

NEWCASTLE UNIVERSITY LIBRARY

M 5

084 09682 1

Thesis L2706

Computer Architectures for
Functional and Logic Languages

by
Roger C. Millichamp

Ph.D. Thesis

December 1983

Computing Laboratory,
University of Newcastle Upon Tyne.

ABSTRACT

In recent years interest in functional and logic languages has grown considerably. Both classes of language offer advantages for programming and have an influential group of people promoting them. As yet no consensus has formed as to which class is best, and such a consensus may never form. Future general-purpose computer architectures may well be required to support both classes of language efficiently. Novel architectures designed to support both classes of languages could even add impetus to the area of hybrid functional/logic languages.

Treleaven et al[68] have proposed a classification of computational mechanisms which they believe underly several types of novel computer architecture (i.e. control flow, data flow and reduction). The classification partitions novel general-purpose architectures into the following classes: control driven - where a statement is executed when it is selected by flow(s) of control, data driven - where a statement is executed when some combination of its arguments are available, and demand driven - where a statement is executed when the result it produces is needed by another, already active instruction.

This thesis investigates the efficient support of both functional and logic languages using an architecture that attempts to be general purpose by embodying all the mechanisms that underly the above classification.

A novel packet communication architecture is presented which integrates the control driven, data driven and demand driven computational mechanisms. A software emulator for the machine was used as the basis for separate implementations of functional and logic languages, which were in turn used to evaluate the effectiveness of the computational

mechanisms described in the classification. These mechanisms allowed functional languages to be implemented with ease, but caused severe problems when used to support logic languages. The difficulties with these mechanisms are taken as signifying that they do not provide adequate support for logic languages. The problems encountered led to the development of a novel implementation technique for logic languages, which also proved to be a good basis for a combined functional and logic model. This model is believed to provide a sound foundation for a parallel computer system that would support functional and logic languages with equal elegance and efficiency, and would therefore also support hybrid languages. The design for such a computer is described at the end of this thesis.

ACKNOWLEDGEMENTS

I would like to thank Prof. Brian Randell and Dr. Philip Treleaven for their help during my research at Newcastle, particularly for suggesting the project upon which this thesis is based. I would like to express my gratitude to Prof. Peter Henderson for his invaluable assistance while I was attempting to understand functional programming and combinators, and to Dr. Simon Jones for several discussions concerning various aspects of logic. I am also indebted to those people who read drafts of my thesis and made useful comments.

The research reported in this thesis was supported by a grant from the Science and Engineering Research Council of Great Britain.

CONTENTS

1	INTRODUCTION.	1
1.1	Machine Architecture.	2
1.2	Computational Mechanisms.	3
1.3	Functional Languages.	4
1.4	Logic Languages	6
1.5	Outline of the Thesis	10
2	FUNCTIONAL AND LOGIC LANGUAGES.	12
2.1	Functional Languages.	12
2.1.1	Structure of Functions.	12
2.1.2	Lambda Conversion	20
2.1.3	Calculi of Lambda Conversion.	26
2.1.4	Characteristics of Functional Languages	27
2.2	Logic Languages	28
2.2.1	Logic Execution Viewed as a Search.	29
2.2.2	Logic Program Format.	32
2.2.3	Resolution Theorem Provers.	35
2.2.4	Unification Algorithm	39
2.2.5	Application of Resolution	43
2.2.6	Relation Names as Terms	45
2.2.7	Negation as Failure	45
2.2.8	Characteristics of Logic Languages.	49
3	CLASSIFICATION OF NOVEL COMPUTER ARCHITECTURES.	50
3.1	Models of Computation	50
3.1.1	Data Mechanisms	51
3.1.2	Control Mechanisms.	51
3.2	Control Flow.	53
3.3	Data Flow	54
3.4	Reduction	55
3.5	Using the Models of Computation	57
4	GENERAL-PURPOSE MACHINE ARCHITECTURE.	63
4.1	Data Format	65
4.2	Instruction Format.	66
4.3	Packet Format	69
4.4	Memory Organisation	70
4.5	Program Execution	71
4.6	Implementing the Models of Computation.	76
4.6.1	Control Flow.	76
4.6.2	Data Flow	77
4.6.3	Reduction	77
4.7	Operation Codes	79
4.8	Implementing Conditionals	82
4.9	Implementing Functions and Procedures	83
4.10	Assessment of the Architecture.	85
4.11	Rules for Architecture Modification	88

5	IMPLEMENTATION TECHNIQUES FOR FUNCTIONAL LANGUAGES.	89
5.1	Combinators	89
5.2	Graph Reduction	93
5.2.1	Graph Structure	93
5.2.2	Graph Manipulation.	96
5.2.3	Performing Reductions	100
5.2.4	Assessment of Graph Reduction	104
6	GRAPH REDUCTION ON THE MACHINE ARCHITECTURE	107
6.1	Instruction Format.	107
6.2	Program Format.	108
6.3	Instruction Execution	109
6.4	Implementing Functions.	111
6.5	A Problem with Lazy Evaluation.	112
6.6	Assessment of Combinator Implementation	113
6.6.1	Parallel Execution of Combinators	113
7	IMPLEMENTATION TECHNIQUES FOR LOGIC LANGUAGES	117
7.1	Summary of Logic Languages.	117
7.2	Search Tree	118
7.3	Unification	119
7.4	Structures.	121
7.5	Negation.	121
7.6	Variable Binding.	123
7.6.1	Copying Pure Code	123
7.6.2	Structure Sharing	124
7.6.3	Assessment of Variable Binding.	126
7.7	Parallelism in Logic Languages.	126
7.7.1	OR-Parallelism.	126
7.7.2	AND-Parallelism	127
7.8	Parallel Implementation	128
7.8.1	Storage Schemes	129
7.8.2	Control Mechanisms.	131
7.8.3	An Alternative Execution Scheme	133
8	LOGIC LANGUAGES ON THE MACHINE ARCHITECTURE	141
8.1	Instruction Format.	141
8.2	Clause Format	143
8.3	Program Format.	143
8.4	Process Format.	144
8.5	Execution Cycle	144
8.6	Implementing Unification.	146
8.7	Implementing Negation	147
8.8	Architecture Modification	149
8.9	Implementing Functors	151
8.10	Assessment.	155

9	COMBINED FUNCTIONAL AND LOGIC ARCHITECTURE.	158
9.1	Combining Functional and Logic Models	158
9.2	Structure of the Combined Architecture.	159
9.3	Structure of Programs	160
9.3.1	Functional Programs	160
9.3.2	Logic Programs.	164
9.4	Program Execution	167
9.4.1	Demand Forwarding	167
9.4.2	Parameter Passing	167
9.4.3	Calling Functions and Relations	168
9.5	Hybrid Programs	176
9.5.1	Simple Programs	176
9.5.2	Complex Programs.	179
9.5.3	Parallelism	180
9.6	Hybrid Languages.	181
9.6.1	Treating Programs as Data	184
9.7	Assessment.	185
10	CONCLUSIONS AND FUTURE WORK	189
10.1	Conclusions	189
10.2	AND-Parallelism	194
10.3	Combinators in Logic.	203
10.4	Hybrid Languages.	210
10.5	Hybrid Computer Architecture.	212
11	REFERENCES.	214

APPENDICES

1.	MACHINE ARCHITECTURE IMPLEMENTATION	220
1.1.	Instruction Format.	220
1.2.	Program Source Format	221
1.3.	Instruction Execution Cycle	222
1.4.	Calling a Procedure	224
1.5.	Returning from a Procedure.	230
1.6.	Emulator Errors	230
1.7.	Emulator Commands	230
1.8.	Example Programs.	233
2.	EXTENDED EXPLANATION OF COMBINATORS	246
2.1.	Compilation to Combinators.	246
2.2.	Recursion using Combinators	250
2.2.1.	Efficiency Considerations for Combinators	254
2.2.2.	Improved Abstraction Rules.	257
2.3.	Graph Reduction	262

3.	FUNCTIONAL LANGUAGE IMPLEMENTATION.	271
3.1.	Instruction Format.	271
3.2.	Program Format.	272
3.3.	Instruction Execution Cycle	273
3.4.	Garbage Collection.	275
3.5.	Example Program	274
4.	LOGIC LANGUAGE IMPLEMENTATION	281
4.1.	Activation Record Format.	281
4.2.	Instruction Source Format	283
4.3.	Program Source Format	284
4.4.	Instruction Execution Cycle	284
4.5.	Program Execution	287
4.6.	Garbage Collection.	287
4.7.	Example Program	287
5.	COMBINED FUNCTIONAL AND LOGIC LANGUAGE IMPLEMENTATION .	293
5.1.	Structure of Activation Records	293
5.2.	Format of Instructions.	294
5.3.	Token Format.	297
5.4.	Instruction Execution	297
5.5.	Instruction Opcodes	299
5.6.	Assessment.	304

CHAPTER ONE

INTRODUCTION

In recent years a considerable amount of interest has developed in two distinct, but related fields. The first is that of parallel machine architecture, and the second is functional and logic languages. As yet no consensus has formed as to which type of language is best, and one may never form. Future computer architectures may therefore be required to support both classes of language; particularly if hybrid functional/logic languages become desirable. This thesis investigates the design of architectures which support both functional and logic languages efficiently.

In 1981 Treleaven, Brownbridge and Hopkins[68] published a classification of parallel architectures in terms of several computational mechanisms which the authors felt to be fundamental to the implementation of control flow, data flow and reduction. The purpose of the work reported here is to investigate the claimed generality of these mechanisms to see if they can be used as a common base for both functional and logic languages.

The investigation was conducted by designing a machine architecture capable of providing equal support for all the computational mechanisms described in [68], and then writing a software emulator for the architecture. The mechanisms provided were then employed to implement both a functional and a logic language. In doing so it was possible to evaluate the usefulness of these computational mechanisms when implementing

functional and logic languages and also draw some conclusions about the claims of generality made by Treleaven et al. The claims were not substantiated and so the investigation was extended in order to produce a common base for both classes of language.

The purpose of this chapter is to give simple explanations of some concepts used throughout the thesis. These concepts are developed into more appropriate and sophisticated ones as the thesis progresses.

1.1. Machine Architecture

The form of machine architecture chosen for the investigation of functional and logic language implementation is based on packet communication [68]. This type of organisation consists of a circular instruction execution pipeline in which processors, communications channels and memories are interspersed with pools of work. This is illustrated in Figure 1.1. The organisation views an executing program as a number of independent information packets, all of which are conceptually active, and that split and merge. For a parallel computer, packet communication is a very simple strategy for allocating packets of work to resources. Each packet to be processed is placed with similar in ones in a pool of work. When a resource becomes idle it takes a packet from its input pool, processes it and places the modified packet in an output pool, and then returns to the idle state. Parallelism is obtained either by having a number of identical resources between pools or by replicating the circular pipelines and connecting them by the communications channels.

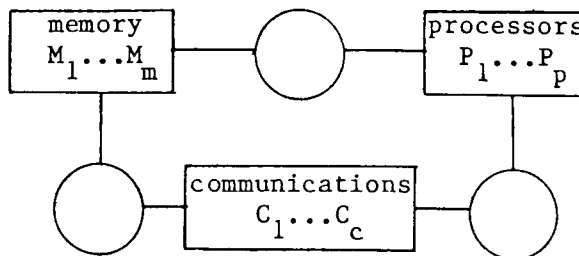


Figure 1.1: Simple packet communication architecture.

1.2. Computational Mechanisms

The computational mechanisms implemented by the packet communication architecture, and which are proposed by Treleaven et al as general purpose, are as follows:

- . Control Driven. Each instruction must wait for a certain number of control signals, each of which request the instruction to execute. Only when all signals have been received will the instruction be obeyed. When the execution of the instruction is completed it will signal other instructions to execute.
- . Data Driven. An instruction is only executed when it has received data for all its arguments. When the instruction has been executed, its result will be sent to further instructions.
- . Demand Driven. An instruction is executed when its result is demanded. This instruction may in turn demand its input data from further instructions, and so on until an instruction receives a demand, but does not propagate one. This instruction generates its result, and passes it back to the instruction which demanded it. Each instruction produces its result when its demands have been satisfied, until finally the program's result is generated.

In this architecture control driven, data driven and demand driven computational mechanisms are represented by control, data and demand tokens, respectively. A token is a message passed between one instruction and another.

1.3. Functional Languages

This section gives a superficial description of functional languages, and is intended to serve as a simple introduction to the subject.

A functional language, as the name implies, is based on the use and manipulation of functions, and as such has only one operator, that of function application. All other features of the language are provided as functions, be they primitive or user defined. Functional languages are closely related to Lambda Notation[15], a description of which will serve as an introduction to the subject. Lambda Notation illustrates a pure form of functional languages: a pure functional language is one whose functions do not have side-effects.

Consider the expression $4*x+4*y+3$. If this were represented as a function of x using Lambda Notation, it would have the form:

$$\lambda x.4*x+3*y+3$$

The λ can be considered as a binding operator. The identifier to right of the λ , x in this case, is the formal parameter. When the function is applied to an argument each occurrence of x in the function is replaced by the argument value. Thus:

$$(\lambda x.4*x+3*y+3) 5 \text{ is equivalent to } 4*5+3*y+3$$

To name such a function one would write the following:

$$f = \lambda x.4*x+3*y+3$$

Functions of more than one argument are defined as:

$$g = \lambda y.\lambda x.4*x+3*y+3$$

To invoke such a function one would write $g\ 4\ 3$, giving the result:

$$4*3+3*4+3$$

The argument supplied to a function may be a simple value, an expression, or another function which is to be used within the body of the called function. For example:

$$\begin{aligned} f &= \lambda x.+ 1\ x \\ h &= (\lambda g.+(g\ 1)\ 2)\ f \\ &= + (f\ 1)\ 2 \\ &= + ((\lambda x.+ 1\ x)1)\ 2 \\ &= + (+ 1\ 1)\ 2 \\ &= + 2\ 2 \\ &= 4 \end{aligned}$$

Any practical language will include some feature which allows conditional evaluation, for the purpose of this work the conditional form below will be used.

$$\text{if } p \text{ then } e_1 \text{ else } e_2$$

Notice that since a function must always return a result both the "then" and "else" arms of the conditional must always be present.

Execution of a Functional Program

This section explains how a functional program is evaluated to produce its result.

A functional program is usually built from a set of function definitions and an expression that calls them. The expression is evaluated to produce its result, and in doing so calls the functions to which it refers, binding the formal and actual arguments as it does so. The bodies of these functions are then evaluated themselves, calling more functions, and so on until no further calls are made. At this point the result of the program can be produced. For example:

$$\begin{aligned} f &= \lambda x. + 1 x \\ g &= \lambda x. + 2 x \\ * (f 1) (g 2) &= * ((\lambda x. + 1 x) 1) ((\lambda x. + 2 x) 2) \\ &= * (+ 1 1) (+ 2 2) \\ &= * 2 4 \\ &= 8 \end{aligned}$$

1.4. Logic Languages

Logic is the second type of language which the architecture described in the thesis aims to support. This section provides a simple explanation of logic languages and their execution.

The major difference between functional and logic languages is that the latter deals with relations rather than functions. A function takes some input and produces a result from it, a relation specifies how its arguments are related to one another. There is no concept of specific parameters being used for input or output values, any parameter may be used for either.

A logic program is built from a collection of relations, each of which consist of a set of clauses. Each clause specifies part of the behaviour of the relation. A clause has the form:

$$H: -G_1, G_2, \dots, G_n.$$

and is read as "H is true if $G_1 \dots G_n$ are true". That is: H is implied by the conjunction of G_1 to G_n . A clause is therefore sometimes called an implication. H is the head and G_1 to G_n form the body of the clause. The head contains the name of the relation to which the clause belongs, and a list of formal parameters:

$$h(A,b,c,d)$$

Logic commonly applies a convention to the use of identifiers: upper case identifiers are variables and lower case identifiers are literal constants or relation names. Each G_i is called a goal and contains the name of the relation which the goal calls, together with a list of actual parameters:

$$g(A,b,c,d)$$

A clause with no body is written as:

$$H.$$

Such a clause is an assertion and states that H is always true because there is no body which constrains H to only be true in certain circumstances. A clause without a head is written:

$$:-G_1, G_2, \dots, G_n.$$

and means that the clause body is never true, nothing can be implied from the body. Such a clause is used as the question which initiates the execution of the program. The question asks what values the parameters of G_1 to G_n must have in order for the question to be true. These values are the results the user requires.

A logic program consists of a number of relations and a clause body which asks the question the program must answer. It is the goals in the question which drive the execution of the program.

```
parent(fred,bert).
parent(fred,joan).
parent(bert,clive).
parent(joan,john).

grandparent(X,Y):-parent(X,Z),parent(Z,Y).

:-grandparent(fred,GP).
```

Figure 1.2: Complete logic program

For instance `parent(fred,bert)` means that fred is the parent of bert. The question in Figure 1.2 asks for all the grandparents of fred, these values are returned in GP.

Logic languages, for the purpose of this thesis, are restricted to Horn clause logic[39]. Horn clauses differ from general clauses in that Horn clauses are only allowed to have one head, general clauses may have any number of heads. The restriction to Horn clauses is one that is commonly made in for logic languages; the reasons for this restriction are explained in Chapter Two.

Execution of a Logic Program

The following section describes how a logic program is executed; the description is simple and is commonly used.

The execution of a program is driven by the execution of the program's question. In the program in Figure 1.2 the question has only one goal. This goal will call the grandparent relation which consists of a single clause.

When a clause is invoked by a call the formal parameters of the clause and actual parameters of the goal are matched by a process called unification. This matching will pass constant actual parameters into the clause, and arrange for the results of the clause to be passed out. If constants appear in the same positions in the formal and actual parameters they must have the same values. If the values are not the same the unification fails and the clause will not be executed, but this failure does not cause the whole relation to fail, all unifications that succeed will have their clauses executed.

In the case of the goal in the question the unification will be successful. The unification of the parameters causes the formal parameter X to be given the value "fred", and the parameter Y is bound to GP. The goals which form the body of the clause will now be executed to find values for Y, and in so doing will find values for Z that are acceptable to all the goals in the clause. The first goal will find two values for Z because the called relation, parent, has two clauses whose first formal parameter is fred. These values of Z will be bert and joan. The second goal will find two values for Y, namely clive and john, because the two values of Z are successfully unified with the clauses in the parent relation which produce these values as results.

This illustrates an important feature of logic programming, namely that one goal may produce several results, all of which must be consistent with the values produced by the other goals in the clause. If a set of values for the variables are acceptable to all the goals in the clause, then the clause is said to have succeeded; it has found a set of results. Since the unification of Y and GP link both variables together, the results for Y are sent to GP. Upon completion of the grandparent clause the original question is complete and the values of

GP printed. These values are clive and john.

Negation of a Goal

Until now we have implied that a clause will only succeed if all the goals in its body succeed. This is not always appropriate. Often a situation arises in which a clause should succeed only if some of the goals in its body fail. To meet this requirement many practical logic systems implement negation, but in doing so step outside the bounds of Horn clause logic (for reasons explained in Chapter Two).

A negated goal is written as $\sim g(t_1, \dots, t_n)$ and is interpreted as: if the call on g fails then $\sim g(t_1, \dots, t_n)$ succeeds, and if the g succeeds then $\sim g(t_1, \dots, t_n)$ fails. Negation is interpreted as failure. A clause body is now said to be built from literals: where a literal may be a goal or a negated goal. An example of negation may be taken from the telephone system. The phone rings if the number is correct and the phone is not engaged. A clause which represents this is written:

$$\text{ring}(P,N):\sim\text{correct}(N),\sim\text{engaged}(P).$$

1.5. Outline of the Thesis

Having briefly presented the background to the thesis, its structure will now be described.

Chapter Two describes the background theory for both functional and logic languages.

Chapter Three explains the classification of computational mechanisms by Treleaven et al. Chapter Four describes a novel architecture which implements these computational mechanisms.

Chapter Five surveys implementation techniques for functional languages. Chapter Six explains how the chosen implementation technique was transferred to the architecture.

Chapter Seven surveys implementation techniques for logic languages. Chapter Eight describes how the selected technique was transferred to the architecture.

Chapter Nine describes a novel architecture which combines functional and logic languages. Lastly Chapter Ten summarises the conclusions drawn from the work reported in this thesis and gives an indication of the direction of future work.

CHAPTER TWO
FUNCTIONAL AND LOGIC LANGUAGES

This chapter describes the background theory for both functional and logic languages. The chapter explains most of the important concepts which form the basis of the language implementations described later in the thesis.

2.1. Functional Languages

This section describes the terms and theory that underly functional languages using the Lambda notation introduced in Chapter One. The topics covered include important aspects of program representation and a comparison of two evaluation strategies, and in particular the termination properties of those strategies.

2.1.1. Structure of Functions

In chapter one the expression

$$\lambda x.4*x+3*y+3$$

was given as an example of a function. Here x is the bound variable of the function; the expression to the right of the "." is the body of the function, and y is said to be free in the function because it is not an argument. The complete Lambda expression is said to be composed by abstraction: that is the bound variable x is abstracted from the body producing a function of one argument. The reverse of abstraction is sub-

stitution and is carried out when a function is applied to an argument. Again this was illustrated in Chapter One:

$$\begin{aligned} (\lambda x.4*x+3*y+3) 5 \\ \text{gives} \\ 4*5+3*5+3 \end{aligned}$$

It is important to distinguish between a function:

$$f=\lambda x.4*x+3*x+3$$

and the expression $f\ 3$, which denotes the result of applying f to 3 .

Arguments and Results

The class of argument values for which a function is defined is called its domain. A function is said to be defined for a particular argument value if it is able to return a result for that argument. The class of values from which the result is selected is called the function's range. A function will map each member of its domain onto the single corresponding member of its range.

A function's type is denoted by the expression:

$$A \rightarrow B$$

which means that A is domain of the function and B is the range of the function. A function which both takes and returns an integer would have the type:

$$\text{integer} \rightarrow \text{integer}$$

A function with several arguments will have a type:

$$A_1 * \dots * A_n \rightarrow B$$

as for example:

$$\text{integer} * \text{integer} \rightarrow \text{real}$$

A function is said to be partial if it is unable to map every element of its domain into its range. Suppose the domain of the reciprocal function is considered to be the class of reals. Then the function is partial because it is unable to map the value 0 into its range. A function which is able to establish the correspondence between each element of the domain and a value in its range is called a total function.

Higher Order Functions

A higher order function is a function which allows other functions to be either its result or an argument or both. This allows functions to be in both the domain and range of a function. Higher order functions are the most powerful feature of functional languages.

Consider the example:

$$(\lambda x. \text{if } x=1 \text{ then } (\lambda y.y+1) \text{ else } (\lambda y.y-1)) 1 2$$

Here the result of the first function is $(\lambda y.y-1)$; which is then applied to 2. Thus the first function yields another function as its result.

Higher order functions permit two features commonly found in functional languages to be incorporated with ease. The first feature is functions of two or more arguments which are written as shown below:

$$g = \lambda y. \lambda x. 4*x+4*y+3$$

A function of one parameter is composed by abstracting the bound variable from its body. To produce a function of two arguments: initially compose a Lambda expression by abstracting the first bound variable x . Then abstract the second bound variable, y , from the result; which yields g , a function of two arguments. When such a function is applied; a function with n arguments returns a function of $n-1$ arguments, which

in turn yields a function of $n-2$ arguments and so on. In the above example if g was applied to an argument, say 1, the result will be a function of one argument:

$$\lambda x.4*x+4*1+3$$

Constructing a multi-argument function in this way is called Currying (after the mathematician Curry, but in fact due to Schonfinkel[66]).

The second language feature provided by higher order functions is the ability to declare functions which are local to others. This enables one to qualify a function with auxiliary definitions, for example:

```
fun =  $\lambda x.$ (if x=1 then f else g)
      where f =  $\lambda y.y+1$ 
           g =  $\lambda y.y-1$ 
```

Figure 2.1: A qualified function.

This program fragment may be written as a Lambda expression:

```
 $\lambda f.\lambda g.\lambda x.$ (if x=1 then f else g)( $\lambda y.y+1$ )( $\lambda y.y-1$ )
= $\lambda x.$ (if x=1 then ( $\lambda y.y+1$ ) else ( $\lambda y.y-1$ ))
```

Thus higher order functions allow local functions to be declared by passing them as argument values to the function that uses them.

Closures

Closures are an important concept for the implementation of functional languages, and one which will be referred to several times in the following chapters. A closure is used as a way of representing a function at the time it is defined, and at any point during its execution.

Whenever the function `fun`, in Figure 2.1, is applied the environment in which its body is obeyed must include `f`, `g` and `x`. The auxiliary definitions `f` and `g` are provided by a static binding established at definition; while `x` is provided when the function is called. The function `fun` must therefore be represented by a structure which holds the code for its body together with an environment which represents the binding of `f` and `g`. This structure is called a closure:

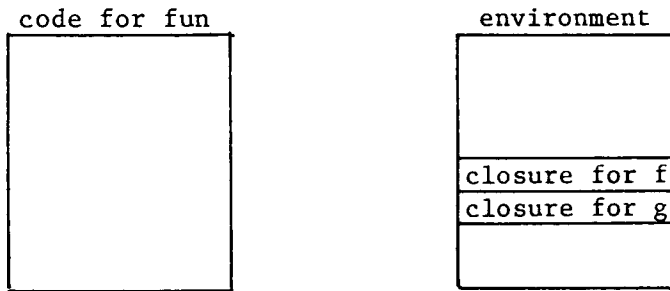


Figure 2.2: Closure for the function `fun`.

A closure is used to represent a function whenever its execution is suspended. Since a function could be considered suspended between its definition and its call, it is represented by a closure. If a function is applied but later suspended for some reason, it will again be represented by a closure, but in this case the closure will be a copy of the original; with the argument bound into it. A closure is necessary because when the execution of the function is restarted, the execution must take place in the same context as that in which it was suspended. A closure allows the context to be carried from the point of suspension to the point of continuation.

When a function `g` is returned as the result of another function `f`, `g`'s execution is suspended until it is applied to its own arguments, so `g` must be represented as a closure. When `g` is applied its execution must continue in the environment in which it was created. This is the

environment created by the call of f , supplemented by g 's arguments. In the example below, the environment of g will contain the binding of x to 1. When g is called it is this binding that gives x its value, not the binding which exists at the point in the program where g (the result of f 1) is used.

```
f = λx.g
    where g = λy.(sin(x)+y)
x = 2

(f 1) 3 => g 3
```

The result will be $\sin(1)+3$.

Scope in Functional Languages

Auxiliary definitions introduce the notion of scope. In the example in Figure 2.1 the scope of f and g are restricted to the body of the Lambda expression. This is the simplest form of scoping; f and g may not to be called from their own bodies: recursion is therefore impossible. The bodies of f and g may however be qualified by further functions:

```
fun = λx.(if x=1 then f else g)
      where
      f = λy.(h y)+1
                where h = y/2
      g = λy.(h y)+1
                where h = y*y
```

Here the scopes of both h 's are restricted to the bodies of the Lambda expressions which they qualify.

Recursive qualifications are possible however using the qualifier `whererec` instead of `where`:

```

fun = λx.(if x=1 then f else g)
      whererec
      f = λy.if y=0 then 1 else f(g 1)
      g = λy.y-1

```

Here the bodies of the functions defined in the whererec may contain references to themselves and to the other functions defined along side them. The environment for f in the where qualification will only contain y, but for the whererec qualification it will also contain f and g. In short whererec introduces a cyclic environment which contains the qualified function. The closure for fun will therefore have the form shown in Figure 2.3.

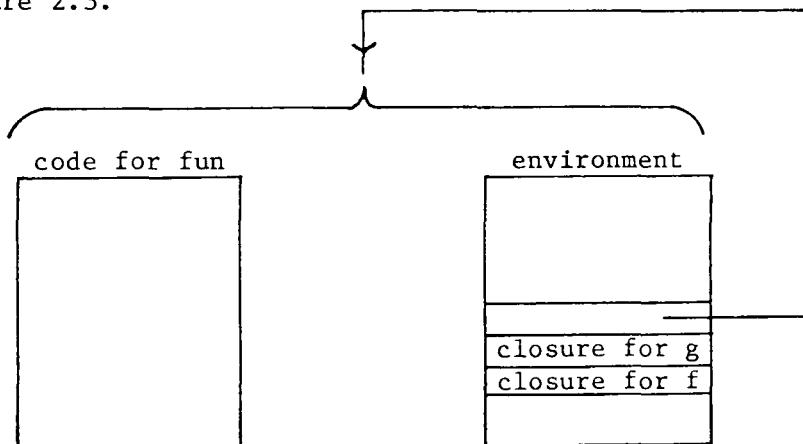


Figure 2.3: Recursive closure for f.

Hence forth all qualified expressions will be considered to be qualified recursively. The qualifier "where" will now be taken to mean whererec.

Function Composition

Function composition defines some rules which control the way function applications may be combined to produce expressions.

Brackets in expressions are left associative by default:

```
f g x
```

has the same meaning as the expression below:

$(f\ g)x$

The "." operator in the expression $f.g\ x$ means apply g to x and apply f to the result:

$f.g\ x = f(g\ x)$

The "." is therefore less binding than function application.

Representing Recursion

This section describes how recursion, which is an important feature of functional languages, may be represented by Lambda Notation. Recursion may be represented in in two ways, firstly by using the name of the function in it own body:

$f = \dots f \dots$

and secondly by supplying the function as an argument to itself:

$f = (\lambda g. \dots g \dots)f$

This is another example of the use of higher order functions and is one which allows a non-recursive function to be made recursive by self-application.

Recursion introduces the possibility of a situation called "Russell's paradox". Consider the function:

$\text{selfzero} = \lambda f. \text{if } f(f) = 0 \text{ then } 1 \text{ else } 0$

If this function is applied to itself, the following expression is obtained:

$\text{selfzero}(\text{selfzero}) = (\text{if } \text{selfzero}(\text{selfzero}) = 0 \text{ then } 1 \text{ else } 0)$

If the predicate is true because $\text{selfzero}(\text{selfzero})=0$ then the result of the function call, $\text{selfzero}(\text{selfzero})$, is 1. The application

selfzero(selfzero) is therefore both 1 and 0 at the same time. Had the function not been used within itself this situation would not have arisen. A practical interpreter could attempt to execute such a function call but it would never terminate.

2.1.2. Lambda Conversion

In this section the evaluation of a Lambda expression is described in more detail. Any architecture which evaluates a functional program must follow the rules explained in this section.

The value of the expression depends solely on the values of its subexpressions. Therefore rules a) to c) hold:

- a) if $M=N$ then $\lambda x.M = \lambda x.N$
- b) if $F=G$ then $F A = G A$
- c) if $A=B$ then $F A = F B$

There are three additional rules for converting one Lambda expression into equivalent expressions.

- 1) $\lambda x.M = \lambda y \{y/x\} M$ providing y does not occur within M .

This means that the choice of bound variable in a Lambda expression does not change the meaning of the expression. The term $\{y/x\}$ is a singleton substitution replacing x by y in M . In general a substitution is a set of term/variable pairs which replace all occurrences of the variable by the associated term in an expression that is applied to the substitution. One may also change the free variable in an expression without changing its meaning. The only qualification is that the new variable must not already occur in M . This is the Alpha (α) rule.

- 2) $(\lambda x.M)N = M\{N/x\}$ provided the bound variables of M are distinct from the free variables of N .

This means that the binding of an argument into a Lambda expression has the same effect as substituting the argument value into the body of the function. This is called the Beta (β) rule. The constraint on the application of this rule is necessary because if N has a free variable which is bound in M , then each occurrence of the variable in N will become bound when the substitution is carried out. If the function under consideration is Curried then the Beta (β) rule must be applied several times, once for each nested Lambda expression.

