R9063 Copy 1

FUNCTIONAL PROGRAMMING WITH SIDE-EFFECTS

by

Mark B. Josephs

Technical Monograph PRG-55

June 1986
(published October 1986)

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

A thesis submitted for the degree of
D.Phil at the University of Oxford
June 1986

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP
8-11 Keble Road, Oxford OX1 3QD, England

Technical Monographs to May 1986

PRG-2	*Outline of a Mathematical Theory of Computation* by Dana Scott. November 1970, 24 p., £0.50
PRG-3	*The Lattice of Flow Diagrams* by Dana Scott. November 1970, 57 p., £1.00
PRG-5	*Data Types as Lattices* by Dana Scott. September 1976, 65 p., £2.00
PRG-6	*Toward a Mathematical Semantics for Computer Languages* by Dana Scott and Christopher Strachey. August 1971, 43 p., £0.60
PRG-9	*The Text of OSPub* by Christopher Strachey and Joseph Stoy. July 1972, 2v. 126, 151 p. £3.50
PRG-10	*The Varieties of Programming Language* by Christopher Strachey. March 1973, 20 p., £0.50
PRG-17	*Report on the Programming Notation 3R* by Andrew P. Black. August 1980, 58 p., £2.30
PRG-18	*The Specification of Abstract Mappings and their Implementation as B+ Trees* by Elizabeth Fielding. September 1980, 74 p. + Appendix, £1.30
PRG-20	*Partial Correctness of Communicating Processes and Protocols* by Zhou Chao Chen and C.A.R. Hoare. May 1981, 23 p., £1.75
PRG-22	*A Model for Communicating Sequential Processes* by C.A.R. Hoare. June 1981, 26 p., £1.30
PRG-23	*A Calculus of Total Correctness for Communicating Processes* by C.A.R. Hoare. April 1981, 31 p., £1.75
PRG-26	*The Consistency of the Calculus of Total Correctness for Communicating Sequential Processes* by Zhou Chao Chen. February 1982, 38 p., £1.80
PRG-29	*Specifications, Programs and Implementations* by C.A.R. Hoare. June 1982, 29 p., £1.75
PRG-32	*The Lispkit Manual* by Peter Henderson, Geraint A. Jones and Simon B. Jones. 1983, 2v., 127, 136 p., £4.00 for both volumes
PRG-34	*Abstract Machine Support for Purely Functional Operating Systems* by Simon B. Jones. August 1983, 53 p. + Appendix, £1.75

PRG-36	*The Formal Specification of a Conference Organising System* by Tim Clement. August 1983, 52 p. + Appendix. £1.75
PRG-37	*Specification-Oriented Semantics for Communicating Processes* by E.R. Olderog and C.A.R. Hoare. February 1984, 81 p., £1.50
PRG-38	*Making Nets Abstract and Structured and Nets and their Relation to CSP* by Ludwik Czaja. January/June 1984, 23, 26 p., £1.30
PRG-39	*μFP - An Algebraic VLSI Design Language* by Mary Sheeran. Ph.D. thesis November 1983, 139 p., £2.50
PRG-40	*The Design and Implementation of Programming Languages* by John Hughes . Ph.D. thesis July 1983, 130 p. + Appendix, £2 50
PRG-42	*A Range of Operating Systems Written in a Purely Functional Style* by Simon B. Jones. February 1985, 44 p., £1.30
PRG-44	*The Weakest Prespecification* by C.A.R. Hoare and He Jifeng. June 1985, 60 p., £0.85
PRG-45	*Laws of Programming - A Tutorial Paper* by C.A.R. Hoare, He Jifeng, I.J. Hayes, C.C. Morgan, J.W. Sanders,. I.H. Sørensen, J.M. Spivey, B.A. Sufrin, A.W. Roscoe. May 1985, 43 p., £2.35
PRG-46	*Specification Case Studies* by Ian Hayes. July 1985, 68 p., £2.50
PRG-47	*Specifying the CICS Application Programmer's Interface* by Ian Hayes. July 1985, 82 p., £3.10
PRG-48	*CAVIAR: A Case Study in Specification* by Bill Flinn and Ib Holm Sørensen. July 1985, 46 p., £2.00
PRG-49	*Specification Directed Module Testing* by Ian Hayes. July 1985, 30 p., £0.90
PRG-50	*The Distributed Computing Software Project* by Roger Gimson and Carroll Morgan. July 1985, 85 p., £4.00
PRG-51	*JSD Expressed in CSP* by K.T. Sridhar and C.A.R. Hoare. July 1985, 40 p., £1.45
PRG-52	*Algebraic Specification and Proof of Properties of Communicating Sequential Processes* by C.A.R. Hoare and He Jifeng. November 1985, 72 p., £0.90
PRG-53	*The Laws of Occam Programming* by A.W. Roscoe and C.A.R. Hoare. February 1986, 86 p., £2 50
PRG-54	*Exploiting Parallelism in the Graphics Pipeline* by Theoharis A. Theoharis. June 1986, 101 p., £2.50

To my parents

FUNCTIONAL PROGRAMMING WITH SIDE-EFFECTS

Mark Brian Josephs,
Wolfson College, Oxford

D.Phil. Thesis,
Trinity Term, 1986

ABSTRACT

In this thesis functional and logic programming languages are combined into a new declarative language. This allows for *lazy single-assignment* to logical variables in an otherwise purely functional language. Efficient solutions to various programming problems are developed. It is also shown that these *programs with side-effects* can be derived by means of transformational programming. On implementations that support parallelism, further interesting possibilities arise, which are discussed briefly.

The formal semantics of this extension to functional programming is investigated. Both denotational and axiomatic semantic descriptions of the language are presented. Programming with side-effects is particularly attractive, because it can be supported by graph reduction based implementations of functional programming languages. Details of a prototype implementation are given.

CONTENTS

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1 AN EXTENSION TO FUNCTIONAL PROGRAMMING

In recent years there has been a growing appreciation of the declarative style of programming, in which we are encouraged to look upon programs as mathematical objects. Ideally, we can understand a program without having to form a mental picture of some kind of machine executing a sequence of instructions. Furthermore, we might hope to be able to *prove*, more easily than hitherto, that a program meets its specification (typically expressed in some rich mathematical notation [25]).

Declarative programming languages can be divided into two classes, the functional and logic styles. A functional program consists of a set of function definitions, and computation involves the evaluation of an expression by using these definitions as rewrite rules. For example, the functional program

```
factorial : Int → Int
factorial n = 1                    if n = 0
              n * factorial (n-1) otherwise
```

can be used to calculate (factorial 1). The evaluation process proceeds through the sequence of reductions (rewrites)

```
        factorial 1 ⟶ 1 * factorial (1-1)
                    ⟶ 1 * factorial 0
                    ⟶ 1 * 1
                    ⟶ 1
```

Logic programming languages are usually based on the Horn clause subset of predicate logic [34] and a unification algorithm [46] for matching atomic formulæ. It is possible to give a declarative reading to logic programs. However, it is also necessary for the programmer to understand the procedural interpretation of the language. Consider once again the factorial example: the clauses

```
factorial(0,1).
factorial(n,x) ⇐ plus(m,1,n), factorial(m,y), times(n,y,x)
```

can be read as

```
          factorial 0 is 1

         For all integers m, n, x, y,
           factorial n is x
              if m plus 1 is n,
                 factorial m is y,
                 and n times y is x
```

The following represents a top-down solution [36] to the problem of determining the value z of factorial 1. It shows how the output z = 1 can be computed.

```
        ∘ ⇐ factorial(1,z)

        |       unifying factorial(1,z) with factorial(n,x)

        ∘ ⇐ plus(m,1,1), factorial(m,y), times(1,y,z)

 m = 0  |

        ∘ ⇐ factorial(0,y), times(1,y,z)

 y = 1  |       unifying factorial(0,y) with factorial(0,1)

        ∘ ⇐ times(1,1,z)

 z = 1  |

        ∘ □
```

'Values produced by the execution of a logic program can contain [logical] variables, in contrast to the totally ground expressions manipulated by functional programs' [14]. Darlington suggests that 'this gives logic languages extra power and enables many elegant solutions to be developed'.

This thesis is concerned with a style of programming, which has been dubbed ***functional programming with side-effects***. It is made possible by combining functional and logic programing languages into a new declarative language. Logical variables are made available for use, in a restricted manner, in functional programs. However, unification is not used as an evaluation mechanism; furthermore, unlike Prolog [12], there is no backtracking on the binding of variables.

The language can be regarded as the extension of an otherwise purely functional language to include a novel feature, a ***lazy single-assignment*** construct. Now, the lack of a notion of assignment (and hence side-effects) is said to be a fundamental property of the class of functional programming languages, so this idea seems to be particularly controversial. For example, Peyton-Jones [42] states, in support of functional programming

' (i) The absence of side-effects leads to clean and simple semantics, which makes programs easier to write, and easier to reason about than conventional languages.

(ii) Distinct subexpressions of a program can safely be evaluated concurrently, since the absence of side-effects ensures that the subexpressions are genuinely independent. This opens up possibilities for the exploitation of parallel hardware. '

In defence of functional programming with side-effects, evidence is provided that

(1) It is not too difficult to develop or understand such programs;

(2) The programs are suitable for parallel execution;

(3) The proposed extension permits the expression of some interesting algorithms that would otherwise be unavailable to the functional programmer;

(4) It is sometimes possible to achieve a gain in efficiency over purely functional programs because, with the assignment construct, we may

be in a position to take advantage of the order in which expressions get evaluated. (It should be noted that we have in mind a particular model of computation, demand-driven graph reduction [16,28,54]. Graph reduction machines provide one method by which functional programming languages can be implemented, and the new language has been specifically designed to fully exploit this graph reduction method of computation.)

Programs with side-effects do not always behave quite as one might expect. There is the possibility that evaluation will deadlock, that is, come to a premature halt, instead of returning the value of an expression. The programmer requires a sound methodology that will facilitate development of programs in the language: for example, the work on weakest preconditions [17,20] for deriving programs in an imperative language. Transformational programming [7,13,37,47] is known to be a useful method for obtaining efficient functional programs from clear, but inefficient, functional programs that act as specifications. Likewise, transforming purely functional programs into functional programs with side-effects has proven to be reasonably straightforward.

Another challenge we address is 'to be able to understand the language without having to understand the implementation' [26]. For example, we would like to understand Prolog without having to think about backtracking but, unfortunately, we would then have no way of knowing when to use the *cut*. Functional programs with side-effects can be written with little or no knowledge of graph reduction. A formal system based on inference rules has been developed for reasoning about such programs. This formal system gives an axiomatic semantics for the language.

1.2 ORDER OF EVALUATION

This section contains a brief discussion of issues relating to the order of evaluation of expressions during the execution of purely functional programs. With this background material it is hoped that the reader will be in a better position to fully appreciate functional programming with side-effects, which is described in the next chapter.

We begin with a short account of the lazy evaluation strategy [19,22] for executing functional programs, used in such languages as SASL [53], KRC [56] and Lispkit [21]. Under lazy evaluation, (i) the arguments to function calls are evaluated at most once, and then only if their values are required, giving call-by-need [60] as opposed to call-by-name or call-by-value semantics, and (ii) this lazy approach also extends to the components of data structures. For example, for (cons E_1 E_2) evaluation of the head, E_1, and the tail, E_2, is delayed until they are needed. It is, of course, one thing to appreciate this, but quite another to realise how it can be implemented. Lazy evaluation is desirable because it can be exploited in a modular style of programming. For example, the first 20 prime numbers are computed by the expression (take 20 primes), where primes has been defined to be the infinite list 2, 3, 5, 7, ... of prime numbers and take is a function such that (take n xs) returns the first n elements of the list xs.

Another possible evaluation strategy allows for concurrent evaluation of expressions. Superficially, it might be thought that lazy and parallel evaluation strategies are irreconcilable. However, we have only to realise that, if a function is strict, then it is perfectly consistent with an otherwise lazy approach for the evaluation of a function call to proceed concurrently with the evaluation of its arguments. Thus, for example, in evaluating (E_1 + E_2) we might want E_1 and E_2 to be evaluated in parallel. Two disadvantages with parallel evaluation are that it is difficult to implement efficiently, even on distributed computing architectures, and that unnecessary (parallel) evaluation of expressions can waste resources, possibly producing a non-terminating computation.

An important observation is that the order of evaluation of expressions is something that is not usually taken into account when first writing a functional program. Our primary concern is to ensure that the functions are defined correctly: in other words, we concentrate on the declarative reading of the program, or equivalently its logic, under Kowalski's definition ***Algorithm = logic + control*** [35]. However, if we are to analyse the space and time complexities of functional programs (with or without parallelism), then it becomes necessary to consider the order in which expressions are evaluated.

For example, in the KRC prelude [56] the minimum element of a list xs is defined by head(sort xs), where sort is insertion sort. Because of lazy evaluation the minimum element is determined in linear time, even though it takes quadratic time to completely sort a list by this method.

As a second example: we ought to be able to determine the length of a list in constant space, but the program

```
length : List * → Int
length = length' 0

length' : Int → List * → Int
length' n  nil          =   n
length' n (cons x xs) =   length' (n+1) xs
```

takes space proportional to the length of the list, because, under lazy evaluation, the additions are nested and not calculated until the entire list has been parsed [29]. Thus:

```
length [3,6] ⟶ length' 0 [3,6]
             ⟶ length' (0+1) [6]
             ⟶ length' ((0+1)+1) nil
             ⟶ ((0+1)+1))
             ⟶ (1+1)
             ⟶ 2
```

We conclude this section by noting that it is possible to alter the order of evaluation of expressions by annotating a program with control functions [8,9,29,30,48]. Hughes [29,30] has suggested a function that has the effect of synchronizing the execution of parallel computations. When misused this can cause evaluation to deadlock. As was mentioned earlier, this undesirable behaviour can occur in functional programs with side-effects; we shall be concerned with ensuring that our programs are free from deadlock.

1.3 ORGANIZATION OF THE THESIS

The thesis has been organized as follows:-

In Chapter 2 a functional language with side-effects is introduced. The syntax and semantics of the language are described informally, and many programming examples are given.

Chapter 3 is devoted to an explanation of a formal method of program development based on transformational programming. Many of the examples of Chapter 2 are reworked as transformation problems. It is shown that side-effects can be used as an alternative to tupling [40,41] or continuations [51,61].

In Chapter 4 a method of computation based on graph reduction is introduced and a denotational semantics for the language is presented; this does not cover parallel evaluation. The semantics provides some justification for the transformation rules of Chapter 3.

Chapter 5 describes an axiomatic semantics, which extracts from the process of graph reduction those properties essential for reasoning about the language. The axiomatic semantics is also useful for understanding parallel and synchronized execution of programs.

A summary of the achievements embodied in the thesis and a discussion of related work appear in Chapter 6. Finally, details of an actual implementation of a language with side-effects are to be found in the appendix, along with the results of some experimental comparisons.

CHAPTER 2

PROGRAMMING WITH SIDE-EFFECTS

2.1 INTRODUCTION

The essence of programming with side-effects can be stated as follows:-

Unlike a conventional functional language, an applicative expression may contain occurrences of ***uninstantiated*** variables. During the process of evaluating such expressions, it is possible for these variables to become bound to further expressions. An attempt to evaluate an uninstantiated variable will ***suspend*** until the variable gets instantiated, that is, bound to some expression. Thus, it is possible for a state of deadlock to arise during program execution: this occurs when all demands for the values of expressions have been suspended.

This chapter begins with a brief and informal account of the syntax and semantics of a language which supports programming with side-effects. The remainder of the chapter is then devoted to the presentation of a selection of programming examples in this language. It is hoped that these examples demonstrate that the new style of programming made possible by side-effects is both useful and not too difficult to understand. To be sure, an additional burden has been placed on the programmer, who now has to take care to avoid the possibility of deadlock during program execution. However, in the next chapter it is shown that a systematic method of ***program transformation*** can be adopted for programming with side-effects. In this method side-effects are introduced in a controlled and safe manner.

We shall not concern ourselves at this point with how the language can be implemented, other than to note that: functional languages can be implemented on graph reduction machines, and side-effects can be accommodated without modifying such machines. (The possibility of binding variables by graph reduction was recognised in [15], namely, '***The logic based programming languages*** are supported by a facility that permits a packet to be treated as a variable by allowing a reduction to have the side-effect of assigning new contents to any argument packet.') Further explanation can be found in Chapter 4 and in the appendix.

2.2 THE SYNTAX OF THE EXTENDED LANGUAGE

In this section we give an informal description of the syntax of a functional language that incorporates the extension.

An expression is evaluated according to a program that consists of a set of function definitions, as in KRC [56]. In fact a syntax has been adopted that is based on Orwell [58] and Miranda [57] which include a polymorphic type system [39]. Function definitions can be viewed as recursion equations or rewrite rules, and take the form

```
<function name> : <type>
<function name> <pattern>* = <expression>
```

Here, a pattern can be formed from variables and basic values (eg. numbers) by means of certain constructors (eg. cons). Any variables appearing in patterns have the right hand side of the equation as their scope. To improve readability, variables are marked as var parameters if they are to be assigned to as a side-effect of a call to the function. In addition to these user-declared functions, we will assume the existence of some primitive functions, for instance, the common arithmetic operators.

The category of expressions is defined to include variables, basic values, function names and function applications, as well as the forms

```
                <side-effect> ; <expression>
                <declaration> ; <expression>
```

where

```
                <side-effect> ::= assign <variable> = <expression>
                <declaration> ::= var <variable>
```

Note that for (var x ; E), the scope of x is E. Of course, variables can only appear within their scope. For example,

```
foo x (cons y (var ys)) = var zs ; assign ys = tail zs ; f x zs
```

is a well-formed definition provided f is the name of some function.

Function application is left associative, that is, we write $(E_1\ E_2\ E_3)$ to mean $((E_1\ E_2)\ E_3)$. Other conventions that have been adopted include (i) ++ as an infix form of append, and (ii) a notation based on set abstraction, so that, for example, [y←xs|y<x] denotes the list of numbers drawn from xs that are less than x.

2.3 SEMANTICS

The language is based on lazy evaluation, as is the case for many functional languages. Furthermore, although its var and assign constructs have their counterparts in imperative languages, for example, Pascal, there are some subtle differences between assign and a standard assignment statement. We can give an informal semantics to this extension as follows:-

A. Evaluation of var x ; E

1. A new location in the store is allocated to x and marked as unset. (We call x an *uninstantiated* variable.)
2. E is evaluated.

B. Evaluation of assign $x = E_1$; E_2

1. If x is not uninstantiated, an error is reported for improper assignment. Otherwise:
2. E_1 is stored *in unevaluated form* at x, that is, x is bound to the *expression* E_1.
3. E_2 is evaluated.

Thus, assign is a lazy single-assignment construct.

C. Evaluation of an uninstantiated variable is suspended until an expression is assigned to it. Evaluation then continues with that expression.

Notes: 1) (var x ; var y ; E) = (var y ; var x ; E)

since distinct new locations are allocated to x and y in each case. Hence, we can abbreviate such expressions as simply

(var x, y ; E)

2) (<u>assign</u> $x = E_1$; <u>assign</u> $y = E_2$; E)
= (<u>assign</u> $y = E_2$; <u>assign</u> $x = E_1$; E)

since the expressions E_1 and E_2 are stored in unevaluated form. Again we adopt the abbreviated notation

(<u>assign</u> $x = E_1$, $y = E_2$; E)

3) Some functional languages have a construct such as <u>where</u> for making local recursive definitions. Then

(E <u>where</u> $x_1 = E_1$, ... , $x_n = E_n$)
= (<u>var</u> x_1, ... , x_n ; <u>assign</u> $x_1 = E_1$, ... , $x_n = E_n$; E)

since in both cases all occurrences of x_i in E, E_1, ... , E_n refer to E_i. Further explanation can be found in [31], which describes an efficient implementation of <u>where</u>.

4) Order of evaluation of expressions is important. For example, addition only remains a commutative operator if it evaluates its operands in parallel.

To see this, suppose instead that addition evaluates its left operand before its right. We can compare the expressions

(add (<u>assign</u> x = 3 ; 2) (sub 4 x))

and

(add (sub 4 x) (<u>assign</u> x = 3 ; 2))

where x is an uninstantiated variable, in the following way.

Evaluation of (<u>assign</u> x = 3 ; 2) binds x to 3 and returns 2. Evaluation of (sub 4 x) returns 1, if x = 3, but suspends if x is uninstantiated. Now, because the right operand of an addition only gets evaluated once the left operand has returned its value, it can be seen that

```
(add (assign x = 3 ; 2) (sub 4 x))
```

evaluates to 3, whereas evaluation of

```
(add (sub 4 x) (assign x = 3 ; 2))
```

deadlocks.

We therefore have to take order of evaluation into account so as to guarantee termination.

2.4 EXAMPLES OF PROGRAMMING WITH SIDE-EFFECTS

The first two problems are taken from [4]. Bird uses transformation techniques to develop efficient programs from inefficient programs that act as specifications. ***Tupling*** is used to improve efficiency by avoiding repeated traversal of a data structure, but as a result the programs lose their clarity. It is claimed that our solutions to the problems are no less efficient and, given some familiarity with the extended language, are not too difficult to develop.

