>/]
L12:    DMOVEM  AC2,14+Base      ; declare(F, AC2, x)
 code for: Call Write(x)
 code for: Write
        DMOVE   AC2,0+Base       ; trans.load(P, [L0,#000]).dest.(AC2)
        CAIE    AC3,P            ; trans.skip.if.in(AC2, P)
        JRST    0,L13            ; trans.jump(L13, TRUE)
 code for: x
        DMOVE   AC4,14+Base      ; trans.load(F, [L0,#014]).dest.(AC4)
        HLRZ    ENV,AC2
        JSP     LNK,0(AC2)       ; trans.call(AC2, AC4)
        JRST    0,L14            ; trans.jump(L14, TRUE)
L13:    OUs     0,[ASCIZ/*C*L?Cwrong expression in <Call> not <Procedure>/]
```

Snapshot 4.7: Flow Diagrams with Environments. BCPL

```
let trans.C(node) be switchon type^node into
                                    by R1.1, R3.1, R7.5, RA.1   (4.7.1)
{ case T..Dummy:
    {}; endcase                        by R1.1, R2.8, RA.1   (4.7.2)

  case N2..Sequence:
    trans.C(p1^node); trans.C(p2^node); endcase
    by R1.1, R2.9, R3.2/twice, R7.6/twice, RA.1/3 times, RA.2/twice  (4.7.3)

  case N3..DefinitionByDenotationCom:
    trans.E(p2^node).dest.(first.reg)
    { let old.env = this.env
      declare(domain.of(first.reg), first.reg, p1^node)
      trans.C(p3^node)
      reset(old.env)
    }; endcase
    by R1.1, R2.9, R2.11, R3.2/3 times, R3.3, R5.2, R5.4, R7.1, R7.2, R7.6
    RA.1/4 times, RA.2/twice                                       (4.7.4)

  case N3..ConditionalCom:
    trans.E(p1^node).dest.(first.reg)
    { let econd.code = forward(D..COD)
      let fcond.code = forward(D..COD)
      trans.skip.if.in(first.reg, D..T)
      trans.jump.to(fcond.code)
      { let econd.code = forward(D..COD)
        let fcond.code = forward(D..COD)
        trans.jump.if.false(first.reg, fcond.code)
        trans.C(p2^node)
        trans.jump.to(econd.code)
        fix.here(fcond.code)
        trans.C(p3^node)
        fix.here(econd.code)
      }
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      Cwrong("condition in <If> not <Boolean>")
      fix.here(econd.code)
    }; endcase
    by R1.1, R2.9, R2.11, R3.2/4 times, R3.3, R5.2, R5.4, R6.2/twice, R6.7
    R7.6/3 times, RA.1/4 times, RA.2/8 times                      (4.7.5)
```

Snapshot 4.7 (continued)

```
  case N2..While:
    {0 let restart.code = here(D..COD)
       trans.E(p1^node).dest.(first.reg)
       { let econd.code = forward(D..COD)
         let fcond.code = forward(D..COD)
         trans.skip.if.in(first.reg, D..T)
         trans.jump.to(fcond.code)
         { let fcond.code = forward(D..COD)
           trans.jump.if.false(first.reg, fcond.code)
           trans.C(p2^node)
           trans.jump.to(restart.code)
           fix.here(fcond.code)
         }
         trans.jump.to(econd.code)
         fix.here(fcond.code)
         Cwrong("condition in <While> not <Boolean>")
         fix.here(econd.code)
    }0; endcase
    by R1.1, R2.8, R2.9/twice, R2.11, R3.2/4 times, R3.3, R5.2, R5.4, R6.1
    R6.2/twice, R6.3, R6.7, R7.6/twice, RA.1/3 times, RA.2/7 times   (4.7.6)

  case N2..Call:
    trans.E(p1^node).dest.(first.reg)
    { let econd.code = forward(D..COD)
      let fcond.code = forward(D..COD)
      trans.skip.if.in(first.reg, D..P)
      trans.jump.to(fcond.code)
      test weight^p2^node=max.reg
      then { let old.env = this.env
             let dmp.loc = trans.dump(first.reg)
             trans.E(p2^node).dest.(first.reg)
             trans.call(dmp.loc, first.reg).dest.(first.reg)
             reset(old.env)
           }
      or   { let nxt = next(first.reg)
             trans.E(p2^node).dest.(nxt)
             trans.call(first.reg, nxt).dest.(first.reg)
           }
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      Cwrong("expression in <Call> not <Procedure>")
      fix.here(econd.code)
    }; endcase
    by R1.1, R2.9/twice, R2.11/twice, R3.2/4 times, R3.3/twice, R3.6, R5.2
    R5.4/twice, R5.11, R6.2, R6.6, R6.7, R7.6/twice, R9.1, RA.1/5 times
    RA.2/6 times
                                                                    (4.7.7)
}

let trans.E(node).dest.(reg) be switchon type^node into
                              by R1.1, R3.1, R5.1, R7.5, RA.1      (4.7.8)
{ case T..Ident:
    look.up(node).dest.(reg); endcase
        by R1.1, R2.2, R2.7, R2.14, R3.2, R5.3, R7.4, RA.1/twice  (4.7.9)
```

Snapshot 4.7 (continued)

```
case N3..DefinitionByDenotationExp:
  trans.E(p2^node).dest.(reg)
  { let old.env = this.env
    declare(domain.of(reg), reg, p1^node)
    trans.E(p3^node).dest.(reg)
    reset(old.env)
  }; endcase
  by R1.1, R2.9, R2.11, R3.2/3 times, R3.3, R5.3, R5.4, R7.1, R7.2, R7.6
  RA.1/4 times, RA.2/twice                                          (4.7.10)

case N3..ConditionalExp:
  trans.E(p1^node).dest.(reg)
  { let econd.code = forward(D..COD)
    let fcond.code = forward(D..COD)
    trans.skip.if.in(reg, D..T)
    trans.jump.to(fcond.code)
    { let econd.code = forward(D..COD)
      let fcond.code = forward(D..COD)
      trans.jump.if.false(reg, fcond.code)
      trans.E(p2^node).dest.(reg)
      trans.jump.to(econd.code)
      fix.here(fcond.code)
      trans.E(p3^node).dest.(reg)
      fix.here(econd.code)
    }
    trans.jump.to(econd.code)
    fix.here(fcond.code)
    Ewrong("condition in <If> not <Boolean>").dest.(reg)
    fix.here(econd.code)
  }; endcase
  by R1.1, R2.9, R2.11, R3.2/4 times, R3.3, R5.3/3 times, R5.4
  R6.2/twice, R6.7, R7.6/3 times, RA.1/4 times, RA.2/8 times        (4.7.11)

case N2..Abstraction:
  { let ntry.domF = forward(D..F)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code)
    trans.entry(ntry.domF, node)
    { let old.env = this.env
      declare(domain.of(first.par), first.par, p1^node)
      trans.E(p2^node).dest.(first.reg)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..F, ntry.domF).dest.(reg)
  }; endcase
  by R1.1, R2.2, R2.7, R2.14, R3.2/twice, R5.3/twice, R5.8, R5.9, R5.10
  R6.4, R7.1, R7.2, RA.1/3 times, RA.2/5 times                      (4.7.12)
```

Snapshot 4.7 (continued)

```
case N2..Routine:
  { let ntry.domP = forward(D..P)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code)
    trans.entry(ntry.domP, node)
    { let old.env = this.env
      declare(domain.of(first.par), first.par, pl^node)
      trans.C(p2^node)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..P, ntry.domP).dest.(reg)
  }; endcase
  by R1.1, R2.2, R2.7, R2.14, R3.2/twice, R5.3, R5.8, R5.9, R5.10, R6.4
  R7.1, R7.2, RA.1/3 times, RA.2/5 times                               (4.7.13)

case N2..Application:
  trans.E(pl^node).dest.(reg)
  { let econd.code = forward(D..COD)
    let fcond.code = forward(D..COD)
    trans.skip.if.in(reg, D..F)
    trans.jump.to(fcond.code)
    test weight^p2^node=max.reg
    then { let old.env = this.env
           let dmp.loc = trans.dump(reg)
           trans.E(p2^node).dest.(reg)
           trans.call(dmp.loc, reg).dest.(first.reg)
           reset(old.env)
         }
    or   { let nxt = next(reg)
           trans.E(p2^node).dest.(nxt)
           trans.call(reg, nxt).dest.(first.reg)
         }
    trans.jump.to(econd.code)
    fix.here(fcond.code)
    Ewrong("expression in <Call> not <Function>").dest.(reg)
    fix.here(econd.code)
  }; endcase
  by R1.1, R2.9/twice, R2.11/twice, R3.2/4 times, R3.3/twice, R3.6, R5.3
  R5.4/twice, R5.11, R6.2, R6.6, R6.7, R7.6/twice, R9.1, RA.1/5 times
  RA.2/6 times
}                                                                      (4.7.14)
```

CHAPTER 5

Continuations

In this chapter we analyse the correspondence between continuations and code pointers. As an example, we use the Flow Diagrams Language with Jumps based on Table 11.1 of [Sto77]. We will also define all transformation rules required for the final example of [Sto77] and for GEDANKEN [Rey70]. In the following sections any reference to these languages is made with respect to the Original Specification and final CGP in BCPL as shown respectively in Appendices D and E.

Snapshot 5.1 below shows the semantic specification used as an example in this chapter. The semantic equation for blocks is, however, missing. This equation requires special treatment and will be analysed separately later in this chapter. We also remove from Table 11.1 of [Sto77] the two syntactic constructs [True] and [False], assuming a similar strategy to that one of the previous chapter.

The required analysis starts at the level of the Semantic Transformations. We therefore bring the transformation process up to the level of the Syntactic Transformations, as shown in Snapshot 5.2

Snapshot 5.1: Flow Diagrams with Jumps. Original Specification

Syntactic Categories

i:Ide.	identifiers
c:Com.	commands
e:Exp.	expressions

Syntax

c ::= **Dummy** | **If** e **Then** c_1 **Else** c_2 | c_1;c_2 | **While** e **Do** c_1 |
   Let i=e **In** c_1 | **Goto** e

e ::= i | **If** e_1 **Then** e_2 **Else** e_3 | **Let** i=e_1 **In** e_2

Snapshot 5.1 (continued)

Semantic Domains

T=[{ TRUE } + { FALSE }].	truth values
s:S.	machine states
A.	answers
c:C=[S → A].	command cont.
e:E=[T + C].	expression values
d:D=E.	denotations
k:K=[E → C].	expression cont.
w:W=[K → C].	expression closures
G=[C → C].	command closures
p:U=[Ide → D].	environments

Semantic Primitive

Wrong:C.	undefined

Semantic Domains of 'Interest'

ENV=U.	environments
REG=E.	registered values
STA=S.	states
ANS=A.	answers

Semantic Equations

$$C:[Com \to U \to G]. \tag{5.1.1}$$

$$C[\mathbf{Dummy}]pc= \\ c. \tag{5.1.2}$$

$$C[\mathbf{If}\ e\ \mathbf{Then}\ c_1\ \mathbf{Else}\ c_2]pc= \\ E[e]p\{\lambda e.Cond\langle C[c_1]pc, C[c_2]pc\rangle e\}. \tag{5.1.3}$$

$$C[c_1;c_2]pc= \\ C[c_1]p\{C[c_2]pc\}. \tag{5.1.4}$$

$$C[\mathbf{While}\ e\ \mathbf{Do}\ c_1]pc= \\ Fix\{\lambda c'.E[e]p\{\lambda e.Cond\langle C[c_1]pc', c\rangle e\}\}. \tag{5.1.5}$$

$$C[\mathbf{Let}\ i{=}e\ \mathbf{In}\ c_1]pc= \\ E[e]p\{\lambda e.C[c_1](p[e\ In\ D/[i]])c\}. \tag{5.1.6}$$

$$C[\mathbf{Goto}\ e]pc= \\ E[e]p\{\lambda e.e?C \to e|C, Wrong\}. \tag{5.1.7}$$

$$E:[Exp \to U \to W]. \tag{5.1.8}$$

$$E[i]pk= \\ k\{p[i]\}. \tag{5.1.9}$$

$$E[\mathbf{If}\ e_1\ \mathbf{Then}\ e_2\ \mathbf{Else}\ e_3]pk= \\ E[e_1]p\{\lambda e.Cond\langle E[e_2]pk, E[e_3]pk\rangle e\}. \tag{5.1.10}$$

$$E[\mathbf{Let}\ i{=}e_1\ \mathbf{In}\ e_2]pk= \\ E[e_1]p\{\lambda e.E[e_2](p[e\ In\ D/[i]])k\}. \tag{5.1.11}$$

5.1 Semantic Transformations

5.1.1 Splitting Continuations

Expression Continuations: An expression continuation abstracts the meaning of the rest of the program which is expecting a particular value. As we have seen, values are kept in destinations, either registers or positions in an activation record. From a code generation point of view an expression continuation involves two activities, one to load a value into a destination, the other to continue with the appropriate continuation. It seems natural then, to split every occurrence of such an expression into its two components: a destination and a command continuation. This splitting activity will apply also to environment continuations (without a load), like the one required for GEDANKEN.

Definition: An 'internal' domain of interest KON is required and defined as follows:

$$
\begin{array}{llcl}
\text{for } i=1\ldots n & & & \\
 & \text{let } A_i & = & \text{any domain} \\
 & \text{let } D_i & = & \text{any domain} \\
 & \text{let } K_i & = & [A_i \rightarrow COD] \\
 & \text{let } B_1 & = & [K_1 \rightarrow COD] \\
 & B_2 & = & [[D_1 \times K_2] \rightarrow COD] \\
 & \ldots & = & \ldots \\
 & B_n & = & [[D_1 \times D_2 \times \ldots D_{n-1} \times K_n] \rightarrow COD] \\
 & & & \\
 & KON & = & \sqcup K_i
\end{array}
\qquad [D10]
$$

$\sqcup$ indicates domain union. Every summand K_i of KON is a function $[A_i \rightarrow COD]$ which produces code and appears as the last parameter of a function B_i that also produces code. To be general, we should not restrict this definition to 'the last parameter', but all our examples are written in such a way that an expression continuation is the last function in the sequence of curried applications. So we simplify the definition of rules by imposing a 'style'

Snapshot 5.2: Flow Diagrams with Jumps. Syntactic Transformations

```
let C(node, p, c) be switchon type^node into            by R1.1, R3.1    (5.2.1)
{ case [Dummy]:
    c; endcase                                           by R1.1         (5.2.2)

  case [If e Then c1 Else c2]:
    E([e], p, λe.e?T→e→C([c1], p, c),C([c2], p, c),Wrong); endcase
                                      by R1.1, R1.4, R3.2/3 times       (5.2.3)

  case [c1;c2]:
    C([c1], p, C([c2], p, c)); endcase            by R1.1, R3.2/twice    (5.2.4)

  case [While e Do c1]:
    Fix(λc'.E([e], p, λe.e?T→e→C([c1], p, c'),c,Wrong)); endcase
                                      by R1.1, R1.4, R3.2/3 times       (5.2.5)

  case [Let i=e In c1]:
    E([e], p, λe.C([c1], p([e In D/[i]]), c)); endcase
                                             by R1.1, R3.2/3 times       (5.2.6)

  case [Goto e]:
    E([e], p, λe.e?COD→e|COD,Wrong); endcase           by R1.1, R3.2     (5.2.7)
}

let E(node, p, k) be switchon type^node into              by R1.1, R3.1  (5.2.8)
{ case [i]:
    k(p([i])); endcase                              by R1.1, R3.2/twice  (5.2.9)

  case [If e1 Then e2 Else e3]:
    E([e1], p, λe.e?T→e→E([e2], p, k),E([e3], p, k),Wrong); endcase
                                      by R1.1, R1.4, R3.2/3 times      (5.2.10)

  case [Let i=e1 In e2]:
    E([e1], p, λe.E([e2], p([e In D/[i]]), k)); endcase
                                             by R1.1, R3.2/3 times      (5.2.11)
}
```

of utilising expression continuations. For example, in Snapshot 5.1 we can see that K_i domains are: K=[E→C], W=[K→C] and G=[C→C] because C=COD after the state analysis. Of these only K appears in any B_i domain, i.e: the domain [Exp→U→W]=[Exp→U→K→C] which, after the Syntactic Transformations, is (the data type): [[Exp x U x K]→COD]. So, in this example, KON=K, the domain of expression continuations. In GEDANKEN, KON=[K + X], because in that language there are both expression and environment continuations.

An alternative way of defining KON, instead of internally defining it, is to allow the user to supply the required information. KON would then be defined as a 'domain of interest' in the semantic specification.

To analyse expression continuations in KON we need to consider: applications, abstractions as parameters and variables as parameters:

Applications: We must explicitly divide an applied occurrence of an expression continuation. On the one hand to prepare a value for the destination or environment analysis, and on the other, to prepare a reference for the continuation analysis:

when e_0:KON $\qquad e_0(e_1) \quad \Rightarrow \quad e_1\ ;\ e_0$ **In** COD $\qquad$ [R4.1]

This way of splitting continuations requires an extension to R5.7 which was defined in section 3.4.1. Now, there are many more cases that require a load operation; here is the final version of that rule:

$$\left.\begin{array}{c} \{C_0;\ e;\ C_1\} \\ \text{or} \\ e_0 \rightarrow e,\ e_2 \\ \text{or} \\ e_0 \rightarrow e_1,\ e \end{array}\right| \begin{array}{c} \Rightarrow \\ \\ \Rightarrow \\ \\ \Rightarrow \end{array} \left|\begin{array}{c} \{C_0;\ C;\ C_1\} \\ \text{or} \\ e_0 \rightarrow C,\ e_2 \\ \text{or} \\ e_0 \rightarrow e_1,\ C \end{array}\right. \qquad [R5.7]$$

when NeedsLoad $\qquad$ **where** C = trans.load(DOM(e), e) **In** REG

where NeedsLoad = e:REG and not e:COD and ((e=$\underline{i}$ and not i:ENV) or e=$\underline{E_0!E_1}$ or e=$\underline{\#e_0}$ or e=$\underline{n}$ or e=$\underline{q}$)

Abstractions as parameters: This is the case of a definition of an expression continuation. We associate the bound variable with the destination and the body with a command continuation in the following way:

$$e_0(P,\ \lambda i.e_1) \quad \Rightarrow \quad e_0(P).\mathbf{cont}.(e_1).\mathbf{dest}.(i) \qquad [R4.2]$$

when $(\lambda i.e_1)$:KON $\qquad$ i is bound in e_1

Snapshot 5.3: Flow Diagrams with Jumps. Splitting Continuations

```
let C(node, p).cont.(c) be switchon type^node into           by R4.5   (5.3.1)
{ case [Dummy]:                                                       no change

  case [If e Then c1 Else c2]:
    E([e], p).cont.(e?T>e>C([c1], p).cont.(c),C([c2], p).cont.(c),Wrong
     ).dest.(e); endcase                          by R4.2, R4.6/twice   (5.3.3)

  case [c1;c2]:
    C([c1], p).cont.(C([c2], p).cont.(c)); endcase   by R4.6/twice   (5.3.4)

  case [While e Do c1]:
    Fix(λc'.E([e], p).cont.(e?T>e>C([c1], p).cont.(c'),c,Wrong).dest.(e))
    endcase
                                                       by R4.2, R4.6   (5.3.5)
  case [Let i=e In c1]:
    E([e], p).cont.(C([c1], p([e In D/[i]])).cont.(c)).dest.(e); endcase
                                                       by R4.2, R4.6   (5.3.6)
  case [Goto e]:
    E([e], p).cont.(e?COD>e|COD,Wrong).dest.(e); endcase   by R4.2   (5.3.7)
}

let E(node, p).cont.(k).dest.(?) be switchon type^node into
                                                       by R4.3   (5.3.8)
{ case [i]:
    p([i]); k; endcase                                 by R4.1   (5.3.9)

  case [If e1 Then e2 Else e3]:
    E([e1], p
     ).cont.(e?T>
              e>E([e2], p).cont.(k).dest.(?),E([e3], p).cont.(k).dest.(?),
              Wrong).dest.(e); endcase          by R4.2, R4.4/twice   (5.3.10)

  case [Let i=e1 In e2]:
    E([e1], p).cont.(E([e2], p([e In D/[i]])).cont.(k).dest.(?)).dest.(e)
    endcase
                                                   by R4.2, R4.4   (5.3.11)
}
```

Variables as parameters: When a variable which is an expression continuation is passed as a parameter, we do not have an indication of the destination, so we leave it undefined:

```
                                   __        __
                                     |      |  let v(D).cont.(i In COD)
              let v(D, i) be C       |  =>  |         .dest.(? In d)         [R4.3]
when i:KON=[d>COD]                 __|      |__be C

                                   __        __
when i:KON=[d>COD]     e(P, i)       |  =>  |  e(P).cont.(i In COD)          [R4.4]
                                   __|      |__      .dest.(? In d)
```

These two rules must be regarded as temporary transformations. Their immediate resolution is the analysis of destinations, continuations and environments.

Command Continuations: For command continuations, and by symmetry with expression continuations, we define:

when i:COD **let** v(D, i) **be** C => **let** v(D).**cont.**(i) **be** C [R4.5]

when e_1:COD $e_0(P, e_1)$ => $e_0(P)$**.cont.**(e_1) [R4.6]

Snapshot 5.3 shows the result of splitting expression continuations in this way.

5.1.2 Destination Analysis

The analysis above requires a new set of rules which should be compared with R5.1, R5.3, R5.4 and R5.11 defined in the Destination Analysis of Chapters 3 and 4. These new transformation rules convert **.cont.**() and **.dest.**() constructions. The result of their application is shown in Snapshot 5.4.

let v(D)**.cont.**(P)**.dest.**(?:d) **be** C => **let** v(D)**.cont.**(P)**.dest.**(reg) **be** C [R5.12]
when d$\underline{C}$REG

e**.cont.**(P)**.dest.**(?:d) => e**.cont.**(P)**.dest.**(reg) [R5.13]
when d$\underline{C}$REG

e**.cont.**(P)**.dest.**(i) => e**.cont.**(P)**.dest.**(i) **rename** i=>(i$\underline{=}a_k$)→reg+k, reg [R5.14]
when i:REG

e**.cont.**(P)**.dest.**(?:d) => e**.cont.**(P)**.dest.**(**first**.reg) [R5.15]
when e:TEM and d$\underline{C}$REG

Snapshot 5.4: Flow Diagrams with Jumps. Destination Analysis

```
let C(node, p).cont.(c) be switchon type^node into
{ case [Dummy]:                                                    no change
                                                                   no change
  case [If e Then c1 Else c2]:
   E([e], p
     ).cont.(first.reg?T→first.reg→C([c1], p).cont.(c),C([c2], p).cont.(c),
                 Wrong
     ).dest.(first.reg); endcase                       by R5.2, R5.14   (5.4.3)

  case [c1;c2]:                                                    no change

  case [While e Do c1]:
   Fix(λc'.E([e], p
          ).cont.(first.reg?T→first.reg→C([c1], p).cont.(c'),c,Wrong
           ).dest.(first.reg)); endcase                by R5.2, R5.14   (5.4.5)

  case [Let i=e In c1]:
   E([e], p).cont.(C([c1], p([first.reg In D/[i]])).cont.(c)
     ).dest.(first.reg); endcase                       by R5.2, R5.14   (5.4.6)

  case [Goto e]:
   E([e], p).cont.(first.reg?COD→first.reg|COD,Wrong).dest.(first.reg)
   endcase                                             by R5.2, R5.14   (5.4.7)
}

let E(node, p).cont.(k).dest.(reg) be switchon type^node into
{ case [i]:                                                  by R5.12   (5.4.8)
   p([i]).dest.(reg); k; endcase                             by R5.3    (5.4.9)

  case [If e1 Then e2 Else e3]:
   E([e1], p
    ).cont.(reg?T→
          reg→E([e2], p).cont.(k).dest.(reg),
          E([e3], p).cont.(k).dest.(reg),Wrong
    ).dest.(reg); endcase                    by R5.13/twice, R5.14   (5.4.10)

  case [Let i=e1 In e2]:
   E([e1], p).cont.(E([e2], p([reg In D/[i]])).cont.(k).dest.(reg)
    ).dest.(reg); endcase                             by R5.13, R5.14   (5.4.11)
}
```

5.1.3 Continuation Analysis

Command Continuations: We need to interface the relative position of code fragments to the linear sequence of code instructions. We wish to to do this in such a way that the code generated for one particular construction depends on the immediate context of its appearance. The structure of continuations provides precisely this interface, since a continuation is the meaning of the rest of the program. In terms of code generation, a continuation is a reference to the starting position of the code generated for the rest of the program. That is, it must be a forward reference. Even after splitting, expression continuations preserve this property through the Destination Analysis. For example (5.1.4) specifies that the semantic value of a sequence of commands is a function that depends on the first command, the environment and a continuation value. This continuation is, in turn, a function of the second command, the environment and the continuation of the sequence. The corresponding procedural text, before the Continuation Analysis is:

```
case [c1;c2]:
  C([c1], p).cont.(C([c2], p).cont.(c)); endcase
```

We can see that in order to plant code for a sequence of commands we have to plant code for the first command, in the presence of the given environment and with a reference to the code planted for the second command (the .cont.(e) construction). This can be expressed as:

```
case [c1;c2]:
  C([c1], p).cont.(... code yet to be planted ...); endcase
```

We wish to plant code sequentially, so the reference to the starting position, in the linear sequence of code instructions, of the code associated with the second command is not known until code has been generated for the first one. The reference to the code generated for the second command is then a forward reference. We generalise this idea by stating that every **.cont.**(e) construction is a forward reference, with associated creation and fixing activity:

$$\textbf{when } e_1 \underline{\neq} i \qquad \left\lceil\, e(P).\textbf{cont}.(e_1)A \,\right\rfloor \;\Rightarrow\; \left\lceil\, \begin{array}{l} \{\ \textbf{let}\ \text{continue} = \text{forward(COD)} \\ \quad e(P).\textbf{cont}.(\text{continue})A \\ \quad \text{fix.here(continue)} \\ \quad e_1 \\ \} \end{array} \right\rfloor \qquad [R6.9]$$

In (5.5.4) we show the result of this rule when applied to the sequence of commands shown above. Note that it is up to each command to decide whether or not the forward reference is used. For example in (5.5.7), the corresponding procedure to generate code for a goto will disregard this forward reference; exactly what one would expect, since the original specification (5.1.7) also disregarded the continuation. By contrast in (5.5.5), the corresponding procedure to generate code for a while-loop makes an explicit reference to the forward reference, when breaking out of the loop in: trans.jump.if.false(first.reg, continue).

This mechanism requires a forward reference for every sequence of two commands. However, it is not the case that a label or chain should be created for every command. It is up to the particular interpretation of <u>forward</u> to efficiently implement forward references. In particular they should not be created until some construction really requests it. Also, the optimising transformations will use the fact that code is generated in sequence, to avoid each command planting a jump instruction to the next.

We also rename continuations appearing as formal parameters so that all continuations, now references to the code, are homogeneously named:

```
let v(D).cont.(i)A be C  | => | let v(D).cont.(i)A be C        [R6.10]
                         |    | rename i=>continue
```

Call: The conversion for abstraction with continuations involves a lambda abstraction for the return continuation:

```
                            |    | { let ntry.code = forward(DOM(e1))
                            |    |   let exit.code = forward(COD)
                            |    |   let skip.code = forward(COD)
       e(P0, λi.e1, P1)A    | => |   trans.jump.to(skip.code)        [R6.11]
when i:COD and λi.e1:TEM    |    |   trans.entry(ntry.code, node)
                            |    |   [exit.code/i]e1
                            |    |   trans.exit(exit.code, node)
                            |    |   fix.here(skip.code)
                            |    |   e(P0, ntry.code, P1)A
                            |    | }
```

So that, if the return continuation is taken, a jump will be planted to <u>exit.cont</u> (the code generated by <u>trans.exit</u>).

If an abstraction is used inside an environment definition (or any other [/] construction) then we define:

```
                            |    | e3
     e(P0, e0([e1/e2], P1)A | => | e(P0, e0([ntry.code/e2], P1)A    [R6.12]
when e1:TEM                 |    | where except for the last statement
                            |    | e3 is the same as R6.11
```

5.2 **Optimising Continuations**

5.2.1 **Flow Analysis**

The way that we are handling forward references might result in a jump instruction being planted, whose effect will be to jump to the next instruction in sequence (a jump to program counter plus one). This situation can easily be trapped if we add a flag, to every forward reference passed as a parameter, which indicates whether or not its associated code will be planted immediately after. We have to delay this analysis until after the

Environment Analysis, when the structure of forward references remains constant. As usual, we formalise this observation with conversion rules.

First, a transformation for a I:COD (a chain or label) passed as a parameter at the top level declaration:

```
let v(D).cont.(I)A be              |  | let v(D).cont.(I, jump)A be
switchon E into                    |  | switchon E into
{ case [s]: { C0; C1; C2 }         |=>| { case [s]: { C0; C3; C2 }          [R8.1]
         endcase                   |  |          endcase
  ...                              |  |    ...
}                                  |  | }
```

```
when   C1 = e(P).cont.(I)A or C1=e(P, I)A
       e  is not one of: fix.here, trans.load,
          trans.entry, trans.thunk.entry, trans.exit, trans.thunk.exit,
          trans.jump.if.true or trans.jump.if.false.
where  C3 = e(P).cont.(I, boo)A or (depending on C1) C3=e(P, I, boo)A
       boo= true.jump if C2:COD
       boo= jump      otherwise
```

Next, a transformation for those I:COD which are either free in the procedure where this transformation is applied or locally declared (through a **let** but not as a formal parameter).