- 3) $(M \text{ where } x=N) = \lambda x.M N$

The expression on the left has the same meaning as M would have if N were substituted for x throughout. This rule follows from the description of auxiliary functions as arguments to Lambda expression, and from rule 2) above.

In rule 2) the problem that occurred when passing one function to another can be illustrated as follows. Suppose that a Lambda expression is being passed as an argument:

$$\begin{array}{c} x = 1 \\ (\lambda g.\lambda x.g+4*x_1)(x_2*3)3 \end{array}$$

Here there are two occurrences of the variable x , denoted by x_1 and x_2 to distinguish them. The variable x_1 is bound in the called function and x_2 is free in its argument. After one substitution the result is:

$$\begin{array}{c} x = 1 \\ (\lambda x.x_2*3+4*x_1)3 \end{array}$$

The variable x_2 is free in the first expression (and has the value 1),

but bound in the second (and has the value 3). Consequently the variable's value has been changed because it has been passed in a parameter. To avoid this problem the free variables in the arguments of the function f must be distinct from f 's bound variables. This may be accomplished in a practical implementation of the Lambda notation by systematically replacing each variable by a unique number at compile time. This number is generated from the type of variable, free or bound, and from its position in the expression.

If an expression A can be obtained from another expression B by the application of rules one to three, then A and B are said to be convertible. If rule 2) is applied in such a way as to replace a function application by the body of the function after the argument has been substituted, the expression is said to have been reduced. If rule 2) is used in the reverse direction the expression is said to have been expanded. If A is convertible to B using only rule 1) plus reduction steps, then A is reducible to B .

The repetition of reduction steps provides a method of evaluating an expression and producing its normal form. The normal form of an expression has the same meaning as the original, but it is now in its simplest form. The normal form can therefore be considered to be the expression's result. Reduction therefore provides a method of transforming an expression into its result. In producing the result the rules 1-3 above never change the meaning of an expression, only its form. The expression $(+ 1 2)$ has the same meaning as 3. There is consequently no distinction between expressions and data; they are just different ways of denoting the same thing. However some expressions have no normal form. For example reduction will not change the expression:

$(\lambda x. x x)(\lambda x. x x)$

After one reduction the result would be:

$(\lambda x. x x)(\lambda x. x x)$

Evaluation Strategies

Given that there are a number of function applications in an expression a choice must be made as to the order in which their reductions are to be carried out. The Church-Rosser theorem[16] states that if two different evaluation sequences are used on a given Lambda expression, and both give a result in its normal form, then both normal forms will be equivalent up to the renaming of variables.

In the interpretation of functional languages there are two evaluation sequences of interest: innermost and outermost; both of which are described below. The former evaluates an expression by evaluating the lowest level subexpressions first; the ones which have no subexpressions nested within them. All data in these subexpressions will be directly available. When the expressions at the lowest level have been evaluated, the expressions in the level above will have direct access to the results, so they can be evaluated, and so on until the final result is produced:

$$\begin{array}{l} (* (+ 1 2) (- 3 4)) \\ \quad (* 3 -1) \\ \quad \quad -3 \end{array}$$

In this example the + and - are obeyed first followed by the *. Innermost reduction is driven by the availability of data.

Innermost evaluation unfortunately introduces a problem. It is not always appropriate to obey a subexpression simply because it has all the necessary data. An example of this is the conditional expression:

if p then e_1 else e_2

When a conditional expression is executed the predicate should be obeyed first and the appropriate arm of the conditional then selected according to the result. However if the evaluation followed the innermost rule then the predicate and both arms of the conditional would be evaluated in parallel. This could lead to an erroneous program if one of the arms caused an error, or did not terminate. If the predicate was executed first and selected appropriate arm for execution the situation may not have arisen. Thus innermost evaluation takes no account of the context of an expression; if an expression can be executed it will be.

Outermost evaluation will always obey the outermost function application first, and in so doing it will request the values of its arguments. The arguments may also be expressions which will in turn request the values of their subexpressions, and so on until a value is found for all the arguments of a function. The process is then reversed, each expression returning a result to its parent. For example in:

* a b
a: + 1 2
b: - 3 4

the * is the outermost function and is evaluated first, and so requests the values of a and b, which are then evaluated:

* a b
a: 3
b: -1

This in turn allow the result of the complete expression to be found

* 3 -1
-3

Outermost evaluation allows the implementation of the by-need mechanism in which each function only requests the value of those arguments which are necessary to produce the result. In particular a conditional expression would evaluate its predicate and then evaluate either the "then" or "else" expression. In this way redundant, or possibly erroneous, computation is avoided. Such a computation is known as safe because errors occur only if they are unavoidable. Evaluation by-need implies the use of the by-name parameter passing mechanism because each parameter must only be evaluated when its value is required and must therefore be passed in an unevaluated form.

A variation of the by-need evaluation strategy is lazy evaluation which can be considered as using the by-name mechanism, but replacing the expression with its result once the result has been produced. A shared expression is therefore evaluated when its result is required, but once the result has been produced it is remembered so as to avoid repeatedly calculating it when required by other users.

The Church-Rosser theorem states that if an expression has a normal form then the by-need evaluation scheme will find it. This is true because evaluating the outermost expression first allows the evaluation of those subexpressions whose result is not required to be stopped at the earliest opportunity. This avoids evaluating any expression that does not have a normal form. The selection of an evaluation scheme will therefore affect the termination properties of a program, but not its result if the program does terminate. Outermost evaluation will terminate for the largest possible class of programs.

The implementation of by-need evaluation requires that the execution of the function application be delayed until its result is required. In addition execution must take place in the environment in which the function was created and not that of its use. For this reason a closure must be built to represent the suspended function. When the suspended function's result is required the closure is executed to produce it.

If the "lazy" variation of the by-need mechanism is to be implemented each closure must be flagged to show if it has been reduced to its result. If the flag is set, the closure must be reduced to the result. If not the result may be accessed directly. This form of a closure is called a recipe [35].

The above section describes how an expression may be evaluated. The next section relates the rules which govern the termination of program to rules concerning the program's structure.

2.1.3. Calculi of Lambda Conversion

There are two calculi of Lambda conversion, namely λI and λK . Both of these are based solely on the three rules of Lambda conversion and therefore do not contain conditional expressions. In the former there must be at least one occurrence of each bound variable in the function body. Each bound variable represents a subexpression which is supplied as an argument. This implies that if the complete expression is to have a normal form then each subexpression must also have a normal form because each subexpression must be present. The λI calculus corresponds to strict functions. A strict function is one which must have values for all its arguments before it will produce a result. Since every

subexpression must be reducible to normal form it does not matter the order in which the reductions are carried out. Any evaluation strategy may therefore be used.

The latter calculus, $\lambda\mathbf{K}$, relaxes the restriction placed on bound variables by $\lambda\mathbf{I}$ and allows a bound variable to be omitted from the function body. This means that not every subexpression need have a normal form if the complete expression has a normal form; some subexpressions can be ignored. A language which allows conditional expressions implies the use of $\lambda\mathbf{K}$ conversion because one of the arms of the conditional must be cancelled. An expression conforming to the rules of $\lambda\mathbf{K}$ must be evaluated by-need, for the reasons explained by the second section of the Church-Rosser theorem. Namely one must cancel redundant expressions at the earliest opportunity to maximise the possibility of termination. For example:

$$\lambda x y.x$$

obeys the rules of $\lambda\mathbf{K}$, since the value of y is not required to produce the result of the function.

2.1.4. Characteristics of Functional Languages

An important characteristic of a functional language is the lack of an assignment statement. This means that there can be no side effects from a function, which ensures that a function's result is defined solely by its arguments. A function therefore has the same meaning no matter where it is used, the so-called referential transparency property of functional languages.

Referential transparency is important if both higher order functions and by-need evaluation are to be of practical use. Both these features leave a function in a suspended state: higher order functions because a function may be returned as a result but only applied at a later time; and by-need evaluation because the evaluation of the expression is delayed until its result is required. As a result of this, it is difficult to know when a particular expression is going to be evaluated because the expression will be passed around in this suspended state. Referential transparency ensures that a function's behaviour remains unchanged while it is suspended, because it does not allow any of the global variables to which the function refers to be changed. Any modification to global variables would make the task of writing a large program almost impossible, because the programmer could not predict what result a suspended expression would yield. Thus referential transparency is important if the use of higher order functions and by-need evaluation is to be practical.

Referential transparency allows the manipulation of programs as mathematical entities. Each function is an equation and can be manipulated to change its form without changing its meaning. This allows the correctness of a program to be proved and also permits use of transformations which modify the behaviour of a program without affecting its correctness.

2.2. Logic Languages

This section describes the theory which underlies the use of horn clauses as a programming language, a development which was made possible by the introduction of Resolution theorem proving techniques by Robinson in 1963[60]. This in turn relies on the Unification algorithm that was

described briefly in Chapter One.

The explanation of logic starts with a description of an alternative view of the execution of a logic program; one that is used for the remainder of the thesis. The unification algorithm is described in enough detail to allow it to be implemented, and important aspects of the implementation of negation are also explained.

2.2.1. Logic Execution Viewed as a Search

In Chapter One the execution of a logic program was described in terms of procedure calls. Here the execution of a logic program is viewed as a search for the program's result. By viewing the execution of the program as a search, an explanation of a program's execution is able to describe what the program does, not how it does it. The use of searches to describe the execution of a logic program also reveals a connection with reduction.

The description of program execution by searching has two parts: the first describes how a clause may be used to specify the results the search must produce, and the second introduces a graphical representation of the search itself.

The question posed by the user can be regarded as a specification to which the program's results must conform. The search proceeds by repeatedly transforming the specification into an equivalent ones that are a step closure solution. The search for a program's result can be illustrated using the program:

```
parent(fred,bert).
parent(fred,joan).
parent(bert,clive).
parent(joan,john).

grandparent(X,Y):-parent(X,Z),parent(Z,Y).

:-grandparent(fred,GP).
```

The question asks for the grandparents of fred, which is the initial specification of the results the program must produce. When the question is executed the grandparent relation is invoked, which transforms the specification into:

```
parent(fred,Z),parent(Z,GP)
```

The new specification still specifies the grandparent of fred, but in a different form. The grandparent of fred is the parent of fred's parent. If the first goal in the new specification is executed Z will be given two values: bert and joan, because the goal parent(fred,Z) matches with two assertions in the parent relation. These two values give rise to two versions of the second goal in the specification above, both are specifications of the grandparent of fred:

```
parent(bert,GP)
parent(joan,GP)
```

When these goals are executed GP is given two values, one from each goal. These values are clive and john because the two clauses in the parent relation which match the goals listed above give GP these two values. The grandparents of fred are therefore clive and john, satisfying all the specifications produced during the execution of the program.

The description of the execution of a logic program given above is related to reduction because of the way specifications are transformed into equivalent ones. In reduction expressions are reduced by transforming them into equivalent ones. The difference between

searching and reduction is that in reduction the expression is simplified until it becomes the result; while in logic the specification is simplified until it specifies the result in a way that can be satisfied directly by the assertions of the program.

A diagram of the execution of a program will take the form of a tree; the so-called search tree, the structure of which reflects the structure of the search for the result. For example the execution of the grandparent program will produce the search tree shown in below.

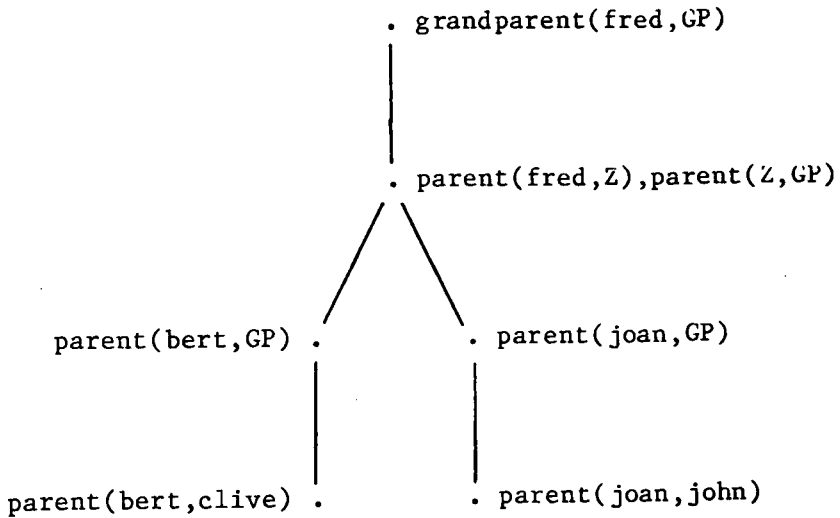


Figure 2.4: Search tree for "grandparent(fred,GP)".

Each node in Figure 2.4 represents a specification of the results the search must produce; the arcs of the tree join one generation of specifications to the next. The top node is the question posed by the user and beneath it is the specification of a grandparent: the body of the sole clause in the grandparent relation. Both specify the grandparent of fred. At this point the tree divides into two branches because two assertions match the first goal of the grandparent clause. The first goal of the grandparent clause produces two values for Z: these are bert and joan. The nodes at the top of each branch are the

second goal in the grandparent clause with the values for Z substituted. The two specifications in these nodes still specify the grandparents of fred, they are the parents of bert and joan. The nodes at the bottom of each branch of the tree are the final specifications of the grandparents of fred, they give the parents of bert and joan, namely clive and john, which are the final results of the program. Notice each leaf only produces one result, because only the leaves will match the assertions of the program which produce the result values. In a full search tree there will be branches which lead to failure. They have not been included in this example for reasons of clarity.

Theoretical Basis for Logic

Most logic interpreters are based around resolution theorem provers. The resolution algorithm may be used to prove theorems, but is in fact a refutation procedure. A refutation procedure is an algorithm which disproves theorems. To use a refutation procedure to prove a theorem correct, the theorem must be negated and the negative refuted. This is what occurs during the execution of a logic program.

2.2.2. Logic Program Format

Before describing Resolution it is important to understand why a logic program takes the form it does. This description of logic program format has two sections, the first gives a more detailed description of the syntax of a goal, and the second describes why the question of a logic program appears on the right of an implication symbol.

In Chapter One the format of a program was described as consisting of a collection of relations, and a question about them. This would be written as a set of clauses as shown in the grandparent program above.

Each goal or head has the form:

$$\text{name}(\text{term}_1, \text{term}_2, \dots, \text{term}_n)$$

In the case of the head, the name specifies the relation to which the clause belongs. All clauses with the same name at their head belong to the same relation, the relation being known by this name. The terms in the head define the formal parameters of the clause. Thus for a goal the name specifies the relation which is to be called and the terms form the actual parameters.

A term may consist of a variable, a constant or a more complex structure built from function applications (or functors as they are referred to in logic). Functors are not functions in the usual sense, since they do not return values but are in fact constructors. Constructors are functions which build structures; cons is a popular example which returns a cell with two components. A head or goal can therefore have the form such as shown below.

$$a(l, X, \text{cons}(Y, \text{cons}(Z, l)))$$

Here "a" is the relation name, l is a constant parameter, X is a variable being passed as a parameter and cons is a functor. The third argument is a structure built from nested functor applications. In general, a term which uses a functor may have the form:

$$\text{functor}(\text{term}_1, \dots, \text{term}_n)$$

A term may not have the name of a relation as its value. Logic there-

fore does not have higher order relations, the logic equivalent of higher order functions.

The following paragraphs explain why the question given to a logic program must appear on the right of an implication. Suppose that all the relations of the program are represented by the clauses A_1 to A_n , and the question by a conjunction of literals B . If all the implications in the program hold for the data supplied by B , then one may state that the clauses imply B also holds, which may be written:

$$B:-(A_1 \wedge \dots \wedge A_n)$$

where \wedge means conjunction (and). It is the task of the theorem prover to show that this implication does indeed hold. The resolution technique is a refutation procedure which can only prove that a theorem does not hold. To prove a theorem by refutation one must disprove its negative. Resolution must therefore prove that $\sim(B:-(A_1 \wedge \dots \wedge A_n))$ does not hold.

An implication may be expressed as the truth table given below, in which truth values for P and Q are specified together with the third column which signifies if the implication holds.

P	Q	$P: \sim Q$	$\sim Q \vee P$
T	T	T	T
T	F	T	T
F	T	F	F
F	F	T	T

\vee means disjunction (or) and \sim means not

So an implication may also be written as $\sim Q \vee P$ which may be represented by the same truth table.

Given that $P:-Q$ is the same as $\sim Q \vee P$, we may rewrite:

$$B:- (A_1 \wedge \dots \wedge A_n)$$

as:

$$B \vee \sim (A_1 \wedge \dots \wedge A_n)$$

which of course must be negated for a refutation procedure such as resolution. This gives:

$$\sim (B \vee \sim (A_1 \wedge \dots \wedge A_n))$$

Simplifying

$$\begin{aligned} \sim (B \vee \sim (A_1 \wedge \dots \wedge A_n)) &= \sim B \wedge \sim (\sim (A_1 \wedge \dots \wedge A_n)) \\ &= \sim B \wedge A_1 \wedge \dots \wedge A_n \end{aligned}$$

which corresponds to the clauses:

$$\begin{array}{c} A_1 \\ \cdot \\ \cdot \\ \cdot \\ A_n \\ :-B \end{array}$$

So to apply the resolution theorem proving technique, the question B must be negated to turn it into a refutation, which is why B appears on the right hand side of an implication without a head. As was explained in Chapter One, such an implication is never true, it specifies that B does not hold and therefore that $\sim B$ does hold.

2.2.3. Resolution Theorem Provers

Resolution is the theory which underlies the use of logic as a programming language: most schemes for implementing logic, will in fact, implement resolution in some way.

As was mentioned earlier resolution is a refutation procedure which means it proves that a set of disjunctions of literals is false (that they are inconsistent). Resolution operates by finding complementary literals and cancelling them.

The program $\sim B \wedge A_1 \wedge \dots \wedge A_n$ must therefore be transformed into a set of disjunctions. We may rewrite the conjunction of literals as a set, without any loss of information. Thus the expression now has the form:

$$\{\sim B, A_1, \dots, A_n\}$$

All that remains is to transform each clause into a disjunction. This may be done by representing the implication $P:-Q$ as $P \vee \sim Q$. The symbol Q denotes a clause body, which is now negated. If

$$Q = Q_1 \wedge \dots \wedge Q_n$$

then by De Morgan's Theorem

$$\sim Q = \sim Q_1 \vee \dots \vee \sim Q_n$$

so the implication may now be rewritten as

$$\sim Q_1 \vee \dots \vee \sim Q_n \vee P$$

which is a disjunction. The program has now been transformed into a set of disjunctions; the form the resolution algorithm requires.

Resolution is based on the notion of a clash. A clash may be defined as follows: given a set of clauses

$$\{A_1, \dots, A_n, B\}$$

each literal in B , called L , must have a complement, $\sim L$, which appears in only one of A_i . For any clash:

$$\{A_1, \dots, A_n, \sim B\},$$

we may construct the resolvent:

$$[A_1, \dots, A_n, \sim B],$$

defined as:

$$(A_1 - \{\sim L_1\}) \vee \dots \vee (A_n - \{\sim L_n\}) \vee (B - \{L_1, \dots, L_n\})$$

It is known that if the resolvent is the empty clause, denoted by \square , then there is no way of assigning truth values to the literals in A_1 to A_n and B , so as to make the expression $(A_1 \wedge \dots \wedge A_n \wedge \sim B)$ true (see [14] for a proof). In short we have now disproved:

$$\sim(B : -A_1, \dots, A_n)$$

so by refutation we have proved that:

$$B : -A_1, \dots, A_n$$

and so the question B is implied by the clauses A_i of the program. If no resolvent \square is found after all clashes have been resolved then B is not implied by A_1, \dots, A_n .

Instead of proving a theorem in one resolution step it may be proved by repeated application of the resolution principle to clashes of individual literals, as shown below[14]:

$$\begin{aligned} S &: -P. \\ U &: -S. \\ P &. \\ &: -U. \end{aligned}$$

Rewriting the implications as disjunctions we have:

- 1) $S \vee \sim P$
- 2) $U \vee \sim S$
- 3) P
- 4) $\sim U$

which together constitute the set of disjunctions. Now it is possible to begin to resolve the clashes: initially that between 2) and 4) is resolved giving the resolvent:

5) $\sim S$

which replaces clauses 2) and 4). A resolvent may replace a clash because the resolvent is a logical consequence of the two clauses which clash (see [14] for a proof). Next the clash between 5) and 1) which gives the sixth resolvent:

6) $\sim P$

This leaves the clash between 6) and 3) whose resolvent is \square . Since each clash is replaced but its resolvent, and the resolvent is a logical consequence of the clash, clauses 6) and 3) are a logical consequence of the original program. The resolvent of clauses 6) and 3) is therefore the resolvent of the whole program. Thus \square is the resolvent of clauses 1) to 3), and so U follows from 1) to 3), as can be verified by inspecting the original program.

This entire process is called a deduction and can be illustrated graphically as shown in Figure 2.5 [14].

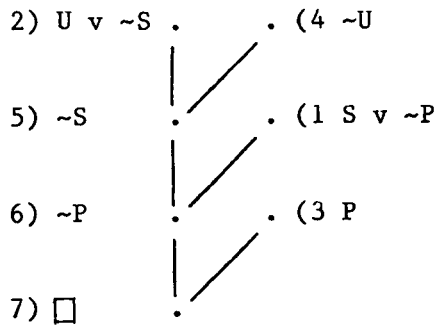


Figure 2.5: Graphic representation of a deduction.

Here node 7) is the empty clause. Clauses 2) and 4) form a clash whose resolvent is clause 5). This in turn clashes with clause 1) with

resolvent clause 6). The resolvent of the clash between clauses 6) and 3) is the empty clause.

Resolution therefore allows a theorem to be refuted by cancelling complementary literals. If the result is the empty clause the theorem does not hold.

2.2.4. Unification Algorithm

The role of the Unification algorithm in resolution is to recognise clashes but, for reasons of simplicity, many descriptions of logic languages describe unification as a parameter passing mechanism. A more formal and precise description is given here. The unification algorithm is the feature of logic languages which allow them to deal with relations rather than functions.

The clashes which occur during resolution are independent of the choice of variables, but dependent on the constant terms in the literals, which must be equal. The unification algorithm ensures that corresponding constant terms in the literals are the same, and renames variables in such a way as to allow clashes between literals which use different variables to be identified.

Given a set of terms S , the unification algorithm will find a substitution that will make all the terms identical. Such a substitution is called the unifier of S . If there is more than one unifier for any given S then the most general unifier is the one which has the smallest number of substitution pairs $\{t/v\}$. The unification algorithm finds the most general unifier.

For example, take the set of terms

$$\{p(A, X, f(g(Y))), p(Z, f(Z), f(U))\}$$

which when applied to the substitution $\{1/X, 2/A\}$:

$$\{p(A, X, f(g(Y))), p(Z, f(Z), f(U))\} \{1/X, 2/A\}$$

produces the set:

$$\{p(2, 1, f(g(Y))), p(Z, f(Z), f(U))\}$$

Here all occurrences of X have been replaced by 1 and all those of Y by 2, as the substitution specifies. The most general unifier for the terms is $\{A/Z, f(A)/X, g(Y)/U\}$:

$$\{p(A, X, f(g(Y))), p(Z, f(Z), f(U))\} \{A/Z, f(A)/X, g(Y)/U\}$$

This give the following expression when the substitution is carried out:

$$\{p(A, f(A), f(g(Y))), p(A, f(A), f(g(Y)))\}$$

The unification algorithm operates by passing over all the terms in a set, looking for positions in which symbols of each term are different. The algorithm constructs a set D of all the symbols that disagree at a particular position. If it finds such symbols it performs the following steps:

- 1) If D contains only constants then they must be equal; if not the Unification fails.
- 2) If D contains a variable v and a term t; add $\{t/v\}$ to the substitution being constructed providing t does not contain v. Otherwise the unification fails.

3) If D contains two variables construct the substitution $\{v_1/v_2\}$

4) Repeat steps two and three until D is exhausted.

All the expressions in the set must have the same number of terms, and every expression must always have a representative in D. The constraint made in alternative 2), that v must not occur in t, is made to avoid a cyclic substitution. Suppose that the term t is:

$$t = p(X, Y)$$

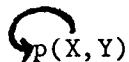
and that v is X. The substitution created by the unification algorithm is t/v , i.e. $p(X, Y)/X$. If this substitution is applied to the literal (X) then the substitution process will never terminate. After one substitution the result will be:

$$(p(X, Y))$$

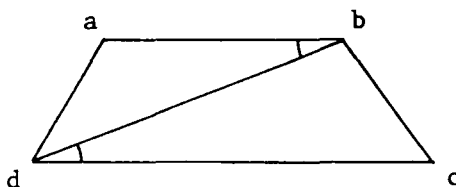
now X must be substituted again, giving:

$$(p(p(X, Y), Y))$$

and so on. As was stated earlier; a substitution replaces all the occurrences of the variable, so each new X leads to another substitution. To avoid non-termination one must use the occur check which fails the unification if v is an element of t. In practice the occur check is seldom incorporated into an implementation of logic because the such cycles arise infrequently, and are expensive to detect. Many interpreters will represent a cyclic substitution by a cyclic list because such structures can be useful. The example substitution will therefore have the form:


$$\text{p}(X, Y)$$

If two literals can be unified, and one is negated, then the two form a clash. The literals which clash are usually contained in clauses; it is these clauses which must be applied to the unifying substitution, not just the two literals. For example suppose that a program was written to prove that the angles indicated in the diagram below are equal[14].



The program will have the form:

$p(X,Y,U,V):-t(X,Y,U,V)$	if X,Y,U,V is a trapezium then X,Y is parallel to U,V
$e(X,Y,V,U,V,Y):-p(X,Y,U,V)$	the angles X,Y,V and U,V,Y are equal if the lines X,Y and U,V are parallel
$t(a,b,c,d)$	a,b,c,d forms a trapezium
$:e(a,b,d,c,d,b)$	are the angles a,b,d and c,d,b equal?

which when transformed into disjunctions produces:

- 1) $p(X,Y,U,V) \vee \sim t(X,Y,U,V)$
- 2) $e(X,Y,V,U,V,Y) \vee \sim p(X,Y,U,V)$
- 3) $t(a,b,c,d)$
- 4) $\sim e(a,b,d,c,d,b)$

which produces the resolvents:

- | | |
|----------------------|---|
| 5) $\sim p(a,b,c,d)$ | resolvent of 4) and 2). Notice that the resolvent has been applied to the unifier $\{a/X,b/Y,d/V,c/U\}$ |
| 6) $\sim t(a,b,c,d)$ | resolvent of 5) and 1)
unifier= $\{a/X,b/Y,c/U,V/a\}$ |
| 7) \square | resolvent of 6) and 3)
unifier= $\{\}$ |

The unification algorithm is therefore an essential aspect of any implementation of resolution. It is unification which allows clashes to be recognised so that the resolution algorithm may cancel them.

2.2.5. Application of Resolution

Resolution is a theorem proving technique which can be applied to logic programs by viewing the program as a theorem. In the descriptions given above the clause B corresponds to the question, and the clauses A_i to the program.

The Resolution algorithm uses clauses that have been transformed from implications to disjunctions in order to reveal the clashes, and then selects those to be resolved. This differs from the description, given in Chapter One, of the way a logic program is interpreted. Firstly no transformation of the program is carried out and secondly the choice of which clash to resolve is made according to some simple rules. These discrepancies may be reconciled as described below.

Since the transformations from clausal form to disjunctions is accomplished by the application of some simple rules there is no point in carrying out the transformation if the clashes can be identified without doing so. The transformation of each clause produces one positive literal for the head; while the remaining goals are negated because they come from the body. The question of the program is on the right-hand-side of an implication, which when transformed to a disjunction results in each goal becoming negated. The literals of the question therefore form clashes with the heads of all the clauses referred to by the question. Thus every goal in the question identifies a clash and there is hence no need to transform the clauses into disjunctions.

The resolvents of these clashes will be the body of the clause whose head is part of the clash. The goals in the body refer to the other clause heads and create clashes in the same way.

Consider the clauses

$$P_1 :- Q.$$

$$:- P_2$$

where the subscripts of P denoted different versions of P not, different clause names. When the two clauses are transformed into conjunctions, P_2 will become negated while P_1 will not. P_1 and P_2 will therefore form a clash. The clash can however be identified without performing the transformation because P_2 is on the left of an implication and P_1 is on the right.

Any practical interpreter will follow some simple rules that select which clashes to resolve. In most Prolog interpreters these rules involve resolving the goals in a clause body from left to right, and trying the clauses in the called relation in a top to bottom manner to do so. Other strategies are possible and include the parallel ones described in Chapter Seven. Any goal may form clashes with the heads of several clauses, each of which may lead to an independent refutation. The nondeterminism of logic languages comes about by following to completion the resolutions of all clashes created by a goal.

After a theorem has been proved using resolution the substitutions carried out by the unification algorithm will have assigned values to all the variables in the question. These values are the results the user requires.

2.2.6. Relation Names as Terms

As was stated above, a term may not have a relation name as its value. Hence Horn clause logic is unable to provide higher order clauses, the logic language counterpart of higher order functions. If such a feature is provided it will allow the programmer to ask what relation could produce a given result when supplied with specified data. This is a very difficult question for an interpreter to answer.

2.2.7. Negation as Failure

Negation is an important aspect of many practical logic languages because programs must often be able to test for the failures of goals as well as their success. This section explains precisely what negation as failure means from a programmers point of view, and how negation affects other parts of logic languages. Towards the end of the section a description of the less obvious, but very important, aspects of the implementation is given.

The restriction of Horn clauses only having one head makes the introduction of negation desirable. Consider the implication:

$$P_1 \vee P_2 :- Q.$$

which reflects the way P_1 and P_2 are related to Q . If one constructs the truth table for such an implication it would have the form:

P_1	P_2	Q	$P_1 \vee P_2$	$P_1 \vee P_2 :- Q$
F	F	F	F	T
T	F	F	T	T
F	T	F	T	T
T	T	F	T	T
F	F	T	F	F
T	F	T	T	T
F	T	T	T	T
T	T	T	T	T

If one wants P_1 to be true when Q is true then P_2 must be false, because if it is true the implication will hold regardless of the value of P_1 . That is:

$$P_1 :- Q, \sim P_2.$$

Consider the example:

$$\text{sad}(X) \vee \text{angry}(X) :- \text{rain}.$$

which is read, X is sad or X is angry if it is raining. If one called sad and the body proved to be true, one could not be certain if sad had been satisfied or if angry had been satisfied. Either implication could hold, so one can not determine which result to return to the caller of sad, true or false. This is the reason Horn clauses are restricted to one head. If the implication holds, then to be certain that X is sad, X must not be angry. This gives the stronger clause:

$$\text{sad}(X) :- \text{rain}, \sim \text{angry}(X).$$

Negation of a goal is therefore an important part of a practical logic language, and may be provided by interpreting negation to mean "failure to prove". To prove $\sim P$ attempt to prove P by all possible means, and if no proof can be found then $\sim P$ succeeds.

Negation as failure makes the so-called closed world assumption: which means that all information about a particular relation is held by the program. The definition of an implication

$P:-Q$.

states that if Q is true then so is P , but not the converse. Suppose that the above clause is the only one in the P relation, and that somewhere there is a call $\sim P$. This call will be true if the interpreter fails to prove P . In other words $\sim P$ will be true if Q is false. Thus the result of Q becomes the result of P . Hence "implication" has become equality. Consider the example:

$\text{rain}:-\text{hot},\text{humid}$.

This implication means that if it is hot and humid then it is raining. If the closed world assumption is applied and this is the only clause that defines rain, then the fact that it is not hot or not humid means that it is not raining. Rain is entirely defined by the values of hot and humid.