For all problems for which a formal specification, in the form of a functional program, has been given, the reader is referred to Chapter 3, where the solutions are systematically derived by means of transformational programming.

EXAMPLE 1

Consider the data type of binary trees defined by

```
type Tree = tip Int | fork Tree Tree
```

We wish to change a given tree into a second tree identical in shape to the first. However, each tip value should be replaced by the minimum tip value of the tree.

```
transform : Tree → Tree
transform t = replace t (tmin t)

replace : Tree → Int → Tree
replace (tip n)    m =  tip m
replace (fork L R) m =  fork (replace L m) (replace R m)

tmin : Tree → Int
tmin (tip n)     =  n
tmin (fork L R)  =  (tmin L) MIN (tmin R)
```

METHOD

We introduce a local variable v to hold the minimum tip value, (tmin t), for a given tree t. Thus, for t = (tip n), we wish to assign v = n, and for t = (fork L R), we wish to assign v = (y MIN z), where we have introduced y and z to stand for (tmin L) and (tmin R), respectively.

Suppose also that m stands for the minimum tip value of the tree to be transformed. Of course, for this tree v = m. We only have to realize one more thing: namely, a tip is replaced by a tip, and a fork by a fork, and we are ready to formulate our solution.

SOLUTION 1

```
transform : Tree → Tree
transform t = var m ; replace t m m

replace : Tree → Int → Int → Tree
replace (tip n)    m (var v) = assign v = n ; tip m
replace (fork L R) m (var v) = var y, z ; assign v = y MIN z ;
                               fork (replace L m y)
                                    (replace R m z)
```

A declarative or logical reading can be given to this program as follows:

$\forall x, t{:}\mathrm{Tree}.\ x = \mathrm{transform}\ t \Leftrightarrow \exists m{:}\mathrm{Int}.\ x = \mathrm{replace}\ t\ m\ m$

$\forall x{:}\mathrm{Tree};\ n, m, v{:}\mathrm{Int}.\ x = \mathrm{replace}\ (\mathrm{tip}\ n)\ m\ v \Leftrightarrow v = n \wedge x = \mathrm{tip}\ m$

$$\begin{aligned}&\forall x, L, R{:}\mathrm{Tree};\ m, v{:}\mathrm{Int}.\\ &x = \mathrm{replace}\ (\mathrm{fork}\ L\ R)\ m\ v \Leftrightarrow \exists y, z{:}\mathrm{Int}.\ \ v = y\ \mathrm{MIN}\ z\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ x = \mathrm{fork}\ (\mathrm{replace}\ L\ m\ y)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{replace}\ R\ m\ z)\end{aligned}$$

From this we can infer that if transform is applied to a given tree and evaluates to some new tree, then this resulting tree will indeed meet the specification. What this logical reading does not tell us is how much, if any, of the transformed tree can be computed. For this *operational* aspect of the behaviour of the program, an understanding of demand-driven evaluation is required.

From an alternative point of view, the declarative reading reflects a parallel evaluation strategy, by which evaluation of an expression (whether being assigned to a variable, or occurring in a data structure or as an argument in a function call) proceeds eagerly rather than lazily. To reason about our language we require a more elaborate formal system into which some concept of delayed evaluation has been built: see the

Returning to Solution 1, we can deduce that all the forks and tips can be computed in response to demands. However, a demand for a tip value will be suspended until all the tips have been computed. This means that our program is unsuitable - it would deadlock - if, say, we wanted to determine the minimum tip value by sending demands only along some selected branch of the tree. For example,

```
        findval (transform (fork (tip 2) (tip 1)))
```

will deadlock rather than evaluate to 1, for `findval` defined as

```
findval : Tree → Int
findval (tip n)     = n
findval (fork L R) = findval L
```

Thus, our program has a different operational behaviour from that given by Bird, which is the following:

```
transform : Tree → Tree
transform t = t'
              where (t', m) = repmin t m

repmin : Tree → Int → (Tree x Int)
repmin (tip n)     m = (tip m, n)
repmin (fork L R) m = (fork L' R', y MIN z)
                      where (L', y) = repmin L m
                            (R', z) = repmin R m
```

Bird's program, which is deadlock-free, was synthesized from the definition `repmin t m = (replace t m, tmin t)`

Fortunately, a minor modification to Solution 1 is all that is required to make it free from deadlock: the definition of `replace` is annotated with the `val` combinator [29] (the use of which makes functions ***strict***), so that

```
replace (fork L R) m (var v) = var y, z ; assign v = y MIN z ;
                               val (val fork (replace L m y))
                                             (replace R m z)
```

The (complete) transformed tree is then constructed in reponse to an initial demand.

Note that for (val E_1 E_2), E_2 gets evaluated before E_1 is (evaluated and) applied to it: this corresponds to call-by-value. The simple expression (E_1 E_2) gives call-by-need, that is, E_2 gets evaluated only when first required. So, by annotating the program with val, the minimum tip value is computed in response to a demand for *any* tip's new value. Our program is no less efficient than Bird's and is now deadlock-free: an experimental comparison of the programs appears in the appendix.

On implementations that support parallelism, it would be possible to use Hughes' par combinator [29,30] in place of val: evaluation of (par E_1 E_2) involves the concurrent reduction of E_2 and (E_1 E_2), its value being that of (E_1 E_2). (***Technical note*** : we assume that expressions cannot be garbage collected while they are being evaluated; although the value of the expression may not itself be required, evaluation of the expression may have a vital side-effect.)

EXAMPLE 2

This tree transformation problem is similar to Example 1. Again the transformed tree is to have the same shape as the original tree, but this time the original tip values must be sorted into increasing order and then allocated to the new tips from left to right.

```
transform : Tree → Tree
transform t = replace t (sort (tips t))

replace : Tree → List Int → Tree
replace (tip n)    us = tip (head us)
replace (fork L R) us = fork (replace L us)
                             (replace R (drop (size L) us))
```

```
tips : Tree → List Int
tips (tip n)    = [n]
tips (fork L R) = tips L ++ tips R


size : Tree → Int
size (tip n)    = 1
size (fork L R) = size L + size R


drop : Int → List * → List *
drop  0     xs          = xs
drop (n+1) (cons x xs) = drop n xs
```

(Note that this specification is slightly simpler than that given by Bird.)

METHOD

Suppose some subtree is to be replaced. It will suffice to have access to that part of the sorted list of tip values remaining after values for all tips on subtrees-to-the-left have been removed. We store this sublist in the variable us. Having allocated the initial values on us to the tips in the subtree, the rest can be passed on via some shared variable ws, say. Furthermore, a list vs of the tip values in the subtree can be produced.

So, at the root of the tree, vs is a list of all the tip values in the tree and us is a sorted version of vs. (Note also that ws will be instantiated to nil during execution.)

SOLUTION 2

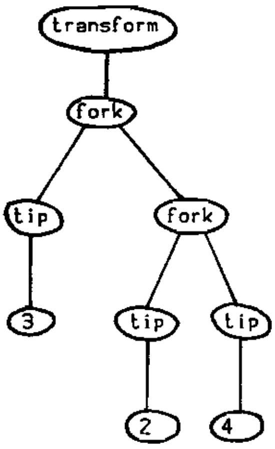
```
transform : Tree → Tree
transform t = var vs, ws ; replace t (sort vs) vs ws

replace : Tree → List Int → List Int → List Int → Tree

replace (tip n) us (var vs) (var ws)
                   = assign ws = tail us, vs = [n] : tip (head us)

replace (fork L R) us (var vs) (var ws)
                   = var xs, ys, zs ; assign vs = xs ++ ys ;
                     fork (replace L us xs zs) (replace R zs ys ws)
```

for some suitably defined function sort. Since sort is strict, an attempt to determine a new tip value will deadlock if all the tips do not eventually get demanded. As in Example 1, this can be avoided by using val or par (in an identical way).

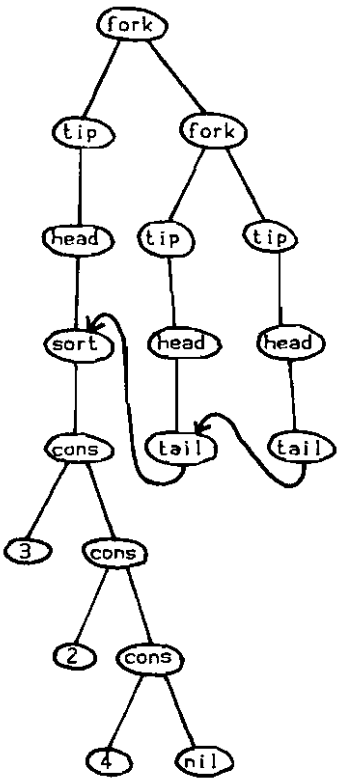
Note that the list of tip values is created by appending sublists together. It should be possible to formulate a more efficient solution that avoids the use of ++. We revise our method as follows.

ALTERNATIVE METHOD

As will be explained in Chapter 4, we would like an expression-graph such as



to reduce to



A program that achieved this reduction with minimal overheads might be regarded as an 'optimal' solution to the problem. We devise such a program by modifying Solution 2.

An extra argument `rs` is used by `replace` to keep a list of tip values on subtrees-to-the-right. vs is now used to store, for a given subtree, the list of its tip values appended on to `rs`. Thus, at the root of the tree to be transformed, `rs = nil` and vs is a list of all tip values (as before).

SOLUTION 2A

```
transform : Tree → Tree
transform t = var vs, ws ; replace t (sort vs) vs ws nil

replace : Tree → List Int → List Int
                             → List Int → List Int → Tree

replace (tip n) us (var vs) (var ws) rs
           = assign ws = tail us, vs = cons n rs ; tip (head us)

replace (fork L R) us (var vs) (var ws) rs
           = var ys, zs ;
             fork (replace L us vs zs ys) (replace R zs ys ws rs)
```

In Examples 1 and 2 we have used algorithms that would probably be favoured by a Prolog programmer. The cost of our efficient graph reduction mechanism is unexpected deadlock for the naive programmer.

In the next two examples we shall be concerned with the space complexity of some programs. Even for purely functional programs, lazy evaluation makes analysis difficult. For example, the expression (1..n), representing the list of integers from 1 to n, requires space linear in n if fully evaluated (without any part of the resulting list being garbage collected); whereas (head (1..n)) evaluates to 1 in constant space, since the list from 2 to n does not get constructed.

EXAMPLE 3

In [29] an analysis is presented of a function split which takes a list xs of characters and returns a pair comprising the first line of characters and the rest of the list. If the list xs is built up lazily, then in many circumstances its members can be garbage collected soon after they have been produced. For example, the split may be performed just so as to determine the first character of the first and second lines of xs. However, Hughes has shown that there are situations in which a 'sequential evaluation' strategy makes it impossible to obtain an expected constant space solution, no matter how we define the function split. Therefore he devises new primitives - par and synch - with which he annotates split in order to allow for the possibility of execution in constant space.

We now present a version of split that can be run in constant space even though ***only sequential evaluation is required.***

SOLUTION 3

```
split : List Char → List Char → List Char
split (cons x xs) (var ys) = assign ys = xs ; nil if x = CR
                             cons x (split xs ys) otherwise
```

Here, (split xs ys), where ys is an uninstantiated variable, will evaluate lazily to the first line of xs. ys then becomes bound to the remainder of xs.

Hughes gives a program that uses split in order to compute the length of the first line of a list and the length of the remainder of the list. We might write this as

```
program : List Char → (Int x Int)
program xs = var rest ; val pair (length (split xs rest))
                                 (length rest)
```

```
pair : * → ** → (* x **)
pair x y = (x.y)
```

Now the evaluation of either length (or both) requires only constant space, given a suitable garbage collector and assuming we use the space efficient definition of length formulated by Hughes. (His version of length is formed by annotating the definition given in Section 1.2 of Chapter 1 so that length' n (cons x xs) = val length' (n+1) xs .)

Why have we used val in the definition of program xs? This is because (length rest) can never be determined without parsing the first line of xs. This might have been inadvertantly attempted, resulting in deadlock, had we written

```
program xs = var rest ; (length (split xs rest), length rest)
```

We mention here that Hughes' synch no longer has to be treated as a primitive. He gives the following description of it:-

$$\text{synch } E = (E \;.\; E)$$

However, the two copies of E which are returned are actually different: call them E_1 and E_2. No demand is propagated from E_1 or E_2 to E until both have been demanded.

The following definition of synch achieves the same effect.

```
synch : * → (* x *)
synch x = var y. z ; ((assign z = x ; y) . (assign y = x ;  z))
```

Evaluation of (synch E) returns a pair of expressions. The combination of lazy evaluation and side-effects means that the values of both expressions must be demanded before E gets evaluated (to v say); only then can the value (v, v) of (synch E) be determined.

EXAMPLE 4

No set of examples of functional programs would be complete without inclusion of quicksort.

```
sort : List Int → List Int
sort  nil          = nil
sort (cons x xs) = sort [y←xs|y<x] ++ cons x (sort [y←xs|y≥x])
```

An analysis of the space/time complexities of this program appears in [29]. It is shown to have O(nlogn) average and $O(n^2)$ worst case time complexity, the same as for the imperative version of quicksort. However, we wish to reduce from quadratic to linear its (worst case) space complexity.

METHOD

Rather than compute [y←xs|y<x] and [y←xs|y≥x] separately, we can combine them by using a suitably defined function that partitions xs. This leads to the following program, where partition is similar to the function split of the previous example.

```
sort : List Int → List Int
sort  nil          = nil
sort (cons x xs) = var ys ;
                   sort (partition x xs ys) ++ cons x (sort ys)


partition : Int → List Int → List Int → List Int
partition x  nil (var ys) = assign ys = nil ; nil
partition x (cons z zs) (var ys)
                          = var ys' ;
                            assign ys = cons z ys' ;
                            partition x zs ys'          if x≤z
                            cons z (partition x zs ys)  otherwise
```

The program is deadlock-free: see Chapter 3 for its formal derivation and a proof of freedom from deadlock. It can be further improved by use of an accumulating parameter, as shown in Solution 4.

Solution 4

```
sort : List Int → List Int
sort = sort' nil


sort' : List Int → List Int → List Int
sort' rs nil          = rs
sort' rs (cons x xs) = var ys ;
                       sort' (cons x (sort' rs ys))
                             (partition x xs ys)
```

where partition is as defined above.

We observe that (partition x xs ys), where xs is a list of integers and ys is an uninstantiated variable, evaluates (lazily) to a list, us say; this evaluation has the side-effect of binding ys to a list, vs say. Thus, xs gets partitioned into two sublists, us and vs, and #xs = #us + #vs.

sort can be shown to have linear space complexity, as follows:-

Without loss of generality, we can assume that the list being sorted is not referenced elsewhere. (If this were not the case, more space would be required during sorting. The amount of additional space would be linear in the length of this list, and so can safely be ignored, since we are only trying to prove that sort runs in linear space.)

Consider some call of partition, (partition x xs ys), occurring during the sorting of a list. ***The only reference to* xs *will be from this call.*** So, rewriting the expression according to the definition of partition will free the cons cells from which the list xs is built up: the cells can be garbage collected and, hence, reallocated. In particular, they can be used to construct the lists us and vs. Thus, only a constant amount of space is required during the evaluation of (partition x xs ys); that is, its storage needs are independent of the length of xs.

Now, for any expression E that evaluates to a list, let S⟦E⟧ denote the space required for it to be so evaluated.

<u>THEOREM</u> For any list xs and expression rs that evaluates to a list

$$S[\![\text{sort' rs xs}]\!] = S[\![\text{rs}]\!] + O(\#\text{xs})$$

<u>PROOF</u>

Because partition runs in constant space, we have that

$\exists$ *some integer* k *s.t. for any list* xs.

S⟦sort' rs (cons x xs)⟧ ≤ S⟦sort' (cons x (sort' rs vs)) us⟧ + k

for some lists us, vs, *with* #us + #vs = #xs ... (1)

We shall prove, by induction on #xs, that for any list xs,

$$S[\![\text{sort' rs xs}]\!] \le S[\![\text{rs}]\!] + k\#\text{xs}$$

<u>Base Case</u> #xs = 0 ⇒ xs = nil

$$S[\![\text{sort' rs nil}]\!] = S[\![\text{rs}]\!] \le S[\![\text{rs}]\!] + k\#\text{nil}$$

<u>Inductive Case</u> We can assume that

$$S[\![\text{sort' rs ys}]\!] \le S[\![\text{rs}]\!] + k\#\text{ys}$$

for any list ys *with* #ys≤#xs. Then

```
  S⟦sort' rs (cons x xs)⟧
≤ S⟦sort' (cons x (sort' rs vs)) us⟧ + k. for some lists us, vs.
                                           with #us + #vs = #xs. by (1)
≤ S⟦cons x (sort' rs vs)⟧ + k#us + k    by ind. hyp.
```

$= S[\![\texttt{sort' rs vs}]\!] + k(\#\texttt{us} + 1)$
$\leq S[\![\texttt{rs}]\!] + k(\#\texttt{us} + \#\texttt{vs} + 1)$ *by ind. hyp.*
$= S[\![\texttt{rs}]\!] + k\#(\texttt{cons x xs})$

□

COROLLARY For any list xs

$$S[\![\texttt{sort xs}]\!] = O(\#\texttt{xs})$$

PROOF $S[\![\texttt{sort xs}]\!] = S[\![\texttt{sort' nil xs}]\!] + O(1)$
$= S[\![\texttt{nil}]\!] + O(\#\texttt{xs})$ *by Theorem*
$= O(\#\texttt{xs})$

□

The remaining examples explore further the use of side-effects as a way of combining computations. It is also shown that side-effects can be used as an alternative to *continuations*. We shall return to quicksort in Example 7.

EXAMPLE 5

The Fibonacci function can be defined as follows:-

```
fib : Int → Int
fib  0     = 1
fib  1     = 1
fib (n+2) = fib (n+1) + fib n
```

Treating the above as a program rather than simply as a specification, it is obviously a very inefficient way to determine the nth Fibonacci number: it gives rise to many repeated computations.

METHOD

There are many efficient algorithms for computing Fibonacci numbers [5.41], but the ones we shall consider simply 'remember' the value of fib (n-1) when evaluating fib n.

A linear time solution can be achieved (using a recursive where construct) by ***tupling***, as follows:-

```
fib : Int → Int
fib  0    = 1
fib (n+1) = fst (fibs n)

fst : (* x **) → *
fst (x , y) = x

fibs : Int → (Int x Int)
fibs  0    = (1 , 1)
fibs (n+1) = (x + y , x) where (x , y) = fibs n
```

So, fibs n = (fib (n+1) , fib n).

The above solution necessitates the construction and subsequent destruction of tuples. We believe this to be a source of inefficiency, and consider the following solution (using side-effects) to be an improvement.

SOLUTION 5

```
fib : Int → Int
fib  0    = 1
fib (n+1) = var x ; fib' n x

fib' : Int → Int → Int
fib'  0     (var x) = assign x = 1 ; 1
fib' (n+1)  (var x) = var y ; assign x = fib' n y ; x + y
```

Here, z = fib' n x ⇔ x = fib n ∧ z = fib (n+1) . Note, however, this program only makes progress under the assumption that + evaluates its left argument before its right, or alternatively evaluates these two arguments in parallel.

It is also possible to use continuations instead of tuples to obtain a linear time Fibonacci function. In the following table we give such a program; alongside appears a similar program that relies on side-effects. The programs are free from deadlock, irrespective of the order in which + evaluates its arguments.

ALTERNATIVE SOLUTIONS

```
fib : Int → Int                      | fib : Int → Int
fib  0     = 1                       | fib  0     = 1
fib (n+1) = fib' n (λx y. x)         | fib (n+1) = var x, y ;
                                     |               fib" n x y x
                                     |
fib' : Int →                         | fib" : Int →
      (Int → Int → Int) → Int        |        Int → Int → Int  → Int
fib'  0     θ = θ 1 1                | fib"  0    (var x) (var y) t
                                     |     = assign x = 1, y = 1 ; t
fib' (n+1) θ = fib' n                | fib" (n+1) (var x) (var y) t
              (λy z. θ (y + z) y)    |     = var z ; assign x = y+z ;
                                     |       fib" n y z t
```

The above solutions can be derived from the definitions

```
fib' n θ = θ (fib (n+1)) (fib n)
fib" n (var x) (var y) t = assign x = fib (n+1), y = fib n ; t
```

Here the function θ is a continuation; the integer t is a kind of continuation that has been applied to its arguments.

EXAMPLE 6

The task of an assembler is to translate an assembly language listing into a form suitable for machine execution. A formal specification of a simple assembler appears in [50]. In a given line of assembly code a symbolic operand referring to some label can appear, but the location associated with the label may not be determined until a line occurring further on in the listing gets assembled.

METHOD

The standard method is to construct a symbol table relating labels to locations in the store of the machine. We shall sketch a solution that takes advantage of lazy evaluation and side-effects to build up the symbol table 'on-the-fly'.

Each line of assembly code will consist of three fields:-

```
type A = Label × (Opcode × Operand)
```

The Label field can be left 'blank', that is, we assume blank ∈ Label.