```
{ e0                 |  | { e0
  let I0 = E         |  |   let I0 = E
  C0; C1; C2         |=>|   C0; C3; C2                  [R8.2]
  I1(I0)             |  |   I1(I0)
  e1                 |  |   e1
}                    |  | }
```

```
when   C1 = e(P).cont.(I)A or C1 = e(P, I)A
       and (fixed or fixing or global)
       and I1 is one of: fix.here, trans.exit or trans.thunk.exit.
where  C3 = e(P).cont.(I,  boo)A or (depending on C1) C3=e(P, I, boo)A
       fixed = I=I0 and E=here(P)
       fixing= I=I0 and E=forward(P)
       global= I free in the procedure where this transformation is applied.
       boo   = true.jump if (fixing and C2:COD) or fixed or global
       boo   = false.jump otherwise
or
when   C1 is in the context of:
              for I2=e2 to e3 do { fix.here(I0); C0; C1; C2 }
       I0 = e!I2 and I = e!(I2+1)
where  C3 as before
       boo  = true.jump  if C2:COD, boo = false.jump otherwise
```

5.2.2 False Jumps

Now that a flag indicates if a jump is truly required, we can eliminate those jumps where the flag is the constant FALSE. Obviously we can not eliminate those where the flag is a variable:

$$\{ C_1;\ C_2;\ C_3 \} \quad \Rightarrow \quad \{C_1;\ C_3\} \qquad \text{[R8.3]}$$

when C_2= trans.jump.to(i, false.jump)

This resembles the 'constant-folding' mechanism of traditional optimising compilers, applied to the flow of control instead of the analysis of expressions.

5.2.3 Conditional Jumps

The simplest form of a conditional, selecting two continuations according to the value contained in a register: reg ⊁ c, c′, is transformed by R6.2 to: trans.jump.if.false(reg, c′); trans.jump.to(c, E), where E is either <u>jump</u>, <u>not jump</u> or <u>true.jump</u> as a result of R8.1 or R8.2. These two statements will generate a conditional jump, and if E evaluates to true, it will be followed by an unconditional one. For example, the generated DEC-10 code might be:

```
        JUMPE   reg,L1
        JRST    L2
L1:     ...(code for c )...
L2:     ...(code for c')...
```

Which can be improved to:

```
        JUMPNE  reg,L2
        ...(code for c )...
L2:     ...(code for c')...
```

This case does not happen very often in our examples, but when it does, results in a multiplication of the code generated in crucial code areas like the test of an iteration. It seems necessary then to improve this area. The following transformation optimises the form of the CGP to avoid this case:

```
J1(I0, I1)  |    | test E2
J0(I2, E2)  | => | then {J2(I0,I2); J0(I1,E1 }     [R8.4]
           _|    |__or   J1(I0, I1)
```

when $J_0 \equiv$ trans.jump.to
$J_1 \equiv$ trans.jump.if.false or $J_1 \equiv$ trans.jump.if true
$E_2 \equiv$ jump or $E_2 \equiv$ not jump or $E_2 \equiv$ true.jump
where $J_2 \equiv$ reverse of J_1
$E_1 \equiv$ the result of R8.1 or R8.2

Note that if $E_2 \equiv$ true.jump, then the **test** statement above is equivalent to an **if** statement (conditional compilation in BCPL). Examples of the application of this rule can be found in (7.4.2) and (D.2.15).

In Snapshot 5.5 we display the last version of the transformation process for the current example of this chapter.

Snapshot 5.5: Flow Diagrams with Jumps. BCPL

```
let trans.C(node).cont.(continue, jump) be switchon type^node into
                                  by R6.10, R7.5, R8.1, RA.1        (5.5.1)
{ case T..Dummy:
    trans.jump.to(continue, jump); endcase
                                  by R6.1, R6.10, R8.1, RA.1        (5.5.2)

  case N3..ConditionalCom:
    {0 let continuel = forward(D..COD)
       trans.E(p1^node).cont.(continuel, false.jump).dest.(first.reg)
       fix.here(continuel)
       trans.skip.if.in(first.reg, D..T)
       trans.jump.to(Wrong, true.jump)
       { let fcond.code = forward(D..COD)
         trans.jump.if.false(first.reg, fcond.code)
         trans.C(p2^node).cont.(continue, true.jump)
         fix.here(fcond.code)
         trans.C(p3^node).cont.(continue, jump)
    }0; endcase
    by R6.1, R6.2/twice, R6.7, R6.9, R6.10/twice, R7.6/3 times, R8.1/twice
    R8.2/twice, RA.1/4 times, RA.2/6 times
                                                                    (5.5.3)
```

Snapshot 5.5 (continued)

```
case N2..Sequence:
  { let continuel = forward(D..COD)
    trans.C(p1^node).cont.(continuel, false.jump)
    fix.here(continuel)
    trans.C(p2^node).cont.(continue, jump)
  }; endcase
  by R6.9, R6.10, R7.6/twice, R8.1, R8.2, RA.1/3 times, RA.2/3 times
                                                                (5.5.4)

case N2..While:
  { let restart.code = here(D..COD)
    let continuel = forward(D..COD)
    trans.E(p1^node).cont.(continuel, false.jump).dest.(first.reg)
    fix.here(continuel)
    trans.skip.if.in(first.reg, D..T)
    trans.jump.to(Wrong, true.jump)
    trans.jump.if.false(first.reg, continue)
    trans.C(p2^node).cont.(restart.code, true.jump)
  }; endcase
  by R6.1/twice, R6.2/twice, R6.3, R6.7, R6.9, R6.10, R7.6/twice
  R8.2/3 times, RA.1/3 times, RA.2/5 times                      (5.5.5)

case N3..DefinitionByDenotationCom:
  {0 let continuel = forward(D..COD)
     trans.E(p2^node).cont.(continuel, false.jump).dest.(first.reg)
     fix.here(continuel)
     { let old.env = this.env
       declare(domain.of(first.reg), first.reg, p1^node)
       trans.C(p3^node).cont.(continue, jump)
       reset(old.env)
  }0; endcase
  by R6.9, R6.10, R7.1, R7.2, R7.6, R8.1, R8.2, RA.1/4 times
  RA.2/3 times                                                  (5.5.6)

case N1..Goto:
  { let continuel = forward(D..COD)
    trans.E(p1^node).cont.(continuel, false.jump).dest.(first.reg)
    fix.here(continuel)
    trans.skip.if.in(first.reg, D..COD)
    trans.jump.to(Wrong, true.jump)
    trans.jump.to(first.reg, true.jump)
  }; endcase
  by R6.1/twice, R6.2, R6.7, R6.9, R7.6, R8.2/3 times, RA.1/twice
  RA.2/3 times                                                  (5.5.7)
}

let trans.E(node).cont.(continue, jump).dest.(reg) be
switchon type^node into                by R6.10, R7.5, R8.1, RA.1   (5.5.8)
{ case T..Ident:
    look.up(node).dest.(reg); trans.jump.to(continue, jump); endcase
                               by R6.1, R6.10, R7.4, R8.1, RA.1/twice   (5.5.9)
```

Snapshot 5.5 (continued)

```
case N3..ConditionalExp:
  {0 let continuel = forward(D..COD)
     trans.E(p1^node).cont.(continuel, false.jump).dest.(reg)
     fix.here(continuel)
     trans.skip.if.in(reg, D..T)
     trans.jump.to(Wrong, true.jump)
     { let fcond.code = forward(D..COD)
       trans.jump.if.false(reg, fcond.code)
       trans.E(p2^node).cont.(continue, true.jump).dest.(reg)
       fix.here(fcond.code)
       trans.E(p3^node).cont.(continue, jump).dest.(reg)
  }0; endcase
  by R6.1, R6.2/twice, R6.7, R6.9, R6.10/twice, R7.6/3 times, R8.1/twice
  R8.2/twice, RA.1/4 times, RA.2/6 times
                                                                  (5.5.10)
case N3..DefinitionByDenotationExp:
  {0 let continuel = forward(D..COD)
     trans.E(p2^node).cont.(continuel, false.jump).dest.(reg)
     fix.here(continuel)
     { let old.env = this.env
       declare(domain.of(reg), reg, p1^node)
       trans.E(p3^node).cont.(continue, jump).dest.(reg)
       reset(old.env)
  }0; endcase
  by R6.9, R6.10, R7.1, R7.2, R7.6, R8.1, R8.2, RA.1/4 times
  RA.2/3 times
}                                                                 (5.5.11)
```

5.3 Ellipsis

When adding the missing equation for blocks, we are presented with a choice of semantic 'styles' to define the value associated with each label within the block. We could express the equation with functions that collect label names and their associated continuation values in lists which are used to update the environment. This is done in [Mos74] and [MaS76]. Alternatively, we can use an ellipsis (...) which makes names and values 'visible' without the need of any extra collection, as in [Sto77]. Both methods define the same semantic value, Each one, however, corresponds to a different code generation strategy. Consider the two different styles of Snapshot 5.6. The first equation (5.6.1), in which label values are defined with Fix, corresponds to a one pass CGP. While the second (5.6.2), in which label

Snapshot 5.6: Blocks: two different 'styles'. Original Specification

Semantic Equation for Blocks (Using Ellipsis)

$$\begin{aligned}&C[\mathbf{Begin}\ i_1{:}c_1;\ \ldots i_2{:}c_2\ \mathbf{End}]pc=\\&\quad Fix(\lambda\langle c',\ldots,c''\rangle.\\&\qquad \{\langle C[c_1]p'c'',\ldots,C[c_2]p'c\rangle\\&\quad Where\ p'=p[c'/[i_1]]\ldots[c''/[i_2]]\})\downarrow 1.\end{aligned} \tag{5.6.1}$$

Semantic Equation for Blocks (Using Lists)

$$\begin{aligned}&C[\mathbf{Begin}\ c_1\ \mathbf{End}]pc=\\&\quad C[c_1]p''c\\&\quad Where\ p''=Fix(\lambda p'.p[L[c_1]p'c/I[c_1]]).\end{aligned} \tag{5.6.2}$$

$$\begin{aligned}&C[c_1;c_2]pc=\\&\quad C[c_1]p\{C[c_2]pc\}.\end{aligned} \tag{5.6.3}$$

$$\begin{aligned}&C[i{:}c_1]pc=\\&\quad C[c_1]pc.\end{aligned} \tag{5.6.4}$$

$$I{:}[Com \rightarrow Ide^*]. \tag{5.6.5}$$

$$\begin{aligned}&I[c_1;c_2]=\\&\quad I[c_1]\%I[c_2].\end{aligned} \tag{5.6.6}$$

$$\begin{aligned}&I[i{:}c_1]=\\&\quad \langle[i]\rangle.\end{aligned} \tag{5.6.7}$$

$$L{:}[Com \rightarrow U \rightarrow C \rightarrow C^*]. \tag{5.6.8}$$

$$\begin{aligned}&L[c_1;c_2]pc=\\&\quad L[c_1]p\{C[c_2]pc\}\%L[c_2]pc.\end{aligned} \tag{5.6.9}$$

$$\begin{aligned}&L[i{:}c_1]pc=\\&\quad \langle C[c_1]pc\rangle.\end{aligned} \tag{5.6.10}$$

---

values are collected in lists, corresponds to a two pass CGP, since the associated code generation procedures will pass the text twice for each command, one by $C[c_1]p''c$, the other indirectly through $L[c_1]p'c$. This happens according to our particular interpretation of <u>Fix</u> and our general treatment of continuations.

This is a clear example of the way, that one can direct the structure of the CGP by expressing the concrete semantics in different ways. We have chosen

Snapshot 5.7: Flow Diagrams with Ellipsis. Original Specification

Extensions to Snapshot 5.1

Syntax

c ::= **Begin** l_1; ...l_2 **End** | **Switchon** e **Into** c_1; ...c_2 | **Case** n:c_1 |
  Default:c_1 | **Endcase**
l ::= i:c
e ::= n | ...

Syntactic Domains

l:Lco.	labeled com.
n:Nml.	numerals

Semantics

e:E=[T + C + N].	expression values
N.	integers
M=[INT + { DefM }].	case constants
p:U=[[Ide → D] x C].	environments
pNDC==p↓2.	endcase selector
N:[Nml → N].	
I:[Nml → INT].	
Switch:[N → C* → M* → C → C].	

Switch=
  Lam n<c_1,...,c_k><m_1,...,m_k>c.
  for i=1 to k if m_i:INT and n=m_i|INT then c_i
  for i=1 to k if m_i=DefM then c_i
  otherwise c.

Semantic Domain of 'Interest' Isomorphic with N

INT. compile-time integers

E[n]pk=
  k(N[n]). (5.7.1)

E[n]pk=
  k(N[n]). (5.7.2)

M:[Com → M]. (5.7.3)

M[**Case** n:c_1]=
  I[n]. (5.7.4)

M[**Default**:c_1]=
  DefM. (5.7.5)

C[**Case** n:c_1]pc=
  C[c_1]pc. (5.7.6)

C[**Default**:c_1]pc=
  C[c_1]pc. (5.7.7)

C[**Endcase**]pc=
  pNDC. (5.7.8)

Snapshot 5.7 (continued)

```
C[Switchon e Into c1; ...c2]pc=
  E[e]p{λe.e?N>Switch(e|N)<C[c1]p'c,...,C[c2]p'c><M[c1],...,M[c2]>c,Wrong}
  Where p'=p[c/NDC].                                                  (5.7.9)

C[Begin l1; ...l2 End]pc=
  Fix(λ<c',...,c''>.
       {<C(2↓[l1])p'c'',...,C(2↓[l2])p'c>
  Where p'=p[c' In D/1↓[l1]]...[c'' In D/1↓[l2]]})↓1.                 (5.7.10)
```

the ellipsis method for no special reason and not because a two pass CGP is uninteresting. Further research could easily incorporate the second method.

We will also extend the language used in [Sto77] by adding a Switchon statement. This feature also is conveniently defined using an ellipsis making a nice companion to the block. Snapshot 5.7 shows the required modifications and extensions to the original specification of Snapshot 5.1.

In this specification, we are using the operator ↓ and a short hand for expressions inside ellipsis which have different interpretations depending on context:

Semantic Context
$.\downarrow.:[[[D_1 \times \ldots \times D_n] \times N\ ] \rightarrow D_k]$
$d=\langle D_1,\ldots, d_k,\ldots, d_n\rangle:[D_1 \times \ldots \times D_k \times \ldots \times D_n]$
$k:N$ and $1 \le k \le n$

$$d\downarrow k \text{ and also } k\downarrow d \quad = \quad d_k \tag{D11}$$

Syntactic Context
$.\downarrow.:[S \times N] \rightarrow S_k]$
$[s]=[s_1 \ldots s_n]:S$
$k:N$ and $1 \le k \le n$

$$[s]\downarrow k \text{ and also } k\downarrow[s] \quad = \quad [s_k] \tag{D12}$$

So that ↓ is used to extract individual components of tuples or node-offspring.

In a syntactic context: $[\ s_1 \dots s_2\ ]$ is short for: $[\ s_1 \dots s_k \dots s_n\ ]$.

And in a semantic context:

$\langle f(i', i''),\dots,g(i'')\rangle$ is short for: $\langle f(i_1, i_2),\dots,f(i_k, i_{k+1}),\dots,g(i_n)\rangle$

where $1 <= k < n$.

So (5.7.10) in effects stands for:

$$
\begin{array}{l}
C[\mathbf{Begin}\ i_1{:}c_1;\ \dots i_k{:}c_k;\ \dots i_n{:}c_n\ \mathbf{End}]\rho c= \\
\quad Fix(\lambda\langle c_1,\dots,c_k,\dots,c_n\rangle. \\
\qquad \{\langle C([c_1])\rho' c_2,\dots,C([c_k])\rho' c_{k+1},\dots,C([c_n])\rho' c\rangle \\
\quad \text{Where } \rho'=\rho[c_1 \text{ In } D/[i_1]]\dots[c_k \text{ In } D/[i_k]]\dots[c_n \text{ In } D/[i_n]] \\
\qquad \})\downarrow 1.
\end{array}
$$

The reason for this is that under certain constraints imposed by this treatment, the use of ellipsis in our transformations is much more simplified. An example of the kind of restriction imposed is reflected in the semantic specification of the **switchon** statement (5.7.9). It is not the same as the corresponding statement in BCPL. We have to avoid control dropping into the next **case** statement, because this requires an explicit c_{k+1}. So our example is rather like a PASCAL **case** than a BCPL **switchon** statement. In short, in WFF_t, sub-indices cannot be expressions, and we are left only with decorations (quotes or digits) which we have to assume refer to the next item in the ellipsis.

5.3.1 Syntactic Transformations

Having presented this restricted use of ellipsis, we move now to consider the required transformations. An ellipsis in a syntactic element denotes a n-ary node of the parse tree. The first requirement is to open this node, to be able to see every sub-component:

```
                   |  case [s1 ... sn]:
case               |  { let n = open.node(node)
[s1 ... sn]:   =>  |    e
e                  |    close.node(n)
                   |  } rename [s1]=>select(n, inx)
                   |           [sn]=>select(n, length(n))
                   |           n=>node.vec
```

This rule produces the desired effect, we can see every element of a node by means of the select primitive. However, BCPL provides a quick way of accessing elements of a vector with the '!' operator (defined in Appendix C). Also, later rules require a general mechanism to release other sort of data structures, hence instead of close.node we shall use a primitive named freevec (which happens to be a suitable BCPL library routine). Accordingly, we rewrite this rule as:

```
                   |  case [s1 ... sn]:
case               |  { let n = open.node(node)
[s1 ... sn]:   =>  |    e                                   [R3.7]
e                  |    freevec(n)
                   |  } rename [s1]=>n!inx, [sn]=>n!!n,
                   |           n=>node.vec
```

open.node is a primitive procedure which depends on the structure of the n-ary node. It stores each individual offspring in a different cell of a vector, returning a pointer to that vector (with the size stored in its zero word). Each sub-component can then be accessed via node.vec!inx, and the size with !node.vec (equivalent to node.vec!0). freevec releases the space occupied by the vector. inx is used in relation to the transformations below. A pre-condition imposed by this transformation is that every ellipsis, used inside the expression 'e' above, has to have the same length as the ellipsis of its parent node. In Snapshot 5.8 we show the result of this transformation, together with those corresponding to the Normalisation and Syntactic Transformations.

Snapshot 5.8: Flow Diagrams with Ellipsis. Syntactic Transformations

```
let C(node, p, c) be switchon type^node into                    by R1.1, R3.1  (5.8.1)
{ case [Case n:c1]:
    C([c1], p, c); endcase                                      by R1.1, R3.2  (5.8.2)

  case [Default:c1]:
    C([c1], p, c); endcase                                      by R1.1, R3.2  (5.8.3)

  case [Endcase]:
    p(NDC); endcase                                             by R1.1, R3.2  (5.8.4)

  case [Switchon e Into c1; ...c2]:
    { let node.vec2 = open.node(p2^node)
      { let p' = p([c/NDC])
        E([e], p, λe.e?N→
                     Switch(e|N, <C(node.vec2!inx, p', c),...,
                                  C(node.vec2!!node.vec2, p', c)
                                 >, <M(node.vec2!inx),...,
                                    M(node.vec2!!node.vec2)
                                   >, c),Wrong)
      }
      freevec(node.vec2)
    }; endcase          by R1.1, R1.3, R3.2/7 times, R3.3, R3.7  (5.8.5)

  case [Begin l1; ...l2 End]:
    { let node.vec = open.node(node)
      Fix(λ<c',...,c''>.
          { let p' = p([c' In D/1↓node.vec!inx],...,
                       [c'' In D/1↓node.vec!!node.vec])
            <C(2↓node.vec!inx, p', c''),...,
             C(2↓node.vec!!node.vec, p', c)
            >
          })↓1
      freevec(node.vec)
    }; endcase          by R1.1, R1.3, R3.2/4 times, R3.3, R3.7  (5.8.6)
}

let E(node, p, k) be switchon type^node into                    by R1.1, R3.1  (5.8.7)
{ case ...

  case [n]:
    k(N([n])); endcase                                    by R1.1, R3.2/twice  (5.8.8)
}

let M(node) be switchon type^node into                          by R1.1, R3.1  (5.8.9)
{ case [Case n:c1]:
    I([n]); endcase                                             by R1.1, R3.2  (5.8.10)

  case [Default:c1]:
    DefM; endcase                                                      by R1.1  (5.8.11)
}
```

Note that in WFF_m '↓' has lower precedence than '!'.

It might be argued that this method it too biased to the systems programming language BCPL, which we are using as target. Or that we are imposing, in this correspondence, one particular strategy of our own, which might not be general enough. What we are in effect showing is a mechanism, one way to achieve the analysis necessary to transform semantic equations into a particular form suitable for an algorithmic interpretation.

5.3.2 Continuation Analysis

We start by looking at those tuples containing ellipsis. Since in our examples, this kind of tuple is used only to define either procedures or functions (in TEM), code structures (in COD) or constants,. We will assume that they are not used for anything else.

```
                <e_1,...,e_n>      =>    { C_1; C_2; C_3; C_4 }            [R6.13]
where   C_1 = let c = E
        C_2 = for inx=1 to s-1 do C_f
        C_3 = unless s=0 do C_u
        C_4 = freevec(c)
        e_i :D for i=1 to n

              |  { e_1; fix.with(c!inx,r) }  if DCTEM
        C_f = |  { fix.here(c!inx); e_1   }  if DCCOD
              |  c!inx := e_1                otherwise

              |  { e_n; fix.with(c!s,r) }    if DCTEM
        C_u = |  { fix.here(c!s); e_n   }    if DCCOD
              |  c!s   := e_n                otherwise
rename  s=>!node.vec
        c=>DC[COD+TEM]>code.vec, cons.vec
        E=>DC[COD+TEM]>forward.vec(s, D), newvec(s)
        r=destination of e_i
```

<u>forward.vec</u> gets as many forward references as indicated by its first parameter, storing them in a vector and returning a pointer to it. <u>newvec</u> gets a vector without initialising it. <u>fix.with</u> has to plant code to move the closure value kept in the destination indicated by its second parameter to the object described in its first parameter.

Next, two transformations for lists (indicated with a '*') appearing as parameters of a procedure call. The first one for lists which do not produce code. The second one, for those that do, requires a skip over the code that will be generated. In both cases we transform, as above, moving C_4 after the call, and replacing the parameter by the reference to the vector:

$$e_0(P_0,\ e_1,\ P_1)A \quad \Rightarrow \quad \{\ C_1;\ C_2;\ C_3;\ C_5;\ C_4\} \qquad \text{[R6.14]}$$

when e_1:D* and not $\overline{\text{DCCOD}}$
where C_1, C_2, C_3 and C_4 are inherited from the transformation of e_1 as a result of R6.13
$C_5 = e_0(P_0$, cons.vec, $P_1)A$

$$e_0(P_0,\ e_1,\ P_1)A \quad \Rightarrow \quad \left|\begin{array}{l} \{\ C_1 \\ \text{let s = forward(COD)} \\ \text{trans.jump.to(s)} \\ C_2;\ C_3 \\ \text{fix.here(s)} \\ e_0(P_0,\ \text{code.vec},\ P_1)A \\ C_4 \end{array}\right. \qquad \text{[R6.15]}$$

when e_1:COD*
where C_1, C_2, C_3 and C_4 same as R6.14
rename s=>skip.code

For the minimal fix point finder of a code list we require:

$$\text{Fix}(\lambda\langle i_1,\ldots,i_n\rangle.\{\ C_0;e_1\ \}) \quad \Rightarrow \quad \{\ C_1;\ C_0;\ C_2;\ C_3;\ C_5;\ C_4\ \} \qquad \text{[R6.16]}$$

when e_1:COD* and $e_1 = \langle e_1,\ldots,e_n\rangle$
where $C_5 = e(P_0$, code.vec, $P_1)A$
C_1, C_2, C_3 and C_4 same as R6.14
rename i_1=>code.vec!inx, i_n=>code.vec!!node.vec

And finally, selecting a particular element of a list can in certain cases be ignored:

when e:COD* $\quad n \downarrow e$ or $e \downarrow n \quad \Rightarrow \quad C\ ;\ e$ $\qquad$ [R6.17]
where C = null if n=1
C = trans.jump.to(code.vec!n) otherwise

5.3.3 Environment Analysis

Multiple declarations using the ellipsis are iteratively treated as follows:

when i:ENV i(P) => { C_1; C_2 } [R7.7]
where P = [e_1/e_2],...,[e_3/e_4]
 C_1= **for** inx=1 **to** s-1 **do** declare(e_2, e_1)
 C_2= **unless** s=0 **do** declare(e_4, e_3)
rename s=>!node.vec

5.3.4 Optimising Continuations

We also optimise **for-unless** constructions. When the last expression in the ellipsis is appropriately related to those in the iteration:

for I=e_0 **to** e_1-1
do C_1
unless e_1=0 => **for** I=e_0 **to** e_1 **do** C_1 [R8.5]
do C_2
when C_1= [I/e_1]C_2

5.3.5 BCPL

Firstly $\downarrow$, which as explained above is used to extract individual components of tuples or node-offspring: In BCPL this is done via 'selectors', the following transformation makes up a selector name by juxtaposition of the character 'p' and the integer 'n':

n$\downarrow$e or e$\downarrow$n => pn^e **where** pn is a 'selector' [RA.6]

And secondly, instead of BCPL procedures which generate code, when a semantic valuator is associated with a constant value (like in **M**), we make functions with the BCPL **valof** and **resultis** constructions:

let v(P) **be** C => **let** v(P)=**valof** C [RA.7]
when not v(P):COD

and for every **case** inside C above:

case I: E; **endcase** => **case** I: **resultis** E [RA.8]

These rules describe basically what we require, in effect there are other cases to consider, like when E in RA.8 is in fact a block, we leave this unspecified since it is not required for our example.

Snapshot 5.9: Flow Diagrams with Ellipsis. BCPL

```
let trans.C(node).cont.(continue, jump) be switchon type^node into
                                  by R4.5, R6.10, R7.5, R8.1, RA.1     (5.9.1)
{ case N2..Case:
    trans.C(p2^node).cont.(continue, jump); endcase
                        by R4.6, R6.10, R7.6, R8.1, RA.1/twice, RA.2   (5.9.2)

  case N1..Default:
    trans.C(p1^node).cont.(continue, jump); endcase
                        by R4.6, R6.10, R7.6, R8.1, RA.1/twice, RA.2   (5.9.3)

  case T..Endcase:
    trans.jump.to(look.up(NDC), true.jump); endcase
                                       by R6.1, R7.4, R8.2, RA.1       (5.9.4)

  case N2..Switchon:
    { let node.vec2 = open.node(p2^node)
      { let old.env = this.env
        declare(D..COD, continue, NDC)
        {0 let continue1 = forward(D..COD)
           trans.E(p1^node).cont.(continue1, false.jump).dest.(first.reg)
           fix.here(continue1)
           trans.skip.if.in(first.reg, D..N)
           trans.jump.to(Wrong, true.jump)
           { let cons.vec2 = newvec(!node.vec2)
             let code.vec2 = forward.vec(!node.vec2, D..COD)
             let skip.code = forward(D..COD)
             trans.jump.to(skip.code, true.jump)
             for inx=1 to !node.vec2
             do { fix.here(code.vec2!inx)
                  trans.C(node.vec2!inx).cont.(continue, true.jump)
                }
             fix.here(skip.code)
             for inx=1 to !node.vec2
             do cons.vec2!inx := trans.M(node.vec2!inx)
             Switch(first.reg, code.vec2, cons.vec2).cont.(continue, jump)
             freevec(code.vec2)
             freevec(cons.vec2)
        }0
        reset(old.env)
      }
      freevec(node.vec2)
    }; endcase
  by R4.2, R4.6/3 times, R5.2, R5.14, R6.1, R6.2, R6.7, R6.9
  R6.10/4 times, R6.13/twice, R6.14, R6.15, R7.2, R7.3, R7.6/3 times
  R8.1/3 times, R8.2/3 times, R8.5/twice, RA.1/twice, RA.2/8 times
                                                                        (5.9.5)
```

Snapshot 5.9 (continued)

```
case NX..Block:
  { let node.vec = open.node(node)
    { let old.env = this.env
      let code.vec = forward.vec(!node.vec, D..COD)
      for inx=1 to !node.vec
      do declare(D..COD, code.vec!inx, p1^node.vec!inx)
      for inx=1 to !node.vec-1
      do { fix.here(code.vec!inx)
           trans.C(p2^node.vec!inx).cont.(code.vec!(inx+1), false.jump)
         }
      unless !node.vec=0
      do { fix.here(code.vec!!node.vec)
           trans.C(p2^node.vec!!node.vec).cont.(continue, jump)
         }
      freevec(code.vec)
      reset(old.env)
    }
    freevec(node.vec)
  }; endcase
  by R4.6/twice, R6.10, R6.13, R6.16, R6.17, R7.2, R7.3, R7.6/twice, R7.7
  R8.1, R8.2, R8.5, RA.1, RA.2/4 times, RA.6/3 times                  (5.9.6)
}

let trans.E(node).cont.(continue, jump).dest.(reg) be
switchon type^node into      by R4.3, R5.12, R6.10, R7.5, R8.1, RA.1    (5.9.7)
{ case ...

  case T..Numeral:
    trans.N(node).dest.(reg); trans.jump.to(continue, jump); endcase
                 by R4.1, R5.3, R6.1, R6.10, R8.1, RA.1/twice, RA.2     (5.9.8)
}

let trans.M(node) = valof switchon type^node into      by RA.1, RA.7    (5.9.9)
{ case N2..Case:
    resultis trans.I(p1^node)              by RA.1/twice, RA.2, RA.8   (5.9.10)

  case N1..Default:
    resultis DefM                                      by RA.1, RA.8   (5.9.11)
}
```

5.4 Further Developments: GEDANKEN

The rules described in this section are only required for GEDANKEN. All references below to (E.1.x) and (E.2.y) refer respectively to the Original Specification and final CGP in BCPL as shown in Appendix E.

One of our WFF_s forms of ellipsis allows the definition of 'iterative' lambda expressions of the form:

$$\lambda p. \ldots e$$

In GEDANKEN, this form of ellipsis is used in the sequence constructions of expressions (E.1.10) and parameters (E.1.20), as reproduced in Snapshot 5.10:

Snapshot 5.10: GEDANKEN: Sequences. Original Specification

```
E[e1, ... ,e2]pk=
  E[e1]p{λe. ...E[e2]p{λe'.k{Seq<e,...,e'>}}}.                    (5.10.1)

P[p1, ... ,p2]pex=
  Ccoerce e
  {λe.e?F>U[p1]p{e|F}1{λp'. ...U[p2]p'{e|F}(#[p1, ... ,p2])x},Cerror}
  .
                                                                 (5.10.2)
```

Such expressions would appear at the level of the Syntactic Transformations as a parameter of a call of the form:

$$e_0(P, \lambda i. \ldots e)$$

This expression in effect is short for:

$$e_0(P, \lambda i_1. \; e_0(P, \lambda i_2. \ldots e_0(P, \lambda i_k. \ldots e_0(P, \lambda i_n.e) \ldots))$$

where $1 <= k <= n$

n is the size of the node where the ellipsis occurred. So Snapshot 5.11 in effect stands for Snapshot 5.12.

Snapshot 5.11: GEDANKEN: Sequences. Syntactic Transformations

```
case [e1, ... ,e2]:
  { let node.vec = open.node(node)
    E(node.vec!inx, p,
      λe. ...E(node.vec!!node.vec, p, λe'.k(Seq(<e,...,e'>))))
    freevec(node.vec)
  }; endcase                              by R1.1, R3.2/4 times, R3.7  (5.11.1)

case [p1, ... ,p2]:
  { let node.vec = open.node(node)
    Ccoerce(e,
            λe.e?F→
               U(node.vec!inx, p, e|F, 1,
                  λp'. ...U(node.vec!!node.vec, p', e|F,
                              #[p1, ... ,p2], x)),Cerror)
    freevec(node.vec)
  }; endcase                              by R1.1, R3.2/3 times, R3.7  (5.11.2)
```

Snapshot 5.12: GEDANKEN: Sequences (in effect). Syntactic Transformations

```
case [e1, e2,...,ek,...,en]:
  { let node.vec = open.node(node)
    E(node.vec!1, p, λe1.
     E(node.vec!2, p, λe2.
      ...
       E(node.vec!k, p, λek.
        ...
         E(node.vec!n, p, λen.
          k(Seq(<e1,e2,...,ek,...,en>)))...)...))
    freevec(node.vec)
  }; endcase

case [p1, p2,...,pk,...,pn]:
  { let node.vec = open.node(node)
    Ccoerce(e, λe.e?F→ U(node.vec!1, p, e|F, 1, λp'.
                        U(node.vec!2, p, e|F, 2, λp'.
                         ...
                          U(node.vec!k, p, e|F, k, λp'.
                           ...
                            U(node.vec!n, p', e|F, n, x)...)...)),
                      Cerror)
    freevec(node.vec)
  }; endcase
```

Snapshot 5.13: GEDANKEN: Sequences. Splitting Continuations

```
case [e1, ... ,e2]:
  { let node.vec = open.node(node)
    for inx=1 to !node.vec-1
    do E(node.vec!inx, p).cont.(...).dest.(e)
    unless !node.vec=0
    do E(node.vec!!node.vec, p).cont.(Seq(<e,...,e'>); k).dest.(e')
    freevec(node.vec)
  }; endcase                                by R4.1, R4.2, R4.6, R4.7  (5.13.1)

case [p1, ... ,p2]:
  { let node.vec = open.node(node)
    Ccoerce(e
           ).cont.(e?F>
                    for inx=1 to !node.vec-1
                    do U(node.vec!inx, p, e|F, inx).cont.(...).dest.(p')
                    unless !node.vec=0
                    do U(node.vec!!node.vec, p', e|F, #[p1, ... ,p2]
                         ).cont.(x).dest.(?),Cerror).dest.(e)
    freevec(node.vec)
  }; endcase                                by R4.2, R4.4, R4.6, R4.7  (5.13.2)
```

5.4.1 Splitting Continuations

Such 'syntactically sugared' form of ellipsis involves in these two cases, an iterative definition of an expression continuation (5.11.1) and of an environment continuation (5.11.2) which are transformed as follows:

```
                                | for inx=1 to !node.vec-1
when e0:KON  e0(P, λi. ...e) => | do e0(P).cont.(...).dest.(i)      [R4.7]
                                | unless !node.vec=0 do e
```

The .cont.(...) construction, is a temporary expression which indicates a reference to the position, in the linear sequence of code instructions, of the code generated by the next iteration. This is a forward reference which is analysed below in the Continuation Analysis. The result of this and all other Splitting Continuations rules, when applied to Snapshot 5.11 results in Snapshot 5.13.

5.4.2 Destination Analysis

Iterative Creation: In (5.13.1), the iteration is producing values which need to be preserved until the next statement outside the iteration is reached. This requires a dump operation, expressed as:

```
                                 __       __
                                   |     |  { let old.env = this.env
                                   |     |    let old.off = this.off
      for I=1 to E0 do C1          |  => |    for I=1 to E0 do C4          [R5.16]
      unless E1 do C2              |     |    unless E1 do C5
                                   |     |    reset(old.env)
                                 __|     |__}
when    C1 = e1(P1).cont.(...).dest.(I1)
        C2 = { C7; e2(P2).cont.(C3).dest.(I2); C8 }
        C3 = e3(<I1,...,I2>)
where   C4 = e1(P1).cont.(trans.dump(I1); ...).dest.(I1)
        C5 = { C7; e2(P2).cont.(C6).dest.(I2); C8 }
        C6 = e3(old.off).dest.(I1)
        any C7, C8
```

<u>this.off</u> is a global primitive variable containing a description of the current workspace area. It is used to remember the beginning of the 'dumping' area in <u>old.off</u>. In such a way that e_3, instead of expecting a tuple as a parameter, expects <u>old.off</u> which, together with the now different description associated with <u>this.off</u>, define the boundaries of the area where the tuple values have being dumped.

Iterative Conservation: The rule above, applies when registered values are defined in each iteration. Those that need to be preserved during the whole iteration require a dump operation before the iteration starts, and a load at each step.