Implementing Negation

This section describes how negation must be implemented. Several problems can occur when interpreting a negated goal. For instance, if $g(X)$ succeeds then care must be taken when interpreting the $\sim g(X)$. The literal $\sim g(X)$, in which X is not bound, means that there exists a value of X which makes $\sim g(X)$ succeed. If g is successful, and binds a value to the previously unbound X , then following a simple minded interpretation of negation, $\sim g(X)$ should fail because g succeeded. To fail $\sim g(X)$, however, is to state that there is no X which makes $\sim g(X)$ true. Simply because one value has been found which makes $\sim g(X)$ fail, one is not justified to assume that there is no value of X which makes $\sim g(X)$ succeed, therefore one is not justified in failing $\sim g(X)$. Unfortunately $\sim g(X)$ cannot be allowed to succeed either because the value of X which makes

it succeed has not been found, therefore one may not assume it exists. The result of $\sim g(X)$ is consequently unknown, and the only course open to an interpreter in this situation is to stop the program and print an error message. There is no difficulty if X has a value before $\sim g(\lambda)$ is obeyed because one is simply finding out if g holds for X or not. It is the occasions where g binds a value to X the cause the problems.

If $g(X)$ fails then $\sim g(X)$ will succeed, but a successful literal may be expected to produce the value for X which allowed it to succeed. There will, however, be no value for X because g failed and "not" will be unable to produce one for itself. The program must therefore continue with X being undefined, a situation which is not entirely satisfactory.

A further problem which occurs when interpreting negation as failure can be illustrated by the following example.

$P: \sim \sim P.$

This expression may be rewritten as:

$$\begin{aligned} P \vee \sim(\sim P) &= P \vee P \\ &= P \end{aligned}$$

so the original implication for P should be proved true. If the interpreter tries to find this solution it will never succeed because in order to evaluate $\text{not}(P)$ it must first evaluate P , which leads to infinite recursion. In some circumstances it is possible to detect these loops, but such checks are seldom included in practice.

In spite of the deficiencies described above, negation as failure is an important and useful part of any practical logic language.

2.2.8. Characteristics of Logic Languages

Logic languages derive their power from two sources. The first is their ability to deal with relations rather than functions, and the second is the way they search for results.

Dealing with relations avoids redundancy because one relation may be used in several ways. In functional languages separate functions must be written for each different mode of use.

Searching for results relieves the programmer of the task of explicitly describing how results are produced. Logic languages also suffer from one drawback in this respect. Logic languages are not allowed to modify their data, and may not therefore use the power of searching on data acquired at run time. This is a considerable disadvantage. Some logic languages therefore allow assertions to be added at run time even though this will introduce the problems described earlier. An interesting area of current research is the development of meta logical operations which, amongst other things, will allow data to be included at run time without causing any difficulties.

CHAPTER THREE

CLASSIFICATION OF NOVEL COMPUTER ARCHITECTURES

This chapter describes the classification of Treleaven et al, in which the authors propose a collection of mechanisms which, they argue, form the basis for a general purpose computer architecture. If the authors' claims are justified the mechanisms described in this chapter can be used as a common base for the implementation of functional and logic languages. The declared aim of the work reported in this thesis is to find such a common base, and the classification is taken as the starting point of the work.

The classification proposes a set of data mechanisms and control mechanisms which can be used to construct models of computation by selecting a member of each set.

3.1. Models of Computation

A model of computation is an abstract description of the way instructions are selected for execution, and the way data is passed between instructions. Such a model may be divided into two parts, the data mechanism and the control mechanism. The various mechanisms proposed by the classification together provide the generality the authors claim for the classification.

3.1.1. Data Mechanisms

The data mechanism defines how data is shared or accessed by instructions. The two types are listed below:

Value: Each instruction that uses an argument is sent a separate copy of the value.

Reference: Each instruction that uses an argument holds its address, which is used to access the argument's value.

The value mechanism implies that instructions hold a copy of the data items they require. This gives great scope for parallelism because there will be no contention for data. In contrast the reference mechanism allows the use of a shared memory to hold values, which implies that contention will occur if several instructions attempt to access a value simultaneously.

3.1.2. Control Mechanisms

The control mechanism defines how processors execute a program: more precisely how the execution of one instruction causes the execution of another, and thus how the pattern of control is built up throughout the program. There are three types of control mechanism:

Control Driven: An instruction is executed when selected by explicit flows of control.

Data Driven: An instruction is executed when its data is available.

Demand Driven: An instruction is executed when its result is requested.

The control driven mechanism only allows an instruction to execute as a result of explicit control signals; indeed it may need many such signals before it will execute. Although a control signal is often sent to indicate the availability of data; this is by no means the only reason. The control driven mechanism is the most general of the three because one can use any combination of conditions to trigger the execution of instructions. Unfortunately it does force the responsibility for controlling the execution of the program onto the programmer. In a parallel machine the problem (of specifying flows of control) is made more acute by the need to avoid the non-determinacy which can result if the execution of instructions is not synchronised properly.

The data driven mechanism will execute an instruction when all its operands are available. Data driven execution is thus the same mechanism used by the innermost execution of functional languages, and consequently suffers from the same problems. The data driven mechanism relieves the programmer of the responsibility of managing the execution of the program.

Lastly the demand driven mechanism executes an instruction when its result is requested by an already active instruction. Demand driven execution has two phases: propagating demands for data, and passing results back. Demand driven execution is frequently used to implement need driven execution, where the demand for a result is only propagated when the result is necessary. If need-driven execution is not used, some other way of deciding when a demand is to be propagated must be

found.

There are three common models of computation which are be built from the above data and control mechanisms: control flow, data flow and reduction. Each will be examined in turn.

3.2. Control Flow

The control flow model uses the control driven control mechanism and the reference data mechanism. Each instruction expects a specific number of control signals to arrive before it will execute. Each instruction's data values are held in separate memory locations, the addresses for which are embedded in the instruction.

A control flow program may be viewed as a directed graph in which the nodes represent instructions and each arc defines the the path along which a "control token" may flow, carrying with it the signal to execute.

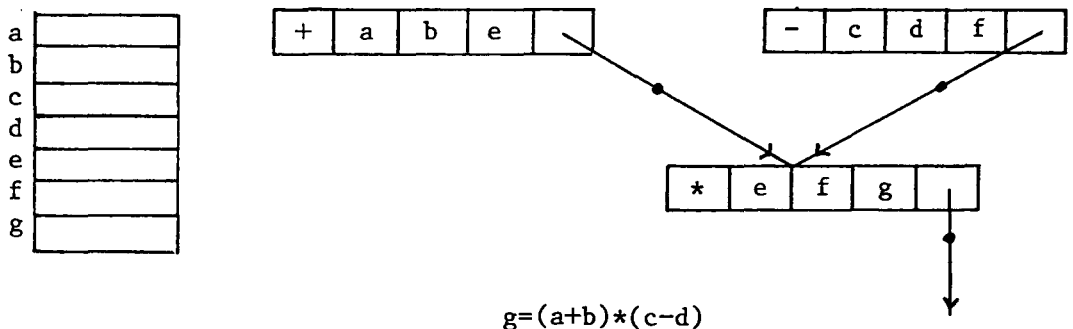


Figure 3.1: Simple control flow program.

Control flow is the most flexible of the three models because it is the most primitive, but the programmer must also manage every aspect of the program's behaviour.

3.3. Data Flow

A data flow model uses the data driven control mechanism and the value data mechanism. (In the model, the control mechanism and the data mechanism are supported by a single device called a "data token".) A data flow instruction will only execute when all its data is available. The data is embedded in the instruction by the time it is executed.

A data flow program may also be viewed as a directed graph, namely a collection of instructions joined by arcs along which the data tokens flow. A data token is used to pass data between instructions and consists of the address of the destination instruction, together with the value. Not only must the token be specify the correct instruction but also the correct argument position within instruction. A data token therefore signals the availability of the data, and passes the value to the destination.

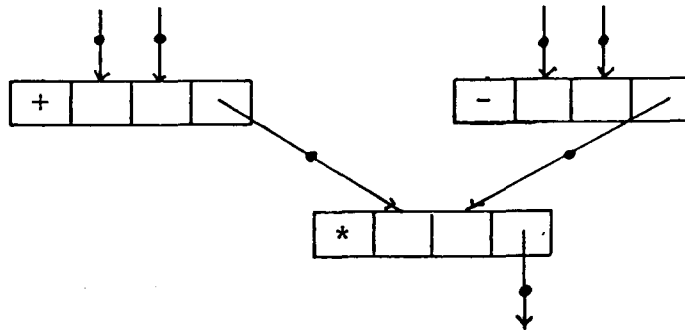


Figure 3.2: Simple data flow program.

In summary, the data flow model allows great parallelism, because the execution of the program is constrained only by the availability of data. The model also relieves the programmer of the task of managing the execution of the program.

3.4. Reduction

The reduction model has two basic forms: string reduction and graph reduction. The former uses a data driven control mechanism and a value data mechanism; while graph reduction uses a need driven control mechanism and a reference data mechanism.

As was explained in Chapter Two, reduction is the manipulation of expressions by simple rules until the expression is in its simplest form. Code and data are considered the same, and are held together in the same memory, this equivalence is one of the significant differences between reduction and control flow or data flow. A reduction machine does not allow the value of data to be changed.

The two variations of reduction, string reduction and graph reduction, are described below.

String Reduction

A string reduction program is represented as a nested set of expressions: it is evaluated by finding subexpressions containing only literal values and then reducing the subexpressions to their result. In the example below the subexpressions $(+ 1 2)$ and $(- 3 2)$ will be reduced first:

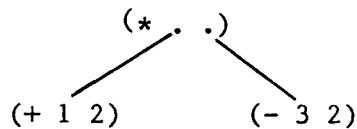
```
(* (+ 1 2) (- 3 2))
      to give
      (* 3 1)
```

The resulting multiplication has no subexpressions, and consequently may be reduced to give the final answer, namely 3.

Graph Reduction

In graph reduction a program is represented as a graph. Each subexpression of the program can be considered to be an instruction. Demand tokens, carrying the demands for data, will be propagated down the arcs which join consumers of data with its producers. The producer instructions will demand their own input data if necessary, and will then be reduced to their results. These results will pass back along the arcs to their consumers.

As an example consider the expression $(* (+ 1 2) (- 3 2))$:



The "*" instruction will be started by a demand to obtain the result. It will propagate a demand to each of its arguments causing them to be reduced:



Finally the multiplication will be reduced to give the result, 3.

The particular combinations of control and data mechanisms used by string and graph reduction are appropriate for the following reasons. In string reduction the only effective means of communication between sections of the string is by having the two communicating components adjacent to one another. Adjacency is used for communication because accessing separate sections of the string forces the processor to skip the intervening portion of the string to find the addressed section.

This will be a very inefficient operation. The value data mechanism is used because having functions adjacent to their arguments implies that each function application must have a copy of its components; the function body and the argument value. The value data mechanism will provide each subexpression with its own copy of all the data it uses. The data driven control mechanism is used because only the immediate context of an instruction is required to determine if it will execute or not. This control mechanism therefore requires information only about adjacent subexpressions.

Graph reduction uses the reference data mechanism and the demand driven control mechanism because the use of graphs to represent programs allows subexpressions to be addressed at will. To be efficient demand propagation requires direct access to the expression whose result is to be requested. The reference data mechanism provides the access required, and also permits common subexpressions to be shared. Graph reduction can also be data driven by starting at the leaves of the tree and working up the tree towards the root.

3.5. Using the Models of Computation

This section highlights some problems that a model of computation must overcome to be practical; it examines the implementation of procedures and iteration. Both of these topics are important aspects of programming languages; any deficiencies in these areas will have repercussions when the architecture is applied practically.

Procedure Calls

In control flow, data flow and reduction procedures are implemented by having a separate process for each invocation. Consequently an instruction will have a two part address, namely the process identifier, and the instruction's location within the process. (Addresses for the data memory used by control flow will also have the same format.) An address therefore has the form:

P/L
P = process identifier
L = location

The process identifier of the process for the called procedure is generated by a separate instruction and then passed to the instructions which will call the procedure and pass the parameters. The return address will contain the identifier of the calling process which allows the results to be returned.

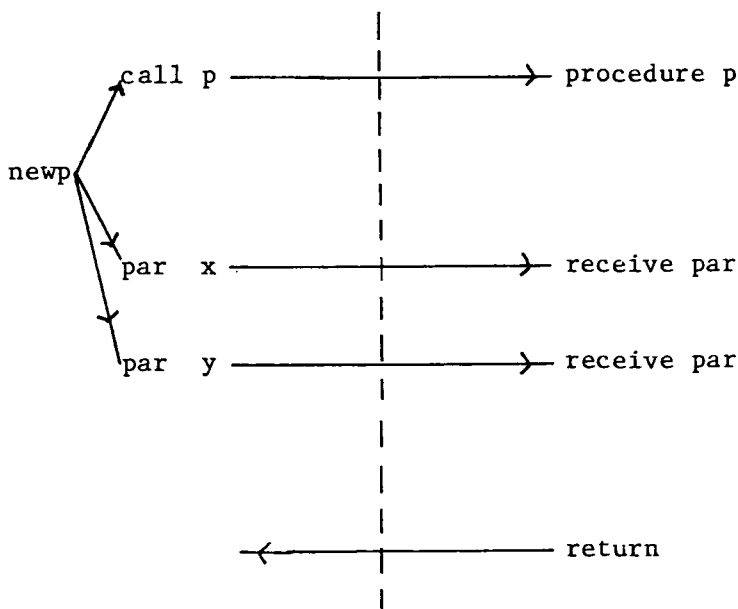


Figure 3.3: Procedure calling in control flow and data flow. The code to the left of Figure 3.3 is the calling code while that to the

right is the procedure. The new process identifier is generated by the newp instruction and passed to the call instruction and the "par" instructions. The call instruction sends the return address to the called procedure, and the "par" instructions each pass one parameter. Each parameter is received by an instruction in the called procedure. The parameters are then sent to all the procedure's instructions which require them. When the result produced it is passed to the return instruction which sends it to the instruction whose address was specified in the return address.

Iteration

Iteration involves the repeated use of a section of code and its associated memory locations. Iteration therefore incurs problems of ensuring the uniqueness of each instruction and memory location. If each iteration is allowed to execute in parallel there will be several copies of an instruction active at once. Thus memory locations; or instruction arguments, will need to hold multiple values, one for each execution of the loop body. There are two solutions to the problem [68]:

- 1) Do not allow parallel execution of the loop bodies. This is commonly used by control flow and may be achieved by using an extra "synchronisation" token. This token is released at the end of the loop; the first instruction in the loop is forced to wait for this token, which holds up the execution of the entire loop until it arrives. Once one execution has been started (by means of an initial token) the subsequent one will wait until the previous one has finished.

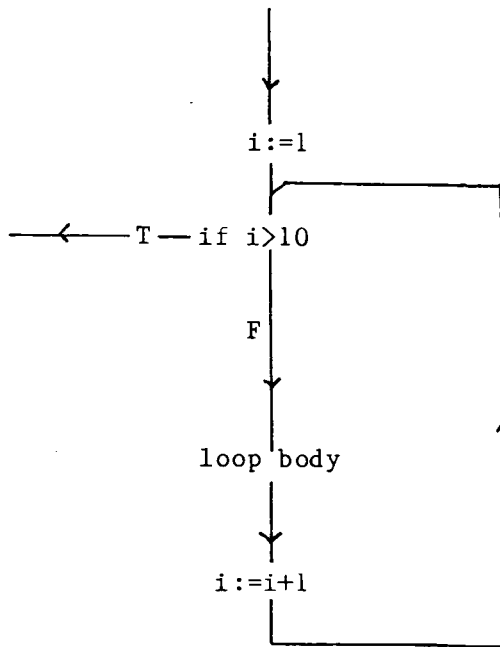


Figure 3.4: Iteration using a synchronisation token.

2) Implement iteration by means of tail recursion. This is commonly used by reduction as well as control flow and data flow. If each iteration is represented by the recursive call of a procedure containing the body of the loop, then each iteration will have its own process, and therefore its own unique locations, thus avoiding conflicts.

<pre>for i := 1 to 10 do print(i)</pre>	<pre>procedure printloop(i) begin print(i) if i < 10 then printloop(i+1) end printloop(1)</pre>
<p>iterative</p>	<p>tail recursion</p>

Figure 3.5: Comparison between iteration and tail recursion.

In Figure 3.5, the iteration section represents the way the code is usually written; the tail recursive section illustrates how the same effect can be achieved using recursion.

Data flow[1] and control flow machines sometimes provide an additional solution to the problem of iteration:

- 3) An additional level of process identifiers are provided which are used purely for providing a separate address for each iteration. This process identifier is an iteration number which is generated by incrementing the iteration number of the token which arrives at a special controlling instruction at the head of the loop. An address will now have the form:

P/I/L
P = process identifier
I = iteration number
L = location

Each iteration of a loop will have consecutive iteration numbers. Unfortunately this simple scheme will not work if loops are nested. Consider the nested loops:

```
for i := 1 to 10
do   for j := 1 to 20
      do   ...
```

When the first inner loop is started the iteration number passed to it will be 1, so the first iteration number of the inner loop will be 2, the same as the number for the second iteration of the outer loop. So the program will fail. To overcome this will require either a more sophisticated way of generating iteration numbers, or a separate number for each loop. Both would be clumsy, so in general solutions 1) or 2) are usually preferred.

The computational mechanisms described above will form the basis of the investigation to find a common way to support both functional and logic languages. The next chapter describes an architecture which implements the computational mechanisms, and which allows programs using a mixture of the models described above to be executed.

CHAPTER FOUR
GENERAL-PURPOSE MACHINE ARCHITECTURE

This chapter describes a general-purpose architecture based on the computational mechanisms described in Chapter Three. An emulator for the architecture was implemented to allow the evaluation of the computational mechanisms for the implementation of functional and logic languages. A more detailed description of various aspects of the architecture, together with some examples of program execution, are given in Appendix One.

The architecture described here has three major components: the processor, the active memory, and the passive memory. The processor obeys the instructions; the active memory AM, holds "active instructions", namely instructions that have received at least one token. AM also transmits packets, generated by the processor, to destinations in AM. The passive memory PM holds the data for the processes created during the execution of the program. In addition, PM holds the definition of the program being executed in an area referred to as the definition memory, DM. This architecture, shown below, forms the basis of the emulator used in this thesis.

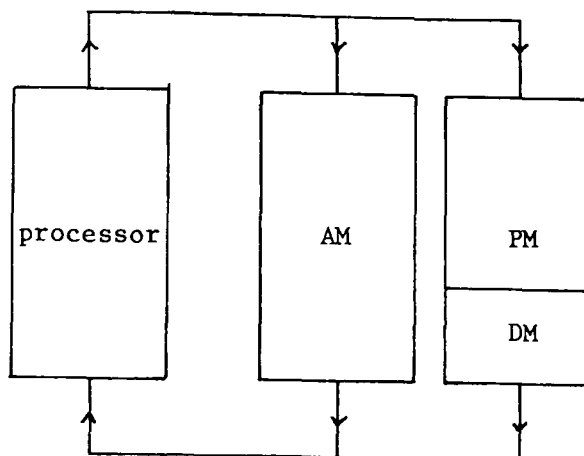


Figure 4.1: Packet Communication Architecture.

In terms of the example packet communication architecture described in Chapter One, AM and PM together form the memory. The "pool of work" between the processor and memory is implemented as a queue of executable instructions which are held in AM. There is no packet pool for holding packets generated by the processor; it is simpler to synchronise the activity of the processor and AM on a single processor machine. The communication resources have been omitted from the above architecture because the architecture has only one pipeline.

A basic objective of any architecture is to gather the operators and operands of an instruction together, so that the instruction can be obeyed. The architecture must also arrange for the instruction to be obeyed at the correct time. In a packet communication architecture both objectives are met, at least in part, by passing messages between instructions. These messages are held in packets which will typically contain the address of the packet's destination and some data. In data flow, for example, a packet is used to implement the data token. The address specifies the instruction and argument position of the destination, while the data in the packet will be one of the operands the des-

mination is waiting for. In control flow a control token will also be implemented as a packet, but it will only contain the destination address. In reduction a demand token will contain the address of the source of the demand, so that the destination instruction may return the result when it is produced.

The generation of tokens also controls the flow of execution in the program. Each instruction will have certain arguments which expect tokens, while others generate tokens. The processor will deal with each argument of an instruction at the time the argument value is required by the instruction's operation.

This thesis uses three versions of the architecture described in this chapter; but all three are founded on a common base which implements the computational mechanisms. The following sections described operation of the common base and the format of the data structures it uses.

4.1. Data Format

The data held in both AM and PM are complete instruction arguments, not just simple values. This also applies to the contents of data tokens. The architecture therefore allows one instruction to supply a complete argument of another instruction, instead of just a simple value, which provides great flexibility in the formation of instructions.

4.2. Instruction Format

The format of an instruction is similar to the control and data flow instructions described in Chapter Three. Each instruction consists of the opcode, a token count, and a number of arguments holding the data upon which the instruction will operate. These arguments also include the information necessary to dispatch the results to other parts of the program.

Each instruction therefore has the format:



Figure 4.2: Instruction format.

- 1) count. The number of data or control tokens that must be received before the instruction may be executed
- 2) opcode. The operation code for the instruction.
- 3) arguments. A set of arguments, each conforming to the rules given below.

Argument Format

An argument may hold either an input operand or an output destination. The sections of the instruction which hold the arguments are referred to as argument slots. Each argument consists of three fields:

- 1) The argument type.
- 2) An integer value, which is usually the operand of the instruction.
- 3) A machine address referencing a memory location.

Argument Types for Input

Each input argument specifies how an instruction is to obtain one operand, input arguments reside in the lower numbered argument slots of an instruction. The number of arguments used for input depends on the instruction. The input argument types are:

- 1) unk : unknown. The argument has an undefined value at present but will be replaced by the contents of a data token. This argument type is a data token acceptor. The token count of the instruction must be greater than or equal to the number of unk arguments.
- 2) litv : literal value. The argument has a literal value; an integer.
- 3) pm : PM address. The argument is held in a PM location, the address of which is held in the current argument.
- 4) am : AM address. The argument is held in an AM location, the address of which is held in the current argument.
- 5) prop : propagate demand. The argument is to be demanded from another instruction, the address of which is held in the current argument.

Argument Types for Output

An output operand is the destination for the result produced by an instruction, although in control flow an output argument may also be used to send a control token. Output arguments reside in the higher numbered argument slots of an instruction. The output argument types are:

- 1) spare : There is no argument in this slot. The demand propagation mechanism uses spare output arguments to hold the return address for the result.
- 2) unk : unknown. The destination will be supplied by a data token. This argument is therefore a data token acceptor even though it will eventually be used as an output argument.
- 3) sig : signal. This will send a control token to the instruction whose address is in the argument.
- 4) pm : passive memory address. The result is to be stored in the PM at the address given in the argument.
- 5) am : active memory address. The result is to be stored in a data token and sent to the instruction whose address is held in the argument. The address also indicates the argument slot to which the token is to be sent.
- 6) prop : propagate demand. The output argument is to be demanded from the instruction whose address is in the current argument.

4.3. Packet Format

Packets are used to implement tokens, of which there are three basic types, a control token which is used to trigger the execution of other instructions: the data token which allows data to be sent from one instruction to another, and a demand token which signals the request for data to the instruction which is to produce it. Each packet consists of: a type field, the address of the destination instruction including the number of the argument slot within the instruction to which the packet is to be sent, and the argument being transmitted. The packet argument is a complete instruction argument, as described above, and not just a simple value. A packet has the format:

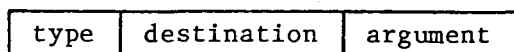


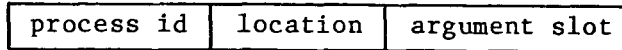
Figure 4.3: Packet format.

A packet's type can be:

- 1) cont : control token. The token only contains the address of the destination.
- 2) data : data token. The argument in the token is copied into the slot specified by the destination address of the token.
- 3) dem : demand token. The token holds the AM address of the sender in its argument field, the slot number of the address refers to the argument that propagated this demand.

4.4. Memory Organisation

Both the AM and PM are divided into processes and accept the same type of address. An address has the fields:



The process identifier, location number and the argument slot are all integers. The slot number has no significance to PM and so it is ignored. The process identifier "-1" has a special meaning: whenever an instruction refers to such a process the emulator will replace the -1 by the identifier of the process that the instruction belongs to, this is termed relocation. Relocation allows the code for a process to be written without knowing the identifier of the process the code will eventually occupy, which in turn allows the code to be executed in any process.

Each location within AM or PM can only hold an instruction; if an instruction argument is to be held in the location then by convention the value is held in argument one.

DM is a specific process in PM which is divided into procedures, each one of which occupies a specific range of locations. For example DM could have the format:

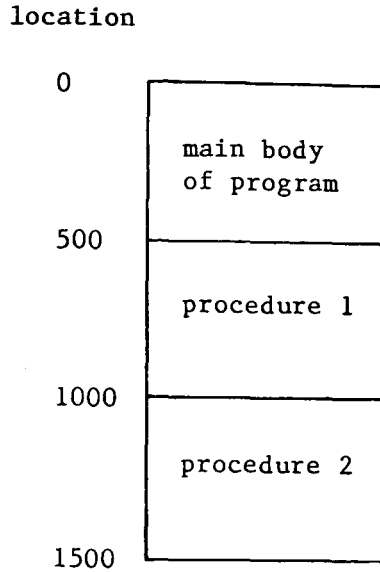


Figure 4.4: Format of DM.

4.5. Program Execution

This section describes how the structures used in the common base architecture are used in the execution of an instruction. The same sequence of operations is used in all versions of the emulator.

When the code to be executed is initially loaded into DM a check is made to see which instructions in the "main body" of the program are executable. Those instructions that are found to be suitable are placed on the queue of executable instructions, and are copied into AM. It is these instructions that are responsible for triggering the execution of the entire program.

When an instruction is loaded into DM the count field must be equal to the total number of control and data tokens the instruction expects to receive. Each argument which expects to receive a data token must be of type unk. An instruction becomes executable in two stages: first it is copied from DM to AM when it receives its first token, and secondly

it is placed in a queue of executable instructions when it becomes fully executable.

When a token is sent to an instruction the instruction will normally be resident in AM. If the instruction is not in AM, then ~~the processor~~ load it from DM. The destination address of the token must be mapped into a DM address so the prototype of the destination may be accessed. This mapping is carried out by copying the destination address and replacing the copy's process number with the process number of DM. The location number of the destination of the token and the prototype in DM will be the same, so the modified address now points to the correct prototype. A copy of the prototype is then placed in the AM location specified by the destination address of the token.

An instruction will only be considered for execution once the count of data or control tokens expected has become zero and, in general, if it has at least one output argument. An instruction's count is decremented each time it receives a data or control token. All executable instructions are placed on a queue in AM from which the processor selects the top one for execution.

Once an instruction has been selected for execution the processor will inspect the operation code and carry out the required task. This will involve accessing each input argument when its value is required; those which are not needed are ignored. Accessing arguments is carried out in the following manner:

- 1) litv : no action is necessary, the instruction may use the value directly.

- 2) pm : the new argument is loaded from the specified location in PM.
- 3) am : the new argument is loaded from the specified location in AM.
- 4) prop : this will result in a demand being propagated to the instruction whose address is held by the argument. Having propagated the demand the count of expected tokens will be incremented by one to signify that the data token carrying the result must be awaited. Demand propagation is explained in more detail below.
- 5) unk : there should be no arguments of this type because the count in the instruction is zero.

Once the processor has produced the instruction's result it will be dispatched to the destinations specified by the output arguments. This will be carried out in the following manner:

- 1) sig : a signal will be sent to the instruction whose address is in the argument.
- 2) pm : store the result in the specified location in PM.
- 3) am : send a data token to the specific argument in the specified instruction.
- 4) prop : the output argument is to be produced as the result of another instruction. A demand for the argument is propagated to that instruction in the same way as for an input argument.
- 5) spare : there is no consumer pointed to by this argument.

6) unk : there will be no arguments of this type because the count is zero.

Having been completed the instruction may either be deleted, or retained in AM. Appendix One describes the details of this operation.

The fact that the objects referred to by arguments are themselves arguments gives great flexibility. For example an argument may propagate a demand and receive as the result another argument which may propagate a further demand, and so on.

Demand Propagation

Demand propagation is the most sophisticated of the computation mechanisms; it uses arguments of type prop, the only input argument type which generates tokens. The demand token carries with it the address of its source; the source must await the arrival of the result before it may resume execution. The destination instruction for the demand will receive the demand token and place the source's address in its first "spare" output argument. The source will therefore be sent the result when it is produced. For example take the two instructions below:

1)

0	op	prop 2
---	----	--------	-------

2)

?	op	spare
---	----	-------	-------

Question marks are used in this section to denote undefined addresses or values. Instruction one has an input argument of type prop: the destination for the demand token is instruction two. When the processor attempts to access the prop argument it will propagate the demand token

to the destination. The demand token will have the format:

dem	2	1/2
-----	---	-----

The type of the token is "dem", the destination is instruction two, and the source is instruction one, argument slot two. Both addresses would normally include a process number, but these are omitted here for simplicity. When the demand token arrives at the destination it is placed in the first spare output argument, transforming instruction two in the way shown below:

2)

?	am 1/2
---	-------	--------

After transmitting the token instruction one will be transformed to:

1)

1	...	unk
---	-----	-----	-------

The prop argument has been changed to an argument of type unknown, and the count set to one. Instruction one must therefore wait for a data token.

When instruction two has produced its result it will dispatch it to the destination specified by its output arguments. One of these destinations is argument slot two of instruction one. Instruction one will therefore receive a copy of instruction two's result, and will become executable because receiving the data token will reduce the count to zero.

4.6. Implementing the Models of Computation

This section illustrates the generality of the architecture by describing how the three important models of computation may be implemented on it.

An instruction may use the control flow, data flow or reduction styles of computation, and even a combination of them. All that is necessary is to put arguments of the appropriate type in the instruction. If a mixture of control flow and data flow is used, the token count of an instruction must initially be set to the total number of control and data tokens the instruction expects.

The following sections describe the way each of the models of computation may be implemented.

4.6.1. Control Flow

A control flow instruction will have either literal values or PM addresses as its input arguments. The output arguments will be of two sorts: the PM addresses of the locations that are to hold the result, and arguments of type sig which contain the address of the instructions to which a control token must be sent.

Initially an instruction's count will be equal to the number of control tokens it expects; when the count becomes zero, due to receiving tokens, the instruction is executed. A control flow instruction could therefore have the format:

n	op	pm ?	pm ?	pm ?	sig ?	sig ?
---	----	------	------	------	-------	-------

The first two arguments get values from PM, and the third places the result in PM after it has been calculated. The sig output arguments send signals to those instructions which use the result. The output arguments are dealt with from left to right. A PM address which places data in a memory location must therefore appear to the left of the signals to instructions that will load data from that location.

4.6.2. Data Flow

In data flow input arguments will either be unknown or literal values. The output arguments will all be AM addresses. The initial token count will equal the number of unknown arguments. A data flow instruction could therefore have the format:

2	op	unk	unk	am ?	am ?	am ?
---	----	-----	-----	------	------	------

The first two arguments receive data in tokens while the remaining three dispatch the result. When all the data tokens have been received the result is calculated and sent in data tokens to the destinations specified in the output arguments.

4.6.3. Reduction

Reduction execution can be driven in two ways: by the availability of data or the need for data. Both forms of reduction replace an instruction by its result. This is achieved by retaining the reduced form of the instruction in AM, as explained further in Appendix One.

By Availability

For "by-availability" the executable instructions in the "main block" of the program will have literal values as their arguments, a count of zero and at least output argument. These instructions will therefore execute immediately, produce their results and then be retained in AM. A control token will be sent from each output argument to the consumers, which will load the result from the location in AM that previously held the producer instruction, and which now holds the reduced form of the instruction. AM is therefore used to hold data as well as instructions which is consistent with functional languages which do not distinguish between the two.