Similarly, a line of machine code has three fields for numbers denoting the location, machine instruction and operand.

```
type M = Int × Int × Int
```

A symbol table has type

```
type Table = List (Label × Int)
```

No details will be given of the functions for decoding symbolic instructions into machine instructions and looking up operands in the symbol table.

```
decode : Opcode → Int
look_up: Table → Operand → Int
```

The assembler can be expressed as a function assemble that takes an assembly language listing and produces a machine code listing. It calls upon the auxiliary function assemble' which has parameters n, the number of the line being assembled, convert, the function that converts operands into integers, and rest, that part of the symbol table to be built from the remainder of the assembly listing. assemble' handles the label field of each line of assembly code, recording a new entry in the table rest whenever a non-blank label field is encountered. The remaining tasks of the assembly process are passed on to the function assemble".

SOLUTION 6

```
assemble : List A → List M
assemble xs = var symtab; assemble' xs 1 (look_up symtab) symtab

assemble': List A → Int → (Operand → Integer) → Table → List M
assemble'nil n convert (var rest) = assign rest = nil ; nil
assemble'(cons (label, line) xs) n convert (var rest)
     = assemble" line n xs convert rest       if label = blank
       var rest';
       assign rest = cons (label, n) rest';
       assemble" line n xs convert rest'      otherwise

assemble": (Opcode × Operand) → Int → List A →
                (Operand → Integer) → Table → List M
assemble" (opcode, operand) n xs convert (var rest)
     = val (cons (n, decode opcode, convert operand))
           (val (assemble' xs) (n+1) convert rest)
```

Note 1. Initially assemble' is called at line 1 of the listing with rest = symtab, since the entire symbol table has to be constructed.

Note 2. assemble" has been annotated with val to ensure that (i) all the listing is parsed before any look ups (by convert) are made in the symbol table, and (ii) the new line number is calculated as each line is assembled.

EXAMPLE 7

Consider once again the task of sorting a list by means of the quicksort algorithm (Example 4). Note that an imperative version of quicksort will complete the partitioning of a list *before* sorting the sublists so generated. Because of lazy evaluation, in the functional programs considered so far these sublists are generated ***incrementally***: the sublists are produced as sorting progresses. Such unwanted laziness can have a detrimental effect on the space complexity of a program. Thus, although we were successful in obtaining a linear space version of quicksort, it is worthwhile trying to develop a solution that executes in a similar way to an imperative program.

The following program seems an appropriate staring point:-

```
sort : List Int → List Int
sort = sort' nil

sort' : List Int → List Int → List Int
sort' rs nil        = rs
sort' rs (cons x xs) = sort' (cons x (sort' rs [y←xs|y≥x]))
                             [y←xs|y<x]
```

In this last equation, we should like [y←xs|y<x] and [y←xs|y≥x] to be ***fully*** evaluated to lists before sorting can begin. This can be achieved by defining a function partition that is supplied with an appropriate continuation. We shall also demonstrate a related method involving side-effects.

CONTINUATION METHOD

Partitioning an empty list results in two empty sublists, to which the continuation can be applied. For a non-empty list, with first element z say, the tail of the list should be partitioned and z cons-ed on to the appropriate sublist; after which the continuation can be applied.

SOLUTION

```
sort' : List Int → List Int → List Int
sort' rs nil         = rs
sort' rs (cons x xs) = partition x xs
                         (λus vs. sort' (cons x (sort' rs vs)) us)


partition : Int → List Int →
                    (List Int → List Int → List Int) → List Int
partition x  nil θ = θ nil nil
partition x (cons z zs) θ
        = partition x zs (λus vs. θ us (cons z vs)) if x≤z
          partition x zs (λus vs. θ (cons z us) vs) otherwise
```

SIDE–EFFECTS METHOD

Let us, vs be the sublists resulting from partitioning a list. If the list is empty then partition must assign us = nil, vs = nil. If the list is non-empty, with head z, then z is the first element of either us or vs. Thus, in one case partition must assign us = cons z us', where us', vs are the sublists resulting from partitioning the remainder of the list; in the other case partition must assign vs = cons z vs', for sublists us, vs'.

SOLUTION

```
sort' : List Int → List Int → List Int
sort' rs  nil        = rs
sort' rs (cons x xs) = var us, vs ;
                       partition x xs us vs
                       (sort' (cons x (sort' rs vs)) us)


partition : Int → List Int →
                     List Int → List Int → List Int → List Int
partition x  nil (var us) (var vs) ws
        = assign us = nil, vs = nil ; ws
partition x (cons z zs) (var us) (var vs) ws
        = var vs' ; assign vs = cons z vs' ;
          partition x zs us vs' ws            if x≤z
          var us' ; assign us = cons z us' ;
          partition x zs us' vs ws            otherwise
```

The above solutions run in linear space. This can be shown by (rather difficult) analysis, similar to that of Solution 4.

2.5 CONCLUSION

A new extension that combines features of functional, logic and imperative languages has been presented and used to solve a variety of programming problems. Program execution is solely by demand-driven graph reduction, and so it should be possible to extend graph reduction based implementations of functional languages to support programs with side-effects. A consequence of the decision not to use unification as the evaluation mechanism is that some programs deadlock: the order in which expressions are evaluated must be taken into account by the programmer if programs are to be produced that never deadlock. However, deadlock can sometimes be avoided by use of control annotations.

The extension enables some interesting programs to be developed that would otherwise be unavailable to the functional programmer. (It is hoped that this has been well illustrated by the worked examples.) In particular:

(1) Side-effects allow computations to be combined together in a new way and it is suggested that the standard tupling alternative is often less efficient. (Only constant factor improvements in space and time complexity are expected in general; though in Example 3 an order of magnitude space improvement was obtained.)

(2) Hughes suggested a synchronization function to reduce the space requirements of certain programs. His function can be defined in terms of side-effects, or programs with side-effects can be used directly to give space efficient solutions.

(3) Examples have been given of programs with side-effects that are very similar to purely functional programs involving continuations. However, the side-effects solutions are not higher-order, since functions do not have to be passed as arguments in calls to other functions.

CHAPTER 3

TRANSFORMATION

3.1 INTRODUCTION

Chapter 2 has demonstrated the possibilities of (a) programming directly in a language with side-effects, and (b) improving upon the performance of existing purely functional programs by developing equivalent programs that use side-effects. In this chapter we aim to give a more formal treatment to the development process of (b).

We shall adopt the following as a working hypothesis:- The programmer is capable of writing clear (modular) functional programs. It may often be the case that such programs are less efficient, that is, run more slowly or use more space, than is desired; so it is further assumed that the programmer knows some standard transformation techniques. Our objective is to provide additional methods that will allow use to be made of side-effects.

The next section serves to introduce an extension to the fold/unfold method of Burstall and Darlington [7]. The transformation system includes new rules that cater for programs with side-effects. In the remainder of the chapter a few simple transformation strategies are described and are applied to various programs. Many of the examples from Chapter 2 are reworked as transformation problems. Strategies for developing programs with side-effects are employed and can be seen as alternatives to strategies that make use of tuples and continuations.

Adopting a transformational approach to the development of programs with side-effects benefits us in two ways. Firstly, a synthesized program is known to meet its specification. That is, it is equivalent to the original purely functional program. Secondly, we are aware of those steps in the transformation that can give rise to a program which deadlocks. It can sometimes be revealed by local analysis whether such steps are in fact *safe* (have not introduced deadlock). If a step is unsafe, changing order of evaluation by careful use of control annotations or attempting a different transformation can often rectify the situation. When local analysis is unhelpful, a direct proof that a program is free from deadlock may still be possible.

Note that our methods are formal, but not automatic. Machine assistance may be helpful during program transformation [18], but there appears to be only a remote possibility of being able to compile from inefficient functional programs into efficient programs with side-effects.

3.2 TRANSFORMATION RULES

This section assumes some familiarity with the Burstall-Darlington approach to program transformation. Two limitations of their system are that (i) transformation retains correctness, but termination may be lost unless some extra restriction is imposed, and (ii) no general conditions are given under which transformations are known to improve the efficiency of programs. The system has been extended with two new transformation rules, *introducing assigned variables* and the *move-in transformation*, so that programs with side-effects can be developed. The move-in transformation is not always safe, in that it may introduce deadlock. Therefore, conditions under which the rule can be used safely are also provided.

Although the Burstall-Darlington approach is concerned with the redefinition of functions, it can be looked upon as establishing the equality of certain expressions. We take a 'disjoint and exhaustive subset' [7] of these equations as the new program. When viewed in this way, the fold, unfold and abstraction rules amount to nothing more than substituting equals for equals. As is demonstrated by the following (trivial) example, care has to be taken to ensure that the new program still terminates:-

From the program

```
f : Int → Int
f n = 0
```

we can obtain

```
f n = 0          by definition
    = f n        by folding
```

However, the program

```
f : Int → Int
f n = f n
```

is not equivalent to the original definition of f, because evaluation of (f E), for some expression E, fails to terminate instead of returning 0.

The programmer can check for termination either by reasoning (usually in some informal, operational way) about the new program, or by using the technique given in [33,47] .

More subtle termination problems can arise from transforming a non-strict function into a strict one. For example, we might give an inductive definition to f above. That is,

```
f : Int → Int
f  0    = 0
f (n+1) = 0
```

can be treated for most purposes as equivalent to the original definition of f. However, for an undefined argument, f is now itself undefined, whereas previously it returned 0. Bird had to guard against this in devising 'circular programs' [4]. He has shown that the partial approximations to a program, as defined by fixed point theory, can be used to establish that a program is well-behaved.

Turning now to programs with side-effects, the Burstall-Darlington transformation rules can be used as before. However, (i) unfolding, and (ii) using properties of operators, are best applied to expressions without side-effects. For otherwise:-

(i) To see the problem with unfolding, consider the definition

```
double : Int → Int
double n = n + n
```

Now compare the evaluation of (double (<u>assign</u> x = 3 ; 2)) with that of ((<u>assign</u> x = 3 ; 2) + (<u>assign</u> x = 3 ; 2)). The former returns the value 4 and has the side-effect of binding 3 to x. The latter gives an error from attempting to assign to x twice, even though it can be derived from the expression (double (<u>assign</u> x = 3 ; 2)) by unfolding according to the definition of double.

(ii) The commutative property of addition is an example of a mathematical law that no longer holds when dealing with expressions involving side-effects: see Chapter 2, Section 2.3, Note (4). In this case the problem is one of introducing deadlock by altering the order in which expressions are evaluated.

In Note (3) of Section 2.3, we saw that the recursive <u>where</u> construct can be re-expressed in terms of <u>var</u> and <u>assign</u> as follows:-

(E <u>where</u> x = E') = (<u>var</u> x ; <u>assign</u> x = E' ; E)

Hence, it is possible to devise a rule, equivalent to <u>where</u>-abstraction, for a language with side-effects:

<u>DEFINITION</u> By ***introducing an assigned variable***, we can transform an expression E into (<u>var</u> x ; <u>assign</u> x = E' ; E) , for some variable x and expression E', provided E is not already in the scope of x.

For example, we can perform the following transformation.

```
foo (baz x) = var y ; assign y = baz x ; foo (baz x)
                                introducing an assigned variable
            = var y ; assign y = baz x ; foo y
                                referential transparency
```

This is equivalent to the transformation

```
foo (baz x) = foo y where y = baz x
                                abstraction
```

Note that we have to be slightly careful in using referential transparency: within an expression (<u>assign</u> x = E' ; E), we cannot replace occurrences of x by E' unless evaluation of E' is free from side-effects.

Finally, we provide a transformation rule that can be used to take advantage of the order in which expressions are evaluated:-

<u>Definition</u> The *move-in transformation* takes the form

$$(\underline{\texttt{assign}}\ x = E\ ;\ E')\ =\ E''$$

where E'' is E' with some subexpression E''' replaced by

$$(\underline{\texttt{assign}}\ x = E\ ;\ E''')$$

There are some problems in transforming an expression E_1 into an expression E_2 by this rule:-

Evaluation of E_2 might suspend rather than give the same value as E_1 . For example, (<u>assign</u> x = not ; x true) evaluates to false, whereas evaluation of (x (<u>assign</u> x = not ; true)) suspends.

Although E_1 and E_2 may have the same *value*, their evaluations may differ in *effect*. For example, (<u>assign</u> x = 3 ; K 1 2) returns the value 1 and binds x to 3, whereas (K 1 (<u>assign</u> x = 3 ; 2)) returns 1, but fails to assign to x.

However, a condition sufficient to ensure the safety of the move-in transformation

$$(\underline{\texttt{assign}}\ x = E\ ;\ E')\ =\ E''$$

is that: evaluation of E'' has the side-effect of binding x to E. If this condition is not satisfied and E'' is strict in x, then the move-in introduces deadlock.

Consider the three general cases:

$(\underline{\texttt{assign}}\ x = E\ ;\ (E_1\ E_2)) = ((\underline{\texttt{assign}}\ x = E\ ;\ E_1)\ E_2)$ (1)

$(\underline{\texttt{assign}}\ x = E\ ;\ (E_1\ E_2)) = (E_1\ (\underline{\texttt{assign}}\ x = E\ ;\ E_2))$ (2)

$(\underline{\texttt{assign}}\ x = E\ ,\ y = E_1\ ;\ E_2) =$
$(\underline{\texttt{assign}}\ y = (\underline{\texttt{assign}}\ x = E\ ;\ E_1)\ ;\ E_2)$ (3)

Case (1) is always safe because of normal order reduction; for case (2), we have to look at E_1 ; for case (3), E_2 has to be analysed. Thus:

(2) is safe if evaluation of E_1 does not need the value x, and E_1 reduces to a strict function, for example, E_1 = not. It introduces deadlock if evaluation of E_1 does demand the value of x, for example, $E_1 = x$. If E_1 reduces to a non-strict function then we cannot rely on the safety of the transformation; though it might still be possible to prove that we have developed a program that is free from deadlock.

Similarly, (3) is safe if evaluation of E_2 needs the value of y but not x, or is guaranteed to evaluate y even if evaluation of x suspends. It introduces deadlock if it needs only the value of x, or does not evaluate y until the value of x has been determined. Thus, if add evaluates its arguments from left to right, (3) is safe for E_2 = (add y x), but deadlocks for E_2 = (add x y). Again, more information is required to determine its safety in other situations.

The reasoning suggested in the last two paragraphs is rather operational in nature and requires an appreciation of order of evaluation. It is suggested that the axiomatic semantics described in Chapter 5 provides some help in this: see Section 5.3 in particular. A few examples of safe move-in transformations are also considered in Chapter 4, and shown to preserve equivalence.

It is worth mentioning that the properties of val and par ensure that the following move-in transformations are safe.

(<u>assign</u> $x = E$; (val E_1 E_2)) = (val E_1 (<u>assign</u> $x = E$; E_2))
(<u>assign</u> $x = E$; (par E_1 E_2)) = (par E_1 (<u>assign</u> $x = E$; E_2))
(<u>assign</u> $x = E$; (par E_1 E_2)) = (par (<u>assign</u> $x = E$; E_1) E_2)

3.3 SIDE-EFFECTS AS AN ALTERNATIVE TO TUPLING

In this section we shall demonstrate a new program transformation strategy made possible by side-effects. It is very similar to the ***tupling strategy*** [40, 41] and is almost as easy to use: we just have to watch out for deadlock when performing a move-in transformation. The idea is to achieve a gain in efficiency 'by introducing a new recursive definition which intertwines what were originally separate computations' [7].

Suppose we wish to combine the computation of $(f\ x)$ and $(g\ x)$. When using the tupling strategy, the eureka step is to define a new function

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

We then try to synthesize a more efficient version of h. Note that it would make no difference if we defined $h\ x = (g\ x\ ,\ f\ x)$. However, with side-effects we can try to combine the expressions in two distinct ways. The way that is chosen should take advantage of the order in which the expressions are to be evaluated. That is,

$h\ x$ ($\underline{\text{var}}$ y) = $\underline{\text{assign}}$ $y = g\ x\ ;\ f\ x$

is suitable if the value of $(f\ x)$ is required before that of $(g\ x)$, whereas

$h\ x$ ($\underline{\text{var}}$ y) = $\underline{\text{assign}}$ $y = f\ x\ ;\ g\ x$

is appropriate when the value of $(g\ x)$ gets demanded first.

To see what happens in practice, we shall solve various problems by these strategies. We start with a simple example, the Fibonacci function (Example 5 of Chapter 2).

Given the definition

```
fib : Int → Int
fib  0     = 1
fib  1     = 1
fib (n+2) = fib (n+1) + fib n
```

we might decide to combine computation of (fib (n+1)) and (fib n).

Tupling Strategy

Define a new function `fibs` such that

```
fibs : Int → (Int × Int)
fibs n = (fib (n+1), fib n)
```

It is then possible to redefine `fib` in terms of `fibs`, as follows.

```
fib (n+1) = fst (fib (n+1), fib n)     by definition of fst
          = fst (fibs n)      folding with definition of fibs
```

It remains to synthesize a more efficient version of `fibs`. This can be achieved by giving an inductive definition to `fibs`.

```
fibs 0 = (fib 1, fib 0)    instantiating n to 0 in definition of fibs
       = (1,1)             unfolding with definition of fib

fibs (n+1)  = (fib (n+2), fib (n+1))
                           instantiating n to (n+1) in definition of fibs
            = (fib (n+1) + fib n, fib (n+1))
                           unfolding with definition of fib
            = (x + y, x) where (x, y) = (fib (n+1), fib n)
                           abstraction
            = (x + y, x) where (x, y) = fibs n
                           folding with definition of fibs
```

The transformation has produced a solution that computes `fib` in linear time, namely

```
fib : Int → Int
fib  0     = 1
fib (n+1) = fst (fibs n)

fibs : Int → (Int × Int)
fibs  0    = (1,1)
fibs (n+1) = (x + y, x) where (x, y) = fibs n
```

SIDE-EFFECTS STRATEGY

Consider the function

```
fib' : Int → Int → Int
fib' n (var x) = assign x = fib n ; fib (n+1)
```

Now,

```
fib (n+1) = var x ; assign x = fib n ; fib (n+1)
                         introducing an assigned variable
          = var x ; fib' n x     folding with definition of fib'
```

Consider, once again, the cases n=0, n>0.

```
fib' 0 (var x) = assign x = fib 0 ; fib 1
               = assign x = 1 ; 1


fib' (n+1) (var x) = assign x = fib (n+1) ; fib (n+2)
                   = assign x = fib (n+1) ; fib (n+1) + fib n
                            unfolding with definition of fib
                   = assign x = fib (n+1) ; x + fib n
                            referential transparency
                   = var y ; assign y = fib n ,
                                    x = fib (n+1) ; x + fib n
                            introducing an assigned variable
                   = var y ; assign y = fib n ,
                                    x = fib (n+1) ; x + y
                            referential transparency
                   = var y ;
                     assign x = (assign y = fib n ; fib (n+1)) ;
                     x + y   move-in transformation
                   = var y ; assign x = fib' n y ; x + y
                            folding with definition of fib'
```

Note that the move-in transformation used above is safe under the assumption that evaluation of $(E_1 + E_2)$ demands the value of E_1, even though evaluation of E_2 might suspend.

Thus, the solution based on the side-effects strategy is

```
fib : Int → Int
fib  0    = 1
fib (n+1) = var x ; fib' n x


fib' : Int → Int → Int
fib'  0    (var x) = assign x = 1 ; 1
fib' (n+1) (var x) = var y ; assign x = fib' n y ; x + y
```

We shall now tackle Example 1 of Chapter 2 by transformational programming. The development of a solution based on tupling is described in [4]. Here only the side-effects strategy will be demonstrated. It will prove necessary to make use of *control annotations* if deadlock is to be avoided. So, consider the following specification of the problem.

```
transform : Tree → Tree
transform t = replace t (tmin t)


replace : Tree → Int → Tree
replace (tip n)    m = tip m
replace (fork L R) m = fork (replace L m) (replace R m)


tmin : Tree → Int
tmin (tip n)    = n
tmin (fork L R) = (tmin L) MIN (tmin R)
```

TRANSFORMATION

First we introduce a function replace' defined by

```
replace' : Tree → Int → Int → Tree
replace' t m (var v) = assign v = tmin t ; replace t m
```

This allows us to synthesize a new definition of transform :-

```
transform t = replace t (tmin t)
            = var m ; assign m = tmin t ; replace t (tmin t)
                      introducing an assigned variable
            = var m ; assign m = tmin t ; replace t m
                      referential transparency
            = var m ; replace' t m m
                      folding with definition of replace'
```

Redefining replace' by cases, gives us

```
replace' (tip n) m (var v) = assign v = tmin (tip n) ;
                              replace (tip n) m
                                  instantiating tip n for t
                           = assign v = n ; tip m
                                  unfolding with definitions
                                    of tmin and replace
```

```
  replace' (fork L R) m (var v)
= assign v = tmin (fork L R) ; replace (fork L R) m
              instantiating fork L R for t
= assign v = (tmin L) MIN (tmin R) ;
  fork (replace L m) (replace R m)
              unfolding with definitions of tmin and replace
= var y , z ; assign v = y MIN z , y = tmin L , z = tmin R ;
  fork (replace L m) (replace R m)
              introducing assigned variables
```

At this stage of the derivation we should like to employ the move-in transformation to give us

```
  var y , z ; assign v = y MIN z ;
  fork (assign y = tmin L ; replace L m)
       (assign z = tmin R ; replace R m)
```

However, we cannot show that this is a safe step to take. Indeed, it was shown in Chapter 2, that in certain circumstances the program so produced will deadlock.