```
                                 __       __
                                   |     |  { let dmp.loc = trans.dump(I1)
      for I=1 to E do C1           |  => |    for I=1 to E do C3           [R5.17]
      C2                           |     |    C2
                                 __|     |__}
when    C1 = e1(P1, I1, P2)A
        I1:REG and any C2
where   C3 = { C1; trans.load(DOM(I1), dmp.loc).dest.(I1) }
```

Parameters: When a value in REG is passed as a parameter at a top level procedure declaration, by symmetry, we rename it throughout the body of the procedure:

when i:REG **let** $v(D_0, i, D_1)$ **be** C => **let** $v(D_0, i, D_1)$ **be** C **rename** i=>reg [R5.18]

so that now all REG values are homogeneously named.

Dyadic Operations: In (E.1.9), the semantic equation for a case statement, the relational operator '=' is used to test for a particular run-time registered value. Associated with R6.2 we define a transformation rule for all relational dyadic operators:

$e_0 o e_1$ => trans.skip.if(i.skipXX,e_0,e_1) [R5.19]
when e_0:REG and e_1:REG and o is one of: =, **Eq, Ne, Ls, Le, Gr, Ge**
where XX is respectively one of: EQ, EQ, NE, LT, LE, GT, GE

<u>trans.skip.if</u> generates a skip instruction whose nature is indicated by the primitive constant value i.skipXX. An example of a transformation by R5.19 can be found in (E.2.9).

In Snapshot 5.14, we show the current state of both sequencing constructs.

5.4.3 Continuation Analysis

Conditional Skip: R5.19 requires a further extension to R6.2 similar to that one of section 4.3.2 to deal with run-time type checking. There, the code associated with the boolean part of a conditional was a test and skip instruction. Now R5.19 introduces precisely the same sequence.

Snapshot 5.14: GEDANKEN: Sequences. Destination Analysis

```
case [e1, ... ,e2]:
  { let node.vec = open.node(node)
    { let old.env = this.env
      let old.off = this.off
      for inx=1 to !node.vec-1
      do { E(node.vec!inx, p).cont.(...).dest.(reg); trans.dump(reg) }
      unless !node.vec=0
      do E(node.vec!!node.vec, p
          ).cont.(trans.dump(reg); Seq(old.off).dest.(reg); k
           ).dest.(reg)
      reset(old.env)
    }
    freevec(node.vec)
  }; endcase                                       by R5.14, R5.16  (5.14.1)

case [p1, ... ,p2]:
  { let node.vec = open.node(node)
    Ccoerce(reg
        ).cont.(first.reg?F→
                  { let dmp.loc = trans.dump(first.reg)
                    for inx=1 to !node.vec-1
                    do (U(node.vec!inx, p, first.reg|F, inx).cont.(...)
                          trans.load(F, dmp.loc).dest.(first.reg)
                            ).dest.(p′)
                    unless !node.vec=0
                    do U(node.vec!!node.vec, p′, first.reg|F,
                         #[p1, ... ,p2]).cont.(x).dest.(?)
                  },Cerror
        ).dest.(first.reg)
    freevec(node.vec)
  }; endcase                          by R5.14, R5.15, R5.17, R5.18  (5.14.2)
```

So once more, we extend the **when** and **where** constructions of R6.2 as follows:

```
when    (E0IsDes or E0IsIde or E0IsSkp) and i:REG
where   C3 = E0IsIde → null,
             Reverse and E0IsInt → trans.skip.if.not.in(P),
             Reverse and E0IsDya → trans.skip.if(ReveDya(I), P),
             e0
        C4 = E0IsSkp → trans.jump.to(FalseCo), JumpRut(i, FalseCo)
        E0IsDya = e0=trans.skip.if(I, P)
        E0IsSkp = E0IsInt or E0IsDya
```

Snapshot 5.15: GEDANKEN: Sequence of Parameters. Continuation Analysis

```
case [p1, ... ,p2]:
  { let node.vec = open.node(node)
    {0 let continue2 = forward(COD)
       Ccoerce(reg).cont.(continue2).dest.(first.reg)
       fix.here(continue2)
       trans.skip.if.in(first.reg, F)
       trans.jump.to(Cerror)
       { let dmp.loc = trans.dump(first.reg)
         for inx=1 to !node.vec-1
         do ({ let continue1 = forward(COD)
               U(node.vec!inx, p, first.reg|F, inx).cont.(continue1)
               fix.here(continue1)
             }
            trans.load(F, dmp.loc).dest.(first.reg)).dest.(p')
         unless !node.vec=0
         do U(node.vec!!node.vec, p', first.reg|F, #[p1, ... ,p2]
              ).cont.(continue
              ).dest.(?)
    }0
    freevec(node.vec)
  }; endcase                  by R6.1, R6.2, R6.7, R6.9, R6.10, R6.18  (5.15.1)
```

Iteration: Recall that in R4.7 above we introduced an expression of the form **.cont.**(...) when analysing expressions continuations in KON with ellipsis of the form: λp. ... e. The **.cont.**(...) construction denotes the code planted in the next iteration of the **for** loop produced by R4.7, this is a forward reference, for which we already have enough machinery to transform appropriately:

```
         __                __
           |            | { let continue = forward(COD)
e(P).cont.(...)A  | =>  |   e(P).cont.(continue)A            [R6.18]
           |            |   fix.here(continue)
         __|            |__}
```

The way that this rule affect the procedural text associated with the sequence of expressions is so similar to the way that the sequence of parameters is affected that we proceed by displaying the latter only, as shown in Snapshot 5.15.

5.4.4 Environment Analysis

The process of Splitting Continuations, as described in section 5.1.1, is applied not only to expression continuations in [REG>COD], but also to environment continuations in [ENV>COD]. This particular analysis affects a few cases in the analysis of GEDANKEN. There are two moments to consider: declaration and elimination.

Declaration: In section 4.4, we defined R7.1 and R7.3 to transform the definition of a new environment with a declare-undeclare pair. Now we need a similar rule to transform the equivalent case of a declaration continuation. This rule is used in GEDANKEN in the equations for a non-recursive declaration (E.2.15) and of an abstraction (E.2.26):

```
                  __         __
                    |       |  { let old.env = this.env
         { e(P)A.dest.(i)   |       |    e(P)A
when i:ENV  C            |  =>  |    C                          [R7.8]
         }                  |       |    reset(old.env)
                  __|       |__}
```

Elimination: Recall R7.6, the rule that eliminated environments from parameter lists. Such elimination was possible because of the existence of a global symbol structure. For the same reason, we introduce below a few rules to eliminate the environment in three other constructions.

In Snapshot 5.15, and as a result of R4.7, we find an iterative construction defining a new environment in each iteration:

```
for I=1 to E         | => |  for I=1 to E                  [R7.9]
do e(P)A.dest.(i)__|    |__do e(P)A
```

Also, in (E.2.21), and as a result of R4.1, an environment might appear as a single variable standing in a block as a statement:

Snapshot 5.16: GEDANKEN: Sequence of Parameters. Environment Analysis

```
case [p1, ... ,p2]:
  { let node.vec = open.node(node)
   {0 let continue2 = forward(COD)
      Ccoerce(reg).cont.(continue2).dest.(first.reg)
      fix.here(continue2)
      trans.skip.if.in(first.reg, F)
      trans.jump.to(Cerror)
      { let dmp.loc = trans.dump(first.reg)
        for inx=1 to !node.vec-1
        do { { let continue1 = forward(COD)
               U(node.vec!inx, first.reg|F, inx).cont.(continue1)
               fix.here(continue1)
             }
             trans.load(F, dmp.loc).dest.(first.reg)
           }
        unless !node.vec=0
        do U(node.vec!!node.vec, first.reg|F, #[p1, ... ,p2]
             ).cont.(continue)
   }0
   freevec(node.vec)
  }; endcase
                                  by R7.6/twice, R7.9, R7.11  (5.16.1)
```

when i:ENV $\{ C_0;\ i;\ C_1 \} \quad \Rightarrow \quad \{ C_0;\ C_1 \}$ [R7.10]

In Snapshot 5.15, (E.2.22) and (E.2.28) and as a result of R4.4 which introduced undefined environments of the form **.dest.**(?:d) with d$\underline{C}$ENV:

when d$\underline{C}$ENV e(P)A**.dest.**(?:d) $\Rightarrow$ e(P)A [R7.11]

Snapshot 5.16 shown the result of the Environment Analysis.

5.4.5 BCPL

Finally, two more transformation rules from WFF_s to WFF_t. In (5.16.1), the WFF_s operator '#' is used to denote the size of the sequence. In WFF_t this size is accessed with a selector:

$$\#[s_1 \ldots s_n] \quad \Rightarrow \quad size\hat{}[s_1 \ldots s_n] \qquad [RA.9]$$

An integer expression, such as size^E, a constant, or !node.vec, which is not used as a control structure, but as a code generation value, needs to be given a descriptor (a type definition) so that the machine interface can interrogate it. This distinguishes it from other expressions like a location, an instruction, or a pointer to the symbol table.

when e_1:INT $\quad e_0(P_0, e_1, P_1)A \quad \Rightarrow \quad e_0(P_0, e_2, P_1)A$ where e_2=make.num(e_1) [RA.10]

The primitive function <u>make.num</u> provides such a data definition. The internal 'domain of interest' INT, indicates the 'compile-time' integers. The numerals, the **for** loop control variable and the size of vectors and syntactic nodes are all in INT, namely: n:INT, inx:INT, !E:INT and size^E:INT. The result of these rules can be found in (E.2.9) and (E.2.20).

CHAPTER 6

The Lambda Calculus

In this chapter we describe a correspondence between the Standard Denotational Semantics of the Lambda Calculus(LC) and a Code Generation Process. Two semantics are considered in turn, both based on [Rey74] as described in [Sto77] and extended with a few basic constructions to allow the translation of realistic programs. The first semantics is direct. The second utilises continuations in a way we have seen in previous chapters. Apart from the new transformations required to distinguish between call-by-value and call-by-name, the important aspect of this chapter is the degree of confidence of the correctness of our transformational system that results from the comparison between both direct and continuation cases. The resultant CGPs, two programs derived from two congruent specifications are different but produce the same code in the examples we tried.

6.1 Direct Semantics

In this section, the first version of the semantics of the lambda calculus is transformed from the original specification of Snapshot 6.1 in the source metalanguage WFF_s, up to the final target version in BCPL (WFF_t).

6.1.1 Syntactic Transformations

Non-Strict And Thunk: Consider the definition of Strict:

$$\text{Strict } (\lambda i.e)(e_1) \quad = \quad \begin{cases} \textbf{Bot},\textbf{Top} \text{ or ? If } e_1 \text{ is } \textbf{Bot},\textbf{Top} \text{ or ?} \\ (\lambda i.e)(e_1) \quad \text{Otherwise} \end{cases} \qquad [D13]$$

The function returned by Strict, is that which, if applied to an improper

Snapshot 6.1: The Lambda Calculus(Direct). Original Specification

Syntactic Categories

i:Ide. identifiers
n:Num. numerals
e:Exp. lambda-expressions
o:Opr. operators

Syntax

e ::= i | n | $e_1 o e_2$ | $e_1 e_2$ | **Lam** i.e_1 | **Lam Val** i.e_1
o ::= + | - | * | / | /\ | \/ | > | < | = | <= | >= | #

Semantic Domains

T. truth values
N. integers
w:W=[N + T + F + { Err }]. expression values
f:F=[W → W]. function values
p:U=[Ide → W]. environments

Semantic Domains of 'Interest'

ENV=U. environments
REG=W. registered values
TEM=F. templates

Semantic Primitives(undefined)

N:[Num → N].
O:[Opr → W → W → W].

Semantic Equations

E:[Exp → U → W]. (6.1.1)

E[i]p=
p[i]. (6.1.2)

E[n]p=
N[n]. (6.1.3)

E[$e_1 o e_2$]p=
(λww'.O[o]ww')(E[e_1]p)(E[e_2]p). (6.1.4)

E[$e_1 e_2$]p=
(λw.w?F→(λw'.(w|F)w')(E[e_2]p),Err)(E[e_1]p). (6.1.5)

E[**Lam** i.e_1]p=
λw.E[e_1](p[w/[i]]). (6.1.6)

E[**Lam Val** i.e_1]p=
Strict(λw.E[e_1](p[w/[i]])). (6.1.7)

value, returns also an improper value. We can think of a strict function as 'looking' at its argument as soon as applied. Whereas a non-strict function will 'disregard' its argument until needed. In implementation terms, this corresponds respectively to call-by-value and call-by-name (or the more efficient call-by-need). But the way that these two features are implemented, influence not only the defined function, but also any and all of its applied occurrences. In the former case, an argument is 'evaluated' and passed to the function; in the latter, arguments are not 'evaluated', but associated with a special kind or function called a 'thunk' after P.Z.Ingerman [Ing61]. These are like mini-functions without arguments and with the environment already bound in. For more implementation details of thunks see [Gri71] or [Bor79]. In the version of the lambda calculus that we have chosen to work with, both forms coexist, the strict abstraction in (6.1.7) demands call-by-value and the non-strict abstraction in (6.1.6) call-by-name. This is why the function Strict can not be eliminated as R3.6 indicates. Hence, we redefine this rule to eliminate the use of Strict only when there is a non-strict function demanding call-by-name.

Strict(e) => e [R3.6]
when e:TEM and there is no e_2 such that Non-Strict(e_2):TEM

In LC, R3.6 does not apply because there is a non-strict expression in (6.1.6). This method implies a switch in the form of the transformation process, which depends on the existence of call-by-name expressions:

1) No call-by-name expressions: (There is no Non-Strict(e):TEM). In this case, the function Strict can be eliminated, the transformation process is switched to a state in which every e:TEM is strict and no analysis of name-value expressions is made. This is the case in all previous example

languages where all occurrences of Strict defining functions in TEM were eliminated.

2) There are call-by-name expressions: (Non-Strict(e):TEM exists). If Strict(e):TEM exists, then in this context Strict imposes in effect a call-by-value definition. In which case we can not eliminate Strict because it is a carrier of information for later analysis. Hence the transformation process is switched to the opposite state, in which unless otherwise indicated, every e:TEM is non-strict. This is the case in the current example language; Strict will remain in context, marking the 'value' abstraction (6.1.7).

At the moment of the call there is no indication whatsoever of the sort of argument required. We have to record this before the semantic analysis begins. The following transformation will mark such an argument:

$$\left|\begin{array}{c} ee_1 \\ \text{or} \\ (\lambda\ i.\ \ldots\ ei\ \ldots\ )e_1 \end{array}\right| \Rightarrow \left|\begin{array}{c} e(\text{Thunk}(e_1)) \\ \text{or} \\ e(\lambda\ i.\ \ldots\ ei\ \ldots\ )(\text{Thunk}(e_1)) \end{array}\right| \qquad [R3.8]$$

where $e_1 \neq i$
when e:TEM and there is an e_2 such that Non-Strict(e_2):TEM

In other words, R3.8 is applied to those expressions occurring as arguments of a template and if, somewhere in the specification, there is a non-strict function which is a member of TEM. In LC, R3.8 applies in (6.1.5). The function Thunk is defined by:

$$\text{Thunk} \quad : \quad [\text{Exp} \rightarrow \text{THU}] \qquad (D14)$$

Where THU is a domain of 'interest'. For example: Thunk(e)A:THU is true, regardless of the functionality of e.

Snapshot 6.2: The Lambda Calculus(Direct). Splitting Continuations

```
let E(node, p) be switchon type^node into
{ case [i]:                                          by R1.1, R3.1   (6.2.1)
    p([i]); endcase
                                                     by R1.1, R3.2   (6.2.2)
  case [n]:
    N([n]); endcase
                                                     by R1.1, R3.2   (6.2.3)
  case [e1oe2]:
    { let w = E([e1], p); let w' = E([e2], p); O([o], w, w') }; endcase
                         by R1.1, R1.2, R3.2/3 times, R3.3, R3.4   (6.2.4)
  case [e1e2]:
    { let w = E([e1], p); w?F→{ let w' = Thunk(E([e2], p)); w|F(w') },Err }
    endcase              by R1.1, R3.2/3 times, R3.3/twice, R3.8   (6.2.5)
  case [Lam i.e1]:
    λw.E([e1], p([w/[i]])); endcase               by R1.1, R3.2/twice   (6.2.6)
  case [Lam Val i.e1]:
    Strict(λw.E([e1], p([w/[i]]))); endcase   by R1.1, R3.2/3 times   (6.2.7)
}
```

This also requires a redefinition of DOM:

DOM(Thunk(e)) = DOM(e) [D15]

So that later in the analysis, transformations not concerned with THU can find the appropriate domain.

Before the Destination Analysis the transformation process looks like Snapshot 6.2. This analysis has isolated the three main areas of concern. At the moment of application (6.2.5), the function Thunk marks the argument as 'needs-thunk'. The 'name' abstraction (6.2.6) has no special markers and the function Strict marks the 'value' abstraction (6.2.7) as 'needs-coercion'.

6.1.2 **Destination Analysis**

Firstly, some redefinition of earlier rules: Recall R5.3 and R5.4, defined in section 3.4.1. We have to extend their conditions to accept the new thunk constructions:

```
                      e(P)           =>     e(P).dest.(reg) In COD          [R5.3]
when (e(P):REG or e(P):THU) and not e:ENV

          { let i=e(P); C }          =>   { e(P).dest.(i) In COD; C }      [R5.4]
when (e(P):REG or e(P):THU) and not e:ENV, rename i=>(i=a_k)>reg+k, reg
```

In a similar way, R5.8 and R5.10, both defined in section 4.2.1, now require a condition also accepting thunks:

```
                               e     =>     [first.reg/reg]e                [R5.8]
when e≠i and (e:TEM or e:THU)

when e:TEM or e:THU            e     =>     trans.load(DOM(e), e) In REG    [R5.10]
```

Secondly, the strict marker is signalling that a 'name' argument is supplied to a 'value' abstraction. This means that functions marked with Strict have to be interpreted as requiring a coercion of the argument to call-by-value, hence its argument (the thunk) must be executed immediately on entry and its destination, as for any other form of call, is always first.reg.

```
                           __        __
                             |      |  Strict(λi.
                             |      |  { trans.call(i).dest.(first.reg)
                 Strict(e)   |  =>  |    [first.reg/i]e_1              [R5.20]
where  e = λi.e_1          __|      |__})
when   e:TEM and there is an e_2 such that Non-Strict(e_2):TEM
```

6.1.3 Continuation Analysis

Thunks are analysed in a similar way to abstractions. The difference lies in the names (and therefore the effect) of the procedures to plant code for entry and exit:

```
                                   { let ntry.code = forward(DOM(e1))
                                     let exit.code = forward(COD)
                                     let skip.code = forward(COD)
                                     trans.jump.to(skip.code)
                                     trans.thunk.entry(ntry.code, node)
when e1:THU    e(P0, e1, P1)A   =>   e2                                  [R6.19]
                                     trans.thunk.exit(exit.code, node)
                                     fix.here(skip.code)
                                     e(P0, ntry.code, P1)A
                                   } where e2=e3 if e1=Thunk(e3)
                                           e2=e1 otherwise
```

The marker <u>Strict</u> as left by R5.20 needs further processing. The look up process expects always a 'name' value, hence it is not sufficient to coerce 'name' arguments. In the case of a strict abstraction, we have to make again a thunk of the 'value' argument. This inefficiency can be avoided by letting the look up process check for the particular type of value, but we are interested in comparing (in section 6.3) this direct case with the continuation case. Hence we define:

```
                        Strict(C)        =>    { C1; C3; C4 }               [R6.20]
when   C:TEM and C = { C1; C2 }
       C1= trans.call(P).dest.(first.reg)
       C2= any
where
            { let ntry.code = forward(DOM(first.reg))
              let exit.code = forward(COD)
              let skip.code = forward(COD)
              trans.jump.to(skip.code)
       C3= <  trans.thunk.entry(ntry.code, node)
              trans.load(DOM(first.reg), first.reg).dest.(first.reg)
              trans.thunk.exit(exit.code, node)
              fix.here(skip.code)
            }
       C4= {ntry.code/first.reg}C2
```

Note that in this and the next rule, we use the special substitution rule { / }, which was defined in section 4.2.4.

6.1.4 Optimising Transformations

This analysis, is necessary regardless of the implementation choice given for first.reg and first.par. We need to ensure that the allocations of destinations within thunks, have the same extent as the thunk.

```
 { C0                             |  |  { C0
   { let ntry.code = E0           |  |    { let ntry.code = E0
     let exit.code = E1           |  |      let exit.code = E1
     let skip.code = E2           |  |      let skip.code = E2
     trans.jump.to(P1)            |  |       { let old.env = this.env
     trans.thunk.entry(P2)        |=>|         let dmp.loc = trans.dump(i)
     C1                           |  |         trans.jump.to(P1)          [R9.6]
     trans.thunk.exit(P3)         |  |         trans.thunk.entry(P2)
     C2                           |  |         {dmp.loc/i}C1
   }                              |  |         trans.thunk.exit(P3)
 }                                |  |         {dmp.loc/i}C2
when i:REG                        |  |         reset(old.env)
     and i is free in C1          |  |       }
                                        }
```

Snapshot 6.3: The Lambda Calculus(Direct). BCPL

```
let trans.E(node).dest.(reg) be switchon type^node into
                                              by R5.1, R7.5, RA.1   (6.3.1)
{ case T..Ident:
    look.up(node).dest.(reg); endcase       by R5.3, R7.4, RA.1/twice   (6.3.2)

  case T..Numeral:
    trans.N(node).dest.(reg); endcase      by R5.3, RA.1/twice, RA.2   (6.3.3)

  case T..Plus: case T..Minus: case T..Mult: case T..Div: case T..And:
  case T..Or: case T..GreaterThan: case T..LessThan: case T..Equal:
  case T..LessOrEqual: case T..GreaterOrEqual: case T..NotEqual:
    trans.E(p1^node).dest.(reg)
    test weight^p2^node=max.reg
    then { let old.env = this.env
           let dmp.loc = trans.dump(reg)
           trans.E(p2^node).dest.(reg)
           trans.O(type^node, dmp.loc, reg).dest.(reg)
           reset(old.env)
         }
    or   { let nxt = next(reg)
           trans.E(p2^node).dest.(nxt)
           trans.O(type^node, reg, nxt).dest.(reg)
         }; endcase
    by R5.3, R5.4/twice, R5.6, R7.6/twice, R9.1, RA.1/7 times, RA.2/5 times
                                                                    (6.3.4)
```

Snapshot 6.3 (continued)

```
case N2..Application:
  trans.E(p1^node).dest.(reg)
  { let econd.code = forward(D..COD)
    let fcond.code = forward(D..COD)
    trans.skip.if.in(reg, D..F)
    trans.jump.to(fcond.code, true.jump)
    test reg=max.reg
    then { let old.env = this.env
           let dmp.loc = trans.dump(reg)
           { let ntry.domW = forward(D..W)
             let exit.code = forward(D..COD)
             let skip.code = forward(D..COD)
             trans.jump.to(skip.code, true.jump)
             trans.thunk.entry(ntry.domW, node)
             trans.E(p2^node).dest.(first.reg)
             trans.thunk.exit(exit.code, node)
             fix.here(skip.code)
             trans.load(D..W, ntry.domW).dest.(reg)
           }
           trans.call(dmp.loc, reg).dest.(first.reg)
           reset(old.env)
         }
    or   { let nxt = next(reg)
           { let ntry.domW = forward(D..W)
             let exit.code = forward(D..COD)
             let skip.code = forward(D..COD)
             trans.jump.to(skip.code, true.jump)
             trans.thunk.entry(ntry.domW, node)
             trans.E(p2^node).dest.(first.reg)
             trans.thunk.exit(exit.code, node)
             fix.here(skip.code)
             trans.load(D..W, ntry.domW).dest.(nxt)
           }
           trans.call(reg, nxt).dest.(first.reg)
         }
    trans.jump.to(econd.code, true.jump)
    fix.here(fcond.code)
    trans.load(D..W, Err).dest.(reg)
    fix.here(econd.code)
  }; endcase
  by R5.3/twice, R5.4/twice, R5.7, R5.8, R5.10, R5.11, R6.2, R6.6, R6.7
  R6.19, R7.6/twice, R8.2/3 times, R9.1, RA.1/4 times, RA.2/15 times
                                                                    (6.3.5)
```

Snapshot 6.3 (continued)

```
case N2..Abstraction:
  { let ntry.domF = forward(D..F)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domF, node)
    { let old.env = this.env
      declare(domain.of(first.par), first.par, p1^node)
      trans.E(p2^node).dest.(first.reg)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..F, ntry.domF).dest.(reg)
  }; endcase
  by R5.3/twice, R5.8, R5.9, R5.10, R6.4, R7.1, R7.2, R8.2, RA.1/3 times
  RA.2/5 times                                                      (6.3.6)

case N2..ValAbstraction:
  { let ntry.domF = forward(D..F)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domF, node)
    trans.call(first.par).dest.(first.reg)
    {0 let ntry.domW1 = forward(D..W)
       let exit.code1 = forward(D..COD)
       let skip.code1 = forward(D..COD)
       { let old.env = this.env
         let dmp.loc = trans.dump(first.reg)
         trans.jump.to(skip.code1, true.jump)
         trans.thunk.entry(ntry.domW1, node)
         trans.load(domain.of(dmp.loc), dmp.loc).dest.(first.reg)
         trans.thunk.exit(exit.code1, node)
         fix.here(skip.code1)
         declare(domain.of(ntry.domW1), ntry.domW1, p1^node)
         trans.E(p2^node).dest.(first.reg)
         reset(old.env)
    }0
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..F, ntry.domF).dest.(reg)
  }; endcase
  by R5.3/twice, R5.8, R5.9, R5.10, R5.20, R6.4, R6.20, R7.1, R7.2
  R8.2/twice, R9.2, R9.6, RA.1/3 times, RA.2/8 times                (6.3.7)
}
```

Snapshot 6.4: The Lambda Calculus(Continuation). Original Specification

Syntactic Categories

i:Ide.	identifiers
n:Num.	numerals
e:Exp.	lambda-expressions
o:Opr.	operators

Syntax

e ::= i | n | $e_1 o e_2$ | $e_1 e_2$ | **Lam** i.e_1 | **Lam Val** i.e_1

o ::= + | - | * | / | /\ | \/ | > | < | = | <= | >= | #

Semantic Domains

T.	truth values
N.	integers
A.	answers
$w:W=[K \rightarrow A]$.	expression closures
$k:K=[E \rightarrow A]$.	expression continuations
$e:E=[N + T + F + \{ Err \}]$.	expression values
$F=[W \rightarrow W]$.	function values
$p:U=[Ide \rightarrow W]$.	environments

Semantic Domains of 'Interest'

ENV=U.	environments
REG=E.	registered values
TEM=F.	templates
THU=W.	thunks

Semantic Primitives (undefined)

$N:[Num \rightarrow N]$.

$O:[Opr \rightarrow E \rightarrow E \rightarrow W]$.

Semantic Equations

$E:[Exp \rightarrow U \rightarrow W]$.

$$E[i]pk= \quad p[i]k. \tag{6.4.1}$$

$$E[n]pk= \quad k(N[n]). \tag{6.4.2}$$

$$E[e_1 o e_2]pk= \quad E[e_1]p(\lambda e.E[e_2]p(\lambda e'.O[o]ee'k)). \tag{6.4.3}$$

$$E[e_1 e_2]pk= \quad E[e_1]p(\lambda e.e?F \rightarrow (\lambda w'.(e|F)w'k)(\lambda k.E[e_2]pk),kErr). \tag{6.4.4}$$

$$E[\mathbf{Lam}\ i.e_1]pk= \quad k(\lambda wk'.E[e_1](p[w/[i]])k'). \tag{6.4.5}$$

$$E[\mathbf{Lam\ Val}\ i.e_1]pk= \quad k(\lambda wk'.w(\lambda e.E[e_1](p[\lambda k.ke/[i]])k')). \tag{6.4.6}$$

(6.4.7)

6.2 Continuation Semantics

Let us turn our attention, to another version of a DS specification for the same language. It is the continuation semantics, as described in Snapshot 6.4.

Recall that in the previous section, we had to mark 'name' arguments (R3.8). This was done by reference to the existence of a non-strict(e) in TEM. In this version of the LC, the domain W of expression closures, already indicates where thunks are, this is why among the domains of 'interest', we have defined THU=W. In Snapshot 6.5 we show the state of the transformation process before the Destination Analysis so that it can be compared with Snapshot 6.2. In particular, note the definition of w′ in (6.2.5) and (6.5.5), the request for a thunk in the argument for a call is indicated in the former by <u>Thunk</u> (as a result of R3.8). In the latter, the thunk is indicated by the abstraction λk.e which is in THU (this abstraction has been carried over from the original specification), so both expressions are in THU and are converted in a similar way to thunks, respectively by R6.19 and R6.21 (to be defined below). Also, note in (6.2.7) and (6.5.7) how 'value' abstractions are, at this point, quite different: In the former, <u>Strict</u> marks such abstraction to allow R5.20 and R6.20 to transform accordingly. In the latter, the argument w is involved in an application (to find the value associated with the 'name' argument); this will be transformed by R5.21 and the value in the declaration of [i] is again remade into a 'name' argument, namely: λk.{ e; k }. and transformed by R6.21 (both these rules are defined below).

Snapshot 6.5: The Lambda Calculus(Continuation). Splitting Continuations

```
let E(node, p).cont.(k).dest.(?) be switchon type^node into
                                          by R1.1, R3.1, R4.3    (6.5.1)
{ case [i]:
  p([i]).cont.(k).dest.(?); endcase
                                          by R1.1, R3.2, R4.4    (6.5.2)
  case [n]:
  N([n]); k; endcase
                                   by R1.1, R3.2/twice, R4.1     (6.5.3)
  case [e1oe2]:
  E([e1], p
  ).cont.(E([e2], p).cont.(O([o], e, e').cont.(k).dest.(?)).dest.(e')
  ).dest.(e); endcase   by R1.1, R3.2/3 times, R4.2/twice, R4.4  (6.5.4)
  case [e1e2]:
   E([e1], p
   ).cont.(e?F->
           { let w' = λk.E([e2], p).cont.(k).dest.(?)
             e|F(w').cont.(k).dest.(?)
           },Err; k).dest.(e); endcase
      by R1.1, R3.2/4 times, R3.3, R4.1, R4.2, R4.4/twice        (6.5.5)
  case [Lam i.e1]:
  λw.λk'.E([e1], p([w/[i]])).cont.(k').dest.(?); k; endcase
                         by R1.1, R1.2, R3.2/3 times, R4.1, R4.4  (6.5.6)
  case [Lam Val i.e1]:
  λw.λk'.w().cont.(E([e1], p([λk.{ e; k }/[i]])).cont.(k').dest.(?)
           ).dest.(e)
  k; endcase by R1.1, R1.2, R3.2/5 times, R4.1/twice, R4.2, R4.4  (6.5.7)
}
```

6.2.1 Destination Analysis

With the new domain of 'interest' THU, we can still apply R5.4, R5.8 and R5.10, but we need a new rule for explicit calls of thunks which should be compared with R5.20.

