A Semantic Algebra Approach to Denotational Semantics of Programming Languages

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
at the
University of Waikato
by
ANTONY LLOYD HOSKING

University of Waikato
1986

A B S T R A C T

The basic theory and techniques of *denotational semantics* are presented and developed. The λ-*calculus* language is introduced and used as an example to illustrate the denotational approach to semantic specification. An efficient *call-by-need* interpreter for the λ-calculus is developed and implemented. The denotational approach to programming language specification is enhanced by the definition of a *semantic algebra*. Semantic definitions using the semantic algebra are given for a simple extension of the pure λ-calculus and for a simple Pascal-like imperative programming language. An implementation of the basic semantic algebra is given, allowing the semantic description for the λ-calculus to be directly executed.

A C K N O W L E D G E M E N T S

Having completed this thesis it is my pleasure to acknowledge the many people who encouraged, inspired and supported me throughout its production. Particular thanks go to : my supervisor, Prof. E.V. Krishnamurthy, for his open guidance, helpful suggestions and challenging discussions; to my parents, for their support and encouragement throughout this Master's programme; and finally to the late Dr. John Sanderson for first introducing me to the λ-calculus.

C O N T E N T S

Chapter 1
GENERAL INTRODUCTION

1.1 Study of Formal Semantics

In his paper titled *A Basis for a Mathematical Theory of Computation* [1963], John McCarthy called for the development of a mathematically rigorous theory of computation. He saw the achievement of a number of goals as being necessary to this development, one of which was the the development of a universal "programming language" or formalism for describing computations, having the facility for describing different kinds of data and well suited to the formal description of procedures. Although McCarthy's paper did not in any way achieve this goal it did point the way to further work, initiating work on the theory of how computations are built up from elementary operations and how the data spaces are constructed on which these operations are defined. McCarthy recognised that the lack of an adequate mathematical formalism for describing computations was hindering the development of the *science* of computing.

Since all algorithms for computers are eventually encoded in some form of programming language, the goal of precisely defining computations depends on being able to give a precise semantics to programming languages. There are various categories of people who need to be able to precisely communicate their understanding about programming languages. Language *designers* need to be able to formally analyse the similarities and differences between existing languages and to explain their language designs completely and precisely. *Implementers* want to be sure that their implementations conform to the designers' specifications of the language. *Programmers* applying the language need to be able to make rigorous statements about the behaviour of the programs that they write. Classical problems have been determining program equivalence and program correctness. All of the above people need a universal formalism for defining the *semantics* of programming languages. For these reasons considerable effort has been put into attaining the goals specified by McCarthy and for developing methods for formalising programming language semantics.

1.2 Methods for Formal Semantic Definition

Three main methods for giving semantic descriptions of programming languages have been developed : the *operational*, *denotational* and *axiomatic* approaches. Each of these approaches is detailed in McGettrick [1980]. Comparisons of the approaches are made in Trakhtenbrot et al. [1984] while their complementary use is argued for in Donahue [1976].

1.2.1 Operational Semantics

The operational approach proceeds by defining an abstract machine along with a set of primitive instructions for it. The effect on the "state" of the machine is specified for each of the primitive instructions. The semantic definition of the programming language of interest specifies a translation into the primitive instruction set of the abstract machine - i.e. the semantics is defined by a particular implementation of the language on the abstract machine. By fully understanding the code for the abstract machine and tracing the execution of a translated program step by step, the precise effect of the program is determined. An example of this approach is the definition of pure LISP via an interpreter for the so-called "SECD machine" in Henderson [1980].

1.2.2 Denotational Semantics

The denotational approach to semantic definition of programming languages sprang almost directly from the work of McCarthy [1963]. "Semantic valuation functions" are given which map syntactic objects in a program to the abstract values which they *denote*. The original work by Strachey [1966] mapped the syntactic objects of a large segment of CPL into λ-expressions. At that time it was doubtful as to whether any reasonable mathematical interpretation existed for the equations specifying the evaluations to the underlying set of abstract values. However, as the result of a very fruitful collaboration with Dana Scott culminating in the monumental paper *Data Types as Lattices* (Scott [1976]), the lattice-theoretic approach to the theory of computation has come to provide the necessary theory underpinning the method.

1.2.3 Axiomatic Semantics

In the axiomatic approach an axiom is associated with each kind of statement in the programming language - i.e. the statement is regarded as specifying relations between assertions about the state of the machine before and after the statement is executed.

1.2.4 Comparison of the Three Methods

There are various arguments for and against each of the three methods outlined above. In many cases the particular method used depends on the focus of interest in the language. The operational method is often of most value to the implementer, the axiomatic to the programmer and the denotational to the language designer, yet these are not hard and fast distinctions. Each method is of value in all spheres of interest in programming languages. However it has been claimed (see Stoy [1977]) that the denotational approach is most generally applicable. Operational definitions contain extra implementation details which hide the semantics in complicated detail, whereas axiomatic semantics give a less than complete description of a language, concentrating only on those properties of concern to the programmer.

1.3 Aims and Scope of this Thesis

The advantages of denotational semantics (i.e. the mathematical foundations of Scott and the language definition techniques of Strachey) lead us to adopt the denotational approach to programming language definition in this thesis. We aim to achieve a practical working familiarity with denotational semantic definitions by giving definitions of simple example languages. Furthermore, a *semantic algebra* approach to denotational semantics is advocated to improve language definition. Some example languages are defined to illustrate this algebraic approach.

In Chapter 2 we introduce the λ-calculus as the notational device for denotational semantic definitions, and also expose it as a simple functional programming language in its own right, with the facility to represent various common concepts of computation.

Chapter 3 is devoted to the basic notions of the denotational approach to semantic specification, and to Scott's theory of

computation; these are followed by an example of application of the approach to the definition of the λ-calculus. Proofs are provided to show the equivalence of convertible expressions and the correctness of any λ-calculus implementation based on the conversion rules.

Chapter 4 describes the philosophy behind and implementation of an efficient λ-calculus interpreter. It also discusses the three important parameter passing mechanisms popularly known as *call-by-value*, *call-by-name* and *call-by-need*.

The concepts of Mosses' semantic algebra (see Mosses [1980, 1981, 1984]) are introduced and further developed in Chapter 5. This algebraic approach is advocated as a useful enhancement of denotational semantics allowing semantic definitions to be clearer, concise and more readable. Given an implementation of a basic semantic algebra, along with a language definition in terms of the algebra, the semantic definition may be directly executed so providing an implementation of the language.

Chapter 6 surveys the possible applications of the semantic algebra and outlines possible further developments.

Chapter 2
THE LAMBDA CALCULUS

2.1 Introduction

The λ-calculus language achieves all its power from the behaviour of functions. No clear distinction is made between functions and their arguments. Functions can be passed as arguments to other functions; and because functions and their arguments are regarded as equivalent, a symmetric notation is used for the operation of applying one to the other - rather than $f(x)$ we write fx.

We allow just one primitive operation : viz. application, denoted by juxtaposition (see below).

2.1.1 Axiom of Comprehension. If M is a valid expression which takes values in D, then there exists a function f:D′ → D such that $fx = M$ for all x∈D′.

2.1.2 Axiom of Extensionality. For two functions f,g:D′ → D, if $fx = gx$ for all x∈D′, then $f = g$.

2.1.3 Definition. The expressions of the lambda calculus are defined by the grammar:

```
<expression> ::=  <variable>
                | <expression> <expression>
                | λ <variable> . <expression>
                | ( <expression> )
```

Assume that there is an infinite sequence of variables (or names).

An expression of the form

<expression> <expression>

is called an application. Applications apply a function to some operand, and include function application in the form of expressions such as $(\lambda x.x)a$. By convention application associates to the left - e.g. $abc \equiv (ab)c$.

An expression of the form

λ <variable> . <expression>

is called an abstraction and represents the λ-calculus equivalent of a function. When an abstraction such as $\lambda x.E$ occurs (for some λ-

expression E), E is taken as extending as far as possible to the right, to the first unmatched closing bracket or to the end of the expression, whichever comes first - e.g.

$\lambda x.\lambda y.a \equiv \lambda x.(\lambda y.a)$;

$\lambda x.a\ b \equiv \lambda x.(a\ b)$; and

$(\lambda x.\lambda y.N)a\ b \equiv ((\lambda x.(\lambda y.N))a)b$, for some λ-expression N.

The production

(<expression>)

allows the arbitrary grouping of expressions.

Note that lower case letters are used for variables and upper case letters for arbitrary expressions, and in this notation all abstractions have only one argument.

A function g with the property that

$$g(x)\ (y) = f(x,y)$$

is called the curried version of f. Evidently curried functions can be used to represent functions of several variables.

A variable in an expression can occur in one of two ways : viz. free or bound. A bound variable is involved in an abstraction. When the abstraction is applied to some argument, it is the bound variables that get replaced by substitution.

2.1.4 Definition - free

(1) x occurs free in x (but not in any other variable).

(2) x occurs free in XY iff. it occurs free in X or Y (or both).

(3) x occurs free in $\lambda y.X$ iff. x and y are different variables and x occurs free in X.

(4) x occurs free in (X) iff. x occurs free in X.

2.1.5 Definition - bound

(1) No variable occurs bound in y, for any variable y.

(2) x occurs bound in XY iff. it occurs bound in X or Y (or both).

(3) x occurs bound in $\lambda y.X$ iff. x and y are the same variable or if x occurs bound in X.

(4) x occurs bound in (X) iff. x occurs bound in X.

Example - In the expression $(\lambda x.\lambda y.xy)ab$ x and y occur bound while a and b occur free.

A particular variable can be bound at one place in an expression but free at another place. A variable can also be free in some sub-expression, but bound in the overall expression.
Example - x is both bound and free in $(\lambda x.x)x$.

For an abstraction $\lambda x.M$, we call x the bound variable (or binder) of the abstraction, and expression M the body (or form) of the abstraction.

2.2 Function Application and Evaluation

Here we introduce the formal syntactic transformations (conversions) of the λ-calculus which allow the manipulation of λ-expressions.

2.2.1 Definition - Substitution. Let x be a variable and M and X expressions. Then $[M/x]X$ is the expression X' defined as follows.

1. If X is a variable, two cases occur :
   (a) If $X \equiv x$ then $X' \equiv M$.
   (b) If $X \not\equiv x$ then $X' \equiv X$.
2. If X is an application YZ then
   $$X' \equiv ([M/x]Y)\ ([M/x]Z)\ .$$
3. If X is an abstraction $\lambda y.Y$, two cases occur :
   (a) If $y \equiv x$ then $X' \equiv X$.
   (b) If $y \not\equiv x$:
      (i) If x does not occur free in Y or if y does not occur free in M, then $X' \equiv \lambda y.[M/x]Y$.
      (ii) If x does occur free in Y and y does occur free in M, then $X' \equiv \lambda z.\ [M/x]\ ([z/y]Y)$ where z is some variable not occuring free in M or Y.

This is the formal syntactic detail of substitution. Intuitively, the expression $X' \equiv [M/x]X$ is obtained by substituting M for x in X, with any necessary name changes to avoid confusion.

Now that substitution has been formally defined, let us consider conversion rules for performing transformations on λ-expressions. We write

$$X \ \mathbf{cnv}\ Y$$

to indicate that either side may be replaced by the other whenever one of them occurs as an expression or as a sub-expression in a larger expression. We adopt the following :

2.2.2 Conversion Rules

α - If y is not free in X then
$\lambda x.X$ **cnv** $\lambda y.[y/x]X$.

β - $(\lambda x.M)N$ **cnv** $[N/x]M$.

η - If x is not free in M then
$\lambda x.Mx$ **cnv** M .

Remembering that the λ-calculus is to simulate the behaviour of functions we note that :

- α-conversion is a renaming rule allowing a bound variable (corresponding to the formal parameter of a function) to be renamed without collision.
- Substitution of formal parameters for the corresponding actual parameters is simulated by β-conversion.
- η-conversion eliminates a redundant formal parameter from a function to give an equivalent function.

When used in the left-to-right direction, the β- and η-conversion rules both replace an expression containing an abstraction with some other (often simpler) expression. Such a conversion is therefore called reduction, and the expression replaced is called a redex : i.e.

$(\lambda x.M)N$ is a β-redex; and

$\lambda x.(Mx)$ is an η-redex if x is not free in M.

The symbol **red** is commonly used instead of **cnv** to indicate a reduction; thus A **red** B asserts that A may be transformed to B by one or more reduction steps, and possibly some α-conversions. An expression containing *no* redexes is said to be normal or in normal form.

Examples of reduction :

(a) $(\lambda a.a)b$ **red** b

- $\lambda a.a$ is the identity function.

(b) $(\lambda a.b)c$ **red** b

- $\lambda a.b$ is the constant function, always yielding b.

(c) $(\lambda x.xa)(\lambda y.y)$ **red** $(\lambda y.y)a$
red a

(d) $(\lambda x.\lambda y.y)a\ b$ **red** $(\lambda y.y)b$

red b

(e) $(\lambda x.\lambda y.yx)z(\lambda u.c)$ **red** $(\lambda y.yz)\lambda u.c$

red $(\lambda u.c)z$

red c

(f) $(\lambda x.\lambda y.yx)ab$ **red** $(\lambda y.ya)b$

red ba

- $\lambda x.\lambda y.yx$ reverses the order of its operands.

Try : $(\lambda x.\lambda y.yx)yz$

red $([y/x](\lambda y.yx))z$

$\equiv (\lambda z.\ [y/x]\ (\ [z/y]\ yx))z$

- by substitution rule 3(b)(ii).

red $(\lambda z.[y/x]\ (zx))z$

red $(\lambda z.(zy))z$

red zy - as expected.

(g) $(\lambda x.xx)(\lambda x.xx)$ **red** $(\lambda x.xx)(\lambda x.xx)$

- this has no normal form; evaluation never terminates and it is said to be irreducible.

(h) $(\lambda x.\lambda y.y)\ ((\lambda x.xx)(\lambda x.xx))$.

- attacking the left-most β-redex,

$(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))$ **red** $\lambda y.y$.

- on the other hand, on attacking the other β-redex as in (g), evaluation may never terminate.

Example (g) raises an important question. Is it possible that two different reduction sequences might terminate with different results? Further, is there an order of evaluation which is guaranteed to terminate, whenever a particular expression is reducible to normal form? These questions led to the following important results.

2.2.3 Church-Rosser Theorem 1. If X **cnv** Y then there exists an expression Z such that X **red** Z and Y **red** Z.

2.2.4 Corollary. No expression can be converted to two distinct normal forms (i.e. normal forms which are not α-convertible). Thus no two orders of evaluation can reduce to different normal forms, although some may not terminate at all - i.e. there is at most one normal form for any λ-expression.

2.2.5 Church-Rosser Theorem 2. If A **red** B, and B is in normal form, then there exists a normal order reduction from A to B, where normal order reduction is that which at each stage reduces the left-most redex in the expression. (No argument is evaluated unless needed.)

Normal order reduction is guaranteed to terminate with a normal form if any order of evaluation does, so applying normal order reduction :

$(\lambda x.\lambda y.y)\ ((\lambda x.xx)(\lambda x.xx))\ b$

red $(\lambda y.y)b$

red b .

Another order of evaluation is applicative order where the function and argument of an application are separately evaluated to normal form before the function is actually applied to the argument. The first Church-Rosser Theorem (2.3.3) means that whenever this order of evaluation terminates it will give the same result as normal order. However, in certain cases it is less powerful than normal order - e.g. for example (h) above it will never terminate since the argument of the application has no normal form. An advantage of applicative order is that when it does terminate it is often faster than normal order since it evaluates its arguments only once, before they are substituted into the body of the function. On the other hand, normal order must evaluate an argument as many times as its corresponding binder occurs in the body of the function, after the substitution. But this is the source of the extra power of normal order evaluation over all other forms of evaluation : "as many times as the binder occurs" sometimes means "*never*" when the binder does not occur in the body.

Normal order and applicative order β-reduction are sometimes respectively known as call-by-name and call-by-value function calling mechanisms. We discuss these mechanisms further in Chapter 3 where we indicate the preferred "call-by-name" semantics for the λ-calculus and in Chapter 4 where we develop an interpreter for the λ-calculus having the efficiency of call-by-value reduction along with the power of call-by-value.

2.3 Representation of some common functions and data types

Here we give direct *representations* in the λ-calculus of some commonly used functions and data types.

2.3.1 Boolean : Represent and denote true by

$$\mathbf{t} \equiv \lambda x.\lambda y.x$$

and false by

$$\mathbf{f} \equiv \lambda x.\lambda y.y \ .$$

$\mathbf{t}\ a\ b \equiv (\lambda x.\lambda y.x)a\ b$ **red** $(\lambda y.a)b$ **red** a

$\mathbf{f}\ a\ b \equiv (\lambda x.\lambda y.y)a\ b$ **red** $(\lambda y.y)b$ **red** b

Define not as

$$\text{Not} \equiv \lambda u.u\ \mathbf{f}\ \mathbf{t} \ .$$

Not $\mathbf{t} \equiv (\lambda u.u\ \mathbf{f}\ \mathbf{t})\mathbf{t}$ **red** $\mathbf{t}\ \mathbf{f}\ \mathbf{t}$ **red** $\mathbf{f}$

Not $\mathbf{f} \equiv (\lambda u.u\ \mathbf{f}\ \mathbf{t})\mathbf{f}$ **red** $\mathbf{f}\ \mathbf{f}\ \mathbf{t}$ **red** $\mathbf{t}$

Define and as

$$\text{And} \equiv \lambda u.\lambda v.u\ v\ \mathbf{f} \ .$$

And $X\ Y \equiv (\lambda u.\lambda v.u\ v\ \mathbf{f})XY$ **red** $X\ Y\ \mathbf{f}$

X	Y	And $X\ Y$
t	t	t t f **red** t
t	f	t f f **red** f
f	t	f t f **red** f
f	f	f f f **red** f

Similarly define or, equivalence and implication as

$$\text{Or} \equiv \lambda u.\lambda v.u\ \mathbf{t}\ v \ ;$$

$$\text{Equiv} \equiv \lambda u.\lambda v.uv(v\ \mathbf{f}\ \mathbf{t}) \ ; \text{ and}$$

$$\text{Impl} \equiv \lambda u.\lambda v.uv\ \mathbf{t} \ .$$

2.3.2 Pairs : Represent the pair $\langle X,Y\rangle$ as

$$\langle X,Y\rangle \equiv \lambda u.uXY \ .$$

A function to form a pair can be defined as

$$\text{Pair} \equiv \lambda a.\lambda b.\lambda u.uab \ .$$

Pair $XY \equiv (\lambda a.\lambda b.\lambda u.uab)XY$ **red** $\lambda u.uXY \equiv \langle X,Y\rangle$

Functions to select the first and second elements of a pair can also be defined as

$$I_1 \equiv \lambda z.z\ \mathbf{t} \ ; \text{ and}$$

$$I_2 \equiv \lambda z.z\ \mathbf{f} \ .$$

$I_1\ \langle X,Y\rangle \equiv (\lambda z.z\ \mathbf{t})(\lambda u.uXY)$

red $(\lambda u.uXY)$t

red t X Y

red X

I_2 $\langle X,Y\rangle \equiv (\lambda z.z\ \mathbf{f})(\lambda u.uXY)$

red $(\lambda u.uXY)$f

red f X Y

red Y

2.3.3 The Non-negative Integers : Represent and denote the non-negative integers $0,1,2,\ldots,n$ as

$$\overline{0} \equiv \lambda f.\lambda x.x \ ;$$

$$\overline{1} \equiv \lambda f.\lambda x.fx \ ;$$

$$\vdots$$

$$\overline{n} \equiv \lambda f.\lambda x.\underbrace{f(f(f\ldots(fx)\ldots))}_{n\ f's}$$

$$\equiv \lambda f.\lambda x.f^n x \ .$$

$\overline{n}AB = (\lambda f.\lambda x.f^n x)AB$

red $(\lambda x.A^n x)B$

red $A^n B$

i.e. $\overline{n}$ is the abstraction that applies its first argument n times to its second.

Define the successor function as

$$\text{Succ} = \lambda k.\lambda f.\lambda x.f(kfx) \ .$$

Succ $\overline{n} \equiv (\ \lambda k.\lambda f.\lambda x.f(kfx)\ )\ (\lambda f.\lambda x.f^n x)$

red $\lambda f.\lambda x.f((\lambda f.\lambda x.f^n x)\ fx)$

red $\lambda f.\lambda x.f(f^n x)$

$\equiv \lambda f.\lambda x.f^{n+1}x$

$\equiv \overline{n+1}$

So Succ $\overline{2}$ **red** $\overline{3}$ for example.

Consider,

$\overline{2}$ Succ $\equiv (\lambda f.\lambda x.f(fx))$ Succ

red $\lambda x.\text{Succ}(\text{Succ}\ x)$

$\equiv \lambda x.\text{Succ}^2 x$.

Now,

$\overline{2}$ Succ $\overline{n}$ **red** $(\lambda x.\text{Succ}^2 x)\overline{n}$

red $\text{Succ}^2\overline{n}$

$\equiv$ Succ(Succ $\overline{n}$)

red Succ $\overline{n+1}$

red $\overline{n+2}$.

So addition may be implemented as

Sum $\equiv \lambda m.\lambda n.m$ Succ n .

Sum $\overline{i}$ $\overline{j}$ $\equiv (\lambda m.\lambda n.m$ Succ $n)\overline{i}$ $\overline{j}$

red $\overline{i}$ Succ $\overline{j}$

red $\text{Succ}^i\overline{j}$

red $\overline{i+j}$

Implement a zero test as

IsZero $\equiv \lambda k.k$(t f)t .

IsZero $\overline{0} \equiv (\lambda k.k$(t f)t) $(\lambda f.\lambda x.x)$

red $(\lambda f.\lambda x.x)$(t f)t

red $(\lambda x.x)$t

red t

IsZero $\overline{n} \equiv (\lambda k.k$(t f)t) $(\lambda f.\lambda x.f^n x)$

red $(\lambda f.\lambda x.f^n x)$(t f)t

red $(\text{t f})^n$t

$\equiv$ t f ((t f)(...((t f)t)...))

red f

Multiplication is implemented as

Product $\equiv \lambda m.\lambda n.m(n$ Succ$)\overline{0}$.

Product $\overline{a}$ $\overline{b} \equiv (\lambda m.\lambda n.m(n$ Succ$)\overline{0})\overline{a}$ $\overline{b}$

red $\overline{a}(\overline{b}$ Succ$)\overline{0}$

red $(\overline{b}$ Succ$)^a\overline{0}$

red $(\overline{b}$ Succ$)^{a-1}\overline{b}$

red $(\overline{b}$ Succ$)^{a-2}\overline{b+b}$

.
.
.

red $\overline{a \times b}$

2.3.4 Recursive Definitions : Consider the definition for the factorial function

$$\text{Fact} \equiv \lambda n.(\text{IsZero } n)\ \overline{1}\ (\text{Product } n\ (\text{Fact } (\text{Pred } n)))$$

where Pred[1] maps integer representations to their predecessors.This equation must be solved for Fact. Supposing

$$\text{H} \equiv \lambda f.\lambda n.(\text{IsZero } n)\ \overline{1}\ (\text{Product } n\ (f\ (\text{Pred } n)))$$

then it is required that

$$\text{Fact } \mathbf{cnv}\ \text{H Fact}\ .$$

i.e. Fact must be a <u>fixed point</u> of H.

Recall that

$$(\lambda x.xx)\ (\lambda x.xx)\ \mathbf{cnv}\ (\lambda x.xx)\ (\lambda x.xx)\ .$$

Some D is required so that DD is a fixed point of H.

i.e. so that DD **cnv** H(DD).

Define

$$\text{D} \equiv \lambda x.\text{H}(xx)\ .$$

Thus,

$$\text{DD} \equiv (\lambda x.\text{H}(xx))(\lambda x.\text{H}(xx))$$
$$\mathbf{cnv}\ \text{H}\ (\ (\lambda x.\text{H}(xx))\ (\lambda x.\text{H}(xx))\ )$$
$$\equiv \text{H(DD)}\ .$$

Abstracting on H produces

$$\text{Y} \equiv \lambda h.(\lambda x.h(xx))(\lambda x.h(\ xx\ ))\ ,$$

which renders a fixed point of any expression when applied to that expression. Thus,

$$\text{Fact} \equiv \text{YH}$$

may be taken as a solution for the factorial function.

<u>Conditional</u> expressions here have taken the form

$$\lambda x.(Px)(Ax)(Bx)$$

where P is an expression which, when applied to suitable operands, evaluates to t or f.

$$(\lambda x.(Px)(Ax)(Bx))\ z\ \mathbf{red}\ (Pz)(Az)(Bz)$$
$$\mathbf{red} \begin{cases} Az \text{ if } Pz\ \mathbf{red}\ \mathbf{t} \\ Bz \text{ if } Pz\ \mathbf{red}\ \mathbf{f} \end{cases}$$

---

1. A possible expression for Pred is

$$\text{Pred} \equiv \lambda k.(k\ (\lambda p.\lambda u.u(\text{Succ}(p\ \mathbf{t}))(p\ \mathbf{t}))\ (\lambda u.u\ \overline{0}\ \overline{0}))\ \mathbf{f}\ .$$

2.4 Summary

In this chapter we have introduced nothing more than a *syntactic* description of the behaviour of functions. The λ-expressions in 2.3 above have no true meaning other than an arbitrary interpretation we choose to give them. Although the expressions of the λ-calculus may be formally manipulated using the conversion rules we cannot claim that these syntactic transformations truly capture the meaning of functions. However, it is no coincidence that the transformations permitted by the conversion rules do seem to mirror the syntactic behaviour of functions - this behaviour has been deliberately constructed. In the following chapters we will proceed to give a mathematical semantics (or meaning) to each of the possible syntactic expressions of the λ-calculus.