Instead, we return to the original specification and convince ourselves that it is acceptable to annotate replace as follows:-

replace (fork L R) m = par (par fork (replace L m)) (replace R m)

That is, we use an 'eager' rather than 'lazy' version of fork. This will allow us to move-in the assignments safely.

The previous derivation only has to be changed slightly, and we arrive at the stage

```
  replace' (fork L R) m (var v)
= var y , z ; assign v = y MIN z , y = tmin L , z = tmin R ;
  par (par fork (replace L m)) (replace R m)
= var y , z ; assign v = y MIN z ;
  par (par fork (assign y = tmin L ; replace L m))
      (assign z = tmin R ; replace R m)
              move-in transformation
= var y , z : assign v = y MIN z ;
  par (par fork (replace' L m y)) (replace' R m z)
              folding with definition of replace'
```

SOLUTION

```
transform : Tree → Tree
transform t = var m ; replace' t m m


replace' : Tree → Int → Int → Tree
replace' (tip n)      m (var v) = assign v = n ; tip m
replace' (fork L R) m (var v) = var y , z ; assign v = y MIN z ;
                                  par (par fork (replace' L m y))
                                      (replace' R m z)
```

As an alternative, we might have chosen to annotate replace with val. Then the move-in

```
  var y , z ; assign v = y MIN z , y = tmin L , z = tmin R ;
  val (val fork (replace L m)) (replace R m)
= var y , z ; assign v = y MIN z , y = tmin L ;
  val (val fork (replace L m)) (assign z = tmin R ; replace R m)
```

is clearly safe. However, the further move-in

```
  var y , z ; assign v = y MIN z ;
  val (val fork (assign y = tmin L ; replace L m))
      (assign z = tmin R ; replace R m)
```

relies on evaluation of (assign z = tmin R ; replace R m) not requiring the value of y. Now, R is a given tree, but m is dependent on y. Fortunately, evaluation of (replace R m) does return an answer without demanding the value of m. This can be shown by structural induction on R:

PROOF

(i) replace (tip n) m = tip m

and m is not evaluated because tip is lazy (non-strict).

(ii) Suppose evaluation of (replace L' m) and (replace R' m) do not evaluate m. Then nor does evaluation of (replace (fork L' R')), since

```
replace (fork L' R') = val (val fork (replace L' m))
                           (replace R' m)
```

$\Box$

Thus, using `val` instead of `par` in the above solution also yields a program that is free from deadlock.

A rather different solution involving side-effects can be developed. Instead of `(tmin t)` being determined as a side-effect of the evaluation of `(replace t m)`, the replacement tree is determined as a side-effect of the evaluation of the minimum tip value. This time the original specification should be annotated so that

```
transform t = val (replace t) (tmin t)
```

Our solution will make use of the primitive `seq`, where (seq E_1 E_2) evaluates E_1, and, if and when this returns a value, evaluates to E_2.

TRANSFORMATION

Define `tmin'` by

```
tmin' : Tree → Tree → Int → Int
tmin' t (var s) m = assign s = replace t m ; tmin t
```

Synthesize a new definition of `transform`.

```
transform t = val (replace t) (tmin t)
            = var m ; assign m = tmin t ; val (replace t) m
                      introducing an assigned variable
            = var m ; assign m = tmin t ; seq m (replace t m)
                      identity
            = var m, s ; assign s = replace t m, m = tmin t ;
              seq m s
                      introducing an assigned variable
            = var m, s ;
              assign m = (assign s = replace t m ; tmin t) ;
              seq m s
                      move-in transformation
            = var m, s ; assign m = tmin' t s m ; seq m s
```

The identity (val E x) = (seq x (E x)) should be evident because evaluation of both expressions involves first the evaluation of x and then the evaluation of (E x).

Synthesize a new definition of tmin'.

```
  tmin' (tip n) (var s) m
= assign s = replace (tip n) m ; tmin (tip n)
= assign s = tip m ; n


  tmin' (fork L R) (var s) m
= assign s = replace (fork L R) m ; tmin (fork L R)
= assign s = fork (replace L m) (replace R m) ;
  (tmin L) MIN (tmin R)
= var L', R';
  assign s = fork L' R', L' = replace L m, R' = replace R m ;
  (tmin L) MIN (tmin R)
                  introducing assigned variables
= var L', R'; assign s = fork L' R';
   (assign L' = replace L m ; tmin L)
  MIN
   (assign R' = replace R m ; tmin R)
                  move-in transformation
= var L', R'; assign s = fork L' R';
  (tmin' L L' m) MIN (tmin' R R' m)
```

The move-in performed above is safe because MIN is strict in both arguments and (tmin L), (tmin R), are independent of L', R'.

SOLUTION

```
transform : Tree → Tree
transform t = var m, s ; assign m = tmin' t s m ; seq m s


tmin' : Tree → Tree → Int → Int
tmin' (tip n)    (var s) m = assign s = tip m ; n
tmin' (fork L R)(var s) m = var L', R'; assign s = fork L' R';
                            (tmin' L L' m) MIN (tmin' R R' m)
```

The synthesis of solutions to Example 2 of Chapter 2 is similar to that given above: the details are omitted here. The final problem to be tackled in this section is quicksort (Example 4).

```
sort : List Int → List Int
sort  nil         = nil
sort (cons x xs) = sort [y←xs|y<x] ++ cons x (sort [y←xs|y≥x])
```

<u>TRANSFORMATION</u>

Define

```
partition : Int → List Int → List Int → List Int
partition x xs (var ys) = assign ys = [y←xs|y≥x] ; [y←xs|y<x]
```

Redefine sort in terms of partition.

```
sort'(cons x xs) = sort [y←xs|y<x] ++ cons x (sort [y←xs|y≥x])
                 = var ys ; assign ys = [y←xs|y≥x] ;
                   sort [y←xs|y<x] ++ cons x (sort ys)
                      introducing an assigned variable
                 = var ys ;
                   sort (assign ys = [y←xs|y≥x] ; [y←xs|y<x])
                   ++ cons x (sort ys)
                      move-in transformation
                 = var ys ;
                   sort (partition x xs ys) ++ cons x (sort ys)
                      folding with definition of partition
```

The move-in above is safe because sort is strict and ++ forces evaluation of its first argument.

Synthesize a new definition of partition.

```
partition x nil (var ys) = assign ys = [y←nil|y≥x] ; [y←nil|y<x]
                         = assign ys = nil ; nil
```

```
  partition x (cons z zs) (var ys)
= assign ys = [y←(cons z zs)|y≥x] ; [y←(cons z zs)|y<x]
```

<u>CASE</u> x≤z

```
  partition x (cons z zs) (var ys)
= assign ys = cons z [y←zs|y≥x] ; [y←zs|y<x]
= var ys' ; assign ys = cons z ys', ys' = [y←zs|y≥x] ;
  [y←zs|y<x]
= var ys' ; assign ys = cons z ys' ; partition x zs ys'
```

CASE x>z

```
  partition x (cons z zs) (var ys)
= assign ys = [y←zs|y≥x] ; cons z [y←zs|y<x]
= cons z (assign ys = [y←zs|y≥x] ; [y←zs|y<x])
                 move-in transformation
= cons z (partition x zs ys)
```

It is not possible to determine by local analysis whether the move-in is safe or introduces deadlock. This could be remedied by employing a cons that evaluated its tail eagerly. Instead, the solution is presented accompanied by a proof that it is free from deadlock.

SOLUTION

```
sort : List Int → List Int
sort  nil          = nil
sort (cons x xs) = var ys ;
                     sort (partition x xs ys) ++ cons x (sort ys)
```

```
partition : Int → List Int → List Int → List Int
partition x  nil (var ys) = assign ys = nil ; nil
partition x (cons z zs) (var ys)
                         = var ys' ;
                           assign ys = cons z ys' ;
                           partition x zs ys'          if x≤z
                           cons z (partition x zs ys)  otherwise
```

The main proof will require the following Lemma.

LEMMA Given x, an integer, xs, a list of integers, and ys, an uninstantiated variable. Then, (partition x xs ys) evaluates (lazily) to a list without deadlocking. Further, if it has been evaluated to a list, then ys must have been instantiated to a list.

PROOF By structural induction on xs.

CASE xs = nil

Trivial, since (partition x nil ys) = (assign ys = nil ; nil).

CASE xs = cons z zs

a) $x \leq z$

```
  (partition x (cons z zs) ys)
= (var ys' ; assign ys = cons z ys' ; partition x zs ys')
```

But (partition x zs ys') evaluates to a list, ps say, with ys' being instantiated to a list, qs say, by inductive hypothesis.

Therefore, (partition x xs ys) evaluates to the list ps, with ys instantiated to the list (cons z qs).

b) $x > z$

```
(partition x (cons z zs) ys) = (cons z (partition x zs ys))
```

and, again the inductive hypothesis gives us that, (partition x xs ys) evaluates to a list without deadlocking, and with ys instantiated to a list. □

We can now prove freedom from deadlock for quicksort itself.

THEOREM Given xs, a list of integers. Then, (sort xs) evaluates (lazily) to a list without deadlocking.

PROOF By induction on #xs.

BASE CASE #xs = 0, ie. xs = nil. Trivial, since (sort nil) = nil.

INDUCTIVE CASE Suppose true for #xs ≤ n.

Consider (sort (cons x xs))

```
      = (var ys;  sort (partition x xs ys) ++ cons x (sort ys))
```

By Lemma, (partition x xs ys) evaluates (lazily) to a list, ps say, without deadlocking, and, by inductive hypothesis, (sort ps) evaluates to a list without deadlocking, since #ps $\leq$ #xs = n. ... (1)

Now, in evaluating (***E*** ++ ***E'***), ***E*** is demanded first. So, (sort (partition x xs ys)) does indeed get evaluated.

Of course, we know that even just to determine the first element (or, more precisely, the first cons cell) of the sorted list, the entire list of integers being sorted must be examined. Thus, (partition x xs ys) must get fully evaluated to a list.

Then, by the Lemma, ys is always instantiated to a list, prior to any evaluation of (sort ys).

But #ys $\leq$ #xs = n, so by inductive hypothesis (sort ys) evaluates to a list without deadlocking. ... (2)

From (1) and (2), we have that (sort xs) evaluates (lazily) to a list without deadlocking.

$\square$

3.4 SIDE-EFFECTS AS AN ALTERNATIVE TO CONTINUATIONS

In transformational programming many improvements made possible by tupling can also be achieved by using continuations. In general, the eureka step is to take some function f, and define a new function f' of the form

$f'\ x\ \theta = \theta\ (f\ x)$

where θ is a continuation. Just as there is a strategy involving side-effects that seems closely related to tupling, so there is a second strategy that resembles the use of continuations. In this approach, the function f' is defined by

$f'\ x$ (<u>var</u> y) t = <u>assign</u> $y = f\ x\ ;\ t$

Here, t acts as a continuation that has already been applied to its argument. The use of these strategies will be demonstrated on a couple of examples in the remainder of this section. Note that solutions involving side-effects will be developed ***without*** using the move-in transformation.

Consider once more the Fibonacci function.

<u>CONTINUATION STRATEGY</u>

Define

```
fib' : Int → (Int → Int → Int) → Int
fib' n θ = θ (fib (n+1)) (fib n)
```

so that fib can be redefined by

```
fib (n+1) = (λx y. x) (fib (n+1)) (fib n)
          = fib' n (λx y. x)
                        folding with definition of fib'
```

Giving an inductive definition to fib':-

```
fib' 0 θ = θ (fib 1) (fib 0)
         = θ 1 1
              unfolding with definition of fib
```

```
fib' (n+1) θ = θ (fib (n+2)) (fib (n+1))
             = θ (fib (n+1) + fib n)(fib (n+1))
                                       unfolding with definition of fib
             = (λy z. θ (y+z) y) (fib (n+1)) (fib n)
             = fib' n (λy z. θ (y+z) y)
                        folding with definition of fib'
```

<u>SOLUTION</u>

```
fib : Int → Int
fib  0    = 1
fib (n+1) = fib' n (λx y. x)
```

```
fib' : Int → (Int → Int → Int) → Int
fib'  0    θ = θ 1 1
fib' (n+1) θ = fib' n (λy z. θ (y+z) y)
```

<u>SIDE-EFFECTS STRATEGY</u>

Define

```
fib' : Int → Int → Int → Int → Int
fib' n (var x) (var y) t = assign x = fib (n+1), y = fib n ; t
```

Now,

```
fib (n+1) = var x, y ; assign x = fib (n+1), y = fib n ; x
                        introducing assigned variables
          = var x, y ; fib' n x y x
                        folding with definition of fib'
```

Synthesize a new definition of fib'.

```
fib' 0 (var x) (var y) t = assign x = fib 1, y = fib 0 ; t
                         = assign x = 1, y = 1 ; t
```

```
  fib' (n+1) (var x) (var y) t
= assign x = fib (n+2), y = fib (n+1) ; t
= assign x = fib (n+1) + fib n, y = fib (n+1) ; t
= var z ; assign x = y + z, y = fib (n+1), z = fib n ; t
              introducing an assigned variable
= var z ; assign x = y + z : fib' n y z t
              folding with definition of fib'
```

SOLUTION

```
fib : Int → Int
fib  0    = 1
fib (n+1) = var x, y ; fib' n x y x
```

```
fib' : Int → Int → Int → Int → Int
fib'  0    (var x) (var y) t = assign x = 1, y = 1 ; t
fib' (n+1) (var x) (var y) t = var z ; assign x = y + z ;
                               fib' n y z t
```

As a second example, consider Example 7 of Chapter 2.

```
sort : List Int → List Int
sort = sort' nil
```

```
sort' : List Int → List Int → List Int
sort' rs nil         = rs
sort' rs (cons x xs) = sort' (cons x (sort' rs [y←xs|y≥x]))
                             [y←xs|y<x]
```

CONTINUATION STRATEGY

Define

```
partition : Int → List Int →
                  (List Int → List Int → List Int) → List Int
partition x xs θ = θ [y←xs|y<x] [y←xs|y≥x]
```

so that

```
sort' rs (cons x xs) = sort' (cons x (sort' rs [y←xs|y≥x]))
                             [y←xs|y<x]
                                        by definition
                     = (λus vs. sort' (cons x (sort' rs vs)) us)
                       [y←xs|y<x] [y←xs|y≥x]
                     = partition x xs
                       (λus vs. sort' (cons x (sort' rs vs)) us)
                                        by folding
```

It is straightforward to synthesize an efficient version of partition.

Thus:

```
partition x nil θ = θ [y←nil|y<x] [y←nil|y≥x]
                           instantiating xs to nil
                  = θ nil nil
```

```
partition x (cons z zs) θ
    = θ [y←zs|y<x] (cons z [y←zs|y≥x]) if x≤z
      θ (cons z [y←zs|y<x]) [y←zs|y≥x] otherwise
                      instantiating xs to (cons z zs) and simplifying
    = (λus vs. θ us (cons z vs)) [y←zs|y<x] [y←zs|y≥x] if x≤z
      (λus vs. θ (cons z us) vs) [y←zs|y<x] [y←zs|y≥x] otherwise
    = partition x zs (λus vs. θ us (cons z vs)) if x≤z
      partition x zs (λus vs. θ (cons z us) vs) otherwise
                      folding with definition of partition
```

<u>SOLUTION</u>

```
sort' : List Int → List Int → List Int
sort' rs  nil         = rs
sort' rs (cons x xs) = partition x xs
                        (λus vs. sort' (cons x (sort' rs vs)) us)
```

```
partition : Int → List Int →
                     (List Int → List Int → List Int) → List Int
partition x  nil θ = θ nil nil
partition x (cons z zs) θ
        = partition x zs (λus vs. θ us (cons z vs)) if x≤z
          partition x zs (λus vs. θ (cons z us) vs) otherwise
```

<u>SIDE-EFFECTS STRATEGY</u>

The eureka step is as follows:-

```
partition : Int → List Int →
                     List Int → List Int → List Int → List Int
partition x xs (var us) (var vs) ws
        = assign us = [y←xs|y<x], vs = [y←xs|y≥x] ; ws
```

Thus, sort' can be redefined by

```
sort' rs (cons x xs) = sort' (cons x (sort' rs [y←xs|y≥x]))
                             [y←xs|y<x]
                     = var us, vs ;
                       assign us = [y←xs|y<x], vs = [y←xs|y≥x] ;
                       sort' (cons x (sort' rs vs)) us
                             introducing assigned variables
                     = var us, vs ;
                       partition x xs us vs
                       (sort' (cons x (sort' rs vs)) us)
                             folding with definition of partition
```

Specializing the definition of partition to the various cases, gives:-

CASE xs = nil

```
partition x nil (var us) (var vs) ws
        = assign us = [y←nil|y<x], vs = [y←nil|y≥x] ; ws
        = assign us = nil, vs = nil ; ws
```

CASE xs = cons z zs with x≤z

```
partition x (cons z zs) (var us) (var vs) ws
        = assign us = [y←zs|y<x], vs = cons z [y←zs|y≥x] ; ws
        = var vs' ; assign vs = cons z vs', us = [y←zs|y<x],
                           vs' = [y←zs|y≥x] ; ws
                              introducing an assigned variable
        = var vs' ; assign vs = cons z vs' ;
          partition x zs us vs' ws
                              folding with definition of partition
```

CASE xs = cons z zs with x>z

```
partition x (cons z zs) (var us) (var vs) ws
        = assign us = cons z [y←zs|y<x], vs = [y←zs|y≥x] ; ws
        = var us' ; assign us = cons z us', us' = [y←zs| y<x],
                             vs = [y←zs|y≥x] ; ws
                              introducing an assigned variable
        = var us' ; assign us = cons z us' ;
          partition x zs us' vs ws
                              folding with definition of partition
```

SOLUTION

```
sort' : List Int → List Int → List Int
sort' rs nil          = rs
sort' rs (cons x xs) = var us, vs ;
                        partition x xs us vs
                        (sort' (cons x (sort' rs vs)) us)


partition : Int → List Int →
                    List Int → List Int → List Int → List Int
partition x  nil (var us) (var vs) ws
        = assign us = nil, vs = nil ; ws
partition x (cons z zs) (var us) (var vs) ws
        = var vs' ; assign vs = cons z vs' ;
          partition x zs us vs' ws              if x≤z
          var us' ; assign us = cons z us' ;
          partition x zs us' vs ws              otherwise
```

3.5 CONCLUSION

In this chapter we have demonstrated that transformational programming can be used in the development of programs with side-effects from purely functional programs. We have identified two new transformation rules, introducing assigned variables and the move-in transformation, which are unique to programming with side-effects. Furthermore, strategies for developing programs with side-effects have been presented and have been compared with well-known strategies involving tuples and continuations on a small selection of problems.

By adopting this formal methodology it has proven possible to isolate those steps in the development process at which there is a danger that deadlock might be introduced. This simplifies the task of ensuring that programs are free from deadlock. Control annotations (especially par [29,30]) can sometimes be useful in developing deadlock-free programs. An attempt has been made to outline the sort of reasoning that is required for that purpose and mention has been made of various problems that might arise. To summarise: deadlock-free programs with side-effects can be developed by a transformational programmer who has a good understanding of order of evaluation.

CHAPTER 4

DENOTATIONAL SEMANTICS

4.1 INTRODUCTION

This chapter provides a formal semantic description of a functional programming language with side-effects. The description takes the form of a *denotational semantics* for the language. That is, [51]

> 'We give "semantic valuation functions", which map syntactic constructs in the program to the abstract values (numbers, truth values, functions etc.) which they denote. These valuation functions are usually recursively defined: the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents ...'

A valuation $\mathcal{E}$ is defined which gives a meaning to any well-formed expression. $\mathcal{E}$ can be thought of as an abstract model for a graph reduction machine. It gives call-by-need semantics [60] to function application and provides for lazy constructors [19]. A straightforward meaning is given to the <u>var</u> and <u>assign</u> constructs in functional programs with side-effects.

The chapter is organized as follows. It begins with an informal description of the process of graph reduction. This has been provided so that additional insight may be gained into the semantic equations of Section 4.3. In that section the syntax and semantics of the programming language are given. Simple proofs of the equivalence of some expressions are also demonstrated.

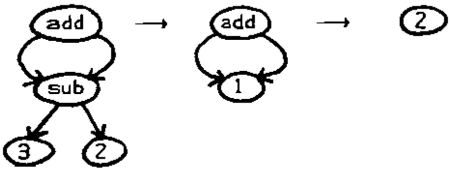
4.2 GRAPH REDUCTION

Graph reduction, as a method by which functional programming languages can be implemented, was invented by Wadsworth [60] and popularised by Turner [54]. In this section we shall give a brief description of the process; further explanation, as well as details of optimisations, can be found in [27,31,43,52,55].