```
                              |   | { trans.call(e).dest.(first.reg)
      e().cont.(e1).dest.(i)  | =>|   [first.reg/i]e1
when e:THU                    |   | }                           [R5.21]
```

6.2.2 Continuation Analysis

The conversion for thunks, can not be the same as that one used in the direct semantics. The case that we are now considering, involves a lambda abstraction for the return continuation:

```
                                 __     __
                                   |   |  { let ntry.code = forward(DOM(e1))
                                   |   |    let exit.code = forward(COD)
                                   |   |    let skip.code = forward(COD)
                                   |   |    trans.jump.to(skip.code)
                                   |   |    trans.thunk.entry(ntry.code, node)
        e(P0, λi.e1, P1)A          | =>|    [exit.code/i]e1                  [R6.21]
                                   |   |    trans.thunk.exit(exit.code, node)
                                   |   |    fix.here(skip.code)
                                   |   |    e(P0, ntry.code, P1)A
when λi.e1:THU                   __|   |__ }
```

And we redefine R6.12 to cope also with thunks:

```
                                 __     __
                                   |   |  e3
     e(P0, e0([e1/e2], P1)A        | =>|  e(P0, e0([ntry.code/e2], P1)A  [R6.12]
                                   |   |  where except for the last statement
when e1:TEM or e1:THU            __|   |__ e3 is the same as R6.11 or R6.21
```

6.3 Comparison

J. Reynolds [Rey74] and J. Stoy [Sto76] have shown the congruence of direct and continuation semantic descriptions by setting up predicates using the framework of [MaS76]. Since direct and continuation semantics can be proven congruent, it seems then natural to compare the corresponding generated code generation processes. However, as indicated in Chapter 1, we are not concerned with formal proofs of correctness. Not because such proofs are uninteresting, but because we devoted our research to develop a transformation system able to generate efficient and usable code generators. We hope that once this has been achieved, future generations will continue the work and hopefully prove it correct. However, a glance comparing both final versions shown in Snapshot 6.3 and Snapshot 6.6 (below) is sufficient to give us confidence in the correctness of our transformations. Moreover, lacking correctness proofs, a further and more interesting comparison can be made, i.e: the code generated by each one.

When running both CGP over input programs like the following:

```
comment:        True                ≡ Lam x. Lam y. x
comment:        False               ≡ Lam x. Lam y. y
comment:        Let     i=e In e′   ≡ (Lam     i.e′)(e)
comment:        Let Val i=e In e′   ≡ (Lam Val i.e′)(e)

Let Val Y = Lam f.(Lam x.f(xx))(Lam x.f(xx))
In Let Val Fact = Y(Lam f.Lam Val n.(n=0) 1 (n*f(n-1)))
In Let Val Cons = Lam Val x.Lam Val y.Lam Val z.(z=1)xy
In Let Val Car = Lam Val x.x 1
In Let Val Cdr = Lam Val x.x 2
In Let Val Apply = Y(Lam f.Lam Val g.Lam n.(n=0) 0 (Cons (g n) (f(g)(n-1))))
In Car(Cdr(Apply Fact 7))
```

we found that the code generated by both is without exception precisely the same. This result can be expressed thus:

```
    DIRECT        -----------> CGP-d ----------> +---------+
       |                                         |         |
    congruent                                    |  same   |
  specifications                                 | DEC-10  |
       |                                         |  code   |
CONTINUATION ----------> CGP-c ----------->      |         |
                                                 +---------+
```

An excellent result which, unfortunately, does not prove that the transformation process is correct, but that it is consistent, at least for the examples we tried. It might be the case of both CGPs generate the same 'wrong' code. That this is not so, was checked again empirically, by running the code that they produce, verifying that correct answers were produced.

However, it appears that the treatment of forward references, or Continuation Analysis is correct. The CGP derived from the continuation semantics consists of a procedure which keeps always a forward reference to the next expression to translate, where as the CGP associated with the direct semantics consists of a procedure with no 'context' information. This information is only used to plant jump instructions to follow the flow of control. The flow of control of the CGP generated from the direct semantics, depends heavily on the form of the concrete semantics, and hence by the particular order imposed by the transformation process. The form of the expression continuations dictates the flow of control in the continuation CGP. The fact that both orders coincide, derives only from our design decision of imposing a left to right order. If the specifications are rewritten with an unspecified order of evaluation, for example, using a list evaluation operator like le of [Sto77] pp-265, then the flow of control could differ if different assumptions regarding le were chosen.

With respect to efficiency, again there is a penalty paid by the form of the semantic specifications. Consider the treatment of the environment, In the continuation case, denoted values are in [K>A], the domain of expression closures. And this domain is the one associated with thunks. So in this case, the CGP was directed to make thunks, before declaring the parameter of a 'value' abstraction. In the direct specification, declarations are not forced to be thunks, we could avoid R6.20 and let look.up decide what kind of object has been declared. We implemented such a version and indeed the direct CGP that resulted was more efficient, because thunks are not created while declaring the argument of 'name' abstractions. However, R6.20 allows the transformation process to produce two CGPs, different in their form but equivalent in their translation. This is what we set out to achieve.

Snapshot 6.6: The Lambda Calculus(Continuation). BCPL

```
let trans.E(node).cont.(continue, jump).dest.(reg) be
switchon type^node into              by R5.12, R6.10, R7.5, R8.1, RA.1    (6.6.1)
{ case T..Ident:
   look.up(node).cont.(continue, jump).dest.(reg); endcase
                           by R5.13, R6.10, R7.4, R8.1, RA.1/twice    (6.6.2)

  case T..Numeral:
   trans.N(node).dest.(reg); trans.jump.to(continue, jump); endcase
                     by R5.3, R6.1, R6.10, R8.1, RA.1/twice, RA.2    (6.6.3)

  case T..Plus: case T..Minus: case T..Mult: case T..Div: case T..And:
  case T..Or: case T..GreaterThan: case T..LessThan: case T..Equal:
  case T..LessOrEqual: case T..GreaterOrEqual: case T..NotEqual:
   {0 let continue2 = forward(D..COD)
     trans.E(p1^node).cont.(continue2, false.jump).dest.(reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       test weight^p2^node=max.reg
       then { let old.env = this.env
              let dmp.loc = trans.dump(reg)
              trans.E(p2^node).cont.(continue1, false.jump).dest.(reg)
              fix.here(continue1)
              trans.O(type^node, dmp.loc, reg).cont.(continue, jump
                       ).dest.(reg)
              reset(old.env)
            }
        or  { let nxt = next(reg)
              trans.E(p2^node).cont.(continue1, false.jump).dest.(nxt)
              fix.here(continue1)
              trans.O(type^node, reg, nxt).cont.(continue, jump
                       ).dest.(reg)
            }
   }0; endcase
  by R5.13, R5.14/twice, R6.9/twice, R6.10, R7.6/twice, R8.1, R8.2/twice
  R9.1, RA.1/7 times, RA.2/7 times
                                                                  (6.6.4)
```

Snapshot 6.6 (continued)

```
case N2..Application:
  {0 let continuel = forward(D..COD)
    trans.E(p1^node).cont.(continuel, false.jump).dest.(reg)
    fix.here(continuel)
    { let fcond.code = forward(D..COD)
      trans.skip.if.in(reg, D..F)
      trans.jump.to(fcond.code, true.jump)
      test reg=max.reg
      then { let old.env = this.env
             let dmp.loc = trans.dump(reg)
             { let ntry.domW = forward(D..W)
               let exit.code = forward(D..COD)
               let skip.code = forward(D..COD)
               trans.jump.to(skip.code, true.jump)
               trans.thunk.entry(ntry.domW, node)
               trans.E(p2^node).cont.(exit.code, false.jump
                           ).dest.(first.reg)
               trans.thunk.exit(exit.code, node)
               fix.here(skip.code)
               trans.load(D..W, ntry.domW).dest.(reg)
             }
             trans.call(dmp.loc, reg).cont.(continue, true.jump
                          ).dest.(first.reg)
             reset(old.env)
           }
      or   { let nxt = next(reg)
             { let ntry.domW = forward(D..W)
               let exit.code = forward(D..COD)
               let skip.code = forward(D..COD)
               trans.jump.to(skip.code, true.jump)
               trans.thunk.entry(ntry.domW, node)
               trans.E(p2^node).cont.(exit.code, false.jump
                           ).dest.(first.reg)
               trans.thunk.exit(exit.code, node)
               fix.here(skip.code)
               trans.load(D..W, ntry.domW).dest.(nxt)
             }
             trans.call(reg, nxt).cont.(continue, true.jump
                             ).dest.(first.reg)
           }
      fix.here(fcond.code)
      trans.load(D..E, Err).dest.(reg)
      trans.jump.to(continue, jump)
  }0; endcase
  by R5.3, R5.4, R5.7, R5.8, R5.10, R5.13, R5.14, R5.15, R6.1, R6.2, R6.6
  R6.7, R6.9, R6.10/twice, R6.21, R7.6/twice, R8.1/twice, R8.2/4 times
  R9.1, RA.1/4 times, RA.2/15 times                                (6.6.5)
```

Snapshot 6.6 (continued)

```
case N2..Abstraction:
  { let ntry.domF = forward(D..F)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domF, node)
    { let old.env = this.env
      declare(domain.of(first.par), first.par, p1^node)
      trans.E(p2^node).cont.(exit.code, false.jump).dest.(first.reg)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..F, ntry.domF).dest.(reg)
  }
  trans.jump.to(continue, jump); endcase
  by R5.3, R5.8, R5.9, R5.10, R5.13, R6.1, R6.10, R6.11, R7.1, R7.2, R8.1
  R8.2/twice, RA.1/3 times, RA.2/5 times
                                                                  (6.6.6)

case N2..ValAbstraction:
  { let ntry.domF = forward(D..F)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domF, node)
    trans.call(first.par).dest.(first.reg)
    {0 let ntry.domW1 = forward(D..W)
      let exit.code1 = forward(D..COD)
      let skip.code1 = forward(D..COD)
       { let old.env = this.env
         let dmp.loc = trans.dump(first.reg)
         trans.jump.to(skip.code1, true.jump)
         trans.thunk.entry(ntry.domW1, node)
         trans.load(domain.of(dmp.loc), dmp.loc).dest.(first.reg)
         trans.thunk.exit(exit.code1, node)
         fix.here(skip.code1)
         declare(D..W, ntry.domW1, p1^node)
         trans.E(p2^node).cont.(exit.code, false.jump).dest.(first.reg)
         reset(old.env)
    }0
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..F, ntry.domF).dest.(reg)
  }
  trans.jump.to(continue, jump); endcase
  by R5.3/twice, R5.7, R5.8, R5.9, R5.10, R5.13, R5.14, R5.21, R6.1/twice
  R6.10, R6.11, R6.12, R6.21, R7.1, R7.2, R8.1, R8.2/4 times, R8.3, R9.2
  R9.6, RA.1/3 times, RA.2/9 times
}                                                                 (6.6.7)
```

CHAPTER 7

From Standard to Implementation DS

Our conjecture concerning the relationship between compilers and semantic equations is that not only the semantic equations can dictate the structure of a compiler, but conversely, intuitions and experience of compiler writers could 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:

<u>Standard</u> Denotational Semantics (SDS):

A canonical definition free of bias towards any particular implementation.

<u>Implementation</u> Denotational Semantics (IDS):

Embodying all implementation strategies desired.

In this chapter, we will show how four implementation issues can be encapsulated at this level, namely:

1 Efficiency in boolean expressions.

2 Efficiency in arithmetic expressions.

3 The allocation of locations.

4 Declaration and invocation records.

Snapshot 7.1: Boolean Expressions(SDS). Original Specification

Syntactic Domain

b:Bex. boolean expressions

i:Ide. identifiers

Syntax

$b ::= i \mid b_1/\backslash b_2 \mid b_1\backslash/b_2 \mid b_1\text{->}b_2,b_3$

Semantics Domains

s:S=[Ide → T]. states

c:C=[S → S]. command continuations

k:K=[T → C]. expression continuations

t:T=[{ TRUE } + { FALSE }]. truth values

Semantic Domains of 'Interest'

REG=T. registered values

STA=S. states

Semantic Equations

B:[Bex → K → C]. (7.1.1)

B[i]k=
 λs.k(s[i])s. (7.1.2)

$B[b_1/\backslash b_2]k=$
 $B[b_1]\{\lambda t.t \to B[b_2]k,k\mathrm{FALSE}\}$. (7.1.3)

$B[b_1\backslash/b_2]k=$
 $B[b_1]\{\lambda t.t \to k\mathrm{TRUE},B[b_2]k\}$. (7.1.4)

$B[b_1\text{->}b_2,b_3]k=$
 $B[b_1]\{\lambda t.t \to B[b_2]k,B[b_3]k\}$. (7.1.5)

7.1 Boolean Expressions

Consider the SDS specification of boolean expressions of Snapshot 7.1. Boolean expressions viewed in this way are like any other expression with the exception that they evaluate to boolean values. So for example, the expression:

a/\b/\((c/\d\/e/\f)/\(g\/h))

compiled with the corresponding CGP shown in Snapshot 7.2 will generate the DEC-10 code shown below that snapshot. But it happens that boolean expressions can be evaluated in a completely different way. Their evaluation

Snapshot 7.2: Boolean Expressions(SDS). BCPL

```
let trans.B(node).cont.(continue, jump).dest.(reg) be
switchon type^node into                                                  (7.2.1)
{ case T..Ident:
    trans.load(D..Ide, node).dest.(reg); trans.jump.to(continue, jump)
    endcase                                                              (7.2.2)

  case N2..And:
    {0 let continuel = forward(D..COD)
       trans.B(pl^node).cont.(continuel, false.jump).dest.(reg)
       fix.here(continuel)
       { let fcond.code = forward(D..COD)
         trans.jump.if.false(reg, fcond.code)
         trans.B(p2^node).cont.(continue, true.jump).dest.(reg)
         fix.here(fcond.code)
         trans.load(D..T, FALSE).dest.(reg)
         trans.jump.to(continue, jump)
    }0; endcase                                                          (7.2.3)

  case N2..Or:
    {0 let continuel = forward(D..COD)
       trans.B(pl^node).cont.(continuel, false.jump).dest.(reg)
       fix.here(continuel)
       { let fcond.code = forward(D..COD)
         trans.jump.if.false(reg, fcond.code)
         trans.load(D..T, TRUE).dest.(reg)
         trans.jump.to(continue, true.jump)
         fix.here(fcond.code)
         trans.B(p2^node).cont.(continue, jump).dest.(reg)
    }0; endcase                                                          (7.2.4)

  case N3..Conditional:
    {0 let continuel = forward(D..COD)
       trans.B(pl^node).cont.(continuel, false.jump).dest.(reg)
       fix.here(continuel)
       { let fcond.code = forward(D..COD)
         trans.jump.if.false(reg, fcond.code)
         trans.B(p2^node).cont.(continue, true.jump).dest.(reg)
         fix.here(fcond.code)
         trans.B(p3^node).cont.(continue, jump).dest.(reg)
    }0; endcase                                                          (7.2.5)
}
```

```
       MOVE   AC1,a   |L3:  SETZ   AC1,0  |       MOVE   AC1,g
       JUMPE  AC1,L1  |L4:  JUMPE  AC1,L5 |       JUMPE  AC1,L8
       MOVE   AC1,b   |     SETO   AC1,0  |       SETO   AC1,0
       JRST   0,L2    |     JRST   0,L7   |       JRST   0,L11
L1:    SETZ   AC1,0   |L5:  MOVE   AC1,e  |L8:    MOVE   AC1,h
L2:    JUMPE  AC1,L10 |     JUMPE  AC1,L6 |       JRST   0,L11
       MOVE   AC1,c   |     MOVE   AC1,f  |L9:    SETZ   AC1,0
       JUMPE  AC1,L3  |     JRST   0,L7   |       JRST   0,L11
       MOVE   AC1,d   |L6:  SETZ   AC1,0  |L10:   SETZ   AC1,0
       JRST   0,L4    |L7:  JUMPE  AC1,L9 |L11:   ; result in AC1
```

need not produce a value but can select the next path of the computations. 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.

Snapshot 7.3: Boolean Expressions(IDS). Original Specification

Semantic Equations

$B:[Bex \rightarrow C \rightarrow C \rightarrow C]$. (7.3.1)

$B[i]cc'=$
$\lambda s.s[i] \rightarrow cs,c's.$ (7.3.2)

$B[b_1/\backslash b_2]cc'=$
$B[b_1]\{B[b_2]cc'\}c'.$ (7.3.3)

$B[b_1\backslash/b_2]cc'=$
$B[b_1]c\{B[b_2]cc'\}.$ (7.3.4)

$B[b_1->b_2,b_3]cc'=$
$B[b_1]\{B[b_2]cc'\}\{B[b_3]cc'\}.$ (7.3.5)

This model of boolean expressions with two continuations as described in Snapshot 7.3, corresponds precisely to a way that efficient compilers implement them, namely as true and false chains. A simple extension to the Continuation Analysis to cope with pairs of continuations will produce the efficient CGP for boolean expressions as described in Snapshot 7.4 with the corresponding 'ideal' code for the same expression shown below it.

Note that not only the number of generated instructions has been reduced (from 29 to 16), due to the absence of register assignments (SETZ and SETO) and jumps (JRST), but also the length of the CGP is shorter because in

Snapshot 7.4: Boolean Expressions(IDS). BCPL

```
let trans.B(node).cont.(continue, continuel, jump) be
switchon type^node into                                         (7.4.1)
{ case T..Ident:
    trans.load(D..Ide, node).dest.(first.reg)
    test jump
    then { trans.jump.if.true(first.reg, continue)
           trans.jump.to(continuel, not jump)
         }
    or   trans.jump.if.false(first.reg, continuel); endcase     (7.4.2)

  case N2..And:
    { let continue2 = forward(D..COD)
      trans.B(p1^node).cont.(continue2, continuel, false.jump)
      fix.here(continue2)
      trans.B(p2^node).cont.(continue, continuel, jump)
    }; endcase                                                  (7.4.3)

  case N2..Or:
    { let continue2 = forward(D..COD)
      trans.B(p1^node).cont.(continue, continue2, true.jump)
      fix.here(continue2)
      trans.B(p2^node).cont.(continue, continuel, jump)
    }; endcase                                                  (7.4.4)

  case N3..Conditional:
    { let continue2 = forward(D..COD)
      let continue3 = forward(D..COD)
      trans.B(p1^node).cont.(continue2, continue3, false.jump)
      fix.here(continue2)
      trans.B(p2^node).cont.(continue, continuel, false.jump)
      trans.jump.to(continue, true.jump)
      fix.here(continue3)
      trans.B(p3^node).cont.(continue, continuel, jump)
    }; endcase                                                  (7.4.5)
}
```

```
        MOVE    AC1,a   |       MOVE    AC1,d   |L2:    MOVE    AC1,g
        JUMPE   AC1,L4  |       JUMPN   AC1,L2  |       JUMPN   AC1,L3
        MOVE    AC1,b   |L1:    MOVE    AC1,e   |       MOVE    AC1,h
        JUMPE   AC1,L4  |       JUMPE   AC1,L4  |       JUMPE   AC1,L4
        MOVE    AC1,c   |       MOVE    AC1,f   |L3:    ; here if true
        JUMPE   AC1,L1  |       JUMPE   AC1,L4  |L4:    ; here if false
```

(7.4.2) and (7.4.3) there is no need to generate those instructions (no trans.load) and in general there are less forward-fix constructions.

Snapshot 7.5: Arithmetic Expressions(SDS). Original Specification

Syntactic Categories

e:Exp.	expressions
i:Ide.	identifiers
n:Num.	numerals
o:Opr.	operators

Syntax

$e ::= i \mid n \mid e_1oe_2$

$o ::= + \mid - \mid * \mid /$

Semantic Domains

n:N.	integers
s:S=[Ide $\rightarrow$ N].	states

Semantic Domains of 'Interest'

REG=N.	registered values
STA=S.	states

Semantic Primitives (undefined)

N:[Num $\rightarrow$ N].

O:[Opr $\rightarrow$ N $\rightarrow$ N $\rightarrow$ N].

Semantic Equations

E:[Exp $\rightarrow$ S $\rightarrow$ N].

(7.5.1)

E[i]=
  Strict(λs.s[i]).

(7.5.2)

E[n]=
  Strict(λs.N[n]).

(7.5.3)

$E[e_1oe_2]=$
  $E[e_1] \underline{+} \lambda n.(E[e_2] \underline{+} \lambda n'.Strict(\lambda s.O[o]nn'))$.

(7.5.4)

7.2 Arithmetic Expressions

To substantiate the claim of producing an efficient compiler, we must ensure that expressions are compiled into efficient code. For example, consider the SDS specification of arithmetic expressions of Snapshot 7.5, with associated CGP as shown in Snapshot 7.6 and example of code generation in the left hand column below it. To generate the 'ideal' code of the right hand side, a better algorithm can easily be implemented; we will follow the one given in [Bor79]. The modified parts of the semantic specification are shown in Snapshot 7.7 where the new operators '--' and '//' are respectively the

Snapshot 7.6: Arithmetic Expressions(SDS). BCPL

```
let trans.E(node).dest.(reg) be switchon type^node into                (7.6.1)
{ case T..Ident:
    trans.load(D..Ide, node).dest.(reg); endcase                       (7.6.2)

  case T..Numeral:
    trans.N(node).dest.(reg); endcase                                  (7.6.3)

  case T..Plus: case T..Minus: case T..Mult: case T..Div:
    trans.E(p1^node).dest.(reg)
    test weight^p2^node=max.reg
    then { let old.env = this.env
           let dmp.loc = trans.dump(reg)
           trans.E(p2^node).dest.(reg)
           trans.O(type^node, dmp.loc, reg).dest.(reg)
           reset(old.env)
         }
    or   { let nxt = next(reg)
           trans.E(p2^node).dest.(nxt)
           trans.O(type^node, reg, nxt).dest.(reg)
         }; endcase                                                    (7.6.4)
}
```

```
        MOVE    AC1,a    |     MOVE    AC1,c
        MOVE    AC2,b    |     IMUL    AC1,d
        IMUL    AC1,AC2  |     MOVE    AC2,e
        MOVE    AC2,c    |     IMUL    AC2,f
        MOVE    AC3,d    |     SUB     AC1,AC2
        IMUL    AC2,AC3  |     MOVE    AC2,g
        MOVE    AC3,e    |     ADD     AC2,h
        MOVE    AC4,f    |     IDIV    AC1,AC2
        IMUL    AC3,AC4  |     MOVE    AC2,a
        SUB     AC2,AC3  |     IMUL    AC2,b
        MOVE    AC3,g    |     xDIVr   AC1,AC2 ;pseudo-op xDIV=reverse(IDIV)
        MOVE    AC4,h    |
        ADD     AC3,AC4  |     Code for:         a*b/((c*d-e*f)/(g+h))
        IDIV    AC2,AC3  |     Left hand side produced by Snapshot 7.6
        IDIV    AC1,AC2  |     Right hand side produced by Snapshot 7.8
```

reverse of '-' and '/'. In fact, these equations, together with the Optimising Transformations abstract the 'register-allocation' techniques of 'tree weighting' and 'dumping', as it can be seen in Snapshot 7.8 when we transform accordingly.

It is interesting to observe, in the translation of the two crucial areas of boolean and arithmetic expressions, the two different methods used in our IDS specifications to improve the kind of code that our CGPs produce, In the

Snapshot 7.7: Arithmetic Expressions(IDS). Original Specification

Modifications to Snapshot 7.5

Syntax

o ::= + | - | -- | * | / | //

Semantic Domain

T=[{ TRUE } + { FALSE }]. booleans

Semantic Domains of 'Interest'

REG=[N + T]. registered values

BOO=T. compile-time booleans

Semantic Primitives (undefined)

IfReverseNeeded:[Exp → BOO].

RLeaf:[Exp → Opr → S → N → N].

Reverse:[Exp → Exp].

IsLeaf:[Exp → BOO].

Semantic Equation

$$E[e_1 o e_2]= IfReverseNeeded[e_1 o e_2] \to E(Reverse[e_1 o e_2] | Exp), (E[e_1] \underline{+} \lambda n. IsLeaf[e_2] \to Strict(\lambda s. RLeaf[e_2][o]sn), (E[e_2] \underline{+} \lambda n'. Strict(\lambda s. O[o]nn'))) . \quad (7.7.4)$$

former, the functionality of the main valuator B was redefined as a function of two command continuations (C), instead of one expression continuation (K=[T→C]), so that the associated CGP could keep track of its operational context. In the latter, the new primitive function IsLeaf was introduced to detect the moment when the translator 'sees' the 'leaf' of an expression, so that the appropriate operation acting on memory could be generated, instead of a load followed by an operation acting on registers. Another primitive function IfReverseNeeded, was introduced to reverse a node to reduce the number of registers required.

Two questions immediately arise:

- Are the SDS and IDS equivalent specifications?
- Is it possible to automatically generate the IDS from the SDS?

Snapshot 7.8: Arithmetic Expressions(IDS). BCPL

A Fragment

```
case T..Plus: case T..Minus: case T..RevMinus: case T..Mult: case T..Div:
case T..RevDiv:
  test IfReverseNeeded(node).dest.(reg)
  then trans.E(Reverse(node)).dest.(reg)
  or   { trans.E(p1^node).dest.(reg)
         test IsLeaf(p2^node).dest.(reg)
         then RLeaf(p2^node, type^node, reg).dest.(reg)
         or   test weight^p2^node=max.reg
              then { let old.env = this.env
                     let dmp.loc = trans.dump(reg)
                     trans.E(p2^node).dest.(reg)
                     trans.O(type^node, dmp.loc, reg).dest.(reg)
                     reset(old.env)
                   }
              or   { let nxt = next(reg)
                     trans.E(p2^node).dest.(nxt)
                     trans.O(type^node, reg, nxt).dest.(reg)
                   }
       }; endcase                                                  (7.8.4)
```

Firstly, the congruence of our IDS specifications with respect to their SDS ones have been proven congruent in [Ras80]. Secondly, we believe that the IDS specification of boolean expressions, which neatly clarifies what happens with these expressions, can be considered - within the frame of a von Neumann sequential architecture - a SDS specification. The fact that it abstracts an implementation idea does not add more light than the fact that environments and states refer to implementations. The treatment of boolean expressions as switches over the flow of control, in the context of programming languages, can be traced back to 1955 (PP-2 compiler - pp 246 in [Knu80]). However, with respect to our IDS treatment of arithmetic expressions, it it possible for the transformation process, to spot the cases where no order of evaluation is implied by a semantics of [e′oe′′] and introduce tree weighting itself. This, we believe, can be done and must be done, if one wishes to start with SDS semantics.

Snapshot 7.9: The Store with Locations(SDS). Original Specification

Syntactic Categories

c:Com.	commands
e:Exp.	expressions
i:Ide.	identifiers

Syntax

c ::= **Let** i:=e **In** c_1 | i:=e

Semantic Domains

e:E.	expression values
$c:C=[S \rightarrow S]$.	state transformations
T=[{ TRUE } + { FALSE }].	truth values
l:L.	locations
$p:U=[Ide \rightarrow L]$.	environments
$s:S=[[L \rightarrow E] \times [L \rightarrow T]]$.	machine states

Semantic Domains of 'Interest'

ENV=U.	environments
REG=E.	registered values
STA=S.	states
LOC=L.	locations

Semantic Primitives

$Conts:[L \rightarrow S \rightarrow E]$.
Conts=
$\lambda ls.(s\downarrow 1)l$.

$Lose:[L \rightarrow C]$.
Lose=
$\lambda ls.\langle s\downarrow 1, \lambda l'.l=l' \rightarrow FALSE,(s\downarrow 2)l'\rangle$.

$Extend:[L \rightarrow C]$.
Extend=
$\lambda ls.\langle s\downarrow 1, \lambda l'.l=l' \rightarrow TRUE,(s\downarrow 2)l'\rangle$.

$NewL:[S \rightarrow L]$.
NewL=
$\lambda s.l$ **where** $(s\downarrow 2)l=FALSE$.

$Update:[L \rightarrow E \rightarrow C]$.
Update=
$\lambda les.\langle \lambda l'.l=l' \rightarrow e,(s\downarrow 1)l', s\downarrow 2\rangle$.

Semantic Equations

$\mathbf{E}:[Exp \rightarrow U \rightarrow S \rightarrow E]$.
$\mathbf{C}:[Com \rightarrow U \rightarrow C]$. (7.9.1)

$\mathbf{C}$[**Let** i:=e **In** c_1]p=
$\mathbf{E}[e]p \;\underline{+}\; \lambda e.\{NewL \;\underline{+}\; \lambda l.\{Extend\ l \;\underline{o}\; Update\ le \;\underline{o}\; \mathbf{C}[c_1](p[l/[i]]) \;\underline{o}\; Lose\ l\}\}$
. (7.9.2)

$\mathbf{C}$[i:=e]p=
$\mathbf{E}[e]p \;\underline{+}\; \lambda e.Update(p[i])e$. (7.9.3)

7.3 Marking locations in use

Consider now the allocation of locations, in the language of Snapshot 7.9 which can be thought as an extension to Snapshot 7.5. The corresponding CGP, shown in Snapshot 7.10, works well and generates the expected code. However, the way that we would implement the primitives: NewL, Extend and Lose is not in the form that this SDS specification dictates. The functions NewL and Extend, which obtain and mark unused locations when necessary, seem to be abstracting a 'free storage' mechanism which is not the one dictated by a block structured discipline. The problem is that the location-deallocation mechanism, where the area function of the state ([L $\succ$ T]) indicates which locations are in use, requires the function Lose to deallocate locations when required. 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 storage in usage are as dynamic as the environment.

Accordingly, we rewrite in IDS the SDS definition. Those parts that differ from Snapshot 7.9 are shown in Snapshot 7.11. The corresponding CGP fragment is shown in Snapshot 7.12. The code that both specifications generate is the same, the main difference is the absence of the primitive Lose, whose activity now is taken by reset. So, what have we achieved with the IDS specification? We have shown how a realistic implementation treats the allocation of locations in a block structured language (level-offset pairs) by allocating them at compile-time, Also, we have shown that the State Analysis and Environment Analysis are both capable of transforming the corresponding semantics.

Snapshot 7.10: The Store with Locations(SDS). BCPL

```
let trans.C(node) be switchon type^node into
{ case N3..Let:                                                      (7.10.1)
    trans.E(p2^node).dest.(first.reg)
    { let l = NewL()
      Extend(l)
      Update(l).dest.(first.reg)
      { let old.env = this.env
        declare(domain.of(l), l, p1^node)
        trans.C(p3^node)
        reset(old.env)
      }
      Lose(l)
    }; endcase
                                                                     (7.10.2)
  case N2..Assignment:
    trans.E(p2^node).dest.(first.reg)
    Update(look.up(p1^node)).dest.(first.reg); endcase
}                                                                    (7.10.3)
```

Snapshot 7.11: The Store with Locations(IDS). Original Specification

Modifications to Snapshot 7.9

Semantic Domains

$p:U=[[Ide \rightarrow L] \times [L \rightarrow T]]$. environments

$s:S=[L \rightarrow E]$. machine states

Semantic Primitives

$Extend:[L \rightarrow U \rightarrow U]$.

$Extend=$

$\lambda lp.\langle p\downarrow 1, \lambda l'.l=l' \rightarrow TRUE, (p\downarrow 2)l' \rangle$.

$NewL:[U \rightarrow L]$.

$NewL=$

$\lambda p.l$ **where** $(p\downarrow 2)l=FALSE$.

$Update:[L \rightarrow E \rightarrow C]$.

$Update=$

$\lambda les.\lambda l'.l=l' \rightarrow e, sl'$.

Semantic Equation

C[Let i:=e **In** c_1]p=

$E[e]p \pm \lambda e.\{\lambda l.\{\lambda p'.\{Update\ le\ \underline{o}\ C[c_1](p'[l/[i]])\}\}(Extend\ lp)\}(NewL\ p)$
.
(7.11.2)

Snapshot 7.12: The Store with Locations(IDS). BCPL

A Fragment

```
case N3..Let:
  trans.E(p2^node).dest.(first.reg)
  { let old.env = this.env
    let l = NewL()
    Extend(l)
    Update(l).dest.(first.reg)
    declare(domain.of(l), l, pl^node)
    trans.C(p3^node)
    reset(old.env)
  }; endcase                                              (7.12.2)
```

7.4 Declaration and Invocation Environment

If we consider the virtual machine behaviour at the different times of declaration, invocation and execution of a function or 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 function or procedure (FP) there is:

(I)	Local binding:	A function to give values for everything which is bound within the FP.
(II)	External binding:	A similar but not equal function to give values for everything which is free in the FP.
(III)	Local workspace:	A function to keep track of those locations defined within the FP
(IV)	Return continuation:	The function mapping what remains to be done when the FP terminates.
(V)	Current continuation:	The function mapping what remains to be done in the FP.

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 ALGOL60. 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

Snapshot 7.13: Environment (SDS). Original Specification

```
Syntactic Domains
e:Exp.                                  expressions
i:Ide.                                  identifiers

Syntax
e ::= i | Let i(i1)=e1 In e2 | e1(e2)

Semantic Domains
t:T=[{ TRUE } + { FALSE }].             truth values
v:V.                                    storable values
e:E=[V + F].                            expression results
c:C=[S → S].                            command continuations
k:K=[E → C].                            expression continuations
d:D=[L + F].                            denotable values
s:S=[[L → V] x [L → T]].                machine states
l:L.                                    locations
  F=[V → K → C].                        function values
p:U=[Ide → D].                          environments

Semantic Primitives
New:[S → L].
New=
  λs.l where (s↓2)l=FALSE.