A reduction instruction which is driven by the availability of data could have the form:

l	op	litv ?	am ?	sig ?	sig ?
---	----	--------	------	-------	-------

The input arguments will be a mixture of literal values and AM addresses, all output arguments will send signals.

Alternatively, the result could have been returned in a data token instead of being held in AM. In this case the instruction will not need to be retained because each consumer has a copy of the result. This, in fact, corresponds to data flow.

By Need

For "by-need" the only instruction that will be executable when the program is loaded is the one which will propagate the initial demand for data. All other instructions will have either literal values or

arguments of type prop as the input arguments. All output arguments will be of type spare so they can be used to hold the addresses of the sources of the demands propagated to the instruction. An instruction could therefore have the format:

n	op	litv ?	prop ?	prop ?	spare	spare
---	----	--------	--------	--------	-------	-------

Each instruction will access the arguments it needs. In the case of prop arguments this will result in a demand being propagated to the instruction which will produce the data. When the producer of the result is executed it will return the value calculated in a data token to all the instructions which left their addresses in its output arguments. This will allow them to proceed with their own execution.

4.7. Operation Codes

This section describes the top layer of the architecture which provides a set of instructions that allow programs to be written for any of the models of computation. These instructions do not form the basis of the architectures for functional or logic languages; they are included solely to allow programs to be written which demonstrate that all the computational mechanisms are supported. Examples of these programs are given in Appendix One.

Arithmetic Instructions

add,sub,mul,div,rem,lt,le,eq,ge,gt,ne

Each arithmetic instruction takes the first two arguments as its input operands and distributes the appropriate result to the output destinations held in the remaining arguments. A boolean result is returned

from the comparison operators, "lt" to "ne", and is represented by an integer value, the number one for true and zero for false.

Distribution Instructions

dist,distl

Both instructions take their first argument and distribute it to the output destinations specified by the remaining arguments. The instruction "dist" will dereference any address and propagate any demand specified by the first argument. Addresses will only be dereferenced once by the dist instruction but demands will be propagated repeatedly as described above. The "distl" instruction will distribute the first argument exactly as it is, the distl instruction will "distribute literally". If either instruction is to distribute an address it is first relocated. This allows an address to be sent which points into the current process' address space, in either AM or PM.

Other Instructions

read

Reads an integer from the user and dispatches it to the destinations in its arguments.

print

Prints the integer value of its first argument and distributes this value to the destinations specified by its remaining arguments.

cond

The conditional instruction. This instruction first gets the value of the predicate, which is the instruction's first argument. The predicate will be either a literal value, an AM or PM address, or an argument of type "prop". Having got the predicate value one of the two arms of the conditional are selected. Argument two is selected if the predicate is true (returned one) and argument three is chosen if it is false (returned zero). The arms will usually have one of two argument types:

sig: If selected a control token is sent to the address specified by the argument.

prop: A demand is propagated to obtain the result from the selected section of code. When the result returned it is dispatched to the output destinations specified by arguments four and above of the conditional instruction.

If the arm is to have no effect when it is selected then it should be of type "spare", any other type will result in the argument being distributed via the output arguments.

call

The call instruction is responsible for calling a procedure or function and has only one input argument, this is the procedure identifier. The procedure identifier is an index into DM which identifies the procedure to be called. The call instruction will generate a new process for the procedure to execute in and pass the return address to it.

param

The param instruction is placed immediately before a call in a data flow program. Each parameter will be sent to a particular argument slot of the param instruction in data tokens. When all the parameters have arrived the instruction will perform two tasks. It will signal the call instruction that the parameters are ready, and send each parameter to the instruction which will pass the parameter to the procedure.

ret

The return instruction, which will have at most two arguments. The first argument will be the return address, and the second the value to be returned. The return instruction will send a token to the instruction pointed to by the return address, and the token will contain the result if there is one.

4.8. Implementing Conditionals

A conditional instruction may either be data-driven, or need-driven. To implement the former the instruction must be made to wait until the tokens which indicate the availability of its arguments have arrived. The data may be sent in data tokens, or be held in memory and its availability signaled by a control token. Need driven execution can be implemented by making each argument propagate a demand for its value. A conditional instruction will always look at its arguments in the order it needs them, thus only those results that are required will be demanded.

4.9. Implementing Functions and Procedures

Recall that in data and control flow procedures are constructed using processes. The call instruction must create the new process, start its execution and pass the return address to it.

In contrast, a function call for a reduction machine must be implemented differently because the function body should, conceptually at least, overwrite the call. In practice a different approach is adopted. Once the called function has been invoked the call instruction will be modified to become an instruction that will hold the result. The return instruction is sent the return address (the address of the call instruction) when the function is invoked, and will dispatch the result to this address when it is received from the body of the function.

Procedure Format

The procedure format has two sections, the parameter passing and return section, followed by the procedure body. At the top of the procedure there will be a "dist1" instruction whose first argument is a literal value equal to the number of parameters the procedure expects. This instruction is never executed, it is there simply to provide a record for the call instruction to consult. The next instruction will be a return instruction. The first argument of the return instruction is the return address, which is sent by the call instruction. Following the return instruction there are n instructions that are responsible for distributing the n parameters within the procedure body. The body of the procedure can be any combination of instructions, but it must arrange for the return instruction to be executed when the body of the procedure has been completed.

The format for a procedure will therefore be:

```
dist1,n
ret
dist1,unk
.
.
.
dist1,unk
procedure
body
```

Figure 4.5: Procedure Format

Procedure Call Format

A procedure call is constructed from the call itself, followed by n "dist" or "dist1" instructions. The first argument of these distribution instructions will hold the value which is to be the procedure parameter, the second argument will be the address within the called procedure of the corresponding parameter handling instruction. This address is passed to the instruction by the call instruction when it has generated the new process. The second argument of the dist instructions will initially be of type unk so the distribution instruction can receive the address in a data token. A call will therefore have the format:

```
call,p
dist[1],"parameter value",unk
.
.
.
dist[1],"parameter value",unk
```

Figure 4.6: Procedure call.

The procedure parameters will generally be of two sorts, either an input value or an address to which a result must be sent.

4.10. Assessment of the Architecture

Several problems were encountered during the implementation of the architecture. The first of these concerns instructions in the body of a called procedure which are immediately executable. Such instructions must be executed immediately the procedure is called. To implement this could involve searching the entire body of the procedure to find such instructions, alternatively the instructions could be chained together in some way, possibly with the head held by an argument of the distal instruction at the top of the procedure. Both these alternatives add somewhat to the complexity of the architecture, but neither will help demonstrate the computational mechanisms the architecture implements. For this reason the problem was ignored. Any immediately executable instructions must be, in effect, compiled out and the results they would have produced placed in the correct arguments of the instructions which require the results.

The second difficulty involved functions used in reduction, and was resolved as described earlier. A function call in a reduction machine will usually be overwritten by the body, but this is difficult in a packet communication architecture because it implies that both the function body and the calling code will have the same process identifier. Thus the distinction between several invocations of the same procedure will be lost because the locations used by the instructions will clash. An alternative scheme will be to change the location of each instruction as it is copied into the calling code so that each address is again unique. This is impractical, however, because the input and output

arguments of the instructions also contain addresses, which must in turn be modified. As a result the compromise solution described earlier was adopted.

The last problem concerns garbage collection. Normally a return instruction will delete a process and all its data, but this will remove the possibility of returning results which are held in the generating processes data area. To allow this, and to do garbage collection as well, will either involve some way of allowing the program to explicitly delete a structure when it is no longer useful, which is difficult to determine, or alternatively a mark scan garbage collector could be used. The former is the most efficient because exactly what storage is to be freed is always known, a garbage collector is easier to implement. Since the garbage collector does not help demonstrate the implementation of the computational mechanisms it is not included.

The flexibility provided by allowing memory locations and data tokens to hold instruction arguments proved most useful, particularly when implementing procedure calls.

Of the other packet communication architectures in the literature two are related to this project. The first is ALICE (Applicative Language Idealised Computing Engine) which has been developed by Darlington and Reeve at Imperial College [27], and the second is ZAPP (Zero Assignment Parallel Processor) which has been produced by Sleep at the University of East Anglia [67].

ALICE is aimed at the implementation of functional languages and is based around reduction. The architecture is a packet communication architecture which implements control flow. In ALICE an instruction may have two states: asleep or awake; an instruction is only executed when

it is awake and has all the data it requires. Execution by need is performed by giving all instructions (bar one) a sleep status when the program is loaded into the machine. When an instruction requires the result of another it places its own address in an output argument of the producer of the data, and wakes producer. The instruction which requires the data goes to sleep to await the result. When the the producer has been reduced to its result it wakes all the instructions which asked for the result, which in turn load it from the location which held the producing instruction. ALICE is therefore an architecture whose mechanisms are used to support reduction. Interestingly the architecture implements reduction in terms of control flow; the active instruction becoming the passive result to which the consumer refers. The architecture used in this thesis, however, implements reduction in terms of data flow. Unfortunately ALICE may suffer from contention for access to the result, but the architecture used in this thesis does not. To avoid contention the architecture described here makes use of the token, which must be sent to signal the availability of the result, to carry the result to the instructions which demanded it. In this way each instruction receives a copy of the result and so there is no contention. ALICE implements its pool of executable instructions as a pool of packets containing the instructions which are distributed amongst the processors. The architecture described here implements the pool as a queue for the architecture's single processor. Both methods would seem to be appropriate for the architecture that uses them.

The ZAPP architecture supports functional languages using data flow. This architecture is a particular way of evaluating combinator expressions, which can be used to implement functional programs. A topic which is described in Chapter Five. The paper describing ZAPP

gives scant details of how the design will be realised, so it is difficult to make detailed comparisons with the architecture described here. The evaluation scheme used is essentially demand driven, but demands are propagated before it is known if the result is needed. This is termed rash evaluation and allows greater parallelism than the pure by-need mechanism. The termination properties of the latter are preserved by only allowing each rash evaluation a limited amount of resources. When these are exhausted the evaluation is suspended until the resources are renewed. In this way rash evaluation may still be controlled, and stopped when it is discovered that the result which it will produce is not needed.

4.11. Rules for Architecture Modification

The architecture described above will be used in the remainder of this thesis as the basis of an investigation into the support of a functional and a logic language. By using the architecture it will be possible to evaluate the computational mechanisms described in Chapter Three for supporting both types of language.

The architecture may be thought of as being divided into two layers. The bottom layer implements the computational mechanisms, and the top layer implements the operation codes of the instructions. When implementing either of the languages it will be necessary to modify the top layer to incorporate the operation codes required by the language. No modification of the bottom layer should, however, be made. If such a modification to the computational mechanisms proves necessary it indicates a flaw in the classification in Chapter Three, and demonstrates that the computational mechanisms described are not able to support the language in question.

CHAPTER FIVE

IMPLEMENTATION TECHNIQUES FOR FUNCTIONAL LANGUAGES

Perhaps the most common way to implement a functional language is to use Landin's SECD machine[49]. The architecture described in Chapter Four is, however, tailored to reduction as a mechanism for supporting functional languages. Unfortunately the SECD machine is not a reduction machine because it separates program and data, so it will not be considered further. The form of reduction that will be used in this thesis is graph reduction, in particular the scheme proposed by Turner[69]. The description of graph reduction given in this chapter is divided into two sections: the first section is devoted to a description of combinators, the operators used in Turner's graph reduction scheme. The second section describes Turner's graph reduction scheme itself. The description of combinators is confined to the three simplest examples because these are sufficient to illustrate the principles involved. A description of the remaining combinators may be found in Appendix Two.

Using combinators, and particularly graph reduction, provides an elegant way of implementing a functional language. Several of the features required by such languages are provided implicitly.

5.1. Combinators

This section describes combinators, which are the instructions used in graph reduction to bind a function's arguments into the function's body. A combinator is an operator which has as its arguments several

expressions, and a value, and which applies these expressions to the value.

The central notion of both Lambda Notation and combinators is to substitute an argument value into a function body. The Lambda notation searches the function body for each occurrence of the bound variable, and replaces the each occurrence by the argument value. Combinators operate by distributing the function argument throughout the function's body so that a copy of the argument arrives at each element in the body. As the argument arrives it is either rejected, and the original element kept, or it is accepted and the original element overwritten. Two combinators are used as the acceptor and the rejector of arguments. These combinators are:

- 1) **I**, the identity function: takes the identity of its argument. This is the acceptor of a function argument and is defined by the rule:

$$Ix = x$$

The function argument x is passed to the **I**, whose operation leaves x as the result. So if an **I** appears in an expression it will eventually be replaced by the function argument.

- 2) **K**, Keep: keeps its first argument and rejects the second. This is the rejector of function arguments and is defined by the rule:

$$Kyx = y$$

The identifier y is the symbol in the expression which is to be kept, and x is the function argument which is to be rejected. The combinator is applied to both its operands, it retains y and discards x .

The way these combinators may be used to accomplish argument binding may be demonstrated using the function:

$$\text{fun } x = g \text{ h } x$$

When represented using **K** and **I** the function body will have the form:

$$(\mathbf{K}g)(\mathbf{K}h)\mathbf{I}$$

The symbols which are not the bound variable are protected by **Ks**, and each occurrence of the bound variable is replaced by an **I**.

If the function **fun** is applied to an argument, say the value **l**, the argument must be distributed throughout the function's body. When the argument value arrives at each segment of the body, the combinators will either accept or reject it, for example:

$$\begin{aligned} \text{fun } l &\Rightarrow (\mathbf{K}gl)(\mathbf{K}hl)(\mathbf{I}l) \\ &\Rightarrow g \text{ h } l \end{aligned}$$

The **Ks** reject the **l** and keep the symbol, **g** or **h**, and the **I** accepts the argument and is replaced by **l**.

The **K** and **I** combinators perform the accepting and rejecting of the function arguments, but a third combinator is required to carry out the distribution. This is the **S** combinator, which is defined thus:

- 3) **S**, Substitute: substitutes its third argument into its first two arguments. **S** is defined by the rule:

$$\begin{aligned} \mathbf{S}fgx &= fx(gx) \\ \text{or more clearly} &= (fx)(gx) \end{aligned}$$

The **S** combinator applies **f** and **g** to the function argument **x**. The symbols **f** and **g** denote the expressions into which **x** is substituted, they may also contain **S** combinators which will cause further distributions.

To use the **S** combinator the function's body must be divided as follows. First add all the default brackets, which are left associate as explained in Chapter Two. This transforms the function body into:

((**Kg**)(**Kh**)) **I**

The left-hand part of each function application is known as the operator (the function) and the right the operand (the argument). This scheme applies recursively to subexpressions, so **Kg** is the operator of the operator and **Kh** the operand of the operator. The division stops when the subexpressions which contain the **Ks** and **Is** are reached. For each operator/operand pair introduce an **S** combinator to distribute the function argument to the operator and operand. The introduction of **S** combinators starts at the highest level operator/operand pair and adds an **S** to distribute the bound variable. This gives the expression:

S ((**Kg**)(**Kh**)) **I**

The introduction of the **Ss** works progressively down the levels of nesting, finally producing the expression:

S (**S** (**Kf**)(**Kg**)) **I**

When the combinator expression for fun is applied to **l**, the following reductions take place:

fun l =>	S (S (Kg)(Kh)) I l	
	=> S (Kg)(Kh) l (Il)	1st S reduced
	=> (Kg1)(Kh1) l	2nd S and I reduced
	=> g h l	both Ks reduced

Therefore applying a combinator expression to an argument value reproduces the function body with the argument value substituted in place of the bound variable. The compilation process transforms an expression into combinators, which define how the argument is to be substituted into the expression. Substitution reverses the compilation process and

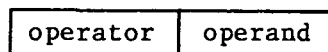
re-creates the original expression, but now with the argument value in the correct positions.

5.2. Graph Reduction

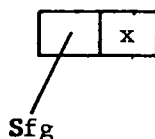
The combinators described in the previous section provide an elegant way to represent functions, and can form the basis for an implementation of a functional language. Such an implementation can either use the combinator expressions themselves or the graphical representation of them. Turner[69] suggests using the latter.

5.2.1. Graph Structure

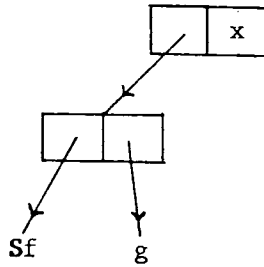
Turner's graphs are built using the operator/operand structure of the combinator expression, and take a form which approximates to a binary tree. Each node in a graph represents a function application and contains two fields: the left one is the operator and the right the operand:



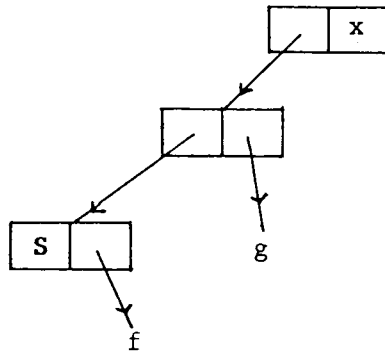
Each field in a cell may contain either a pointer to another cell, or a literal value such as a piece of data or a combinator such as **S**. The graph may be constructed from an expression by successively dividing it into operator/operand pairs and introducing a cell for each. For example take the expression **S f g x**. It will initially be divided to produce the expression **(S f g)x**, which will be represented as the node:



where x is assumed to be a literal. If the operator is divided again the result will be $(S f)g$:



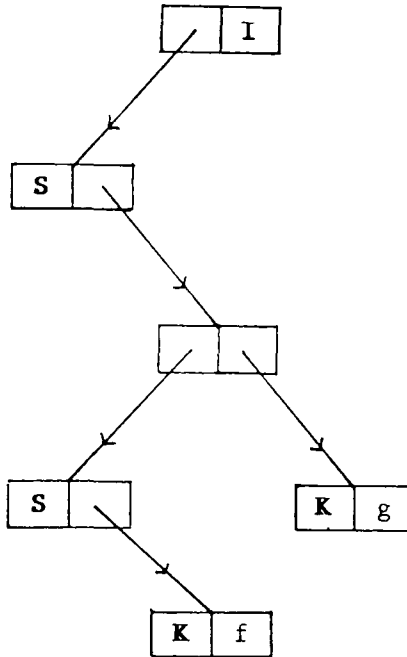
Sf is now divided to produce the final graph:



In the discussion so far only the outer-most combinator of the expression has been converted to a graph; the remainder of the expression is held in the outermost combinator's arguments. Each of these is now converted in turn using the same algorithm. For example the expression:

$S(S(Kf)(Kg))I$

will produce the graph:



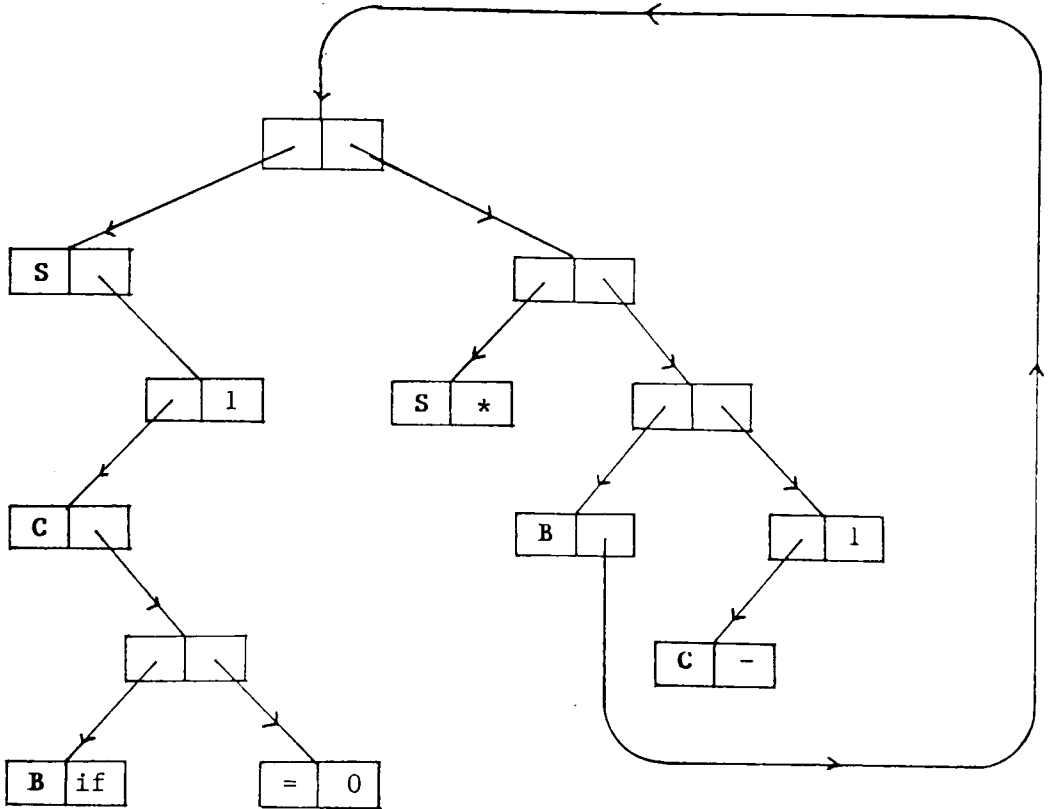
A more realistic example could be based on factorial:

```
factorial = λn.(if x = 0 then 1  
                else n * fac(n - 1) )
```

The combinator representation of which is [69]:

```
factorial = S (C ((B if) (= 0)) 1) (S * ((B factorial) (C - 1)),)
```

This expression uses a simplified representation of recursion, the usual representation is described in Appendix Two. The graph constructed from the expression is:



5.2.2. Graph Manipulation

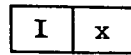
The graphs described above are a particular way of representing an expression; reduction operates by reducing one expression to another until the final result is obtained. It follows therefore that graph reduction must manipulate the graph which represents the program until the graph represents the program's result.

The reduction of a function is carried out in two stages: the first uses the combinators to substitute the argument value into the function body, and the second reduces the function body to its result. The reduction is controlled by the combinators, and other operators such as plus, which the graph contains. Each operator defines a reduction rule which specifies how the graph is to be manipulated.

The following three sections give the graphical representation of the reduction rules for **S**, **K** and **I**.

I Combinator

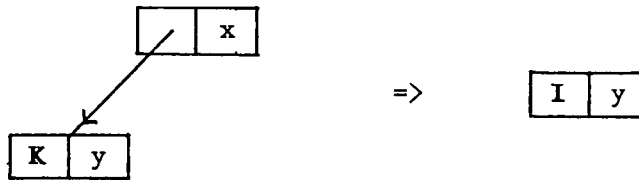
The graphical representation of **I** is:



The reduction of **I** has no effect, the node is retained in the same form so **x** may be accessed by other combinators.

K Combinator

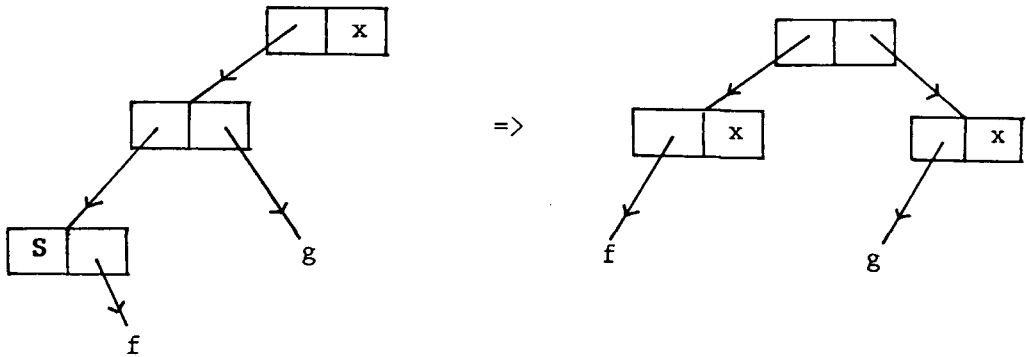
The reduction of **K** may be represented by the graphs:



The graph on the left is modified so that the top node becomes the one shown on the right. The root of the result must represent the reduced form of **K**, which is the value **y**. The **I** combinator is introduced because each node must have an operator, the **I** is chosen because it will not change the meaning of **y**.

S Combinator

The reduction of **S** may be represented by the graph:

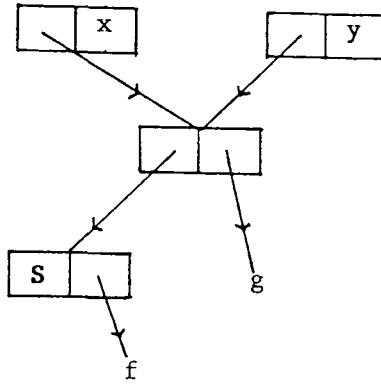


The top cell of the graph on the right is modified to reflect the result of **S**, which is $(fx)(gx)$. The two lower nodes of the resulting graph are new - they are not nodes from the old graph modified to hold the new function applications - only the top node is retained and modified.

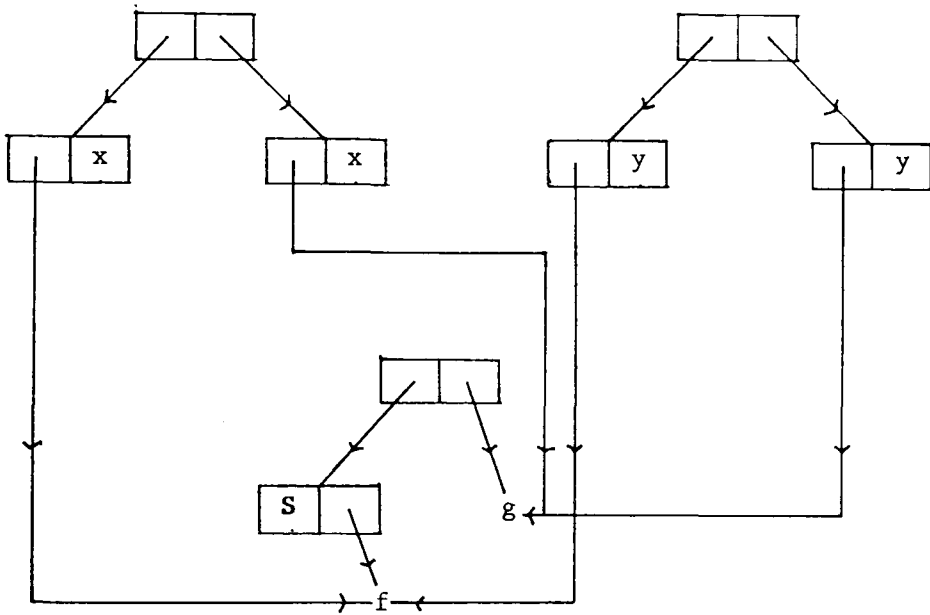
Protecting Function Definitions

In the above descriptions of graph reduction only the top node of a graph has been modified. Any other nodes in the resulting graph are new, they are not nodes from the old graph given new uses. The old nodes may not be re-used because they may be shared by other parts of the graph.

The generation of new nodes is an essential feature of the operation of **S** and the related combinators (**B**, **C**, **S'**, **B'**, **C'** explained in Appendix Two). The use of new nodes is necessary because whenever a bound variable is substituted into a function body a copy of the body must be taken to avoid corrupting the function definition. For example take the following code which represents a function definition used in two calls:



The lower two cells represent the function definition, and the top two cells represent two function applications. This graph will be reduced to:



Both reductions above have been carried out using the root of each graph to provide the third operand of **S**, so both roots have been modified to reflect the reduction of **S**. Neither reduction has affected the function definition which is left undisturbed in the resulting graph.

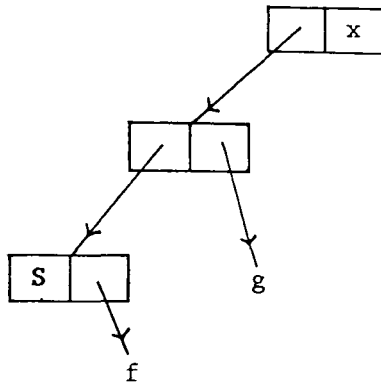
Copying function bodies when binding arguments could introduce difficulties when implementing lazy evaluation. Any reductions performed on the copied body will not benefit any future callers of the function because the caller refers to the function definition, and not the copy upon which the reductions are carried out. In practice combinators avoid this problem because they only copy those parts of the function body which contain the bound variable. The sections of the function which are constant with respect to the bound variable are the only subexpressions whose reduction should benefit future callers. These constant subexpressions are retained in the definition and referred to by pointers from the copied body. The reduction of such subexpressions is therefore carried out in the definition of the function, and their reduction will therefore benefit future callers.

So far the description of graph reduction has concentrated on the operations carried out by each combinator. No attempt has been made to give an account of how these operations are actually implemented. This omission is corrected in the following section.

5.2.3. Performing Reductions

This section covers two topics: firstly the order in which the reductions are carried out, and the properties this confers on the program. Secondly how graph reduction can be implemented.

The reduction of an expression is driven by need; so the outer-most combinator must be reduced first. The outer-most combinator will be the one contained in the leaf cell at the extreme left of the tree. In the example below the outer-most combinator is the **S**.



Once the outer-most combinator has been found the processor reaches back up the tree to find its operands, and then performs the required reduction. This reduction will manipulate the graph; which will usually result in another operator becoming the left most in the graph. This is the operator which must be reduced next. Often the operator to be reduced will not be a combinator, but another instruction such as if or plus. The construction of a combinator expression is such that when an argument is substituted far enough to allow the result to be partially evaluated, this evaluation is carried out. For example consider the reduction of the expression:

$$(f x) (g x)$$

when applied to the value 1. The combinator expression which represents $(f x)(g x)$ is:

$$S (S (Kf) I) (S (Kg) I) 1$$

The reduction using the outer-most rule will be:

$$\begin{aligned} S (S (Kf) I) (S (Kg) I) 1 &\Rightarrow S (K f) I 1 (S (K g) I 1) \\ &\Rightarrow K f 1 (I 1) (S (K g) I 1) \\ &\Rightarrow f (I 1) (S (K g) I 1) \end{aligned}$$

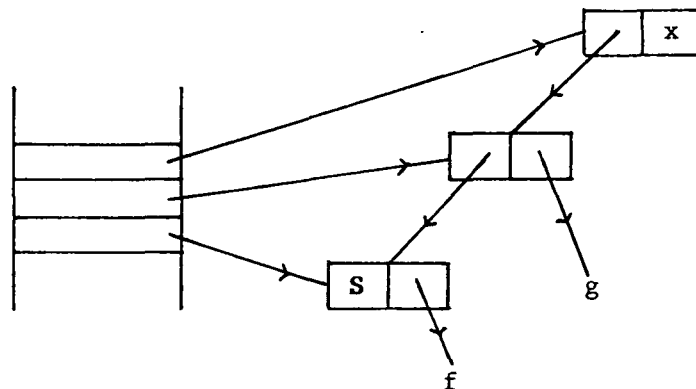
Now f is applied to $I 1$ because it has become the left most operator. Notice that the argument of f , namely $I 1$, is not even reduced before it

is substituted into f , because f may not need the value of its argument to produce its result. The second half of the expression has not been reduced at all. Substitution is only carried out as far as necessary to produce the result; if the result could be produced without the value of gx , the argument value is never substituted into that part of the expression. In short always reducing the outer-most left-most operator is in fact reduction by need.

There are two accepted ways of performing reductions on a single processor machine. One using a stack[69] and the other reversed pointers[17].