The key idea is that expressions are represented in graphical form. Reduction rules can then be performed on these *expression-graphs* in such a way that explicit copying of subexpressions can be avoided. For example, in the sequence of reductions

```
(λx. add x x) (sub 3 2) ⟶ add (sub 3 2) (sub 3 2)
                        ⟶ add 1 1
                        ⟶ 2
```

the expression (sub 3 2) is not copied. Instead, an instance of the body of the λ-expression is created, with *pointers* to (sub 3 2) substituted for the two occurrences of x. When add evaluates one of its arguments the expression (sub 3 2) is *over-written* by its value 1. Hence, the reduction does not have to be repeated for the second argument. This is the 'simultaneous contraction of redexes' noted in [60]. Thus:



Evaluation of an expression involves the repeated application of reduction rules until it has been transformed into ***head normal form***¹ (sometimes called ***canonical form*** or ***lazy normal form***): that is to say, a λ-expression, a partially applied function or a data structure, such as a number or (cons E_1 E_2). Thus, the expressions (λx. add 3 5), (add 2), true and (cons (add 3 5) 2) are all in head normal form; whereas, (add 3 5) itself is not.

Graph reduction machines employ a ***normal order*** reduction strategy. That is, (E_1 E_2) is evaluated by first evaluating E_1 and then applying the resulting function to E_2. By the Church-Rosser Theorem this ensures that a head normal form can be found whenever one exists.

A rather subtle point is that, although normal order graph reduction implements lazy evaluation, it does not achieve ***full laziness*** [28,29]. This simply means that a subexpression in the body E of a λ-expression (λx.E) may be re-evaluated each time the λ-expression is applied, even if it contains no occurrences of x. Thus, (add 3 5) is re-evaluated whenever (λx. add 3 5) is applied to some argument. It is therefore prudent to transform or compile (λx. add 3 5) into an equivalent expression such as ((λy x. y) (add 3 5)), before performing graph

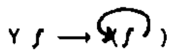
¹[H P Barendregt. *The lambda calculus: its syntax and semantics*. North-Holland 1984] uses the term *head normal form* to mean something slightly different.

reduction. Because a pointer to (add 3 5) is substituted for y, the expression ((λy x. y) (add 3 5)) reduces to (λx. 8) on being applied to an expression.

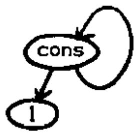
So far we have seen that graphical-expressions facilitate the sharing of subexpressions. A new possibility is also opened up, that of building *circular* structures. Turner [54] recognized that an efficient circular version of the Paradoxical Combinator Y can be given: the rule

$$Y\ f \longrightarrow f\ (Y\ f)$$

is replaced by



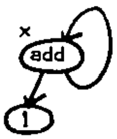
Circular structures also allow for a convenient representation of certain *infinite* data structures. For example,



represents an infinite list of ones. A further point is that some 'silly recursions' can be detected. For instance,

```
x : Int
x = add 1 x
```

can be represented by the graph



The node x can be marked as *not ready* while the value of x is being determined. Thus, when add demands the value of its second argument, the infinite loop can be detected.

Finally, we note that graph reduction machines are usually driven by a printing routine; which for example, when reducing an expression to a list, will force evaluation of all members of the list.

The above properties of the graph reduction method of computation are reflected by the semantic equations of the next section.

4.3 DENOTATIONAL SEMANTICS

In this section, we first give the syntax of the programming language. This is followed by the semantic domains and equations. Finally, explanation of the equations is provided and the semantics is employed to prove the equivalence of some expréssions.

SYNTACTIC CATEGORIES

$x \in$ **Ide**	identifiers (variables)
$p \in$ **Prim**	primitives
$E \in$ **Exp**	expressions

SYNTAX

$E ::= x \mid p \mid E\ E \mid \lambda x.E \mid \underline{\text{var}}\ x\ ;\ E \mid \underline{\text{assign}}\ x = E\ ;\ E$

We will assume that cons $\in$ **Prim** is the only constructor of positive arity. Though, further constructors could be accommodated by making suitable extensions to the semantic domains. (A deficiency of the language is that it does not support the direct definition of functions by pattern-matching. However, a method for transforming pattern-matching into simpler constructs can be found in [1].)

SEMANTIC DOMAINS

$\mathbf{Bv} = \mathbf{Bool} + \mathbf{Num} + \{\underline{nil}\}$	basic values
$A \in \mathbf{Ans} = \mathbf{Bv} + (\mathbf{Ans} \times \mathbf{Ans}) + \{\underline{suspend}, \underline{error}\}$	answers
$\alpha \in \mathbf{Loc}$	locations
$\varrho \in \mathbf{Env} = \mathbf{Ide} \rightarrow \mathbf{Loc}$	environments
$\sigma \in \mathbf{Store} = \mathbf{Loc} \rightharpoonup \mathbf{Sv}$	stores
$\theta \in \mathbf{Cont} = \mathbf{Store} \rightarrow (\mathbf{Ans} \times \mathbf{Store})$	continuations
$\kappa \in \mathbf{ECont} = \mathbf{Ev} \rightharpoonup \mathbf{Cont}$	expression continuations
$\nu \in \mathbf{Clo} = \mathbf{ECont} \rightarrow \mathbf{Cont}$	closures
$\epsilon \in \mathbf{Ev} = \mathbf{Bv} + (\mathbf{Loc} \times \mathbf{Loc}) + (\mathbf{Loc} \rightarrow \mathbf{Clo})$	expressed values
$\mathbf{Sv} = \mathbf{Ev} + \mathbf{Clo} + \{\underline{unset}, \underline{NotReady}, \underline{free}\}$	stored values

AUXILIARY FUNCTIONS

new : $\mathbf{Store} \rightharpoonup \mathbf{Loc}$, satisfies $\sigma(new\ \sigma) = \underline{free}$

wrong : **Cont** , is defined by $wrong\ \sigma = (\underline{error}\ .\ \sigma)$

print : $\mathbf{Ev} \rightharpoonup \mathbf{Cont}$

$$
\begin{aligned}
print\ \epsilon = {} & \epsilon \sqsubseteq (\mathbf{Loc} \rightarrow \mathbf{Clo}) \longrightarrow wrong\,; \\
& \epsilon \sqsubseteq (\mathbf{Loc} \times \mathbf{Loc}) \longrightarrow \lambda\sigma.\ ((A_1\ .\ A_2)\ .\ \sigma_2) \\
& \qquad\qquad \underline{where}\ (\alpha_1\ .\ \alpha_2) = \epsilon \\
& \qquad\qquad\qquad (A_1\ .\ \sigma_1) = force\ \alpha_1\ print\ \sigma \\
& \qquad\qquad\qquad (A_2\ .\ \sigma_2) = force\ \alpha_2\ print\ \sigma_1\ ; \\
& \epsilon \sqsubseteq \mathbf{Bv} \longrightarrow \lambda\sigma.\ (\epsilon\ .\ \sigma)
\end{aligned}
$$

SEMANTIC FUNCTIONS

A typical computation will be

$$\mathcal{E}[\![E]\!]\,\varrho_0\,\kappa_0\,\sigma_0 \ \underline{where}\ \kappa_0 = print$$

E is the expression to be evaluated (and printed); ϱ_0 is the arid environment, in which no variables have been declared; σ_0 is the empty store, in which all locations are *free* . A well-formedness condition on E is that it may only contain variables declared within var or λ constructs.

The valuation $\mathcal{E}$ is defined as follows:-

$$\mathcal{E} : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{ECont} \to \mathbf{Cont}$$

$$\mathcal{E}[\![x]\!]\varrho\kappa = force\,(\varrho[\![x]\!])\,\kappa$$

$$\mathcal{E}[\![p]\!]\varrho\kappa = \kappa\,(\mathcal{P}[\![p]\!])$$

$$\mathcal{E}[\![E_1\ E_2]\!]\varrho\kappa = \mathcal{E}[\![E_1]\!]\varrho\kappa' \ \underline{where}\ \kappa' = \mathcal{A}[\![E_2]\!]\varrho\kappa$$

$$\mathcal{E}[\![\lambda x.\,E]\!]\varrho\kappa = \kappa\,(\lambda\alpha.\,\mathcal{E}[\![E]\!](\varrho \oplus \{x \mapsto \alpha\}))$$

$$\mathcal{E}[\![\underline{\text{var}}\ x\ ;\ E]\!]\varrho\kappa = \lambda\sigma.\,\mathcal{E}[\![E]\!](\varrho \oplus \{x \mapsto \alpha\})\,\kappa\,(\sigma \oplus \{\alpha \mapsto \underline{unset}\})$$
$$\underline{where}\ \alpha = new\ \sigma$$

$$\mathcal{E}[\![\underline{\text{assign}}\ x = E_1\ ;\ E_2]\!]\varrho\kappa = \lambda\sigma.\ (\sigma\alpha = \underline{unset} \longrightarrow$$
$$\mathcal{E}[\![E_2]\!]\varrho\kappa(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho\})\ .$$
$$(\underline{error}\ ,\ \sigma)\ )$$
$$\underline{where}\ \alpha = \varrho[\![x]\!]$$

$$force : \mathbf{Loc} \to \mathbf{ECont} \to \mathbf{Cont}$$

$$force\ \alpha\,\kappa = \lambda\sigma.\ \underline{case}\ \sigma\alpha\ \underline{of}$$
$$\underline{NotReady}\ .\ \underline{unset} : (\underline{suspend}\ ,\ \sigma)$$
$$\epsilon : \kappa\,\epsilon\,\sigma$$
$$\nu : \nu\,\kappa'\,(\sigma \oplus \{\alpha \mapsto \underline{NotReady}\})$$
$$\underline{where}\ \kappa' = \lambda\epsilon\,\sigma'.\ \kappa\,\epsilon\,(\sigma' \oplus \{\alpha \mapsto \epsilon\})$$
$$\underline{end}$$

$$
\begin{aligned}
&\mathcal{A} : \mathbf{Exp} \to \mathbf{Env} \to \mathbf{ECont} \to \mathbf{ECont}\\
&\mathcal{A}[\![E]\!]\varrho\kappa = \lambda\epsilon.\ \epsilon \in (\mathbf{Loc} \to \mathbf{Clo}) \longrightarrow \theta\ ,\ \mathit{wrong}\\
&\qquad\qquad \underline{\mathit{where}}\ \theta = E \in \mathbf{Ide} \longrightarrow \theta_1\ ,\ \theta_2\\
&\qquad\qquad\qquad \theta_1 = \epsilon\,(\varrho[\![E]\!])\,\kappa\\
&\qquad\qquad\qquad \theta_2 = \lambda\sigma.\ \epsilon\,\alpha\,\kappa\,\sigma'\\
&\qquad\qquad\qquad\qquad \underline{\mathit{where}}\ \alpha = \mathit{new}\ \sigma\\
&\qquad\qquad\qquad\qquad\qquad \sigma' = \sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E]\!]\varrho\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{P} : \mathbf{Prim} \to \mathbf{Ev}\\
&\mathcal{P}[\![\mathtt{true}]\!] = \underline{\mathit{true}}\\
&\mathcal{P}[\![\mathtt{nil}]\!] = \underline{\mathit{nil}}\\
&\mathcal{P}[\![\mathtt{not}]\!] = \lambda\alpha\kappa.\ \mathit{force}\,\alpha\,(\lambda\epsilon.\ \epsilon \in \mathbf{Bool} \longrightarrow \kappa\ (\neg\epsilon)\ ,\ \mathit{wrong})\\
&\mathcal{P}[\![\mathtt{head}]\!] = \lambda\alpha\kappa.\ \mathit{force}\,\alpha\,(\lambda\epsilon.\ \epsilon \in (\mathbf{Loc} \times \mathbf{Loc}) \longrightarrow\\
&\qquad\qquad\qquad \mathit{force}\,\alpha_1\,\kappa\ \underline{\mathit{where}}\ (\alpha_1, \alpha_2) = \epsilon\ \ ,\ \mathit{wrong})\\
&\mathcal{P}[\![\mathtt{Y}]\!] = \lambda\alpha\kappa.\ \mathit{force}\,\alpha\,(\lambda\epsilon.\ \epsilon \in (\mathbf{Loc} \to \mathbf{Clo}) \longrightarrow\\
&\qquad\qquad \lambda\sigma.\ \mathit{force}\,\alpha'\,\kappa\,\sigma'\ \ \underline{\mathit{where}}\ \alpha' = \mathit{new}\ \sigma\\
&\qquad\qquad\qquad\qquad \sigma' = \sigma \oplus \{\alpha' \mapsto \epsilon\,\alpha'\}\ ,\\
&\qquad\qquad \mathit{wrong})\\
&\mathcal{P}[\![\mathtt{cons}]\!] = \lambda\alpha\kappa.\ \kappa\,(\mathit{cons}\ \alpha)\\
&\mathcal{P}[\![\mathtt{add}]\!] = \lambda\alpha\kappa.\ \kappa\,(\mathit{add}\ \alpha)\\
&\mathcal{P}[\![\mathtt{div}]\!] = \lambda\alpha\kappa.\ \kappa\,(\mathit{div}\ \alpha)\\
&\mathcal{P}[\![\mathtt{val}]\!] = \lambda\alpha\kappa.\ \kappa\,(\mathit{val}\ \alpha)\\
&\mathcal{P}[\![\mathtt{seq}]\!] = \lambda\alpha\kappa.\ \kappa\,(\mathit{seq}\ \alpha)\\
&\mathcal{P}[\![\mathtt{if}]\!] = \lambda\alpha\kappa.\ \kappa\,(\mathit{if}\ \alpha)
\end{aligned}
$$

etc.

$$
\begin{aligned}
cons &= \lambda\alpha_1\alpha_2\kappa.\ \kappa\,(\alpha_1, \alpha_2)\\
add &= \lambda\alpha_1\alpha_2\kappa.\ force\,\alpha_1\,(\lambda\epsilon_1.\ \epsilon_1 \mathrel{E} \mathbf{Num} \longrightarrow \theta\ ,\ wrong\\
&\qquad\qquad \underline{where}\ \ \theta = force\,\alpha_2\,(\lambda\epsilon_2.\ \epsilon_2 \mathrel{E} \mathbf{Num} \longrightarrow\\
&\qquad\qquad\qquad \kappa\,(\epsilon_1 + \epsilon_2)\ ,\ wrong)\ )\\
div &= \lambda\alpha_1\alpha_2\kappa.\ force\,\alpha_1\,(\lambda\epsilon_1.\ \epsilon_1 \mathrel{E} \mathbf{Num} \longrightarrow \theta\ ,\ wrong\\
&\qquad\qquad \underline{where}\ \ \theta = force\,\alpha_2\,(\lambda\epsilon_2.\ (\epsilon_2 \mathrel{E} \mathbf{Num}) \wedge (\epsilon_2 \neq 0) \longrightarrow\\
&\qquad\qquad\qquad \kappa\,(\epsilon_1 \div \epsilon_2)\ ,\ wrong)\ )\\
val &= \lambda\alpha_1\alpha_2\kappa.\ force\,\alpha_2\,(\lambda\epsilon.\ force\,\alpha_1\,(\lambda\epsilon_1.\ \epsilon_1 \mathrel{E} (\mathbf{Loc} \to \mathbf{Clo}) \longrightarrow \epsilon_1\,\alpha_2\kappa\ .\ wrong))\\
seq &= \lambda\alpha_1\alpha_2\kappa.\ force\,\alpha_1\,(\lambda\epsilon.\ force\,\alpha_2\,\kappa)\\
if &= \lambda\alpha_1\alpha_2\kappa.\ \kappa\,(if'\,\alpha_1\,\alpha_2)\\
if' &= \lambda\alpha_1\alpha_2\alpha_3\kappa.\ force\,\alpha_1\,(\lambda\epsilon.\ \epsilon \mathrel{E} \mathbf{Bool} \longrightarrow (\epsilon \longrightarrow force\,\alpha_2\,\kappa\ ,\ force\,\alpha_3\,\kappa),\ wrong)
\end{aligned}
$$

COMMENTS

An expressed value ϵ corresponds to the head normal form of an expression E. Since the store may change, from σ to σ' say, during the evaluation of E, the rest of the computation is modelled by an expression continuation κ, such that

$$\mathcal{E}[\![E]\!]\rho\kappa\sigma = \kappa\epsilon\sigma'$$

The equations for var and assign correspond closely to their informal description given in Chapter 2, Section 2.3.

A closure ν represents an expression that has been stored in unevaluated form. The value ϵ of such an expression will only be computed if demanded. Propagation of demand to a location α in the store is modelled by the function *force*. Thus, by forcing a closure stored at α, the contents ν of α are replaced by ϵ. During the computation of ϵ, α is marked as not ready. This method implements lazy evaluation, because once ν has been replaced by ϵ, demands sent to α simply return ϵ. Thus:

<u>PROPOSITION</u> $force\,\alpha\,(\lambda\varepsilon.\ force\,\alpha\,\kappa) = force\,\alpha\,\kappa$

<u>PROOF</u> Show $force\,\alpha\,(\lambda\varepsilon.\ force\,\alpha\,\kappa)\sigma = force\,\alpha\,\kappa\,\sigma$ by case analysis on $\sigma\,\alpha$.
Trivial for $\sigma\,\alpha$ equal to $\underline{NotReady}$ or $\underline{unset}$.

$$\varepsilon = \sigma\alpha \Rightarrow force\,\alpha\,(\lambda\varepsilon.\ force\,\alpha\,\kappa)\sigma = (\lambda\varepsilon.\ force\,\alpha\,\kappa)\,\varepsilon\,\sigma = force\,\alpha\,\kappa\,\sigma$$

$$\begin{aligned}\nu = \sigma\alpha \Rightarrow\ & force\,\alpha\,(\lambda\varepsilon.\ force\,\alpha\,\kappa)\,\sigma \\ &= \nu\ (\lambda\varepsilon\,\sigma'.\ (\lambda\varepsilon.\ force\,\alpha\,\kappa)\,\varepsilon\,(\sigma' \oplus \{\alpha \mapsto \varepsilon\}))\,(\sigma \oplus \{\alpha \mapsto \underline{NotReady}\}) \\ &= \nu\ (\lambda\varepsilon\,\sigma'.\ force\,\alpha\,\kappa\,(\sigma' \oplus \{\alpha \mapsto \varepsilon\}))\,(\sigma \oplus \{\alpha \mapsto \underline{NotReady}\}) \\ &= \nu\ (\lambda\varepsilon\,\sigma'.\ \kappa\,\varepsilon\,(\sigma' \oplus \{\alpha \mapsto \varepsilon\}))\,(\sigma \oplus \{\alpha \mapsto \underline{NotReady}\}) \\ &= force\,\alpha\,\kappa\,\sigma \qquad \Box\end{aligned}$$

The expression continuation $\mathcal{A}[\![E]\!]\rho\kappa$ takes a function and applies it to E. To preserve laziness, if E is not an identifier then it is stored as a closure at some location α and the function is actually applied to α. The special treatment of identifiers avoids unnecessary ***indirection*** [54] which would complicate the semantics of <u>assign</u>. For example, evaluation of $((\lambda x.\ \underline{\texttt{assign}}\ x = E_1\ ;\ E_2)\ y)$, in some environment ρ, must bind x to $\rho[\![y]\!]$, rather than some location at which $\mathcal{E}[\![y]\!]\rho$ is stored, so as to handle the side-effect properly.

The valuation $\mathcal{P}$ gives values to the primitive functions of the language. In particular, partially applied functions take values in $(\mathbf{Loc} \to \mathbf{Clo})$.

Evaluation of an expression results in an error in certain circumstances, namely:-

(i) Evaluation of $(E_1\ E_2)$ gives rise to an error if E_1 does not evaluate to a function. For example, an error arises from evaluating `(cons nil 4 6)`, because `(cons nil 4)` denotes an S-expression rather than a function.

(ii) During evaluation an attempt is made to assign to a variable that has already been instantiated. This reflects the fact that <u>assign</u> is a single-assignment construct.

(iii) A primitive function is applied to arguments for which it is undefined, for example (div 4 0).

(iv) An attempt is made to print a function. Only S-expressions can be output as answers.

Evaluation of an expression suspends when

(i) There is a demand for the value of an uninstantiated variable, that is, one which is bound to an ***unset*** location.

(ii) The value of an expression that is already under evaluation is required. This is detected by marking a location as ***NotReady*** while evaluating it contents.

The reason for distinguishing between ***suspend*** and ***error*** is that, on an implementation that supports parallel evaluation of expressions, a suspended computation will resume if and when an expression is stored at the ***unset***/***NotReady*** location. [32] refers to this as 'suicidal suspensions'. However, in the semantics presented here, no attempt is made to model parallel evaluation.

Finally, we demonstrate the equivalence of some expressions. The first proposition proves that

$$\begin{aligned} &(\underline{\texttt{assign}}\ x = E_1\ ;\ \underline{\texttt{assign}}\ y = E_2\ :\ E) \\ = &(\underline{\texttt{assign}}\ y = E_2\ ;\ \underline{\texttt{assign}}\ x = E_1\ ;\ E) \end{aligned}$$

The other propositions illustrate safe instances of the move-in transformation. It should be noted that many equivalences cannot be proven so simply; whether or not an expression is actually stored and, if so, the particular location used, may be reflected by the equations, but not be of semantic importance.