NewL:[[L → C] → C].
NewL=
  λk:[L → C]s.(kl(Toggle TRUE ls)
  Where l=New s).

FreeL:[L → C → C].
FreeL=
  λlcs.c(Toggle FALSE ls).

Toggle:[T → L → C].
Toggle=
  λtls.<s↓l,λl'.l=l'→t,(s↓2)l'>.

Assign:[L → V → C → C].
Assign=
  λlvcs.c<λl'.l=l'→v,(s↓1)l',s↓2>.

Load:[L → K → C].
Load=
  λlks.k((s↓1)l)s.

Wrong:C.                                undefined

Semantic Domains of 'Interest'
ENV=U.                                  environments
REG=[E + L].                            registered values
STA=S.                                  states
TEM=F.                                  templates
```

Snapshot 7.13 (continued)

Semantic Equations

$R:[Exp \rightarrow U \rightarrow K \rightarrow C]$. (7.13.1)

$R[Let\ i(i_1)=e_1\ In\ e_2]pk=$
$R[e_2]p''k$
$Where\ p''=$
Fix
$(\lambda p'.p$
$[\lambda vk'.NewL\{\lambda l.Assign\ lv\{R[e_1](p'[l/[i_1]])\{\lambda e'.FreeL\ l\{k'e'\}\}\}\}/[i]])$
$.$ (7.13.2)

$R[e_1(e_2)]pk=$
$R[e_1]p\{\lambda e.e?F\rightarrow R[e_2]p\{\lambda e'.e'?V\rightarrow\{e|F\}(e'|V)k,Wrong\},Wrong\}$. (7.13.3)

$R[i]pk=$
$\{\lambda d.d?L\rightarrow Load(d|L)k,k\{d|F\}\}(p[i])$. (7.13.4)

new declarations. For a full description of this model, see [Bor79].

If we now look at the domain definitions and equations of Snapshot 7.13, we can see that there is no clear mathematical machinery to abstract our model at the different times of declaration and invocation. Moreover, 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 function or 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.

Also, the domain of locations is not abstracted at an appropriate level. In the implementation of block structured languages it is reasonable to associate variables to 'offsets' within the workspace of a function or procedure (or perhaps 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. F will be Invocation Record Frame and U an Invocation Record, or in terms of [Bor79] a Mini-Process State Descriptor (mPSD).

```
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 K].               environments
     I    II     III     IV
```

We now describe the parts of the environment, or mPSD in detail.

7.4.1 (I) Local Binding

The binding map:

```
m:M=[Ide → D].                           binding map
```

is the same as the the original environment domain. It binds identifiers to their denoted values. The empty binding map is defined to be:

```
Nild:D.                                  undefined
Nilm:M.
Nilm=
  λ[i].Nild.
```

7.4.2 (II) External Binding

(Or 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 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.

```
LookUp:[Ide → U → G].
LookUp[i]p=
  (λd.d=Nild→LookUp[i](pEXT),d?O→Loc(Nloc<pBAS,d|O>) In G,d?F→d|F In G,Tg)
  (p[i]).
```

7.4.3 (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 than 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 is a finite number of locations and a function:

```
Loc:[N → L].
```

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:

```
Nloc:[[B x O] → N].
```

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 guarantee the existence of the corresponding location. We therefore need the function <u>New</u> 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 no 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[NextO(pTOP) / TOP] = p′
Where p′=λX.NextO(pTOP)          If X=TOP
          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[o/[i]],NextO o>.

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

7.4.4 (IV) Initial and Return Continuation

The forth element in a function closure (F), is a member of the domain of function values (P):

```
P=[U → V → C].                                        function values
```

It models the meaning of the function which is expecting an environment and an actual value for its formal parameter. While in an environment (U), it is a member of the domain of return continuations (C). In relation to [Bor79], (IV) can be seen as a reference to the current continuation field of the calling mPSD.

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

```
Activate:[F → [B x O] → V → K → C].
Activate f<b,o>vk=
   {f↓4}<f↓1,f↓2,<NewB<b,o>,f↓3>,k>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.

Snapshot 7.14: Environment (IDS). Original Specification

Semantic Equations

```
R:[Exp → U → K → C].
                                                              (7.14.1)
R[Let i(i1)=e1 In e2]pk=
  R[e2]p3k
  Where p3=
  Fix(λp'.(λ<m,o>.
          p[<m,p',o,
             λp''v.{λl.Assign lv{R[e1]p''{pRET}}}(New<p''BAS,
                                                       p''[i1]|O
                                                      >)
             >/[i]])(BindF[i1]Nilm FirstO)).                  (7.14.2)

R[e1(e2)]pk=
  R[e1]p{λe.e?F→R[e2]p{λe'.e'?V→Activate(e|F)(pLOC)(e'|V)k,Wrong},Wrong}
                                                              (7.14.3)
R[i]pk=
  {λg.g?L→Load(g|L)k,g?F→k(g|F),Wrong}(LookUp[i]p).           (7.14.4)
```

After incorporating the new environment structure and their associated primitive functions, the IDS definition looks as Snapshot 7.14. The equation (7.14.2) shows how the binding map (m) is formed from the empty one (Nilm) with an additional binding of the parameter $[i_1]$ 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 function value in P.

A Posteriori Evaluation: In general, it seems that our transformational system developed beyond our initial goal. There are many issues, like the treatment of recursion, which are perfectly transformable from a SDS specification. Up to the time of their development they were considered not transformable from other than an IDS specification. As opposed to the preceding three sections, where we have shown the use of an IDS to generate a more efficient CGP, the IDS version that we have just developed is not used any longer in the same way. It represents ideas that we had early in our research, with respect to the the way that we were going to handle recursive procedure and functions. The transformations of Chapter 4 show how our system is quite capable of recognising the crucial moment of entry, exit and call, without the need of any implementation idea abstracted at the level of IDS. In this respect, section 7.4 is a blind-alley.

CHAPTER 8

Conclusion

Every BCPL snapshot shown speaks for itself. The two main examples of the correspondence described are shown in Appendices D and E. They are the final example language of [Sto77] and GEDANKEN. The structure and operation of the code generators obtained for all examples shown is in effect very similar to the one we might have produced by hand. Moreover, the structure of the code that these programs generate is as efficient as the code a hand coded program would generate. To our knowledge, today, there are no compiler generators, directed from a denotational semantics, which achieve this level of efficiency, nor systems whose output is a program written in a systems programming language.

Our research has shown that this task is possible. However, we do not claim a level of generality which allows the transformation of every possible semantic specification. Our transformation system (and associated implementation), needs further investigation: an exhaustive analysis which has been started in order to allow completion of our major examples. A good step in this direction would be the definition of a canonical form of the concrete semantics. The problem is that there are many different ways of representing a function whose only importance is its value (referential transparency). But if a semantic directed generator depends on the concrete semantics, then the way that a semantic function is described is important. P. Mosses's SIS system achieves generality (and correctness) by uniformly translating into lambda, the cost is the lack of efficiency. If one believes that GEDANKEN is not general enough, then we failed to achieve generality. Believing this or not, what we have gained is efficiency at the same level of a hand coded compiler. The cost of our method is that a non-uniform translation is not as 'automatically' correct, as one which faithfully implements the conversions of the lambda calculus.

We have mainly concentrated on the final part of a compiler because this area relates directly to a semantic specification. A syntactic specification, of course, relates to the initial part. The middle area, compile-time type checking, has only been partially considered:

If CHA is the representation of the source program as a character string, SYM the internal representation as a sequence of symbols, TRE the internal representation of programs in the form of a tree and COD the final outcome of a compiler, then today, we are equipped with the following systems which generate programs written in BCPL:

PROGRAM	FUNCTION	UNDERLYING THEORY	SOLVED BY	REFERENCE
scanner	[CHA → SYM]	Finite state machine	LEXGEN	[Suf78a]
parser	[SYM → TRE]	Push down automata	LL1	[Suf78b]
translator	[TRE → COD]	Denotational Semantics	ISL	this thesis

And we still require:

checker	[TRE → TRE]	Denotational Semantics	not done	

It is now imperative to prove that our transformations are correct and preserve meaning. A first attempt to prove this was to regard the generated program as an operational definition and then to relate it to the original specification proving the congruence of the definitions. The problems with this method are, firstly, that it is very difficult, because the domains are very dissimilar. Secondly, this proof has to be restated for each new language.

An alternative approach would be to formalise the semantics of the metalanguage in which the transformations themselves are expressed and then relate the WFF_s to the WFF_t through them. This method is attractive not only because of the possibilities of proving the correctness of our system in a general and language independent way, but also because, having formalised the transformations, one could design an automaton to perform their action. As M. Henson suggested, this would be a compiler-compiler-compiler (3 times). At present, we have implemented (after several years of programming effort) our ISL system (briefly described in Appendix A), which consists of a collection of BCPL modules, which perform the different levels of transformation, rather like a collection of experts, each one relating to a particular denotational feature (a domain of interest) and to a particular implementation technique (of our choice),

An interesting open question is the possibility of extracting, from the DS specification, information about the kind of 'virtual machine' that a particular language might require. At present, we recognise only the location where for example, the CGP must plant code for procedure entry and exit. Our translator generates statements of the form trans.entry(P) and trans.exit(P), but it is unable to predict the sort of code that these procedures should plant, i.e: should the first one get space for an activation record from a stack or from a heap?

Another interesting open question, is the relationship between the translator, as described above and: interpreter:[TRE → MEANING]. Our transformation system has been oriented to produce code generators; a similar system, using similar techniques could be written to produce an interpreter.

We have developed a system to generate code generators for a class of programming languages, with a target code as general as can be expressed within the constraints of the generated primitives. We could actually fix the programming language, and generalise the target machines for a wider class of hardware configurations, say for mini-computers. This might prove to be very useful when considering that today, hardware developments change faster than software developments.

Finally, recall the problem of generating a parser from a BNF specification as an analogy to the problem of generating a code generator, as presented in Chapter 1. Suppose that one wishes to generate or hand write a top down analyser, with one symbol of lookahead and no backtracking. This means that the original BNF specification, has to be rewritten to fulfil the one-track condition [Bor79]. What we have done, is to design and implement an automatic system, analog to an LL1 parser generator, which also expects the specification written in a particular form. The problem is that we do not know, for certain, which are the conditions that the denotational specification has to fulfil. In this sense, we believe that our research is a step towards the definition of such conditions.

* * *

References

AaU69 A.V.Aho and J.D.Ullman. Syntax Directed Translations and the Pushdown Assembler. pp 37-56 in J.Computer and Systems Sciences 3-1, 1969

AaU72 A.V.Aho and J.D.Ullman. The Theory of Parsing, Translation, and Compiling. Prentince-Hall, 1972.

ADA80 Formal Definition of the ADA Programming Language. Preliminary version. INRIA, 1980.

Bjo77 D.Bjorner. Programming languages: Formal development of interpreters and compilers. pp 1-21 in International Computing Symposium, ed E.Morlet and D.Ribbens, North-Holland, Amsterdam, 1977.

Bor79 R.Bornat. Understanding and Writing Compilers. Macmillan. 1979

Bro60 R.A.Brooker and D.Morris. An Assembly Program for a Phrase Structure Language. Harrogate Conference of the British Computer Society, 1960. also in the Computer Journal 3-3

Bro62 R.A.Brooker and D.Morris. A General Translation Program for Phrase Structure Languages. JACM 9, 1962.

Bro63 R.A.Brooker. The Compiler-Compiler. Annual Review pp 229-275 in Automatic Programming, 3. Pergamon, Elmsford, N.Y. 1963.

Cur58 J.B.Curry and R.Feys. Combinatory Logic, Volume I. North-Holland, Amsterdam, 1958.

Gan80 H.Ganzinger. Transforming Denotational Semantics into Practical Attribute Grammars. pp 1-69 in Semantic Directed Compiler Generation, ed N.D.Jones, Lecture Notes in Computer Science 94. Springer-Verlag, Berlin,1980.

Gau81 M.C.Gaudel. Compiler generation from formal definition of programming languages: a survey. pp 96-114 in Formalization of Programming Concepts, Intl. Colloquium, Peniscola, Spain. Lecture Notes in Computer Science 107. Springer-Verlag, Berlin,1981.

Gri71 D.Gries. Compiler Construction For Digital Computers. Wiley International Edition, 1971.

Hen82 M.Henson and R.Turner. Completion Semantics and Interpreter Generation. ACM Principles of Programming Languages, 1982

Ing61 P.Z.Ingerman. Thunks. pp 55-58 in CACM 4-1, 1961.

Iro61 E.T.Irons. A Syntax Directed Compiler for ALGOL60. pp 51-55 in CACM 4-1, 1961.

Jon80 N.D.Jones and D.A.Schimdt. Compiler Generation from Denotational Semantics. pp 70-93 in Semantic Directed Compiler Generation, ed N.D.Jones, Lecture Notes in Computer Science 94. Springer-Verlag, Berlin,1980.

Knu68 D.E.Knuth. Semantics of Context-Free Languages. pp 127-146 in Math. Systems Theory J. 2-2, 1968.

Knu80 D.E.Knuth and L.Trabb Pardo. The Early Development of Programming Languages. pp 197-273 in A History of Computing in the Twentieth Century. Academic Press, 1980

Lan65 P.J.Landin. A Correspondence Between ALGOL60 and Church's Lambda-Notation. pp 89-101,158-165 in CACM 8, 1965.

Lew68 P.M.Lewis and R.E.Sterns. Syntax-Directed Transduction. pp 464-488 in JACM 15-3, 1968.

Lew79 J.Lewi, K. De Vlaminck, J.Huens and M.Huybrechts. A Programming Methodology in Compiler Construction. North-Holland, Amsterdam, 1979.

MaS76 R.E.Milne and C.Strachey. A Theory of Programming Language Semantics. Chapman and Hall, London 1976.

McC60 J.McCarthy. Programs with Common Sense. Proceedings of the Symposium on the Mechanization of Thought Processes. National Physiology Laboratory, Teddington, England, 1960.

Mos74 P.D.Mosses. The Mathematical Semantics of ALGOL60; Technical Monograph PRG-12, Programming Research Group, University of Oxford, 1974.

Mos75 P.D.Mosses. Mathematical Semantics and Compiler Generation. PhD. thesis. University of Oxford, 1975.

Mos76 P.D.Mosses. Compiler Generation using Denotational Semantics. Mathematical Foundations of Computer Science. Lecture Notes in Computer Science 45. Springer-Verlag, 1976.

Mos78 P.D.Mosses. SIS: A Compiler Generator System using Denotational Semantics. Reference Manual, University of Aarhus, 1978.

Mos79 P.D.Mosses. SIS - Semantic Implementation System. Reference Manual and User Guide. DAIMI MD-30, Computer Science Department, Aarhus University, Aarhus, Denmark, 1979.

Pau81 L.Paulson. A Semantic Directed Compiler Generator. ACM Principles of Programming Languages, 1982.

Ple82 J.M.Bodwin, L.Bradley, K.Kanda, D.Litle and U.F.Pleban. Experience with an Experimental Compiler Generator Based on Denotational Semantics. pp 216-229 in Proceedings of the ACM-SIGPLAN'82 Symposium on Compiler Construction. 1982.

Ran75 System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering, SE-1. 1975

Ras79 M.R.Raskovsky and R.Turner. Compiler Generation and Denotational Semantics. Fundamentals of Computation Theory, 1979.

Ras80 M.R.Raskovsky and P.Collier. From Standard to Implementation Denotational Semantics. pp 94-139 in Semantic Directed Compiler Generation, ed N.D.Jones, Lecture Notes in Computer Science 94. Springer-Verlag, Berlin,1980.

Ras81 M.R.Raskovsky. Generating a Real Compiler from a Denotational Semantics. Seminar Review. Institut National de Recherche en Informatique et en Automatique, 1981.

Ras82 M.R.Raskovsky. Denotational Semantics as a Specification of Code Generators. pp 230-244 in Proceedings of the ACM-SIGPLAN'82 Symposium on Compiler Construction. 1982.

Rey70 J.C.Reynolds. GEDANKEN - A Simple Typless Language Based on the Principle of Completeness and the Reference Concept. pp 308-319 in CACM 13-5, 1970

Rey72 J.C.Reynolds. Definitional Interpreters for Higher-order Programming Languages. Proceedings of the 25th ACM National Conference, 1972.

Rey74 J.C.Reynolds. On the Relation Between Direct and Continuation Semantics. pp 141-156 in Proceedings of the Second Colloquium on Automata, Languages and Programming. Springer-Verlag, Berlin, 1974.

Ric79 M.Richards and C.Whitby-Strevens. BCPL, the Language and its Compiler. Cambridge University Press, 1979.

Sco70 D.Scott. Outline of a Mathematical Theory of Computation. Technical Monograph PRG-2, Programming Research Group, University of Oxford, 1970.

Set82 R.Sethi. Control Flow Aspects of Semantic Directed Compiling. pp 245-260 in Proceedings of the ACM-SIGPLAN'82 Symposium on Compiler Construction. 1982.

Sim68 O.J.Dahl, B.Myhrhaug and K.Nygaad. The Simula 67 Common Base Language. Norwegian Computing Centre, Oslo. 1968

Sto76 J.E.Stoy. The Congruence of two Programming Language Definitions. Theoretical Computer Science, Vol 13-2, 1981.

Sto77 J.E.Stoy. Denotational Semantics: The Scott Strachey Approach to Programming Language Theory, The MIT Press, 1977.

Str66 C.Strachey. Towards a Formal Semantics. pp 198-220 in Formal Language Description Languages for Computer Programming. ed. T.B.Steel, North-Holland, Amsterdam, 1966.

Str73 C.Strachey. The Varieties of Programming Languages. Technical Monograph PRG-10, Programming Research Group, University of Oxford, 1973.

Suf77 B.Sufrin. FORM: A Text Editor. Department of Computing Science, Essex University, 1977.

Suf78a B.Sufrin. LEXGEN: A Lexical Analyser Generator. Department of Computing Science, Essex University, 1978.

Suf78b B.Sufrin. LL1: A Parser Generator. Department of Computing Science, Essex University, 1978.

Ten76 R.D.Tennent. The Denotational Semantics of Programming Languages. pp 437-453 in CACM 19-8, 1976

Wij75 A.van Wijngaarden et al. Revised Report on the Algorithmic Language ALGOL68, pp 1-236 in Acta Informatica 5, 1975.

APPENDIX A

The Implementation

We briefly describe the implementation of a system named ISL (Implementation Semantic Language), which is the result of the programming efforts during our research. This major software project was thought, in the beginning, to be the basis of our contribution. Such tremendous programming effort became an anonymous contribution in the light of the developments that were to come, All transformations described have been automatically carried out by ISL, all final CGPs have been successfully compiled in BCPL, loaded with the machine interface provided by the ISL library and tested accordingly.

A.1 Early History

The ideas for a system which could serve the purpose of aiding in the construction of the code generation phase of a compiler, originated in the spring of 1977. Two main projects were developed: An interpreter for a toy language, with an ALGOL60 level of difficulty and the denotational semantics for the same language, whose original semantic description was informally described in English.

It was understood at the time, and still remains the corner stone of the research, that denotational semantics has abstracted, at an appropriate level, the behaviour of a program, and both, semantic description and implementation, were addressing the same issue at different levels of abstraction. When both projects were completed we had:

LEX	scanner
SYN	parser
DS	interpreter

The left hand side, above, consists of language descriptions with different underlying theories. The second column consists of programs written in BCPL. At Essex, there are two systems to aid in the construction of these programs: One, a lexical analyser generator LEXGEN [Suf78a], the other a parser generator LL1 [Suf78b]. Both systems generate BCPL programs. While staring at the semantic description and at the interpreter, hand written in BCPL, we could picture a way to go from one into the other, a parallel was immediately drawn; what was required was the missing generator.

A.2 **Pilot Project**

This thesis grew out of this idea. A main design decision was taken at an early stage. We were going to address the problem of automatically generating a code generator, as opposed to an interpreter; firstly, because it was thought to be harder and secondly, because we envisaged efficiency in the target code.

The implementation of ISL began in 1979. The original idea was to produce a system to transform 'simple programming language specifications' into BCPL programs constituting the code generation phase of a compiler for the given language. But, however 'simple' the languages, this was considered a major undertaking and we needed some experimenting in order to get experience in the problems to come. Therefore the first pilot project was to take the early denotational description and hand write, a stepwise transformation into a compiler that we also quickly hand coded as target. This project was immediately followed by a language oriented automatic transformation, which gave the necessary insight into the problems to come. Even though the system was oriented to transform only one language, an internal representation of the source semantic specification was required, We needed a parser and a type checker for the semantic metalanguage. The latter was not only required as an aid in writing semantic descriptions; it was already understood at that time that such information would be paramount in the transformations to follow. One can understand why P. Mosses first SIS system did not have type checking, because his transformations do not address the semantic objects described, only the three main syntactic constructions of the lambda calculus. But we wished to recognise semantic objects like environments and continuations; hence, type checking was a main requirement for subsequent use in the transformation process.

A.3 The ISL System

At that time (fall of 1979), we had such a large program in BCPL, that an overlay system was immediately designed, and subsequently, we rewrote and partitioned the early system in different modules:

ISLINI - Initialisation.

Any language processor has some common activity at initialisation time such as: opening files for input, output and listing, predefine names and initialise stack. The ISL system requires it and also all compilers generated with the aid of LEXGEN, LL1 and ISL. This module constitutes precisely this process. To aid in the initialisation parts of the generated compilers, an auxiliary library named PROLIB was implemented out of its code. PROLIB provides the front-end process to any language processor and has been used for several years by the author and his students.

ISLPAR - The parser.

Generated with LEXGEN and LL1. The input language was not fixed for a long time, so these automatic aids proved to be invaluable. A description of the concrete syntax is included at the end of this section.

ISLDES - Description phase.

A semantic description consists of a set of mutually recursive equations and definitions whose order is irrelevant. Hence, a complete separate pass over the internal tree representation was designed to fix forward references.

ISLEXP - Type checking expressions.

Because of the complex functionality definitions, this module proved to have a level of difficulty equivalent to an ALGOL68 type checker. It took more than 6 man-months to develop.

ISLDDT - Interactive debugging.

Invaluable aid which can be interleaved between any other module, allowing the scrutiny of every piece of information in the internal tree representation and interfaces to DEC-10-DDT for machine code debugging.

ISLOUT - Pretty printing.

Used in combination with the text editor FORM [Suf77] to produce all snapshots of this thesis.

At this stage we were able to construct a well formed internal representation of a semantic specification, which closely followed the theoretical ideas of the Advice Taker: a property list for any expression about which information is known that does not follow from its structure [McC60],

The next stage, was to perform the transformations. This was done by recursively walking and transforming in core the internal tree representation. One different module was written for every different activity. So the system became a collection of 'experts' which performed the different levels of transformations. Each module constituted an active filter, a tree to tree translator. In between each module, ISLOUT or ISLDDT could be called to perform intermediate listings or debugging. The final version consists of the following modules:

ISLTR1 - Normalisation
ISLTR2 - State Analysis
ISLTR3 - Syntactic Transformations
ISLTR4 - Splitting Continuations
ISLTR5 - Destination Analysis
ISLTR6 - Continuation Analysis
ISLTR7 - Environment Analysis
ISLTR8 - Optimising Continuations
ISLTR9 - Optimising Transformations
ISLTRA - BCPL

As it can be seen, they correspond to each stage described in Chapters 3 to 6. They are processed in strict sequence and because of the pragmatics of their activity, the order can not be altered.

Being an overlayed system, there is no theoretical limit in the number of 'experts' that we could add or interleave. If in the future, we wish to include a different denotational feature, or a different implementation technique, then it is as simple as adding another module. The size of each module is, in general, a couple of pages of BCPL code. They are short because of the existence of a library ISLLIB which contains common code to all modules.

All the ISL system consists of 10.700 lines of pure (comments are not counted) BCPL code.

A.4 Concrete Syntax of WFFs

```
Comment         ::=  COMMEN
EndOfLine       ::=  EOL
Numeral         ::=  NUMERAL
Quotation       ::=  QUOTATION
CurlyName       ::=  CURNAM
DomainName      ::=  SEMNAM
DomainToken     ::=  SEMNAM | SEMLST
SyntacticName   ::=  SYNNAM
SemanticName    ::=  SEMNAM
SyntacticToken  ::=  SYNNAM | SYNVAR | QUOTATION | TERMINAL
SemanticToken   ::=  SEMNAM | SEMVAR | SEMLST | CURNAM | SynInsideSem
SynInsideSem    ::=  "[|" Syntax "|]"
Isl             ::=  "Isl_eol_comment"
Syn             ::=  "Syn_eol_comment"
Sem             ::=  "Sem_eol_comment"
End             ::=  "End_eol_comment"
S               ::=  IslSpe
IslSpe          ::=  Isl IslBod End
IslBod          ::=  [ IslSpe | SynSpe | SemSpe | GetSpe ]*
SemSpe          ::=  Sem SemDef*
SynSpe          ::=  Syn SynDef*
GetSpe          ::=  "Get" Quotation IslBod
SynDef          ::=  SyntacticName [ SynNamDom | SynNamPro ]
SynNamDom       ::=  : DomainName . Comment
SynNamPro       ::=  "::=" SynAlt
SynAlt          ::=  Syntax [ "|" EndOfLine? Syntax ]*
Syntax          ::=  SyntacticToken SyntacticToken*
SemDef          ::=  SemanticName SemDefNam | CurlyName SemDefCur
SemDefNam       ::=  SemUnd | SemDefSel | SemDom | SemPriEqu | SemDefNamEqu |
                     SemDefNamCol
SemUnd          ::=  "?" . Comment
SemDefSel       ::=  "==" SemLam . Comment
SemDom          ::=  . Comment
SemPriEqu       ::=  LamLst = SemEqu . Comment
SemDefNamEqu    ::=  = [ SemDomFun | SemNulEqu ]
SemDomFun       ::=  DomBra . Comment
SemNulEqu       ::=  SemEqu . Comment
SemDefNamCol    ::=  : [ SemNamFun | SemDefNamColNam ]
SemNamFun       ::=  DomBra . Comment
SemDefNamColNam::=  SemanticName [ SemNamDom | SemNamDomFun ]
SemNamDom       ::=  . Comment
SemNamDomFun    ::=  = DomBas . Comment
SemDefCur       ::=  SemCurUnd | SemCurFun | SemCurEqu
SemCurUnd       ::=  "?" . Comment
SemCurFun       ::=  : DomBas . Comment
SemCurEqu       ::=  LamLst? = SemEqu . Comment
LamLst          ::=  LamBas LamBas*
LamBas          ::=  LamTok | < [ LamTok [ , [ LamTok | "..." ] ]* ]? >
LamTok          ::=  SemanticToken [ : DomBas ]?
DomExp          ::=  DomEx1 [ "->" DomEx1 ]*
DomEx1          ::=  DomEx2 [ + DomEx2 ]*
```

```
DomEx2        ::=  DomEx3 [ . DomEx3 ]*
DomEx3        ::=  { SemanticName } | DomBas
DomBas        ::=  DomainToken | DomBra
DomBra        ::=  "[" DomExp "]" "*"?
SemEqu        ::=  SemExp
SemExp        ::=  SemE00 ["Where" LamLst = SemExp ]?
SemE00        ::=  SemE01 [ OprPrim SemE01 ]*
SemE01        ::=  SemE02 | SemLam
SemE02        ::=  SemE03 [ OprCond SemExp , SemE01 ]?
SemE03        ::=  SemE04 [ OprInte DomBas ]*
SemE04        ::=  SemE05 [ OprDoma DomBas ]*
SemE05        ::=  SemE06 [ OprDyad SemE06 ]*
SemE06        ::=  SemE07 [ OprRela SemE07 ]?
SemE07        ::=  SemE08 [ OprList SemE08 ]*
SemE08        ::=  OprMona? SemE09
SemE09        ::=  SemE11 [ SemE10 "..." ? ]*
SemE10        ::=  SemE11 | SemPostAss
SemE11        ::=  ( SemExp ) | { SemExp } | Numeral | SemanticToken |
                   Quotation | SemTup
SemLam        ::=  "Lam" LamLst . [ SemE01 | "..." SemE01 ]
SemPostAss    ::=  "[" SemExp / SemExp "]"
SemTup        ::=  < [ SemExp [ , [ SemExp | "..." ] ]* ]? >
OprPrim       ::=  ; | * | @ | "=>"
OprCond       ::=  "->"
OprInte       ::=  "?" | "??"
OprDoma       ::=  "|" | "In"
OprDyad       ::=  + | -
OprRela       ::=  = | "Eq" | "Ne" | "Ls" | "Le" | "Gr" | "Ge"
OprList       ::=  ! | ^ | %
OprMona       ::=  #
```

Concrete symbol	Snapshot form	Comment
[|	**[**	Open Syntax
|]	**]**	Close Syntax
Lam	λ	Lambda
*	$\underline{*}$	Composition with side effects
@	$\underline{+}$	Composition without side effects
;	$\underline{o}$	Composition
->	$\rightarrow$	Conditional and Function Constructor
.	x	Cross product
!	†	Chop
^	$\downarrow$	Select
SEL==λi.e	iSEL==e	Selector

APPENDIX B

Transformation Rules

Rules marked with a * next to their number, are those redefined or extended.