Chapter 3
DENOTATIONAL SEMANTICS

3.1 Introduction

This chapter introduces the basic notions of the denotational (or mathematical or functional) specification of semantics for programming languages developed by Dana Scott and Christopher Strachey, and gives a formal semantic definition of the λ-calculus (itself a simple functional programming language - cf. Chapter 2). The underlying theoretical aspects of the denotational technique comprise Scott's approach to the theory of computation, involving continuous functions on continuous lattices, and his models for the value spaces needed in programming language semantics. This theory is outlined briefly in Milne & Strachey [1976], Stoy [1982] and in Stoy's chapter 3 of Bjørner & Jones [1982]. It is discussed in more depth by Scott [1976] and Stoy [1977]. Further references to the λ-calculus and Scott's definitions are to be found in Wadsworth [1976,1978].

An elementary introduction to denotational semantics is that of Tennent [1976], who includes a number of example applications. The more detailed texts are Stoy [1977] and the definitive work by Milne & Strachey [1976]. Gordon [1979] is an introductory text with a non-mathematical approach.

An interesting further development by Scott [1982] acknowledges the mathematical difficulty of the lattice-theoretic approach, and envisages a different approach to constructing the domains required for denotational semantics using a so-called information system - see also Larsen & Winskel [1984]. Scott [1982] also refers to another approach, that of neighbourhood systems, considered by Stoy [1982] and Bjørner & Jones [1982].

The basic approach of denotational semantics is to define semantic evaluation functions, which map syntactic constructs of the language into the abstract values which they denote. A construct's denotation is usually specified in terms of the values denoted by its syntactic subcomponents. The results of these functions may be able to be worked out in an obvious way, suggesting a possible implementation. In the case of the λ-calculus the concepts of function and functional application and abstraction are to be modelled. Definition techniques

rarely use the basic theory explicitly, and so it is discussed only briefly here, being a large subject in itself.

3.2 Mathematical Foundations

Semantic models typically use functions of higher order, called functionals - i.e. functions whose arguments or results are functions or other infinite objects. However, higher-order functions in semantic models can be mathematically troublesome. Some areas of difficulty may be identified as follows.

Recursion : Recall the definition of the factorial function in 2.3.4. The method for obtaining an expression for the factorial function involves finding a fixed point of the function H; however there is no guarantee that such a fixed point actually exists. Furthermore, given that there is a fixed point, there may exist more than one, so which one does YH produce? Recursive definitions of functions must be permitted. (Also the arguments and results of such functions may be functions or functionals, or other infinite and recursively defined objects).

Self-application : Returning once again to the definition of the factorial function, notice that in the definition

$$Y \equiv \lambda h.(\lambda x.h(xx))(\lambda x.h(xx))$$

x is applied to itself. Self-application can lead to paradoxes.

Example

Define $U \equiv \lambda y.$ if $y(y)=a$ then b else a ,

but attempting to evaluate $U(U)$ gives

$$U(U) \equiv \text{if } U(U)=a \text{ then } b \text{ else } a \ ,$$

which is a contradiction.

Moreover, self-application implies that every expression may denote a function, requiring that the space of denotable values be isomorphic to the space of functions on those values - i.e. to each value there corresponds a unique function and vice versa.

Example

Suppose the space of functions was defined to be

$$F = D \rightarrow E$$

where the domain D of denotable values also includes the functions F. Then instead of recursion in the definition of the mapping itself, there is now circularity in the

specification of its domain. This last specification does not have a straightforward solution.

Infinitary Objects : Many functions are infinitary objects - i.e. they contain an infinite amount of information (e.g. the mapping of all the integers into their successors). Infinitary objects must be dealt with as the limit of a set of finite approximations.

The mathematical theory of computation developed by Scott, using ideas from lattice theory and topology, provides satisfactory solutions to these problems. Along with the more detailed expositions of this theory in the references, the brief overview now presented provides reassurance that the semantic models to be used here will be mathematically sound, as long as they are limited to suitably defined functions and domains.

3.2.1 Basic Concepts

A class of "data types" termed domains, and a class of functions (including those of higher order) rich enough to allow natural models of computation (including recursion and self-application) and finite approximations, yet avoiding the above difficulties, can be defined. In a Scott domain a sequence of better and better approximations converge to a well-behaved limit which is also in the domain. All "operations" defined on the data type must be continuous functions in order to preserve these limits.

Primitive domains may be formed by adjoining to finite or denumerable sets such as $\{true,false\}$, or $\{..., -2, -1, 0, 1, 2,...\}$ three special objects "$\perp_D$" (termed bottom, being the the "under-defined" element of the respective domain D, containing no information), "$\top_D$" (termed top, being the element which is "over-defined", containing inconsistent information) and "$?_D$" (an "error" element). $\perp$, $\top$ and ? are called the improper elements of the domain D, all other elements being proper. The following may be considered as primitive domains :

$N = \{..., -2, -1, 0, 1, 2,...\}^\circ$	integers
$T = \{true,false\}^\circ$	truth values
$H = \{"a","b",...\}^\circ$	characters

where $\{...\}^\circ$ denotes the augmentation of the set $\{...\}$ by $\perp$ and $\top$. In such domains the notion of approximation is simple :

$\perp$ approximates all elements, all elements approximate $\top$, and all other pairs are incomparable - i.e. neither element of the pair approximates the other. Hence there are no nontrivial limits or recursive definitions of elements in primitive domains. The error element ? gives a natural value for use as a result for computations that go wrong.

Non-primitive domains may be constructed in a number of ways. If D and D_i ($i \geq 0$) are *any* domains, then the following are also domains :

(i)	$D_0 \times D_1 \times \ldots \times D_n$ $(n \geq 1)$	product domain
(ii)	$D_0 + D_1 + \ldots + D_n$ $(n \geq 1)$	sum domain
(iii)	$D_1 \rightarrow D_2$	function domain
(iv)	$D^n = \underbrace{D \times D \times \ldots \times D}_{n\ D's}$	domain of lists of length n
(v)	$D^* = D^0 + D^1 + D^2 + \ldots$	domain of finite lists

Except for the special treatment of $\perp$, $\top$ and ?, the elements of $D_0 \times D_1 \times \ldots \times D_n$ correspond to finite lists of length n, having a typical proper element written as $\langle \delta_0, \ldots, \delta_n \rangle$ (where $\delta_m \in D_m$ when $n \geq m \geq 0$). An element of $D_0 + D_1 + \ldots + D_n$ corresponds to an element of either D_0 or D_1 or any of the other constituent domains. The domain $D_1 \rightarrow D_2$ consists of continuous functions from D_1 to D_2. D^n is the domain of n-tuples of elements of D. The domain D^0 has only one proper element - i.e. the empty list $\langle\rangle$. A typical proper member of D^1 is a list of length one written $\langle \delta \rangle$ ($\delta \in D$). A typical proper element of D^n is written $\langle \delta_0, \ldots, \delta_n \rangle$ (for $\delta_m \in D$ when $n \geq m \geq 0$). D^* is the domain of all the finite lists of elements from D, having a typical proper element δ^*. All of these constructed domains contain the special elements $\perp$, $\top$ and ?, along with approximation relations derived from those of the constituent domains - e.g. for function domain $D_1 \rightarrow D_2$, f approximates g when $f(x)$ approximates $g(x)$ for all $x \in D_1$.

A domain definition may use several of the constructions. The binary operators have a precedence convention, with "$\times$" having the highest precedence and "$\rightarrow$" the lowest. Association is to the right. For a sum domain $X = \ldots + Y + \ldots$ the following suffix notations for operations of inspection, projection and injection are used :

(i) for $\chi \in X$,

$$\chi \in Y \equiv \begin{cases} true, & \text{if } \chi \text{ corresponds to some element } \gamma \in Y \\ false, & \text{if } \chi \text{ corresponds to an element of some other summand domain} \\ \bot_T,\ \top_T \text{ or } ?_T, & \text{if } \chi \equiv \bot_X,\ \top_X \text{ or } ?_X \end{cases}$$

(ii) for $\chi \in X$,

$$\chi | Y \equiv \begin{cases} \gamma, & \text{if } \chi \text{ corresponds to } \gamma \in Y \\ ?_Y, & \text{if } \chi \text{ corresponds to an element of some other summand domain} \\ \bot_Y,\ \top_Y \text{ or } ?_Y, & \text{if } \chi \equiv \bot_X,\ \top_X \text{ or } ?_X \end{cases}$$

(iii) for $\gamma \in Y$,

γ in $X \equiv \chi$, where $\chi \in X$ corresponds to γ.

Constant and identity functions over any domain are continuous, and any function defined by abstractions and combinations is continuous when the constituent subexpressions define continuous functions on domains.

On primitive domains the requirement of continuity reduces to monotonicity. A function f is monotone if $f(x)$ approximates $f(y)$ when x approximates y. It is clearly reasonable to assume that given more information about the argument of a function, we obtain at least as much information (if not more) about its result.

3.2.2 Recursion

In 3.2 we indicated the difficulties encountered when we wish to give a mathematical meaning to a recursive function - i.e. the existence and identity of the fixed point solution. We need a "fixed-point-finding" function *fix* that produces an appropriate fixed point solution to recursive equations of the form $f = F(f)$, where F is a higher-order transformation $F: D \rightarrow D$ for some domain D. Since D is a domain it has an approximation ordering relation defined on it, and a "worst" element $\bot_D$. By monotonicity,

$$F(\bot) \text{ approximates } F(F(\bot));$$

and by induction

$$\bot,\ F(\bot),\ldots,\ F^i(\bot),\ldots$$

is a sequence of better and better approximations converging (by continuity) to a limit f such that $F^i(\bot)$ approximates f for all $i \geq 0$

and $F(f) = f$. It has been shown (see Scott [1976], Stoy [1977]) that for any domain D there is a *continuous* fixed point function fix:[D $\rightarrow$ D] $\rightarrow$ D such that for any continuous F:D $\rightarrow$ D :

(i) $fix(F) = \lim_{i \rightarrow \infty} F^i(\perp)$

exists and is a solution of the equation $f = F(f)$; and

(ii) any other solution of the equation is approximated by $fix(F)$.

By using the function fix we can be confident that a solution to a recursive equation of the form $f = F(f)$ exists and that this solution is the minimal or least fixed point of the equation. Note that the λ-expression Y (used in 2.3.4) actually denotes the least fixed point function fix (for a proof see Stoy [1977], p.127).

Scott resolved the paradox arising from self-application (as outlined in 3.2) by showing that domains, as well as domain elements, may be recursively defined (see Scott [1976]). A domain is required that is isomorphic with its own function space. Such a domain may be constructed explicitly. Alternatively, a suitable candidate may be found in which the required domain may be embedded. The approach used by Scott [1976] exposes the complete lattice $P\omega$ (i.e. the power-set of the non-negative integers ordered by the set inclusion relation $\subseteq$) as a domain in which every one of our required domains may be embedded as a subspace. Domains which contain themselves are known as reflexive domains and are specified by reflexive domain equations. Fixed point solutions to these equations are constructed by a limiting process (using fix) just as fixed point solutions to recursive function definitions are obtained.

3.3 Semantics of the λ-Calculus

The Church-Rosser theorems ensure that the formal syntactic system of the λ-calculus is self-consistent. It remains to show that the λ-calculus can describe the behaviour of functions.

3.3.1 Syntax

Firstly, the syntax of the λ-calculus is specified as the domain of semantic interpretation :

Syntactic Categories

I $\in$ Ide (usual identifiers)

$E \in$ **Exp** (λ-expressions)

Syntax

$E ::= I \mid \lambda I.E \mid E_0E_1$

In the language there are two kinds of objects : identifiers and expressions. **Ide** is the set of "usual" identifiers. The symbol 'λ' stands for itself. The syntax is presented in a variant of Backus-Naur form, where Greek capital letters (here I and E) are used for syntactic variables instead of names in angle brackets. In this case E is an element of the set of λ-expressions, called **Exp**. The production E_0E_1 says that an expression can consist of two expressions in juxtaposition. Subscripts allow us to refer to components separately.

Notes on syntax :
(i) Syntax may be thought of as occurring in two forms. Generative syntax tells how expressions may be constructed from their components, whereas analytic or abstract syntax tells how to analyse a given expression, to say what kind of expression it is and to extract its components. Abstract syntax involves a set of predicate functions and selector functions :

Predicates	Corresponding Selectors
Is_Variable(s)	Name_Of(s)
Is_Lambda_Abstraction(s)	Binder_Of(s), Form_Of(s)
Is_Application(s)	Function_Of(s), Argument_Of(s)
Is_Lambda_Term(s)	

These constitute the parser of any interpreter for the language.

(ii) It is often convenient to model these predicates and selectors by means of parse trees (cf. the graph description used in the interpreter to follow in Chapter 4). Each production in the language corresponds to a particular kind of node, determined from a special label attached to it (cf. VAR, APP and LAM used in the interpreter). The components of a production are represented by branches from the node, each with appropriate selector names. In this model the parser is the program mapping strings in the language into the appropriate tree (the function REPRESENT in the interpreter). There are also the

appropriate selectors for each of the three possible kinds of expression node (cf. NAM_OF, FUN_OF, ARG_OF, BND_OF, FRM_OF).

(iii) The problem of ambiguity in parsing may also be ignored. If the given syntax is taken to be the actual grammar of the language, then it is ambiguous : $E_1E_2E_3$ has two possible parses. Let us simply assume that the parser has sorted out such ambiguities by using precedence and association rules, parentheses etc. The conventions of application associating to the left and the bracketing rule for λ-abstractions have already been given. These same rules are used in the interpreter, so the user need not submit a fully-bracketed term for reduction.

3.3.2 Semantics

The language semantics define mappings from the syntactic categories into a domain of values of expressions, which first of all must be defined. There are various ways to choose a domain. It may contain only functions, so that

$$\mathbf{E} = [\mathbf{E} \rightarrow \mathbf{E}].$$

This does not include primitive values, and is appropriate since there is no way in the current syntax of expressing any values other than functions. Yet this domain is not reminiscent of the value domains of ordinary programming languages, which usually include primitive values such as truth values and numbers, as well as functions, procedures etc. A domain which does include primitive values might be defined as

$$\mathbf{E} = \mathbf{B} + [\mathbf{E} \rightarrow \mathbf{E}]$$

where B is the domain of primitive or basic values - e.g. B could be a sum domain with components such as N, T and H. For the moment, let us consider the first of these domains (consisting of functions only).

An expression has a value which depends on its environment which gives the meaning of the identifiers in the expression - i.e. the values the identifiers denote. An environment is specified by a function from Ide to a domain D of denotable values (sometimes called denotations). In the λ-calculus D happens to be the same as E, the domain of expression values. Let us write $\rho[\![I]\!]$ for the value denoted by I in the environment ρ. Our second semantic domain is given by

$$\mathbf{U} = [\mathbf{Ide} \rightarrow \mathbf{E}].$$

In summary, with ϵ taking values in E and ρ denoting environments in U, write :

Value Domains

$\epsilon \in \mathbf{E} = [\mathbf{E} \rightarrow \mathbf{E}]$ Values of expressions.

$\rho \in \mathbf{U} = [\mathbf{Ide} \rightarrow \mathbf{E}]$ Environments.

We now consider the semantic evaluation functions for the λ-calculus. These map syntactic constructs into the semantic domain values they denote.

First we define a function for producing new environments. The value denoted by an identifier depends on its context, and is determined by scope rules. This is formalised by giving each different scope a separate ρ. New environments are formed out of old by the notation

$$\rho[\delta/I].$$

The environment $\rho[\delta/I]$ differs from ρ only in that it maps I to δ (whether or not ρ mapped I to any other value at all).

3.3.3 Definition - Conditional expression.

For $b \in T$ (the domain of truth-values) and $x, y \in D$, the expression

$b \rightarrow x,y$ has value $\begin{cases} x \text{ if } b \text{ is } \textit{true} \\ y \text{ if } b \text{ is } \textit{false} \\ \bot_D,\ \top_D \text{ or } ?_D \text{ if } b \text{ is } \bot_D,\ \top_D \text{ or } ?_D. \end{cases}$

3.3.4 Definition. The environment $\rho[\delta/I]$ is defined by :

$$(\rho[\delta/I])\ \llbracket I' \rrbracket \equiv (I' = I) \rightarrow \delta,\ \rho\llbracket I' \rrbracket.$$

$\rho[\delta/I]$ denotes an environment mapping. Here ρ continues to denote the same environment, and $\rho[\delta/I]$ denotes another environment, usually different from ρ.

Let us now define the semantic function

$$E : [\mathbf{Exp} \rightarrow [\mathbf{U} \rightarrow \mathbf{E}]] ,$$

which relates expressions to their values. If $E\llbracket E \rrbracket\rho$ denotes the value of the expression E in the environment ρ, E is defined by cases which match the syntactic description of **Exp** - i.e.

Semantic Functions

$$E\llbracket I \rrbracket\rho \equiv \rho\llbracket I \rrbracket \quad (1)$$

$$E\llbracket \lambda I.E \rrbracket\rho \equiv \lambda\epsilon.E\llbracket E \rrbracket(\rho[\epsilon/I]) \quad (2)$$

$$E\llbracket E_0 E_1 \rrbracket\rho \equiv (E\llbracket E_0 \rrbracket\rho)(E\llbracket E_1 \rrbracket\rho) \quad (3)$$

where the brackets ⟦⟧ enclose syntactic objects in the language. The notation $\rho[\epsilon/I]$ is mixed, the I denoting a syntactic object and the others semantic values. The right hand side of each equation specifies a value in **E**.

3.3.5 Extending the λ-calculus to Include Basic Values

By defining the semantic domain for expressible values of the λ-calculus so as to include basic values - i.e.

$$\mathbf{E} = \mathbf{B} + [\mathbf{E} \rightarrow \mathbf{E}] \quad ,$$

the semantics may be altered as follows.

To permit some way of actually expressing values from **B**, one may introduce :

a new syntactic category of bases : $B \in \mathbf{Bas}$;

a new production to the syntax of expressions : E ::= B ;

a new semantic function B : $\mathbf{Bas} \rightarrow \mathbf{E}$; and

a new semantic equation in the definition of E :

$$E⟦B⟧\rho = B⟦B⟧ \ .$$

In addition, the domain of expressible values now consists of two components (basic values and functions), so equations (2) and (3) must be altered. A form of "coercion" using injection is used to make sure that all functions formed by λ-abstraction are actually elements of the overall domain **E** of expression values rather than just the domain of functions $[\mathbf{E} \rightarrow \mathbf{E}]$. Similarly projection is used to ensure that only the function value $(\epsilon[\mathbf{E} \rightarrow \mathbf{E}])$ corresponding to a given expression $(\epsilon\mathbf{E})$ is ever applied as a function :

$$E⟦\lambda I.E⟧\rho \equiv (\lambda\epsilon.E⟦E⟧\ (\rho[\epsilon/I]))\ \text{in}\ E$$

$$E⟦E_0E_1⟧\rho \equiv (E⟦E_0⟧\rho|[\mathbf{E} \rightarrow \mathbf{E}])(E⟦E_1⟧\rho)$$

The detail of the semantic definition for the basic values has been omitted, but could include definitions for the integers, truth values etc. The value of a base given by B is not restricted to the **B** summand of **E**, allowing the basic values to come equipped with their own primitive functions $(\epsilon[\mathbf{E} \rightarrow \mathbf{E}])$. The semantics of the λ-calculus with basic values is summarised at the end of this chapter.

3.4 Implementation : Conversion Rules

This leads to a proof that the conversion rules of the λ-calculus provide a correct implementation of the language, so that no allowable

conversion will alter the value denoted by an expression - i.e. the proof that :

$$E_0 \text{ cnv } E_1 \Rightarrow (E\llbracket E_0 \rrbracket\rho \equiv E\llbracket E_1 \rrbracket\rho) \qquad \text{(for all } \rho\text{)}.$$

Note : If $A\rho \equiv B\rho$ for all ρ, then A is equivalent to B, by extensionality.

3.4.1 Definition. $E_0 \equiv E_1$ means $E\llbracket E_0 \rrbracket \equiv E\llbracket E_1 \rrbracket$.

3.4.2 Theorem. E_0 cnv $E_1 \Rightarrow E_0 \equiv E_1$.

Recall the explicit conversion rules :

α - If y is not free in X then
$\lambda x.X$ cnv $\lambda y.[y/x]X$.

β - $(\lambda x.M)$ cnv $[N/x]M$.

η - If x is not free in M then
$\lambda x.Mx$ cnv M .

Two expressions E_0 and E_1 are convertible if either :

(a) one may be transformed into the other by direct application of one of the conversion rules;

(b) one may be transformed into the other by application of a rule to one of its subcomponents;

(c) one may be transformed into the other by a finite sequence of permissible conversions.

In all three cases it must be shown that $E_0 \equiv E_1$.

Proof : Case (c) is easily dealt with. The symbol $\equiv$ denotes an equivalence relation - i.e. symmetric, transitive and reflexive. Thus if equivalence is preserved by each step singly, then it is preserved by the whole sequence, so only the validity of each single step need be proved.

For case (b) induction is used on the structure of the expressions E_0 and E_1. Each possible structure for expressions is considered separately. The result is clearly trivial for simple identifiers. So it remains to prove :

(i) if E cnv E′ then λI.E $\equiv$ λI.E′

and (ii) if E_2 cnv E_2' and E_3 cnv E_3' then $E_2E_3 \equiv E_2'E_3'$.

(i) $E \equiv E'$ is assumed by induction.

So $E\llbracket E \rrbracket\rho = E\llbracket E' \rrbracket\rho$ (for all ρ)

In particular $E\llbracket E \rrbracket(\rho[\epsilon/I]) = E\llbracket E' \rrbracket(\rho[\epsilon/I])$ (for all ϵ,ρ)

By extensionality,

$$\lambda\epsilon.E\llbracket E\rrbracket(\rho[\epsilon/I]) = \lambda\epsilon.E\llbracket E'\rrbracket(\rho[\epsilon/I]) \qquad \text{(all } \rho\text{)}$$

So, by (2),

$$E\llbracket\lambda I.E\rrbracket\rho = E\llbracket\lambda I.E'\rrbracket\rho \qquad \text{(all } \rho\text{)}$$

i.e. $\lambda I.E \equiv \lambda I.E'$.

(ii) Similarly, assuming $E_2 \equiv E_2'$ and $E_3 \equiv E_3'$ by induction, then

$$(E\llbracket E_2\rrbracket\rho)(E\llbracket E_3\rrbracket\rho) = (E\llbracket E_2'\rrbracket\rho)(E\llbracket E_3'\rrbracket\rho) \qquad \text{(all } \rho\text{)}$$

By (3),

$$E\llbracket E_2E_3\rrbracket\rho = E\llbracket E_2'E_3'\rrbracket\rho \qquad \text{(all } \rho\text{)}$$

i.e. $E_2E_3 \equiv E_2'E_3'$.

Finally (a) is proved - i.e. that the explicit rules preserve equivalence :

3.4.3 Lemma. If I is not free in E then

$$E\llbracket E\rrbracket(\rho[\delta/I]) = E\llbracket E\rrbracket\rho \qquad \text{(for all } \rho,\delta\text{)}$$

Proof : (by structural induction)

If E is an identifier, not equal to I, then the result is trivial.

If E is the expression $\lambda I_1.E'$ then assume that I is not free in E' and that the result holds for E' by induction.

So by extensionality,

$$\lambda\epsilon.E\llbracket E'\rrbracket(\rho[\delta/I]) = \lambda\epsilon.E\llbracket E'\rrbracket\rho \qquad \text{(all } \rho,\delta,\epsilon\text{)}$$

By (2),

$$E\llbracket\lambda\epsilon.E'\rrbracket(\rho[\delta/I]) = E\llbracket\lambda\epsilon.E'\rrbracket\rho \qquad \text{(all } \rho,\delta,\epsilon\text{)}$$

i.e. in particular,

$$E\llbracket\lambda I_1.E'\rrbracket(\rho[\delta/I]) = E\llbracket\lambda I_1.E'\rrbracket\rho \qquad \text{(all } \rho,\delta\text{)}$$

If E is the expression E_1E_2, then assume that I is not free in E_1,E_2 and that the result holds for both E_1 and E_2.

By (3),

$$\begin{aligned} E\llbracket E_1E_2\rrbracket(\rho[\delta/I]) &= (\,E\llbracket E_1\rrbracket(\rho[\delta/I])\,)\,(\,E\llbracket E_2\rrbracket(\rho[\delta/I])\,) \\ &= (E\llbracket E_1\rrbracket\rho) \\ &= (E\llbracket E_2\rrbracket\rho) && \text{(by assumption)} \\ &= E\llbracket E_1E_2\rrbracket\rho && \text{(by (3))} \end{aligned}$$

3.4.4 Substitution Lemma. For all ρ,

$$E\llbracket[E_1/I]E_0\rrbracket\rho = E\llbracket E_0\rrbracket(\rho[E\llbracket E_1\rrbracket\rho/I]).$$

Define

E_0' to mean $[E_1/I]E_0$ (4)

and

ρ' to mean $\rho[E\llbracket E_1\rrbracket\rho/I]$. (5)

The result required is that

$$E[\![E_0']\!]\rho \equiv E[\![E_0]\!]\rho'.$$

Let us call the left- and right-hand-sides of this assertion L and R respectively.