<u>Proposition</u> Suppose $\alpha = \varrho[\![x]\!]$, $\beta = \varrho[\![y]\!]$, $\alpha \neq \beta$ and $\sigma\,\alpha = \sigma\,\beta = \underline{\mathit{unset}}$. Then

$$
\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ x = E_1\ ;\ \underline{\texttt{assign}}\ y = E_2\ ;\ E]\!]\varrho\kappa\sigma \\
= {}&\mathcal{E}[\![\underline{\texttt{assign}}\ y = E_2\ ;\ \underline{\texttt{assign}}\ x = E_1\ ;\ E]\!]\varrho\kappa\sigma
\end{aligned}
$$

<u>Proof</u>

$$
\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ x = E_1\ ;\ \underline{\texttt{assign}}\ y = E_2\ ;\ E]\!]\varrho\kappa\sigma \\
= {}&\mathcal{E}[\![\underline{\texttt{assign}}\ y = E_2\ ;\ E]\!]\varrho\kappa(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho\}) \\
= {}&\mathcal{E}[\![E]\!]\varrho\kappa(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \mathcal{E}[\![E_2]\!]\varrho\})
\end{aligned}
$$

and similarly for $\mathcal{E}[\![\underline{\texttt{assign}}\ y = E_2\ ;\ \underline{\texttt{assign}}\ x = E_1\ ;\ E]\!]\varrho\kappa\sigma$. $\Box$

<u>Proposition</u> Suppose $\alpha = \varrho[\![x]\!]$, and $\sigma\,\alpha = \underline{\mathit{unset}}$. Then

$$
\mathcal{E}[\![\underline{\texttt{assign}}\ x = E\ ;\ (E_1\ E_2)]\!]\varrho\kappa\sigma = \mathcal{E}[\![(\underline{\texttt{assign}}\ x = E\ ;\ E_1)\ E_2]\!]\varrho\kappa\sigma
$$

<u>Proof</u>

$$
\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ x = E\ ;\ (E_1\ E_2)]\!]\varrho\kappa\sigma \\
= {}&\mathcal{E}[\![E_1\ E_2]\!]\varrho\kappa(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E]\!]\varrho\}) \\
= {}&\mathcal{E}[\![E_1]\!]\varrho(\mathcal{A}[\![E_2]\!]\varrho\kappa)(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E]\!]\varrho\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![(\underline{\texttt{assign}}\ x = E\ ;\ E_1)\ E_2]\!]\varrho\kappa\sigma \\
= {}&\mathcal{E}[\![\underline{\texttt{assign}}\ x = E\ ;\ E_1]\!]\varrho(\mathcal{A}[\![E_2]\!]\varrho\kappa)\sigma \\
= {}&\mathcal{E}[\![E_1]\!]\varrho(\mathcal{A}[\![E_2]\!]\varrho\kappa)(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E]\!]\varrho\})
\end{aligned}
$$

$\Box$

<u>Proposition</u> Suppose $\alpha = \varrho[\![x]\!]$, $\beta = \varrho[\![y]\!]$, $\alpha \neq \beta$ and $\sigma\,\alpha = \sigma\,\beta = \underline{\mathit{unset}}$. Then

$$
\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ x = E_1\ ;\ \underline{\texttt{assign}}\ y = E_2\ ;\ y]\!]\varrho\kappa\sigma \\
= {}&\mathcal{E}[\![\underline{\texttt{assign}}\ y = (\underline{\texttt{assign}}\ x = E_1\ ;\ E_2)\ ;\ y]\!]\varrho\kappa\sigma
\end{aligned}
$$

<u>Proof</u>

$$
\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ x = E_1\ ;\ \underline{\texttt{assign}}\ y = E_2\ ;\ y]\!]\varrho\kappa\sigma \\
= {}&\mathcal{E}[\![y]\!]\varrho\kappa(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \mathcal{E}[\![E_2]\!]\varrho\}) \\
= {}&\mathit{force}\ \beta\,\kappa\,(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \mathcal{E}[\![E_2]\!]\varrho\}) \\
= {}&\mathcal{E}[\![E_2]\!]\varrho(\lambda\varepsilon\,\sigma'.\ \kappa\,\varepsilon\,(\sigma' \oplus \{\beta \mapsto \varepsilon\}))(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \underline{\mathit{NotReady}}\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ y = (\underline{\texttt{assign}}\ x = E_1\ ;\ E_2)\ ;\ y]\!]\varrho\kappa\sigma \\
= {}&\mathcal{E}[\![y]\!]\varrho\kappa(\sigma \oplus \{\beta \mapsto \mathcal{E}[\![\underline{\texttt{assign}}\ x = E_1\ ;\ E_2]\!]\varrho\}) \\
= {}&\mathit{force}\ \beta\,\kappa\,(\sigma \oplus \{\beta \mapsto \mathcal{E}[\![\underline{\texttt{assign}}\ x = E_1\ ;\ E_2]\!]\varrho\}) \\
= {}&\mathcal{E}[\![\underline{\texttt{assign}}\ x = E_1\ ;\ E_2]\!]\varrho(\lambda\varepsilon\,\sigma'.\ \kappa\,\varepsilon\,(\sigma' \oplus \{\beta \mapsto \varepsilon\}))(\sigma \oplus \{\beta \mapsto \underline{\mathit{NotReady}}\}) \\
= {}&\mathcal{E}[\![E_2]\!]\varrho(\lambda\varepsilon\,\sigma'.\ \kappa\,\varepsilon\,(\sigma' \oplus \{\beta \mapsto \varepsilon\}))(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \underline{\mathit{NotReady}}\})
\end{aligned}
$$

$\Box$

<u>PROPOSITION</u> Suppose E_2 is not an identifier, $\alpha = \varrho[\![x]\!]$, and $\sigma\,\alpha = \underline{unset}$. Then

$$\mathcal{E}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ (\texttt{not}\ E_2)]\!]\varrho\kappa\sigma = \mathcal{E}[\![\texttt{not}\ (\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2)]\!]\varrho\kappa\sigma$$

<u>PROOF</u>

$$\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ (\texttt{not}\ E_2)]\!]\varrho\kappa\sigma\\
&= \mathcal{E}[\![\texttt{not}\ E_2]\!]\varrho\kappa(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho\})\\
&= \mathcal{E}[\![\texttt{not}]\!]\varrho(\mathcal{A}[\![E_2]\!]\varrho\kappa)(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho\})\\
&= \mathcal{A}[\![E_2]\!]\varrho\kappa(\mathcal{P}[\![\texttt{not}]\!])(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho\})\\
&= \mathcal{P}[\![\texttt{not}]\!]\beta\kappa(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \mathcal{E}[\![E_2]\!]\varrho\})\ \textit{for some}\ \beta\\
&= \textit{force}\,\beta\,(\lambda\varepsilon.\ \varepsilon \in \mathbf{Bool} \longrightarrow \kappa\,(\neg\varepsilon)\ ,\ \textit{wrong})(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \mathcal{E}[\![E_2]\!]\varrho\})\\
&= \mathcal{E}[\![E_2]\!]\varrho(\lambda\varepsilon\,\sigma'\ .\ (\lambda\varepsilon.\ \varepsilon \in \mathbf{Bool} \longrightarrow \kappa\,(\neg\varepsilon)\ ,\ \textit{wrong})\,\varepsilon\,(\sigma' \oplus \{\beta \mapsto \varepsilon\}))\\
&\qquad(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \underline{\textit{NotReady}}\})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{not}\ (\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2)]\!]\varrho\kappa\sigma\\
&= \mathcal{E}[\![\texttt{not}]\!]\varrho(\mathcal{A}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2]\!]\varrho\kappa)\sigma\\
&= \mathcal{A}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2]\!]\varrho\kappa(\mathcal{P}[\![\texttt{not}]\!])\sigma\\
&= \mathcal{P}[\![\texttt{not}]\!]\beta\kappa(\sigma \oplus \{\beta \mapsto \mathcal{E}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2]\!]\varrho\})\ \textit{for some}\ \beta\\
&= \textit{force}\,\beta\,(\lambda\varepsilon.\ \varepsilon \in \mathbf{Bool} \longrightarrow \kappa\,(\neg\varepsilon)\ ,\ \textit{wrong})\\
&\qquad(\sigma \oplus \{\beta \mapsto \mathcal{E}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2]\!]\varrho\})\\
&= \mathcal{E}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2]\!]\varrho\\
&\qquad(\lambda\varepsilon\,\sigma'\ .\ (\lambda\varepsilon.\ \varepsilon \in \mathbf{Bool} \longrightarrow \kappa\,(\neg\varepsilon)\ ,\ \textit{wrong})\,\varepsilon\,(\sigma' \oplus \{\beta \mapsto \varepsilon\}))\\
&\qquad(\sigma \oplus \{\beta \mapsto \underline{\textit{NotReady}}\})\\
&= \mathcal{E}[\![E_2]\!]\varrho(\lambda\varepsilon\,\sigma'\ .\ (\lambda\varepsilon.\ \varepsilon \in \mathbf{Bool} \longrightarrow \kappa\,(\neg\varepsilon)\ ,\ \textit{wrong})\,\varepsilon\,(\sigma' \oplus \{\beta \mapsto \varepsilon\}))\\
&\qquad(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho,\ \beta \mapsto \underline{\textit{NotReady}}\})
\end{aligned}$$

□

Although the transformation

$$(\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ (\texttt{not}\ E_2)) = (\texttt{not}\ (\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2))$$

is also perfectly safe for $E_2 \in \mathbf{Ide}$, this is not revealed by the semantic equations: $\mathcal{E}[\![\texttt{not}\ (\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ E_2)]\!]\varrho\kappa\sigma$ takes the same value as before, but now

$$\begin{aligned}
&\mathcal{E}[\![\underline{\texttt{assign}}\ x\ =\ E_1\ ;\ (\texttt{not}\ E_2)]\!]\varrho\kappa\sigma\\
&= \mathcal{E}[\![E_2]\!]\,(\lambda\varepsilon.\ \varepsilon \in \mathbf{Bool} \longrightarrow \kappa\,(\neg\varepsilon)\ ,\ \textit{wrong})\,(\sigma \oplus \{\alpha \mapsto \mathcal{E}[\![E_1]\!]\varrho\})
\end{aligned}$$

4.4 CONCLUSION

In this chapter a formal semantics has been presented for a functional language with side-effects. The semantic equations for the var and assign constructs are particularly straightforward. Because it models order of evaluation and allows for side-effects, the denotational semantics is more complicated than that usually given to purely functional languages, for example the semantics of the λ-calculus in [51]. Even so, it remains a *standard* semantics: it has not proven necessary to follow the suggestion in [38] (p.275) that semantic equations for call-by-need are best formulated in *store* semantics.

The semantics also provides a formal, abstract description of the process of graph reduction. Note that we have given a semantics *directly* to the programming language. In [21,31] the language has first to be compiled into a form suitable for a stack machine; the operation of the machine is then specified by a set of state transition rules.

CHAPTER 5

AXIOMATIC SEMANTICS

5.1 INTRODUCTION

The axiomatic semantics presented in this chapter is intended to be useful to the programmer in reasoning about programs with side-effects. Rules are given from which can be deduced some of the consequences of executing a program. The denotational semantics of Chapter 4 is rather unwieldy for such reasoning.

The axiomatic semantics provides information about the state of the graph reduction machine *during* the evaluation of an expression; this contrasts with Hoare's axiomatic semantics [23] which describes the relationship between the state before and the state after executing a command. Boehm [6] has suggested that 'one might benefit from matching the programming logic to the programming language' : our axiomatic semantics has been designed for those functional programming languages in which order of evaluation has to be taken into account. There is a superficial resemblance to Plotkin's Structured Operational Semantics [44]. Such a semantics could be given to a functional language with side-effects. However, the *entire* state of the machine would have to be considered in specifying the transition relation, a complication which we are able to avoid.

The axiomatic semantics is intended to complement the transformational approach to the development of programs with side-effects. It is a formal system based on a set of inference rules, some of which correspond to reduction rules; while others prescribe a particular order of evaluating expressions. The system does not allow us to prove very much. It does, however, describe order of evaluation in a rather concise way. Even the parallel evaluation of expressions can be specified. Note that order of evaluation must be taken into account when programming with side-effects. Otherwise, we are liable to believe that a program is correct, only to discover when we try to run it that it deadlocks. This was discussed in Example 1 of Chapter 2. To take another example, the program

```
foo : Int
foo = var x ; add x (baz x)

baz : Int → Int
baz (var y) = assign y = 1 ; 2
```

can be given the declarative reading

$$\exists x{:}\,\mathrm{Int}.\quad foo = add\ x\ (baz\ x)$$
$$\forall n,y{:}\,\mathrm{Int}.\quad n = baz\ y \Leftrightarrow y = 1 \wedge n = 2$$

Now, from this declarative reading, we are lead to believe that foo = 3, because:

1. $\exists x.$ foo = add x (baz x)
2. $\exists x, w.$ foo = add x w $\wedge$ w = baz x
3. $\exists x, w.$ foo = add x w $\wedge$ x = 1 $\wedge$ w = 2
4. foo = add 1 2
5. foo = 3

However, assuming add evaluates its arguments from left to right, evaluation of foo will in fact proceed as follows:-

$$\text{foo} \longrightarrow \text{add x (baz x)} \ \textit{\textbf{for some uninstantiated variable}} \ \text{x}$$

At this point the value of x is required. Because x is uninstantiated, evaluation suspends. That is, evaluation of foo deadlocks.

It should also be recognized that knowledge of the ***strictness*** of a function is insufficient to determine the order in which it evaluates its arguments. For example, add is strict in both its arguments:-

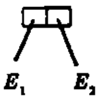
$$\text{add} \perp \boldsymbol{E} = \perp$$
$$\text{add} \ \boldsymbol{E} \perp = \perp$$

That is, it is undefined if either argument is undefined, but this provides no information as to whether add evaluates its arguments from left to right, from right to left, or even in parallel.

In the next section we present the system and give it an informal interpretation in terms of graph reduction. In Section 5.3 we explain how to reason directly about programs. It is also shown that the axiomatic semantics can be of assistance in determining whether the move-in transformation of Chapter 3 is safe to use. In Section 5.4 the limitations of the system are considered and the modelling of parallel evaluation is discussed. In Section 5.5 we make some concluding remarks.

5.2 AN AXIOMATIC SEMANTICS

In this section a system of inference rules is presented. By using the rules we can try to deduce the value of a particular expression (the head normal form to which it reduces) from a statement that the expression is evaluated. These rules are intended to correspond to our intuition about demand-driven graph reduction. Note that the expression (E_1 E_2) should be thought of as being represented in the graph by the application node [S4]

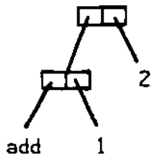


We shall construct predicates out of the following three formulae:-

x *delay* E *force* x x *eval* E

In reasoning about the evaluation of some expression, these formulae can be interpreted as follows.

x *delay* E states that at some stage of the computation E is stored ***in unevaluated form*** at x. For example, x *delay* (add 1 2) means that the machine must engage in the action of building the graph



The root node of this graph corresponds to the location denoted by x.

force x asserts that demand gets propagated to x, so causing evaluation of any expression stored at x. This is similar to the role of the function *force* defined in Chapter 4.

x *eval* E asserts that the machine must evaluate E, that is, perform a sequence of reductions so as to determine the head normal form of E, if one exists. If E does reduce to some head normal form H, then H is stored at x. For example, x *eval* (add 1 2) means that the value, 3, of (add 1 2) is eventually stored at x. Note that we do not specify when this evaluation takes place. So, for x *eval* E_1 and y *eval* E_2 , either E_1 or E_2 might be evaluated first, or their evaluations may be somehow intertwined.

Under this interpretation of the formulæ the first inference rule should come as no surprise.

<u>RULE OF *force*</u>

$$\frac{\mathit{force}\ x \qquad x\ \mathit{delay}\ E}{x\ \mathit{eval}\ E}$$

Predicates can be formed by combining these formulæ with the standard logical connectives, but we will only have occasion to use conjunction and existential quantification over identifiers. The set of inference rules includes reduction rules for the primitive reducible functions of the language. The <u>var</u>/<u>assign</u> constructs of functional programming with side-effects also have rules attributed to them. 'Rules of Reference' establish the relationship between identifiers and expressions. Finally, 'Rules of Order' reflect a restricted order of evaluation for expressions.

We shall have occasion to use $free[\![E]\!]$, the set of free variables in an expression E, and $E[y/x]$, the expression formed by replacing all free occurrences of x in E by y. These are defined formally at the end of this section.

The system has been so devised that an expression E, which does not contain free variables, evaluates to some head normal form H if and only if, from

$$x \textit{ eval } E$$

for some x, it is possible to deduce

$$\exists x_1 ... x_n .\ x \textit{ eval } H \qquad \textit{where } \{x_1, \ldots, x_n\} = free[\![H]\!] - \{x\}$$

Here, x_1 , ... , x_n denote new locations that are allocated during the reduction. For example, the value of

```
(var y : cons y (assign y = 1 ; nil))
```

is (cons y (<u>assign</u> y = 1 ; nil)) for some uninstantiated variable y. Thus, from

x *eval* (<u>var</u> y : cons y (<u>assign</u> y = 1 ; nil))

we should able to deduce

∃y. x *eval* (cons y (<u>assign</u> y = 1 ; nil))

The scope of the variable y in the program text is reflected by the use of existential quantification in the program logic.

<u>RULES OF REDUCTION</u>

In the λ-calculus, expressions can be simplified by the β-reduction rule. The following inference rule achieves the same effect.

(λ-Rule)
$$\frac{x \textit{ eval } ((\lambda y . E)\ z)}{x \textit{ eval } (E[z/y])}$$

Variable declaration introduces a new location and so takes the form

(<u>var</u>-Rule)
$$\frac{x \textit{ eval } (\underline{\text{var}}\ y\ ;\ E)}{\exists y .\ x \textit{ eval } E} \qquad \textit{provided } x \neq y$$

Note that: x *eval* (<u>var</u> x ; E) can be handled by first renaming the bound occurrence of x. For this purpose, we shall take for granted the following (α-conversion) rule:-

$$(\underline{\text{var}}\ x\ ;\ E) = (\underline{\text{var}}\ y\ ;\ (E[y/x]))\ \textit{for any } y,\ y \notin \textit{free}[\![E]\!]$$

The rule for assignment is

(<u>assign</u>-Rule)
$$\frac{x \textit{ eval } (\underline{\text{assign}}\ y = E_1\ :\ E_2)}{x \textit{ eval } E_2 \ \wedge\ y \textit{ delay } E_1}$$

This reflects the fact that the assignment construct is lazy.

Primitive functions, defined by reduction rules of the form

$$E_1 \longrightarrow E_2$$

are covered by the following schema:-

(Reduction) $$\frac{x\ \mathit{eval}\ E_1}{x\ \mathit{eval}\ E_2} \qquad \textit{provided } E_1 \longrightarrow E_2$$

Thus, the rule for head is

$$\frac{x\ \mathit{eval}\ (\mathrm{head}\ (\mathrm{cons}\ E_1\ E_2))}{x\ \mathit{eval}\ E_1}$$

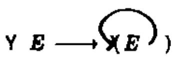
The rule for add is

$$\frac{x\ \mathit{eval}\ (\mathrm{add}\ n_1\ n_2)}{x\ \mathit{eval}\ n} \qquad \textit{where } n,\ n_1,\ n_2 \textit{ are integers with } n = n_1 + n_2$$

More interesting is the rule for Turner's circular version of the Paradoxical Combinator Y [54].

$$\frac{x\ \mathit{eval}\ (\mathrm{Y}\ E)}{x\ \mathit{eval}\ (E\ x)}$$

This is based on the reduction rule



Care has to be taken over the val combinator [29], giving call-by-value,

$$\text{val } E_1 \; E_2 \longrightarrow E_1 \; E_2$$
$$\text{val } E \; \bot \; = \; \bot$$

since val is not specified simply by a reduction rule. The correct inference rule is

$$\frac{x \; \mathit{eval} \; (\text{val } E \; H)}{x \; \mathit{eval} \; (E \; H)} \quad \mathit{for} \; H \; \mathit{in} \; \mathit{head} \; \mathit{normal} \; \mathit{form}$$

N.B. The above rule is illustrative of our method for specifying order of evaluation. It is not permitted for the reduction

$$\text{val } E_1 \; E_2 \longrightarrow E_1 \; E_2$$

to take place until E_2 has first been reduced to head normal form.

Similarly, we have

$$\frac{x \; \mathit{eval} \; (\text{seq } H \; E)}{x \; \mathit{eval} \; E} \quad \mathit{for} \; H \; \mathit{in} \; \mathit{head} \; \mathit{normal} \; \mathit{form}$$

RULES OF REFERENCE

The Rules of Reduction only permit us to perform reductions at the outermost level of an expression. For instance, we can deduce x *eval* true from x *eval* (not false), but no means has been provided for deducing x *eval* false from x *eval* (not (not false)). Hence, we must extend the system with new rules. These, when combined with the Rules of Order, will enable us to evaluate subexpressions and substitute their values back into the main expression.