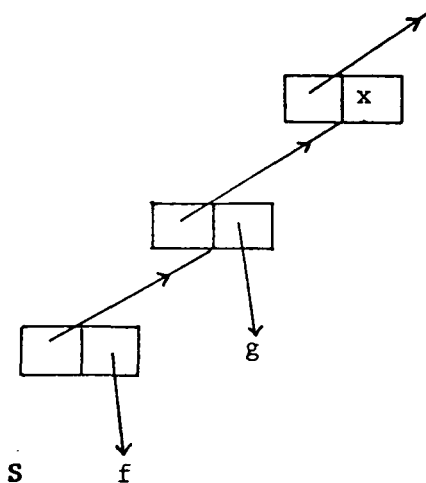
Using a Stack

A stack can be used to record the path followed down the tree to find a reducible combinator. As already explained the processor always follows the leftmost branch of the tree to find a reducible expression. When the combinator is reduced the processor uses the earlier entries on the stack to find the combinator's arguments. An example of using a stack to represent a reducible expression is:



Using Pointer Reversal

Pointer reversal arranges that as the processor progresses down the operator chain it reverses the pointers to record the path. When the leaf cell is found the combinator is removed from the node and held separately to make room for the last reversed pointer. The chain of reversed pointers is then used to access the combinator's operands:



When the **S** combinator is reduced it reaches back up the tree using the reversed pointers. Following the operator chain is, in fact, demand propagation. Recording the path followed corresponds to retaining the address of the instruction to which the result is to be returned.

As an alternative to reaching back up the tree the processor can follow a scheme which adheres more strictly to the rules of reduction. The rules of reduction require that each function application is replaced by its result, which applies to user defined functions and combinators alike. Taking **S** as an example, the application of **S** to **f** should return a result S_f :

$$Sfgx \Rightarrow S_f g x$$

The new "combinator" S_f is then applied to g :

$$S_f g x \Rightarrow S_{fg} x$$

Only now can the final result be produced:

$$S_{fg} x \Rightarrow fx (gx)$$

Each stage in the combinator's reduction produces a new combinator which is applied to the remaining arguments. This scheme will either require nodes to expand in order to hold each new combinator (because the node must hold the values of the arguments incorporated into the new "combinators"), or alternatively pointers to be used to point to nodes which represent the new combinators. The latter of these two schemes will be the easiest to implement but it could be argued that the normal pointer scheme does this anyway. The operator pointer of a cell points to the function application that produces a new combinator, so it points to a cell which represents the new combinator. Reaching back up the tree is therefore an acceptable optimisation of true reduction.

5.2.4. Assessment of Graph Reduction

The basic task of combinators is to bind function arguments into function bodies. Without them this is carried out by a rather complex side effect of the call instruction, as in the SECD[49] machine for example. However if combinators are used this is no longer the case. The binding of arguments becomes the responsibility of a set of simple instructions that can be incorporated easily into a machine. This is one of the most elegant aspects of combinators: they allow argument substitution to be defined in terms of simple reduction rules. This elegance complements the elegance of the call/return mechanism of reduction. As was explained in Chapter Three, the called body overwrites the

call instruction, the result then overwrites the body, and therefore the call, thereby accomplishing the return. Incorporating binding as a side effect of function calls spoils the elegance of this call and return mechanism.

The main disadvantage of combinators is that a new copy of the function body is taken each time a bound variable is bound into the body. This is, arguably, an inefficient operation in both space and time. However, graph reduction provides many of the features required by a functional language in an elegant way, since they arise naturally from the way that graph reduction is implemented. The simplicity that results does much to outweigh the inefficiency.

The elegance mentioned above can be illustrated by the two most important features of functional languages: higher order functions, and evaluation by-need, particularly when incorporating laziness. Both considerably increase the power of the language, and both rely on the notion of a closure for their implementation. As was described in Chapter Two, a closure represents the association of a function body and its environment. The purpose of a closure purpose is to allow the execution of a function to be suspended, and then restarted.

Combinators implement closures as a natural consequence of their operation. Consider the example:

$$f = \lambda x.g \ x$$

where $g = \lambda y.\sin(x)+y$

$f \ 1$

The function f is applied to the argument 1 which is substituted throughout f and the body of g , at least in principle. In actual fact the substitution will not have been carried out but instead will have

been suspended until g is called. The evaluation of a combinator expression is carried out by-need. The substitution of the argument will be suspended until the evaluation of f must resume in order to produce g . The suspended substitution of the value of x represents a closure and is implemented as a partially evaluated combinator expression.

For the above reasons combinators have generated considerable interest, and have given rise to several machine designs which use them for argument binding, amongst these the ZAPP architecture, described in Chapter Four, and SKIM[17], developed at Cambridge, are the best known.

CHAPTER SIX

GRAPH REDUCTION ON THE MACHINE ARCHITECTURE

This chapter describes the implementation of graph reduction on the emulated architecture described in Chapter Four. The aim of the implementation is to demonstrate that the architecture is able to support graph reduction.

6.1. Instruction Format

As described in Chapter Four the instruction format used by the architecture is:

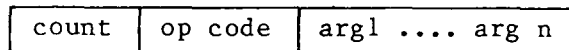


Figure 6.1: Instruction format.

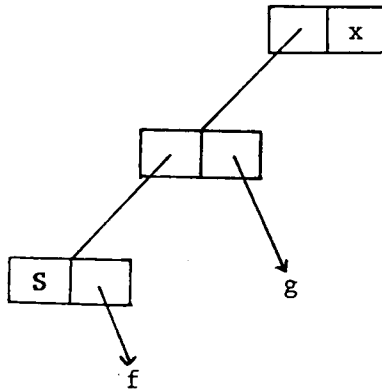
Combinators and the other basic operators in the machine are implemented as instructions. For example the operation code of an instruction could contain **S** or **K** combinators. An apply cell from Turner's graphs will be built from an "apply" instruction (see Appendix Three), the first two arguments of which form the operator/operand pair of the cell.

The only modification to the instruction format is to allow input arguments to be of type "spare", for reasons that will be explained later. In addition, the argument types of "unk" and "am" will be used during the execution of the program in order to implement demand

propagation, but they should not appear in the source of a user's program.

6.2. Program Format

A program is a textual representation of its graph. Each node is formed by an instruction; the arguments of which are either pointers to other instructions or literal values. For example the graph for Sfgx is:



which is represented by the program:

```
1: apply, prop 2, x
2: apply, prop 3, g
3: S, f
```

Figure 6.2: Program for Sfgx.

The top two nodes of the graph are represented by instructions one and two, while the bottom node is represented by instruction three. The arcs between the nodes of the graph are represented by the prop arguments in the program.

6.3. Instruction Execution

This section describes the way instructions are executed. The scheme used is based on pointer reversal and the partial reduction of expressions as explained in Chapter Five. The following paragraphs describe how the scheme is mapped onto the architecture.

The execution of the program in Figure 6.2 will start when instruction one receives a demand; the return address for which is represented by a "*" in the figure below:

```
1: apply prop 2, x, *
2: apply prop 3, g
3: S f
```

Instruction one will now be executed and argument one will propagate a demand to instruction two. The prop argument is changed to unk and the address of the source of the demand is placed as an output argument in the destination:

```
1: apply, unk, x, *
2: apply, prop 3, g, 1
3: S, f
```

Instruction two now has an output argument and so it will execute and propagate a demand to instruction three. Again the prop argument is replaced by "unk" and the return address placed in an output argument of the destination. Instruction three will, however, eventually have three input arguments, so space must be left for them. For this reason two extra input arguments are placed in the instruction, both of which are of type spare. The return address for the demand is then placed in the first output argument. This technique is used for all the instructions implemented on the architecture. The program will now have the format:

```
1: apply, unk, x, *
2: apply, unk, g, 1
3: S, f, spare, spare, 2
```

The chain of reversed pointers is now complete. In a sequential interpreter the leaf of the tree will now reach back up the chain of reversed pointers to find its arguments and perform the required reduction. In this implementation a different approach is adopted. When the demand arrives at the third instruction it will execute. The rules for executing an instruction have been modified from those used previously, an instruction is now considered executable if all its input arguments have values, as before, or if some of its input arguments are of type spare. When an instruction executes it inspects its input arguments, and if some are of type spare it will return the its own address as its result. In the example therefore, the S instruction will return its own address as its result to the apply instruction:

```
1: apply, unk, x, *
2: apply, 3, g, 1
3: S, f, spare, spare, 2
```

Instruction two will now execute because it has values for all its input arguments. When an apply instruction executes it overwrites itself with the instruction whose address held in argument one. In doing so the apply instruction places its second operand in the first spare input slot of the copied instruction, and then copies across its own output arguments. The program will now have the form:

```
1: apply, unk x, *
2: S, f, g, spare, 1
3: S, f, spare, spare
```

Note that instruction three is not modified in any way; it is still available for use in any other sections of the program that share it. Notice also that although the form of the second instruction has been

changed, its meaning has not, the instruction still applies "S f" to g.

The new version of instruction two will immediately execute, and the process repeated to produce the program given below. Again the apply instruction will copy the result of its demand and add its own input and output arguments:

```
1: S, f, g, x, *
2: S, f, g, spare
3: S, f, spare, spare
```

Instruction two is retained so it may be used by other sections of the program. The complete S instruction will be reduced since all its arguments have values. The final result will be the program:

```
1: apply, prop 4, prop 5, *
2: S, f, g, spare
3: S, f, spare, spare
4: apply, f, x
5: apply, g, x
```

Instructions four and five are the two cells introduced by the operation of S which apply f to x and g to x. The execution of all instructions therefore follows the rules of reduction throughout.

6.4. Implementing Functions

In the original architecture functions were implemented by processes. In particular the processes used for reduction were implemented so that the result overwrote the call. The processes were necessary to keep the addresses for each invocation of a function unique. The technique relied on a new copy of a function body being taken from DM whenever it was needed. This is not appropriate for graph reduction because the operation of combinators ensures that a copy of a function body is taken every time the function is applied. As a result the call instruction is not used by graph reduction; the use of combinators makes

it completely redundant. Since combinators handle all the copying of function definitions the entire graph can now be held in one process in AM.

6.5. A Problem with Lazy Evaluation

In the original architecture, an instruction will be copied into AM when it receives its first token. This is still the case, but the use of lazy evaluation presents a potential problem. After a copy of an instruction is taken from DM it will be executed and its result will reside in AM. The reduction will not therefore benefit future callers of the code held in DM. Such a situation could occur if a function body contained a constant expression, $+ 1 2$ for example. This expression must be reduced in such a way as to allow future users of the body held in DM to benefit from the result. In actual fact the addressing scheme used in the architecture ensures that this is the case. When a token is sent to an instruction it is assumed to reside in AM; since all the destination addresses for tokens specify AM locations. If the instruction is not present it is found in DM by performing a simple mapping on the address. When a constant expression is reduced, the demand tokens sent to it will force a copy of the expression to be taken from DM. The expression is therefore reduced in AM, and the result also held in AM. When the next user of the expression tries to access it, the new user will refer to the same address as the original demand. So the result held in AM will be found. Consequently there will be no need to refer to the code in DM. In this way the result masks the code in DM that produced it, thereby allowing future users of the expression to benefit from the first reduction. Usually the result overwrites the expression in reduction, but the masking described above has the same effect.

Unfortunately this problem was not discovered until graph reduction was implemented.

6.6. Assessment of Combinator Implementation

In this section the implementation of combinators on the architecture is assessed and the decisions taken justified.

6.6.1. Parallel Execution of Combinators

Two possible ways of implementing graph reduction were considered: using a stack, or pointer reversal. The problems with using each will now be described to demonstrate why the system employed was chosen.

Using Stacks

When performing graph reduction using stacks each stack represents a demand chain. In the parallel execution of a program there will be several demand chains active at any time, it follows therefore that there will be several stacks. Each stack is used to allow the combinators being reduced to reach up the tree to find their operands. Unfortunately none of the computational mechanisms provided by the emulated architecture make use of stacks, so this method of graph reduction is not suitable for the work described here. To use stacks at all would require the computational mechanisms to be modified, a situation which should be avoided if at all possible. The use of stacks it is therefore not considered further.

Using Pointer Reversal

If graph reduction is performed using pointer reversal each demand chain is represented by the chain of reversed pointers. Each chain leads back to the root of the expression being reduced. If two chains clash for the use of a shared section of code then one will arrive before the other. The first to arrive will proceed to reverse the code's pointers in the usual way, since it cannot detect that the code is shared and consequently cannot treat it in a special way. When the second demand arrives it will find no forward chain to follow and will therefore have to wait until the previous reduction has been completed and the pointers restored. This scheme is not ideal because it requires the execution of shared code to be performed sequentially, and there is no mechanism within the architecture which allows waiting users to be informed when the shared code is free. In short to deal with each demand sequentially will require demand propagation to be modified. To modify demand propagation in this manner would be to admit that the mechanisms embodied in the architecture are inadequate. This should only be done if there is no alternative because to do otherwise may invalidate the results of the work. The scheme described earlier overcomes this problem because no combinator reaches up the tree.

Pointer reversal also imposes a performance overhead. The primary reason for this lies in the optimisation used when reducing an instruction to its result. The reader will recall that the instruction reaches back up the tree to find its arguments, and in doing so effectively flattens the graph which represents it. If a section of code is shared, and each user is dealt with in turn, then each user flattens the tree. Thus the optimisation of reaching back up the tree actually increases

the overheads because the same section of the tree is repeatedly flattened. If the operation of demand propagation is followed throughout the situation could be improved. The demand propagation scheme of reducing combinators adopted for the architecture flattens the instruction in stages. Each time a stage is complete, those instructions which share the result are informed and passed the address of the reduced instruction. Each section of the tree is therefore flattened only once.

The scheme of graph reduction proposed in this chapter is, therefore, both more efficient than simple pointer reversal, and more suitable for the architecture.

Summary

Most of the above comments refer to the implementation techniques available for graph reduction and few to the mechanisms provided by the architecture. This is indicative of the fact that no major problems were encountered when implementing graph reduction. Only one minor change to the emulator was necessary, which allowed input arguments to be of type spare. This modification allows partially evaluated instructions to be held in memory for future use. When combinators were implemented they presented no major difficulties; the code necessary was only slightly longer than that required to implement the instructions of the architecture described in Chapter Four.

All the modifications made to the emulator were made to the top layer of the architecture referred to in Chapter Four, the layer which implements the instructions. No modifications were made to the lower level, the layer which implements the computational mechanisms described in Chapter Three. This indicates that the mechanisms described in the

classification are able to support functional languages, and in particular graph reduction.

CHAPTER SEVEN

IMPLEMENTATION TECHNIQUES FOR LOGIC LANGUAGES

The purpose of this chapter is to describe the various options available when implementing logic languages, and to justify the choices made when implementing logic on the emulated architecture.

Before commencing the main body of this chapter the description of logic given earlier in the thesis is summarised.

7.1. Summary of Logic Languages

A logic program is built from a collection of relations. Each relation consists of several clauses which collectively define how a relation's parameters are related to one another. For example the grandparent relation:

```
grandparent(X,Y) :- parent(X,Z),parent(Z,Y)
```

relates X and Y in such a way as to make Y the grandparent of X.

Each clause in a relation defines part of the relation's behaviour, typically it will define the relation for certain combinations of input parameter values. Each clause consists of a head and a body, and the body in turn consists of a set of goals. The head, which specifies the formal parameters, is to the left of the implication symbol, ":-", and the body is to the right. Each goal in the body calls the relation it names and passes the specified actual parameters.

There are two special types of clause in a logic program. One with no body is an assertion:

```
parent(fred,bert)
```

and specifies that the formal parameters are always related, in this case that fred is the parent of bert. A clause with no head is a question:

```
:- grandparent(fred,GP)
```

and asks in what circumstances the relation holds, in this case what values of GP exist such that each value is a grandparent of fred.

7.2. Search Tree

The concept of a search tree was introduced in Chapter Two and is important in this chapter. The description given in Chapter Two is therefore summarised here. Recall that the execution of the program starts with the execution of the user's question, which specifies the result the user requires. A goal is selected from this specification and executed. Each clause which is successfully unified in the called relation gives rise to a modified form of the specification. Each new specification is then executed, and a goal selected from it. The whole process is then repeated.

Each time a goal is executed a new activation record is created which reflects the modified form of the specification. The whole process gives rise to the search tree described in Chapter Two, and illustrated again below. Each node represents an activation record.

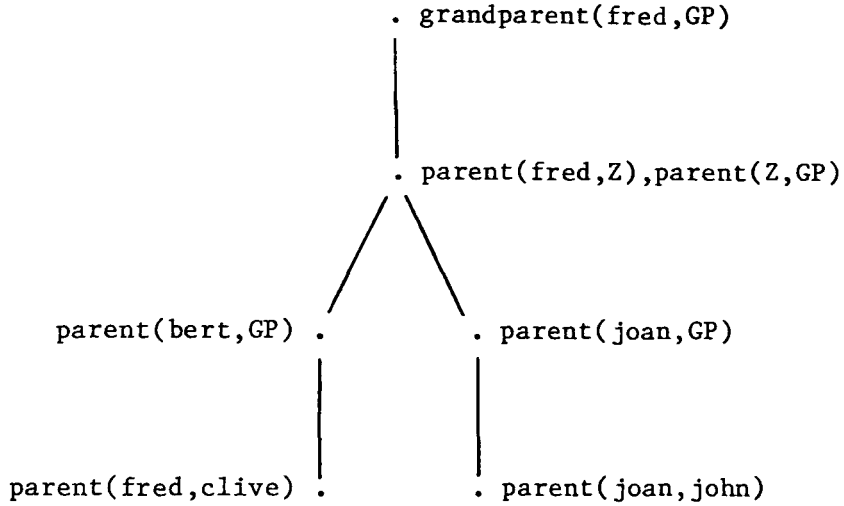


Figure 7.1: Search tree for "grandparent(fred,GP)"

7.3. Unification

In Chapter Two the unification algorithm was described as producing substitutions which make elements of a set the same. In a practical interpreter the substitutions are implemented as parameter bindings. The elements of the set to be unified will be the parameters of the goal and the head of the called clause.

Consider the example:

$$\{g(X),g(Y)\}$$

the the element to the left is the caller, and the one on the right is the head of the called clause, therefore X is the actual parameter and Y the formal parameter. Constant terms in the two literals (i.e. the head and goal) to be unified must be equal, as before. If one of X or Y is a variable, and the other is a constant, then the substitution will be $\{t/v\}$, where t is the constant term and v is the variable, as explained in Chapter Two. Suppose X is a constant value, and that Y is a variable, the value X must be substituted into the body of the clause of

which $g(Y)$ is the head. The substitution to be chosen must therefore be $\{X/Y\}$, namely substitute X for Y . If Y is a constant and X a variable then the $\{Y/X\}$ substitution must be chosen because it allows the result to be passed back to the caller. If both terms are variables, then Y will eventually have a value to be passed back to the caller, so one must choose the same substitution as one would if Y were constant in the first place, namely $\{Y/X\}$.

When implementing logic the binding of two parameters with unknown values is usually represented as a pointer from the formal parameter to the actual parameter. When a value is bound to the formal parameter, the pointer is followed and the actual parameter used to hold the value, thereby accomplishing the substitution of the formal parameter for the actual parameter. In this way the result generated by the clause is passed to the caller. If a single formal parameter has two actual parameters unified with it, then one of the actual parameters is made to point to the other. For example consider the equal relation, written:

$\text{equal}(X,X).$

If the equal relation is called by the goal:

$\text{equal}(A,B)$

then after X and A have been unified the binding will be:

```

formal          actual
 X  ───────────> A
    
```

Now X and A are the same variable, so the unification of X and B is in fact the unification of A and B , giving the final binding of:

```

formal          actual
 X  ───────────> A
                                )
                                v
                                B
    
```


The binding of A and B remains in existence after the execution of equal has finished; any value bound to A or B is automatically bound to the other, thereby ensuring the equality of A and B.

7.4. Structures

Structures are built from functors applications, and may cause some problems when implementing logic. Each structure will typically contain references to variables belonging to the clause which created the structure. If the structure is returned as a result then the variables the structure refers to must persist after the clause which generated the structure has terminated. For this reason many interpreters use an auxiliary stack to hold variables referred to from structures. Activation records on the auxiliary stack are only popped when the structures which refer to the activation record are deleted. This occurs when the variable which holds the structure is deleted, or when the branch of the tree which created the structure fails.

7.5. Negation

As described in Chapter Two, when a goal is negated its success is interpreted as failure, and its failure as success. In terms of the search tree, when a negated goal gives rise to a subtree, the failure of one of the subtree's branches means that the negated goal has succeeded in that branch. As with any other goal, if a goal is successful then the branch which gave rise to the success will execute the next goal in

the clause containing the successful goal. Consider the example below, which is true if there are only two generations to X's family, i.e. that X is a parent but not a grandparent.

`twogen(X) :- ~grandparent(X,Y),parent(X,U)`

A search tree for `twogen(john)` would have the form:

```

      . twogen(john)
      |
      . ~grandparent(john,Y),parent(john,U)
      |
      . parent(john,Z),parent(Z,Y)
```

The goal at the leaf fails because john has no children. The subtree was generated by `~grandparent(john,X)`, so the `~grandparent(john,X)` goal has succeeded. The next goal to be executed is `parent(john,U)`. The same would be true no matter how many nodes lie between the one which executed the negated goal and the goal which failed. For example:

```

      . ~grandparent(john,Y),parent(john,U).
      .
      .
      .
      . parent(john,Z),parent(Z,Y)
```

the next goal to be executed is still `parent(john,U)`. If a goal which is negated succeeds, then the clause to which it belongs fails. This failure is treated in the same way as the failure of any other clause. If the branch of the tree was created by a negated goal, the goal following the negated goal must be executed. If the branch was not created by a negated goal, the branch is not pursued any further in the search for results.

7.6. Variable Binding

There are two standard ways to implement variable binding in logic:

- 1) Each variable is a pointer to a concrete representation of a term.
- 2) Each variable is implemented as a pointer to a shared template of the term. A template represents the format of a structure. The pointer or molecule has two components, the first points to the template for the term, and the second to the environment in which the variables of the template should be dereferenced.

The first method is referred to as "copying pure code"[56] and the second as "Structure Sharing"[22].

Both methods hold the values for variables in environments, or activation records, which are created when a clause is called. The environment will hold the values for all the variables used in the clause, and all the goals within a clause refer to these variables by using an index into the environment.

7.6.1. Copying Pure Code

Each time a term is constructed a concrete representation of the term is built. If the term contains any constants, the concrete representation of the term will hold a pointer to the values. If the term refers to variables with undefined values, the representation of the term will contain a pointer to the variable's location in the activation record where the variable was introduced.

7.6.2. Structure Sharing

Structure Sharing seeks to reduce the number of copies made of items by using templates to represent every structure used in a program. In doing so it avoids the need to create a concrete copy of any structure. The variables referred to by the template are implemented as indexes to their entries in an environment. The molecule which points to the structure will specify the environment in which the variables will be dereferenced. There are two types of structure that may be represented by templates: goals and terms. A goal only contains constant values: an index into the environment for a variable or a literal constant. Each instance of a goal is therefore pure and may share one template.

Structure Sharing can also result in the saving of space because of its ability to share a single template of a term, or the components of a term. Consider the example of append:

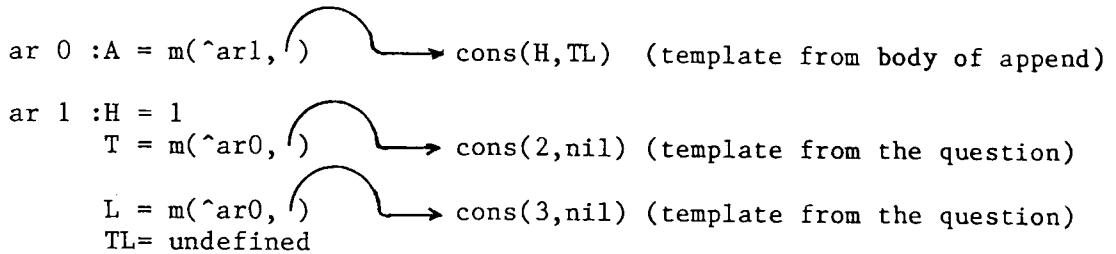
```
append(nil,L,L).
append(cons(H,T),L,cons(H,TL):-append(T,L,TL).

:-append(cons(1,cons(2,nil)),cons(3,nil),A).
```

The first execution of the append relation will result in its second clause being executed within the environment:

```
H = 1
T = cons(2,nil)
L = cons(3,nil)
TL= undefined
```

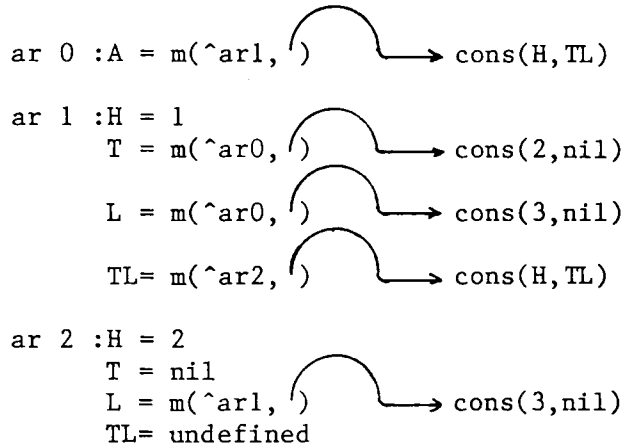
Instead of copying terms which form the value of a variable, it is possible to simply store a molecule which points to the original template:



ar = activation record
 m = molecule(environment pointer,template pointer)
 ^arn = a pointer to activation record n

In the figure above ar0 is the activation record for the question, and ar1 the activation record for the execution of the second clause of the append relation.

The second call of append will be handled in the same way:



The second activation record is created by the recursive call to append made from ar1's clause.

By sharing terms a great deal of copying and rebuilding of structures is avoided. The functor cons was chosen above because it is the most common. In both diagrams the environments for the failed calls of append(nil,L,L) are not shown for reasons of clarity.

7.6.3. Assessment of Variable Binding

The copying of pure code will consume a significant amount of space because each functor cell must be created as required. Accessing a term's components will be fast because one may go directly to the appropriate part and obtain the desired value; there is only a limited need to access the environment.

In contrast Structure Sharings allows binding to be established quickly because the structure need not actually be created, but access to values may be less efficient because the interpreter must repeatedly refer to the environment.

7.7. Parallelism in Logic Languages

There are two ways in which a logic program can give rise to parallel execution. These are termed OR-parallelism and AND-parallelism; they are both described below.

7.7.1. OR-Parallelism

The name OR-parallel is used because parallelism occurs only where execution of alternative clauses from the same relation occurs: that is each branch of the search tree is pursued in parallel with the others. From the search tree shown in Figure 7.1 one can observe that all the branches are independent of one another. If several branches share a variable, then each branch in fact refers to a different instance of the variable, thus the independence of each branch is ensured.

7.7.2. AND-Parallelism

AND-parallelism occurs within a clause, and is generally more difficult to achieve than OR-parallelism. Consider the sole clause in the grandparent relation:

```
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
```

Normally the goals in a clause will be selected from left to right, but when using AND-parallelism both the goals are obeyed at the same time.

The difficulty of implementing AND-parallelism lies in the way variables are shared between goals of a clause; all goals must agree on the value for a particular variable. There are two ways to achieve this. Firstly all goals in a clause can be run to completion and the values returned for each variable can be compared so that a consistent set is found. This is the could be called an atomic execution scheme and suffers from the disadvantage that many results are produced but then discarded, wasting the processing effort put into them. Consider the grandparent clause, above, when executing the question:

```
:-grandparent(fred,GP).
```

When using the atomic scheme the first goal will produce two values for Z: bert and joan. The second goal will produce four pairs of values for Z and Y because it is not aware of the values for Z chosen by the first goal, because the goals are obeyed in isolation. Of the four pairs of values produced by goal two, only two are satisfactory; those which have bert and joan as the value for Z. If the value for Z had been available to goal two during its execution this goal could have avoided producing the superfluous results.

The second alternative is described by Pollard[59] and starts all goals in parallel but makes use of values produced by one goal to direct the search of the other goals in a clause, even during their execution. As each value for a variable is created any part of the search tree which has an inconsistent value for the same variable will be deleted, and therefore as little computation as possible will be wasted. Only branches that are likely to lead to acceptable solutions will be followed. This scheme is rather complicated, and may have considerable overheads. It remains to be seen if the overheads are worth the extra parallelism.

An AND-parallel scheme will usually include OR-parallelism, giving a search tree:

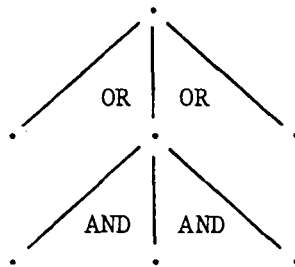


Figure 7.2: AND/OR search tree.

The top subtree represents the parallel execution of the clauses from one relation, while the bottom one represents the parallel execution of the goals in a clause.

7.8. Parallel Implementation

From this point onwards only the implementation of OR parallelism is considered. The reasons for this are explained at the end of the following chapter.

7.8.1. Storage Schemes

The major problem that occurs when implementing an OR parallel scheme is that for managing the storage of the alternative values of each variable. Each time the search tree branches, the possibility of producing an additional answer to the user's question is introduced. This implies that additional space for the answer, and all intermediate results, must be created. In general it is not easy to determine in advance which variables will be required to hold additional results. The only choice therefore is to allow any variable to hold the results.

In the following three sections possible ways of dealing with these difficulties are outlined.

A Simple Storage Scheme

A simple way of producing space for each result is to make a copy of every uninstantiated variable when the search tree branches. This makes each branch of the tree totally independent except for shared results which have been generated higher up the tree. Each branch may now be executed independently.

One Solution at a Time

Conery and Kibler[24] describe a scheme which produces results by OR-parallelism, but only returns the results to the caller one at a time. If the first result is not satisfactory the caller asks the goal for an alternative, and continues to do so until the results have been exhausted. The scheme would be better if Conery and Kibler had allowed several searches to continue concurrently by passing all results to the

following goals simultaneously. In their scheme any results which have not been demanded are stored in the producer awaiting a demand. The paper does not describe how these results are to be held. The scheme therefore produces results by OR parallel execution, but does not use them in parallel. In view of the sacrifice of parallelism made, the implementation scheme would not form a good foundation for the evaluation of a set of mechanisms in which parallelism is the predominant feature. The scheme is not, therefore, considered further.

Multi-Value Variables

The last way of allocating space for the alternative results of a logic program is to allow a variable to hold more than one value[59]. This implies that each variable has a flexible structure in which there is one partition for each value. As each value is produced it is stored in a newly created partition of the appropriate variable. Now there is no longer any need to duplicate the environment of a branch every time the search tree divides. Instead each variable is able to store any results that may be produced. Since the variable must be shared between all the parts of the search tree beneath the node where it is introduced, it follows that its storage space must reside in the activation record corresponding to this node.

Comparison of Storage Schemes

When comparing these storage schemes, the simpler OR scheme suffers from the obvious disadvantage of consuming large amounts of space. Most environments will be used by searches that will ultimately fail. These searches are unlikely to make full use of the environment because they have not run to completion. Only successful searches will use every

variable because only they will provide values for variables introduced near the root of the tree. A lot of space is therefore wasted.

Multi-value variables are more efficient in their use of space because new partitions are only created when required. This does however imply a flexible data structure with the inherent overheads of pointers and management.

7.8.2. Control Mechanisms

The other major choice for the implementor concerns the control mechanism. There are two types available: one follows from the procedural description of logic, and the other from the search tree. Both schemes use the same mechanism when calling a relation. A new activation record is created for each clause in the called relation and a unification with each clause head is attempted. All successfully called clauses proceed with their execution while the remainder are deleted since they have failed. The two schemes differ in the way activation records and the results are managed on the completion of a clause.

Procedure Model

When using the procedure calling model, the activation record for a clause is deleted when clause is complete. The procedure model must therefore use the auxiliary stack to hold results. It is the responsibility of the clause receiving the results to pass them onto the next goal in the clause's body, and therefore to start an instance of the goal for each result tuple returned.