Proof : Once again, structural induction is used, this time on the structure of E_0. The cases are based on the definition of substitution (2). The expression E_0' is defined :

1. E_0 is a variable :
   (a) If $E_0 \equiv I$ then $E_0' \equiv E_1$.
   (b) If $E_0 \not\equiv I$ then $E_0' \equiv E_0$.
2. E_0 is an application E_2E_3 :
   $E_0' \equiv ([E_1/I]E_2)([E_1/I]E_3)$.
3. E_0 is an abstraction $\lambda I_1.E_2$:
   (a) If $I_1 \equiv I$ then $E_0' \equiv E_0$.
   (b) If $I_1 \not\equiv I$ then
      (i) If I does not occur free in E_2 or I_1 does not occur free in E_1 then $E_0' \equiv \lambda I_1.[E_1/I]E_2$;
      (ii) Otherwise $E_0' \equiv \lambda I_2.[E_1/I]([I_2/I_1]E_2)$ where I_2 is a new variable which does not occur free in E_1 or E_2.

1. E_0 is a variable.

(a) If E_0 is I then E_0' is E_1.

So L = $E[\![E_1]\!]\rho$.

R = $E[\![I]\!]\rho'$

= $\rho'[\![I]\!]$ (by (1))

= $(\rho[E[\![E_1]\!]\rho/I])[\![I]\!]$ (definition of ρ')

= $E[\![E_1]\!]\rho$. (by 3.3.4)

So L = R.

(b) If E_0 is I_1, not I, then E_0' is E_0, or I_1.

L = $E[\![I_1]\!]\rho = \rho[\![I_1]\!]$. (by (1))

R = $E[\![I_1]\!]\rho' = \rho'[\![I_1]\!]$

= $(\rho[E[\![E_1]\!]\rho/I])[\![I_1]\!]$

= $\rho[\![I_1]\!]$. (by 3.3.4 since $I \not\equiv I1$)

So L = R.

2. E_0 is E_2E_3.

Then E_0' is $E_2'E_3'$ where E_2' is $[E_1/I]E_2$ and E_3' is $[E_1/I]E_3$.

$L = E\llbracket E_2' E_3' \rrbracket \rho$

$= (E\llbracket E_2' \rrbracket \rho)(E\llbracket E_3' \rrbracket \rho)$ (by (3))

$= (E\llbracket E_2 \rrbracket \rho')(E\llbracket E_3 \rrbracket \rho')$ (by the inductive hypothesis)

$= E\llbracket E_2 E_3 \rrbracket \rho'$

$= R.$

3. E_0 is $\lambda I_1.E$.

So $R = E\llbracket \lambda I_1.E \rrbracket \rho'$

$= \lambda\epsilon.E\llbracket E \rrbracket(\rho'[\epsilon/I_1])$ (by (2))

(a) If I_1 is I then E_0' is E_0.

So $L = E\llbracket \lambda I.E \rrbracket \rho$

$= \lambda\epsilon.E\llbracket E \rrbracket(\rho[\epsilon/I])$ (by (2))

and $R = \lambda\epsilon.E\llbracket E \rrbracket(\rho'[\epsilon/I])$ (since I_1 is I)

But $\rho'[\epsilon/I] = \rho[E\llbracket E_1 \rrbracket \rho/I][\epsilon/I]$ (defn. of ρ')

$= \rho[\epsilon/I].$ (from 3.3.4)

So $L = R$.

(b) If I_1 is not I then E_0' is $\lambda I_2.[E_1/I][I_2/I_1]E$, where I_2 is specified below.

Let $E_2 = [I_2/I_1]E$, so that E_0' is $\lambda I_2.[E_1/I]E_2$.

Then $L = E\llbracket \lambda I_2.[E_1/I]E_2 \rrbracket \rho$

$= \lambda\epsilon.E\llbracket [E_1/I]E_2 \rrbracket(\rho[\epsilon/I_2])$ (by (2))

$= \lambda\epsilon.E\llbracket E_2 \rrbracket \rho''_\epsilon$ (by hypothesis)

where $\rho''_\epsilon = \rho[\epsilon/I_2][E\llbracket E_1 \rrbracket(\rho[\epsilon/I_2])/I]$.

So, by substituting the definition of E_2,

$L = \lambda\epsilon.E\llbracket [I_2/I_1]E \rrbracket \rho''_\epsilon$

$= \lambda\epsilon.E\llbracket E \rrbracket(\rho''_\epsilon[E\llbracket I_2 \rrbracket \rho''_\epsilon/I_1])$ (by hyp.)

$= \lambda\epsilon.E\llbracket E \rrbracket(\rho''_\epsilon[\rho''_\epsilon\llbracket I_2 \rrbracket/I_1])$ (by (1))

and also

$R = \lambda\epsilon.E\llbracket E \rrbracket(\rho'[\epsilon/I_1]).$ (by (2))

In order to proceed I_2 must be known.

(i) If I is not free in E or if I_1 is not free in E_1, then I_2 is I_1. In both cases $\rho''_\epsilon = \rho[\epsilon/I_1](E\llbracket E_1 \rrbracket(\rho[\epsilon/I_1])/I]$.

So $\rho''_\epsilon\llbracket I_2 \rrbracket = \rho''_\epsilon\llbracket I_1 \rrbracket = \epsilon$ (by 3.3.4 since $I \neq I_1$)

So $L = \lambda\epsilon.E\llbracket E \rrbracket(\rho''_\epsilon[\epsilon/I_1])$ (substituting for $\rho''_\epsilon\llbracket I_2 \rrbracket$)

$= \lambda\epsilon.E\llbracket E \rrbracket \rho''_\epsilon$ (since $\rho''_\epsilon\llbracket I_1 \rrbracket = \epsilon$)

Now either I is not free in E, in which case

$$E\llbracket E\rrbracket\rho''_\epsilon = E\llbracket E\rrbracket(\rho[\epsilon/I_1])$$

and $E\llbracket E\rrbracket(\rho'[\epsilon/I_1]) = E\llbracket E\rrbracket(\rho[\epsilon/I_1])$

and hence L = R;

or I_1 is not free in E_1, and then

$$E\llbracket E_1\rrbracket(\rho[\epsilon/I_1]) = E\llbracket E_1\rrbracket\rho$$

so that $\rho''_\epsilon = \rho[\epsilon/I_1][E\llbracket E_1\rrbracket\rho/I]$ (from previous eq. for ρ''_ϵ)

$= \rho'[\epsilon/I_1]$ (since $I_1 \neq I$ then the two postfixes commute)

So $L = \lambda\epsilon.E\llbracket E\rrbracket\rho''_\epsilon$

$= \lambda\epsilon.E\llbracket E\rrbracket(\rho'[\epsilon/I_1])$

$= R.$

(ii) If I is free in E and I_1 is free in E_1, then I_2 is a new identifier which is not free in E or E_1.

So $\rho''_\epsilon = \rho[\epsilon/I_2][E\llbracket E_1\rrbracket(\rho[\epsilon/I_2])/I]$ (by definition)

$= \rho[\epsilon/I_2][E\llbracket E_1\rrbracket\rho/I]$ (by 3.4.3, as I_2 is not free in E_1)

So $\rho''_\epsilon\llbracket I_2\rrbracket = \epsilon$ (as I_2 is not I)

and $\rho''_\epsilon[\rho''_\epsilon\llbracket I_2\rrbracket/I_1] = \rho''_\epsilon[\epsilon/I_1]$

$= \rho[\epsilon/I_2][E\llbracket E_1\rrbracket\rho/I][\epsilon/I_1].$

So $L = \lambda\epsilon.E\llbracket E\rrbracket(\rho[\epsilon/I_2][E\llbracket E_1\rrbracket\rho/I][\epsilon/I_1])$

$= \lambda\epsilon.E\llbracket E\rrbracket(\rho[E\llbracket E_1\rrbracket\rho/I][\epsilon/I_1])$

(by 3.4.3, as I_2 is not free in E)

$= \lambda\epsilon.E\llbracket E\rrbracket(\rho'[\epsilon/I_1])$ (by def. of ρ')

$= R.$

This completes the proof of Lemma 3.4.4, and we may proceed to complete the proof of Theorem 3.4.2..

It must be shown that

$$E_0 \text{ cnv } E_1 \Rightarrow E_0 \equiv E_1$$

for α-, β-, and η- conversion.

<u>α-conversion.</u>

$E_0 \equiv \lambda I.E$

$E_1 \equiv \lambda I'.[I'/I]E$ where I' is not free in E.

Then $E\llbracket E_1\rrbracket\rho \equiv \lambda\epsilon.E\llbracket[I'/I]E\rrbracket I(\rho[\epsilon/I'])$ (by (2))

$\equiv \lambda\epsilon.E\llbracket E\rrbracket(\rho[\epsilon/I'][E\llbracket I'\rrbracket(\rho[\epsilon/I'])/I])$ (by Lemma 3.4.4)

$\equiv \lambda\epsilon.E\llbracket E\rrbracket(\rho[\epsilon/I'][\epsilon/I])$ (by (1))

$\equiv \lambda\epsilon.E\llbracket E\rrbracket(\rho[\epsilon/I])$

(by Lemma 3.4.3 and 3.3.4, as I′ is not free in E)

$\equiv E\llbracket E_0\rrbracket\rho.$ (by (2))

This holds for all ρ, so $E_0 \equiv E_1$.

β-conversion.

$E_0 \equiv (\lambda I.E)E'$

$E_1 \equiv [E'/I]E.$

Then $E\llbracket E_0\rrbracket\rho \equiv (E\llbracket\lambda I.E\rrbracket\rho)(E\llbracket E'\rrbracket\rho)$ (by (3))

$\equiv (\lambda\epsilon.E\llbracket E\rrbracket(\rho[\epsilon/I]))(E\llbracket E'\rrbracket\rho)$ (by (2))

$\equiv E\llbracket E\rrbracket(\rho[E\llbracket E'\rrbracket\rho/I])$

$\equiv E\llbracket[E'/I]E\rrbracket\rho$ (by Lemma 3.4.4)

$\equiv E\llbracket E_1\rrbracket\rho.$

This holds for all ρ, so $E_0 \equiv E_1$.

η-conversion.

$E_0 \equiv \lambda I.E(I)$ where I is not free in E,

$E_1 \equiv E.$

Then $E\llbracket E_0\rrbracket\rho \equiv \lambda\epsilon.E\llbracket E(I)\rrbracket(\rho[\epsilon/I])$ (by (2))

$\equiv \lambda\epsilon.(E\llbracket E\rrbracket(\rho[\epsilon/I]))(E\llbracket I\rrbracket(\rho[\epsilon/I]))$ (by (3))

$\equiv \lambda\epsilon.(E\llbracket E\rrbracket\rho)(\epsilon)$ (by Lemma 3.4.3, as I is not free in E)-*

So for all ϵ',

$(E\llbracket E_0\rrbracket\rho)\epsilon' \equiv (\lambda\epsilon.(E\llbracket E\rrbracket\rho)(\epsilon))\epsilon'$

$\equiv (E\llbracket E\rrbracket\rho)\epsilon'$

$\equiv (E\llbracket E_1\rrbracket\rho)\epsilon'$

Thus by extensionality, $E_0 \equiv E_1$.

Suppose that the semantics for the λ-calculus with basic values were used, as summarised at the end of this chapter. Then the proof of Theorem 3.4.2 is no longer valid. The equation * above becomes

$$E\llbracket E_0\rrbracket \equiv \lambda\epsilon.(E\llbracket E\rrbracket\rho|[\mathbf{E} \rightarrow \mathbf{E}])(\epsilon) \text{ in } \mathbf{E} .$$

If now, $E\llbracket E\rrbracket\rho$ is not in the $[\mathbf{E} \rightarrow \mathbf{E}]$ subdomain of $\mathbf{E}$ then $E\llbracket E_0\rrbracket\rho \equiv ?$ and E_0 and E_1 are no longer equivalent. Hence if basic values are allowed as well as functions, then in η-conversion the expression being applied must be a function. This means that the η-conversion rule is not universally valid in the λ-calculus extended with basic values.

This completes the proof of Theorem 3.4.2.

3.5 Equivalence of Non-Convertible Expressions.

Now consider whether two equivalent expressions are necessarily convertible.

$$\text{i.e. } E_0 \equiv E_1 \Rightarrow E_0 \ \mathbf{cnv}\ E_1 .$$

Two cases arise.

Case 1 : E_0 and E_1 have normal forms.

3.5.1 Theorem. Suppose E_0 and E_1 are λ-expressions in normal form, not convertible, and let $I_1, I_2, \ldots, I_t$ include all the free variables in E_0 and E_1. Then there exist sequences $U_1, U_2, \ldots, U_t$ and $H_1, H_2, \ldots, H_s$, where each U_j is either I_j or some combinator, and each H_k is either a combinator or some variable not otherwise occurring, such that for

$$E_i' = [U_1/I_1][U_2/I_2]\ldots[U_t/I_t]E_i \qquad (i=0,1)$$

one has

$$E_i' H_1 H_2 \ldots H_s X_0 X_1 = X_i \ , \qquad (i=0,1)$$

where X_0 and X_1 are two variables not otherwise occurring. (For a proof see Curry et al [1972], pp.156-162.)

This theorem implies that

$$E_0' H_1 H_2 \ldots H_s X_0 X_1 \equiv \mathbf{K} \ , \text{ and}$$

$$E_1' H_1 H_2 \ldots H_s X_0 X_1 \equiv \mathbf{KI} \ .$$

Since $\mathbf{K} \not\equiv \mathbf{KI}$ in the space **E** for all ρ, by extensionality $E_0' \not\equiv E_1'$, and so also $E_0 \not\equiv E_1$.

Thus if E_0 and E_1 have normal forms and are not convertible, then

$$E_0 \not\equiv E_1 .$$

$$\text{i.e. } E_0 \ \mathbf{cnv}\ E_1 \Rightarrow E_0 \not\equiv E_1 .$$

So if E_0 and E_1 have normal forms, then

$$E_0 \ \mathbf{cnv}\ E_1 \Longleftrightarrow E_0 \equiv E_1 .$$

Case 2 : E_0 and E_1 do not have normal forms.

It is shown that in this case the above statement is not true, by giving an example of two equivalent expressions which are non-convertible. Upper case letters are used here to denote λ-expressions ($\in$**Exp**). The corresponding lower case letters are used to denote the values of those expressions in some environment ρ.

Consider the following pure combinators (whose values are independent of their environment) :

$\mathbf{G} \equiv \lambda Y.\lambda F.F(YF)$

$\mathbf{Y_1} \equiv \lambda F.\ (\lambda X.F(XX))\ (\lambda X.F(XX))$

$Y_2 \equiv Y_1G$.

3.5.2 Definition. In the λ-calculus a fixed point of a λ-expression F is any X such that

$$F(X) \text{ cnv } X.$$

3.5.3 Definition. A fixed point operator is any Z such that for all F, ZF is a fixed point of F.

$$\text{i.e. } F(ZF) \text{ cnv } ZF \qquad \text{for all } F \in \mathbf{Exp}.$$

3.5.4 Lemma. All fixed points of G are fixed point operators.

Proof : Let X be a fixed point of G, so that X cnv GX. Then for all F,

$$\begin{aligned} XF &\text{ cnv } GXF \\ &\equiv (\lambda.Y.\lambda F.F(YF))XF \qquad \text{(definition of G)} \\ &\text{cnv}_\beta\ F(XF). \end{aligned}$$

3.5.5 Lemma. Y_1 is a fixed point operator.

Proof : Clearly,

$$\begin{aligned} Y_1F &\equiv (\lambda F.\ (\lambda X.F(XX))\ (\lambda X.F(XX)))\ F \\ &\text{cnv } (\lambda X.F(XX))\ (\lambda X.F(XX)) \\ &\text{cnv } F\ ((\lambda X.F(XX))\ (\lambda X.F(XX))) \end{aligned}$$

and also,

$$\begin{aligned} F(Y_1F) &\equiv F\ ((\lambda F.\ (\lambda X.F(XX))\ (\lambda X.F(XX)))\ F) \\ &\text{cnv } F\ ((\lambda X.F(XX))\ (\lambda X.F(XX))) \end{aligned}$$

Thus since A cnv B $\iff$ B cnv A, then

$$F(Y_1F) \text{ cnv } Y_1F.$$

i.e. Y_1 is a fixed point operator.

3.5.6 Lemma. Y_2 is a fixed point operator.

Proof : Y_1 is a fixed point operator (Lemma 3.5.5), so

$$Y_2 \equiv Y_1G$$

is a fixed point of G. So Y_2 is a fixed point operator (Lemma 3.5.4).

3.5.7 Theorem. Y_1 cnv Y_2.

Proof : See Böhm [1971]. The proof proceeds by counting the number of λ's in the expressions convertible from each of Y_1 and Y_2.

Now, for any ρ,

$E[\![G]\!]\rho = \mathbf{g} = \lambda y.\lambda f.f(yf)$

$E[\![Y_1]\!]\rho = \mathbf{y_1} = \lambda f.\ (\lambda x.f(xx))\ (\lambda x.f(xx))$

$E[\![Y_2]\!]\rho = \mathbf{y_2} = \mathbf{y_1 g}$

since G, Y_1, Y_2 are pure combinators, their values being independent of their environment.

3.5.8 Lemma. If $x = \mathbf{g}x$ then $xf = f(xf)$ for all $f \in \mathbf{E}$.

Proof : $xf \equiv \mathbf{g}xf \equiv (\lambda y.\lambda f.f(yf))xf \equiv f(xf)$

(by Theorem 3.4.2 : convertibility implies equivalence)

Scott's theory provides the result that $\mathbf{y_1}$ is the <u>minimal</u> fixed point operator. That is :

(1) If $x = fx$, then $\mathbf{y_1}f \sqsubseteq x$;

(2) If y is a fixed point operator, so that $yf = f(yf)$ for all $f \in \mathbf{E}$, then $\mathbf{y_1} \sqsubseteq y$.

The notation $\sqsubseteq$ here is from Scott's theory and is read "is less-defined than" or "is weaker than".

3.5.9 Theorem. $\mathbf{y_1} \equiv \mathbf{y_2}$.

Proof : $\mathbf{g}\mathbf{y_1} \equiv (\lambda y.\lambda f.f(yf))\mathbf{y_1}$

$\equiv \lambda f.f(\mathbf{y_1}f)$

$\equiv \lambda f.\mathbf{y_1}f$ (since $\mathbf{y_1}$ is a fixed point operator - analogue of Lemma 3.5.5)

$\equiv \mathbf{y_1}$ (η-conversion)

So $\mathbf{y_1}$ is a fixed point of $\mathbf{g}$. So $\mathbf{y_1 g} \sqsubseteq \mathbf{y_1}$ by (1) above. But $\mathbf{y_1 g} = \mathbf{y_2}$, and so $\mathbf{y_2} \sqsubseteq \mathbf{y_1}$. On the other hand,

$\mathbf{g}\mathbf{y_2} \equiv \mathbf{g}(\mathbf{y_1 g})$

$\equiv \mathbf{y_1 g}$ (since $\mathbf{y_1}$ is a fixed point operator)

$\equiv \mathbf{y_2}$ (by definition),

so $\mathbf{y_2}$ is a fixed point of $\mathbf{g}$.

By Lemma 3.5.8 $\mathbf{y_2}$ is a fixed point operator. Hence $\mathbf{y_1} \sqsubseteq \mathbf{y_2}$ by (2) above.

So $\mathbf{y_1} \equiv \mathbf{y_2}$.

3.5.10 Corollary. $Y_1 \equiv Y_2$, even though Y_1 c$\not{\text{n}}$v Y_2.

In summary : convertible expressions are equivalent; non-convertible expressions are non-equivalent if both have normal forms, but not necessarily otherwise.

3.6 Discussion

In 3.4 and 3.5 the validity of the conversion rules has been proven. This means that any interpreter for the λ-calculus based on these rules will *never* give a wrong answer. However, such an interpreter is not guaranteed to produce an answer for every possible λ-expression - e.g. a normal order interpreter (terminating only when it reduces an expression to normal form) will not terminate on either :

(a) $(\lambda x.xx)(\lambda x.xx)$; or

(b) $\lambda f.\ (\lambda x.f(xx))\ (\lambda x.f(xx))$.

Expression (a) has value $\perp$, so for the interpreter to terminate given input (a) is out of the question (as it would be a solution of the Halting problem). But expression (b) actually denotes a proper value, the least fixed point function *fix*. So particular interpreters may fail to find a result even in those cases when one does exist. We could force termination for expression (b) by having the evaluator terminate whenever it reaches a normal form expression or an abstraction. However, for this modified interpreter we pay a price : since results are no longer necessarily in normal form the Church-Rosser Theorem (2.2.3) cannot be used to tell immediately whether two particular results are distinct and therefore (by 3.5.1) unequal - e.g. the representation for the non-negative integers given in 2.3.3 could no longer be used. This price would not be too high if our value space included atoms along with some appropriate primitive functions acting on them - for the atomic values the interpreter would produce some sort of canonical output (such as the numerals for the integers), for functions some equivalent expression would be output while for other values either an error stop would occur or the interpreter would fail to terminate.

The alternative interpreters so far described have not altered the semantics of the language since the λ-expressions still behave correctly when applied to other expressions. Our only problem has been in selecting an appropriate representation of the output value. A change which does affect the semantics is call-by-value. Under this scheme both the function and argument of an application are evaluated before the application itself is performed. So the expression

$(\lambda y.0)((\lambda x.xx)(\lambda x.xx))$

fails to terminate, whereas the normal order interpreter would return 0 as the result. The λ-calculus with call-by-value semantics is clearly radically different to that with call-by-name. Briefly, β-conversions must be restricted and recursively defined functions are altered since Y ceases to be a least fixed point operator, instead always returning $\perp$ (the call-by-value interpreter would not terminate).

3.6.1 Conclusion

Semantics for the λ-calculus have now been established. A class of functions has been found which the λ-calculus describes, without contradiction.

3.6.2 Summary of syntax and semantics of the λ-calculus

Syntactic Categories

$I \in \mathbf{Ide}$ (the usual identifiers)

$E \in \mathbf{Exp}$ (λ-expressions)

Syntax

$E ::= I \mid \lambda I.E \mid E_0 E_1$

Value Domains

$\epsilon \in \mathbf{E} = [\mathbf{E} \rightarrow \mathbf{E}]$ (values of expressions)

$\rho \in \mathbf{U} = [\mathbf{Ide} \rightarrow \mathbf{E}]$ (environments)

Semantic Functions

$$E : [\mathbf{Exp} \rightarrow [\mathbf{U} \rightarrow \mathbf{E}]]$$

$$E[\![I]\!]\rho = \rho[\![I]\!] \quad (1)$$

$$E[\![\lambda I.E]\!]\rho = \lambda\epsilon.E[\![E]\!](\rho[\epsilon/I]) \quad (2)$$

$$E[\![E_0 E_1]\!]\rho = (E[\![E_0]\!]\rho)(E[\![E_1]\!]\rho) \quad (3)$$

3.6.3 Semantics of the λ-calculus with atoms

Syntactic Categories

$I \in \mathbf{Ide}$ (the usual identifiers)
$B \in \mathbf{Bas}$ (bases or atoms)
$E \in \mathbf{Exp}$ (λ-expressions)

Syntax

$E ::= I \mid B \mid \lambda I.E \mid E_0 E_1$

Value Domains

$\beta \in \mathbf{B}$ (basic values)
$\epsilon \in \mathbf{E} = \mathbf{B} + [\mathbf{E} \to \mathbf{E}]$ (values of expressions)
$\rho \in \mathbf{U} = [\mathbf{Ide} \to \mathbf{E}]$ (environments)

Semantic Functions

$B : [\mathbf{Bas} \to \mathbf{E}]$
(detailed definition omitted)

$E : [\mathbf{Exp} \to [\mathbf{U} \to \mathbf{E}]]$

$E\llbracket I \rrbracket\rho = \rho\llbracket I \rrbracket$
$E\llbracket B \rrbracket\rho = B\llbracket B \rrbracket$
$E\llbracket \lambda I.E \rrbracket\rho = \lambda\epsilon.E\llbracket E \rrbracket(\rho[\epsilon/I])$ in $\mathbf{E}$
$E\llbracket E_0 E_1 \rrbracket\rho = (E\llbracket E_0 \rrbracket\rho\llbracket [\mathbf{E} \to \mathbf{E}] \rrbracket)(E\llbracket E_1 \rrbracket\rho)$

Chapter 4
A LAMBDA-CALCULUS INTERPRETER

4.1 Introduction

In this chapter an interpreter for the λ-calculus is developed. The implementation given here in FranzLISP (Foderaro et al. [1983], Wilensky [1984]) follows that of Aiello & Prini [1981]. Their implementation was developed by making successive transformations of a straightforward implementation of β-reduction. The overriding concern is that the interpreter be as efficient as possible.