The first rule explains what it means to evaluate an identifier.

(Reference)

$$\frac{x \ \mathit{eval}\ y \qquad y\ \mathit{eval}\ E}{x\ \mathit{eval}\ E}$$

The rule

(Abstraction)

$$\frac{x\ \mathit{eval}\ (E_1\ E_2)}{\begin{array}{l}\exists y.\ x\ \mathit{eval}\ (y\ E_2)\ \wedge\ y\ \mathit{delay}\ E_1\\ \exists y.\ x\ \mathit{eval}\ (E_1\ y)\ \wedge\ y\ \mathit{delay}\ E_2\end{array}} \quad \begin{array}{l}\mathit{provided}\\ y \notin \{x\} \cup \mathit{free}[\![E_1\ E_2]\!]\end{array}$$

enables us to focus on a particular subexpression by abstracting it out of the main expression. The fact that x can be thought of as an application node with pointers to E_1 and E_2 provides some justification for this rule. The first inference could also have been written as

$$\exists y.\ x\ \mathit{eval}\ (y\ E_2)\ \wedge\ y\ \mathit{eval}\ E_1$$

since normal order reduction implies $\mathit{force}\ y$, as will be given in the Rules of Order.

It remains to provide a mechanism by which an identifier, denoting some location, can be replaced by the expression stored at that location. Thus:

(Substitution)

$$\frac{x\ \mathit{eval}\ (y\ E_2) \qquad y\ \mathit{eval}\ E_1}{x\ \mathit{eval}\ (E_1\ E_2)} \qquad\qquad \frac{x\ \mathit{eval}\ (E_1\ y) \qquad y\ \mathit{eval}\ E_2}{x\ \mathit{eval}\ (E_1\ E_2)}$$

Rules of Order

Normal order reduction is specified by the rule

(Normal Order Rule)

$$\frac{x \ eval \ (y \ E)}{force \ y}$$

Evaluation of an identifier involves sending demand to the location denoted by the identifier. Thus:

(Indirection Rule)

$$\frac{x \ eval \ y}{force \ y}$$

This should be compared with the semantic equation $\mathcal{E}[\![y]\!]\rho\kappa = force\,(\rho[\![y]\!])\,\kappa$ of Chapter 4.

The following schema is applicable to any primitive p of arity n.

(Strictness) $$\frac{x \ eval \ (p \ E_1 \ \dots \ E_{i-1} \ y \ E_{i+1} \ \dots \ E_n)}{force \ y}$$

provided (i) $p \ E_1 \ \dots \ E_{i-1} \perp E_{i+1} \ \dots \ E_n = \perp$

and (ii) *E_k is in head normal form if p evaluates its k^{th} argument before its i^{th}.*

For example,

$$\frac{x \; eval \; (\text{head } y)}{force \; y}$$

$$\frac{x \; eval \; (\text{add } y \; E)}{force \; y} \qquad\qquad \frac{x \; eval \; (\text{add } n \; y)}{force \; y} \; integer \; n$$

Note that the asymmetry of the above rules for add reflects a left-argument-before-right evaluation strategy. The operand E_2 in the expression (add E_1 E_2) may be left unevaluated if evaluation of E_1 failed, for some reason, to yield an integer.

val gives applicative order evaluation. So, in the case of val, the strictness rule becomes

$$\frac{x \; eval \; (\text{val } E \; y)}{force \; y}$$

AN EXAMPLE OF DEDUCTION

Consider the example given in Chapter 2, Section 2.3, Note (4). We used informal reasoning to show that

```
        (add (assign x = 3 ; 2) (sub 4 x))
```

evaluates to 3 , whereas evaluation of

```
        (add (sub 4 x) (assign x = 3 ; 2))
```

suspends, for x an uninstantiated variable. With our axiomatic semantics we might reason as follows.

1. y *eval* (add (<u>assign</u> x = 3 ; 2) (sub 4 x))	
2. ∃u. y *eval* (add u (sub 4 x)) ∧ u *eval* (<u>assign</u> x = 3 ; 2)	(Lemma 1)
3. ∃u. y *eval* (add u (sub 4 x)) ∧ u *eval* 2 ∧ x *delay* 3	(<u>assign</u>-Rule)
4. ∃u. y *eval* (add 2 (sub 4 x)) ∧ x *delay* 3	(Lemma 2)
5. y *eval* (add 2 (sub 4 x)) ∧ x *delay* 3	(∃-Elimination)
6. ∃v. y *eval* (add 2 v) ∧ v *eval* (sub 4 x) ∧ x *delay* 3	(Lemma 3)
7. ∃v. y *eval* (add 2 v) ∧ v *eval* (sub 4 x) ∧ *force* x ∧ x *delay* 3	(Strictness)
8. ∃v. y *eval* (add 2 v) ∧ v *eval* (sub 4 x) ∧ x *eval* 3	(Rule of *force*)
9. ∃v. y *eval* (add 2 v) ∧ v *eval* (sub 4 3)	(Substitution)
10. ∃v. y *eval* (add 2 v) ∧ v *eval* 1	(Reduction)
11. ∃v. y *eval* (add 2 1)	(Substitution)
12. y *eval* (add 2 1)	(∃-Elimination)
13. y *eval* 3	(Reduction)

This proves that (add (<u>assign</u> x = 3 ; 2) (sub 4 x)) evaluates to 3. It uses the lemmas

<u>LEMMA 1</u>

$$\frac{x \; \mathit{eval} \; (\mathrm{add} \; E_1 \; E_2)}{\exists y. \; x \; \mathit{eval} \; (\mathrm{add} \; y \; E_2) \wedge y \; \mathit{eval} \; E_1}$$

provided $y \notin \{x\} \cup \mathit{free}[\![E_1]\!] \cup \mathit{free}[\![E_2]\!]$

<u>PROOF</u>

1. x *eval* (add E_1 E_2)	
2. $\exists z.$ x *eval* (z E_2) ∧ z *eval* (add E_1)	(Abstraction)
3. $\exists y,z.$ x *eval* (z E_2) ∧ z *eval* (add y) ∧ y *delay* E_1	(Abstraction)
4. $\exists y,z.$ x *eval* (add y E_2) ∧ y *delay* E_1	(Substitution)
5. $\exists y.$ x *eval* (add y E_2) ∧ y *delay* E_1	(∃-Elimination)
6. $\exists y.$ x *eval* (add y E_2) ∧ *force* y ∧ y *delay* E_1	(Strictness)
7. $\exists y.$ x *eval* (add y E_2) ∧ y *eval* E_1	(Rule of *force*)

LEMMA 2

$$\frac{x\ eval\ (E_1\ y\ E_3) \wedge y\ eval\ E_2}{x\ eval\ (E_1\ E_2\ E_3)}$$

PROOF

1. $x\ eval\ (E_1\ y\ E_3) \wedge y\ eval\ E_2$
2. $\exists z.\ x\ eval\ (z\ E_3) \wedge z\ eval\ (E_1\ y) \wedge y\ eval\ E_2$ (Abstraction)
3. $\exists z.\ x\ eval\ (z\ E_3) \wedge z\ eval\ (E_1\ E_2)$ (Substitution)
4. $\exists z.\ x\ eval\ (E_1\ E_2\ E_3)$ (Substitution)
5. $x\ eval\ (E_1\ E_2\ E_3)$ (∃-Elimination)

LEMMA 3

$$\frac{x\ eval\ (\text{add}\ n\ E)}{\exists y.\ x\ eval\ (\text{add}\ n\ y) \wedge y\ eval\ E}$$

for any integer n*, provided* $y \notin \{x\} \cup free[\![E]\!]$

PROOF

1. $x\ eval\ (\text{add}\ n\ E)$
2. $\exists y.\ x\ eval\ (\text{add}\ n\ y) \wedge y\ delay\ E$ (Abstraction)
3. $\exists y.\ x\ eval\ (\text{add}\ n\ y) \wedge force\ y \wedge y\ delay\ E$ (Strictness)
4. $\exists y.\ x\ eval\ (\text{add}\ n\ y) \wedge y\ eval\ E$ (Rule of *force*)

Consider now evaluation of (add (sub 4 x) (assign x = 3 ; 2)). We are unable to prove ***within the system*** that evaluation suspends. However, any attempt to prove that it reduces to some H is destined to fail. For example,

1. y *eval* (add (sub 4 x) (assign x = 3 ; 2))
2. ∃u. y *eval* (add u (assign x = 3 ; 2)) ∧
   u *eval* (sub 4 x) (Lemma 1)
3. ∃u, v. y *eval* (add u v) ∧ u *eval* (sub 4 x) ∧
   v *delay* (assign x = 3 ; 2) (Abstraction)

and, because we lack the information *force* v, the value of u cannot be deduced.

For completeness, we conclude this section with the definitions of *free* and substitution.

The notion of *free variables* of an expression, can be formalized as follows:-

$$
\begin{aligned}
&\mathit{free} : \mathbf{Exp} \to \mathbb{P}\,\mathbf{Ide}\\
&\mathit{free}[\![x]\!] = \{x\}\\
&\mathit{free}[\![p]\!] = \{\}\\
&\mathit{free}[\![E_1\ E_2]\!] = \mathit{free}[\![E_1]\!] \cup \mathit{free}[\![E_2]\!]\\
&\mathit{free}[\![\lambda x.\, E]\!] = \mathit{free}[\![E]\!] - \{x\}\\
&\mathit{free}[\![\underline{\text{var}}\ x\ ;\ E]\!] = \mathit{free}[\![E]\!] - \{x\}\\
&\mathit{free}[\![\underline{\text{assign}}\ x = E_1\ ;\ E_2]\!] = \{x\} \cup \mathit{free}[\![E_1]\!] \cup \mathit{free}[\![E_2]\!]
\end{aligned}
$$

We next define a process of syntactic substitution. The substitution of the identifier x' for x in an expression is defined recursively as follows:-

$$E[x/x] = E$$

For $x' \neq x$:-

$$
\begin{aligned}
&x[x'/x] = x'\\
&y[x'/x] = y \quad \mathit{for}\ y \neq x\\
&p[x'/x] = p\\
&(E_1\ E_2)[x'/x] = E_1[x'/x]\ (E_2[x'/x])\\
&(\lambda x.E)[x'/x] = \lambda x.E\\
&(\lambda x'.E)[x'/x] = \lambda y.(E[y/x'][x'/x])\\
&\qquad\qquad \mathit{for\ some}\ y.\quad y \notin \{x, x'\} \cup \mathit{free}[\![E]\!]\\
&(\lambda y.E)[x'/x] = \lambda y.(E[x'/x])\ \mathit{for}\ y \neq x,\ y \neq x'\\
&(\underline{\text{var}}\ x\ ;\ E)[x'/x] = \underline{\text{var}}\ x\ ;\ E\\
&(\underline{\text{var}}\ x'\ ;\ E)[x'/x] = \underline{\text{var}}\ y\ ;\ (E[y/x'][x'/x])\\
&\qquad\qquad \mathit{for\ some}\ y.\quad y \notin \{x, x'\} \cup \mathit{free}[\![E]\!]\\
&(\underline{\text{var}}\ y\ ;\ E)[x'/x] = \underline{\text{var}}\ y\ ;\ (E[x'/x])\ \mathit{for}\ y \neq x,\ y \neq x'\\
&(\underline{\text{assign}}\ y = E_1 ; E_2)[x'/x] = \underline{\text{assign}}\,(y[x'/x]) = (E_1[x'/x])\ ;\ (E_2[x'/x])
\end{aligned}
$$

5.3 REASONING ABOUT PROGRAMS

As described so far, the axiomatic semantics allows us to reason about expressions in which all functions are either primitives or locally defined as λ-expressions. In this section inference rules will be given to functions defined in a program. Rules will even be prescribed for definitions of functions that involve pattern-matching. Finally, the use of inference rules in reasoning about the safety of the move-in transformation of Chapter 3 will be demonstrated.

A reducible function f of arity n can be defined in a program by an equation of the form

$$f\ x_1\ \dots\ x_n = E$$

where $x_1, \dots, x_n$ are distinct identifiers and ignoring <u>var</u> parameter annotations. In reasoning about the evaluation of expressions according to this program, we can make use of the rule of reduction

$$\frac{x\ \mathit{eval}\ (f\ y_1\ \dots\ y_n)}{x\ \mathit{eval}\ (E[y_1/x_1]\dots[y_n/x_n])}$$

Thus, the functions foo and baz defined in Section 5.1 by

```
foo = var x ; add x (baz x)
baz (var y) = assign y = 1 ; 2
```

give rise to the rules

$$\frac{x\ \mathit{eval}\ \texttt{foo}}{x\ \mathit{eval}\ (\underline{\texttt{var}}\ \texttt{x ; add x (baz x)})} \qquad \frac{x\ \mathit{eval}\ (\texttt{baz}\ y)}{x\ \mathit{eval}\ (\underline{\texttt{assign}}\ y = 1\ ;\ 2)}$$

This method naturally extends to definitions involving pattern-matching, but additional rules of order are also required. That is, the program

$$\begin{aligned} &f\ \mathtt{nil} &&= E_1 \\ &f\ (\mathtt{cons}\ x_1\ x_2) &&= E_2 \end{aligned}$$

has the following three rules associated with it:-

$$\frac{x\ \mathit{eval}\ \ (f\ \mathtt{nil})}{x\ \mathit{eval}\ E_1} \qquad \frac{x\ \mathit{eval}\ \ (f\ \ (\mathtt{cons}\ y_1\ y_2))}{x\ \mathit{eval}\ \ (E_2[y_1/x_1][y_2/x_2])} \qquad \frac{x\ \mathit{eval}\ (f\ \ y)}{\mathit{force}\ y}$$

Returning to the solution presented to Example 1 in Chapter 2, the program

```
transform : Tree → Tree
transform t = var m ; replace t m m


replace : Tree → Int → Int → Tree
replace (tip n)    m (var v) = assign v = n ; tip m
replace (fork L R) m (var v) = var y ; var z ;
                               assign v = y MIN z ;
                               fork (replace L  m y)
                                    (replace R  m z)
```

gives rise to the following inference rules:-

$$\frac{x\ \mathit{eval}\ (\mathtt{transform}\ y)}{x\ \mathit{eval}\ (\underline{\mathtt{var}}\ \mathtt{m}\ ;\ \mathtt{replace}\ y\ \mathtt{m}\ \mathtt{m})} \qquad \frac{x\ \mathit{eval}\ (\mathtt{replace}\ y\ E_1\ E_2)}{\mathit{force}\ y}$$

$$\frac{x\ \mathit{eval}\ (\mathtt{replace}\ \ (\mathtt{tip}\ y_1)\ y_2\ y_3)}{x\ \mathit{eval}\ (\underline{\mathtt{assign}}\ y_3 = y_1\ ;\ \mathtt{tip}\ y_2)} \quad \frac{x\ \mathit{eval}\ (\mathtt{replace}\ \ (\mathtt{fork}\ y_1\ y_2)\ y_3\ y_4)}{x\ \mathit{eval}\ \begin{array}{l}(\underline{\mathtt{var}}\ \mathtt{y}\ ;\ \underline{\mathtt{var}}\ \mathtt{z}\ ; \\ \quad \underline{\mathtt{assign}}\ y_4 = \mathtt{y}\ \mathtt{MIN}\ \mathtt{z}\ ; \\ \quad \mathtt{fork}\ (\mathtt{replace}\ y_1\ y_3\ \mathtt{y}) \\ \qquad\quad (\mathtt{replace}\ y_2\ y_3\ \mathtt{z}))\end{array}}$$

If multiple patterns occur in a single definition then the appropriate set of rules of strictness will depend upon the order in which patterns are matched. Also, for certain data types pattern matching can be 'lazy' [59]. For example, whether f is strict when defined by $f\ (x,y) = E$ depends on the implementation - it is in Orwell [58], but not in Miranda [57] where it is treated as $f\ z = E$ where $(x,y) = z$. It is obviously a good idea to ensure that the correctness of a program does not rely on a particular implementation of pattern-matching.

Note that, because of the relationship between where and var/assign (Chapter 2, Section 2.3, Note 3), we can derive the rule

$$\frac{x\ eval\ (E\ \underline{\text{where}}\ x_1 = E_1,\ \ldots\ ,\ x_n = E_n)}{\exists x_1,\ldots,x_n.\ x\ eval\ E \wedge x_1\ delay\ E_1 \wedge \ldots \wedge x_n\ delay\ E_n}$$

provided $x \notin \{x_1,\ldots,x_n\}$

Note also that E_1 where $(x,y) = E_2$ is treated as

$$E_1\ \underline{\text{where}}\ x = \text{fst}\ z,\ y = \text{snd}\ z,\ z = E_2$$

An important use of the axiomatic semantics is to check the safety of a move-in transformation when synthesizing a new function definition by program transformation. The rules explained informally in Chapter 3, Section 3.2, can be re-expressed succinctly as follows:-

Consider the move-in transformation $(\underline{\text{assign}}\ x = E\ ;\ E')$ = E'' where E'' is E' with some subexpression $E^{\dagger}$ replaced by

$$(\underline{\text{assign}}\ x = E\ ;\ E^{\dagger})$$

We can prove *by local reasoning* that the transformation is safe if from *y eval E"* we can still deduce *x delay E*.

Likewise, the transformation is known to introduce deadlock if from *y eval E"* we can deduce *force x*, but not *x delay E*.

Thus, examples of safe move-in transformations are

(assign x = E ; (E_1 E_2)) = ((assign x = E ; E_1) E_2)
(assign x = E ; (val E_1 E_2)) = (val E_1 (assign x = E ; E_2))
(assign x = E ; (add E_1 E_2)) = (add (assign x = E ; E_1) E_2)
(assign x = E ; (add n E')) = (add n (assign x = E ; E'))

Examples of the transformation introducing deadlock are

(assign x = E ; (x E')) = (x (assign x = E ; E'))
(assign x = E ; (val E' x)) = (val (assign x = E ; E') x)
(assign x = E ; (add x E')) = (add x (assign x = E ; E'))

5.4 LIMITATIONS OF THE AXIOMATIC APPROACH

Our first remark is that the semantic model is not sufficiently 'fine' to distinguish between computations that deadlock and those that loop (diverge, never terminate). For example, evaluation of (var x ; x) deadlocks, whereas evaluation of

```
        finite (Y (cons 1))
```

where finite is defined by

```
finite : List * → Bool
finite nil          =  true
finite (cons x xs)  =  finite xs
```

diverges: but in both cases we can do no more than to try in vain to find a head normal form. This is no great disadvantage for sequential evaluation, since it is reasonable to treat programs that come to a premature halt and those that run forever as equally bad. However, it does limit what can be deduced about parallel evaluation, which will be discussed next.

Can the rules be modified to allow for parallel evaluation of expressions? Not only can this be done, but it can be done very simply.

Suppose, for example, that add is to evaluate both operands in parallel. We need only relax its second rule of strictness. That is,

$$\frac{x \; \mathit{eval} \; (\text{add} \; n \; y)}{\mathit{force} \; y} \quad \mathit{for\ any\ integer} \; n$$

to the more general

$$\frac{x \; \mathit{eval} \; (\text{add} \; E \; y)}{\mathit{force} \; y}$$

Its two rules of strictness are now symmetrical.

Hughes' par combinator [29, 30] has the rule of reduction

$$\frac{x \; \mathit{eval} \; (\text{par} \; E_1 \; E_2)}{x \; \mathit{eval} \; (E_1 \; E_2)}$$

and rule of order

$$\frac{x \; \mathit{eval} \; (\text{par} \; E \; y)}{\mathit{force} \; y}$$

Note that the move-in transformation

$(\underline{\text{assign}}\ x = \boldsymbol{E}\ ;\ \text{par}\ \boldsymbol{E}_1\ \boldsymbol{E}_2) = (\text{par}\ \boldsymbol{E}_1\ (\underline{\text{assign}}\ x = \boldsymbol{E}\ ;\ \boldsymbol{E}_2))$

is safe because from y *eval* (par $\boldsymbol{E}_1$ ($\underline{\text{assign}}$ x = $\boldsymbol{E}$; $\boldsymbol{E}_2$)) we can deduce x *delay* $\boldsymbol{E}$.

Now we clarify our previous remark about the problem of failing to distinguish between deadlock and divergence. Consider the expression

(par (K $\boldsymbol{H}$) $\boldsymbol{E}$)

where evaluation of $\boldsymbol{E}$ (i) deadlocks, or (ii) diverges. For (i) it seems reasonable to say that (par (K $\boldsymbol{H}$) $\boldsymbol{E}$) evaluates to $\boldsymbol{H}$. However, for (ii) a particular implementation of parallel evaluation may diverge by using all its resources to evaluate $\boldsymbol{E}$: we have to wait an indefinite time for the result. (This is analogous to the danger of not finding a normal form even though one exists, when applicative order reduction is used instead of normal order reduction.) Unfortunately, the rules allow us to deduce x *eval* $\boldsymbol{H}$ from x *eval* (par (K $\boldsymbol{H}$) $\boldsymbol{E}$) in both case (i) and case (ii).