Search Tree Model

The second scheme follows the structure of the search tree. As each goal is executed a new activation record is created for all the clauses in the called relation whose heads were successfully unified with the goal. The results are extracted when the last goal of the question gives rise to a branch of the tree which is unified with an assertion. Nodes will only be deleted from the tree when a branch fails or successfully reaches a conclusion.

Assessment of the Control Mechanisms

The procedure calling model will use as little space as possible because memory is reclaimed at the earliest opportunity. Against it, however, is the complexity of using several results returned from one goal to start several versions of the next goal.

The disadvantage of following the search tree model is obvious; it creates an activation record for each goal executed and maintains it until the branch terminates due to success or failure. The benefits of this scheme come from the simple way alternative searches are dealt with. There is no need for a clause to start a parallel execution of a subsequent goal because there is no set of alternative values for the goal to operate on. Each branch of the tree only produces one value for each variable, consequently each branch is responsible for starting the single execution of the next goal selected.

A good way to implement a logic language is to use a combination of the procedure model and the search tree model. The procedure model will be used to allocate storage, and the search tree model to control the

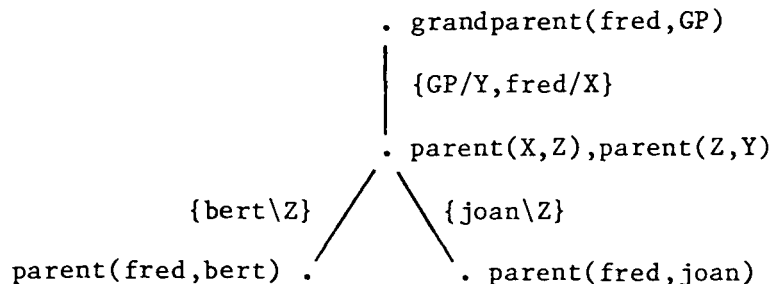
flow of execution.

Pollard[59] describes a scheme which combines multi-valued variables and the search tree model. This scheme has the simplicity of the search tree model and the efficiency of the procedure model, but with the overheads of multivalued variables. Pollard also describes a way of associating a result with the branch of the tree which produced it. In the following section an alternative is described.

7.8.3. An Alternative Execution Scheme

The scheme proposed below uses the search tree model and the simple storage scheme, but copies the stack in a piecemeal fashion. This has several advantages over Pollard's scheme.

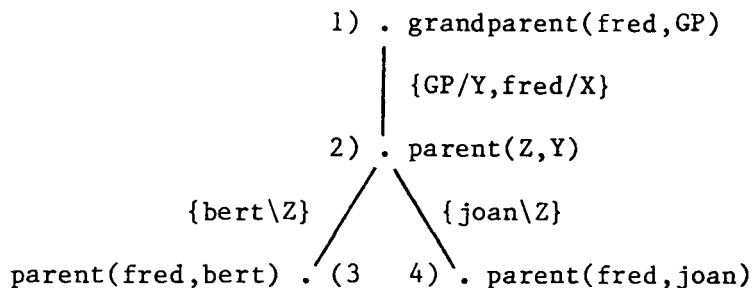
When the program commences execution, the first goal in the user's question is obeyed. All clauses in the corresponding relation which are successfully unified with the goal are allowed to proceed with their execution in parallel. Each will now execute the first goals in their bodies, so new activation records are created, and the successfully called clause allowed to execute. The generation of activation records for a particular branch will proceed until a node is created which represents an assertion. For instance if one considers the grandparent relation, the tree at this stage in its execution will have the form:



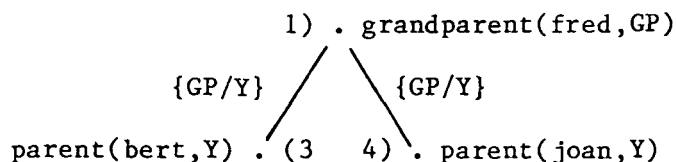
In the figure above the substitutions on the arc are the result of the

unification carried out when the goal at the head of the clause for the node above the arc is executed. The symbol \ denotes a substitution that passes a result back to the caller.

The leaf nodes of the diagram above have each found an assertion which satisfies the goal parent(fred,Z). The remaining goal to be executed is parent(Z,Y), the second goal in the body of the grandparent clause. Since the first goal has been executed it may be removed from the tree.

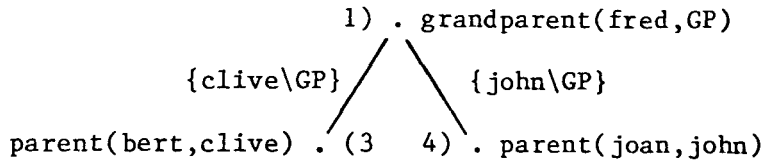


All nodes now represent clauses which have started their execution, but have not completed it. Nodes 3) and 4) have finished executing their bodies but have yet to return their results, whereas node 2) still has one goal outstanding. Since both the leaf nodes provide a value for Z, node 2) may now be executed using the two values produced. This is achieved by copying down node 2) into nodes 3) and 4), and placing the value for Z into each as this is done. The arcs of the tree point from one generation to the previous one so each node can refer to the node which called it.



The new goals in nodes 3) and 4) are executed to provide values for Y. Notice that node 2) has been deleted because it is only connected to its

parent, nodes 3) and 4) have taken over its operation. After the execution of the goals belonging to nodes 3) and 4) is completed the tree has the form:



Both the leaf nodes now have values for Y, and since the execution of the clauses they represent has finished, node 1) may be copied down and executed:

```
grandparent(fred,clive) . (3  4) . grandparent(fred,joan)
```

Node 1) has now been deleted because it is disconnected from the tree. When nodes 3) and 4) terminate, they will attempt to copy down their callers. On finding that there are no nodes above them they will print their results.

The operation of the model may be summarised as follows:

- 1) When a relation is called, an activation record is created for each clause in the relation.
- 2) Next a unification of the calling goal and the called clause is attempted.
- 3) Failure will lead to deletion of the activation record.
- 4) Lastly, the relation specified by the first goal in each clause body is called. The whole process is then repeated.

Steps one to four are repeated generating as many nodes and branches as are necessary to reach the assertions of the program. When a branch reaches a leaf cell it will proceed to step five:

- 5) When a clause finishes execution, or an assertion is unified with the calling goal, the parent's activation record is copied down into the activation record for the terminated clause. As this is carried out any results produced by the terminated clause are put into the new copy of the parent's activation record.
- 6) The clause belonging to the copied activation record is allowed to continue its execution using the new copy, which it does by returning to step one.

Garbage collection of redundant activation records may be achieved in either of two ways:

- 1) In step 3) a failed unification will cause its activation record to be deleted. The parent of this activation record could maintain a count of all active descendants. Upon termination of a descendant, the parent's count is decremented, and its activation record deleted if the count becomes zero. This process would ripple up the tree deleting as many activation records as possible.
- 2) A mark scan garbage collector could be implemented by starting the mark phase from each active goal. Each branch that is still active will have such a goal, but the redundant ones will not. The activation records in redundant branched will not therefore be marked and consequently be reclaimed by the scan phase.

Unification in the Alternative Logic Scheme

As described earlier, the unification algorithm often requires the modification of actual parameters before the body of the clause can be executed. For example the actual parameters may need to be chained, as was the case for the equal relation. It follows therefore, that if two branches of the tree are to be executed in parallel then each branch must have a copy of all the actual parameters which are modified during unification. One way to achieve this would be to use the formal parameter to hold the result, and copy it into the caller's activation record when the caller is copied down. Unfortunately this is not possible because a result will often require several pieces of data, such as the result its self and the information necessary to build a chain of actual parameters, which is more than the formal parameters can hold. Thus instead of using the formal parameters to hold the result, a copy is made of each actual parameter which is modified. These copies may then be chained together and used to hold the result. When the clause's caller is copied down, the information contained in the copies of the actual parameters is used to pass the result of the terminated clause into the copied version of the caller's activation record.

Assessment of the New Logic Scheme

The alternative logic scheme described above is believed to have the simplicity of the search tree model together with the efficiency of the procedural model. The main advantage it has over the use of multi-valued variables is that it avoids contention for data, and that the storage structure used in the new scheme is more efficient.

In the multi-valued variable scheme all the values for a variable are held at the node of the search tree which introduced the variable. In order to access the values of the variable the program must refer to the processor which holds that node. Any references to a variable will therefore give rise to contention, both for access to the processor, and the communication paths which lead to it. By keeping all branches of the tree totally independent, all contention of this type is avoided.

Each variable in a multi-valued scheme will have a complex structure built from cells containing values and pointers to other cells. Each time the value of a variable is required, the structure must be searched to find the correct instance of the variable. Performance may be improved by using, for example, binary trees or hash tables, but in both cases space will be consumed by collision chains etc. Even so, no matter how efficient the search is, it must still be performed. A simple indexed addressing scheme cannot be used for a multi-valued variable because only some of the potential values of the variable will exist at any one time. Thus implementing parallelism with multi-valued variables imposes an efficiency penalty. By comparison^{to} new logic scheme consumes space only when making copies of some actual parameters. These copies contain values that would be created, and therefore stored, no matter which logic scheme was chosen. The parameters which are copied are only those which are given values during the execution of the clause. The new logic scheme does not therefore waste space. The storage structure used in the new logic scheme is as simple as that used in a sequential logic interpreter, and therefore has the same efficiency when being accessed.

Garbage collection of multi-valued variables will require all accesses to a variable to be suspended until the garbage collection of redundant elements is complete. This is because the data structure which represents the variable may enter an inconsistent state during the garbage collection, and may therefore deliver spurious data if it is accessed at this time. The new logic scheme, in contrast, does not garbage collect individual values from an activation record, only complete activation records, and then only after all their users have ceased to exist. There is no need, therefore, to suspend access to an activation record during garbage collection, and hence garbage collection may proceed in parallel with program execution, the two are isolated.

One potential disadvantage of the new model is the overhead of copying the parent's activation record down, which is carried out for every goal in the program. Any logic implementation must, however, create an activation record for each clause of the called relation. So the new scheme does not introduce any new overheads, indeed it gains because it re-uses existing activation records. Another potential problem is the possibility that the parent's activation record is larger than the child's, which will mean that the latter must be expanded when the parent's activation record is copied into it. In practice a parallel machine architecture will probably allocate memory in fixed length sections to simplify memory management. Analysis of some logic programs may show that a certain size of record will be sufficient for virtually all clauses. Should this prove to be the case it would be possible to build most activation records with one segment of that size. The overhead of expanding activation records will therefore have almost disappeared.

In summary the new logic scheme simplifies several aspects of an OR-parallel implementation of logic, and is more efficient than multi-valued variables.

CHAPTER EIGHT

LOGIC LANGUAGES ON THE MACHINE ARCHITECTURE

This chapter describes the implementation of OR-parallelism on the architecture specified in Chapter Four. A more detailed description of the modifications to the original emulator may be found in Appendix Four. The aim of the implementation is to demonstrate that the mechanisms provided by the architecture are able to support logic. The scheme used as the basis of the implementation is that described at the end of the previous chapter. Functors have been omitted from the implementation because they add considerably to the complexity of the interpreter, without providing any additional information about the suitability of the computational mechanisms for logic. The way functors could be implemented is described in order to illustrate the extra work and to demonstrate the low value of the additional results. The implementation of logic described here uses structure sharing to reduce the memory requirement as much as possible; this allows more space for the activation records produced by OR parallelism.

8.1. Instruction Format

Four instructions are used by the architecture to implement clauses:

- 1) clause: This is the head of a clause, and provides two pieces of information in its arguments. The first argument is the length of the clause's activation record, the remaining arguments form the list of formal parameters. Each parameter may either be a literal value or a PM address.
- 2) goal: This instruction corresponds to the goal in a clause body. It provides two pieces of information: the first argument is the "procedure" index for the relation being called, the remaining arguments are the actual parameters of the goal. Each parameter may either be a literal value or an index into the activation record. The value held in the activation record may either be a literal or an argument of type unknown.
- 3) ngoal: This instruction implements a negated goal, but in all other respects it is identical to the goal instruction.
- 4) fail: This instruction causes the clause to which it belongs to fail. This instruction is used when implementing negation.
- 5) endc: This instruction is placed at the end of a clause. Its main responsibility is to copy down its parent's activation record. Since the instruction is the last in the clause its execution implies that the clause has been successful.

In the logic implementation there are five arithmetic operators, and six comparisons; all those described in Chapter Four are provided. Each arithmetic operator is implemented as an individual instruction with three arguments. The instruction uses the arguments one and two to calculate argument three. For example add gives argument three a value equal to the sum of arguments one and two. If only two of arguments

have values, the third will be produced as the result. Less than two values will cause an error. Each comparison is implemented as a separate instruction which takes two values, and compares them in the appropriate way. The operation is successful if the comparison returns true, and fails if it is false. Should less than two parameters be supplied; the instruction will abort.

8.2. Clause Format

A clause starts with a clause instruction, and is followed by any number of goal instructions. The last instruction in every clause will be an endc instruction. An example of a clause is:

```
clause 2,1,2,PM address
goal 1,1,1
goal 2,2,1
endc
```

Figure 8.1: Example of a clause.

8.3. Program Format

A logic program is divided into two parts: the user's question, and the relations that will be used to answer the question. Both have broadly the same format: the question is a clause body (a clause without a "clause" instruction), and a relation is a set of clauses whose format is like that shown above. Both the endc instruction at the end of the question, and the one at the end of the last clause in each relation have one operand which is a literal of any value. This is used to signify the boundary between definitions when the clauses reside in DM. The skeleton format of a program is:

```
goal ...
goal ... users question
endc 0
clause ... start of relation
goal ...
goal ...
endc
clause ... second clause in relation
goal ...
goal ...
endc 0 end of relation and program
```

Figure 8.2: Skeleton program.

8.4. Process Format

Whenever a relation is called all its clauses are allocated a process in which clause's instructions will be obeyed. The memory belonging to the process corresponds to the activation record of the clause. Each activation record includes the following information:

- 1) The length of the activation record.
- 2) The address of the next goal to be executed in the clause which belongs to this activation record.
- 3) The address of the goal which called this clause.

8.5. Execution Cycle

The execution cycle of the machine has two major phases, firstly that of calling a relation and performing the corresponding unifications, and secondly copying down the parent's activation record when a clause has finished executing. When a clause is executed each successful instruction is responsible for triggering the following goal's execution. If a goal fails the steps needed to implement negation are

followed. A successful goal triggers the next goal's execution by sending it a control token. The count of all instructions should therefore have an initial value of one, because each will receive one control token. The first instruction in the question must have a count of zero because it triggers the execution of the entire program and therefore must be executable when the program is loaded.

In the following two sections the two halves of the execution of a goal, the call and the return, are described in more detail.

Calling a Relation

When a goal calls a particular relation all the clauses within the relation start executing. The operation of the goal instruction will generate a process for each clause in the relation and set up the clauses' activation records. The first instruction to be executed within the clause is the clause instruction itself.

The primary task of the clause instruction is to carry out the unification of the formal and actual parameters. In doing so it copies all actual parameters into its environment. If the unification is successful the first goal in the clause body is executed, whereas if the unification fails the appropriate action is taken (see the description of negation).

Finishing a Clause

The endc instruction is situated at the end of every clause body in the program including the user's question. When this instruction is executed it takes a copy of its parents activation record and merges the copy with the results from its own clause. Should the clause which has

finished executing have no parent the values of all variables in the activation record are printed.

8.6. Implementing Unification

The implementation of unification on the emulated architecture proved to be more complex than expected, mainly because of difficulties encountered when accessing locations belonging to one process, while executing another. Accesses of this nature must be carried out to obtain the values of actual parameters. Unification in the new logic scheme also requires those actual parameters which are modified to be copied into the called clause's activation record. To access the copied parameters one would want to have an indexed addressing scheme in which the position the argument occupies in the goal gives the index into the table of copied parameters. Unfortunately this would mean wasting space because only modified parameters are copied. This means that the table must be padded to ensure the indexes remain correct. An alternative way to access the copied actual parameters is via an indirection table, in which each location points to the copy of an actual parameter. The index into the indirection table for a parameter will be the same as the index for the parameter in the goal, but only certain elements will point to locations. These are the locations copied during unification. In this way an indexed addressing scheme may still be used, and the minimum possible space consumed. The indirection table is only required during unification and may be discarded afterwards, it will not consume space in the activation record. The implementation of this scheme was found to be undesirably complex because of the addressing problems the architecture creates. So the logic implementation but does in fact copy all actual parameters into the called clause's activation record. Copy-

ing all parameters overcomes the difficulties the architecture has when accessing parameters because both the formal and actual parameters reside in one process, and because there is no need for an indirection table. To further simplify the unification algorithm the occur check is not included.

8.7. Implementing Negation

Negation is implemented using a combination of three instructions: ngoal, fail and endc. The call of a negated goal has the form:

```
ngoal ...
fail ...
goal1 ...
```

the first two instructions form the negated goal, while goal₁ is the next goal of the clause. When the ngoal instruction calls a relation all the activation records for the relation are marked as being negated, and contain a pointer, the negated pointer, to the node which executed the ngoal instruction. The address of goal₁ is recorded as the next goal to be executed in the calling clause. When any of goals which are descendants of the negated goal are executed, the negated pointer is passed on to the called clause, but the negated flag in the new node is not set. This gives the tree the form:

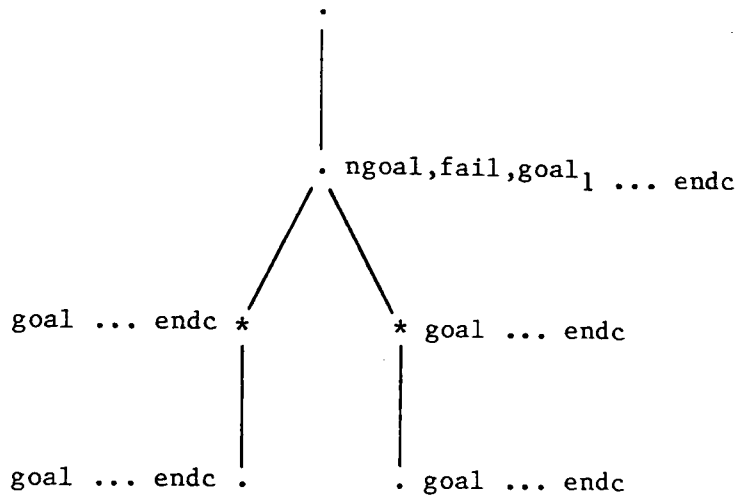


Figure 8.3: Search tree with negated pointers.

The nodes "*" are the one which have the negated flag set. There are two possibilities which are of interest: the failure of one of the branches of the tree descended from the negated goal, and the success of one of these branches.

If a branch fails then goal₁ must be executed because the negated goal has succeeded. This is achieved by copying down the node pointed to by the negated pointer of the failed clause, and executing the clause belonging to the copied node.

If a branch is successful then all the nodes in the branch will have terminated successfully and will have copied down their callers. Eventually the endc instruction for the "*" nodes will be executed. When the endc instruction is executed it will inspect the negated flag, discover the clause is negated, and force the fail instruction in its parent to execute. The clause containing the ngoal instruction will therefore fail. If the failed clause is also descended from a negated goal, it too will have a negated pointer, and the process of negation will be repeated. If it has no negated pointer the execution of the

clause will stop.

8.8. Architecture Modification

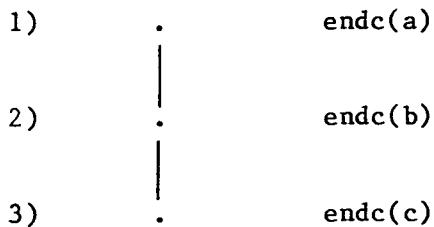
This section describes a modification to the emulator which overcomes a particular difficulty concerned with recursion. The problem involves the clash in the use of memory locations in AM, which is brought about when a recursive clause copies down its ancestor. Unfortunately the problem was only discovered when recursive programs were run on the finished emulator.

In the original architecture an instruction is copied from the DM into the AM when it receives its first argument. Once in the AM it remains there until it is executed, at which time it can either be retained in AM or deleted. This scheme causes a problem when used to implement logic; consider a relation in which a clause is tail recursive:

```

r(      ).
r(      ):- . . . r(      ).
```

When the clause reaches the end of its recursions, because a call matched only the assertion within the relation, the tree will have the form:



The next instruction to be obeyed for each process is endc. The bottom node represents the activation record for the assertion, and the ones above are those for the recursive calls that led to the bottom node.

Both endc instructions for the top two nodes are copies of the same instruction in DM because nodes 1) and 2) are instantiations of the same clause, namely the one that is tail recursive. Both instructions therefore have the same location number in their addresses, but belong to different processes. When the endc for node 3) is executed it will copy down node 2). The endc(b) instruction will, therefore, have moved from the process for node 2) to that of node 3). The endc(b) instruction is now executed and is copied into AM. During the execution of the endc(b) instruction it will copy the activation record for node 1) down into node 3), and then signal the corresponding endc, namely endc(a). It is at this point in the execution that the problem occurs. The executing endc (i.e. endc(b)) is in the process corresponding to node 3). The signaled endc instruction belongs to the activation record which has just been copied down into node 3), and therefore belongs to the same process as node 3). When the signal arrives at the endc(a) instruction the processor will try and copy the endc(a) instruction into AM. As was stated earlier both endc(b) and endc(a) have the same location, but unfortunately they now have the same process. Thus when the processor tries to load endc(a) into AM there will be a clash of addresses which will cause a processor error.

A solution to the above problem is to move every instruction directly from the DM to the execution queue when the instruction is signaled, providing it is immediately executable, which will be the case in a logic program. Since the queue allows more than one copy of an instruction to be held, the clash will not occur.

The modifications required to the emulator were fortunately fairly minor; whenever an instruction is put on the queue of executable instructions it is now removed from AM instead of remaining there, as

occurred in the original architecture. If full use had been made of the features of the architecture when implementing logic additional problems could have occurred. For example if instructions were retained in AM after they have been executed, the clash could still occur. In addition the solution adopted would not have worked if a logic instruction required more than one token before becoming executable. Had this been the case, the instruction would have to be moved from the DM into the AM to await its full complement of tokens, thereby allowing the possibility of a clash. The solution, therefore, only works because of the simple way control tokens are used in the implementation. If the full generality of the computational mechanisms were required an alternative solution would have to be found.

8.9. Implementing Functors

Recall that functors were not implemented because they would not provide any worthwhile information about the computational mechanisms implemented on the architecture, and because of the complexity of their implementation. The following section illustrates the complexity by describing some aspects of the way functors can be implemented. The details given are not important in themselves, they are used to illustrate the complexity functors would introduce.

Functors would be implemented by a new instruction "func" whose arguments provide two pieces of information. The first argument is a literal value which corresponds to the compiled form of the functor name. The remaining arguments are the parameters of the functor which may be: simple values, pointers to other functor applications or indexes into activation records.

As mentioned in Chapter Seven, implementing functors requires an auxiliary stack to hold the variables referred to by structures which are returned as results. The variables used in the structure belong to the clause which generated the structure, but must remain in existence after the clause has terminated. These variables are therefore held in the auxiliary stack, which is only popped when the variables are no longer required. Variables which need not persist after the clause has terminated are held in the ordinary stack. The auxiliary stack will be implemented in AM; while the ordinary stack resides in PM.

At present the process number within an address identifies the memory in AM and PM which belongs to that process. This view will not hold for logic. When the parent's activation record is copied down into a terminated clause's activation record, the values in PM may be overwritten (as described above), but those in AM must not be destroyed because they will be the results of the terminated clause and must therefore remain accessible. The new user must however copy down its own activation record in AM. In order to allow the new user of the PM activation record to keep its own results in AM, a new section of AM must be allocated. Thus one activation record may occupy two processes; one for the memory in PM and another for that in AM.

The splitting of a logic activation record across two processes poses an additional problem when fetching operands of an instruction. Until now the activation record for an instruction could be identified by referring to the process number of the executing instruction. This method will still work for variables which are sited in PM, but not for those in AM. To overcome this problem a link between the activation record in PM and that in AM is placed in the head of the PM section of the activation record.

Functor expressions are always pointed to by molecules. If a functor expression need not persist after its clause has terminated, the variables referred to by the expression may reside in either AM or PM. The entry in the expression which refers to the variable will give the correct index into the appropriate memory to obtain the value. If the location is in PM the processor identifies the correct process by using the process number in the molecule. If the location is in AM the processor finds the correct process by following the link from PM to AM. If a functor expression is to persist however, the variables referred to must all reside in AM. The process number in the molecule which points to the expression will identify the correct process in AM. A molecule which points to a local structure (one which will not persist) will therefore have a PM address; a molecule which refers to a structure which is to be a result will have an AM address. In both cases the process number in the molecule will refer to the activation record which holds the expression's variables.

The construction of a molecule introduces a further problem, namely that all the pure code is held in DM. The address the code for the functor expression will, therefore, have two components: the process identifier which specifies DM, and the location within the process for the code itself. Unfortunately this exhausts the fields available in a machine address (argument slots are of no use here). A molecule requires the process number for an activation record as well as the pointer to the pure code. Thus it is not possible to hold a molecule in a machine address. To overcome this problem the location in a molecule will be treated by convention as a pointer to DM, while the process will specify the activation record containing the structure's variables. If the molecule is of type AM the variables reside in AM,

and if it is of type PM the variables reside in PM or AM, as described earlier.

There are two remaining facets to the implementation of functors, namely unification and garbage collection. For unification two of the possible cases are of interest here:

- 1) If two structures are unified then the algorithm will pass along them to unify their components; the variables and constants within the structures are unified in the way described in Chapter Two. At each stage, functors in the two structures which correspond to one another must have the same name, i.e. the literal value of the first argument of the functor must be the same.
- 2) The unification of an undefined variable and a structure can occur two ways:
 - a) If the undefined variable is an actual parameter and the structure is a formal parameter, then the structure's variables will reside in AM because the structure must outlive its clause. The actual parameter is made to hold the molecule which points to the structure.
 - b) In the reverse situation to a) the formal variable will be made to hold a molecule pointing to the actual parameter.

Garbage collection of PM activation records proceeds as described in Chapter Seven. The activation records can be removed when a particular branch of the tree fails. The AM processes may be collected using the mark-scan algorithm. Those AM processes that belong to a branch may be marked by following the the pointers from every molecule in the branch, and the links from the PM activation records of the branch. The

scan phase will now pass through both PM and AM.

It may be seen from the above description that to include an functions in the implementation of logic (on the architecture described in Chapter Four) would have required a considerable amount of work. However the results obtained would not have been particularly relevant to this thesis because all modifications are to the architecture itself and not to the computational mechanisms implemented. It is the evaluation of the computational mechanisms which forms the basis of the work reported here.

8.10. Assessment

When the scheme described in Chapter Seven was implemented on the emulator, severe problems were encountered. Most of these were discovered while the implementation was being designed. This section describes these problems and discusses the conclusions that may be drawn from them.

When implementing an OR-parallel logic scheme on the emulator for the architecture described in Chapter Four, only a small percentage of the original facilities proved useful. Neither data flow nor the demand propagation facilities were used, and control flow is only used in the limited fashion mentioned above. Although one can in principle demand the result (success or failure) from the relation which a goal calls, the complexity of dealing with several results returned in reply to one demand makes this alternative too complex. To implement logic by demand propagation at all would require modification to the mechanism because one demand usually produces only one result. The data flow model is even less appropriate to the implementation of logic. In an OR-parallel

scheme all the clauses of a relation are used, hence there is no firing rule associated with the operation. In the OR-parallel scheme being used the goals within a clause are obeyed left to right, so the flow of data has no effect on the flow of control. Wise[80] has proposed a data flow implementation of logic which forces the programmer to annotate his program to make the flow of data explicit. This provides further evidence that data flow is not able to deal with a pure logic program.

As only control flow proved useful the features of logic had to be build from the primitive operations that this model provides. In effect one is forced to resort to the lowest level features of the emulator to achieve any success at all. It is only the flexibility of control flow that prevented total failure, which serves to emphasise the inadequacies of the mechanisms described in Chapter Three. In short this architecture is no more amenable to logic than the conventional von Neuman machine.

As the mechanisms provided by the architecture were not very helpful in supporting logic, one may conclude that logic does not fit into the classification described in Chapter Three. The primary reason for this would appear to be that none of the existing mechanisms allow for the parallel execution of alternative forms of a procedure/function/clause, each of which return their own results. To use any of the mechanisms provided by the architecture would effectively mean writing a simulator for the logic machine using the model of computation chosen, rather than implementing logic in terms of the model.

The OR-parallel scheme which was adopted simplified the task of implementing logic considerably and therefore allowed the problems produced by the mechanisms to be isolated from those produced by the archi-

ecture. Had another more complex scheme been used the problems of the architecture may well have swamped those of the mechanisms, thereby calling into doubt the validity of the above comments. It is the difficulty of allowing one call on a relation to return several results which led to the scheme described in the previous chapter being devised. The main virtue of the new scheme, from the implementation point of view, is that it keeps the results produced by clauses separate from one another. This simplified the task of writing the emulator for the logic machine considerably because it minimised the complexity of memory management. The problems encountered when trying to handle dynamic data structures in the type of memory used (see Appendix one) were considerable. The whole of the original emulator relied on the mechanisms provided by the architecture to drive the memory. When those mechanisms were discarded the logic emulator had to use the memory in its raw state, which proved a complex task particularly when trying to unify and merge parameters.

It was the desire for simplicity that led to AND-parallelism being omitted from the implementation. To implement a powerful scheme one would want to follow ideas related to those of Pollard[59]. To do this all the values for each variable must be stored together, which in turn implies that multi-valued variables must be used. It was the desire to avoid the complexity that this entails which led to the adoption of the OR-parallel scheme already described. Since it was the inadequacy of the mechanisms and architecture which led to the complexity of multi-valued variables becoming unmanageable in the first place, there seemed little point in proving the mechanisms inadequate for a second time by trying to cope with the additional complexity of AND-parallelism.

CHAPTER NINE
COMBINED FUNCTIONAL AND LOGIC ARCHITECTURE

In this chapter an architecture which is able to support both functional and logic languages is described. Until now the computational mechanisms employed in the architectures described have been those defined in Chapter Three. It has, however, been shown that these mechanisms are not suitable for the efficient support of logic; in this chapter the mechanisms will be modified in the search for a combined architecture for functional and logic languages.

9.1. Combining Functional and Logic Models

In Chapter Seven a new scheme for the parallel execution of logic programs was described. It was based on the notion that upon completion of a clause, the clause will copy down its caller and take over its execution, thereby keeping branches of the search tree independent. Thus when a clause at a leaf terminates it pulls down its caller and executes the remaining goals of the caller. When the caller terminates the clause above the caller is pulled down, and so on until finally the root is pulled down into the leaf and the results are printed.

Reduction also builds a tree, but since the program only produces one result there is no need to pull the root down to the leaves, so instead the tree is collapsed upwards, effectively pulling the results at the leaves progressively towards the root.

This observation can be used to form the basis of a combined functional and logic language architecture. Both types of language will build a tree in the same way, but then manipulate it differently. In effect the tree is built by demand propagation, and the result returned in two ways. Single results pass up the tree, multiple results are copied down. This architecture therefore introduces a new form of demand propagation: demand propagation with multiple results.