The approach used by Aiello & Prini [1981] was to develop a call-by-need reduction algorithm for the λ-calculus. The two best known mechanisms of function-calling (i.e. β-reduction) are call-by-value and call-by-name, as discussed in 2.2.5 and 3.6. Both of these mechanisms have inherent undesirable inefficiencies. With call-by-value, function arguments are always evaluated before the function is applied. This means that arguments which may never be used in the function body are always evaluated. Such compulsory evaluation of all arguments may prove to be non-terminating, resulting in the entire function call being non-terminating. The other option, call-by-name, is equally undesirable since each argument is re-evaluated *every* time it occurs in the function body. The solution put forward, call-by-need, consists of evaluating arguments *by name* the first time they are needed in the evaluation of the function body and subsequently *by value*. In purely applicative languages (i.e. without side effects), this "lazy" evaluation is satisfactory to eliminate the drawbacks of both call-by-name and call-by-value.

Another problem arising in the manipulation of λ-expressions is the possible occurrence of conflicts, when substituting an expression for a variable in another expression. Free variables of the expression being substituted may be captured by a binder of the host expression. This is usually avoided in the λ-calculus by performing an α-conversion on the host, to rename the dummy variable causing the conflict. The graph reduction algorithm developed here treats bound variables as nameless dummies, avoiding α-conversions completely. In this context, actual names are not required to manipulate a term. They are only ever required when the graph representation of an expression

is written out in linear form, so as to resolve name conflicts for the human reader.

4.2 Basic Notions

The λ-calculus definitions given in Chapter 1 are assumed. The LISP representations of λ-expressions (also known as λ-terms) are as follows :

- A variable is any LISP atom other than LAMBDA.
- An application $M\ N$ is the list (M N).
- A λ-abstraction $\lambda x.M$ is a list of the form (LAMBDA x M).

New terminology is also introduced. In the term $\lambda x.M$, M is called the scope of x. Any other binding occurring in M has a narrower scope than x. If N is a subterm of M, then the scope list of an occurrence of N in M is the list of binders of M in whose scope the given occurrence of N is located. In this list, X precedes Y if X has a narrower scope than Y.

Using the above representation, expressions are valid LISP S-expressions. LISP functions can then be coded which identify, construct and select parts of terms, called data structure manipulating primitives. The names of identifying predicates and constructors begin with the prefix "IS_" and "MK_" respectively, while the names of selectors end with the suffix "_OF" :

```
(def IS_LAMBDA_TERM
  (lambda (term)
    (or (IS_VARIABLE term)
        (IS_APPLICATION term)
        (IS_LAMBDA_ABSTRACTION term))))

(def IS_VARIABLE
  (lambda (term)
    (and (atom term)
         (not (eq term 'LAMBDA)))))

(def IS_APPLICATION
  (lambda (term)
    (LENGTHP term 2)))

(def IS_LAMBDA_ABSTRACTION
  (lambda (term)
    (and (LENGTHP term 3)
         (eq (car term) 'LAMBDA)
         (IS_VARIABLE (cadr term)))))

(def MK_VARIABLE
  (lambda (name) name))
```

```
(def MK_APPLICATION
  (lambda (function argument)
    (list function argument)))

(def MK_LAMBDA_ABSTRACTION
  (lambda (binder form)
    (list 'LAMBDA binder form)))

(def NAME_OF
  (lambda (term) term))

(def FUNCTION_OF
  (lambda (term) (car term)))

(def ARGUMENT_OF
  (lambda (term) (cadr term)))

(def BINDER_OF
  (lambda (term) (cadr term)))

(def FORM_OF
  (lambda (term) (caddr term)))
```

The predicate LENGTHP is defined as follows :

```
(def LENGTHP
  (lambda (list n)
    (cond ((IS_VARIABLE list) (eq n 1))
          (t (eq (length list) n)))))
```

An interpreter using normal-order reduction, based on code to perform substitution according to substitution rule 2.2.1, is relatively simple to devise (see Aiello & Prini [1981]). However, the interpreter discussed below simulates substitution by graph reduction.

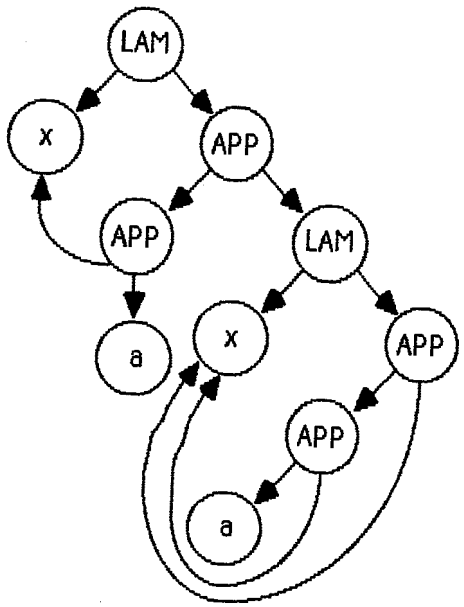
4.3 Reduction to Normal Form : Efficient Algorithms

A new data structure is adopted here to simulate substitutions by graph reduction, and in so doing to perform them more efficiently.

4.3.1 The Data Structure - a graph representation.

Terms are represented as ordered acyclic graphs. The binders of any two distinct λ-abstractions are represented as distinct structures, whereas all occurrences of a bound variable within a binder's scope are represented by the same structure as the binder itself. In FranzLISP, structures which are the same are recognised in terms of being "eq", whereas structures which are distinct are not "eq", although they may be "equal" by having equivalent LISP values. As an example consider the representation of the term

$(\lambda x.xa\ (\lambda x.axx))$:



In the graph :

- a variable is represented by its name enclosed in a circle.
- applications are represented by a circled APP (the left and right outgoing arrows point to the representation of the function and argument respectively).
- λ-abstractions are represented by a circled LAM (the left and right arrows pointing to its binder and form respectively).

In this representation bound variables are *nameless*, although a name is carried for printing purposes. *Distinct* bound variables are always accessed through *different* (i.e. not "eq") pointers, while distinct *occurrences of the same* bound variable are always accessed through *equal* (i.e. "eq") pointers. The following constructor functions are introduced :

```
(def MK_VAR
  (lambda (nam) (list 'VAR nam)))

(def MK_APP
  (lambda (fun arg) (list 'APP fun arg)))

(def MK_LAM
  (lambda (bnd frm) (list 'LAM bnd frm)))
```

```
(def NAM_OF
  (lambda (term) (cadr term)))

(def FUN_OF
  (lambda (term) (cadr term)))

(def ARG_OF
  (lambda (term) (caddr term)))

(def BND_OF
  (lambda (term) (cadr term)))

(def FRM_OF
  (lambda (term) (caddr term)))

(def IS_VAR
  (lambda (term) (eq (car term) 'VAR)))

(def IS_APP
  (lambda (term) (eq (car term) 'APP)))

(def IS_LAM
  (lambda (term) (eq (car term) 'LAM)))
```

In order to transform a λ-expression into the appropriate graph structure environments are used. An environment is a data structure used for representing a function

$$Env : Var \rightarrow Val$$

which associates values with variables. The data structure used for environments is a list of associations (or bindings). The various types of environments used are manipulated by these primitives :

```
(def IS_ARID
  (lambda (env) (null env)))

(def MK_ARID
  (lambda () nil))

(def MK_BIND
  (lambda (var val env)
    (cons (cons var val) env)))

(def VAR_OF
  (lambda (env) (caar env)))

(def VAL_OF
  (lambda (env) (cdar env)))

(def REST_OF
  (lambda (env) (cdr env)))
```

To search an environment for a given variable's associated value, we use the function LOOK_UP. This returns the environment whose first variable is the one being searched for, if the variable is present in the environment, and otherwise returns the arid environment :

```
(def LOOK_UP
  (lambda (var env)
    (cond ((or (IS_ARID env) (eq var (VAR_OF env))) env)
          (t (LOOK_UP var (REST_OF env))))))
```

A REPENV (REPresentation ENVironment) is used for associating variables of terms (i.e. structures satisfying IS_VARIABLE) with their graph representation (i.e. structures satisfying IS_VAR) - viz. a mapping :

$$Repenv : Variable \rightarrow Var.$$

REPRESENT relies on the function REP, whose arguments are the λ-term to be represented and a *Repenv* (initially arid), to represent a term in its graph form.

```
(def REPRESENT
  (lambda (term)
    (REP term (MK_ARID))))

(def REP
  (lambda (term repenv)
    (cond ((IS_VARIABLE term)
           (let ((env (LOOK_UP term repenv)))
                (cond ((IS_ARID env) (MK_VAR (NAME_OF term)))
                      (t (VAL_OF env)))))
          ((IS_APPLICATION term)
           (MK_APP (REP (FUNCTION_OF term) repenv)
                   (REP (ARGUMENT_OF term) repenv)))
          ((IS_LAMBDA_ABSTRACTION term)
           (let ((var (MK_VAR (NAME_OF (BINDER_OF term)))))
                (MK_LAM var
                        (REP (FORM_OF term)
                        (MK_BIND (BINDER_OF term)
                                 var
                                 repenv)))))
          (t (invalid)))))
```

When a variable is encountered which is not a binder of a λ-abstraction, a new structure is created for it only if it does not occur in "repenv" - i.e. if it is *free*. Also "repenv" is only modified when a λ-abstraction is encountered, in which case the form of the term is traversed with a "repenv" obtained from the previous one by adding an association between the binder of the λ-abstraction and its representation.

The following primitive functions are introduced for manipulating scope lists :

```
(def IS_VOID
  (lambda (scolis) (null scolis)))

(def MK_VOID
  (lambda () nil))
```

```
(def MK_SCO
  (lambda (var scolis) (cons var scolis)))

(def HEAD_OF
  (lambda (scolis) (car scolis)))

(def TAIL_OF
  (lambda (scolis) (cdr scolis)))
```

The functions VARS_OF and VALS_OF enable further comment on the function REPRESENT. They are given an environment as argument and defined as follows :

```
(def VARS_OF
  (lambda (env)
    (cond ((IS_ARID env) env)
          (t (MK_SCO (VAR_OF env)
                     (VARS_OF (REST_OF env)))))))

(def VALS_OF
  (lambda (env)
    (cond ((IS_ARID env) (env))
          (t (MK_SCO (VAL_OF env)
                     (VALS_OF (REST_OF env)))))))
```

Let N be a subterm of M. Suppose that (REPRESENT M) evaluates to M_1, and that during this evaluation an occurrence of N is processed by REP with the representation environment "repenv", yielding the result N_1. Then (VARS_OF repenv) is the scope list of that occurrence of N in M, and (VALS_OF repenv) is the scope list of the corresponding occurrence of N_1 in M_1. In other words, in REPRESENT the result of (VARS_OF repenv) is the scope list of "term", and (VALS_OF repenv) is the scope list of (REP term repenv).

4.3.2 A Call-by-Name Interpreter

An efficient interpreter for the λ-calculus which implements a call-by-name normal order reduction algorithm is now introduced. Reduction is simulated using an environment (denoted as "lexenv").

Whenever a redex, say $(\lambda x.M)N$, is to be reduced, the reduction is simulated by adding a binding between x and N in front of the current "lexenv" (N is put in *unreduced*). When some occurrence of x is found within M, the term N bound to x in the current "lexenv" is retrieved and processed. The environment to be used for reducing N is linked to it by constructing a data structure called a suspension. It is actually this suspension that is associated with x when extending "lexenv".

If a λ-abstraction, $\lambda x.M$ say, is to be reduced to its normal form, $\lambda x.N$ say, then a binding between x_{old} and x_{new} is put in front of the current "lexenv". The symbols x_{old} and x_{new} denote the different (not "eq") structures used for representing the binders of $\lambda x.M$ and $\lambda x.N$, respectively. Whenever a λ-abstraction is traversed, an α-reduction is automatically simulated by creating a new structure for the binder of the abstraction. When some occurrence of x_{old} is found within M then x_{new} is retrieved from the current "lexenv" and returned as the normal form of x_{old}. Thus no binder is ever shared among several λ-abstractions. A fresh version of x is introduced, because $\lambda x.M$ may be traversed several times in the same reduction process, yielding many subterms of the form $(\lambda x...)$ in the final term. If some of these subterms are nested, occurrences of x in the body of the innermost subterm are identified with the right binder only if the two binders are kept distinct.

Thus *Lexenv* is a mapping

$$Lexenv : Var \rightarrow Susp + Var$$

where *Susp* is a data structure defined by the following primitives :

```
(def IS_SUSP
  (lambda (susp) (eq (car susp) 'SUSP)))

(def MK_SUSP
  (lambda (term env) (list 'SUSP term env)))

(def TERM_OF
  (lambda (susp) (cadr susp)))

(def ENV_OF
  (lambda (susp) (caddr susp)))
```

By using "lexenv" to simulate reductions, the code to generate a normal form of an expression is as follows :

```
(def REDUCE_TO_NORMAL_FORM
  (lambda (term)
    (UNREPRESENT (RTNF (REPRESENT term) (MK_ARID)))))

(def RTNF
  (lambda (term lexenv)
    (cond ((IS_VAR term) (RTNF_VAR term lexenv))
          ((IS_APP term) (RTNF_APP term lexenv))
          ((IS_LAM term) (RTNF_LAM term lexenv))
          (t (invalid)))))

(def RTNF_VAR
  (lambda (var lexenv)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env) var)
               (t (let ((susp (VAL_OF env)))
```

```
                        (cond ((IS_SUSP susp)
                               (RTNF (TERM_OF susp)
                                     (ENV_OF susp)))
                              (t susp)))))))

(def RTNF_APP
  (lambda (app lexenv)
    (let ((susp (RTLF (FUN_OF app) lexenv)))
         (cond ((IS_SUSP susp)
                (let ((fun (TERM_OF susp)) (env (ENV_OF susp)))
                     (RTNF (FORM_OF fun)
                           (MK_BIND (BND_OF fun)
                                    (MK_SUSP (ARG_OF app)
                                    lexenv)
                           env))))
               (t (MK_APP susp
                          (RTNF (ARG_OF app) lexenv)))))))

(def RTNF_LAM
  (lambda (lam lexenv)
    (let ((newvar (MK_VAR (NAM_OF (BND_OF lam)))))
         (MK_LAM newvar
                 (RTNF (FRM_OF lam)
                       (MK_BIND (BND_OF lam)
                                newvar
                                lexenv))))))

(def RTLF
  (lambda (term lexenv)
    (cond ((IS_VAR term) (RTLF_VAR term lexenv))
          ((IS_APP term) (RTLF_APP term lexenv))
          ((IS_LAM term) (RTLF_LAM term lexenv))
          (t (invalid)))))

(def RTLF_VAR
  (lambda (var lexenv)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env) var)
               (t (let ((susp (VAL_OF env)))
                       (cond ((IS_SUSP susp)
                              (RTLF (TERM_OF susp)
                                    (ENV_OF susp)))
                             (t susp)))))))))

(def RTLF_APP
  (lambda (app lexenv)
    (let ((susp (RTLF (FUN_OF app) lexenv)))
         (cond ((IS_SUSP susp)
                (let ((fun (TERM_OF susp))(env (ENV_OF susp)))
                     (RTLF (FORM_OF fun)
                           (MK_BIND (BND_OF fun)
                                    (MK_SUSP (ARG_OF app)
                                             lexenv)
                                    env))))
               (t (MK_APP susp
                          (RTNF (ARG_OF app) lexenv)))))))

(def RTLF_LAM
```

```
(lambda (lam lexenv)
  (MK_SUSP lam lexenv)))
```

The two mutually recursive functions RTNF and RTLF operate on graph representations of λ-terms and produce a graph representation as a value. RTNF Reduces a Term to Normal Form, if such a normal form exists, and never terminates otherwise. RTLF Reduces a Term to Lambda Form (i.e. to a λ-abstraction), if such a form exists, and otherwise behaves like RTNF. The function UNREPRESENT is used to perform the inverse transformation of REPRESENT.

For a bound variable, both RTNF_VAR and RTLF_VAR respectively attempt to return the normal or lambda form of the value associated with the bound variable in the lexical environment. Free variables are treated trivially by returning the variable itself.

For λ-abstractions RTNF_LAM is an implementation of semantic equation (2) in Chapter 2, attempting to reduce the form of the abstraction (in which the left-most redex may only occur), while RTLF_LAM returns the λ-abstraction in suspension.

An attempt is made in RTNF_APP and RTLF_APP to transform applications into redexes. If this is successful, then such a redex is the left-most one and reduction may be simulated. Reducing an application to a redex may be done by reducing the function part of the application to a λ-abstraction, possibly by reducing some left-most redexes occurring in it. If the function part cannot be reduced to a λ-abstraction, then RTLF reduces it to normal form if any exists, and the search for the left-most redex to be reduced goes on within the argument part of the application.

β-reduction is similar to call-by-name function activations in languages with lexical binding (or static scoping); since in substitution as defined by the substitution rule 3.2.2, no free occurrence of a variable in the term to be substituted becomes bound by any binder of the resulting term. This is as opposed to dynamic binding (or dynamic scoping). Hence, "lexenv" is manipulated in such a way that (VARS_OF lexenv) always represents the scope list of "term".

When a non-applied λ-abstraction is encountered by RTLF, a suspension must be created consisting of the λ-abstraction itself and the current "lexenv" (also called the evaluation environment of the suspension). When such a suspended λ-abstraction is applied to some argument in some environment (called the application environment), its

body will be evaluated in an extension of the evaluation environment obtained by associating the binder of the λ-abstraction with the argument. Thus the application environment is totally ignored. Such a suspended λ-abstraction is often called a function closure.

4.3.3 A Call-by-Need Interpreter

The suspension associated with a bound variable in the "lexenv" is reduced as many times as the bound variable is processed by the interpreter. Since the term appearing in the "lexenv" is always reduced in the same environment, each reduction provides identical results. So the result of the first reduction may replace the suspension itself in the current "lexenv". This allows the call-by-name interpreter to be transformed into call-by-need. A new type of environment is introduced :

$$Lexenv : Var \rightarrow Susp + Var + App + Lam \ .$$

Here a variable may either be associated with a suspension or with a term (which is already in normal form). Modifications are made to RTNF_VAR and RTLF_VAR :

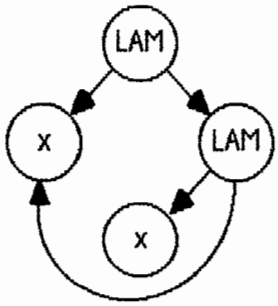
```
(def RTNF_VAR
  (lambda (var lexenv)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env) var)
               (t (let ((susp (VAL_OF env)))
                       (cond ((IS_SUSP susp)
                              (let ((val (RTNF (TERM_OF susp)
                                               (ENV_OF susp))))
                                   (progn (UPDATE_VAL_OF env
                                                         val)
                                          val)))
                             (t susp)))))))))

(def RTLF_VAR
  (lambda (var lexenv)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env) var)
               (t (let ((susp (VAL_OF env)))
                       (cond ((IS_SUSP susp)
                              (let ((val (RTLF (TERM_OF susp)
                                               (ENV_OF susp))))
                                   (progn (UPDATE_VAL_OF env
                                                         val)
                                          val)))
                             (t susp)))))))))
```

where

```
(def UPDATE_VAL_OF
  (lambda (env val) (rplacd (car env) val)))
```

while the term :



is printed as

(LAMBDA x (LAMBDA x1 x)) ,

rather than

(LAMBDA x1 (LAMBDA x x1))

or

(LAMBDA x (LAMBDA x2 x)).

The renaming algorithm used here operates "on the fly" during the reduction process, within RTNF and RTLF. Variable representation is modified as follows :

```
(def IS_VAR
  (lambda (term) (eq (car term) 'VAR)))

(def MK_VAR
  (lambda (name) (list 'VAR name 0)))

(def NAM_OF
  (lambda (var) (cadr var)))

(def REN_OF
  (lambda (var) (caddr var)))

(def NAR_OF
  (lambda (var) (cdddr var)))

(def UPDATE_REN_OF
  (lambda (var ren)
    (rplaca (cddr var) ren)))

(def UPDATE_NAR_OF
  (lambda (var nar)
    (rplacd (cddr var) nar)))
```

A variable now consists of a name as before; an integer indicating its possible RENaming that is initialised to 0; and a list of variables (initially set to "nil") having the same name as the one considered and a NARrower scope, and whose decoration is to be modified when modifying the decoration of the variable being considered. The following functions perform renaming :

```
(def DECORATE
  (lambda (var)
    (progn
      (UPDATE_REN_OF var (add1 (REN_OF var)))
      (DECNAR (NAR_OF var)))))

(def DECNAR
  (lambda (narlist)
    (cond ((null narlist) nil)
          (t (progn (DECORATE (car narlist))
                    (DECNAR (cdr narlist)))))))
```

Note that only bound variables are possibly decorated, and that a binder occurs in the final term whenever RTNF_LAM traverses a λ-abstraction in the term to be reduced. Hence the scope list of an occurrence of a bound variable in the final term is in a sense a representation of the list of activations of MK_LAM which have not yet been completed, when such a variable is returned. It is thus sufficient to provide both RTNF and RTLF with a third argument - viz. namely a scope list called "dynsco", initially set to the void scope list. Whenever a nonapplied λ-abstraction is traversed, the fresh copy of its binder is not only added in front of the current "lexenv" but is also added to the current scope list. The name "dynsco" indicates that the scope lists are processed dynamically.

The decoration process is driven by the function RENAME :

```
(def RENAME
  (lambda (var dynsco)
    (cond ((IS_VOID dynsco) var)
          (t (let ((val (HEAD_OF dynsco)))
               (cond ((eq var val) var)
                     (t (progn
                          (cond
                            ((and (eq (NAM_OF var)
                                      (NAM_OF val))
                                  (eq (REN_OF var)
                                      (REN_OF val)))
                             (progn (DECORATE val)
                                    (UPDATE_NAR_OF
                                      var
                                      (cons val
                                            (NAR_OF var))))
                            (t nil))
                          (RENAME var
                                  (TAIL_OF dynsco)))))))))))
```

REDUCE_TO_NORMAL_FORM becomes :

```
(def REDUCE_TO_NORMAL_FORM
  (lambda (term)
    (UNREPRESENT (RTNF (REPRESENT term)
                       (MK_ARID)
                       (MK_VOID)))))
```

```
(def RTNF
  (lambda (term lexenv dynsco)
    (cond ((IS_VAR term) (RTNF_VAR term lexenv dynsco))
          ((IS_APP term) (RTNF_APP term lexenv dynsco))
          ((IS_LAM term) (RTNF_LAM term lexenv dynsco))
          (t (invalid)))))

(def RTNF_APP
  (lambda (app lexenv dynsco)
    (let ((susp (RTLF (FUN_OF app)
                      lexenv
                      dynsco)))
         (cond ((IS_SUSP susp)
                (let ((fun (TERM_OF susp)) (env (ENV_OF susp)))
                     (RTNF (FORM_OF fun)
                           (MK_BIND (BND_OF fun)
                                    (MK_SUSP (ARG_OF app)
                                             lexenv)
                                    env)
                           dynsco)))
               (t (MK_APP susp
                          (RTNF (ARG_OF app)
                                lexenv
                                dynsco)))))))

(def RTNF_LAM
  (lambda (lam lexenv dynsco)
    (let ((newvar (MK_VAR (NAM_OF (BND_OF lam)))))
         (MK_LAM newvar
                 (RTNF (FRM_OF lam)
                       (MK_BIND (BND_OF lam)
                                newvar
                                lexenv)
                       (MK_SCO newvar dynsco))))))

(def RTLF
  (lambda (term lexenv dynsco)
    (cond ((IS_VAR term) (RTLF_VAR term lexenv dynsco))
          ((IS_APP term) (RTLF_APP term lexenv dynsco))
          ((IS_LAM term) (RTLF_LAM term lexenv dynsco))
          (t (invalid)))))

(def RTLF_APP
  (lambda (app lexenv dynsco)
    (let ((susp (RTLF (FUN_OF app) lexenv dynsco)))
         (cond ((IS_SUSP susp)
                (let ((fun (TERM_OF susp)) (env (ENV_OF susp)))
                     (RTLF (FORM_OF fun)
                           (MK_BIND (BND_OF fun)
                                    (MK_SUSP (ARG_OF app)
                                             lexenv)
                                    env)
                           dynsco)))
               (t (MK_APP susp (RTNF (ARG_OF app)
                                     lexenv
                                     dynsco)))))))
```

```
(def RTLF_LAM
  (lambda (lam lexenv dynsco)
    (MK_SUSP lam lexenv)))
```

A further problem arising in the call-by-need interpreter is that two different occurrences of the same subterm in the final term may cause different name conflicts, hence different decorations are needed. It is not just enough to modify RTNF_VAR and RTLF_VAR as has been done for the rest of the functions in RTNF and RTLF. When a term is returned by RTNF_VAR or RTLF_VAR, it may contain some variables that perhaps cause a name conflict in the current "dynsco", while they caused no conflicts when it was reduced. A function is introduced that recursively examines the term, looking for possible renamings :

```
(def PROPAGATE_RENAMING
  (lambda (term dynsco)
    (cond ((IS_VAR term) (RENAME term dynsco))
          ((IS_APP term)
           (progn (PROPAGATE_RENAMING (FUN_OF term) dynsco)
                  (PROPAGATE_RENAMING (ARG_OF term) dynsco)
                  term))
          ((IS_LAM term)
           (progn (PROPAGATE_RENAMING (FRM_OF term) dynsco)
                  term))
          (t (invalid)))))
```

RTNF_VAR and RTLF_VAR are modified as follows :

```
(def RTNF_VAR
  (lambda (var lexenv dynsco)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env)
                (PROPAGATE_RENAMING var dynsco))
               (t (let ((susp (VAL_OF env)))
                       (cond ((IS_SUSP susp)
                              (let ((val (RTNF (TERM_OF susp)
                                               (ENV_OF susp)
                                               dynsco)))
                                   (progn
                                     (UPDATE_VAL_OF env val)
                                     val)))
                             (t (PROPAGATE_RENAMING
                                  susp
                                  dynsco)))))))))

(def RTLF_VAR
  (lambda (var lexenv dynsco)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env)
                (PROPAGATE_RENAMING var dynsco))
               (t (let ((susp (VAL_OF env)))
                       (cond ((IS_SUSP susp)
                              (let ((val (RTLF (TERM_OF susp)
                                               (ENV_OF susp)
                                               dynsco)))
```

```
                                        (progn
                                          (UPDATE_VAL_OF env val)
                                          val)))
                                    ((IS_LAM susp)
                                     (MK_SUSP susp (MK_ARID)))
                                    (t (PROPAGATE_RENAMING
                                         susp
                                         dynsco)))))))))
```

The code for UNREPRESENT is now given :

```
(def UNREPRESENT
  (lambda (term)
    (cond ((IS_VAR term)
           (cond ((eq (REN_OF term) 0)
                  (MK_VARIABLE (NAM_OF term)))
                 (t (MK_VARIABLE (concat (NAM_OF term)
                                         (REN_OF term))))))
          ((IS_APP term)
           (MK_APPLICATION (UNREPRESENT (FUN_OF term))
                           (UNREPRESENT (ARG_OF term))))
          ((IS_LAM term)
           (MK_LAMBDA_ABSTRACTION (UNREPRESENT (BND_OF term))
                                  (UNREPRESENT (FRM_OF term))))
          (t (invalid)))))
```

4.5 Further Considerations : η-reduction

The interpreter as presented by Aiello and Prini [1981] does not consider any implementation of η-reduction. For any "real" λ-calculus implementation, basic values (as discussed in 3.3.5) would most likely be permitted, and so η-reduction would not be valid and would not be implemented. However for completeness, modification of the existing interpreter to incorporate η-reduction is made here.