There is one further danger arising from the axiomatization. We might be tempted to attribute a value to expressions that would in fact return an error when evaluated. For example, from

x *eval* ($\underline{\text{assign}}$ y = 2 ;. $\underline{\text{assign}}$ y = 3 ; y) .

it is possible to deduce both x *eval* 2 and x *eval* 3, but a graph reduction machine should report an error in performing the evaluation. Care must be taken to keep to the 'single-assignment' principle in writing programs, for example, by developing programs by transformation. It would also be possible to trap such errors while type-checking.[1]

[1]The $\underline{\text{var}}$ parameter annotations provide information to facilitate this. For example, the definition of f should be rejected in the cases (i) f x = $\underline{\text{assign}}$ x = $\boldsymbol{E}_1$. $\boldsymbol{E}_2$ and (ii) f ($\underline{\text{var}}$ x) = $\underline{\text{assign}}$ x = $\boldsymbol{E}_1$; g x where g is defined by g ($\underline{\text{var}}$ y) = $\boldsymbol{E}_2$

One final comment concerns Hughes' suggestion [30] that temporal logic might be used for describing his synchronization primitive. It has already been shown (Chapter 2, Section 2.4, Example 3) that synch can be defined by

```
synch e = var e1 ; var e2 ;
          pair (assign e2 = e ; e1) (assign e1 = e ; e2)
```

Thus, our axiomatic semantics can be used to understand synchronized behaviour. Taking an example from [29], it should be clear that: for the expression

```
        add x y where (x,y) = synch E
```

to evaluate to twice the value of E, requires that add evaluates its arguments in parallel. The point is that we need the information *force* x and *force* y, before we can deduce x *eval* E ∧ y *eval* E .

5.5 CONCLUSION

An axiomatic semantics has been presented for functional programming with side-effects. The inference rules give us a grasp of this method of programming in much the same way as reduction rules help us to reason about purely functional programs. Particularly pleasant is the ease with which rules can be given to specify parallel evaluation.

In postulating the rules appeal has been made to properties underlying evaluation by graph reduction; but, it is hoped that the rules alone are sufficient to provide an adequate understanding of functional programming with side-effects. In particular, they may prove useful in reasoning about the move-in transformation, by which programs with side-effects can be developed.

CHAPTER 6

SUMMARY AND RELATED WORK

In this thesis we have put forward a proposal for a new programming feature in an otherwise purely functional language. The extension makes possible a new style of programming, ***functional programming with side-effects.*** The examples of Chapter 2 demonstrate the feasibility of programming with side-effects. They also provide evidence that solutions to some programming problems can be developed in the extended language that are more efficient than may be possible in a purely functional language. Furthermore, it was shown in Chapter 3 that transformational programming provides a sound methodology for the development of programs with side-effects. It would appear to be the case that the task of ensuring that programs with side-effects are free from deadlock is most easily tackled when such programs are developed by transformation from purely functional programs. It has also been shown that programs with side-effects are closely related to programs involving tuples or continuations.

The formal semantics of the extended language have been considered. In Chapter 4 a denotational semantics was presented. This also serves to provide us with a mathematical model of the process of graph reduction, by which the lazy evaluation strategy can be implemented. Chapter 5 explored a novel programming logic. The logical system gives an axiomatic semantics for the language. It promotes reasoning about programs with side-effects. This reasoning can be performed without knowledge of how a computer executes such programs.

It is important to appreciate that the new feature can readily be accommodated by graph reduction based implementations of functional programming languages. A description can be found in the appendix of the trivial modifications that had to be made to one such implementation. There should, for example, be no difficulties in compiling programs with side-effects into G-machine code [31] or ALICE CTL [45]. Indeed, as was mentioned in Chapter 2, a brief description of the method of binding logical variables by graph reduction appears in the ALICE paper [15]. Thus, programming with side-effects can be supported on both sequential and parallel machines.

Research into functional programming has previously shown that different orders of evaluation can alter the efficiency of programs. Schwarz has proposed call-by-opportunity [48] and Hughes par and synch [29,30] as suitable annotations for changing the flow of control without changing the structure of programs. Other control annotations have been suggested by Burton [8]. We have shown (Example 3 of Chapter 2) that with side-effects it is possible to give a space efficient version of Hughes' split function. Control annotations have also proven useful in developing deadlock-free programs with side-effects.

Previous attempts to incorporate logical variables into functional languages, for example [3, 14], have assumed program execution can involve unification. Programs with side-effects require no more than standard pattern-matching. There is, however, some similarity between our language and the non-backtracking logic based languages, such as the relational language of [10] from which Parlog [11] and Concurrent Prolog [49] have evolved. For example, execution of the logic program

```
mode sort(?,^)
sort(nil,nil).
sort(cons(x,xs),ys) ⇐ partition(x,xs,us,vs), sort(us,ps),
                      sort(vs,qs), append(ps,cons(x,qs),ys)

mode partition(?,?,^,^)
partition(x,nil,nil,nil).
partition(x,cons(z,zs),us,cons(z,vs)) ⇐ x≤z |
                                        partition(x,zs,us,vs)
partition(x,cons(z,zs),cons(z,us),vs) ⇐ x>z |
                                        partition(x,zs,us,vs)

mode append(?,?,^)
append(nil,ys,ys).
append(cons(x,xs),ys,cons(x,zs)) ⇐ append(xs,ys,zs)
```

proceeds in a similar way to that of the functional program

```
sort : List Int → List Int → Bool
sort  nil        (var ys) = assign ys = nil ; true
sort (cons x xs) (var ys) = var ps, qs;
                            partition x xs us vs AND
                            sort us ps AND sort vs qs AND
                            append ps (cons x qs) ys
```

```
partition : Int → List Int → List Int → List Int → Bool
partition x  nil (var us) (var vs) = assign us = nil, vs = nil ;
                                     true
partition x (cons z zs) (var us) (var vs)
       = var vs' ; assign vs = cons z vs' ;
         partition x zs us vs'              if x≤z
         var us' ; assign us = cons z us' ;
         partition x zs us' vs              otherwise


append : List * → List * → List * → Bool
append  nil        ys (var zs) = assign zs = ys ; true
append (cons x xs) ys (var zs) = var zs' ;
                                 assign zs = cons x zs' ;
                                 append xs ys zs'
```

where AND is the standard infix operator. (Detailed explanation of mode declarations and the ***commit bar*** '|' can be found in [10].)

Finally, the language can be compared with notations for describing parallel processes. The idea of suspending processes until a variable becomes instantiated has been suggested by Banatre [2]. There is no synchronization between one process engaging in the ***output*** event of assigning to a variable and other processes that require the value of that variable. This contrasts with the ***hand-shaking*** form of communication, as in CSP [24], in which a process must be ready to receive a message before another process can transmit it. In its treatment of parallelism, programming with side-effects is closer to the languages Parlog and Concurrent Prolog, mentioned above.

POSTSCRIPT. It should also be pointed out that an idea related to programming with side-effects is described in [Arvind, R E Thomas. ***I-Structures: An Efficient Data Type for Functional Languages.*** MIT/LCS/TM-178, 1980]. I-structures were conceived, however, as a means of reducing data dependencies in dataflow languages.

APPENDIX

IMPLEMENTATION

This appendix provides details of a prototype implementation of a functional programming language with side-effects, an extension of the purely functional language Orwell [58]. Side-effects were incorporated by means of a trivial modification to the Modula-2 interpreter for Orwell. The implementation provides a powerful system for experimenting with side-effects and some encouraging results have been produced: particularly, when programs with side-effects were compared with programs involving tuples. It seems also that, by making more extensive changes to the interpreter, a substantial improvement in the space/time performance of programs with side-effects can be obtained.

Orwell supports a recursive <u>where</u> construct. This has simplified the implementation of the <u>var</u> construct, as will be explained. In fact, no changes to the syntax of expressions were needed in order to provide for side-effects. Instead, two new primitives, IsAVar and assign, were added. These can be given types

```
IsAVar : * → **
assign : * → * → ** → **
```

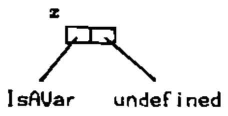
The concrete syntax of the expression (<u>var</u> x ; E) is

(*E* <u>where</u> x = IsAVar undefined)

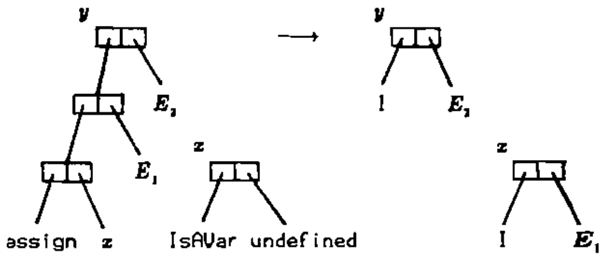
and that of (<u>assign</u> x = E_1 ; E_2) is

(assign x E_1 E_2)

Thus, an uninstantiated variable x is represented in the graph by



The reduction rule for assign can then be pictured as



Implementation essentially involved adding the following two procedures to the module in which the primitive operators for Orwell were defined.

```
PROCEDURE VarProc (VAR x : Object) : Object ;
BEGIN
   RETURN MakeError1 ("IsAVar", x) ;
END VarProc ;

PROCEDURE AssignProc (VAR x, e1, e2 : Object) : Object ;
BEGIN
   IF (INTEGER (x) <= PAIRHI) AND
      (x^.left = IsAVar) THEN
      x^.left := IObj ;
      x^.right := e1 ;
      RETURN e2 ;
   ELSE RETURN MakeError3 ("assign", x, e1, e2) ;
   END ;
END AssignProc ;
```

VarProc deals with an attempt to evaluate an uninstantiated variable. Since we do not have parallelism, no further progress can be made. We choose to report this state of deadlock as an error. AssignProc is the code for the reduction rule shown above.

Note that: it would not have been possible simply to represent an uninstantiated variable by undefined, because the implementation of Orwell does not allocate a new location for x in the expression

$(E$ <u>where</u> x = undefined$)$

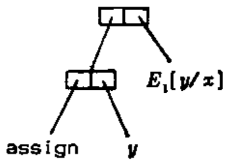
The above method of implementing the assign construct can be improved upon. Consider, for example, the evaluation of (f y), for some uninstantiated variable y, according to the definition

f (var x) = assign $x = E_1$; E_2

An efficient implementation would, in a single reduction step, bind y to $E_1[y/x]$ and rewrite (f y) as $E_2[y/x]$. However, in the prototype implementation, f is defined by

f x = assign x E_1 E_2

(f y) must first be reduced to (assign y $E_1[y/x]$ $E_2[y/x]$) before the above reduction can take place. Thus, the structure



is constructed, only to become garbage after one reduction. In this example, the prototype implementation, compared with an efficient implementation of side-effects, requires one more reduction and two more cells when reducing (f y) to $E_2[y/x]$. For a function such as

f (var x) (var y) = assign $x = E_1$, $y = E_2$: E

which has to be defined by

f x y = assign x E_1 (assign y E_2 E)

two more reductions and five more cells are required.

Thus, when interpreting the following table of results, allowance must be made for the possibility of ***significant improvements*** to the space and time requirements of the side-effects solutions.

EXAMPLE	METHOD	REDUCTIONS[1]	CELLS[1]	DESCRIPTION
1	Tupling	1	1	first tree
	Side-effects$_{T1}$	.98	.82	transformation
	Side-effects$_{T2}$	.80	.73	problem
2	Tupling	1	1	second tree
	Side-effects$_{T3}$	1.00	.77	transformation
	Side-effects$_{T4}$	.97	.78	problem
3	Tupling	{Memory Overflow}		split in
	Continuations	{Memory Overflow}		constant space
	Side-effects$_T$	linear	constant	
4/7	Tupling	1	1	quicksort
	Side-effects$_T$	.71	.64	
	Continuations	.77	.57	
	Side-effects$_C$	.84	.90	
5	Tupling	1	1	Fibonacci
	Side-effects$_T$	.76	.67	numbers
	Continuations	.76	.45	
	Side-effects$_C$	.76	.81	

Side-effects$_T$ = tupling-related side-effects strategy
Side-effects$_C$ = continuations- " " " "

[1]Except for Example 3, the number of reductions and cells are expressed as a fraction of those required by the tupling solution.

The above table compares various solutions to the examples of Chapter 2. For each example, the programs were compared over a wide range of input, that is, trees in Examples 1 and 2, lists in Examples 3 and 4, and integers in Example 5. Side-effects$_{T1}$, Side-effects$_{T2}$, Side-effects$_{T3}$ and Side-effects$_{T4}$ refer to solutions derived from the following definitions:-

```
replace' t m (var v) = assign v = tmin t ; replace t m      (1)

tmin' t m (var s)    = assign s = replace t m ; tmin t      (2)

replace' t us (var vs) (var ws) rs
                     = assign ws = drop (size t) us,
                              vs = tips t ++ rs ;
                       replace t us                         (3)

tips' t us (var s) (var ws) rs
                     = assign ws = drop (size t) us,
                               s = replace t us ;
                       tips t ++ rs                         (4)
```

It seems reasonable to conclude that side-effects offer sizeable gains over tupling, and, if implemented efficiently, some improvement over continuations. Side-effects is the only method that solves the constant space split problem [29,30].

REFERENCES

[1] L Augustsson. *Compiling Pattern Matching.* In Proc. Functional Programming Languages and Computer Architecture. Nancy, 1985 (Lecture Notes in Computer Science 201. Springer-Verlag)

[2] J-P Banatre. *Co-operation Schemes for Parallel Programming.* In Distributed Computing Systems, Synchronization, Control and Communication, Academic Press, 1983

[3] M Bellia, P Degano, G Levi. *The Call by Name Semantics of a Clause Language with Functions.* In Logic Programming, Academic Press, 1982

[4] R S Bird. *Using circular programs to eliminate multiple traverses of data.* Acta Informatica, Vol. 21, 1984

[5] R S Bird. *Tabulation Techniques for Recursive Programs.* Computing Surveys, Vol. 12, No. 4, 1980

[6] H-J Boehm. *Side Effects and Aliasing Can Have Simple Axiomatic Descriptions.* ACM TOPLAS, Vol. 7, No. 4, 1985

[7] R M Burstall, J Darlington. *A Transformation System for developing recursive programs.* J ACM, Vol. 24, No. 1, 1977

[8] F W Burton. *Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs.* ACM TOPLAS, Vol. 6, No. 2, 1984

[9] K L Clark, F G McCabe, S Gregory. *IC-Prolog Language Features.* In Logic Programming, Academic Press, 1982

[10] K L Clark, S Gregory. *A Relational Language for Parallel Programming.* In Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Wentworth, 1981

[11] K L Clark, S Gregory. *PARLOG: parallel programming in logic.* Research Report DOC 84/4, Imperial College London, 1984

[12] W F Clocksin, C S Mellish. *Programming in Prolog.* Springer-Verlag, 1981

[13] J Darlington. *Program transformation.* In Functional Programmming and its Applications, Cambridge University Press, 1982

[14] J Darlington. *Unification of Logic and Functional Languages.* Imperial College London, 1983

[15] J Darlington, M Reeve. *ALICE: a multiprocessor reduction machine for the parallel evaluation of applicative languages.* In Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Wentworth, 1981

[16] A L Davis, R M Keller. *Data Flow Program Graphs.* COMPUTER magazine, IEEE Computer Society, Vol. 15, No. 2, 1982

[17] E W Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976

[18] M S Feather. *A system for assisting program transformation.* ACM TOPLAS, Vol. 4, No. 1, 1982

[19] D P Frieman, D S Wise. *CONS should not evaluate its arguments.* In Proc. 3rd International Colloquium on Automata Languages and Programming, Edinburgh University Press, 1976

[20] D Gries. *The Science of Programming.* Springer-Verlag, 1981

[21] P Henderson. *Functional Programming: application and implementation.* Prentice-Hall, 1980

[22] P Henderson, J H Morris. *A lazy evaluator.* In Proc. 3rd Annual SIGACT-SIGPLAN Symposium on Principles of Programming Languges, Atlanta, 1976

[23] C A R Hoare. *An Axiomatic Basis for Computer Programming.* C ACM, Vol. 12, No. 10, 1969

[24] C A R Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985

[25] C A R Hoare. *Programs are Predicates.* In Mathematical Logic and Programming Languages. Prentice-Hall, 1985

[26] C A R Hoare. Private Communication. 1985

[27] P Hudak, B Goldberg. *Experiments in Diffused Combinator Redution.* In Proc. ACM Symposium on LISP and Functional Programming, Austin, 1984

[28] R J M Hughes. *Graph reduction with super-combinators.* PRG-28, Oxford University Programming Research Group, 1982

[29] R J M Hughes. *The Design and Implementation of Programming Languages.* Oxford University D.Phil Thesis, 1983 (PRG-40)

[30] R J M Hughes. *Parallel functional languages use less space.* Oxford University Programming Research Group, 1984

[31] T Johnsson. *Efficient Compilation of Lazy Evaluation.* In Proc. ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, 1984

[32] R M Keller, M R Sleep. *Applicative Caching: programmer control of object sharing and lifetime in distributed implementations of applicative languages.* In Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Wentworth, 1981

[33] L Kott. *About R. Burstall and J. Darlington's Transformation System.* Universite Paris, 1978

[34] R A Kowalski. *Predicate logic as a programming language.* In Proc. IFIP, North-Holland, 1974

[35] R A Kowalski. *Algorithm = logic + control.* J ACM, Vol.22, 1979

[36] R A Kowalski. *Logic for problem solving.* Elsevier (New York, Amsterdam), 1979

[37] Z Marna, R Waldinger. *A Deductive Approach to Program Synthesis.* ACM TOPLAS, Vol. 2, No.1, 1980

[38] R Milne, C Strachey. *A theory of programming language semantics.* Chapman and Hall, London, and John Wiley, New York, 1976

[39] R G Milner. *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Science, Vol. 17, No. 3 1978

[40] A Pettorossi. *Transformation of Programs and Use of 'Tupling Strategy'.* Informatica 77 Conference Bled Jugoslavia, 1977

[41] A Pettorossi. *Methodologies for Transformation and Memoing in Applicative Languages.* Edinburgh University Ph.D Thesis, 1984

[42] S L Peyton-Jones. *Directions in functional programming research.* In Distributed computing systems programme, IEEE Digital Electronics and Computing Series 5, Peter Peregrinus Ltd, 1984

[43] S L Peyton-Jones. *The Implementation of Functional Programming Languages.* Internal Note 1730, University College London, 1985

[44] G Plotkin. *A Structural Approach to Operational Semantics.* DAIMI FN-19, Aarhus University, 1981

[45] M Reeve. *The ALICE Compiler Target Language.* Imperial College London, 1981

[46] J A Robinson. *A machine oriented logic based on the resolution principle.* J ACM, Vol. 12, No. 1, 1965

[47] W L Scherlis. ***Expression Procedures and Program Derivation.*** Stanford University Ph.D Thesis, 1980

[48] J Schwarz. ***Using Annotations to make Recursion Equations behave.*** IEEE Transactions on Software Engineering, Vol. SE-8, No. 1, 1982

[49] E Y Shapiro, A Takeuchi. ***Object-Oriented Programming in Concurrent Prolog.*** New Generation Computing, Vol. 1, No. 1, 1983

[50] I H Sorensen, B Sufrin. ***Formal Specification and Design of a Simple Assembler.*** Oxford University Programming Research Group, 1985

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

[52] W R Stoye, T J W Clarke, A C Norman. ***Some practical methods for rapid combinator reduction.*** In Proc. ACM Symposium on LISP and Functional Programming, Austin, 1984

[53] D A Turner. ***The SASL manual.*** St. Andrews University, 1976

[54] D A Turner. ***A new implementation technique for applicative languages.*** Software Practice and Experience, Vol. 9, 1979

[55] D A Turner. ***Aspects of the implementation of programming languages.*** Oxford University D.Phil Thesis, 1981

[56] D A Turner. ***Recursion Equations as a programming language.*** In Functional Programmming and its Applications, Cambridge University Press, 1982

[57] D A Turner. ***Miranda: a non-strict functional language with polymorphic types.*** In Proc. Functional Programming Languages and Computer Architecture, Nancy, 1985 (Lecture Notes in Computer Science 201, Springer-Verlag)

[58] P L Wadler. ***An Introduction to ORWELL.*** Oxford University Programming Research Group, 1985

[59] P L Wadler. ***A Splitting Headache: strict vs. lazy semantics for pattern matching in lazy languages.*** Oxford University Programming Research Group, 1985

[60] C P Wadsworth. ***Semantics and Pragmatics of the λ-calculus.*** Oxford University D.Phil Thesis, 1971

[61] M Wand. ***Continuation-Based Transformation Strategies.*** J ACM, Vol.27, No. 1, 1980

ACKNOWLEDGEMENTS

I am very grateful to Richard Bird: I could not have wished for a more helpful supervisor.

I have received much help and encouragement in carrying out my research, and should like to thank Paul Fertig, Michael Goldsmith, Tony Hoare, Nick Holt, Jeremy Jacob, Andrew Newman, Mike Spivey and Phil Wadler for their support. I am also grateful to John Hughes, who first introduced me to graph reduction.

I acknowledge the financial support of the Science and Engineering Research Council of Great Britain.