9.2. Structure of the Combined Architecture

The new architecture is based on packet communication and consists of a processor and three memories: the instruction memory IM, the data memory DM, and the structure memory SM:

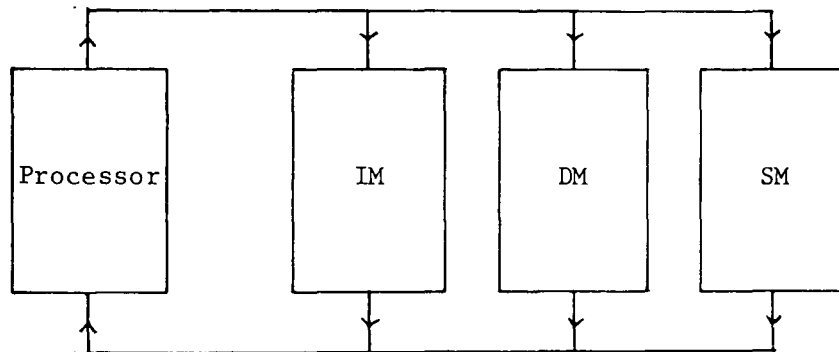


Figure 9.1: Logic machine architecture.

The instruction memory holds all instructions whether active or not. The data memory will hold all activation records each location holds an instruction argument rather than just a value. Lastly the structure memory is used when building structures. It may either hold the auxiliary stack, or a garbage collected heap.

9.3. Structure of Programs

This section describes the way both functional and logic programs are represented in the machine. Function, relation and clause definitions are held in processes allocated by the compiler and which consist of code, held in IM, and skeleton activation records held in DM.

9.3.1. Functional Programs

Functional languages have the notion of scope (described in Chapter Two) which the structure of activation records must reflect. In particular activation records must reflect the structure of recursive qualifications (whererec).

A closure is represented by a process, where the IM section represents the closure's code and the DM section represents the closure's environment (i.e. activation record). A function, f 's, definition is represented by a closure. The first instruction of the function is a "clo" instruction which builds a closure for the function when a demand is propagated to it. The second instruction of the function is a "func" instruction which is responsible for binding the function argument into the activation record. The activation record contains a pointer to a separate process which in turn points to all the functions that are in f 's qualifying list. This process has the form of a program: that is a list of pointers to closures. The first instruction in a program process points to the first and last entries in the list of closures, while the remaining instructions point to one closure each and contain the name of the function which the closure represents. Functions and relations are known by names as well as addresses for reasons that will be explained later. The data structure which represents the

program is:

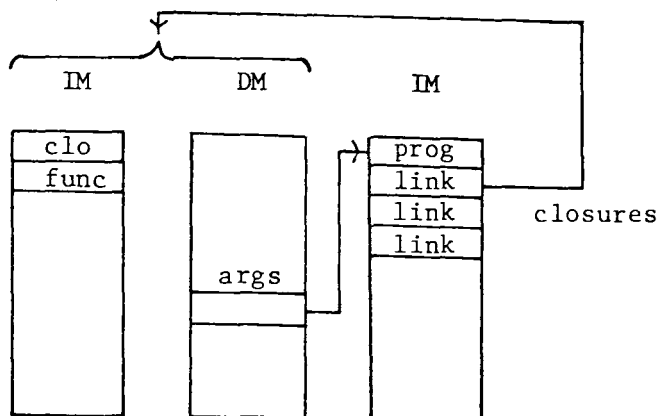


Figure 9.2: Structure of a closure.

Since the environment will be recursive, the closure must appear in its own qualifying list, as shown in Figure 9.2.

In addition, if nested qualifications are used, as in the example below, then the structure of the closures must reflect the nesting:

```

fun = λx.if x=1 then f else g
      where
        f = λy.if y=0 then 1 else f.g 1
        g = λy.h(y-1)
              where
                h = λy.y*y
    
```

The closure for fun will have the format:

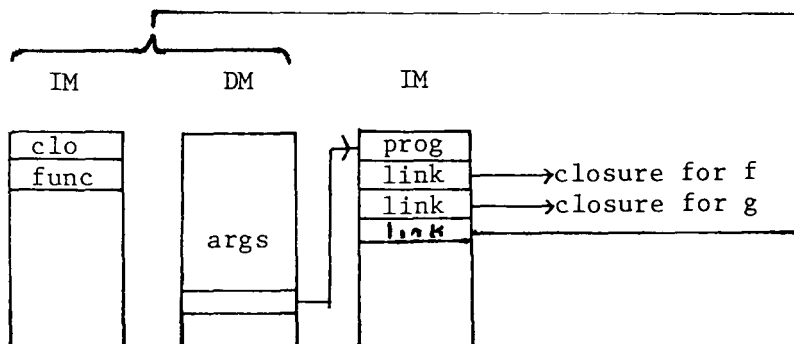


Figure 9.3: Closure for qualified function.

This has the same format as in Figure 9.3, but if the closures for f, g

and h are added then the complete structure will be that shown in Figure 9.4.

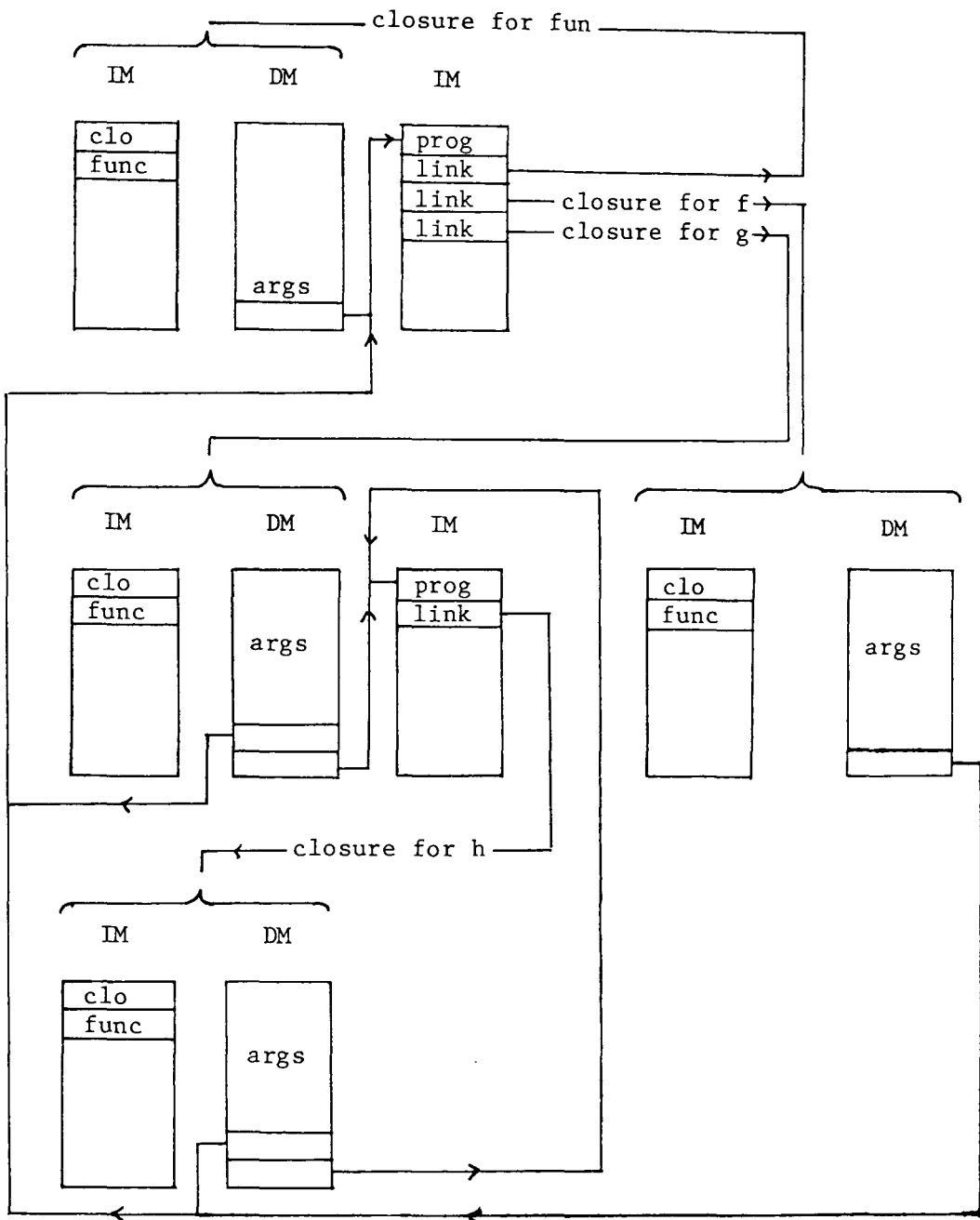


Figure 9.4: Complete structure of a qualified function.

In Figure 9.4 each qualifying function is represented by a pointer to a closure. The activation record for "fun" points to the closures for f and g. If one of the qualifying functions is itself qualified then the additional qualifying functions will give rise to additional pointers.

In the example g is itself qualified, so its activation record will contain two pointers. The first pointer will be to the list of closures of which g is a member, so that recursive calls can be made; and the second pointer to the new functions introduced by the new qualifications, h in the example above. Here the definition of h is not qualified no new functions will be introduced. The function h must, however, be able to refer to itself, and to f and g . The environment for h will therefore contain pointers to the lists containing these functions.

Each time a new set of qualifications are introduced the closures for the new functions inherit the qualification lists from the expression they qualify. This applies only to functions introduced by `whererec` (the default in the example). If the qualification is not recursive the list of qualifying functions will only include those functions introduced by the `where`. The closure for the qualified expression will therefore have only one pointer to the list of functions, as illustrated by Figure 9.5:

```
fun =  $\lambda x$ .if  $x=1$  then  $f$  else  $g$ 
      where (non recursive)
       $f = \lambda y$ . $y+1$ 
       $g = \lambda y$ . $y-1$ 
```

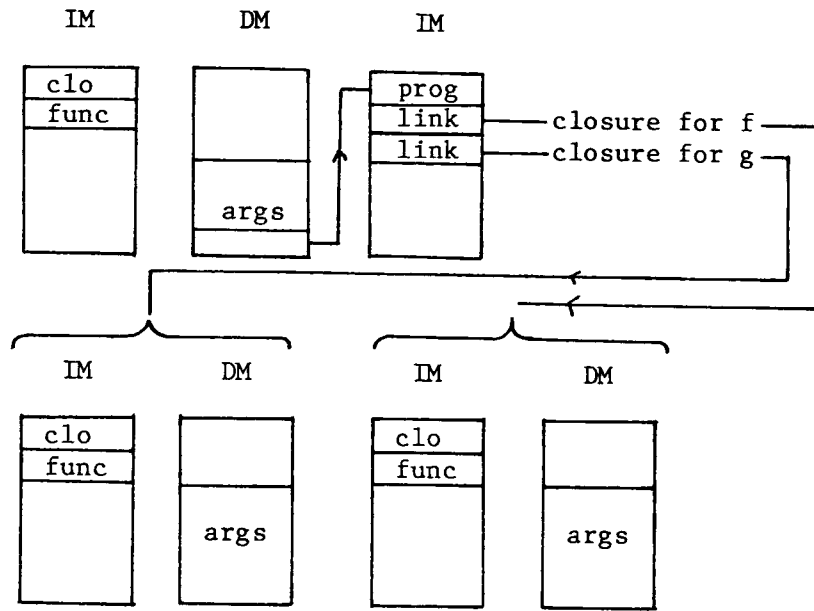


Figure 9.5: Nonrecursive qualification.

9.3.2. Logic Programs

The format of a logic program differs from that of functional programs because it is unusual for logic languages to have any notion of qualified clauses, though this may be useful in large programs. Consequently there is no notion of scope in logic, so any clause may refer to any relation. Each logic program will therefore give rise to a single process which contains pointers to each relation in the program. The program process will have the same format as the corresponding processes of a functional program, namely a prog instruction followed by a sequence of link instructions each of which refers to a single relation. Each relation is represented by a single process which refers to the relation's clauses. The relation process consists of a clo instruction followed by a single rel instruction which are followed by a sequence of rlink (relation link) instructions, each of which points to the closure

for a clause.

A clause is represented by a single process, the IM section of which holds the code for the clause and the DM section holds the clause's skeleton activation record. The code for the clause starts with a "clause" instruction, which is responsible for controlling the execution of the goals in the clause. Each argument of the clause instruction points to one goal in the clause body. The skeleton activation record holds all constant formal parameters and a pointer to program process.

Consider the program:

```
r(...) :- ...  
r(...) :- ...  
  
s(...) :- ...  
s(...) :- ...
```

The data structure which represents the program is:

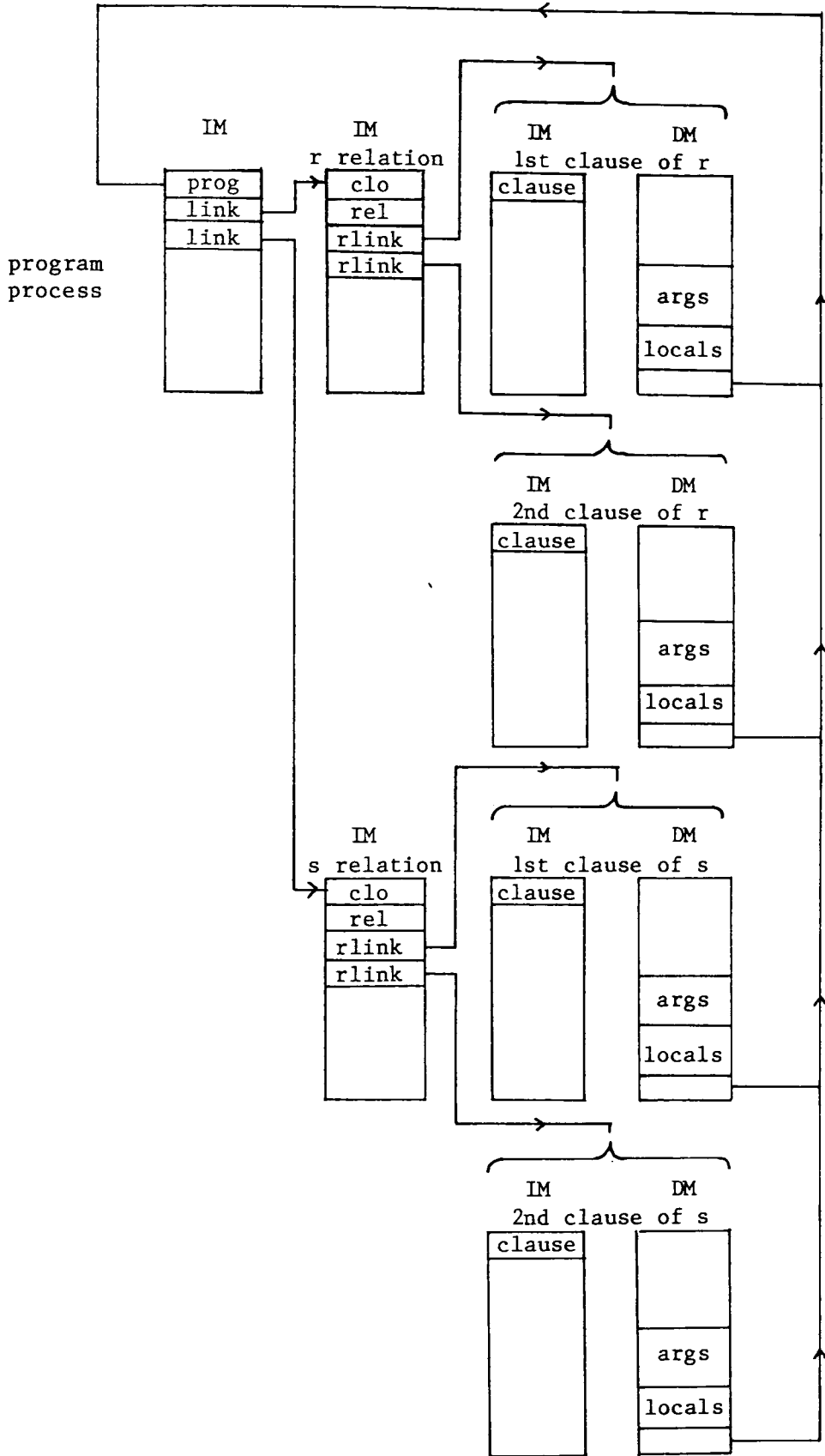


Figure 9.6: Structure of a logic program.

9.4. Program Execution

Both functional and logic languages base the execution of a program around calling functions or relations. These two operations are described in this section. The mechanisms employed for program execution were chosen for their simplicity rather than their efficiency.

9.4.1. Demand Forwarding

In order to simplify the implementation of function and relation calls a slightly modified form of demand propagation is introduced, called demand forwarding. Normally when an instruction receives a demand it will propagate its own demand in order to obtain the data it requires. The instruction will then satisfy the original demand by sending its own result to the source of that demand. When using demand forwarding, the demand received by an instruction is sent unchanged to other instructions pointed to by arguments of type "forward". As a result the destination instruction receives the demand as if it has come directly from the original source. The destination is unaware that the demand has passed through any other instructions to reach it. Only the final destination will satisfy the demand by sending its result to the original source of the demand.

9.4.2. Parameter Passing

Both functional and logic languages are based around the notion of calling functions and relations. Thus an efficient implementation of either type of language must pay particular attention to this topic. The following two sections describe two important aspects of functions and relation calls: the way calls will be carried out, and parameter

passing.

Functional Parameters

Function parameters must be suspended as long as possible in order to provide the semantics of lazy evaluation. When a closure is evaluated it must be replaced by its result; the closure is therefore a recipe.

Logic Parameters

For compatibility with functional languages logic parameters will be implemented using the copying pure code technique because functional languages create concrete copies of all structures they need. If the same technique is applied to logic it will provide a unified scheme for the combined architecture.

9.4.3. Calling Functions and Relations

This section describes how functions and relations may be called in the new architecture.

Representing Names

One important aspect of function and relation calls in the combined architecture is the use of names. Each function or relation has a unique name which is represented as an integer. It is the responsibility of the compiler to ensure that each symbolic representation of a name in the source of a program is given a unique integer to represent it. The reasons for this are due to some features that may be desirable in a hybrid language, and which are described later.

Calling a Function

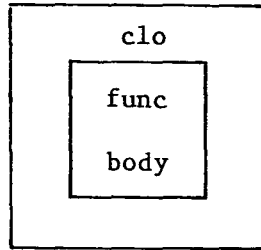
The task of calling a function is carried out by the call instruction which has the arguments:

- 1) the name of the function or a pointer to its closure
- 2) The actual parameter of the function (each call only supplies one parameter)

A call instruction may only carry out a function call when the function is represented by a closure. Therefore if the first argument of a call is a name, the name must be transformed to a closure. The call instruction transforms a name into a closure by propagating a demand to the named function. The function will then build its own closure and return the closure's address to the call instruction, which will apply the closure to the argument. The generation of the closure is carried out by a "clo" instruction which is always the first instruction of a function.

When a call instruction has a closure as its first argument it will propagate a demand to this closure. The first instruction of a closure will usually be a "func" instruction; which carries out argument binding. If a closure expects no arguments its first instruction will be the first instruction of the function's body.

A function is represented by two, conceptually nested, closures. The outer closure is used when a function is referred to by name, and the inner one when it is referred to by a direct pointer. The outer closure contains only the clo instruction, while the inner one contains the func instruction and the function's body:



When a call instruction refers to a function by name, it refers to the outer closure, when the outer closure receives a demand, it returns the inner closure as its result. When the inner closure receives a demand it returns the function's result in the normal way. Both phases of a call will now be described in more detail, first the dereferencing of a name.

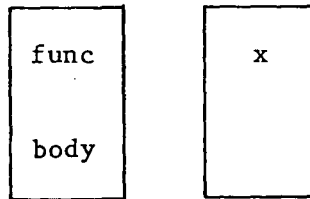
The name which appears in a call instruction has two components: the name of the function itself, and the identifier of the process which holds the list of all the functions in the program. This name may be held in the calling function's activation record if required. When a call instruction issues a demand for the closure to which a name refers, the demand arrives at the prog instruction at the head of the program. The demand contains the name of the function to be dereferenced. The prog instruction will forward the demand to first link instruction in the program (see Figure 9.4). Each link instruction contains the name of the function to which it refers. If the name in the link matches the one in the demand, the demand is forwarded to the function. If the names do not match the demand is forwarded to the next link in the program.

When a demand arrives at a function it is received by the `clo` instruction. The `clo` instruction returns the inner closure of the function as its result. The closure has two components, the process identifier of the process containing the definition of the function, the one which holds the skeleton activation record in DM, and the the address of the `func` instruction.

The second phase of a function call involves propagating a demand to the inner closure. When the demand is received by the `func` instruction, it takes a copy of the activation record belonging to closure and binds the argument into the copy. For example if the call was

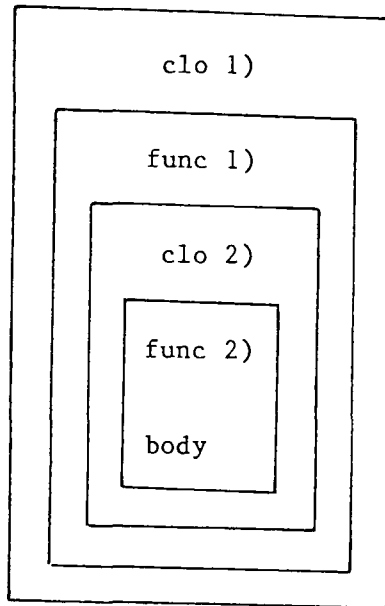
`call f x`

the closure will now have the form:

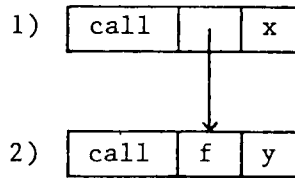


The `func` instruction will propagate a demand to the function body after the argument has been bound to obtain the result of the function. This result will be returned to the call.

Multi argument functions are constructed by nesting closures deeper. For example a function with two arguments has the form:



The call of such a function will have the form:



Call 1) propagates a demand to call 2) to obtain the closure to apply to x. Call 2) produces the closure by applying f to y. Call 2) must first generate the closure of f, which it does by propagating a demand to clo 1). The closure returned to call 2) points to the func 1) instruction. Call 2) now applies this closure to y by propagating a demand to the closure. The result of which is applied to x. The closure is created by clo 2) and contain the activation record constructed by func 1); the one which contains the value of y. This closure is returned to call 1) as the result of its demand. Now call 1) propagates a demand to the func 2), which binds x and produces the final result.

Qualifying functions will inherit values from the functions they qualify. For example in the function below both g and h inherit values from f.

```
f = λx.λy.if x=g then g else h
      where
      g = z-1
      h = λy.f(x+y)
```

The values of x and y are inherited from the qualified function when the closures for g and h are created. This will occur when they are called from the qualified expression or when they are to be returned as the result of the qualified expression. In both cases a demand is propagated to the clo instruction at the head of the qualifying function. The clo instruction will in this case generate a new closure and place the inherited values in it. The closure is then either called directly by the qualified expression, or passed out as the qualified expression's result and called later.

Executing a Function Body

When executing a function body, the func instruction propagates a demand to the body of the function. The instruction which receives the demand will propagate its own demands, and so on. Eventually all demands will be satisfied and the result of the body returned to the func instruction which returns the result to the caller.

Lazy Evaluation

To implement lazy evaluation one must be able to propagate a demand to a piece of code and have the code overwritten by its result. This is easily accomplished if the code to be reduced is in the same process as the instruction which requires the result. Each instruction simply

propagates a demand to the code, the first to arrive causes the code's reduction, and the remainder access the result directly. If the code to be reduced resides in another process, and in particular if it is a qualifying function, then demand propagation is not enough to provide lazy evaluation. The reduction of such a function will therefore require the following steps. If the qualifying function inherits values then a copy of the defining closure must be taken when the first demand arrives, as described above. The copied closure is the one which will be reduced, and it is also the closure to which the other demands must be sent. Unfortunately only the instruction which issues the first demand is aware of the process id given to the closure which is to be reduced. To overcome this difficulty all demands to the closure are propagated via a location in the environment. This location will initially hold the name of the qualifying function. Each demand propagated to a qualifying function will be issued by a call instruction, each call instruction will refer to the function via the location which holds the function's name. When the first demand is propagated to the named function, the process id of the function's closure is returned as the result. This id is stored in the location which originally held the function's name. When the function has been reduced, its result may be accessed by the other call instructions because the calls refer to the function via the location which points to the reduced closure. In this way all users of the qualifying function benefit from its reduction.

Calling a Relation

When calling a relation, the operation of the call instruction will be identical to that of the function call. If the call has a name in its first argument it will propagate a demand to the relation via the

prog and link instructions. The first instruction in the relation will be a clo instruction which will return the closure as its result. The call will then propagate a demand to the first instruction in the closure. This will be a rel instruction, which will be followed by a sequence of rlink instructions. The rel instruction will forward the demand to the first rlink instruction, which will in turn forward it to its clause, and the following rlink. This process continues until all the clauses have received a demand. By using demand forwarding the demand appears to have come directly from the call.

Execution of a Clause

When a demand is propagated to a clause it will arrive at the first instruction, which will be a clause instruction. This instruction will start a new process and copy the skeleton activation record into it. The clause instruction will then unify the formal and actual parameters, placing the appropriate values in the locations of the new activation record. The clause instruction then propagates a demand to each of the goals pointed to by its arguments in turn. If all the goals are successful, the clause instruction will reach up to the calling process and copy this process down into its own process, merging the results as it does so.

If a goal fails then the clause instruction will delete its own process and attempt to garbage collect all processes above it in the tree which have no other descendants.

If the clause is descended from a negated goal the combined architecture will implement negation in the way described in Chapter Seven.

9.5. Hybrid Programs

Since the above representations of functional and logic programs are closely related it is possible to have programs that combine both types of language. In particular hybrid programs allow one type of language to manipulate the other as data. Functional languages may therefore be used as meta languages for logic and vice versa. Languages may also be their own meta language. The next section describes simple hybrid programs which are just a mixture of functional and logic code, while the section following describes how programs involving meta operations may be encoded.

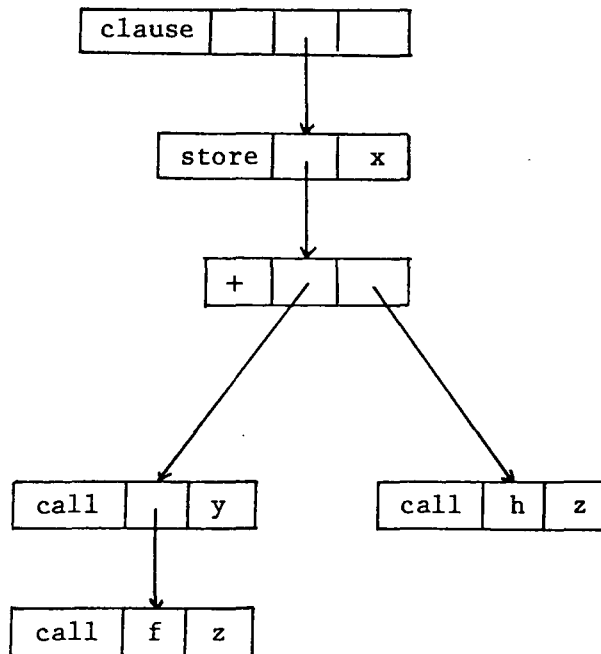
9.5.1. Simple Programs

Hybrid programs may be divided into two areas, namely calling functional programs from logic programs and vice versa. The former is the simplest and is therefore described first.

Calling Functions from Clauses

When a function is called from a clause, the function returns a single result, which will be assigned to a logic variable. The called function will be passed the values of other logic variables as parameters. The `func` and `clo` instructions of the function will operate in the way already described, the `clo` instruction will return a closure and the `func` instruction will start a process and bind the argument into it. If the function had several arguments it will be curried. All that is necessary therefore is to embed some functional code in the clause as if it were a goal, and provide an interface between the logic code and the functional code. The interface is formed by a "store" instruction will

be at the root of the functional expression which will store the expression's result in the clause's activation record. When the store instruction receives a demand from the clause instruction, it will propagate a demand to the functional expression and await its result. When this is received the result is stored in the activation record and the clause instruction informed just as for a successful goal. The code for a functional expression embedded in a clause will therefore have the form:



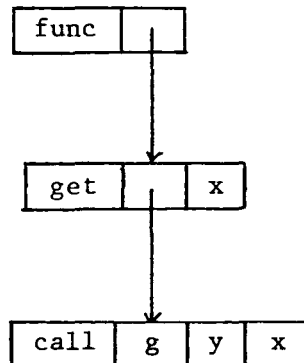
Calling Relations from Functions

In contrast, the calling of relations from functions is more complex because a single question may yield several results, each of which consists of a tuple of values. Both these concepts are alien to the functional style of language.

A logic question in a hybrid program will be compiled as a clause and held in a separate process to that of the functional expression that calls it. The call will be accomplished using a normal call instruction

formatted as a goal which will pass some variables across to act as parameters. The interface between the logic code and the functional code is provided by a "get" instruction. The get instruction is passed a demand from the functional code and then propagates its own demand to the call instruction which acts as the goal providing the value required has not already been produced. When the goal is complete it returns the result (success) to the get instruction which loads the desired result from the activation record location where the called clause placed it. The get instruction then passes the result back to the functional code as if it were its own result.

When the goal instruction receives a demand it will call the relation, which will in turn result in several clauses being executed. The execution of each clause will proceed in the usual way, generating a search tree. The leaves of the tree will pull down their ancestors, and eventually reach the functional expression which made the initial call. Each leaf process will copy down this function. Now there are a set of copies of the function each pursuing their own results. When they terminate they must copy down their caller so that the different results may still be pursued in parallel. In this way a functional program which uses logic will produce a set of results, not just one. The code for calling a goal from a function will have the form:



In order to create a parallel set of functions the func instruction must

have its operation extended to copy down its caller when appropriate. A flag in the activation record indicates if the copying is necessary.

9.5.2. Complex Programs

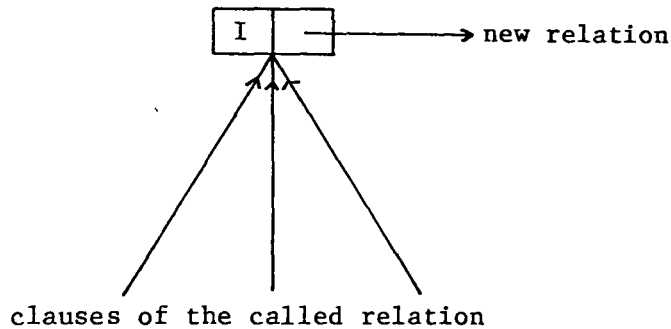
Calling functional programs from logic will never be more complex than the cases described above, because a function will only produce one result, which may easily be accommodated in the logic scheme which allows several results. In contrast calling a clause from a function produces several results, each of which are pursued by separate copies of the caller, as described above. There are situations, however, in which it will be desirable to group all results together and manipulate them as a whole. This can be achieved because the results of a goal collectively form a relation, the name of the relation will be the same as the name in the goal. Consider the example in Figure 1.2. The results of this are effectively the relation:

```
grandparent(fred,clive)
grandparent(fred,john)
```

So to gather all the results of a goal together, a new relation must be created, and the results stored in it.

The facilities described above are provided by the "all" instruction. In most respects the all instruction works in the same way as the call instruction, namely it calls the relation referred to by its first argument and holds the parameters for the call. In addition, however, it sets a flag, "a", in the activation record of the calling function to signify that the function is obeying an all instruction. It is this flag which will cause all the results to be gathered together. The call instruction also creates a new process to hold the relation which will contain the results. This process will eventually hold links to all the

assertions returned as results of the call. Having called the goal the all instruction modifies itself to an I (Identity) instruction, which points to the new process, and waits for all the results to be returned.



When a clause instruction comes to copy down an activation record it will find that the "a" flag is set. The return address of the clause will point to the I instruction which gives the process number of the relation that is to hold the results. The clause instruction will add its own process to the list already present, thereby adding its result to the list of results.

9.5.3. Parallelism

Hybrid programs introduce new possibilities for parallelism; which is the topic of this section.