A simple way to implement η-reduction would be to add a function which operates on the β-reduced result returned by REDUCE_TO_NORMAL-_FORM. This would take place, not on the graph representation of the final term, but on the linear version returned by RTNF. Such a function might be as follows :

```
(def ETA_REDUCE
  (lambda (term)
    (cond ((IS_VARIABLE term) term)
          ((IS_APPLICATION term)
           (MK_APPLICATION (ETA_REDUCE (FUNCTION_OF term))
                           (ETA_REDUCE (ARGUMENT_OF term))))
          ((IS_LAMBDA_ABSTRACTION term)
           (let ((bind (ETA_REDUCE (BINDER_OF term)))
                 (form (ETA_REDUCE (FORM_OF term)))
                (cond ((and (IS_APPLICATION form)
                            (eq bind (ARGUMENT_OF form))
                            (not (IS_FREE_IN
```

```
                                        bind
                                        (FUNCTION_OF form))))
                                (FUNCTION_OF form))
                               (t (MK_LAMBDA_ABSTRACTION bind form)))))
                (t (invalid)))))
```

IS_APPLICATION would need to be modified to cope with testing non-list arguments so that it becomes :

```
(def IS_APPLICATION
  (lambda (term)
    (and (listp term) (LENGTHP term 2))))
```

REDUCE_TO_NORMAL_FORM would be altered accordingly, so that it performs all possible η-reductions on the β-reduced term before returning it.

However, there is an inefficiency in this which may be eliminated. The function ETA_REDUCE performs a recursive traversal of the β-reduced term, in seeking for possible η-reductions to be made. This extra recursive traversal may be merged into the traversal performed by UNREPRESENT, by embedding the code for ETA_REDUCE in UNREPRESENT. The η-reductions are then performed as the term is converted from its graph form to its linear form for printing. This method requires no alteration of REDUCE_TO_NORMAL_FORM. UNREPRESENT is modified as follows :

```
(def UNREPRESENT
  (lambda (term)
    (cond ((IS_VAR term)
           (cond ((eq (REN_OF term) 0)
                  (MK_VARIABLE (NAM_OF term)))
                 (t (MK_VARIABLE (concat (NAM_OF term)
                                         (REN_OF term))))))
          ((IS_APP term) (MK_APPLICATION
                           (UNREPRESENT (FUN_OF term))
                           (UNREPRESENT (ARG_OF term))))
          ((IS_LAM term)
           (let ((bnd (UNREPRESENT (BND_OF term)))
                 (frm (UNREPRESENT (FRM_OF term))))
                (cond ((and (IS_APPLICATION frm)
                            (eq bnd (ARGUMENT_OF frm))
                            (not (IS_FREE_IN
                                   bnd
                                   (FUNCTION_OF frm))))
                       (FUNCTION_OF frm))
                      (t (MK_LAMBDA_ABSTRACTION bnd frm)))))
          (t (invalid)))))
```

For reference to other algorithms based on call-by-need see Aiello and Prini [1981]. A source code listing of the interpreter as implemented with tracing routines is given in Appendix A.

4.6 A Sample Call-by-need Reduction

The output of the tracing routines is reproduced here for reduction of the sample term used by Aiello and Prini [1981]. The current term being reduced is indicated by TRM, the lexical environment by LEX and the dynamic scope list by DYN. Notice that, whereas the result produced without η-reduction would have been

(LAMBDA z (LAMBDA z1 (z z1)))

the result returned after η-reductions is

(LAMBDA z z) :

```
TERM IS : ((LAMBDA x x x) (LAMBDA y LAMBDA z y z))
RTNF CALLED: TRM = ((LAMBDA x (x x)) (LAMBDA y (LAMBDA z (y z))))
             LEX = ARID
             DYN = VOID

-RTLF CALLED: TRM = (LAMBDA x (x x))
              LEX = ARID
              DYN = VOID

-RTLF EXITED: TRM = (LAMBDA x (x x))
              LEX = ARID
              DYN = VOID

-RTNF CALLED: TRM = (x x)
              LEX = (((VAR x 0) SUSP (LAM (VAR y 0)
                                          (LAM (VAR z 0)
                                               (APP (VAR y 0)
                                                    (VAR z 0))))
                      nil))
              DYN = VOID

--RTLF CALLED: TRM = x
               LEX = (((VAR x 0) SUSP (LAM (VAR y 0)
                                           (LAM (VAR z 0)
                                                (APP (VAR y 0)
                                                     (VAR z 0))))
                       nil))
               DYN = VOID

---RTLF CALLED: TRM = (LAMBDA y (LAMBDA z (y z)))
                LEX = ARID
                DYN = VOID

---RTLF EXITED: TRM = (LAMBDA y (LAMBDA z (y z)))
                LEX = ARID
                DYN = VOID

--RTLF EXITED: TRM = (LAMBDA y (LAMBDA z (y z)))
               LEX = (((VAR x 0) SUSP (LAM (VAR y 0)
                                           (LAM (VAR z 0)
                                                (APP (VAR y 0)
                                                     (VAR z 0))))
```

```
                    nil))
              DYN = VOID

--RTNF CALLED: TRM = (LAMBDA z (y z))
               LEX = (((VAR y 0) SUSP (VAR x 0)
                       (((VAR x 0) SUSP (LAM (VAR y 0)
                                             (LAM (VAR z 0)
                                                  (APP (VAR y 0)
                                                       (VAR z 0))))
                         nil))))
               DYN = VOID

---RTNF CALLED: TRM = (y z)
                LEX = (((VAR z 0) VAR z 0)
                       ((VAR y 0) SUSP (VAR x 0)
                        (((VAR x 0) SUSP (LAM (VAR y 0)
                                              (LAM (VAR z 0)
                                                   (APP (VAR y 0)
                                                        (VAR z 0))))
                          nil))))
                DYN = ((VAR z 0))

----RTLF CALLED: TRM = y
                 LEX = (((VAR z 0) VAR z 0)
                        ((VAR y 0) SUSP (VAR x 0)
                         (((VAR x 0) SUSP (LAM (VAR y 0)
                                               (LAM (VAR z 0)
                                                    (APP (VAR y 0)
                                                         (VAR z 0))))
                           nil))))
                 DYN = ((VAR z 0))

-----RTLF CALLED: TRM = x
                  LEX = (((VAR x 0) SUSP (LAM (VAR y 0)
                                              (LAM (VAR z 0)
                                                   (APP (VAR y 0)
                                                        (VAR z 0))))
                          nil))
                  DYN = ((VAR z 0))

------RTLF CALLED: TRM = (LAMBDA y (LAMBDA z (y z)))
                   LEX = ARID
                   DYN = ((VAR z 0))

------RTLF EXITED: TRM = (LAMBDA y (LAMBDA z (y z)))
                   LEX = ARID
                   DYN = ((VAR z 0))

-----RTLF EXITED: TRM = (LAMBDA y (LAMBDA z (y z)))
                  LEX = (((VAR x 0) SUSP (LAM (VAR y 0)
                                              (LAM (VAR z 0)
                                                   (APP (VAR y 0)
                                                        (VAR z 0))))
                          nil))
                  DYN = ((VAR z 0))

----RTLF EXITED: TRM = (LAMBDA y (LAMBDA z (y z)))
                 LEX = (((VAR z 0) VAR z 0)
```

```
                        ((VAR y 0) SUSP (LAM (VAR y 0)
                                             (LAM (VAR z 0)
                                                  (APP (VAR y 0)
                                                       (VAR z 0))))
                         nil))
                DYN = ((VAR z 0))

----RTNF CALLED: TRM = (LAMBDA z (y z))
                 LEX = (((VAR y 0) SUSP (VAR z 0)
                         (((VAR z 0) VAR z 0)
                          ((VAR y 0) SUSP (LAM (VAR y 0)
                                               (LAM (VAR z 0)
                                                    (APP (VAR y 0)
                                                         (VAR z 0))))
                           nil))))
                 DYN = ((VAR z 0))

-----RTNF CALLED: TRM = (y z)
                  LEX = (((VAR z 0) VAR z 0)
                         ((VAR y 0) SUSP (VAR z 0)
                          (((VAR z 0) VAR z 0)
                           ((VAR y 0) SUSP (LAM (VAR y 0)
                                                (LAM (VAR z 0)
                                                     (APP (VAR y 0)
                                                          (VAR z 0))))
                            nil))))
                  DYN = ((VAR z 0) (VAR z 0))

------RTLF CALLED: TRM = y
                   LEX = (((VAR z 0) VAR z 0)
                          ((VAR y 0) SUSP (VAR z 0)
                           (((VAR z 0) VAR z 0)
                            ((VAR y 0)
                             SUSP (LAM (VAR y 0)
                                       (LAM (VAR z 0)
                                            (APP (VAR y 0)
                                                 (VAR z 0))))
                             nil))))
                   DYN = ((VAR z 0) (VAR z 0))

-------RTLF CALLED: TRM = z
                    LEX = (((VAR z 0) VAR z 0)
                           ((VAR y 0) SUSP (LAM (VAR y 0)
                                                (LAM (VAR z 0)
                                                     (APP (VAR y 0)
                                                          (VAR z 0))))
                            nil))
                    DYN = ((VAR z 0) (VAR z 0))

-------RTLF EXITED: TRM = z
                    LEX = (((VAR z 0) VAR z 0 (VAR z 1))
                           ((VAR y 0) SUSP (LAM (VAR y 0)
                                                (LAM (VAR z 0)
                                                     (APP (VAR y 0)
                                                          (VAR z 0))))
                            nil))
                    DYN = ((VAR z 1) (VAR z 0 (VAR z 1)))
```

```
------RTLF EXITED: TRM = z
                   LEX = (((VAR z 0) VAR z 1)
                          ((VAR y 0) VAR z 0 (VAR z 1)))
                   DYN = ((VAR z 1) (VAR z 0 (VAR z 1)))

------RTNF CALLED: TRM = z
                   LEX = (((VAR z 0) VAR z 1)
                          ((VAR y 0) VAR z 0 (VAR z 1)))
                   DYN = ((VAR z 1) (VAR z 0 (VAR z 1)))

------RTNF EXITED: TRM = z1
                   LEX = (((VAR z 0) VAR z 1)
                          ((VAR y 0) VAR z 0 (VAR z 1)))
                   DYN = ((VAR z 1) (VAR z 0 (VAR z 1)))

-----RTNF EXITED: TRM = (z z1)
                  LEX = (((VAR z 0) VAR z 1)
                         ((VAR y 0) VAR z 0 (VAR z 1)))
                  DYN = ((VAR z 1) (VAR z 0 (VAR z 1)))

----RTNF EXITED: TRM = (LAMBDA z1 (z z1))
                 LEX = (((VAR y 0) VAR z 0 (VAR z 1)))
                 DYN = ((VAR z 0 (VAR z 1)))

---RTNF EXITED: TRM = (LAMBDA z1 (z z1))
                LEX = (((VAR z 0) VAR z 0 (VAR z 1))
                       ((VAR y 0) SUSP (LAM (VAR y 0)
                                            (LAM (VAR z 0)
                                                 (APP (VAR y 0)
                                                      (VAR z 0))))
                        nil))
                DYN = ((VAR z 0 (VAR z 1)))

--RTNF EXITED: TRM = (LAMBDA z (LAMBDA z1 (z z1)))
               LEX = (((VAR y 0) SUSP (LAM (VAR y 0)
                                           (LAM (VAR z 0)
                                                (APP (VAR y 0)
                                                     (VAR z 0))))
                       nil))
               DYN = VOID

-RTNF EXITED: TRM = (LAMBDA z (LAMBDA z1 (z z1)))
              LEX = (((VAR x 0) SUSP (LAM (VAR y 0)
                                          (LAM (VAR z 0)
                                               (APP (VAR y 0)
                                                    (VAR z 0))))
                      nil))
              DYN = VOID

RTNF EXITED: TRM = (LAMBDA z (LAMBDA z1 (z z1)))
             LEX = ARID
             DYN = VOID

NORMAL FORM IS : (LAMBDA z z)
```

Chapter 5
THE SEMANTIC ALGEBRA APPROACH TO DENOTATIONAL SEMANTICS

5.1 Introduction

Denotational semantic descriptions, which represent programming language features in terms of the basic operations of λ-notation, are often criticised as being error-prone, tedious to write and difficult to comprehend. Instead of expressing the language semantics in terms of the fundamental concepts in which programming languages are understood, used and designed, the description is ultimately encoded in terms of function abstraction and application - e.g. application is used to represent the fundamental concepts of binding, finding, storing, retrieving, sequencing and argument passing (see Stoy [1977] for denotational descriptions of these concepts).

In addition, conventional semantic descriptions are not modular in that they can be difficult to modify to incorporate new language features. Any standard denotational definition may be extended so long as the extension is based on the original concepts. However, adding a new feature to a language can require rewriting all the original semantic equations (see Stoy [1977]) - e.g. adding non-determinism requires the replacement of domains by "power-domains". Modularity of the semantic definitions would allow such changes to be made without major modifications.

These problems have been pointed out by Mosses [1980,1981,1984] who has suggested an elegant alternative approach that has as its origin the standard techniques used in conventional denotational descriptions. The standard denotational semantics of a programming language are often given in terms of auxiliary operations defined on the semantic domains, as well as the operations of function abstraction and application. Auxiliary operations allow a more concise and readable semantic description, without distracting recourse to repetitive and often cumbersome details. If these operations relating to fundamental concepts of computation are chosen carefully then the clarity of the resulting denotational description can be greatly improved.

The new approach of Mosses is to abstract the *fundamental operational concepts of computation* as explicit operators, just as auxiliary operations are used in the standard approach. The values operated upon (and yielded) by these operators are exclusively *denotations* (i.e. denotable values) of syntactic constructs of the programming language being described, rather than subsidiary objects like states, environments, etc.. The denotations together with the operators constitute the semantic algebra. The semantic equations here give the meaning of each syntactic construct in terms of the fundamental concepts of computation. They are devoid of any assumptions about the structure of the semantic domains, thereby improving the modularity and hence the modifiability of the denotational description.

We now model every construct of the programming language as an action (or "process"), which gives rise to a computation when executed. We have a number of primitive actions as well as action combinators which are the operators corresponding to our fundamental concepts of computation. These primitive actions and action combinators constitute the semantic algebra. Compound or composite actions are made up of simpler actions joined together by action combinators. Examples of primitive actions include binding and finding values in an environment, storing and retrieving values in a store and applying operators. Examples of action combinators include composition, selection, iteration and abstraction. Actions may consume and/or produce sequences of values and have "effects" - e.g. alter the store or the environment.

The semantic algebra itself (i.e. its primitive actions and action combinators) can be defined axiomatically, in which case it constitutes an abstract semantic algebra (see Mosses [1984]), with the attendant advantages that non-determinism and concurrency may be accomodated. Disadvantages of an axiomatic definition of the semantic algebra are in ensuring the consistency and the completeness of the axioms. Alternatively, a concrete semantic algebra may be defined with the primitive actions and action combinators being specified as functions on Scott-domains - i.e. by giving a conventional denotational semantic definition of the semantic algebra, defining the primitive actions and action combinators in λ-notation. We follow Mosses [1981] and Watt [1986] in adopting this second approach.

5.2 A Basic Semantic Algebra

It is clear that the choice of semantic algebra is crucial to providing adequate facilities for the description of various styles of programming language. It seems reasonable to look for a small set of primitive actions and action combinators that have simple properties and allow the easy expression of the fundamental concepts of programming languages. A semantic algebra loosely based on that of Mosses [1981] was outlined by Watt [1986] which he considered adequate to describe simple programming languages. The primitive actions and action combinators of his semantic algebra are developed here.

We define a semantic domain of actions, A, general enough to model most programming language constructs. Each action exhibits one or more facets of operation :

- a functional facet : it consumes some input values and produces some output values.
- an imperative facet : it uses and/or modifies the store.
- a declarative facet : it uses and/or modifies the environment.

A primitive action might have only one facet, but compound actions may display all facets.

We now give an informal description of a particular semantic algebra that is adequate to model simple programming languages.

Primitive functional actions

take L	- produces a single output value, the value of the literal L (no input values).
applyop O	- applies the operator O to the action's input value(s), to yield the action's output value(s).
takefirst	- consumes one or more input values, and outputs only the first of these.
takerest	- consumes one or more input values, and outputs all but the first of these.
copy	- copies all its input values.
discard	- discards all its input values.

skip - does nothing (no input values or output values).

Primitive imperative actions

store - has two input values, a storable value and a location; it places the storable value in the corresponding cell of the store (no output values).

fetch - has one input value, a location; it outputs the value found in the corresponding cell of the store.

allocate - has one input value, a storable value; it places this value in a previously unused cell in the store, and outputs the location of that cell.

Primitive declarative actions

bind I - has one input value, a denotable value; it binds that value to the identifier I in the environment (no output values).

find I - outputs the value bound to identifier I in the environment (no input value).

Action combinators

Each of the following is a compound action, formed by the corresponding action combinator. A_0, A_1, etc. denote actions; IN denotes the list of input values consumed by the compound action, and OUT the list of output values produced by it.

A_1 ! A_2 - performs A_1 followed by A_2, with input IN supplied to A1, and with any output from A_1 becoming the input to A_2; OUT becomes the output from A_2.

A_1 & A_2 - performs A_1 and A_2 collaterally, the input to both being IN; OUT consists of the output from A_1 followed by the output from A_2.

select$A_1A_2A_3$ - performs A_1, with input IN, which must output a single Boolean value; then it performs either A_2

or A_3, with input IN, depending on whether the Boolean value was true or false; OUT becomes the output from whichever action was selected.

iterateA_1A_2 - is equivalent to:
select A_1 (A_2 ! (*iterate*A_1A_2)) *copy*

block A_1 - performs A_1 without any net change to the environment.

abstract A_0 - outputs a closure formed by "freezing" A_0 in the current environment (no input values).

apply - expects as input values a closure followed by any number of argument values; it performs the action of the closure with the arguments as input values. (Note that since the action of the closure was "frozen" in the environment in which it was created, the apply action can have no net effect on the environment.)

5.3 Formal Definition of the Basic Semantic Algebra

Just as a programming language needs to be formally defined so does the basic semantic algebra. In this section we give a concrete denotational description of the basic semantic algebra, in keeping with the scope of this thesis. As indicated in 5.1 the semantic algebra can also be defined axiomatically (see Mosses [1984]).

5.3.1 Operations on Lists. We define three operators, $\downarrow$, $\dagger$ and §, for handling elements of product domains - i.e. lists. These respectively extract components from lists, chop off elements from the beginning of lists and concatenate lists. These operators can obviously be used to influence members of domains of the form D^* as well as members of product domains. Let

$$\langle\rangle\downarrow m \equiv \top ,$$

$$\langle\omega_0,\ldots,\omega_n\rangle\downarrow m \equiv (n+1\geq m\geq 1) \rightarrow \omega_{m-1}, \top ;$$

$$\langle\rangle\dagger m \equiv \langle\rangle ,$$

$$\langle\omega_0,\ldots,\omega_n\rangle\dagger m \equiv (n\geq m) \rightarrow \langle\omega_{m\vee 0},\ldots,\omega_n\rangle, \langle\rangle ;$$

$$\langle\rangle\S\langle\omega_0,\ldots,\omega_n\rangle \equiv \langle\omega_0,\ldots,\omega_n\rangle ,$$

$$\langle\omega_0,\ldots,\omega_n\rangle\S\langle\rangle \equiv \langle\omega_0,\ldots,\omega_n\rangle ,$$

$$\langle\omega_0,\ldots,\omega_n\rangle\S\langle\omega_{n+1},\ldots,\omega_{m+n+1}\rangle \equiv \langle\omega_0,\ldots,\omega_{m+n+1}\rangle .$$

The notation $m \nu 0$ simply means the maximum of m and 0. For brevity the effects of these operators on the improper lists (i.e. $\top$, $\bot$ and ?) are not given here (see Milne & Strachey [1976] for detail).

5.3.2 Semantic Domains. Semantic domains common to most programming languages are :

D	(values)
U	(environments)
S	(stores)

In many programming languages three distinct but overlapping *value* domains are usually identified : denotable values (values that can be bound directly to identifiers), expressible values (values that can be yielded by executing expressions of the language) and storable values (that can be stored in locations). In most cases these are defined as separate semantic domains, along with appropriate coercions between them. For simplicity all values here will be included in a single domain D of denotable values. The value domains used are dependent on the programming language being defined. Here we give definitions of some commonly needed value domains :

$\tau \in T$	(truth values)
$\nu \in N$	(integers)
$\ell \in L$	(locations)
$\phi \in F = D^*$	(files)
$\kappa \in C = U \times A$	(closures)

so that the domain D of values is

$\delta \in D = T + N + L + F + C + A$	(values)

The first two value domains, T and N, need no explanation. A location is the (denotable) value bound to a variable identifier and may be thought of as the address of a cell in the store. A file value consists of a sequence of values which are its elements. A closure is the (denotable) value bound to a procedure identifier consisting of the body of the procedure "frozen" in the environment current at the point of declaration (as opposed to that at the point of call). Contrary to Watt [1986] we include actions in the denotable values so as to allow processes themselves to be bound to an identifier.

Each value in the domain of environments U is a set of bindings of identifiers to (denotable) values - i.e. a mapping from Ide to D. Each declaration produces a new environment by adding a new binding

(perhaps shielding a previous binding of the same identifier). A binding is fixed over the scope of an identifier. So we have

$$U = Ide \rightarrow D \qquad \text{(environments)}$$

Each value in S is a set of cells, each cell containing a (storable) value - i.e. S is a mapping from locations L to D. The contents of any given cell may be updated at any time. We define

$$S = L \rightarrow [D \times T] \qquad \text{(stores)}$$

Intuitively this gives for every location ℓ a value in D (its content) and a "tag" value in T indicating whether ℓ is "in use". This allows modelling of finite stores. Other models of the store are possible - e.g. see Tennent [1976], Stoy [1977], Milne & Strachey [1976] and Watt [1986]. Unlike the value domain D, the semantic domains U and S have structures largely independent of the programming language being described.