B.1 Normalisation

$v[s_1]p=e_1$. ... $v[s_n]p=e_n$.	=>	**if** $n>1$ **let** v node p **be** **switchon** type^node **into** { **case** $[s_1]$: e_1; **endcase** ... **case** $[s_n]$: e_n; **endcase** } **if** $n=1$ **let** v node p **be** e_1	[R1.1]
λip.e	=>	λi.λp.e	[R1.2]
e_0 Where p=e_1	=>	{ **let** p=e_1 ; e_0 }	[R1.3]
Cond<e_1, e_2>e_0	=>	e_0?T $\rightarrow$ (e_0|T $\rightarrow$ e_1, e_2), Wrong	[R1.4]
Scond<e_1, e_2>e_0	=>	e_0??T $\rightarrow$ (e_0|T $\rightarrow$ e_1, e_2),Wrong	[R1.5]

B.2 State Analysis

when i:STA	λi.i	=>	{}	[R2.1]
when i:STA	λi.e	=>	e	[R2.2]
when i:STA	e_0i	=>	e_0	[R2.3]
when e:STA	e_0e	=>	{ e In COD; e_0 }	[R2.4]
when i:STA	e$\rightarrow e_1$,i	=>	e$\rightarrow e_1$,{}	[R2.5]
when i:STA	e$\rightarrow$i,e_1	=>	e$\rightarrow${},e_1	[R2.6]
when i:STA	Strict(λi.e)	=>	λi.e	[R2.7]
	Is	=>	{}	[R2.8]
when e_0:[STA$\rightarrow D_1$]	e_0 $\underline{o}$ e_1	=>	C	[R2.9]

where C = { e_0; e_1 } if D_1=STA or D_1=ANS (i.e: e_0:COD)
C = e_1(e_0 In D_1) otherwise

$e_0 \underline{+} e_1$ => (e_1 **In** $[D \to D_2]$)(e_0 **In** D) [R2.10]
when for any domain D and D_2 e_0:[STA→D] and e_1:[D→[STA→D_2]]

$e_0 \underline{*} e_1$ => (e_0 **In** $[D_1 \to D]$)$\underline{o}$(e_1 **In** $[D \to D_3]$) [R2.11]
when for any domains D, D_1 and D_3 e_0:[D_1→[DxSTA]] and e_1:[D→[STA→D_3]]

when i:STA $i[e_1/e_2]$ => **trans.update**(e_1, e_2) [R2.12]

when i:STA i(e) => **trans.load**(DOM(e), e) **In** REG [R2.13]

when i:STA <e_0, i> => e_0 [R2.14]

when e_1:STA <e_0, e_1> => { e_0 ; e_1 } [R2.15]

B.3 **Syntactic Transformations**

let $vi_1 \ldots i_n$ **be** C => **let** v(i_1,..., i_n) **be** C [R3.1]

$e_0 e_1 \ldots e_n$ => e_0(e_1, ..., e_n) [R3.2]

(λi.e)(e_1) => { **let** i=e_1; e } [R3.3]

(λi.e)(e_1)(e_2) => { **let** i=e_1; e(e_2) } [R3.4]

when not i:COD (λi.e){C; e_1} => C; { **let** i=e_1; e } [R3.5]

Strict(e) => e [R3.6]*
when e:TEM and there is no e_2 such that Non-Strict(e_2):TEM

case [s_1 ... s_n]: e => **case** [s_1 ... s_n]: [R3.7]
{ **let** n = open.node(node)
e
freevec(n)
} **rename** [s_1]=>n!inx, [s_n]=>n!!n,
n=>node.vec

ee_1 or (λ i. ... ei ...)e_1 => e(Thunk(e_1)) or e(λ i. ... ei ...)(Thunk(e_1)) [R3.8]
where $e_1 \neq$ i
when e:TEM and there is an e_2 such that Non-Strict(e_2):TEM

B.4 Splitting Continuations

when e_0:KON $\quad e_0(e_1) \quad$ => $\quad e_1$; e_0 In COD $\quad$ [R4.1]

when (λi.e_1):KON $\quad e_0$(P, λi.e_1) $\quad$ => $\quad e_0$(P).cont.(e_1).dest.(i)
i is bound in e_1 $\quad$ [R4.2]

when i:KON=[d→COD] $\quad$ let v(D, i) be C $\quad$ => $\quad$ let v(D).cont.(i In COD)
.dest.(? In d)
be C $\quad$ [R4.3]

when i:KON=[d→COD] $\quad$ e(P, i) $\quad$ => $\quad$ e(P).cont.(i In COD)
.dest.(? In d) $\quad$ [R4.4]

when i:COD $\quad$ let v(D, i) be C $\quad$ => $\quad$ let v(D).cont.(i) be C $\quad$ [R4.5]

when e_1:COD $\quad e_0$(P, e_1) $\quad$ => $\quad e_0$(P).cont.(e_1) $\quad$ [R4.6]

when e_0:KON $\quad e_0$(P, λi. ...e) $\quad$ => $\quad$ for inx=1 to !node.vec-1
do e_0(P).cont.(...).dest.(i)
unless !node.vec=0 do e $\quad$ [R4.7]

B.5 Destination Analysis

when v(D):REG $\quad$ let v(D) be C $\quad$ => $\quad$ let v(D).dest.(reg) In COD
be C $\quad$ [R5.1]

when not v(D):REG $\quad$ let v(D) be C $\quad$ => $\quad$ let v(D) be [first.reg/reg]C $\quad$ [R5.2]

when (e(P):REG or e(P):THU) and not e:ENV $\quad$ e(P) $\quad$ => $\quad$ e(P).dest.(reg) In COD $\quad$ [R5.3]*

when (e(P):REG or e(P):THU) and not e:ENV, rename i=>(i$\underline{=}a_k$)→reg+k, reg $\quad$ { let i=e(P); C } $\quad$ => $\quad$ { e(P).dest.(i) In COD; C } $\quad$ [R5.4]*

when $e_1 \underline{=}$ e(P)A.dest.(E) $\quad e_2(P_0, e_1, P_1)$ $\quad$ => $\quad e_1$
$e_2(P_0$, E, $P_1)$ $\quad$ [R5.5]

when i:REG and P not null $\quad$ e(P, i) $\quad$ => $\quad$ e(P).dest.(i) In COD $\quad$ [R5.6]

```
                {C_0; e; C_1 }  | => |  {C_0; C; C_1 }
                     or         |    |       or
                 e_0 → e, e_2   | => |  e_0 → C, e_2                      [R5.7]*
                     or         |    |       or
                 e_0 → e_1, e   | => |  e_0 → e_1, C
when    NeedsLoad               |    |  where C = trans.load(DOM(e), e) In REG
where   NeedsLoad = e:REG and not e:COD and
                ((e=i and not i:ENV) or e=E_0!E_1 or e=#e_0 or e=n or e=q)

                            e      =>     [first.reg/reg]e                 [R5.8]*
when e≠i and (e:TEM or e:THU)

when    e:TEM and i:REG     e      =>     [first.par/i]e_1 In DOM(e)        [R5.9]
where   e = λi.e_1

when e:TEM or e:THU         e      =>     trans.load(DOM(e), e) In REG  [R5.10]*

when e:TEM                e(P)     =>     e(P).dest.(first.reg)            [R5.11]

let v(D).cont.(P).dest.(?:d)  |    |  let v(D).cont.(P).dest.(reg)
                        be C  | => |  be C                                 [R5.12]
when dCREG                    |    |

    e.cont.(P).dest.(?:d)      =>     e.cont.(P).dest.(reg)                [R5.13]
when dCREG

      e.cont.(P).dest.(i)  | => |  e.cont.(P).dest.(i)                     [R5.14]
when i:REG                 |    |  rename i=>(i=a_k)→reg+k, reg

    e.cont.(P).dest.(?:d)      =>     e.cont.(P).dest.(first.reg)          [R5.15]
when e:TEM and dCREG

                                |    |  { let old.env = this.env
                                |    |    let old.off = this.off
      for I=1 to E_0 do C_1     | => |    for I=1 to E_0 do C_4            [R5.16]
      unless E_1 do C_2         |    |    unless E_1 do C_5
                                |    |    reset(old.env)
                                |    |  }
when    C_1 = e_1(P_1).cont.(...).dest.(I_1)
        C_2 = { C_7; e_2(P_2).cont.(C_3).dest.(I_2); C_8 }
        C_3 = e_3(<I_1,...,I_2>)
where   C_4 = e_1(P_1).cont.(trans.dump(I_1); ...).dest.(I_1)
        C_5 = { C_7; e_2(P_2).cont.(C_6).dest.(I_2); C_8 }
        C_6 = e_3(old.off).dest.(I_1)
        any C_7, C_8

                                |    |  { let dmp.loc = trans.dump(I_1)
       for I=1 to E do C_1      | => |    for I=1 to E do C_3              [R5.17]
       C_2                      |    |    C_2
                                |    |  }
when    C_1 = e_1(P_1, I_1, P_2)A
        I_1:REG and any C_2
where   C_3 = { C_1; trans.load(DOM(I_1), dmp.loc).dest.(I_1) }
```

```
                                   __        __
         let v(D0, i, D1) be C   |  =>  |  let v(D0, i, D1) be C             [R5.18]
when i:REG                     __|      |__rename i=>reg

                     e0oe1         =>     trans.skip.if(i.skipXX,e0,e1)     [R5.19]
when     e0:REG and e1:REG and o is one of: =, Eq, Ne, Ls, Le, Gr, Ge
where    XX is respectively one of: EQ, EQ, NE, LT, LE, GT, GE

                                 __        __
                                   |      |  Strict(λi.
                                   |      |  { trans.call(i).dest.(first.reg)
                     Strict(e)     |  =>  |    [first.reg/i]e1                [R5.20]
where    e = λi.e1               __|      |__})
when     e:TEM and there is an e2 such that Non-Strict(e2):TEM

                                 __        __
                                   |      |  { trans.call(e).dest.(first.reg)
         e().cont.(e1).dest.(i)    |  =>  |    [first.reg/i]e1                [R5.21]
when e:THU                       __|      |__}
```

B.6 Continuation Analysis

```
                          {C0; i; C1 }__  |  =>  |__{C0; C; C1 }
                                or        |      |      or
when i:COD                 e0 → i, e2     |  =>  |  e0 → C, e2                 [R6.1]
                                or        |      |      or
                           e0 → e1, i     |  =>  |  e0 → e1, C
                                        __|      |__where C = trans.jump.to(i)

                          e0 → e1, e2       =>      C                         [R6.2]*
when    (E0IsDes or E0IsIde or E0IsSkp) and i::REG
where   C  = { C1; C2; C3; C4; C5; C6; C7; C8; C9 }
        C1 = NoEndCo → null, let econd.code = forward(COD)
        C2 = NoFalse → null, let fcond.code = forward(COD)
        C3 = E0IsIde → null,
             Reverse and E0IsInt → trans.skip.if.not.in(P),
             Reverse and E0IsDya → trans.skip.if(ReveDya(I), P),
             e0
        C4 = E0IsSkp → trans.jump.to(FalseCo), JumpRut(i, FalseCo)
        C5 = Reverse → null, e1
        C6 = NoEndCo → null, trans.jump.to(econd.code)
        C7 = NoFalse → null, fix.here(fcond.code)
        C8 = E2IsJmp → null, e2
        C9 = NoEndCo → null, fix.here(econd.code)
        E0IsDes = e0=e(P).dest.(i)A
        E0IsIde = e0=i
        E0IsInt = e0=trans.skip.if.in(P)
        E0IsDya = e0=trans.skip.if(I, P)
        E0IsSkp = E0IsInt or E0IsDya
        E1IsJmp = e1=trans.jump.to(i1)
        E2IsJmp = e2=trans.jump.to(i2)
        E2IsNul = e2={}
```

```
        Reverse = E2IsNul and E1IsJmp
        NoFalse = Reverse or E2IsJmp
        NoEndCo = E2IsNul or E2IsJmp or WillJump(e_1)
        JumpRut = Reverse → trans.jump.if.true, trans.jump.if.false
        FalseCo = Reverse → i_1, E2IsJmp → i_2, fcond.code
        HasCont(e)=TRUE   if e contains continuations which will jump
        HasCont(e)=FALSE otherwise

when i:COD          Fix(λi.e)  | => | { let i = here(COD); e }              [R6.3]
                               |    |__rename i=>restart.code

                               |    | { let ntry.code = forward(DOM(e_1))
                               |    |   let exit.code = forward(COD)
                               |    |   let skip.code = forward(COD)
                               |    |   trans.jump.to(skip.code)
                               |    |   trans.entry(ntry.code, node)
when e_1:TEM   e(P_0, e_1, P_1)A  | => |   e_1                                [R6.4]
                               |    |   trans.exit(exit.code, node)
                               |    |   fix.here(skip.code)
                               |    |   e(P_0, ntry.code, P_1)A
                               |    |__}

when i:TEM          Fix(λi.e)    =>    [ntry.code/i]e                      [R6.5]

when e:TEM               e(P)A   =>    trans.call(e, P)A                   [R6.6]

when i:REG                 i?d   =>    trans.skip.if.in(i, d)              [R6.7]

                             e   =>    check.if.in(i, d)                   [R6.8]
when (e ≡ i?d and not i:REG) or e ≡ i??d

                               |    | { let continue = forward(COD)
                               |    |   e(P).cont.(continue)A
when e_1≠i   e(P).cont.(e_1)A  | => |   fix.here(continue)                 [R6.9]
                               |    |   e_1
                               |    |__}

     let v(D).cont.(i)A be C   | => | let v(D).cont.(i)A be C             [R6.10]
                               |    |__rename i=>continue

                               |    | { let ntry.code = forward(DOM(e_1))
                               |    |   let exit.code = forward(COD)
                               |    |   let skip.code = forward(COD)
        e(P_0, λi.e_1, P_1)A   | => |   trans.jump.to(skip.code)           [R6.11]
when i:COD and λi.e_1:TEM      |    |   trans.entry(ntry.code, node)
                               |    |   [exit.code/i]e_1
                               |    |   trans.exit(exit.code, node)
                               |    |   fix.here(skip.code)
                               |    |   e(P_0, ntry.code, P_1)A
                               |    |__}
```

```
                                          __       __
                                           |      |  e3
        e(P0, e0([e1/e2], P1)A             | =>   |  e(P0, e0([ntry.code/e2], P1)A [R6.12]*
                                           |      |  where except for the last statement
  when e1:TEM or e1:THU                  __|      |__e3 is the same as R6.11 or R6.21

                  <e1,...,en>        =>        { C1; C2; C3; C4 }                [R6.13]
  where  C1 = let c = E
         C2 = for inx=1 to s-1 do Cf
         C3 = unless s=0 do Cu
         C4 = freevec(c)
         ei:D for i=1 to n

            __
           |   { e1; fix.with(c!inx,r) }  if DCTEM
        Cf=|   { fix.here(c!inx); e1   }  if DCCOD
           |__  c!inx := e1               otherwise
            __
           |   { en; fix.with(c!s,r) }    if DCTEM
        Cu=|   { fix.here(c!s); en   }    if DCCOD
           |__  c!s   := en               otherwise
  rename  s=>!node.vec
          c=>DC[COD+TEM]>code.vec, cons.vec
          E=>DC[COD+TEM]>forward.vec(s, D), newvec(s)
          r=destination of ei

                e0(P0, e1, P1)A      =>      { C1; C2; C3; C5; C4}              [R6.14]
  when    e1:D* and not DCCOD
  where   C1, C2, C3 and C4 are inherited from the transformation of e1 as a
          result of R6.13
          C5 = e0(P0, cons.vec, P1)A
                                         __       __
                                           |     |  { C1
                                           |     |    let s = forward(COD)
                                           |     |    trans.jump.to(s)
                e0(P0, e1, P1)A            | =>  |    C2; C3                      [R6.15]
                                           |     |    fix.here(s)
                                           |     |    e0(P0, code.vec, P1)A
                                           |     |    C4
  when     e1:COD*                       __|     |__
  where    C1, C2, C3 and C4 same as R6.14
  rename   s=>skip.code

  Fix(λ<i1,...,in>.{ C0;e1 })    =>     { C1; C0; C2; C3; C5; C4 }              [R6.16]
  when     e1:COD* and e1=<e1,...,en>
  where    C5 = e(P0, code.vec, P1)A
           C1, C2, C3 and C4 same as R6.14
  rename   i1=>code.vec!inx, in=>code.vec!!node.vec

  when    e:COD*     n↓e or e↓n     =>     C ; e                                [R6.17]
  where   C = null if n=1
          C = trans.jump.to(code.vec!n) otherwise
```

```
                        __         __
                          |       |  { let continue = forward(COD)
        e(P).cont.(...)A  |  =>   |    e(P).cont.(continue)A              [R6.18]
                          |       |    fix.here(continue)
                        __|       |__}

                        __         __
                          |       |  { let ntry.code = forward(DOM(e1))
                          |       |    let exit.code = forward(COD)
                          |       |    let skip.code = forward(COD)
                          |       |    trans.jump.to(skip.code)
                          |       |    trans.thunk.entry(ntry.code, node)
when e1:THU  e(P0, e1, P1)A  |  =>  |    e2                                 [R6.19]
                          |       |    trans.thunk.exit(exit.code, node)
                          |       |    fix.here(skip.code)
                          |       |    e(P0, ntry.code, P1)A
                          |       |  } where e2=e3 if e1=Thunk(e3)
                        __|       |__        e2=e1 otherwise

              Strict(C)     =>     { C1; C3; C4 }                       [R6.20]
when    C:TEM and C = { C1; C2 }
        C1= trans.call(P).dest.(first.reg)
        C2= any
where
           __
          |  { let ntry.code = forward(DOM(first.reg))
          |      let exit.code = forward(COD)
          |      let skip.code = forward(COD)
          |      trans.jump.to(skip.code)
      C3= <      trans.thunk.entry(ntry.code, node)
          |      trans.load(DOM(first.reg), first.reg).dest.(first.reg)
          |      trans.thunk.exit(exit.code, node)
          |      fix.here(skip.code)
          |__}
      C4= {ntry.code/first.reg}C2

                        __         __
                          |       |  { let ntry.code = forward(DOM(e1))
                          |       |    let exit.code = forward(COD)
                          |       |    let skip.code = forward(COD)
                          |       |    trans.jump.to(skip.code)
                          |       |    trans.thunk.entry(ntry.code, node)
     e(P0, λi.e1, P1)A    |  =>   |    [exit.code/i]e1                    [R6.21]
                          |       |    trans.thunk.exit(exit.code, node)
                          |       |    fix.here(skip.code)
                          |       |    e(P0, ntry.code, P1)A
when λi.e1:THU          __|       |__}
```

B.7 Environment Analysis

```
                                  __          __
                                    |        |  { let old.env=this.env
                                    |        |    e
               e0(P0, e, P1)A       |  =>    |    e0(P0, P1)A                    [R7.1]
                                    |        |    reset(old.env)
when e:ENV and e≠i                __|        |__}

when i:ENV      i([e1/e2])         =>      declare(DOM(e1), e1, e2)              [R7.2]
                                  __          __
                                    |        |  { let old.env = this.env
                { let i = e         |        |    e
when i:ENV        C                 |  =>    |    C                              [R7.3]
                }                   |        |    reset(old.env)
                                  __|        |__}

when i:ENV              i(P)A      =>      look.up(P)A                           [R7.4]

when i:ENV let v(P0, i, P1)A       =>      let v(P0, P1)A                        [R7.5]

when i:ENV      e(P0, i, P1)A      =>      e(P0, P1)A                            [R7.6]

when   i:ENV            i(P)       =>      { C1; C2 }                            [R7.7]
where  P = [e1/e2],...,[e3/e4]
       C1= for inx=1 to s-1 do declare(e2, e1)
       C2= unless s=0 do declare(e4, e3)
rename s=>!node.vec
                                  __          __
                                    |        |  { let old.env = this.env
              { e(P)A.dest.(i)      |        |    e(P)A
when i:ENV      C                   |  =>    |    C                              [R7.8]
              }                     |        |    reset(old.env)
                                  __|        |__}

           for I=1 to E           __|  =>    |  for I=1 to E                     [R7.9]
           do e(P)A.dest.(i)      __|        |__do e(P)A

when i:ENV      { C0; i; C1 }      =>      { C0; C1 }                            [R7.10]

when dCENV  e(P)A.dest.(?:d)       =>      e(P)A                                 [R7.11]
```

B.8 Optimising Continuations

```
let v(D).cont.(I)A be              |    let v(D).cont.(I, jump)A be
switchon E into                    |    switchon E into
{ case [s]: { C0; C1; C2 }         | => { case [s]: { C0; C3; C2 }        [R8.1]
            endcase                |                endcase
  ...                              |      ...
}                                  |    }
```

when $C_1 \equiv$ e(P).cont.(I)A or $C_1 \equiv$e(P, I)A
e is not one of: fix.here, trans.load, trans.entry, trans.thunk.entry, trans.exit, trans.thunk.exit, trans.jump.if.true or trans.jump.if.false.
where C_3 = e(P).cont.(I, boo)A or (depending on C_1) $C_3 \equiv$e(P, I, boo)A
boo= true.jump if C_2:COD
boo= jump otherwise

```
{ e0                 |    { e0
  let I0 = E         |      let I0 = E
  C0; C1; C2         | =>   C0; C3; C2                [R8.2]
  I1(I0)             |      I1(I0)
  e1                 |      e1
}                    |    }
```

when $C_1 \equiv$ e(P).cont.(I)A or $C_1 \equiv$ e(P, I)A
and (fixed or fixing or global)
and I_1 is one of: fix.here, trans.exit or trans.thunk.exit.
where C_3 = e(P).cont.(I, boo)A or (depending on C_1) $C_3 \equiv$e(P, I, boo)A
fixed = $I \equiv I_0$ and $E \equiv$here(P)
fixing= $I \equiv I_0$ and $E \equiv$forward(P)
global= I free in the procedure where this transformation is applied.
boo = true.jump if (fixing and C_2:COD) or fixed or global
boo = false.jump otherwise

or

when C_1 is in the context of:
for I_2=e_2 to e_3 do { fix.here(I_0); C_0; C_1; C_2 }
$I_0 \equiv$ e!I_2 and $I \equiv$ e!(I_2+1)
where C_3 as before
boo = true.jump if C_2:COD, boo = false.jump otherwise

```
{ C1; C2; C3 }    =>    {C1; C3 }                     [R8.3]
```

when $C_2 \equiv$ trans.jump.to(i, false.jump)

```
J1(I0, I1)    |    test E2
J0(I2, E2)    | => then { J2(I0,I2); J0(I1,E1 }       [R8.4]
              |    or   J1(I0, I1)
```

when $J_0 \equiv$ trans.jump.to
$J_1 \equiv$ trans.jump.if.false or $J_1 \equiv$ trans.jump.if true
$E_2 \equiv$ jump or $E_2 \equiv$ not jump or $E_2 \equiv$ true.jump
where $J_2 \equiv$ reverse of J_1
$E_1 \equiv$ the result of R8.1 or R8.2

```
     for I=e0 to e1-1
     do C1
     unless e1=0               =>   for I=e0 to e1 do C1        [R8.5]
     do C2
when C1 ≡ [I/e1]C2
```

B.9 Optimising Transformations

```
                      test E=max.reg
                      then
                      { let old.env = this.env
                        let D = trans.dump(R)
       C     =>         [R/R+1]{D/R}C                             [R9.1]
                        reset(old.env)
                      } or
                      { let nxt = next(R)
                        [nxt/R+1]C
                      } rename D=>dmp.loc
when   C ≡ { C1; C2; C3 } and C2 ≡ e(P)A.dest.(R+1)
       R ≡ reg or R ≡ first.reg
where  E ≡ weight^[s] if C2 ≡ e(P0, [s], P1)A.dest.(R+1) (P contains an [s])
       E = R otherwise
```

```
 { let old.env = this.env
   C1                             { let old.env = this.env
   { let old.env = this.env           C1
     C2                      =>       C2                          [R9.2]
     reset(old.env)                   reset(old.env)
   }                              }
   reset(old.env)
 }
```

```
   trans.load(E, I).dest.(I)   =>   make.type(I, E)               [R9.3]
when E ≢ domain.of(I)

   trans.load(E, I).dest.(I)   =>   {}                            [R9.4]
when E ≡ domain.of(I)

   E0(domain.of(E), E, P)A     =>   E0(domain.of(xx), xx)A        [R9.5]
when E≢I                              where xx = E
```

```
                                  __          __
                                    |        |   { C0
{ C0                                |        |     { let ntry.code = E0
  { let ntry.code = E0              |        |       let exit.code = E1
    let exit.code = E1              |        |       let skip.code = E2
    let skip.code = E2              |        |       { let old.env = this.env
    trans.jump.to(P1)               |        |         let dmp.loc = trans.dump(i)
    trans.thunk.entry(P2)           |   =>   |         trans.jump.to(P1)            [R9.6]
    C1                              |        |         trans.thunk.entry(P2)
    trans.thunk.exit(P3)            |        |         {dmp.loc/i}C1
    C2                              |        |         trans.thunk.exit(P3)
  }                                 |        |         {dmp.loc/i}C2
}                                   |        |         reset(old.env)
when i:REG                          |        |       }
   and i is free in C1            __|        |__}
```

B.10 BCPL

```
                 Every [s]‾‾|  =>  |‾‾replaced by its appropriate        [RA.1]
                          __|      |__'tag' or 'selector'

Every 'curly' valuator v‾‾|  =>  |‾‾respectively replaced by           [RA.2]
    and every domain d__|      |__trans.v and D..d

                { C0           ‾‾|      |‾‾{ C0
                  e0>e1,e2       |  =>  |     test e0 then e1 or e2     [RA.3]
                  C1             |      |     C1
                }              __|      |__}

                { C0           ‾‾|      |‾‾{ C0
                  e0>e1,{}       |  =>  |     if e0 then e1             [RA.4]
                  C1             |      |     C1
                }              __|      |__}

                { C0           ‾‾|      |‾‾{ C0
                  e0>{},e2       |  =>  |     unless e0 do e2           [RA.5]
                  C1             |      |     C1
                }              __|      |__}

                n↓e or e↓n         =>     pn^e where pn is a 'selector'  [RA.6]

         let v(P) be C         =>     let v(P)=valof C               [RA.7]
when not v(P):COD

and for every case inside C above:

       case I: E; endcase      =>     case I: resultis E             [RA.8]

                #[s1...sn]         =>     size^[s1...sn]                 [RA.9]
```

when e_1:INT $e_0(P_0, e_1, P_1)A$ => $e_0(P_0, e_2, P_1)A$ where e_2=make.num(e_1) [RA.10]

B.11 Cross Reference

Rule	Def.	Use
Normalisation		
R1.1	29	30, 43, 61, 62, 65, 66, 73, 89, 92, 93, 94, 95, 99, 117, 124, 137, 145, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 227, 228, 229, 230, 231, 232, 233, 234, 235
R1.2	30	61, 65, 66, 137, 145, 221, 222, 223, 235
R1.3	30	117, 213, 215, 217, 233
R1.4	80	99, 213, 217, 219
R1.5	80	not used
State Analysis		
R2.1	34	36
R2.2	34	36, 61, 65, 66, 73, 93, 94, 95
R2.3	34	36
R2.4	34	36
R2.5	35	36
R2.6	35	not used
R2.7	37	36, 61, 66, 73, 93, 94, 95
R2.8	40	43, 65, 92, 93
R2.9	40	43, 58, 61, 62, 65, 66, 73, 89, 92, 93, 94, 95
R2.10	41	43
R2.11	58	61, 62, 65, 66, 73, 89, 92, 93, 94, 95
R2.12	59	61, 65, 66
R2.13	60	61, 66
R2.14	60	61, 66, 73, 93, 94, 95
R2.15	61	66
Syntactic Transformations		
R3.1	44	45, 61, 62, 65, 66, 73, 92, 93, 99, 117, 137, 145, 213, 218, 227, 233, 234, 235
R3.2	44	45, 61, 62, 65, 66, 73, 89, 92, 93, 94, 95, 99, 117, 124, 137, 145, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 227, 228, 229, 230, 231, 232, 233, 234, 235
R3.3	44	45, 61, 62, 65, 66, 73, 89, 92, 93, 94, 95, 117, 137, 145, 213, 215, 217, 233
R3.4	44	137
R3.5	44	45, 65
R3.6	70	73, 93, 95, 135
R3.7	116	117, 124, 215, 229, 230, 233, 234
R3.8	136	137

Splitting Continuations

R4.1	100	101, 122, 125, 145, 216, 218, 221, 222, 223, 227, 228, 229, 230, 233, 234
R4.2	100	101, 121, 125, 145, 213, 214, 215, 216, 217, 218, 219, 220, 221, 227, 228, 229, 230, 231, 232, 233, 234, 235
R4.3	101	122, 145, 218, 227, 233, 234, 235
R4.4	101	125, 145, 218, 219, 220, 221, 222, 223, 227, 228, 229, 231, 232, 233, 234, 235
R4.5	102	101, 121, 213
R4.6	102	101, 121, 122, 125, 213, 214, 215, 216, 217, 222, 223, 229, 230, 234
R4.7	125	129, 230, 234

Destination Analysis

R5.1	47	52, 61, 62, 66, 73, 93, 140, 235
R5.2	47	52, 61, 62, 65, 73, 89, 92, 93, 103, 121, 213, 214, 215, 216, 217
R5.3	48	52, 61, 62, 66, 73, 93, 94, 95, 103, 122, 138, 140, 141, 142, 149, 150, 151, 216, 218, 221, 222, 223, 227, 228, 229, 233, 235
R5.4	48	52, 61, 62, 65, 66, 73, 89, 92, 93, 94, 95, 138, 140, 141, 150
R5.5	49	62
R5.6	50	61, 62, 65, 66, 140
R5.7	50	61, 66, 100, 141, 150, 151, 216, 228, 229
R5.8	71	73, 94, 95, 138, 141, 142, 150, 151, 221, 222, 223, 235
R5.9	72	73, 94, 95, 142, 151, 221, 222, 223, 235
R5.10	72	73, 94, 95, 138, 141, 142, 150, 151, 221, 222, 223, 235
R5.11	73	93, 95, 141
R5.12	102	103, 122, 149, 218, 227, 233, 234, 235
R5.13	102	103, 149, 150, 151, 219, 220, 221, 223, 228, 229, 230, 232, 233, 235
R5.14	102	103, 121, 128, 149, 150, 151, 213, 214, 215, 216, 217, 218, 219, 220, 221, 227, 228, 229, 230, 231, 232, 233, 234, 235
R5.15	102	128, 150, 214, 215, 216, 218, 220, 221, 227, 231, 234, 235
R5.16	126	128, 230
R5.17	126	128, 234
R5.18	127	128, 233, 234, 235
R5.19	127	229
R5.20	138	139, 142
R5.21	145	144, 151

Continuation Analysis

R6.1	53	56, 65, 93, 109, 110, 111, 121, 122, 129, 149, 150, 151, 213, 214, 215, 216, 217, 218, 219, 221, 222, 223, 227, 228, 229, 230, 233, 234
R6.2	54	56, 65, 66, 81, 89, 92, 93, 94, 95, 108, 109, 110, 111, 121, 129, 141, 150, 213, 214, 215, 216, 217, 219, 221, 227, 228, 229, 234
R6.3	55	56, 65, 93, 110, 213, 217
R6.4	75	81, 94, 95, 142
R6.5	76	223
R6.6	76	81, 93, 95, 141, 150, 215, 216, 221, 227, 235
R6.7	78	81, 92, 93, 94, 95, 109, 110, 111, 121, 129, 141, 150, 213, 214, 215, 216, 217, 219, 221, 227, 228, 229, 234
R6.8	80	89
R6.9	105	109, 110, 111, 121, 129, 149, 150, 213, 214, 215, 216, 217, 218, 219, 220, 221, 227, 228, 229, 230, 231, 232, 234, 235
R6.10	106	109, 110, 111, 121, 122, 129, 149, 150, 151, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 227, 228, 229, 230, 231, 232, 233, 234,

|235
R6.11 106 |151, 221, 222, 223, 235
R6.12 106 |146, 151
R6.13 118 |121, 122, 215, 229, 233
R6.14 119 |121
R6.15 119 |121, 229
R6.16 119 |122, 215, 233
R6.17 119 |122, 215, 233
R6.18 129 |230, 234
R6.19 139 |141, 144
R6.20 139 |142
R6.21 146 |144, 150, 151

Environment Analysis
R7.1 83 |86, 92, 94, 95, 110, 111, 142, 151, 214, 220, 221, 222, 223
R7.2 83 |86, 92, 94, 95, 110, 111, 121, 122, 142, 151, 213, 214, 215, 217,
|220, 221, 222, 223, 233
R7.3 83 |121, 122, 213, 215, 217, 233
R7.4 84 |86, 93, 110, 121, 140, 149, 216, 218, 227
R7.5 85 |86, 92, 93, 109, 110, 121, 122, 140, 149, 213, 218, 227, 233, 234,
|235
R7.6 85 |86, 89, 92, 93, 94, 95, 109, 110, 111, 121, 122, 131, 140, 141, 149,
|150, 213, 214, 215, 216, 217, 218, 219, 220, 221, 227, 228, 229, 230,
|231, 232, 233, 234, 235
R7.7 120 |122, 215, 233
R7.8 130 |232, 235
R7.9 130 |131, 234
R7.10 131 |234
R7.11 131 |234, 235

Optimising Continuations
R8.1 107 |109, 110, 111, 121, 122, 149, 150, 151, 213, 214, 215, 216, 217, 218,
|219, 220, 221, 222, 223, 227, 228, 229, 230, 231, 232, 233, 234, 235
R8.2 107 |109, 110, 111, 121, 122, 141, 142, 149, 150, 151, 213, 214, 215, 216,
|217, 218, 219, 220, 221, 222, 223, 227, 228, 229, 230, 231, 232, 233,
|234, 235
R8.3 108 |151
R8.4 109 |217
R8.5 120 |121, 122, 215, 229, 233

Optimising Transformations
R9.1 86 |93, 95, 140, 141, 149, 150, 216, 217, 219, 221, 227, 231
R9.2 87 |142, 151
R9.3 88 |not used
R9.4 88 |216
R9.5 88 |not used
R9.6 140 |142, 151

BCPL
RA.1 63 |65, 66, 89, 92, 93, 94, 95, 109, 110, 111, 121, 122, 140, 141, 142,
|149, 150, 151, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223,
|227, 228, 229, 230, 231, 232, 233, 234, 235
RA.2 63 |65, 66, 89, 92, 93, 94, 95, 109, 110, 111, 121, 122, 140, 141, 142,
|149, 150, 151, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223,

		227, 228, 229, 230, 231, 232, 233, 234, 235
RA.3	88	89
RA.4	88	not used
RA.5	89	not used
RA.6	120	122, 215, 233
RA.7	120	122
RA.8	120	122
RA.9	131	234
RA.10	132	229, 234

APPENDIX C

Operators

C.1 Source

d:D=Any Domain

01:

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

This is the conditional function. An expression $t \rightarrow d_1, d_2$ will take the value d_1 when t is True, d_2 when t is False, $\mathbf{Top}_D$ when t is $\mathbf{Top}_T$ and $\mathbf{Bot}_D$ when t is $\mathbf{Bot}_T$.