We can define the semantic domain of actions required to model the basic semantic algebra given in 5.2 as a mapping from one "state" to another. These "states" are elements of the product domain comprising stores, environments and lists of values (the input/output values of actions) - i.e. :

$$\alpha \in A = [U \times S \times D^*] \rightarrow [U \times S \times D^*]$$

We take a typical element of the product domain to be

$$\psi = \langle \rho, \sigma, \delta^* \rangle \in U \times S \times D^*.$$

We also assume a domain of operators on lists of values

$$o \in O = D^* \rightarrow D^* .$$

These are defined arbitrarily depending on the specific programming language being described. Each operator maps a list of denotable values into another list of denotable values. Some examples are defined :

- *equal* expects a pair of denotable values as argument and returns a truth value :

$$equal \equiv \lambda\delta^*.(\delta^* = \langle \delta_1, \delta_2 \rangle) \rightarrow \langle (\delta_1 = \delta_2) \rangle, ?$$

- Operators expecting a pair of integers as argument and returning either a truth value or an integer are :

$$less \equiv \lambda\delta^*.(\delta^* = \langle \nu_1, \nu_2 \rangle) \rightarrow \langle (\nu_1 < \nu_2) \rangle, ?$$

$$less\text{-}equal \equiv \lambda\delta^*.(\delta^* = \langle \nu_1, \nu_2 \rangle) \rightarrow \langle (\nu_1 \leq \nu_2) \rangle, ?$$

$$sum \equiv \lambda\delta^*.(\delta^* = \langle \nu_1, \nu_2 \rangle) \rightarrow \langle (\nu_1 + \nu_2) \rangle, ?$$

$$product \equiv \lambda\delta^*.(\delta^* = \langle \nu_1, \nu_2 \rangle) \rightarrow \langle (\nu_1 \times \nu_2) \rangle, ?$$

- The operators *successor* and *predecessor* expect an integer as argument and return the corresponding integer value :

  $successor \equiv \lambda\delta^*.(\delta^*=\langle\nu\rangle) \rightarrow \langle(\nu + 1)\rangle, ?$

  $predecessor \equiv \lambda\delta^*.(\delta^*=\langle\nu\rangle) \rightarrow \langle(\nu - 1)\rangle, ?$

- *conjunction* expects a pair of truth values and returns their conjunction :

  $conjunction \equiv \lambda\delta^*.(\delta^*=\langle\tau_1,\tau_2\rangle) \rightarrow \langle(\tau_1 \wedge \tau_2)\rangle, ?$

- *complement* returns the complement of its expected truth value argument :

  $complement \equiv \lambda\delta^*.(\delta^*=\langle\tau\rangle) \rightarrow \langle(\neg\tau)\rangle, ?$

- Each of *end-of-file* and *read* expect a file as argument. *end-of-file* returns a truth value while *read* returns a pair consisting of the first element of the file and the file obtained by destructively removing the first element :

  $end\text{-}of\text{-}file \equiv \lambda\delta^*.(\delta^*=\langle\phi\rangle) \rightarrow \langle(\phi = \langle\rangle)\rangle, ?$

  $read \equiv \lambda\delta^*.(\delta^*=\langle\phi\rangle) \rightarrow \langle\phi\downarrow 1,\phi\dagger 1\rangle, ?$

- *write* expects a pair consisting of a file and a denotable value and returns the file extended by that value :

  $write \equiv \lambda\delta^*.(\delta^*=\langle\phi,\delta\rangle) \rightarrow \langle(\phi\S\langle\delta\rangle)\rangle, ?$

- *rewrite* expects an empty argument list and returns an empty file :

  $rewrite \equiv \lambda\delta^*.(\delta^*=\langle\rangle) \rightarrow \langle\langle\rangle\rangle, ?$

5.3.3 Basic Operations on Stores and Environments. There is one constant store of interest, $empty \in S$, defined by :

$$empty \equiv \lambda\ell.\langle false, false\rangle$$

The content of every location in *empty* has been arbitrarily chosen to be *false*. To examine locations in the store we use $hold \in L \rightarrow S \rightarrow D$ and $area \in L \rightarrow S \rightarrow T$ defined

$hold \equiv \lambda\ell.\lambda\sigma.(\sigma\ell)\downarrow 1$ and

$area \equiv \lambda\ell.\lambda\sigma.(\sigma\ell)\downarrow 2$,

which respectively give the content and accessibility of every location in the store. The basic operation of updating a location is performed by $update \in L \rightarrow D \rightarrow S \rightarrow S$ defined :

$$update \equiv \lambda\ell.\lambda\delta.\lambda\sigma.(\lambda\ell'.(\ell' = \ell) \rightarrow \langle\delta, true\rangle, \sigma\ell') .$$

Thus if $\sigma_1 = update\ell\delta\sigma_0$ then σ_1 coincides with σ_0 everywhere except for the pair in $V \times T$ associated with ℓ. The *update* operation not only changes a location's content but also marks it as being in use. Often we need to introduce locations which have not yet been used - i.e. for

which $area\ell\sigma = false$. If the store is full and hence no such location is available then the error element $?_L$ is obtained. Rather than giving a full definition of the function $new \in S \to L$ which achieves this we provide axioms influencing its behaviour - i.e. *new* must be a continuous function such that :

(i) if σ is a store such that $\exists$ a "proper" location ℓ for which $area\ell\sigma = false$ then $new\sigma = \ell$;

(ii) if σ is a store such that $\forall$ proper locations ℓ $area\ell\sigma = true$ then $new\sigma = ?_L$.

The operation $new\sigma$ only returns a location - it does not mark the location as being in use.

Basic operations on environments must also be identified. Analogous to the store *empty* we have $arid \in U$:

$$arid \equiv \lambda I.?_D \ .$$

This returns the error element for all identifiers $I \in Ide$. For environment $\rho \in U$ and identifier $I \in Ide$ we write $\rho[I]$ $(\in D)$ for the denoted value associated with I by ρ. As in definition 3.3.4 a new environment $\rho[\delta/I] \in U$ may be produced from ρ to include a new binding of I to δ. Define $\rho[\delta/I]$ by

$$\rho[\delta/I] \equiv \lambda I'.(I'=I) \to \delta, \rho[I'] \ .$$

5.3.4 Definition of the Semantic Algebra. We now proceed to give the formal definition of the basic semantic algebra. The primitive and compound actions are defined as follows (cf. the informal specifications in 5.2).

- $take\ \delta \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\rangle) \to \langle\rho,\sigma,\langle\delta\rangle\rangle, ?$
- $applyop\ o \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\langle\rho,\sigma,o\delta^*\rangle$
- $takefirst \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\langle\rho,\sigma,\langle\delta^*\downarrow 1\rangle\rangle$
- $takerest \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\langle\rho,\sigma,\delta^*\dagger 1\rangle$
- $copy \equiv \lambda\psi.\psi$
- $discard \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\langle\rho,\sigma,\langle\rangle\rangle$
- $skip \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\rangle) \to \langle\rho,\sigma,\delta^*\rangle, ?$
- $store \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\delta,\ell\rangle) \to \langle\rho,update\ell\delta\sigma,\langle\rangle\rangle, ?$
- $fetch \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\ell\rangle) \to \langle\rho,\sigma,\langle hold\ \ell\ \sigma\rangle\rangle, ?$
- $allocate \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\delta\rangle) \to \langle\rho,\ update\ \ell\ \delta\ \sigma,\ \langle\ell\rangle\rangle, ?$

  where $\ell = new\sigma$.

- $bind\ I \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\delta\rangle) \to \langle\rho[\delta/I],\sigma,\langle\rangle\rangle, ?$
- $find\ I \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\rangle) \to \langle\rho,\sigma,\langle\rho[I]\rangle\rangle, ?$
- $\alpha_1\ !\ \alpha_2 \equiv \lambda\psi.\ \alpha_2\ (\alpha_1\ \psi)$

- In defining the compound action formed by the action combinator "&" a problem arises - i.e. no particular order should be specified in performing A_1 and A_2 *collaterally*. Both A_1 and A_2 can have imperative and/or declarative facets and might possibly interfere with each other. Here we arbitrarily choose to perform A_1 before A_2 :

  $\alpha_1 \;\&\; \alpha_2 \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\langle\rho'',\sigma'',\delta^{*\prime}\S\delta^{*\prime\prime}\rangle$

  where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_1\langle\rho,\sigma,\delta^*\rangle$ and $\langle\rho'',\sigma'',\delta^{*\prime\prime}\rangle=\alpha_2\langle\rho',\sigma',\delta^*\rangle$.

- *select* $\alpha_1\ \alpha_2\ \alpha_3 \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\delta^{*\prime}\downarrow 1 \rightarrow \alpha_2\langle\rho',\sigma',\delta^*\rangle,\ \alpha_3\langle\rho',\sigma',\delta^*\rangle$

  where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_1\langle\rho,\sigma,\delta^*\rangle$

- We want *iterate* $\alpha_1\ \alpha_2 \equiv$ *select* $\alpha_1\ (\alpha_2!(\textit{iterate}\ \alpha_1\ \alpha_2))$ *copy*

  $\equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\delta^{*\prime}\downarrow 1 \rightarrow$

  $(\alpha_2!(\textit{iterate}\ \alpha_1\ \alpha_2))\langle\rho',\sigma',\delta^*\rangle,\ \langle\rho',\sigma',\delta^*\rangle$

  where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_1\langle\rho,\sigma,\delta^*\rangle$.

  If $\alpha'=$*iterate* $\alpha_1\ \alpha_2$ then we want

  $\alpha' \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\delta^{*\prime}\downarrow 1 \rightarrow (\alpha_2!(\alpha'\alpha_1\alpha_2))\langle\rho',\sigma',\delta^*\rangle,\ \langle\rho',\sigma',\delta^*\rangle$

  where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_1\langle\rho,\sigma,\delta^*\rangle$.

  If we regard the right hand side of this equation as a function H of α so that

  $H(\alpha) \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\delta^{*\prime}\downarrow 1 \rightarrow (\alpha_2!(\alpha\alpha_1\alpha_2))\langle\rho',\sigma',\delta^*\rangle,\ \langle\rho',\sigma',\delta^*\rangle$

  where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_1\langle\rho,\sigma,\delta^*\rangle$,

  then we need $\alpha' = H(\alpha')$. Thus α' is a fixed point of H. As our earlier discussions about fixed points have indicated, we require the minimal fixed point of H. So we use our minimal fixed point operator *fix* :

  $\alpha' = \textit{fix}H$.

  iterate $\alpha_1\ \alpha_2 \equiv \textit{fix}(\lambda\alpha.\lambda\langle\rho,\sigma,\delta^*\rangle.\delta^{*\prime}\downarrow 1 \rightarrow$

  $(\alpha_2!(\alpha\alpha_1\alpha_2))\langle\rho',\sigma',\delta^*\rangle,\ \langle\rho',\sigma',\delta^*\rangle)$

  where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_1\langle\rho,\sigma,\delta^*\rangle$.

- *block* $\alpha_1 \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.\langle\rho,\sigma',\delta^{*\prime}\rangle$ where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_1\langle\rho,\sigma,\delta^*\rangle$

- *abstract* $\alpha_0 \equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*=\langle\rangle) \rightarrow \langle\rho,\sigma,\langle\rho,\alpha_0\rangle\rangle$, ?

- *apply* $\equiv \lambda\langle\rho,\sigma,\delta^*\rangle.(\delta^*\downarrow 1=\langle\rho_0,\alpha_0\rangle) \rightarrow \langle\rho,\sigma',\delta^{*\prime}\rangle$, ?

  where $\langle\rho',\sigma',\delta^{*\prime}\rangle=\alpha_0\langle\rho_0,\sigma,\delta^*\dagger 1\rangle$.

This completes our formal definition of the basic semantic algebra.

The source code of a direct implementation of this semantic algebra in Franz LISP (Foderaro et al. [1983], Wilensky [1984]) is listed in Appendix B. Note that the domain of denotable and storable values in the implementation is extended to include all LISP S-expressions.

5.4 The λ-Calculus - A Semantic Algebraic Definition

Here we give a definition of the λ-calculus with atoms using the basic semantic algebra. We permit as basic values the integers and the truth values, along with appropriate primitive functions acting upon them.

Syntactic Categories

I ∈ Ide	(the usual identifiers)
B ∈ Bas	(bases or atoms)
N ∈ Nml	(integer numerals)
E ∈ Exp	(λ-expressions)

Syntax

E ::= I | B | λI.E | E_0E_1 | if E_0 then E_1 else E_2
B ::= true | false | N | succ | pred | not
| eq | lt | sum | product | and

Semantic Functions

N : [Nml → N]
(maps the numerals to their corresponding integer values)

B : [Bas → A]

B[true] = *take true*
B[false] = *take false*
B[N] = *take* N[N]
B[succ] = *abstract* ((*takefirst* ! (*bind* I_1))
! (((*find* I_1) ! *apply*) ! (*applyop successor*)))
(similarly for "pred" and "not")

B[eq] = *abstract* ((*takefirst* ! (*bind* I_1))
! (*abstract* ((*takefirst* ! (*bind* I_2))
! ((((*find* I_1) ! *apply*)
& ((*find* I_2) ! *apply*))
! (*applyop equal*)))))

(similarly for "lt", "sum", "product" and "and")

E : [Exp → A]

E[I] = (*find* I) ! *apply*
E[B] = B[B]
E[λI.E] = *abstract* ((*takefirst* ! (*bind* I)) ! E[E])
E[E_0E_1] = (E[E_0] & (*abstract* E[E_1])) ! *apply*
E[if E_0 then E_1 else E_2] = *select* E[E_0] E[E_1] E[E_2]

We want primitive operators to behave as though they are (curried) λ-abstractions and so their actions are defined similarly to those of abstractions. This means that the action of a primitive operator applied to an insufficient number of arguments will still be partially evaluated, using the available arguments.

The semantics given here specifically belong to the class "call-by-name". Arguments to functions are evaluated only when required, as evidenced by the use of *abstract* in the semantic equation for applications and *apply* in that for identifiers. *abstract* creates a closure consisting of the action of the argument of the application frozen in the environment in which the argument occurs. The action of the function of the application binds this closure to an identifier - i.e. the formal parameter of the function. Whenever this identifier occurs in the body of the function the action of the closure bound to it is executed by *apply*. This evaluation takes place every time the identifier occurs in the body of the function. If the formal parameter identifier never occurs then the action of the argument is never evaluated. As discussed in 3.6 and 4.1 this characterises call-by-name.

Furthermore, an interpreter based on this semantic algebra definition will terminate immediately given any unapplied function (λ-abstraction) as input. This is because the closure formed for an unapplied function is never evaluated by the occurrence of *apply* in

the semantic equation for applications. As discussed in 3.6 such an interpreter will terminate when given the fixed point combinator Y as input (expression (b) in 3.6), returning the equivalent fixed point function of the semantic domain.

Appendix C lists the source code of the interpreter obtained by directly implementing the above semantic definition for the λ-calculus with atoms. It is built on top of the implementation of the basic semantic algebra given in Appendix B.

5.5 A More Practical Example

As a more practical example of the semantic algebra approach to programming language definition we give the semantics of a simple iterative programming language vaguely reminiscent of Pascal. The concrete syntax of this language corresponds to the abstract syntax for the example language EX used by Watt [1986].

Syntactic Categories

I∈Ide	(the usual identifiers)
N∈Nml	(integer numerals)
P∈Prog	(programs)
C∈Com	(commands)
E∈Exp	(expressions)
A∈Args	(arguments)
D∈Decl	(declarations)
B∈Bnds	(binders or parameters)

Syntax

P ::= program I; C.	
C ::= skip	(null command)
| I:= E	(assignment)
| I(A)	(procedure call)
| C_1;C_2	(sequencing)
| if E_1 then E_2 else E_3	(conditional)
| while E do C	(conditional iteration)
| for I:= E_1 to E_2 do C	(specified iteration)
| D;C	(declaration)

```
    | read(I1,I2)                                          (file read)
    | write(I,E)                                          (file write)
    | rewrite(I)                                        (file rewrite)
E ::= N | true | false | I | val(I) | E1+E2 | E1=E2 | E1<E2
    | E1 and E2 | not E | eof(I)
A ::= E | E,A
D ::= const I=E                              (constant declaration)
    | var I:= E                              (variable declaration)
    | proc I(B)=C                          (procedure declaration)
    | D1;D2                                (sequenced declaration)
B ::= I | I,B
```

Semantic Functions

N : [Nml → N]

(maps the numerals to their corresponding integer values)

P : [Prog → A]

P[program I; C.] = C[C]

C : [Com → A]

C[skip] = *skip*

C[I:= E] = (E[E] & *find* I) ! *store*

C[I(A)] = (*find* I & A[A]) ! *apply*

C[C_1;C_2] = C[C_1] ! C[C_2]

C[if E_1 then E_2 else E_3] = *select* E[E_1] E[E_2] E[E_3]

C[while E do C] = *iterate* E[E] C[C]

```
C[for I:= E1 to E2 do C] = evaluatebounds !
                           (iterate withinbounds
                                   ((updatecontrolvar ! E[C]) &
                                    increment)) !
                           terminate
```

where

evaluatebounds = E[E_1] & E[E_2]

withinbounds = *applyop less-equal*

updatecontrolvar = (*takefirst* & (*discard* ! *find* I)) ! *store*

increment = (*takefirst* ! *applyop successor*) & *takerest*

```
          terminate = discard
C⟦D;C⟧ = block (D⟦D⟧ ! C⟦C⟧)
C⟦read(I1,I2)⟧ = find I1 ! fetch ! applyop read !
                 ( ((takefirst & (discard ! find I2)) ! store) &
                   ((takerest & (discard ! find I1)) ! store)
                 )
C⟦write(I,E)⟧ = ((((find I ! fetch) & E⟦E⟧) ! applyop write) & find I)
                ! store
C⟦rewrite(I)⟧ = (applyop rewrite & find I) ! store

    E : [Exp → A]

E⟦N⟧ = take N⟦N⟧
E⟦true⟧ = take true
E⟦false⟧ = take false
E⟦I⟧ = find I
E⟦val(I)⟧ = find I ! fetch
E⟦E1+E2⟧ = (E⟦E1⟧ & E⟦E2⟧) ! applyop sum
E⟦E1=E2⟧ = (E⟦E1⟧ & E⟦E2⟧) ! applyop equal
E⟦E1<E2⟧ = (E⟦E1⟧ & E⟦E2⟧) ! applyop less
E⟦E1 and E2⟧ = (E⟦E1⟧ & E⟦E2⟧) ! applyop conjunction
E⟦not E⟧ = E⟦E⟧ ! applyop complement
E⟦eof(I)⟧ = find I ! fetch ! applyop end-of-file

    A : [Args → A]

A⟦E⟧ = E⟦E⟧
A⟦E,A⟧ = E⟦E⟧ & A⟦A⟧

    D : [Decl → A]

D⟦const I=E⟧ = E⟦E⟧ ! bind I
D⟦var I:= E⟧ = E⟦E⟧ ! allocate ! bind I
D⟦proc I(B)=C⟧ = abstract (B⟦B⟧ ! C⟦C⟧) ! bind I
D⟦D1;D2⟧ = D⟦D1⟧ ! D⟦D2⟧

    B : [Bnds → A]

B⟦I⟧ = bind I
```

B⟦I,B⟧ = (*takefirst* ! *bind* I) & (*takerest* ! *B*⟦B⟧)

Once again, we assume that any ambiguities arising in the syntax are resolved by the use of precedence or association rules, or by explicit bracketing. The conventions for Pascal are assumed here and the cases in which such rules are required are obvious.

This semantic description is relatively easy to comprehend, once familiarity with the semantic algebra has been attained. As an example let us consider the assignment command - i.e. I:= E. First we must collaterally evaluate the expression E (using our semantic function *E*) and find the location which is bound to the variable identifier I in the environment. The resulting pair of values is then piped into the *store* action which completes the assignment.

A more demanding example is procedure declaration and calls. The procedure declaration (i.e. proc I(M)=C) constructs a closure and then binds it to the procedure identifier I. The closure is constructed by *abstract* from an action that expects as input a list of arguments, which it binds to the formal parameters M, and then executes the body of the procedure C. The procedure call (i.e. I(A)) finds in the environment the closure bound to the procedure identifier I, collaterally evaluates the arguments A and then applies the closure to the arguments.

We model the "if" and "while" commands directly by the action combinators *select* and *iterate*. The "for" command (i.e. for I:= E_1 to E_2 do C) illustrates the more general usefulness of these action combinators. This command requires that no value outside the (possibly empty) range defined by the low and high bounds (E_1 and E_2 respectively) may be assigned to the control variable I. The semantic definition for this command makes use of several auxiliary actions - i.e. *evaluatebounds*, *withinbounds*, etc. The pair of input values to the *iterate* composite action (produced by *evaluatebounds*) are fed into both its component actions. *withinbounds* simply compares the two bounds. The compound action (*updatecontrolvar* ! *C*⟦C⟧) stores the first of the bounds in the control variable and then executes the body of the loop. Collaterally with this, *increment* copies the two bounds, incrementing the first. This updated pair of bounds becomes the input to the next iteration. *terminate* simply discards the pair of bounds after the completion of the loop.

This semantic definition of a simple Pascal-like language gives some indication of the general applicability of the semantic algebra approach.

5.6 Summary and Conclusions

In this Chapter we have applied Mosses' idea of a semantic algebra to programming language definition. The concrete basic semantic algebra we have given has been applied to the definition of a simple functional language and a simple procedural language - i.e. respectively the λ-calculus with atoms and a simple Pascal-like language. By defining their semantics using the basic semantic algebra we have shown it to be adequate for describing most of the essential features of conventional procedural and functional languages. In particular cases additional actions may be needed to accomodate such features as dynamic scoping, exceptions, jumps, recursive bindings and concurrency.

Furthermore, we have directly implemented the basic semantic algebra and the semantic description of the λ-calculus with atoms to achieve a working interpreter for the λ-calculus. The source code for these implementations is listed in Appendices B and C.

This chapter provides us with a practical indication that denotational semantic descriptions of programming languages using the semantic algebra approach are practical, comprehensible and concise.

Chapter 6
APPLICATIONS AND FURTHER WORK

This brief concluding chapter presents an overview of the application of denotational and semantic algebra approaches to programming language semantics and also indicates areas requiring further work.

6.1 Applications of Semantic Algebras

Various applications of semantic algebra definitions of programming languages can be envisaged. In Chapter 5, implementing the basic semantic algebra allowed the semantic description of the λ-calculus with atoms to be directly executed. Directly executable semantic descriptions using semantic algebras are advocated by Watt [1986] who also envisaged semantic algebras playing a part in automatic compiler construction. The compiler writing system would assume a common basic semantic algebra for all language descriptions and construct compilers from descriptions in that algebra. The application of denotational semantics to the construction of code generators has been discussed and developed in Mosses [1976, 1980], Jones & Schmidt [1980], Allison [1983], Schmidt [1985] and Vickers [1986].

Axiomatic definition of semantic algebras (Mosses [1984]) will possibly allow the development of automated program verification systems. Axiomatic definitions have been most widely used for such automated proofs of programs (Suzuki [1980]), although denotational semantic specifications have found application in program proving (de Bakker [1977, 1980], Gordon et al. [1979], McGettrick [1980]). Many of the underlying proof methods are due to Manna [1974, 1980]. Other approaches to program verification have been developed by Boyer & Moore [1975], Bates & Constable [1985] and Voda [1985].

6.2 Further Work

The application of the semantic algebra approach requires further investigation in the areas indicated above as well as in other areas. The most obvious need is for the approach to be applied to various

other programming styles thereby providing some sort of "semantic bridge" between them. Prime candidates for investigation include logic programming (van Emden & Kowalski [1976], Kowalski [1983], Kahn & Carlsson [1984], Carlsson [1984] and Lassez & Maher [1985]), object-oriented programming (Cardelli [1984]), stream programming (Ida & Tanaka [1983, 1984] and Pingali & Arvind [1985]) and concurrent programming (de Bruin & Böhm [1985]).

APPENDIX A

AN EFFICIENT CALL-BY-NEED LAMBDA CALCULUS INTERPRETER

```
; The Lambda-Calculus Interpreter.....

(def IS_LAMBDA_TERM
  (lambda (term)
    (or (IS_VARIABLE term)
        (IS_APPLICATION term)
        (IS_LAMBDA_ABSTRACTION term))))

(def IS_VARIABLE
  (lambda (term)
    (and (atom term)
         (not (eq term 'LAMBDA)))))

(def IS_APPLICATION
  (lambda (term) (and (listp term) (LENGTHP term 2))))

(def IS_LAMBDA_ABSTRACTION
  (lambda (term) (and (LENGTHP term 3)
                      (eq (car term) 'LAMBDA)
                      (IS_VARIABLE (cadr term)))))

(def MK_VARIABLE (lambda (name) name))

(def MK_APPLICATION
  (lambda (function argument) (list function argument)))

(def MK_LAMBDA_ABSTRACTION
  (lambda (binder form) (list 'LAMBDA binder form)))

(def NAME_OF (lambda (term) term))

(def FUNCTION_OF (lambda (term) (car term)))

(def ARGUMENT_OF (lambda (term) (cadr term)))
```

```
(def BINDER_OF (lambda (term) (cadr term)))

(def FORM_OF (lambda (term) (caddr term)))

(def LENGTHP (lambda (list n) (eq (length list) n)))

(def MK_VAR (lambda (name) (list 'VAR name 0)))

(def MK_APP (lambda (fun arg) (list 'APP fun arg)))

(def MK_LAM (lambda (bnd frm) (list 'LAM bnd frm)))

(def NAM_OF (lambda (var) (cadr var)))

(def FUN_OF (lambda (term) (cadr term)))

(def ARG_OF (lambda (term) (caddr term)))

(def BND_OF (lambda (term) (cadr term)))

(def FRM_OF (lambda (term) (caddr term)))

(def IS_VAR (lambda (term) (eq (car term) 'VAR)))

(def IS_APP (lambda (term) (eq (car term) 'APP)))

(def IS_LAM (lambda (term) (eq (car term) 'LAM)))

(def IS_ARID (lambda (env) (null env)))

(def MK_ARID (lambda () nil))

(def MK_BIND (lambda (var val env) (cons (cons var val) env)))

(def VAR_OF (lambda (env) (caar env)))

(def VAL_OF (lambda (env) (cdar env)))
```

```
(def REST_OF (lambda (env) (cdr env)))

(def LOOK_UP
  (lambda (var env)
    (cond ((or (IS_ARID env) (eq var (VAR_OF env))) env)
          (t (LOOK_UP var (REST_OF env))))))

(def REPRESENT (lambda (term) (REP term (MK_ARID))))

(def REP
  (lambda (term repenv)
    (cond ((IS_VARIABLE term)
           (let ((env (LOOK_UP term repenv)))
                (cond ((IS_ARID env) (MK_VAR (NAME_OF term)))
                      (t (VAL_OF env)))))
          ((IS_APPLICATION term)
           (MK_APP (REP (FUNCTION_OF term) repenv)
                   (REP (ARGUMENT_OF term) repenv)))
          ((IS_LAMBDA_ABSTRACTION term)
           (let ((var (MK_VAR (NAME_OF (BINDER_OF term)))))
                (MK_LAM var (REP (FORM_OF term)
                                 (MK_BIND (BINDER_OF term)
                                          var
                                          repenv)))))
          (t (invalid term)))))

(def IS_VOID (lambda (scolis) (null scolis)))

(def MK_VOID (lambda () nil))

(def MK_SCO (lambda (var scolis) (cons var scolis)))

(def HEAD_OF (lambda (scolis) (car scolis)))

(def TAIL_OF (lambda (scolis) (cdr scolis)))

(def IS_SUSP (lambda (susp) (eq (car susp) 'SUSP)))
```

```
(def MK_SUSP (lambda (term env) (list 'SUSP term env)))

(def TERM_OF (lambda (susp) (cadr susp)))

(def ENV_OF (lambda (susp) (caddr susp)))

(def REDUCE_TO_NORMAL_FORM
  (lambda (term)
     (UNREPRESENT (RTNF (REPRESENT term) (MK_ARID) (MK_VOID)))))

(def RTNF
  (lambda (term lexenv dynsco)
    (progn (CALL-TRACE 'RTNF term lexenv dynsco)
           (cond ((IS_VAR term)
                  (let ((trm (RTNF_VAR term lexenv dynsco)))
                       (progn (EXIT-TRACE 'RTNF trm lexenv dynsco)
                              trm)))
                 ((IS_APP term)
                  (let ((trm (RTNF_APP term lexenv dynsco)))
                       (progn (EXIT-TRACE 'RTNF trm lexenv dynsco)
                              trm)))
                 ((IS_LAM term)
                  (let ((trm (RTNF_LAM term lexenv dynsco)))
                       (progn (EXIT-TRACE 'RTNF trm lexenv dynsco)
                              trm)))
                 (t (invalid term))))))

(def RTNF_VAR
  (lambda (var lexenv dynsco)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env) (PROPAGATE_RENAMING var dynsco))
               (t (let ((susp (VAL_OF env)))
                       (cond ((IS_SUSP susp)
                              (let ((val (RTNF (TERM_OF susp)
                                               (ENV_OF susp)
                                               dynsco)))
                                   (progn (UPDATE_VAL_OF env val)
```

```
                                        val)))
                                (t (PROPAGATE_RENAMING susp
                                                       dynsco))))))))))

(def RTNF_APP
  (lambda (app lexenv dynsco)
    (let ((susp (RTLF (FUN_OF app) lexenv dynsco)))
         (cond ((IS_SUSP susp)
                (let ((fun (TERM_OF susp))
                      (env (ENV_OF susp)))
                     (RTNF (FORM_OF fun)
                           (MK_BIND (BND_OF fun)
                                    (MK_SUSP (ARG_OF app) lexenv)
                                    env)
                           dynsco)))
               (t (MK_APP susp (RTNF (ARG_OF app) lexenv dynsco)))))))

(def RTNF_LAM
  (lambda (lam lexenv dynsco)
    (let ((newvar (MK_VAR (NAM_OF (BND_OF lam)))))
         (MK_LAM newvar
                 (RTNF (FRM_OF lam)
                       (MK_BIND (BND_OF lam) newvar lexenv)
                       (MK_SCO newvar dynsco))))))

(def RTLF
  (lambda (term lexenv dynsco)
    (progn (CALL-TRACE 'RTLF term lexenv dynsco)
           (cond ((IS_VAR term)
                  (let ((trm (RTLF_VAR term lexenv dynsco)))
                       (progn (EXIT-TRACE 'RTLF trm lexenv dynsco)
                              trm)))
                 ((IS_APP term)
                  (let ((trm (RTLF_APP term lexenv dynsco)))
                       (progn (EXIT-TRACE 'RTLF trm lexenv dynsco)
                              trm)))
                 ((IS_LAM term)
```

```
                (let ((trm (RTLF_LAM term lexenv dynsco)))
                     (progn (EXIT-TRACE 'RTLF trm lexenv dynsco)
                            trm)))
               (t (invalid term))))))

(def RTLF_VAR
  (lambda (var lexenv dynsco)
    (let ((env (LOOK_UP var lexenv)))
         (cond ((IS_ARID env)
                (PROPAGATE_RENAMING var dynsco))
               (t (let ((susp (VAL_OF env)))
                       (cond ((IS_SUSP susp)
                              (let ((val (RTLF (TERM_OF susp)
                                               (ENV_OF susp)
                                               dynsco)))
                                   (progn (UPDATE_VAL_OF env val)
                                          val)))
                             ((IS_LAM susp) (MK_SUSP susp (MK_ARID)))
                             (t (PROPAGATE_RENAMING susp
                                                    dynsco)))))))))

(def RTLF_APP
  (lambda (app lexenv dynsco)
    (let ((susp (RTLF (FUN_OF app) lexenv dynsco)))
         (cond ((IS_SUSP susp)
                (let ((fun (TERM_OF susp))
                      (env (ENV_OF susp)))
                     (RTLF (FORM_OF fun)
                           (MK_BIND (BND_OF fun)
                                    (MK_SUSP (ARG_OF app) lexenv)
                                    env)
                           dynsco)))
               (t (MK_APP susp (RTNF (ARG_OF app) lexenv dynsco)))))))

(def RTLF_LAM (lambda (lam lexenv dynsco) (MK_SUSP lam lexenv)))

(def UPDATE_VAL_OF (lambda (env val) (rplacd (car env) val)))
```

```
(def REN_OF (lambda (var) (caddr var)))

(def NAR_OF (lambda (var) (cdddr var)))

(def UPDATE_REN_OF (lambda (var ren) (rplaca (cddr var) ren)))

(def UPDATE_NAR_OF (lambda (var nar) (rplacd (cddr var) nar)))

(def DECORATE
  (lambda (var)
    (progn (UPDATE_REN_OF var (add1 (REN_OF var)))
           (DECNAR (NAR_OF var)))))

(def DECNAR
  (lambda (narlist)
    (cond ((null narlist) nil)
          (t (progn (DECORATE (car narlist))
                    (DECNAR (cdr narlist)))))))

(def RENAME
  (lambda (var dynsco)
    (cond ((IS_VOID dynsco) var)
          (t (let ((val (HEAD_OF dynsco)))
                  (cond ((eq var val) var)
                        (t (progn
                             (cond
                               ((and (eq (NAM_OF var)
                                         (NAM_OF val))
                                     (eq (REN_OF var)
                                         (REN_OF val)))
                                (progn
                                  (DECORATE val)
                                  (UPDATE_NAR_OF
                                    var
                                    (cons val (NAR_OF var)))))
                               (t nil))
                             (RENAME var (TAIL_OF dynsco)))))))))))
```

```
(def PROPAGATE_RENAMING
  (lambda (term dynsco)
    (cond ((IS_VAR term) (RENAME term dynsco))
          ((IS_APP term) (progn
                           (PROPAGATE_RENAMING (FUN_OF term) dynsco)
                           (PROPAGATE_RENAMING (ARG_OF term) dynsco)
                           term))
          ((IS_LAM term) (progn
                           (PROPAGATE_RENAMING (FRM_OF term) dynsco)
                           term))
          (t (invalid term)))))

(def UNREPRESENT
  (lambda (term)
    (cond ((IS_VAR term)
           (cond ((eq (REN_OF term) 0) (MK_VARIABLE (NAM_OF term)))
                 (t (MK_VARIABLE (concat (NAM_OF term)
                                         (REN_OF term))))))
          ((IS_APP term) (MK_APPLICATION
                           (UNREPRESENT (FUN_OF term))
                           (UNREPRESENT (ARG_OF term))))
          ((IS_LAM term)
           (let ((bnd (UNREPRESENT (BND_OF term)))
                 (frm (UNREPRESENT (FRM_OF term))))
                (cond ((and (IS_APPLICATION frm)
                            (eq bnd (ARGUMENT_OF frm))
                            (not (IS_FREE_IN bnd (FUNCTION_OF frm))))
                       (FUNCTION_OF frm))
                      (t (MK_LAMBDA_ABSTRACTION bnd frm)))))
          (t (invalid term)))))

; INTERIM_UNREPRESENT is used by the tracing functions to print out
; intermediate results of a reduction. It differs from REPRESENT only
; in that it caters for suspensions in the intermediate result.
(def INTERIM_UNREPRESENT
  (lambda (term)
    (cond ((IS_VAR term)
```

```
          (cond ((eq (REN_OF term) 0) (MK_VARIABLE (NAM_OF term)))
                (t (MK_VARIABLE (concat (NAM_OF term)
                                        (REN_OF term))))))
         ((IS_APP term) (MK_APPLICATION
                         (INTERIM_UNREPRESENT (FUN_OF term))
                         (INTERIM_UNREPRESENT (ARG_OF term))))
         ((IS_LAM term) (MK_LAMBDA_ABSTRACTION
                         (INTERIM_UNREPRESENT (BND_OF term))
                         (INTERIM_UNREPRESENT (FRM_OF term))))
         ((IS_SUSP term) (INTERIM_UNREPRESENT (TERM_OF term)))
         (t (invalid term)))))

(def invalid
  (lambda (term)
    (prog ()
      (print term " NOT a valid lambda expression")
      (terpr))))

; The tracing functions...
(def CALL-TRACE
  (lambda (call trm lex dyn)
    (cond (TRACE
           (prog ()
             (setq count (add1 count))
             (do i 1 (1+ i) (> i count) (princ "-" outport))
             (print call outport)
             (princ " CALLED: TRM = " outport)
             (print (INTERIM_UNREPRESENT trm) outport)
             (terpr outport)
             (do i 1 (1+ i) (> i count) (princ " " outport))
             (princ "             LEX = " outport)
             (cond ((not lex) (princ "ARID" outport))
                   (t (print lex outport)))
             (terpr outport)
             (do i 1 (1+ i) (> i count) (princ " " outport))
             (princ "             DYN = " outport)
             (cond ((not dyn) (princ "VOID" outport))
```

```
                (t (print dyn outport)))
            (terpr outport)
            (terpr outport))))))

(def EXIT-TRACE
  (lambda (exit trm lex dyn)
    (cond (TRACE
           (prog ()
             (do i 1 (1+ i) (> i count) (princ "-" outport))
             (print exit outport)
             (princ " EXITED: TRM = " outport)
             (print (INTERIM_UNREPRESENT trm) outport)
             (terpr outport)
             (do i 1 (1+ i) (> i count) (princ " " outport))
             (princ "             LEX = " outport)
             (cond ((not lex) (princ "ARID" outport))
             (t (print lex outport)))
             (terpr outport)
             (do i 1 (1+ i) (> i count) (princ " " outport))
             (princ "             DYN = " outport)
             (cond ((not dyn) (princ "VOID" outport))
                   (t (print dyn outport)))
             (terpr outport)
             (terpr outport)
             (setq count (sub1 count)))))))

; EXPAND expands any bound symbol (e.g. a combinator, function or
; expression name) to its corresponding lambda-calculus definition.
; This allows the pre-definition of LISP variables to stand for
; lambda-calculus expressions, e.g.
;           (setq true '(LAMBDA x LAMBDA y x))
; as for t in 2.3.1.
(def EXPAND
  (lambda (var)
    (cond ((numberp var) var)
          ((boundp var) (symeval var))
          (t var))))
```

```
; insert-brackets brackets a given lambda-term in the conventional
; way, as described in Chapter I. This enables terms to be input in
; the standard form with a minimum of bracketing.
(def insert-brackets
  (lambda (term)
    (cond ((IS_VARIABLE term) (EXPAND term))
          ((or (eq (car term) '!) (eq (car term) 'LAMBDA))
           (list 'LAMBDA (cadr term) (insert-brackets (cddr term))))
         ((LENGTHP term 1) (insert-brackets (car term)))
         ((LENGTHP term 2) (list (insert-brackets (car term))
                                 (insert-brackets (cadr term))))
         (t (insert-brackets
              (append (list
                        (list (car term)
                              (cadr term)))
                      (cddr term)))))))

(def start                    ; starts the lambda-calculus interpreter...
  (lambda ()
    (prog (r result)
      (cond (TRACE (setq outport (outfile 'trace.out))))
      (setq count -1)
      loop
      (princ "> ")
      (setq r (lineread))
      (cond ((equal r '(exit)) (go exit1)))
      (cond (TRACE (prog () (princ "TERM IS : " outport)
      (print r outport)
      (terpr outport))))
      (setq result (REDUCE_TO_NORMAL_FORM (insert-brackets r)))
      (cond (TRACE (prog () (princ "NORMAL FORM IS : " outport)
      (print result outport)
      (terpr outport)
      (terpr outport))))
      (print result)
      (terpr)
      (go loop)
```

```
      exit1
      (cond (TRACE (close outport))))))

(setq TRACE nil)

; The following are the definitions for the data types and functions
; given in 2.3 which may be used in the interpreter. These may be used
; since the interpreter expands any occurrence of a bound symbol to
; its corresponding expression using EXPAND, prior to attempting any
; reduction.

; Boolean values and functions...

; Represent true by...
(setq tt (insert-brackets '(! x ! y x)))

; and false by...
(setq ff (insert-brackets '(! x ! y y)))

(setq Not (insert-brackets '(! u u ff tt)))

(setq And (insert-brackets '(! u ! v u v ff)))

(setq Or (insert-brackets '(! u u tt )))

(setq Equiv (insert-brackets '(! u ! v u v (v ff tt))))

(setq Impl (insert-brackets '(! u ! v u v tt)))

; Pairs...

(setq Pair (insert-brackets '(! u u x y)))

(setq I1 (insert-brackets '(! z z tt)))

(setq I2 (insert-brackets '(! z z ff)))
```

```
; Non-negative integers...

(setq Zero (insert-brackets '(! f ! x x)))

(setq Succ (insert-brackets '(! k ! f ! x f (k f x))))

(setq One (insert-brackets '(Succ Zero)))

(setq Two (insert-brackets '(Succ One)))

(setq Three (insert-brackets '(Succ Two)))

(setq Four (insert-brackets '(Succ Three)))

(setq Five (insert-brackets '(Succ Four)))

(setq Six (insert-brackets '(Succ Five)))

(setq Seven (insert-brackets '(Succ Six)))

(setq Eight (insert-brackets '(Succ Seven)))

(setq Nine (insert-brackets '(Succ Eight)))

(setq Ten (insert-brackets '(Succ Nine)))

(setq IsZero (insert-brackets '(! k k (tt ff) tt)))

(setq Sum (insert-brackets '(! m ! n m Succ n)))

(setq Product (insert-brackets '(! m ! n m (n Succ) Zero)))

(setq Pred
  (insert-brackets
    '(! k (k (! p ! u u (Succ (p tt)) (p tt)) (! u u Zero Zero)) ff)))

(setq Difference (insert-brackets '(! m ! n n Pred m)))
```

```
; Definitions for the factorial function...

(setq Y (insert-brackets '(! h (! x h (x x)) (! x h (x x)))))

(setq H_Fact
  (insert-brackets
    '(! f ! n (IsZero n) One (Multiply n (f (Pred n))))))

(setq Fact (insert-brackets '(Y H_Fact)))

(start)
```

APPENDIX B

IMPLEMENTATION OF THE SEMANTIC ALGEBRA

```
; Some operators...

(def EQUAL
  (lambda (vals)
    (cond ((equal (length vals) 2)
           (list (equal (car vals) (cadr vals))))
          (t (error "EQUAL : length of value list not 2")))))

(def LESS
  (lambda (vals)
    (cond ((equal (length vals) 2)
           (let ((N1 (car vals)) (N2 (cadr vals)))
                (cond ((and (IS-INTEGER N1) (IS-INTEGER N2))
                       (list (<& N1 N2)))
                      (t (error "LESS : non-integer argument")))))
          (t (error "LESS : length of value list not 2")))))

(def SUM
  (lambda (vals)
    (cond ((equal (length vals) 2)
           (let ((N1 (car vals)) (N2 (cadr vals)))
                (cond ((and (IS-INTEGER N1) (IS-INTEGER N2))
                       (list (+ N1 N2)))
                      (t (error "SUM : non-integer argument")))))
          (t (error "SUM : length of value list not 2")))))

(def PRODUCT
  (lambda (vals)
    (cond ((equal (length vals) 2)
           (let ((N1 (car vals)) (N2 (cadr vals)))
                (cond ((and (IS-INTEGER N1) (IS-INTEGER N2))
                       (list (* N1 N2)))
                      (t (error "PROODUCT : non-integer argument")))))
          (t (error "PRODUCT : length of value list not 2")))))

(def SUCCESSOR
```

```
  (lambda (vals)
    (cond ((equal (length vals) 1)
           (let ((N (car vals)))
                (cond ((IS-INTEGER N) (list (1+ N)))
                      (t (error
                           "SUCCESSOR : non-integer argument")))))
          (t (error "SUCCESSOR : length of value list not 1")))))

(def PREDECESSOR
  (lambda (vals)
    (cond ((equal (length vals) 1)
           (let ((N (car vals)))
                (cond ((IS-INTEGER N) (list (1- N)))
                      (t (error
                           "PREDECESSOR : non-integer argument")))))
          (t (error "PREDECESSOR : length of value list not 1")))))

(def CONJUNCTION
  (lambda (vals)
    (cond ((equal (length vals) 2)
           (let ((B1 (car vals)) (B2 (cadr vals)))
                (cond ((and (IS-BOOLEAN B1) (IS-BOOLEAN B2))
                       (list (and B1 B2)))
                      (t (error
                           "CONJUNCTION : non-Boolean argument")))))
          (t (error "CONJUNCTION : length of value list not 2")))))

(def COMPLEMENT
  (lambda (vals)
    (cond ((equal (length vals) 1)
           (let ((B (car vals)))
                (cond ((IS-BOOLEAN B) (list (not B)))
                      (t (error
                           "COMPLEMENT: non-Boolean argument")))))
          (t (error "COMPLEMENT : length of value list not 1")))))

(def END-OF-FILE
  (lambda (vals)
    (cond ((equal (length vals) 1)
```

```
          (let ((F (car vals)))
               (cond ((IS-FILE F) (list (null F)))
                     (t (error "END-OF-FILE : non-file argument")))))
         (t (error "END-OF-FILE : length of value list not 1")))))

(def READ
  (lambda (vals)
    (cond ((equal (length vals) 1)
           (let ((F (car vals)))
                (cond ((IS_FILE F) (list (car F) (cdr F)))
                      (t (error "READ : non-file argument")))))
          (t (error "READ : length of value list not 1")))))

(def WRITE
  (lambda (vals)
    (cond ((equal (length vals) 2)
           (let ((F (car vals)) (val (cadr vals)))
                (cond ((IS-FILE F) (list (append1 F val)))
                      (t (error "WRITE : non-file argument")))))
          (t (error "WRITE : length of value list not 2")))))

(def REWRITE
  (lambda (vals)
    (cond ((null vals) (list nil))
          (t (error "REWRITE : non-empty value list")))))

; The basic operations on stores and environments...

(def EMPTY (lambda (loc) '(nil nil)))

(def HOLD (lambda (loc store) (car (apply store (list loc)))))

(def AREA (lambda (loc store) (cadr (apply store (list loc)))))

(def UPDATE
  (lambda (loc val store)
    (list 'lambda '(loc1)
      (list 'cond (list (list '= 'loc1 (list 'quote loc))
                        (list 'list (list 'quote val) t))
```

```
                (list t (list store 'loc1))))))

; We allow a store with 100 cells...
(setq *storesize* 100)

(def NEW (lambda (store) (NEW-NEXT 1 store)))

(def NEW-NEXT
  (lambda (address store)
    (let ((loc (maknam 'l address)))
         (cond ((>= address *storesize*)
                (error "NEW-NEXT : run out of store"))
               ((AREA loc store) (NEW-NEXT (1+ loc) store))
               (t loc)))))

(def ARID (lambda (ident) (error "ARID : unbound identifier")))

(def MODIFY-ENV
  (lambda (env val ident)
    (list 'lambda '(ident1)
      (list 'cond (list (list 'equal 'ident1 (list 'quote ident))
                        (list 'quote val))
                  (list t (list env 'ident1))))))

; Value domain predicates...

(def IS-BOOLEAN (lambda (val) (or (equal val t) (equal val nil))))

(def IS-INTEGER (lambda (val) (fixp val)))

(def IS-LOCATION (lambda (val) (equal (car (explode val)) 'l)))

(def IS-FILE (lambda (val) (listp val)))

(def IS-CLOSURE (lambda (val) (and (listp val) (= (length val) 2))))

; State selectors...

(def ENV-OF (lambda (state) (car state)))
```

```
(def STORE-OF (lambda (state) (cadr state)))

(def VALS-OF (lambda (state) (caddr state)))

;The semantic algebra...

(def TAKE
  (lambda (val state)
    (cond ((null (VALS-OF state))
           (list (ENV-OF state) (STORE-OF state) (list val)))
          (t (error "TAKE : non-empty value list")))))

(def APPLYOP
  (lambda (op state)
    (list (ENV-OF state) (STORE-OF state)
          (apply op (list (VALS-OF state))))))

(def TAKEFIRST
  (lambda (state)
    (list (ENV-OF state) (STORE-OF state)
          (list (car (VALS-OF state))))))

(def TAKEREST
  (lambda (state)
    (list (ENV-OF state) (STORE-OF state) (cdr (VALS-OF state)))))

(def COPY (lambda (state) state))

(def DISCARD
  (lambda (state) (list (ENV-OF state) (STORE-OF state) nil)))

(def SKIP
  (lambda (state)
    (cond ((null (VALS-OF state)) state)
          (t (error "SKIP : non-empty value list")))))


(def STORE
  (lambda (state)
```

```
    (let ((vals (VALS-OF state)))
         (cond ((= (length vals) 2)
                (let ((val (car vals)) (loc (cadr vals)))
                     (cond ((IS-LOCATION loc)
                            (list (ENV-OF state)
                                  (UPDATE loc val (STORE-OF state))
                                  nil))
                           (t (error "STORE : not a location")))))
               (t (error "STORE : value list is not length 2")))))

(def FETCH
  (lambda (state)
    (let ((store (STORE-OF state)) (vals (VALS-OF state)))
         (cond ((= (length vals) 1)
                (let ((loc (car vals)))
                     (cond ((IS-LOCATION loc)
                            (list (ENV-OF state) store
                                  (list (HOLD loc store))))
                           (t (error "FETCH : not a location")))))
               (t (error "FETCH : value list is not length 1"))))))

(def ALLOCATE
  (lambda (state)
    (let ((store (STORE-OF state)) (vals (VALS-OF state)))
         (cond ((= (length vals) 1)
                (let ((loc (NEW store)))
                     (list (ENV-OF state)
                           (UPDATE loc (car vals) store)
                           (list loc))))
               (t (error "ALLOCATE : value list is not length 1"))))))

(def BIND
  (lambda (ident state)
    (let ((vals (VALS-OF state)))
         (cond ((= (length vals) 1)
                (list (MODIFY-ENV (ENV-OF state) (car vals) ident)
                      (STORE-OF state) nil))
               (t (error "BIND : value list is not length 1"))))))
```

```
(def FIND
  (lambda (ident state)
    (cond ((null (VALS-OF state))
           (let ((env (ENV-OF state)))
                (list (ENV-OF state) (STORE-OF state)
                      (list (apply env (list ident))))))
          (t (error "FIND : non-empty value list")))))

(def APPLY-ACTION
  (lambda (action state)
    (cond ((atom action) (apply action (list state)))
          ((listp action)
           (apply (car action) (append1 (cdr action) state)))
          (t (error "APPLY-ACTION : not a valid action")))))

(def !
  (lambda (A1 A2 state) (APPLY-ACTION A2 (APPLY-ACTION A1 state))))

(def &
  (lambda (A1 A2 state)
    (let ((state1 (APPLY-ACTION A1 state)))
         (let ((state2 (APPLY-ACTION A2 (list (ENV-OF state1)
                                              (STORE-OF state1)
                                              (VALS-OF state)))))
              (list (ENV-OF state2) (STORE-OF state2)
                    (append (VALS-OF state1) (VALS-OF state2))))))))

(def SELECT
  (lambda (A1 A2 A3 state)
    (let ((state1 (APPLY-ACTION A1 state)))
         (let ((state2 (list (ENV-OF state1) (STORE-OF state1)
                             (VALS-OF state))))
              (cond ((car (VALS-OF state1))
                     (APPLY-ACTION A2 state2))
                    (t (APPLY-ACTION A3 state2))))))))

(def ITERATE
  (lambda (A1 A2 state)
    (APPLY-ACTION (list 'SELECT A1
```

```
                    (list '! A2 (list 'ITERATE A1 A2)) 'COPY)
                state)))

(def BLOCK
  (lambda (A1 state)
    (let ((state1 (APPLY-ACTION A1 state)))
         (list (ENV-OF state) (STORE-OF state1) (VALS-OF state1)))))

(def ABSTRACT
  (lambda (A0 state)
    (let ((env (ENV-OF state)))
         (cond ((null (VALS-OF state))
                (list env (STORE-OF state) (list (list env A0))))
               (t (error "ABSTRACT : non-empty value list"))))))

(def APPLY
  (lambda (state)
    (let ((vals (VALS-OF state)))
         (cond ((IS-CLOSURE (car vals))
                (let ((env0 (caar vals))
                      (A0 (cadar vals)))
                     (let (state1 (APPLY-ACTION A0
                                                (list env0
                                                      (STORE-OF state)
                                                      (cdr vals)))))
                          (list (ENV-OF state) (STORE-OF state1)
                                (VALS-OF state1)))))
               (t (error
                    "APPLY : first value not a valid closure"))))))
```