02:

$\cdot \Rightarrow \cdot : [[[D \times [D \rightarrow D_1]] \rightarrow D_1]$

$f:[D \rightarrow D_1]$

x:D

$d \Rightarrow \lambda x.fx$ is the same as $(\lambda x.fx)d$

This operator, which reads as 'produce', is the reverse of application, so that we can read equations from left to right.

03:

$\cdot \underline{o} \cdot [[[D \rightarrow D_1] \times [D_1 \rightarrow D_2]] \rightarrow [D \rightarrow D_2]]$

$f:[D \rightarrow D_1]$

$g:[D_1 \rightarrow D_2]$

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

This is the reversed form of the composition operator.

04:

$\cdot \underline{+} \cdot : [[[D_1 \rightarrow D] \times [D \rightarrow [D_1 \rightarrow D_2]]] \rightarrow [D_1 \rightarrow D_2]]$

Not DCSTA

$f:[D_1 \rightarrow D]$

$g:[D \rightarrow [D_1 \rightarrow D_2]]$

$(f \ \underline{+}\ g)d_1 = g(fd_1)d_1$

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

05:

$\cdot \underline{*} \cdot : [[D_1 \rightarrow [D \times D_2]] \times [D \rightarrow [D_2 \rightarrow D_3]]] \rightarrow [D_1 \rightarrow D_3]$

Not DCSTA

$f:[D_1 \rightarrow [D \times D_2]]$

$g:[D \rightarrow [D_2 \rightarrow D_3]]$

$(f \ \underline{*}\ g)d_1 = gdd_2$ where $\langle d, d_2 \rangle = fd_1$

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

06:

$.|.:[D_1 + \ldots + D_n]$

$i:N$ and $1 <= i <= n$

$d|D$ is the projection of d into the subdomain D_i of $[D_1 + \ldots + D_n]$

07:

.In.

$d:D_i$ $i:N$ and $1 <= i <= n$

d In $[[D_1 + \ldots + D_n]$ is the injection of d into $[D_1 + \ldots + D_n]$

08:

$.\downarrow.$

Semantic Context

$.\downarrow.:[[[D_1 \text{ x } \ldots \text{ x } D_n] \text{ x } N] \rightarrow D_i]$

$d=\langle D_1, \ldots, d_i, \ldots, d_n\rangle:[D_1 \text{ x } \ldots \text{ x } D_i \text{ x } \ldots \text{ x } D_n]$

$i:N$ and $1 <= i <= n$

$d\downarrow i = d_i$

Syntactic Context

$.\downarrow.:[N \text{ x } S] \rightarrow Si]$

$[s]=[s1 \ldots sn]:S$

$i:N$ and $1 <= i <= n$

$i\downarrow[s] = [Si]$

So that $\downarrow$ is used to extract individual components of tuples or node-offspring.

09:

.†.

$d=\langle D_1, \ldots, d_i, d(i+1), \ldots, d_n\rangle:[D_1 \text{ x } \ldots \text{ x } D_n]$

$i:N$ and $1 <= i <= n$

$d \dagger i = \langle d(i+1), \ldots, d_n\rangle$

Operator used to remove elements from tuples.

010:

.%.

$d =\langle D_1, \ldots, d_i\rangle:[D_1 \text{ x } \ldots \text{ x } D_i]$

$d_1=\langle d_j, \ldots, d_n\rangle:[D_j \text{ x } \ldots \text{ x } D_n]$

$i,j:N$ and $j=i+1 <= n$

$d\%d_1 = \langle D_1, \ldots, d_i, d_j, \ldots, d_n\rangle$

Operator used to concatenate tuples.

011:

.?.

$d:[D_1 + \ldots + D_n]$

$i:N$ and $1 <= i <= n$

$d?D_i$ is True if d is in the D_i subdomain of $[D_1 + \ldots + D_n]$, otherwise is False

---

012:

.??.

No semantic difference with '?'. Used by the transformation process to distinguish between compile and run-time type checking.

013:

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

Where o is one of: =, Eq or Ne;

i.e: the first two are equivalent forms of equality and the third one is the inequality operator.

014:

$.o.:[[N \times N] \rightarrow T]$

Where o is one of Lt, Le, Gt, Ge;

i.e: these are the relational operators on integers.

015:

$[\ /\ ]:[[D \times D_1 \times D_2] \rightarrow D]$

$d:D=[D_1 \times \ldots \times D_i \times \ldots \times D_n]$

$i:N$ and $1 <= i <= n$

$r:D_i$

$x:D_1$

$$d[d_1/d_2]=\langle d\downarrow 1, \ldots, d_1\downarrow(i-1), r, d_1\downarrow(i+1), \ldots \rangle$$

$$\text{where } \begin{cases} r = (\lambda x.\ x=d_1 \rightarrow d_2, (d_1\downarrow i)x) \text{ if } D_i=[D_1 \rightarrow D_2] \\ r = d_2 \text{ if } d_1 \text{ is a selector and } d_1 = d\downarrow i \end{cases}$$

This is the postfix operator to create new environments and states. (the concrete notation dSEL, where SEL has been defined as a semantic selector (==), is equivalent to SELd.)

016:

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

$Cond\langle d, d_1\rangle = \lambda t.\ t?T \rightarrow (t \rightarrow d, d_1), (D=COD \rightarrow Wrong, \mathbf{Bot}_D).$

Wrong:COD

017:

SCond

No semantic difference with Cond. Used by the transformation process to distinguish between compile and run-time type checking.

018:

$Fix:[D \rightarrow D]$

$Fix = \lambda F.\bigsqcup_n F^n(\mathbf{Bot}),$

Fix(f) is the minimal fixed point of f; so Fix(f) = f(Fix(f)).

019:
Strict:$[[D_1 \rightarrow D] \rightarrow [D_1 \rightarrow D]]$
(Strict f)d_1 = $\mathbf{Top}_D$, $\mathbf{Bot}_D$ if d_1=**Top**, $\mathbf{Bot}_D$, otherwise f(x).

C.2 **Target**

020:
.!. Vector Application
Provides a way of selecting an element of a vector. A vector is any set of consecutive storage cells. Such a set is introduced by the BCPL function newvec and the ISL primitive functions open.node and forward.vec. The basic form of a vector application is $E_1!E_2$. !E is the same as E!0.

021:
.^. Selector Application
A selector application is the process of applying a selector to perform a byte extraction on a given data structure. Selectors are predefined by the ISL interface, they are: type, p1, p2 ... and weight.

Priority of operators:

The top of the list is the highest priority.
1. name, constant, bracketed expression, **valof**
2. function application.
3. monadic operators: ! **not**
4. !
5. ^
6. + -
7. = **Eq Ne Ls Le Gr Ge**
8. conditional expression

APPENDIX D

Stoy's Final Example

Snapshot D.1: Stoy's Final Example. Original Specification

Syntactic Categories

i:Ide.	identifiers
c:Com.	commands
l:Lco.	labeled com.
e:Exp.	expressions
o:Ops.	binary operators
n:Nml.	numerals
q:Str.	strings

Syntax

c ::= **Dummy** | **If** e **Then** c_1 **Else** c_2 | c_1;c_2 | **While** e **Do** c_1 |
    Let i=e **In** c_1 | **Let** i:=e **In** c_1 | **Goto** e | **Begin** l_1; ...l_2 **End** |
    Call e | **Call** e(e_1) | **Resultis** e | **Break** | **Return** | e:=e_1 |
    c_1 **Repeatwhile** e
l ::= i:c
e ::= i | n | q | e_1oe_2 | **If** e_1 **Then** e_2 **Else** e_3 | **Let** i=e_1 **In** e_2 |
    Let i:=e_1 **In** e_2 | e_1(e_2) | **Fn** i.e_1 | **Rt** c | **Fn** i. **Is** c | **Valof** c |
    Rec i **Fn** i_1.e_1 | **Rec** i **Rt** c
o ::= + | - | * | / | /\ | \/ | > | < | = | <= | >= | #

Semantic Domains

T=[{ TRUE } + { FALSE }].	truth values
N.	integers
Q.	quotations
l:L.	locations
s:S.	machine states
A.	answers
c:C=[S → A].	command cont.
d:D=[T + N + Q + C + F + P + G + L].	denoted values
v:V=[T + N + Q + C + F + P + G].	expression values
e:E=D.	denotations
k:K=[E → C].	expression cont.
w:W=[K → C].	expression closures
f:F=[D → W].	abstractions
P=[D → G].	routines (1 param.)
G=[C → C].	command closures
p:U=[[Ide → D] x K x C x C].	environments

Semantic Domains of 'Interest'

ENV=U.	environments
REG=E.	registered values
TEM=[F + P + G].	templates
STA=S.	states
QUO.	quotations

Snapshot D.1 (continued)

Semantic Primitives (undefined)

N:[Nml → N].
Q:[Str → Q].
O:[Ops → V → V → W].
Assign:[L → V → C → C].
LVal:[E → [L → C] → C].
RVal:[E → [V → C] → C].
Wrong:C.

Semantic Selectors

pRES==p↓2.
pRET==p↓3.
pBRK==p↓4.

Semantic Function for Commands

C:[Com → U → G]. (D.1.1)

C[**Dummy**]pc=
c. (D.1.2)

C[**If** e **Then** c_1 **Else** c_2]pc=
R[e]p{λv.Cond<**C**[c_1]pc,**C**[c_2]pc>v}. (D.1.3)

C[c_1;c_2]pc=
C[c_1]p{**C**[c_2]pc}. (D.1.4)

C[**While** e **Do** c_1]pc=
Fix{λc'.{**R**[e]p'{λv.Cond<**C**[c_1]p'c',c>v}
Where p'=p[c/BRK]}}. (D.1.5)

C[**Let** i=e **In** c_1]pc=
E[e]p{λe.**C**[c_1](p[e In D/[i]])c}. (D.1.6)

C[**Let** i:=e **In** c_1]pc=
R[e]p{λv.LVal{v In E}{λl.**C**[c_1](p[l In D/[i]])c}}. (D.1.7)

C[**Goto** e]pc=
R[e]p{λv.v?C→v|C,Wrong}. (D.1.8)

C[**Begin** l_1; ...l_2 **End**]pc=
Fix(λ<c',...,c''>.
{<**C**(2↓[l_1])p'c'',...,**C**(2↓[l_2])p'c>
Where p'=p[c' In D/1↓[l_1]]...[c'' In D/1↓[l_2]]})↓1. (D.1.9)

C[**Call** e]pc=
R[e]p{λv.v?G→{v|G}c,Wrong}. (D.1.10)

C[**Call** e(e_1)]pc=
R[e]p{λv.v?P→**E**[e_1]p{λe'.(v|P)e'c},Wrong}. (D.1.11)

C[**Resultis** e]pc=
R[e]p{λv.pRES{v In E}}. (D.1.12)

Snapshot D.1 (continued)

C[**Break**]pc=
  pBRK. (D.1.13)

C[**Return**]pc=
  pRET. (D.1.14)

C[c_1 **Repeatwhile** e]pc=
  Fix{λc'.{C[c_1]p'{R[e]p'{λv.Cond<c',c>v}}
  Where p'=p[c/BRK]}}. (D.1.15)

C[e:=e_1]pc=
  L[e]p{λl.R[e_1]p{λv'.Assign lv'c}}. (D.1.16)

<u>Semantic Functions for Expressions</u>

L:[Exp → U → [L → C] → C]. (D.1.17)

L[e]pk:[L → C]=
  E[e]p{λe.LVal ek}. (D.1.18)

R:[Exp → U → [V → C] → C]. (D.1.19)

R[e]pk:[V → C]=
  E[e]p{λe.RVal ek}. (D.1.20)

E:[Exp → U → W]. (D.1.21)

E[i]pk=
  k{p[i]}. (D.1.22)

E[n]pk=
  k{N[n] In E}. (D.1.23)

E[q]pk=
  k{Q[q] In E}. (D.1.24)

E[e_1oe_2]pk=
  R[e_1]p{λv.R[e_2]p{λv'.O[o]vv'k}}. (D.1.25)

E[**If** e_1 **Then** e_2 **Else** e_3]pk=
  R[e_1]p{λv.Cond<E[e_2]pk,E[e_3]pk>v}. (D.1.26)

E[**Let** i=e_1 **In** e_2]pk=
  E[e_1]p{λe.E[e_2](p[e In D/[i]])k}. (D.1.27)

E[**Let** i:=e_1 **In** e_2]pk=
  R[e_1]p{λv.LVal{v In E}{λl.E[e_2](p[l In D/[i]])k}}. (D.1.28)

E[e_1(e_2)]pk=
  R[e_1]p{λv.E[e_2]p{λe'.v?F→(v|F)e'k,Wrong}}. (D.1.29)

E[**Fn** i.e_1]pk=
  k{(λdk.E[e_1](p[d/[i]])k) In E}. (D.1.30)

Snapshot D.1 (continued)

E[**Rt** c]pk=
  k{{λc.**C**[c](p[c/RET])c} In E}. (D.1.31)

E[**Fn** i. **Is** c]pk=
  k{(λdc.**C**[c](p[d/[i]][c/RET])c) In E}. (D.1.32)

E[**Valof** c]pk=
  C[c](p[k/RES])Wrong. (D.1.33)

E[**Rec** i **Fn** i_1.e_1]pk=
  k{Fix(λfdk.E[e_1](p[d/[i_1]][f/[i]])k) In E}. (D.1.34)

E[**Rec** i **Rt** c]pk=
  k{Fix{λcc′.C[c](p[c′/RET][c/[i]])c′} In E}. (D.1.35)

Snapshot D.2: Stoy's Final Example. BCPL

```
let trans.C(node).cont.(continue, jump) be switchon type^node into
                         by R1.1, R3.1, R4.5, R6.10, R7.5, R8.1, RA.1   (D.2.1)
{ case T..Dummy:
    trans.jump.to(continue, jump); endcase
                                   by R1.1, R6.1, R6.10, R8.1, RA.1   (D.2.2)

  case N3..ConditionalCom:
    {0 let continue1 = forward(D..COD)
       trans.R(p1^node).cont.(continue1, false.jump).dest.(first.reg)
       fix.here(continue1)
       trans.skip.if.in(first.reg, D..T)
       trans.jump.to(Wrong, true.jump)
       { let fcond.code = forward(D..COD)
         trans.jump.if.false(first.reg, fcond.code)
         trans.C(p2^node).cont.(continue, true.jump)
         fix.here(fcond.code)
         trans.C(p3^node).cont.(continue, jump)
    }0; endcase
    by R1.1, R1.4, R3.2/3 times, R4.2, R4.6/twice, R5.2, R5.14, R6.1
    R6.2/twice, R6.7, R6.9, R6.10/twice, R7.6/3 times, R8.1/twice
    R8.2/twice, RA.1/4 times, RA.2/6 times
                                                                      (D.2.3)

  case N2..Sequence:
    { let continue1 = forward(D..COD)
      trans.C(p1^node).cont.(continue1, false.jump)
      fix.here(continue1)
      trans.C(p2^node).cont.(continue, jump)
    }; endcase
    by R1.1, R3.2/twice, R4.6/twice, R6.9, R6.10, R7.6/twice, R8.1, R8.2
    RA.1/3 times, RA.2/3 times
                                                                      (D.2.4)

  case N2..While:
    { let old.env = this.env
      let restart.code = here(D..COD)
      declare(D..COD, continue, BRK)
      { let continue1 = forward(D..COD)
        trans.R(p1^node).cont.(continue1, false.jump).dest.(first.reg)
        fix.here(continue1)
        trans.skip.if.in(first.reg, D..T)
        trans.jump.to(Wrong, true.jump)
        trans.jump.if.false(first.reg, continue)
        trans.C(p2^node).cont.(restart.code, true.jump)
      }
      reset(old.env)
    }; endcase
    by R1.1, R1.3, R1.4, R3.2/4 times, R3.3, R4.2, R4.6, R5.2, R5.14
    R6.1/twice, R6.2/twice, R6.3, R6.7, R6.9, R6.10/twice, R7.2, R7.3
    R7.6/twice, R8.2/3 times, RA.1/3 times, RA.2/6 times              (D.2.5)
```

Snapshot D.2 (continued)

```
case N3..DefinitionByDenotationCom:
  {0 let continue1 = forward(D..COD)
     trans.E(p2^node).cont.(continue1, false.jump).dest.(first.reg)
     fix.here(continue1)
     { let old.env = this.env
       declare(domain.of(first.reg), first.reg, p1^node)
       trans.C(p3^node).cont.(continue, jump)
       reset(old.env)
  }0; endcase
  by R1.1, R3.2/3 times, R4.2, R4.6, R5.2, R5.14, R6.9, R6.10, R7.1, R7.2
  R7.6, R8.1, R8.2, RA.1/4 times, RA.2/3 times                     (D.2.6)

case N3..InitialisedDefinitionCom:
  {0 let continue2 = forward(D..COD)
     trans.R(p2^node).cont.(continue2, false.jump).dest.(first.reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       LVal(first.reg).cont.(continue1, false.jump).dest.(first.reg)
       fix.here(continue1)
       { let old.env = this.env
         declare(domain.of(first.reg), first.reg, p1^node)
         trans.C(p3^node).cont.(continue, jump)
         reset(old.env)
  }0; endcase
  by R1.1, R3.2/4 times, R4.2/twice, R4.6, R5.2, R5.14/twice, R5.15
  R6.9/twice, R6.10, R7.1, R7.2, R7.6, R8.1, R8.2/twice, RA.1/4 times
  RA.2/4 times                                                     (D.2.7)

case N1..Goto:
  { let continue1 = forward(D..COD)
    trans.R(p1^node).cont.(continue1, false.jump).dest.(first.reg)
    fix.here(continue1)
    trans.skip.if.in(first.reg, D..COD)
    trans.jump.to(Wrong, true.jump)
    trans.jump.to(first.reg, true.jump)
  }; endcase
  by R1.1, R3.2, R4.2, R5.2, R5.14, R6.1/twice, R6.2, R6.7, R6.9, R7.6
  R8.2/3 times, RA.1/twice, RA.2/3 times                           (D.2.8)
```

Snapshot D.2 (continued)

```
case NX..Block:
  { let node.vec = open.node(node)
    { let old.env = this.env
      let code.vec = forward.vec(!node.vec, D..COD)
      for inx=1 to !node.vec
      do declare(D..COD, code.vec!inx, p1^node.vec!inx)
      for inx=1 to !node.vec-1
      do { fix.here(code.vec!inx)
           trans.C(p2^node.vec!inx).cont.(code.vec!(inx+1), false.jump)
         }
      unless !node.vec=0
      do { fix.here(code.vec!!node.vec)
           trans.C(p2^node.vec!!node.vec).cont.(continue, jump)
         }
      freevec(code.vec)
      reset(old.env)
    }
    freevec(node.vec)
  }; endcase
  by R1.1, R1.3, R3.2/4 times, R3.3, R3.7, R4.6/twice, R6.10, R6.13
  R6.16, R6.17, R7.2, R7.3, R7.6/twice, R7.7, R8.1, R8.2, R8.5, RA.1
  RA.2/4 times, RA.6/3 times
                                                                (D.2.9)

case N1..Call:
  { let continue1 = forward(D..COD)
    trans.R(p1^node).cont.(continue1, false.jump).dest.(first.reg)
    fix.here(continue1)
    trans.skip.if.in(first.reg, D..G)
    trans.jump.to(Wrong, true.jump)
    trans.call(first.reg).cont.(continue, jump).dest.(first.reg)
  }; endcase
  by R1.1, R3.2/twice, R4.2, R4.6, R5.2, R5.14, R5.15, R6.1, R6.2, R6.6
  R6.7, R6.9, R6.10, R7.6, R8.1, R8.2/twice, RA.1/twice, RA.2/3 times
                                                                (D.2.10)
```

Snapshot D.2 (continued)

```
case N2..Call:
  {0 let continue2 = forward(D..COD)
     trans.R(p1^node).cont.(continue2, false.jump).dest.(first.reg)
     fix.here(continue2)
     trans.skip.if.in(first.reg, D..P)
     trans.jump.to(Wrong, true.jump)
     { let continue1 = forward(D..COD)
       test weight^p2^node=max.reg
       then { let old.env = this.env
              let dmp.loc = trans.dump(first.reg)
              trans.E(p2^node).cont.(continue1, false.jump
                     ).dest.(first.reg)
              fix.here(continue1)
              trans.call(dmp.loc, first.reg).cont.(continue, jump
                          ).dest.(first.reg)
              reset(old.env)
            }
       or   { let nxt = next(first.reg)
              trans.E(p2^node).cont.(continue1, false.jump).dest.(nxt)
              fix.here(continue1)
              trans.call(first.reg, nxt).cont.(continue, jump
                          ).dest.(first.reg)
            }
  }0; endcase
  by R1.1, R3.2/3 times, R4.2/twice, R4.6, R5.2, R5.14/twice, R5.15, R6.1
  R6.2, R6.6, R6.7, R6.9/twice, R6.10, R7.6/twice, R8.1, R8.2/3 times
  R9.1, RA.1/5 times, RA.2/6 times                                (D.2.11)

case N1..Resultis:
  { let continue1 = forward(D..COD)
    trans.R(p1^node).cont.(continue1, false.jump).dest.(first.reg)
    fix.here(continue1)
    trans.jump.to(look.up(RES), true.jump)
  }; endcase
  by R1.1, R3.2/twice, R4.1, R4.2, R5.2, R5.3, R5.7, R5.14, R6.1, R6.9
  R7.4, R7.6, R8.2/twice, R9.4, RA.1/twice, RA.2/twice            (D.2.12)

case T..Break:
  trans.jump.to(look.up(BRK), true.jump); endcase
                          by R1.1, R3.2, R6.1, R7.4, R8.2, RA.1  (D.2.13)

case T..Return:
  trans.jump.to(look.up(RET), true.jump); endcase
                          by R1.1, R3.2, R6.1, R7.4, R8.2, RA.1  (D.2.14)
```

Snapshot D.2 (continued)

```
case N2..RepeatWhile:
  { let old.env = this.env
    let restart.code = here(D..COD)
    declare(D..COD, continue, BRK)
    {0 let continue2 = forward(D..COD)
       trans.C(p1^node).cont.(continue2, false.jump)
       fix.here(continue2)
       { let continue1 = forward(D..COD)
         trans.R(p2^node).cont.(continue1, false.jump).dest.(first.reg)
         fix.here(continue1)
         trans.skip.if.in(first.reg, D..T)
         trans.jump.to(Wrong, true.jump)
         test true.jump
         then { trans.jump.if.true(first.reg, restart.code)
                trans.jump.to(continue, jump)
              }
         or   trans.jump.if.false(first.reg, continue)
   }0
   reset(old.env)
  }; endcase
  by R1.1, R1.3, R1.4, R3.2/4 times, R3.3, R4.2, R4.6, R5.2, R5.14
  R6.1/3 times, R6.2/twice, R6.3, R6.7, R6.9/twice, R6.10/twice, R7.2
  R7.3, R7.6/twice, R8.2/4 times, R8.4, RA.1/3 times, RA.2/7 times
                                                                (D.2.15)

case N2..Assignment:
  {0 let continue2 = forward(D..COD)
     trans.L(p1^node).cont.(continue2, false.jump).dest.(first.reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       test weight^p2^node=max.reg
       then { let old.env = this.env
              let dmp.loc = trans.dump(first.reg)
              trans.R(p2^node).cont.(continue1, false.jump
                     ).dest.(first.reg)
              fix.here(continue1)
              Assign(dmp.loc, first.reg).cont.(continue, jump)
              reset(old.env)
            }
       or   { let nxt = next(first.reg)
              trans.R(p2^node).cont.(continue1, false.jump).dest.(nxt)
              fix.here(continue1)
              Assign(first.reg, nxt).cont.(continue, jump)
            }
  }0; endcase
  by R1.1, R3.2/3 times, R4.2/twice, R4.6, R5.2, R5.14/twice, R6.9/twice
  R6.10, R7.6/twice, R8.1, R8.2/twice, R9.1, RA.1/5 times, RA.2/5 times
}                                                               (D.2.16)
```

Snapshot D.2 (continued)

```
let trans.L(node).cont.(continue, jump).dest.(reg) be
                by R1.1, R3.1, R4.3, R5.12, R6.10, R7.5, R8.1, RA.1  (D.2.17)
{ let continue1 = forward(D..COD)
  trans.E(node).cont.(continue1, false.jump).dest.(reg)
  fix.here(continue1)
  LVal(reg).cont.(continue, jump).dest.(first.reg)
}   by R1.1, R3.2/twice, R4.2, R4.4, R5.14, R5.15, R6.9, R6.10, R7.6, R8.1
    R8.2, RA.1/twice, RA.2/twice                                     (D.2.18)

let trans.R(node).cont.(continue, jump).dest.(reg) be
                by R1.1, R3.1, R4.3, R5.12, R6.10, R7.5, R8.1, RA.1  (D.2.19)
{ let continue1 = forward(D..COD)
  trans.E(node).cont.(continue1, false.jump).dest.(reg)
  fix.here(continue1)
  RVal(reg).cont.(continue, jump).dest.(first.reg)
}   by R1.1, R3.2/twice, R4.2, R4.4, R5.14, R5.15, R6.9, R6.10, R7.6, R8.1
    R8.2, RA.1/twice, RA.2/twice                                     (D.2.20)

let trans.E(node).cont.(continue, jump).dest.(reg) be
switchon type^node into
                by R1.1, R3.1, R4.3, R5.12, R6.10, R7.5, R8.1, RA.1  (D.2.21)
{ case T..Ident:
    look.up(node).dest.(reg); trans.jump.to(continue, jump); endcase
    by R1.1, R3.2/twice, R4.1, R5.3, R6.1, R6.10, R7.4, R8.1, RA.1/twice
                                                                     (D.2.22)

  case T..Numeral:
    trans.N(node).dest.(reg); trans.jump.to(continue, jump); endcase
    by R1.1, R3.2/twice, R4.1, R5.3, R6.1, R6.10, R8.1, RA.1/twice, RA.2
                                                                     (D.2.23)

  case T..Quotation:
    trans.Q(node).dest.(reg); trans.jump.to(continue, jump); endcase
    by R1.1, R3.2/twice, R4.1, R5.3, R6.1, R6.10, R8.1, RA.1/twice, RA.2
                                                                     (D.2.24)
```

Snapshot D.2 (continued)

```
case T..Plus: case T..Minus: case T..Mult: case T..Div: case T..And:
case T..Or: case T..GreaterThan: case T..LessThan: case T..Equal:
case T..LessOrEqual: case T..GreaterOrEqual: case T..NotEqual:
  {0 let continue2 = forward(D..COD)
     trans.R(p1^node).cont.(continue2, false.jump).dest.(reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       test weight^p2^node=max.reg
       then { let old.env = this.env
              let dmp.loc = trans.dump(reg)
              trans.R(p2^node).cont.(continue1, false.jump).dest.(reg)
              fix.here(continue1)
              trans.O(type^node, dmp.loc, reg).cont.(continue, jump
                      ).dest.(reg)
              reset(old.env)
            }
       or   { let nxt = next(reg)
              trans.R(p2^node).cont.(continue1, false.jump).dest.(nxt)
              fix.here(continue1)
              trans.O(type^node, reg, nxt).cont.(continue, jump
                      ).dest.(reg)
            }
  }0; endcase
  by R1.1, R3.2/3 times, R4.2/twice, R4.4, R5.13, R5.14/twice, R6.9/twice
  R6.10, R7.6/twice, R8.1, R8.2/twice, R9.1, RA.1/7 times, RA.2/7 times
                                                                  (D.2.25)

case N3..ConditionalExp:
  {0 let continue1 = forward(D..COD)
     trans.R(p1^node).cont.(continue1, false.jump).dest.(reg)
     fix.here(continue1)
     trans.skip.if.in(reg, D..T)
     trans.jump.to(Wrong, true.jump)
     { let fcond.code = forward(D..COD)
       trans.jump.if.false(reg, fcond.code)
       trans.E(p2^node).cont.(continue, true.jump).dest.(reg)
       fix.here(fcond.code)
       trans.E(p3^node).cont.(continue, jump).dest.(reg)
  }0; endcase
  by R1.1, R1.4, R3.2/3 times, R4.2, R4.4/twice, R5.13/twice, R5.14, R6.1
  R6.2/twice, R6.7, R6.9, R6.10/twice, R7.6/3 times, R8.1/twice
  R8.2/twice, RA.1/4 times, RA.2/6 times                          (D.2.26)
```

Snapshot D.2 (continued)

```
case N3..DefinitionByDenotationExp:
  {0 let continue1 = forward(D..COD)
     trans.E(p2^node).cont.(continue1, false.jump).dest.(reg)
     fix.here(continue1)
     { let old.env = this.env
       declare(domain.of(reg), reg, p1^node)
       trans.E(p3^node).cont.(continue, jump).dest.(reg)
       reset(old.env)
  }0; endcase
  by R1.1, R3.2/3 times, R4.2, R4.4, R5.13, R5.14, R6.9, R6.10, R7.1
  R7.2, R7.6, R8.1, R8.2, RA.1/4 times, RA.2/3 times              (D.2.27)

case N3..InitialisedDefinitionExp:
  {0 let continue2 = forward(D..COD)
     trans.R(p2^node).cont.(continue2, false.jump).dest.(reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       LVal(reg).cont.(continue1, false.jump).dest.(first.reg)
       fix.here(continue1)
       { let old.env = this.env
         declare(domain.of(first.reg), first.reg, p1^node)
         trans.E(p3^node).cont.(continue, jump).dest.(first.reg)
         reset(old.env)
  }0; endcase
  by R1.1, R3.2/4 times, R4.2/twice, R4.4, R5.13, R5.14/twice, R5.15
  R6.9/twice, R6.10, R7.1, R7.2, R7.6, R8.1, R8.2/twice, RA.1/4 times
  RA.2/4 times                                                    (D.2.28)
```