APPENDIX C

IMPLEMENTATION OF THE SEMANTIC ALGEBRA

DEFINITION OF THE LAMBDA CALCULUS WITH ATOMS

```
(load 'semalg.l)

(def LENGTHP
  (lambda (list n)
    (cond ((IS-IDENT list) (= n 1))
          (t (= (length list) n)))))

; Syntactic Predicates...

(def IS-IDENT
  (lambda (exp)
    (and (atom exp)
         (not (or (equal exp 'LAMBDA)
                  (IS-BASE exp)
                  (equal exp 'if)
                  (equal exp 'then)
                  (equal exp 'else))))))

(def IS-NUMERAL (lambda (exp) (fixp exp)))

(def IS-BASE
  (lambda (exp)
    (or (equal exp 'true) (equal exp 'false) (IS-NUMERAL exp)
        (equal exp 'succ) (equal exp 'pred) (equal exp 'not)
        (equal exp 'eq) (equal exp 'lt)
        (equal exp 'sum) (equal exp 'product)
        (equal exp 'and))))

(def IS-LAMBDA-ABSTRACTION
  (lambda (exp)
    (and (LENGTHP exp 3)
         (eq (car exp) 'LAMBDA)
         (IS-IDENT (cadr exp)))))

(def IS-APPLICATION (lambda (exp) (LENGTHP exp 2)))
```

```
(def IS-COND
  (lambda (exp)
    (and (LENGTHP exp 6)
         (equal (car exp) 'if)
         (equal (caddr exp) 'then)
         (equal (caddddr exp) 'else))))

(def IS-LAMBDA-EXP
  (lambda (exp)
    (or (IS-IDENT exp)
        (IS-BASE exp)
        (IS-LAMBDA-ABSTRACTION exp)
        (IS-APPLICATION exp)
        (IS-COND exp))))

; Syntactic selectors...

(def BINDER-OF (lambda (exp) (cadr exp)))

(def FORM-OF (lambda (exp) (caddr exp)))

(def FUNCTION-OF (lambda (exp) (car exp)))

(def ARGUMENT-OF (lambda (exp) (cadr exp)))

(def IF-PART-OF (lambda (exp) (cadr exp)))

(def THEN-PART-OF (lambda (exp) (cadddr exp)))

(def ELSE-PART-OF (lambda (exp) (caddddr exp)))

; Semantic Functions...

(def BAS
  (lambda (exp)
    (cond ((equal exp 'true) (list 'TAKE t))
          ((equal exp 'false) (list 'TAKE nil))
          ((IS-NUMERAL exp) (list 'TAKE exp))
          ((equal exp 'succ)
```

```
        (list 'ABSTRACT
          (list '!
                (list '! 'TAKEFIRST '(BIND I1))
                (list '!
                      (list '! '(FIND I1) 'APPLY)
                      (list 'APPLYOP 'SUCCESSOR)))))
       ((equal exp 'pred)
        (list 'ABSTRACT
          (list '!
                (list '! 'TAKEFIRST '(BIND I1))
                (list '!
                      (list '! '(FIND I1) 'APPLY)
                      (list 'APPLYOP 'PREDECESSOR)))))
       ((equal exp 'not)
        (list 'ABSTRACT
          (list '!
                (list '! 'TAKEFIRST '(BIND I1))
                (list '!
                      (list '! '(FIND I1) 'APPLY)
                      (list 'APPLYOP 'COMPLEMENT)))))
       ((equal exp 'eq)
        (list 'ABSTRACT
          (list '!
            (list '! 'TAKEFIRST '(BIND I1))
            (list 'ABSTRACT
                  (list '!
                        (list '! 'TAKEFIRST '(BIND I2))
                        (list '!
                              (list '& (list '! '(FIND I1) 'APPLY)
                                       (list '! '(FIND I2) 'APPLY))
                              (list 'APPLYOP 'EQUAL))))))))
       ((equal exp 'lt)
        (list 'ABSTRACT
          (list '!
            (list '! 'TAKEFIRST '(BIND I1))
            (list 'ABSTRACT
                  (list '!
                        (list '! 'TAKEFIRST '(BIND I2))
                        (list '!
```

```
                                (list '& (list '! '(FIND I1) 'APPLY)
                                         (list '! '(FIND I2) 'APPLY))
                                (list 'APPLYOP 'LESS)))))))
          ((equal exp 'sum)
           (list 'ABSTRACT
             (list '!
               (list '! 'TAKEFIRST '(BIND I1))
               (list 'ABSTRACT
                     (list '!
                           (list '! 'TAKEFIRST '(BIND I2))
                           (list '!
                                 (list '& (list '! '(FIND I1) 'APPLY)
                                          (list '! '(FIND I2) 'APPLY))
                                 (list 'APPLYOP 'SUM)))))))
          ((equal exp 'product)
           (list 'ABSTRACT
             (list '!
               (list '! 'TAKEFIRST '(BIND I1))
               (list 'ABSTRACT
                     (list '!
                           (list '! 'TAKEFIRST '(BIND I2))
                           (list '!
                                 (list '& (list '! '(FIND I1) 'APPLY)
                                          (list '! '(FIND I2) 'APPLY))
                                 (list 'APPLYOP 'PRODUCT)))))))
          ((equal exp 'and)
           (list 'ABSTRACT
             (list '!
               (list '! 'TAKEFIRST '(BIND I1))
               (list 'ABSTRACT
                     (list '!
                           (list '! 'TAKEFIRST '(BIND I2))
                           (list '!
                                 (list '& (list '! '(FIND I1) 'APPLY)
                                          (list '! '(FIND I2) 'APPLY))
                                 (list 'APPLYOP 'CONJUNCTION)))))))))

(def EVAL
  (lambda (exp)
```

```
    (cond ((IS-IDENT exp) (list '! (list 'FIND exp) 'APPLY))
          ((IS-BASE exp) (BAS exp))
          ((IS-LAMBDA-ABSTRACTION exp)
           (list 'ABSTRACT
                 (list '!
                       (list '!
                             'TAKEFIRST
                             (list 'BIND (BINDER-OF exp)))
                       (EVAL (FORM-OF exp)))))
          ((IS-APPLICATION exp)
           (list '! (list '& (EVAL (FUNCTION-OF exp))
                             (list 'ABSTRACT (EVAL (ARGUMENT-OF exp))))
                 'APPLY))
          ((IS-COND exp)
           (list 'SELECT
                 (EVAL (IF-PART-OF exp))
                 (EVAL (THEN-PART-OF exp))
                 (EVAL (ELSE-PART-OF exp))))
          (t (error "Not a valid lambda-expression")))))

; EXPAND expands a bound variable to its corresponding expression.
(def EXPAND
  (lambda (var)
    (cond ((numberp var) var)
          ((boundp var) (symeval var))
          (t var))))

; INSERT-BRACKETS brackets a given expression left-associatively.
(def INSERT-BRACKETS
  (lambda (exp)
    (cond ((atom exp) (EXPAND exp))
          ((or (eq (car exp) '!) (eq (car exp) 'LAMBDA))
           (list 'LAMBDA (cadr exp) (INSERT-BRACKETS (cddr exp))))
          ((IS-COND exp)
           (list 'if (INSERT-BRACKETS (IF-PART-OF exp))
                 'then (INSERT-BRACKETS (THEN-PART-OF exp))
                 'else (INSERT-BRACKETS (ELSE-PART-OF exp))))
          ((= (length exp) 1)
           (INSERT-BRACKETS (car exp)))
```

```
          ((= (length exp) 2)
           (list (INSERT-BRACKETS (car exp))
                 (INSERT-BRACKETS (cadr exp))))
          (t
           (INSERT-BRACKETS
            (append (list
                     (list (car exp)
                           (cadr exp)))
                    (cddr exp)))))))

(def START                                  ; starts the interpreter...
  (lambda ()
    (prog (r result)
      loop
        (princ "> ")
        (setq r (lineread))
        (cond ((equal r '(exit)) (go exit)))
        (print (APPLY-ACTION (EVAL (INSERT-BRACKETS r))
                             '(ARID EMPTY ()) ))
        (terpr)
        (go loop)
        exit)))

; The fixed point function Y is defined...
(setq Y (INSERT-BRACKETS '(! h (! x h (x x)) (! x h (x x)))))

; If H is defined as...
(setq H (INSERT-BRACKETS '(! f ! n (if (lt n 1)
                                       then 1
                                       else (product n (f (pred n)))))))

; then the factorial function is defined...
(setq Fact (INSERT-BRACKETS '(Y H)))

(START)
```

BIBLIOGRAPHY

Abdali, S.K.

(1976) *An Abstraction Algorithm for Combinatory Logic*, Journal of Symbolic Logic, 41(1):222-224, March 1976.

Abramson, H.

(1984) *A Prological Definition of HASL - a Purely Functional Language with Unification Based Conditional Binding Expressions*, New Generation Computing, 2(1):3-35, 1984.

Aiello, L. & G. Prini.

(1981) *An Efficient Interpreter for the Lambda-Calculus*, Journal of Computer and System Sciences, 23(3), December 1981.

Allison, L.

(1983) *Programming Denotational Semantics*, Computer Journal, 26(2):164-174, May 1983.

Apt, K.R. & J.W. de Bakker.

(1976) *Exercises in Denotational Semantics*, in Mathematical Foundations of Computer Science 1976, A. Mazurkiewicz (ed.), Springer-Verlag Lecture Notes in Computer Science 45, 1976.

Arbib, M.A. & E.G. Manes.

(1982) *The Pattern-of-Calls Expansion Is the Canonical Fixpoint for Recursive Definitions*, Journal of the ACM, 29(2):577-602, April 1982.

Backus, J.

(1978) *Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM, 21(8):613-641, August 1978.

(1981) *The Algebra of Functional Programs : Function Level Reasoning, Linear Equations and Extended Definitions*, in Formalization of Programming Concepts, J.Diaz & I. Ramos (eds.), Springer-Verlag Lecture Notes in Computer Science 107, 1981.

(1985) *From Function Level Semantics to Program Transformation and Optimization*, in Mathematical Foundations of Software Development, Springer-Verlag Lecture Notes in Computer Science 185, 1985.

de Bakker, J.W.

(1977) *Semantics and the Foundations of Program Proving*, in Information Processing 77, B. Gilchrist (ed.), North-Holland, 1977.

(1980) Mathematical Theory of Program Correctness, Prentice-Hall, 1980.

Barendregt, H.P.

(1984) The Lambda Calculus : Its Syntax and Semantics, North-Holland, 1984.

Bates, J.L. & R.L. Constable.

(1985) *Proofs as Programs*, ACM Transactions on Programming Languages and Systems, 7(1):113-136, January 1985.

Bird, R.S.

(1984) *The Promotion and Accumulation Strategies in Transformational Programming*, ACM Transactions on Programming Languages and Systems, 6(4):487-504, October 1984.

Bjørner, D. & C.B. Jones.

(1982) Formal Specification and Software Development, Prentice-Hall, 1982.

Böhm, C.

(1971) *The CUCH as a Formal and Description Language*, in Formal Language Description Languages, T.B. Steel (ed.), North-Holland, 1971. (The proof to Theorem 3.5.7 is given on pp.195-196. Notation : Böhm uses θ' and θ'' for Y_1 and Y_2, respectively)

(1975) (ed.), λ-Calculus and Computer Science Theory, Springer-Verlag Lecture Notes in Computer Science 37, 1975.

(1980) *An Abstract Approach to (Hereditary) Finite Sequences of Combinators*, in To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism, J.P. Seldin & J.R. Hindley (eds.), Academic Press, 1980.

Boyer, R.S. & J.S. Moore.
(1975) *Proving Theorems About LISP Functions*, Journal of the ACM, 22(1):129-144, January 1975.

de Bruin, A. & W. Böhm.
(1985) *The Denotational Semantics of Dynamic Networks and Processes*, ACM Transactions on Programming Languages and Systems, 7(4):656-679, October 1985.

Burge, W.H.
(1975) Recursive Programming Techniques, Addison-Wesley, 1975.

Campbell, J.A. (ed.)
(1984) Implementations of PROLOG, Ellis Horwood, Chichester, 1984.

Cardelli, L.
(1984) *A Semantics of Multiple Inheritance*, in Semantics of Data Types, G. Kahn et al (eds.), Springer-Verlag Lecture Notes in Computer Science 173, 1984.

Carlsson, M.
(1984) *On Implementing Prolog in Functional Programming*, New Generation Computing, 2(4):347-359, 1984.

Clarke, T.J.W., P.J.S. Gladstone, C.D. Maclean & A.C. Norman.
(1980) *SKIM - the S,K,I Reduction Machine*, pp. 128-135, Proc. of 1980 ACM Lisp Conf., Stanford, California, August 1980.

Curry, H.B., J.R. Hindley & J.P.Seldin.
(1972) Combinatory Logic, Vol. II, North-Holland, 1972.

Darlington, J.

(1982) *Program Transformation*, in Functional Programming and its Applications : An Advanced Course, J. Darlington, P. Henderson & D.A. Turner (eds.), Cambridge University Press, 1982.

Darlington, J., P. Henderson & D.A. Turner (eds.).

(1982) Functional Programming and its Applications : An Advanced Course, Cambridge University Press, 1982.

Dershowitz, N.

(1985) *Program Abstraction and Instantiation*, ACM Transactions on Programming Languages and Systems,7(3):446-477, July 1985.

Donahue, J.E.

(1976) Complementary Definitions of Programming Language Semantics, Springer-Verlag Lecture Notes in Computer Science 42, 1976.

Donahue, J. & A. Demers.

(1985) *Data Types Are Values*, ACM Transactions on Programming Languages and Systems, 7(3):426-445, July 1985.

Dunlop, D.D. & V.R. Basili.

(1985) *Generalizing Specifications for Uniformly Implemented Loops*, ACM Transactions on Programming Languages and Systems, 7(1):137-158, January 1985.

van Emden, M.H. & R.A. Kowalski.

(1976) *The Semantics of Predicate Logic as a Programming Language*, Journal of the ACM, 23(4):733-742, October 1976.

Foderaro, J.K. et al.

(1983) The FRANZ LISP Manual, University of California, Berkeley, 1983.

Georgeff, M.

(1984) *Transformations and Reduction Strategies for Typed Lambda Expressions*, ACM Transactions on Programming Languages and Systems, 6(4):603-631, October 1984.

Gordon, M.J.C.
(1979) The Denotational Description of Programming Languages, Springer, 1979.

Gordon, M.J., A.J. Milner & C.P. Wadsworth.
(1979) Edinburgh LCF, Springer-Verlag Lecture Notes in Computer Science 78, 1979.

Henderson, P.
(1980) Functional Programming : Application and Implementation, Prentice-Hall, 1980.

Hoare, C.A.R.
(1972) *Notes on Data Structuring*, in Structured Programming, O.-J. Dahl, E.W. Dijkstra & C.A.R. Hoare, Academic Press, 1972.

Hughes, R.J.M.
(1982) *Super-Combinators : A New Implementation Method for Applicative Languages*, pp. 1-10, Proc. of 1982 ACM Conf. on Lisp and Functional Programming, Pittsburgh, PA, August 1982.

Ida, T. & J. Tanaka.
(1983) *Functional Programming with Streams*, in Information Processing 83, R.E.A. Mason (ed.), North-Holland, 1983.
(1984) *Functional Programming with Streams - Part II*, New Generation Computing, 2(3):261-275, 1984.

Jones, N.D. & D.A. Schmidt.
(1980) *Compiler Generation from Denotational Semantics*, Semantics Directed Compiler Generation, Springer-Verlag Lecture Notes in Computer Science 94, 1980.

Kahn, K.M. & M. Carlsson.
(1984) *How to Implement Prolog on a LISP Machine*, in Implementations of PROLOG, J.A. Campbell (ed.) Ellis Horwood, Chichester, 1984.

Kieburtz, R.B.

(1985) The G-Machine : A Fast, Graph-Reduction Evaluator, Oregon Graduate Center Technical Report CS/E-85-002, Beaverton, Oregon, January 1985.

Kowalski, R.

(1983) *Logic Programming*, in Information Processing 83, R.E.A. Mason (ed.), North-Holland, 1982.

Larsen, K.G. & G. Winskel.

(1984) *Using Information Systems to Solve Recursive Domain Equations Effectively*, in Semantics of Data Types, G. Kahn et al (eds.), Springer-Verlag Lecture Notes in Computer Science 173, 1984.

Lassez, J.-L. & M.J. Maher.

(1985) *Optimal Fixedpoints of Logic Programs*, Theoretical Computer Science, 39:15-25, 1985.

Linger, R.C., H.D. Mills & B.I. Witt.

(1977) Structured Programming : Theory and Practice, Addison-Wesley, 1977.

McCarthy, J.

(1963) *A Basis for a Mathematical Theory of Computation*, in Computer Programming and Formal Systems, D. Braffort & D. Hirschberg (eds.), North-Holland, 1963.

McGettrick, A.D.

(1980) The Definition of Programming Languages, Cambridge University Press, 1980.

(1982) Program Verification Using Ada, Cambridge University Press, 1982.

Manna, Z.

(1974) Mathematical Theory of Computation, McGraw-Hill, 1974.

(1980) Lectures on the Logic of Computer Programming, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1980.

Manna, Z. & A. Shamir

(1976) *The Theoretical Aspects of the Optimal Fixed-Point*, SIAM Journal on Computing, 5(3):414-426, September 1976.

Maurer, P.M.

(1983) *The Use of Combinators in Translating a Purely Functional Language to Low-Level Data-Flow Graphs*, Computer Languages, 8(1):27-45, 1983.

Meyer, A.R.

(1982) *What is a Model of the Lambda Calculus?*, Information and Control, 52(1):87-122, January 1982.

(1983) *Understanding ALGOL : The View of a Recent Convert to Denotational Semantics*, in Information Processing 83, R.E.A. Mason (ed.), North-Holland, 1983.

Milne, R. & C. Strachey.

(1976) A Theory of Programming Language Semantics, Chapman & Hall, London, 1976.

Milner, R.

(1978) *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences, 17(3):348-375, December 1978.

Mosses, P.

(1976) *Compiler Generation using Denotational Semantics*, in Mathematical Foundations of Computer Science 1976, A. Mazurkiewicz (ed.), Springer-Verlag Lecture Notes in Computer Science 45, 1976.

(1980) *A Constructive Approach to Compiler Correctness*, in Semantics-Directed Compiler Generation, N.D. Jones (ed.), Springer-Verlag Lecture Notes in Computer Science 94, 1980.

(1981) *A Semantic Algebra for Binding Constructs*, in Formalization of Programming Concepts, J. Diaz & I. Ramos (eds.), Springer-Verlag Lecture Notes in Computer Science 107, 1981.

(1984) *A Basic Abstract Semantic Algebra*, in Semantics of Data Types, G. Kahn, D.B. MacQueen & G. Plotkin (eds.), Springer-Verlag Lecture Notes in Computer Science 173, 1984.

Nielson, F.
(1985) *Program Transformations in a Denotational Setting*, ACM Transactions on Programming Languages and Systems, 7(3):359-379, July 1985.

Noonan, R.E. & D.J.Panton.
(1974) *Structured Recursive Programming*, in Programming Symposium, B. Robinet (ed.) Springer-Verlag Lecture Notes in Computer Science 19, 1974.

Pingali, K. & Arvind.
(1985) *Efficient Demand-Driven Evaluation. Part I*, ACM Transactions on Programming Languages and Systems, 7(2):311-333, April 1985.

Reynolds, J.C.
(1974) *Towards a Theory of Type Structure*, in Programming Symposium, B. Robinet (ed.), Springer-Verlag Lecture Notes in Computer Science 19, 1974.
(1981) The Craft of Programming, Prentice-Hall, 1981.
(1983) *Types, Abstraction and Parametric Polymorphism*, in Information Processing 83, R.E.A. Mason (ed.), North-Holland, 1983.

Richards, H.
(1985) *An Overview of the Burroughs NORMA*, 1985 Aspenas Workshop on Functional Language Implementation.

Scherlis, W.L. & D.S. Scott.
(1983) *First Steps Towards Inferential Programming*, in Information Processing 83, R.E.A. Mason (ed.), North-Holland, 1983.

Schmidt, D.A.
(1985) *Detecting Global Variables in Denotational Specifications*, ACM Transactions on Programming Languages and Systems, 7(2):299-310, April 1985.

Scott, D.S.

(1976) *Data Types as Lattices*, Siam Journal on Computing, 5(3):523-587, September 1976.

(1982) *Domains for Denotational Semantics*, in Automata, Languages and Programming, M. Nielson & E.M. Schmidt (eds.), Springer-Verlag Lecture Notes in Computer Science 140, 1982.

Stenlund, S.

(1972) Combinators, λ-terms and Proof Theory, D. Reidel, Dordrecht, Holland, 1972.

Stoy, J.E.

(1977) Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge Massachussetts, 1977.

(1982) *Some Mathematical Aspects of Functional Programming*, in Functional Programming and its Applications : An Advanced Course, J. Darlington, P. Henderson & D.A. Turner (eds.), Cambridge University Press, 1982.

Strachey, C.

(1966) *Towards a Formal Semantics*, in Formal Language Description Languages for Computer Programming, T.B. Steel (ed.), North-Holland, 1966.

Suzuki, N.

(1980) Automatic Verification of Programs with Complex Data Structures, Garland, New York, 1980.

Tennent, R.D.

(1976) *The Denotational Semantics of Programming Languages*, Communications of the ACM, 19(8):437-453, August 1976.

(1981) Principles of Programming Languages, Prentice-Hall, 1981.

Trakhtenbrot, B.A., J.Y. Halpern & A.R. Meyer.

(1984) *From Denotational to Operational and Axiomatic Semantics for ALGOL-like languages : An Overview*, in Logics of Programs, E. Clarke, & D. Kozen (eds.), Springer-Verlag Lecture Notes in Computer Science 164, 1984.

Turner, D.A.

(1979a) *A New Implementation Technique for Applicative Languages*, Software - Practice and Experience, 9:31-49, 1979.

(1979b) *Another Algorithm for Bracket Abstraction*, Journal of Symbolic Logic, 44(2):267-270, June 1979.

(1982) *Recursion Equations as a Programming Language*, in Functional Programming and its Applications : An Advanced Course, J. Darlington, P. Henderson & D.A. Turner (eds.), Cambridge University Press, 1982.

Vegdahl, S.R.

(1984) *A Survey of Proposed Architectures for the Execution of Functional Languages*, IEEE Transactions on Computers, C-33(12):1050-1071, December 1984.

Vickers, T.N.

(1986) *Quokka : A Translator Generator Using Denotational Semantics*, The Australian Computer Journal, 18(1):9-17, February 1986.

Voda, P.J.

(1985) *A View of Programming Languages as a Symbiosis of Meaning and Computations*, New Generation Computing, 3:71-100, 1985.

Wadsworth, C.P.

(1976) *The Relation Between Computational and Denotational Properties for Scott's D_∞-Models of the Lambda-Calculus*, SIAM Journal on Computing, 5(3):488-521, September 1976.

(1978) *Approximate Reduction and Lambda Calculus Models*, SIAM Journal on Computing, 7(3):337-356, August 1978.

(1980) *Some Unusual λ-Calculus Numeral Systems*, in To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism, J.P. Seldin & J.R. Hindley (eds.), Academic Press, 1980.

Watt, D.A.

(1986) *Executable Semantic Descriptions*, Software : Practice and Experience, 16(1):13-43, January 1986.

Williams, J.H.
(1982) *Notes on the FP Style of Functional Programming*, in Functional Programming and its Applications : An Advanced Course, J. Darlington, P. Henderson & D.A. Turner (eds.), Cambridge University Press, 1982.

Wise, D.S.
(1982) *Interpreters for Functional Programming*, in Functional Programming and its Applications : An Advanced Course, J. Darlington, P. Henderson & D.A. Turner (eds.), Cambridge University Press, 1982.

Wilensky, R.
(1984) LISPcraft, W.W.Norton & Company, New York, 1984.