Snapshot D.2 (continued)

```
case N2..Application:
  {0 let continue2 = forward(D..COD)
     trans.R(p1^node).cont.(continue2, false.jump).dest.(reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       test weight^p2^node=max.reg
       then { let old.env = this.env
              let dmp.loc = trans.dump(reg)
              trans.E(p2^node).cont.(continue1, false.jump).dest.(reg)
              fix.here(continue1)
              trans.skip.if.in(dmp.loc, D..F)
              trans.jump.to(Wrong, true.jump)
              trans.call(dmp.loc, reg).cont.(continue, jump
                        ).dest.(first.reg)
              reset(old.env)
            }
       or   { let nxt = next(reg)
              trans.E(p2^node).cont.(continue1, false.jump).dest.(nxt)
              fix.here(continue1)
              trans.skip.if.in(reg, D..F)
              trans.jump.to(Wrong, true.jump)
              trans.call(reg, nxt).cont.(continue, jump).dest.(first.reg)
            }
  }0; endcase
  by R1.1, R3.2/3 times, R4.2/twice, R4.4, R5.14/twice, R5.15, R6.1, R6.2
  R6.6, R6.7, R6.9/twice, R6.10, R7.6/twice, R8.1, R8.2/3 times, R9.1
  RA.1/5 times, RA.2/7 times
                                                                (D.2.29)

case N2..Abstraction:
  { let ntry.domF = forward(D..F)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domF, node)
    { let old.env = this.env
      declare(domain.of(first.par), first.par, p1^node)
      trans.E(p2^node).cont.(exit.code, false.jump).dest.(first.reg)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..F, ntry.domF).dest.(reg)
  }
  trans.jump.to(continue, jump); endcase
  by R1.1, R1.2, R3.2/3 times, R4.1, R4.4, R5.3, R5.8, R5.9, R5.10, R5.13
  R6.1, R6.10, R6.11, R7.1, R7.2, R8.1, R8.2/twice, RA.1/3 times
  RA.2/5 times
                                                                (D.2.30)
```

Snapshot D.2 (continued)

```
case N1..Routine:
  { let ntry.domG = forward(D..G)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domG, node)
    { let old.env = this.env
      declare(D..COD, exit.code, RET)
      trans.C(p1^node).cont.(exit.code, false.jump)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..G, ntry.domG).dest.(reg)
  }
  trans.jump.to(continue, jump); endcase
  by R1.1, R3.2/3 times, R4.1, R4.6, R5.3, R5.10, R6.1, R6.10, R6.11
  R7.1, R7.2, R8.1, R8.2/twice, RA.1/twice, RA.2/6 times           (D.2.31)

case N2..Routine:
  { let ntry.domP = forward(D..P)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domP, node)
    { let old.env = this.env
      declare(domain.of(first.par), first.par, p1^node, D..COD, exit.code,
              RET)
      trans.C(p2^node).cont.(exit.code, false.jump)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..P, ntry.domP).dest.(reg)
  }
  trans.jump.to(continue, jump); endcase
  by R1.1, R1.2, R3.2/3 times, R4.1, R4.6, R5.3, R5.8, R5.9, R5.10, R6.1
  R6.10, R6.11, R7.1, R7.2, R8.1, R8.2/twice, RA.1/3 times, RA.2/6 times
                                                                   (D.2.32)

case N1..Valof:
  { let old.env = this.env
    declare(D..COD, continue, RES)
    trans.C(p1^node).cont.(Wrong, true.jump)
    reset(old.env)
  }; endcase
  by R1.1, R3.2/twice, R4.4, R4.6, R6.10, R7.1, R7.2, R8.2, RA.1/twice
  RA.2/twice                                                       (D.2.33)
```

Snapshot D.2 (continued)

```
case N3..RecAbstraction:
  { let ntry.domF = forward(D..F)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domF, node)
    { let old.env = this.env
      declare(domain.of(first.par), first.par, p2^node, D..F, ntry.domF,
              p1^node)
      trans.E(p3^node).cont.(exit.code, false.jump).dest.(first.reg)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..F, ntry.domF).dest.(reg)
  }
  trans.jump.to(continue, jump); endcase
  by R1.1, R1.2/twice, R3.2/4 times, R4.1, R4.4, R5.3, R5.8, R5.9, R5.10
  R5.13, R6.1, R6.5, R6.10, R6.11, R7.1, R7.2, R8.1, R8.2/twice
  RA.1/4 times, RA.2/6 times
                                                               (D.2.34)

case N2..RecRoutine:
  { let ntry.domG = forward(D..G)
    let exit.code = forward(D..COD)
    let skip.code = forward(D..COD)
    trans.jump.to(skip.code, true.jump)
    trans.entry(ntry.domG, node)
    { let old.env = this.env
      declare(D..COD, exit.code, RET, D..G, ntry.domG, p1^node)
      trans.C(p2^node).cont.(exit.code, false.jump)
      reset(old.env)
    }
    trans.exit(exit.code, node)
    fix.here(skip.code)
    trans.load(D..G, ntry.domG).dest.(reg)
  }
  trans.jump.to(continue, jump); endcase
  by R1.1, R1.2, R3.2/4 times, R4.1, R4.6, R5.3, R5.10, R6.1, R6.5, R6.10
  R6.11, R7.1, R7.2, R8.1, R8.2/twice, RA.1/3 times, RA.2/7 times (D.2.35)
}
```

APPENDIX E

GEDANKEN

Snapshot E.1: GEDANKEN. Original Specification

Syntactic Categories

b:Bse.	bases
n:Nml.	numerals
c:Chr.	characters
q:Quo.	quotations
i:Ide.	identifiers
e:Exp.	expressions
f:Abs.	abstractions
p:Par.	parameters
s:Prog.	programs
r:RecDec.	recursive dec.
l:LabDec.	label dec.

Syntax

$b ::= n \mid c \mid q$

$p ::= i \mid p_1, \ldots ,p_2 \mid \mathbf{EmptyPar} \mid (p_1)$

$f ::= \mathbf{Lam}\ p.e$

$e ::= b \mid i \mid f \mid e_1 e_2 \mid \mathbf{If}\ e_1\ \mathbf{Then}\ e_2\ \mathbf{Else}\ e_3 \mid e_1\ \mathbf{And}\ e_2 \mid e_1\ \mathbf{Or}\ e_2 \mid$
$\quad \mathbf{Case}\ e_1\ \mathbf{Of}\ e_2, \ldots ,e_3 \mid e_1, \ldots ,e_2 \mid \mathbf{EmptyExp} \mid e_1 = e_2 \mid e_1 := e_2 \mid$
$\quad e_1 ; e_2 \mid p\ \mathbf{Is}\ e_1 ; e_2 \mid r_1 ; \ldots ; r_2 ; l_1 ; \ldots ; l_2 \mid (e_1)$

$r ::= i\ \mathbf{Isr}\ f$

$l ::= i : e$

Semantic Domains

T=[{ TRUE } + { FALSE }].	truth values
n:N.	integers
H.	characters
Q.	quotations
O=[T + N + H + Q].	output values
At=[{ Ll } + { Ul } + GAt].	atoms
GAt.	generated atoms
B=[T + N + H + Q + At].	basic values
f:F=[E → K → C].	functions
c:C=[S → A].	label values
L.	locations
Im=[F x F].	implicit references
Rf=[L + Im].	references
e:E=[B + F + C + Rf].	expressible values
A=[{ Erro } + B + [O x A]].	answers
k:K=[E → C].	expression cont.
x:X=[U → C].	parameter cont.
D=E.	denotable values
p:U=[Ide → D].	environments
V=E.	storable values
S=[[L → [V x T]] x [At → T] x H* x O*].	stores

Snapshot E.1 (continued)

Semantic Primitives (undefined)

```
Ccoerce:[E → K → C].
Cerror:C.
NCequal:[F → K → C].
NCset:[F → K → C].
Select:[C* → N → C → C].
Seq:[E* → F].
B:[Bse → B].
M:[Prog → H* → A].
```

Semantic Domains of 'Interest'

```
ENV=U.        environments
REG=E.        registered values
TEM=F.        templates
```

Semantic Equations

```
E:[Exp → U → K → C].                                              (E.1.1)

E[b]pk=
  k(B[b]).                                                        (E.1.2)

E[i]pk=
  k{p[i]}.                                                        (E.1.3)

E[f]pk=
  k{F[f]p}.                                                       (E.1.4)

E[e_1 e_2]pk=
  R[e_1]p{λe.e?F→E[e_2]p{λe'.{e|F}e'k},Cerror}.                   (E.1.5)

E[If e_1 Then e_2 Else e_3]pk=
  R[e_1]p{λe.e?T→e|T→E[e_2]pk,E[e_3]pk,Cerror}.                   (E.1.6)

E[e_1 And e_2]pk=
  R[e_1]p{λe.e?T→e|T→R[e_2]pk,kFALSE,Cerror}.                     (E.1.7)

E[e_1 Or e_2]pk=
  R[e_1]p{λe.e?T→e|T→kTRUE,R[e_2]pk,Cerror}.                      (E.1.8)

E[Case e_1 Of e_2, ... ,e_3]pk=
  R[e_1]p
  {λe.e?N→Select<E[e_2]pk,...,E[e_3]pk>(e|N)Cerror,
      e?At→(e|At)=L1→k 1,(e|At)=U1→k(#[...]),Cerror,Cerror}.      (E.1.9)

E[e_1, ... ,e_2]pk=
  E[e_1]p{λe. ...E[e_2]p{λe'.k{Seq<e,...,e'>}}}.                  (E.1.10)

E[EmptyExp]pk=
  k{Seq<>}.                                                       (E.1.11)

E[e_1=e_2]pk=
  R[e_1]p{λe.R[e_2]p{λe'.NCequal{Seq<e,e'>}k}}.                   (E.1.12)
```

Snapshot E.1 (continued)

$$\begin{aligned}&E[e_1{:=}e_2]pk=\\&\quad E[e_1]p\{\lambda e.R[e_2]p\{\lambda e'.NCset\{Seq\langle e,e'\rangle\}k\}\}.\end{aligned}\tag{E.1.13}$$

$$\begin{aligned}&E[e_1;e_2]pk=\\&\quad E[e_1]p\{\lambda e.E[e_2]pk\}.\end{aligned}\tag{E.1.14}$$

$$\begin{aligned}&E[p\ \mathbf{Is}\ e_1;e_2]pk=\\&\quad E[e_1]p\{\lambda e.P[p]pe\{\lambda p'.E[e_2]p'k\}\}.\end{aligned}\tag{E.1.15}$$

$$\begin{aligned}&E[r_1;\ \ldots\ ;r_2;l_1;\ \ldots\ ;l_2]pk=\\&\quad Fix(\lambda\langle f',\ldots,f'',c',\ldots,c''\rangle.\\&\qquad\quad (\langle F(2{\downarrow}[r_1])p',\ldots,F(2{\downarrow}[r_2])p',E(2{\downarrow}[l_1])p'\{\lambda e.c''\},\ldots,\\&\qquad\qquad E(2{\downarrow}[l_2])p'k\\&\qquad\quad \rangle\\&\quad \text{Where } p'=p[f'/1{\downarrow}[r_1]]\ldots[f''/1{\downarrow}[r_2]][c'/1{\downarrow}[l_1]]\ldots[c''/1{\downarrow}[l_2]]))\downarrow 2\downarrow 1\\&\quad .\end{aligned}\tag{E.1.16}$$

$$\begin{aligned}&E[(e_1)]pk=\\&\quad E[e_1]pk.\end{aligned}\tag{E.1.17}$$

$$P{:}[Par \to U \to E \to X \to C].\tag{E.1.18}$$

$$\begin{aligned}&P[i]pex=\\&\quad x(p[e/[i]]).\end{aligned}\tag{E.1.19}$$

$$\begin{aligned}&P[p_1,\ \ldots\ ,p_2]pex=\\&\quad Ccoerce\ e\\&\quad \{\lambda e.e?F\to U[p_1]p\{e|F\}1\{\lambda p'.\ \ldots U[p_2]p'\{e|F\}(\#[p_1,\ \ldots\ ,p_2])x\},Cerror\}\\&\quad .\end{aligned}\tag{E.1.20}$$

$$\begin{aligned}&P[\mathbf{EmptyPar}]pex=\\&\quad xp.\end{aligned}\tag{E.1.21}$$

$$\begin{aligned}&P[(p_1)]pex=\\&\quad P[p_1]pex.\end{aligned}\tag{E.1.22}$$

$$R{:}[Exp \to U \to K \to C].\tag{E.1.23}$$

$$\begin{aligned}&R[e]pk=\\&\quad E[e]p\{\lambda e.Ccoerce\ ek\}.\end{aligned}\tag{E.1.24}$$

$$F{:}[Abs \to U \to F].\tag{E.1.25}$$

$$\begin{aligned}&F[\mathbf{Lam}\ p.e]p=\\&\quad \lambda ek.P[p]pe\{\lambda p'.E[e]p'k\}.\end{aligned}\tag{E.1.26}$$

$$U{:}[Par \to U \to F \to N \to X \to C].\tag{E.1.27}$$

$$\begin{aligned}&U[p]pfnx=\\&\quad fn\{\lambda e.P[p]pex\}.\end{aligned}\tag{E.1.28}$$

Snapshot E.2: GEDANKEN. BCPL

```
let trans.E(node).cont.(continue, jump).dest.(reg) be
switchon type^node into
                by R1.1, R3.1, R4.3, R5.12, R6.10, R7.5, R8.1, RA.1   (E.2.1)
{ case T..Numeral: case T..Character: case T..Quotation:
    trans.B(node).dest.(reg); trans.jump.to(continue, jump); endcase
    by R1.1, R3.2/twice, R4.1, R5.3, R6.1, R6.10, R8.1, RA.1/twice, RA.2
                                                                      (E.2.2)

  case T..Ident:
    look.up(node).dest.(reg); trans.jump.to(continue, jump); endcase
    by R1.1, R3.2/twice, R4.1, R5.3, R6.1, R6.10, R7.4, R8.1, RA.1/twice
                                                                      (E.2.3)

  case N2..Lambda:
    trans.F(node).dest.(reg); trans.jump.to(continue, jump); endcase
    by R1.1, R3.2/twice, R4.1, R5.3, R6.1, R6.10, R7.6, R8.1, RA.1/twice
    RA.2
                                                                      (E.2.4)

  case N2..FunctionDesignator:
    {0 let continue2 = forward(D..COD)
       trans.R(p1^node).cont.(continue2, false.jump).dest.(reg)
       fix.here(continue2)
       trans.skip.if.in(reg, D..F)
       trans.jump.to(Cerror, true.jump)
       { let continue1 = forward(D..COD)
         test weight^p2^node=max.reg
         then { let old.env = this.env
                let dmp.loc = trans.dump(reg)
                trans.E(p2^node).cont.(continue1, false.jump).dest.(reg)
                fix.here(continue1)
                trans.call(dmp.loc, reg).cont.(continue, jump
                          ).dest.(first.reg)
                reset(old.env)
              }
         or   { let nxt = next(reg)
                trans.E(p2^node).cont.(continue1, false.jump).dest.(nxt)
                fix.here(continue1)
                trans.call(reg, nxt).cont.(continue, jump).dest.(first.reg)
              }
    }0; endcase
    by R1.1, R3.2/3 times, R4.2/twice, R4.4, R5.14/twice, R5.15, R6.1, R6.2
    R6.6, R6.7, R6.9/twice, R6.10, R7.6/twice, R8.1, R8.2/3 times, R9.1
    RA.1/5 times, RA.2/6 times
                                                                      (E.2.5)
```

Snapshot E.2 (continued)

```
case N3..Conditional:
  {0 let continue1 = forward(D..COD)
     trans.R(p1^node).cont.(continue1, false.jump).dest.(reg)
     fix.here(continue1)
     trans.skip.if.in(reg, D..T)
     trans.jump.to(Cerror, true.jump)
     { let fcond.code = forward(D..COD)
       trans.jump.if.false(reg, fcond.code)
       trans.E(p2^node).cont.(continue, true.jump).dest.(reg)
       fix.here(fcond.code)
       trans.E(p3^node).cont.(continue, jump).dest.(reg)
  }0; endcase
  by R1.1, R3.2/3 times, R4.2, R4.4/twice, R5.13/twice, R5.14, R6.1
  R6.2/twice, R6.7, R6.9, R6.10/twice, R7.6/3 times, R8.1/twice
  R8.2/twice, RA.1/4 times, RA.2/6 times                        (E.2.6)

case N2..And:
  {0 let continue1 = forward(D..COD)
     trans.R(p1^node).cont.(continue1, false.jump).dest.(reg)
     fix.here(continue1)
     trans.skip.if.in(reg, D..T)
     trans.jump.to(Cerror, true.jump)
     { let fcond.code = forward(D..COD)
       trans.jump.if.false(reg, fcond.code)
       trans.R(p2^node).cont.(continue, true.jump).dest.(reg)
       fix.here(fcond.code)
       trans.load(D..T, FALSE).dest.(reg)
       trans.jump.to(continue, jump)
  }0; endcase
  by R1.1, R3.2/3 times, R4.1, R4.2, R4.4, R5.3, R5.7, R5.13, R5.14
  R6.1/twice, R6.2/twice, R6.7, R6.9, R6.10/twice, R7.6/twice, R8.1/twice
  R8.2/twice, RA.1/3 times, RA.2/6 times                        (E.2.7)

case N2..Or:
  {0 let continue1 = forward(D..COD)
     trans.R(p1^node).cont.(continue1, false.jump).dest.(reg)
     fix.here(continue1)
     trans.skip.if.in(reg, D..T)
     trans.jump.to(Cerror, true.jump)
     { let fcond.code = forward(D..COD)
       trans.jump.if.false(reg, fcond.code)
       trans.load(D..T, TRUE).dest.(reg)
       trans.jump.to(continue, true.jump)
       fix.here(fcond.code)
       trans.R(p2^node).cont.(continue, jump).dest.(reg)
  }0; endcase
  by R1.1, R3.2/3 times, R4.1, R4.2, R4.4, R5.3, R5.7, R5.13, R5.14
  R6.1/twice, R6.2/twice, R6.7, R6.9, R6.10/twice, R7.6/twice, R8.1/twice
  R8.2/twice, RA.1/3 times, RA.2/6 times                        (E.2.8)
```

Snapshot E.2 (continued)

```
case N2..Case:
  { let node.vec2 = open.node(p2^node)
    {0 let continue1 = forward(D..COD)
       trans.R(p1^node).cont.(continue1, false.jump).dest.(reg)
       fix.here(continue1)
       { let fcond.code = forward(D..COD)
         trans.skip.if.in(reg, D..N)
         trans.jump.to(fcond.code, true.jump)
         { let code.vec2 = forward.vec(!node.vec2, D..COD)
           let skip.code = forward(D..COD)
           trans.jump.to(skip.code, true.jump)
           for inx=1 to !node.vec2
           do { fix.here(code.vec2!inx)
                trans.E(node.vec2!inx).cont.(continue, true.jump
                        ).dest.(reg)
              }
           fix.here(skip.code)
           Select(code.vec2, reg).cont.(Cerror, true.jump)
           freevec(code.vec2)
         }
         fix.here(fcond.code)
         trans.skip.if.in(reg, D..At)
         trans.jump.to(Cerror, true.jump)
         { let fcond.code = forward(D..COD)
           trans.skip.if(i.skipEQ, reg, L1)
           trans.jump.to(fcond.code, true.jump)
           trans.load(D..N, make.num(1)).dest.(reg)
           trans.jump.to(continue, true.jump)
           fix.here(fcond.code)
           trans.skip.if(i.skipEQ, reg, U1)
           trans.jump.to(Cerror, true.jump)
           trans.load(D..N, make.num(!node.vec2)).dest.(reg)
           trans.jump.to(continue, jump)
  }0
  freevec(node.vec2)
}; endcase
by R1.1, R3.2/6 times, R3.7, R4.1/twice, R4.2, R4.4/twice, R4.6
R5.3/twice, R5.7/twice, R5.13/twice, R5.14, R5.19/twice, R6.1/4 times
R6.2/4 times, R6.7/twice, R6.9, R6.10/4 times, R6.13, R6.15
R7.6/3 times, R8.1/4 times, R8.2/7 times, R8.5, RA.1/twice
RA.2/11 times, RA.10/twice
                                                             (E.2.9)
```

Snapshot E.2 (continued)

```
case NX..Sequence:
  { let node.vec = open.node(node)
    { let old.env = this.env
      let old.off = this.off
      for inx=1 to !node.vec-1
      do { { let continue1 = forward(D..COD)
             trans.E(node.vec!inx).cont.(continue1, false.jump
                   ).dest.(reg)
             fix.here(continue1)
           }
           trans.dump(reg)
         }
      unless !node.vec=0
      do { let continue2 = forward(D..COD)
           trans.E(node.vec!!node.vec).cont.(continue2, false.jump
                 ).dest.(reg)
           fix.here(continue2)
           trans.dump(reg)
           Seq(old.off).dest.(reg)
           trans.jump.to(continue, jump)
         }
      reset(old.env)
    }
    freevec(node.vec)
  }; endcase
  by R1.1, R3.2/4 times, R3.7, R4.1, R4.2, R4.6, R4.7, R5.14, R5.16, R6.1
  R6.9, R6.10, R6.18, R7.6/twice, R8.1, R8.2/twice, RA.1, RA.2/4 times
                                                                   (E.2.10)

case N0..EmptyExp:
  Seq(reg); trans.jump.to(continue, jump); endcase
       by R1.1, R3.2/twice, R4.1, R5.13, R6.1, R6.10, R8.1, RA.1  (E.2.11)
```

Snapshot E.2 (continued)

```
case N2..Equal:
  {0 let continue2 = forward(D..COD)
     trans.R(p1^node).cont.(continue2, false.jump).dest.(reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       test weight^p2^node=max.reg
       then { let old.env = this.env
              let dmp.loc = trans.dump(reg)
              trans.R(p2^node).cont.(continue1, false.jump).dest.(reg)
              fix.here(continue1)
              NCequal(Seq(dmp.loc).dest.(reg)).cont.(continue, jump
                      ).dest.(first.reg)
              reset(old.env)
            }
       or   { let nxt = next(reg)
              trans.R(p2^node).cont.(continue1, false.jump).dest.(nxt)
              fix.here(continue1)
              NCequal(Seq(reg).dest.(nxt)).cont.(continue, jump
                      ).dest.(first.reg)
            }
  }0; endcase
  by R1.1, R3.2/4 times, R4.2/twice, R4.4, R5.14/twice, R5.15, R6.9/twice
  R6.10, R7.6/twice, R8.1, R8.2/twice, R9.1, RA.1/5 times, RA.2/5 times
                                                                (E.2.12)

case N2..Assignment:
  {0 let continue2 = forward(D..COD)
     trans.E(p1^node).cont.(continue2, false.jump).dest.(reg)
     fix.here(continue2)
     { let continue1 = forward(D..COD)
       test weight^p2^node=max.reg
       then { let old.env = this.env
              let dmp.loc = trans.dump(reg)
              trans.R(p2^node).cont.(continue1, false.jump).dest.(reg)
              fix.here(continue1)
              NCset(Seq(dmp.loc).dest.(reg)).cont.(continue, jump
                    ).dest.(first.reg)
              reset(old.env)
            }
       or   { let nxt = next(reg)
              trans.R(p2^node).cont.(continue1, false.jump).dest.(nxt)
              fix.here(continue1)
              NCset(Seq(reg).dest.(nxt)).cont.(continue, jump
                    ).dest.(first.reg)
            }
  }0; endcase
by R1.1, R3.2/4 times, R4.2/twice, R4.4, R5.14/twice, R5.15, R6.9/twice
R6.10, R7.6/twice, R8.1, R8.2/twice, R9.1, RA.1/5 times, RA.2/5 times
                                                                (E.2.13)
```

Snapshot E.2 (continued)

```
case N2..Compound:
  { let continue1 = forward(D..COD)
    trans.E(p1^node).cont.(continue1, false.jump).dest.(reg)
    fix.here(continue1)
    trans.E(p2^node).cont.(continue, jump).dest.(reg)
  }; endcase
  by R1.1, R3.2/twice, R4.2, R4.4, R5.13, R5.14, R6.9, R6.10, R7.6/twice
  R8.1, R8.2, RA.1/3 times, RA.2/3 times                          (E.2.14)

case N3..Dec:
  {0 let continue2 = forward(D..COD)
     trans.E(p2^node).cont.(continue2, false.jump).dest.(reg)
     fix.here(continue2)
     { let old.env = this.env
       let continue1 = forward(D..COD)
       trans.P(p1^node, reg).cont.(continue1, false.jump)
       fix.here(continue1)
       trans.E(p3^node).cont.(continue, jump).dest.(reg)
       reset(old.env)
  }0; endcase
  by R1.1, R3.2/3 times, R4.2/twice, R4.4, R5.13, R5.14, R6.9/twice
  R6.10, R7.6/3 times, R7.8, R8.1, R8.2/twice, RA.1/4 times, RA.2/5 times
                                                                  (E.2.15)
```

Snapshot E.2 (continued)

```
case N2..Block:
  { let node.vecl = open.node(pl^node)
    let node.vec2 = open.node(p2^node)
    { let old.env = this.env
      let code.vecl = forward.vec(!node.vecl, D..F)
      let code.vec2 = forward.vec(!node.vec2, D..COD)
      for inx=1 to !node.vecl
      do declare(D..COD, code.vecl!inx, pl^node.vecl!inx)
      for inx=1 to !node.vec2
      do declare(D..COD, code.vec2!inx, pl^node.vec2!inx)
      for inx=1 to !node.vecl
      do { trans.F(p2^node.vecl!inx).dest.(reg)
           fix.with(code.vecl!inx, reg)
         }
      for inx=1 to !node.vec2-1
      do { fix.here(code.vec2!inx)
           trans.E(p2^node.vec2!inx).cont.(code.vec2!(inx+1), true.jump
                  ).dest.(reg)
         }
      unless !node.vec2=0
      do { fix.here(code.vec2!!node.vec2)
           trans.E(p2^node.vec2!!node.vec2).cont.(continue, jump
                  ).dest.(reg)
         }
      freevec(code.vec2)
      freevec(code.vecl)
      reset(old.env)
    }
    freevec(node.vec2)
    freevec(node.vecl)
  }; endcase
  by Rl.1, Rl.3, R3.2/6 times, R3.3, R3.7/twice, R4.2, R4.4, R5.3/twice
  R5.13, R5.14, R6.10, R6.13, R6.16, R6.17/twice, R7.2, R7.3
  R7.6/4 times, R7.7/twice, R8.1, R8.2, R8.5/3 times, RA.1, RA.2/7 times
  RA.6/5 times
                                                            (E.2.16)

 case Nl..BraExp:
   trans.E(pl^node).cont.(continue, jump).dest.(reg); endcase
   by Rl.1, R3.2, R4.4, R5.13, R6.10, R7.6, R8.1, RA.1/twice, RA.2 (E.2.17)
}

let trans.P(node, reg).cont.(continue, jump) be switchon type^node into
       by Rl.1, R3.1, R4.3, R5.12, R5.18, R6.10, R7.5, R8.1, RA.1  (E.2.18)
{ case T..Ident:
    declare(domain.of(reg), reg, node); trans.jump.to(continue, jump)
    endcase
    by Rl.1, R3.2/twice, R4.1, R5.18, R6.1, R6.10, R7.2, R8.1, RA.1/twice
                                                              (E.2.19)
```

Snapshot E.2 (continued)

```
case NX..SeqPar:
  { let node.vec = open.node(node)
    {0 let continue2 = forward(D..COD)
       Ccoerce(reg).cont.(continue2, false.jump).dest.(first.reg)
       fix.here(continue2)
       trans.skip.if.in(first.reg, D..F)
       trans.jump.to(Cerror, true.jump)
       { let dmp.loc = trans.dump(first.reg)
         for inx=1 to !node.vec-1
         do { { let continue1 = forward(D..COD)
                trans.U(node.vec!inx, first.reg, make.num(inx)
                       ).cont.(continue1, false.jump)
                fix.here(continue1)
              }
              trans.load(D..F, dmp.loc).dest.(first.reg)
            }
         unless !node.vec=0
         do trans.U(node.vec!!node.vec, first.reg, make.num(size^node)
                   ).cont.(continue, jump)
    }0
    freevec(node.vec)
  }; endcase
  by R1.1, R3.2/3 times, R3.7, R4.2, R4.4, R4.6, R4.7, R5.14, R5.15
  R5.17, R5.18, R6.1, R6.2, R6.7, R6.9, R6.10, R6.18, R7.6/twice, R7.9
  R7.11, R8.1, R8.2/3 times, RA.1/twice, RA.2/6 times, RA.9, RA.10/twice
                                                                  (E.2.20)

 case N0..EmptyPar:
   trans.jump.to(continue, jump); endcase
             by R1.1, R3.2, R4.1, R6.1, R6.10, R7.10, R8.1, RA.1  (E.2.21)

 case N1..ParBra:
   trans.P(p1^node, reg).cont.(continue, jump); endcase
   by R1.1, R3.2, R4.4, R5.18, R6.10, R7.6, R7.11, R8.1, RA.1/twice, RA.2
                                                                  (E.2.22)
}

let trans.R(node).cont.(continue, jump).dest.(reg) be
             by R1.1, R3.1, R4.3, R5.12, R6.10, R7.5, R8.1, RA.1  (E.2.23)
{ let continue1 = forward(D..COD)
  trans.E(node).cont.(continue1, false.jump).dest.(reg)
  fix.here(continue1)
  Ccoerce(reg).cont.(continue, jump).dest.(first.reg)
}  by R1.1, R3.2/twice, R4.2, R4.4, R5.14, R5.15, R6.9, R6.10, R7.6, R8.1
   R8.2, RA.1/twice, RA.2/twice                                   (E.2.24)
```

Snapshot E.2 (continued)

```
let trans.F(node).dest.(reg) be     by R1.1, R3.1, R5.1, R7.5, RA.1   (E.2.25)
{ let ntry.domF = forward(D..F)
  let exit.code = forward(D..COD)
  let skip.code = forward(D..COD)
  trans.jump.to(skip.code, true.jump)
  trans.entry(ntry.domF, node)
  { let old.env = this.env
    let continue = forward(D..COD)
    trans.P(p1^node, first.par).cont.(continue, false.jump)
    fix.here(continue)
    trans.E(p2^node).cont.(exit.code, false.jump).dest.(first.reg)
    reset(old.env)
  }
  trans.exit(exit.code, node)
  fix.here(skip.code)
  trans.load(D..F, ntry.domF).dest.(reg)
}   by R1.1, R1.2, R3.2/twice, R4.2, R4.4, R5.3, R5.8, R5.9, R5.10, R5.13
    R6.9, R6.11, R7.6/twice, R7.8, R8.2/3 times, RA.1/3 times, RA.2/7 times
                                                                  (E.2.26)

let trans.U(node, f, reg).cont.(continue, jump) be
        by R1.1, R3.1, R4.3, R5.12, R5.18, R6.10, R7.5, R8.1, RA.1   (E.2.27)
{ let continue1 = forward(D..COD)
  trans.call(f, reg).cont.(continue1, false.jump).dest.(first.reg)
  fix.here(continue1)
  trans.P(node, first.reg).cont.(continue, jump)
}   by R1.1, R3.2/twice, R4.2, R4.4, R5.14, R5.15, R5.18, R6.6, R6.9, R6.10
    R7.6, R7.11, R8.1, R8.2, RA.1/twice, RA.2/twice                 (E.2.28)
```