Parallelism in logic can cause problems if it is not implemented cleanly. One of the advantages of following the search tree when executing logic, described in Chapter Seven was that it kept each branch of the tree independent. This meant that there was no need to pass multiple results produced by one goal along to the next.

If this simplicity is to be retained, the execution of clauses in hybrid programs must also be sequential. Consider the example:

f = X+Y

goal1(...,X)
goal2(...,Y)

The "+" operation is strict and so in a conventional functional language X and Y will be evaluated in parallel. If X and Y each produce several values this should give rise to several parallel executions of f, one for each combination of X and Y. Thus by allowing X and Y to be evaluated in parallel the problem of dealing multiple results has been re-introduced. This can be seen more clearly if the program is translated into logic:

goal(...,x),goal(...,y),add(x,y,f)

The parallel execution of the goals, in fact, corresponds to AND-parallelism. The logic scheme described in Chapter Seven only deals with OR-parallelism. In a hybrid program, therefore, the only source of parallelism must arise from the parallel execution of clauses. Notably all strict operators must be obeyed sequentially in a hybrid program. For example, X+Y could be rewritten as +XY and if the default brackets are added this will become (+ X) Y. Thus X will be evaluated first.

9.6. Hybrid Languages

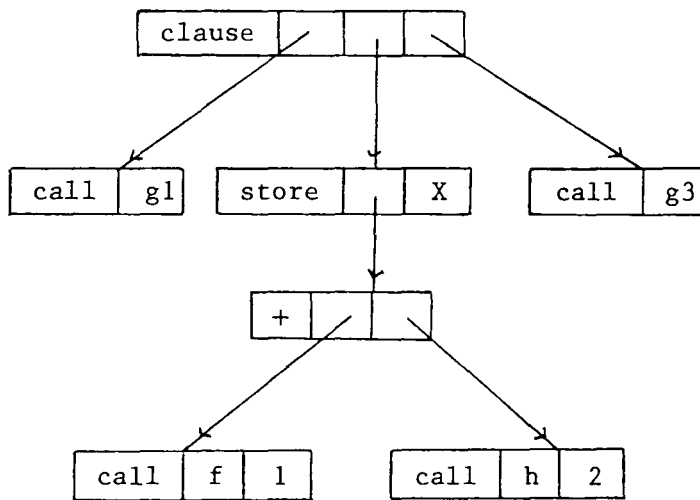
Although this is a thesis primarily concerned with computer architecture it seems desirable to describe the way the hybrid program features described above can be used to provide a hybrid language. There are two important aspects to this, firstly calling one language from another, and secondly using a program as data.

Calling Functions from Relations

Calling functions from logic could be accomplished by writing an assignment as if it were a goal:

$$g(\dots):-g1(\dots),X=+(f\ 1)(h\ 2),g3(\dots).$$

The assignment will be executed in sequence with the other goals. The complete clause giving rise to the codes:

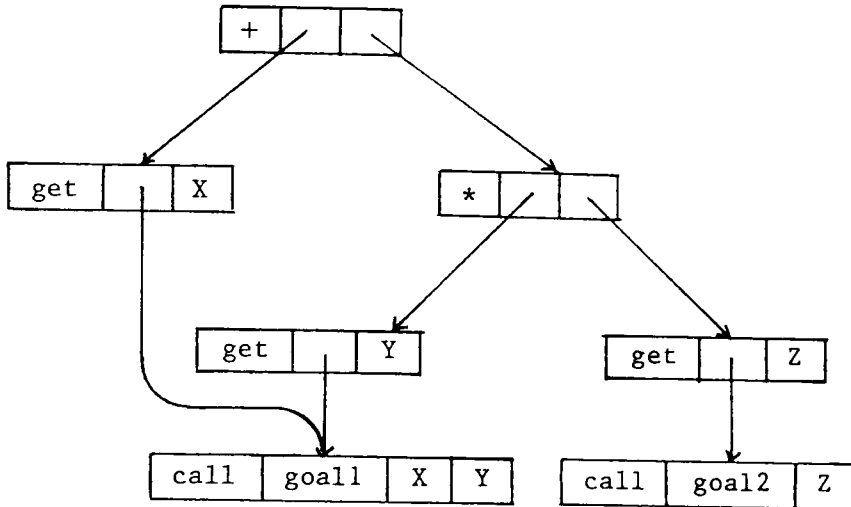


Calling Relations from Functions

The calling of relations from functions in hybrid languages is more difficult because a goal with several parameters may produce values for more than one of them as a result. The call will take the form of an auxiliary definition:

$$\begin{aligned} f &= + X (* Y Z) \\ &\text{where} \\ X, Y &= \text{goal1}(\dots, X, Y) \\ Z &= \text{goal2}(\dots, Z) \end{aligned}$$

Here a demand for X or Y implies the execution of goal1, and a demand for Z implies the execution of goal2. This program will be compiled into the code:

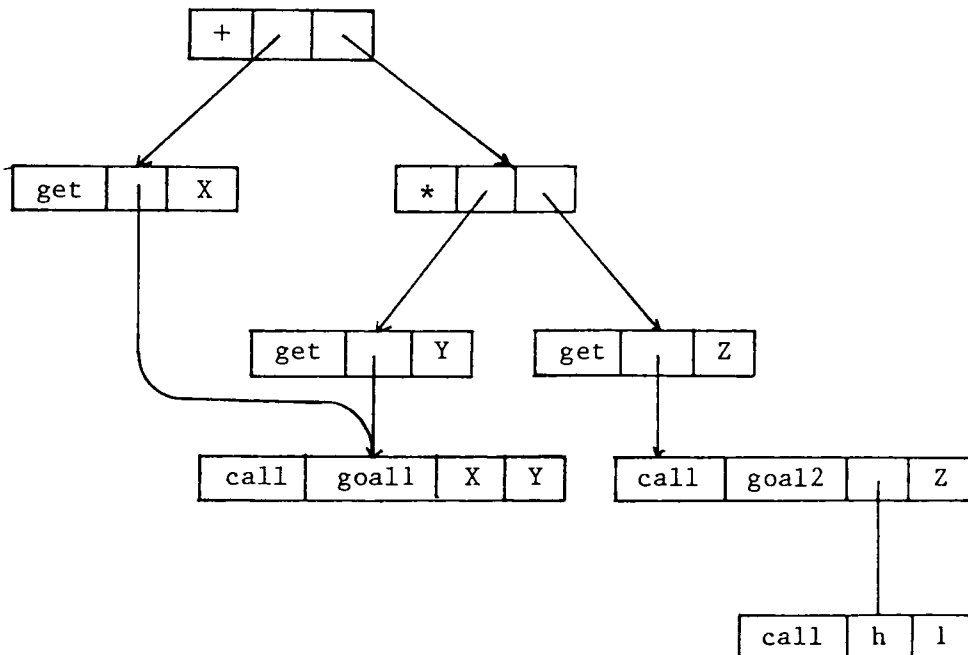


Normally the input parameters of the goal will be defined by the time the goal is obeyed. If they were not, however, the call instruction could have an argument which will demand the parameters. For example:

```

f = + X (* Y Z)
  where
    X,Y = goal1(...,X,Y)
    Z = goal2(a,...,Z)
    a = h l
  
```

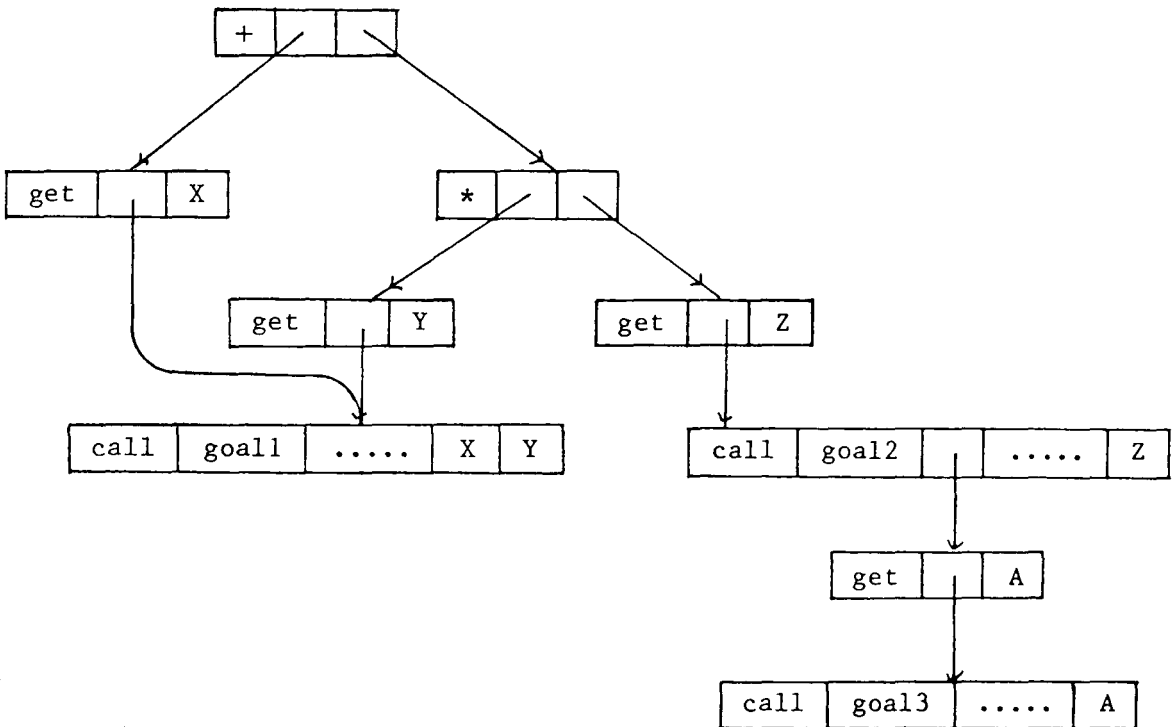
Here "a" is provided by a function, the code for which will be:



Alternately a could be provided by a clause:

f = + X (* Y Z)
where
X,Y = goal1(...,X,Y)
Z = goal2(...,A,Z)
A = goal3(...,A)

which compiles to:



9.6.1. Treating Programs as Data

Instructions are able to refer to programs because they are represented by a process holding link instructions, so instruction arguments are able to refer to programs because arguments can refer to processes. References to programs allows a program to be passed between one instruction and another, and also held in DM locations. In short, programs may be used as data. For example one could write:

f prog x = g x
where
g = h x using prog

in which the definition of h is held by prog. The name in the call

instruction will point to the location in the activation record that holds prog and gives the integer which represents the compiled form of the original name. This will be used by link instructions to forward a demand to the correct closure as already been described. The same can be done for logic:

```
goal1(X,prog) :- goal2(X) using prog
```

Thus hybrid programs can be written in the same way:

```
f Y = X
  where
    X = goal1(Y,X) using prog
goal(X,prog,Y) :- Y = f(X) using prog
```

and programs can also be passed as results.

Since a particular goal or function call is obeyed in the context of the program being treated as data, prog in the example, the context may change because prog may change. The value of prog passed as a parameter may not be the same each time f is called. This is why the "name" argument type is necessary. It can be used to identify the desired relation or function in any program that contains it.

9.7. Assessment

In this section the architecture described in this chapter is assessed both in isolation and from the point of view of language implementation.

The architecture described in this chapter uses demand propagation as its sole computational mechanism, and yet it is able to cope with a relation producing several results, a situation which the previous chapter stated was impossible. These two facts are reconciled by

returning results differently to the way used before. The original scheme passed all results back to the instruction which propagated the demand. The present scheme returns the result of a clause (success or failure) by copying down the caller, and then sending a token to the new copy of the caller. In this way there are as many copies of the caller as there are results. Each copy of the caller, therefore, only receives one result.

Unfortunately the architecture is more complex than is desirable. This arises because using activation records to implement functional languages is more complex than using combinators, and because programs may be treated as data.

Logic is mostly implemented using activation records, and therefore processes, which means that for the architecture functional languages must be implemented in the same way. Unfortunately function calls which cross process boundaries are difficult to achieve because they do not really follow the rules of reduction. Strictly speaking the call should be replaced by the called function's body, but this cannot occur if the call and the body must lie in different processes. The architecture must therefore give the effect of reduction, without actually doing it. This requires a complex interface between the called function and its caller. This situation is made worse in the combined architecture by the introduction of names.

Allowing programs to be treated as data is desirable because it introduces a limited form of higher orderedness, which is particularly useful in logic. Unfortunately it also introduces the complexity of dereferencing names. It remains to be seen if the flexibility produced by treating programs as data is worth the complexity.

The introduction of the "forward" argument type considerably simplifies the task of implementing calls, and is most helpful when dereferencing names. The argument type allows a demand propagated by one instruction to be steered to its destination by others. This relieves the originator of the demand from the work necessary to identify the destination precisely. It simply sends the demand to an instruction which decides where to forward the demand to. This process is repeated until the demand reaches the desired destination. The destination instruction sees the demand as coming directly from the originator, and is therefore unaware of the complexity of the path followed by the demand.

The new architecture differs from that described in Chapter Four in that there is no active memory, instructions reside in IM whether they are active or not. This modification allows reduction to be implemented more cleanly. Consider the example:

$$\begin{aligned} f &= * g g \\ &\text{where} \\ &g = h(+ x y) \end{aligned}$$

The qualifying function g should only be evaluated once. If the concept of the active memory had been retained g will have been moved into the active memory to be executed when g was first called. A way of allowing future callers to benefit from its reduction will therefore have to be found. This was achieved in the original architecture by allowing the reduced code to mask the code in the definition memory. This architecture avoids the problem by allowing code to be reduced in IM, the original definition of a function will therefore contain the result produced by the reduction of a constant expression, g in the example.

The structure of programs and relations used in this architecture was chosen for their simplicity. The sequential execution of links in a program will mean that a call will incur a large overhead. There is no reason, however, why a program needs to be a sequence of links. Another alternative approach would be to represent the program as a binary tree.

CHAPTER TEN

CONCLUSIONS AND FUTURE WORK

This chapter summarise the conclusions reached in the thesis and gives an indication of the directions of future work. The work described has covered three major topics: the design of a packet communication architecture, and the implementation on this architecture of functional languages, and logic languages. The packet communication architecture has been found to adequately support functional languages. Unfortunately the initial architecture provided inadequate support for logic languages. It was therefore necessary to design another architecture which supports "demand propagation with multiple results", a new computational mechanism which can support both functional and logic languages.

10.1. Conclusions

This thesis set out to develop a parallel computer architecture which was capable of supporting functional and logic languages. The initial packet communication architecture was based on the classification of Treleaven et al[68]. The authors claim that their classification describes a set of computational mechanisms which collectively support any type of computation. These claims were evaluated by attempting to implement graph reduction and logic on the packet communication architecture.

In retrospect there is at least one change that would be desirable in the emulated architecture described in Chapter Four. Demand propagation is implemented in such a way that, providing lazy evaluation is not used, the architecture behaves differently depending on the timing of demands. A set of demands arriving simultaneously at an instruction will result in the instruction being reduced once, if the same demands arrive sequentially the instruction is reduced separately so satisfy each demand. This is an undesirable property in an architecture which is intended to evaluate computational mechanisms because one would wish the mechanisms to be implemented in their purest form. It would therefore be better if the machine behaved the same no matter what the timing of demands. This means each demand should give rise to a separate execution of the instruction, each instruction will be reduced, and pass its result back to the source of its demand. Reducing an instruction once for several demands is in fact an optimisation of reducing the instruction separately because the result of the instructions will be the same in each case (assuming a pure reduction scheme). The optimisation is so obvious that the fact that it is an optimisation was overlooked when the architecture was designed. Fortunately this oversight has no effect on the results because demand propagation has only been used lazily in the work reported here.

Graph reduction was implemented on the emulated architecture without undue difficulties, although some modifications to the architecture were required. The modifications were confined to parts of the architecture which are not associated with the implementation of the computational mechanisms; it is therefore possible to conclude that the computational mechanisms implemented by the architecture are capable of supporting graph reduction, and therefore functional languages.

When logic was implemented on the same architecture severe difficulties were encountered. These centred around the inability of the computational mechanisms to support a single instruction producing several results. All the computational mechanisms are based on the premiss that an instruction only produces one result. This meant that virtually all the computational mechanisms had to be discarded. Logic was implemented using control flow, which in effect meant that a logic interpreter had to be written using control flow, rather than the computation mechanisms being used to support logic directly. Even so the storage of multiple results in memory still presented problems. The memory used in the packet communication architecture has quite a complex structure. The coding necessary to hold several results was long and cumbersome because the architecture's memory was only designed to provide the facilities required by the computational mechanisms. To overcome these difficulties each result had to be made independent of the others. This was achieved by generating a separate copy of a calling goal for each result, each copy of the caller therefore dealt with only one result. This scheme is in fact a novel way of executing a logic program using OR-parallelism.

The idea of creating a copy of a goal for each result it receives can be used as a way of implementing demand propagation with multiple results. Instead of a demanded result being returned to the caller the caller is copied down to the result. If there are several results, several copies of the caller are created. This allows functional and logic languages to be implemented using a single computational mechanism. Logic languages use demand propagation, and copy down the calling activation record, reduction uses demand propagation, and copies the result up to the caller. The use of a single mechanism allows func-

tional and logic code to be mixed freely. An architecture based on this notion was described in Chapter Nine.

One undesirable feature of the scheme is the different ways results are treated in logic and functional languages: the caller is copied down in logic, but the result copied up in reduction. It would have been far better to always copy down the caller, but in a functional language only create one copy. This would provide a more uniform way to implement functional and logic languages. Unfortunately one may not copy a caller down to the result in a functional program because reduction requires an expression to be overwritten by its result. Consequently the result may not be copied to a different point in the graph.

Another problem with the architecture described in Chapter Nine is its complexity. This is due, at least in part, to using a mixture of demand propagation and activation records. Logic uses activation records to provide the flexibility which allows any goal to produce a piece of data, and allow any goal to access it. This flexibility is necessary because it is difficult to predict which goals in a clause will produce data and which consume it. Using activation records means that combinators cannot be used to implement reduction, and so all the features which they supply automatically, such as closures, must be provided explicitly in the combined architecture. When this complexity is added to the complexity of using names, the reasons for the complexity of the architecture become clear.

Lazy evaluation in both the original architecture, and the new combined functional and logic architecture, is not implemented very cleanly. In the original architecture there is no way for a function definition to be reduced in such a way as to allow future callers of the

function to benefit from its reduction. The reduction is carried out in AM, but all the callers refer to the definition in DM. The combined functional and logic architecture implements lazy evaluation completely, but not in a very elegant way. The inelegance arises because both architectures rely on demand propagation as their control mechanism, and use a reference data mechanism. This combination is not sufficient, however, to determine all the operations the architecture must carry out to implement lazyness. For example any constant section of a function must be reduced in the function definition, while non-constant sections of code must be executed in the copy of the code created by the function's application. Any architecture which implements lazy evaluation must have some mechanism which determines when a section of code is to be copied before it is reduced, and when it should be reduced in its definition. It is not enough simply to create a new process for each invocation of a function. Combinators provide such a strategy implicitly by only copying those sections of a function's definition which contains the bound variable. Had combinators not provided this feature it would have been necessary to provide it explicitly in the implementation of functional languages described in Chapter Six. One may therefore conclude that to implement functional languages with demand propagation one would wish to use a scheme similar to that of combinators.

The work reported in the thesis can, with some justification, claim to have made some progress towards a simpler way of implementing OR parallelism, and to providing a unified way of implementing functional and logic languages; it cannot, however, claim to have solved the problem completely. To solve the problem completely a way must be found to avoid implement^{ing} logic using activation records. This means identifying the producers and consumers of data within a clause. If activation

records are no longer required, then processes are no longer required, so the complexity of implementing reduction across process boundaries is removed. This may allow combinators to be used, perhaps in a modified form, to implement logic. If this is the case then all the features provided automatically by the combinators will no longer have to be provided explicitly, thereby simplifying the architecture.

The remainder of this chapter is devoted to explanations of possible ways of achieving the above objectives.

10.2. AND-Parallelism

AND-parallelism¹⁵ is one aspect of the implementation of logic which is difficult to accomplish efficiently unless one knows which goals in a clause produce data, and which consume it. A solution to the problems of AND-parallelism may provide a solution to the problems described above.

AND-parallelism allows the goals of a clause to be obeyed in parallel, but it is complex to implement because all goals must agree on the values for each shared variable. AND parallelism also allows relations to be completely flexible. Consider the example below, if the goals of a clause are obeyed sequentially, the clause can only be used in a call which provides values for the parameters A and B.

$$g(A,B,C):-A>B,g_1(A,B,C).$$

If either A or B were undefined (i.e. have no values) in a call of g then the comparison of A and B will cause the execution of the program to stop, even if g_1 is able to provide values for A and B given C. In an AND-parallel scheme the execution of the comparison will be suspended until g_1 can produce values for A and B. A sequential execution of the

goals forces the producer/consumer relationship between the goals to follow a predefined pattern, which in turn restricts the ways a relation can be used, since a relation may not be called in a way which requires an incompatible producer/consumer structure. Another problem that can arise from the sequential execution of goals within a clause is that relations may produce an infinite number of redundant answers, instead of the intended ones. This results from calling a relation with insufficient defined values. For example `concat(1,X,Y)` will produce all pairs of lists such that Y contains 1 as its first element and X is any list at all. If this goal is part of a clause, and the order of the goals in the clause is changed, it may be possible to avoid this situation by allowing other goals to produce values for X and Y before `concat` is called.

The new implementation scheme for logic introduced in Chapter Seven only allows a clause to be obeyed sequentially, and so the scheme will suffer from both the problems described above. One direction that future work could take is to attempt to provide some of the flexibility of AND-parallelism. This can be achieved using modes to indicate if the relation is able to produce a result for a given call. The concept of a mode is used in Edinburgh Prolog[74] where it specifies which actual parameters must be defined in a call and which must not. For example the mode `(+,+,-)` means that the first two parameters supply values to the relation, and the last is the result received from it. In Edinburgh Prolog the programmer must annotate his code to indicate the modes a clause is able to handle, and also annotate the goals within the clause to show which produces or consumes data. This forces the programmer to consider the way his program is going to execute, which seems undesirable. A better solution is to derive the modes automatically at

compile time, an novel algorithm for which is described in the next section. For each mode for which clause may be executed, the new interpreter for logic will have a particular order of goals to follow which will avoid the problems described earlier.

Mode Derivation

The mode derivation algorithm proceeds by finding all acceptable modes for the assertions of a relation and then tests them against those clauses of the relation that have bodies to see if the clauses are well behaved for the chosen mode. An acceptable mode is one for which the clause is able to return a value for all undefined parameters. The algorithm starts by making a list of acceptable modes for all the assertions in the relation. For example consider the following assertion and the list of all possible modes:

r(a,X,X).

- 1) +,+,+
- 2) -,+,+
- 3) +,-,+
- 4) -,-,+
- 5) +,+,-
- 6) -,+,-
- 7) +,-,-
- 8) -,-,-

The formal parameter "a" is a constant while X is a variable. An acceptable mode may have either "a" as + or - because a can either check an input value or supply a result. The remaining two arguments will force the parameters supplied to be the same. If two constants are passed they must be equal. If one value and one variable are passed, the unification algorithm will assign the value to the variable. If two variables are passed, the unification algorithm will make one point to the other; effectively making them the same variable for the remainder

of the execution of the calling clause. The unification does not however provide a value for X. Any mode which does not provide a value in such a situation is unacceptable because it does not allow the program to progress towards a solution. This means that modes 7) and 8) are unacceptable, leaving modes 1) to 6).

Another example could be the assertion:

$r(a,X,Y).$

where "a" is a constant, and X and Y are variables. The only modes which are acceptable are:

- 1) +,+,+
- 2) -,+,+

All the others leave either X or Y undefined after the call because they give these variables a mode of "-".

This process is repeated for all the assertions of the relation. A list of modes which are acceptable to all assertions is then constructed. If modes acceptable to only some assertions were included it will mean that the relation will produce some solutions, and then start to behave badly.

The modes selected by the above algorithm must be tested against the clauses of the relation with bodies. For each mode derived above an ordering of the goals within each clause must be found. Each goal in the clause may only use the acceptable modes of the called relations. If no such order for the goals in a clause can be found the mode being check is deleted from the list. Finding an ordering for the goals in a clause means knowing the acceptable modes of all the called relations, which in turn introduces some difficulties when deriving modes for recursive relations. The algorithm needs to know the acceptable modes

for a recursive relation in order to derive the modes for the same relation. Consider the following clause:

$$g(A,B) :- h(1,C),g(A,C),f(C,B)$$

If the clause is called with mode `-, -` the recursive call is made with mode `-, +` because `h` delivers the value of `C`. Thus to know if mode `-, -` is acceptable to the `g` relation, one must know if mode `-, +` is acceptable to the `g`. If we assume that mode `-, +` is acceptable then mode `-, -` is also acceptable. The mode derivation algorithm will therefore move on to the other clauses in the `g` relation. Now suppose that the algorithm discovers that one of the other clauses finds mode `-, +` unacceptable. This means that the mode `-, -` is no longer acceptable because the clause above can longer make its recursive call. In some circumstances it may be possible to re-order the goals of to change the mode of the recursive call, but in the example this is not possible. The removal of mode `-, -` from the list of acceptable modes may mean that other clauses in the relation can no longer make their recursive calls because they use mode `-, -`, so more modes will be deleted. Mode derivation for recursive relations will in the most general cases lead to a significant overhead. The situation may be illustrated by the table:

	acceptable mode			
	-, -	-, +	+, -	+, +
recursive goal	-, -	*, -	. , -	. , +
mode	-, +	+, -	+, +	. , +

Figure 10.1: Mode table for a clause.

The columns contain the list of modes acceptable to the relation, and

the rows indicate the modes for the recursive call in the clause g. Each acceptable mode gives rise to a particular mode for the recursive call, indicated by character in the appropriate square. The square marked "*" is the one for the case described above. The other modes form the complete picture of the way the modes of the recursive call in the clause g are related to the acceptable modes of the relation.

Recursion therefore introduces severe difficulties which may result in the repeated re-ordering of clause bodies to take account of modes which have been deleted. The problem arises because recursion relates one acceptable mode to another: clauses called with one mode give rise to a recursive call with another mode. Thus when one mode is deleted it may result in other modes being deleted because of the relationship between modes created by recursion. The deleted mode is the one used by the recursive call, which leads to the deletion of the mode used when the recursive clause is called. The problem is worse if there are two recursive goals in a clause, because one acceptable mode will probably be related to two others. Mutual recursion will relate the acceptable modes of several relations.

There is one type of recursion which will cause no problems. If each acceptable mode for a relation is supplied to a recursive clause, and the recursive goal in the clause uses the same mode, then the table will have a series of dots along the leading diagonal:

	acceptable mode				
	+,+,-	+,+,-	+,-,+	-,+ ,+	+,+,+
	-,+,-	.			
recursive goal	+,+,-	.			
mode	+,+,-				
	+,-,+		.		
	-,+ ,+			.	
	+,+,+				.

Now suppose that mode +,+,+ is used by a recursive clause, but the mode is later found to be unacceptable to another clause in the relation. The mode +,+,+ is therefore removed from the list of acceptable modes. The recursion described by the above table does not cause any difficulties because the mode which the recursion dictates should be removed is the same as the one which has been removed anyway. The difficulties only arise if recursion dictates that a different modes must be deleted. Fortunately most recursions are of the simple type illustrated by the table above, so recursion may not cause the overheads described above in most cases.

An example which illustrates the use of the algorithm described above is:

- 1) delete(H,cons(H,T),T).
- 2) delete(X,cons(H,T),cons(H,DX)):-delete(X,T,DX).

This relation deletes the first occurrence of parameter one from parameter two and returns the result in parameter three. If the first parameter is not contained in parameter two the relation fails.

The acceptable modes for clause 1) are:

- 1) +,+,+
- 2) -,+,+
- 3) +,-,+
- 4) +,+,-
- 5) -,+,-

The following modes are omitted because the variables H and T are shared between parameters, and may not therefore have both occurrences undefined:

- 6) -,-,+
- 7) +,-,-
- 8) -,-,-

The selected modes must now be checked against clause 2). Further modes will only be deleted if the goal of clause 2) tries to make a recursive call with an unacceptable mode. All the modes are in fact satisfactory because all the recursive calls are made with the same mode as the call on the relation; recursion will not therefore cause any problems. The table for the recursive will have dots along the leading diagonal.

Does delete behave well for all the acceptable modes? A call with no unknowns will behave well because it simply checks to see if X has been deleted. Those with one unknown will take the two defined values and return the third, there is only one possible value for the result in each case. There is only one mode with two unknowns, namely mode 5). This will produce pairs of results, one pair for each member of the list supplied as the second parameter. Each pair will consist of one member of this list, and a copy of the list with the member deleted. The number of such pairs will be equal to the number of elements in the list. The relation is, therefore, well behaved for all acceptable modes.

In the delete example the modes which were removed all have bad behaviours. Mode 8) will obviously produce an infinite number of results. Mode 7) will produce all pairs of lists whose only difference is the membership of the first parameter. Mode 6) will produce an infinite number of results. It asks for any atom which when deleted from any list produces the specified list. The algorithm has therefore successfully identified those modes for which the delete relation is able to produce a result.

As a final example consider the member relation which returns true if parameter one is a member of the list passed as the second parameter:

```
1) member(H,cons(H,T))
2) member(X,cons(H,T)):-member(X,T).
```

The complete list of modes will be:

```
1) +,+
2) -,+
3) +,-
4) -,-
```

The last is deleted by the assertion because the parameters of clause 1) share H. Mode 3) is deleted because it will not supply a value for H. The resulting list has only those modes for which the relation is well behaved.

```
1) +,+
2) -,+
```

Mode 1) checks to see if parameter one is a member of parameter two. Mode 2) produces a set of results which contains all the elements of the second parameter. The deleted modes both behave badly. Mode 3) will produce an infinite number of of lists which had the first parameter as a member. Lastly mode 4) will obviously produce an infinite number of results.

The mode derivation algorithm described above is somewhat simplified because it assumes that a parameter is either completely undefined or completely defined. If the parameter is a structure it may contain some defined variables and some undefined ones. The algorithm must therefore be extended to apply to variables contained by parameters, instead of just the complete parameters.

The ideas of mode derivation allow the producers and consumers of data to be identified, and therefore go some way to solving some of the problems associated with the architecture described in Chapter Nine. Since the modes of all goals in a clause are now known it is even possible to implement logic using data flow. It is also possible to use combinators, which is the topic of the next section.

10.3. Combinators in Logic

Combinators have been used by Turner[69] to implement functional languages. If the mode derivation algorithm outlined above is practical it may be possible to use combinators to implement logic.

Each mode defines input and output parameters, so each mode restricts the relation in such a way as to turn it into a function. If there is a different version of a clause for each mode then combinators can be used to substitute the arguments into the clause body. Within the body of the clause the modes used by each goal are also known. This allows the goal which produces the value for a particular variable to be identified. Combinators can now be used to distribute the result to the other goals of the clause.

The combinators used to represent a clause will be the same as those used for functions but with one addition; the **R** combinator which is used to return a clause's result.

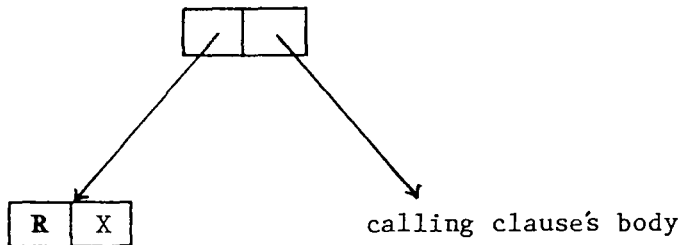
Returning From a Clause

The results of a clause are returned by the **R** (Result) combinator, of which there will be one for each result the clause returns. There are in fact two versions of the combinator: **R** and **R'**. The reduction rules for **R** and **R'** are:

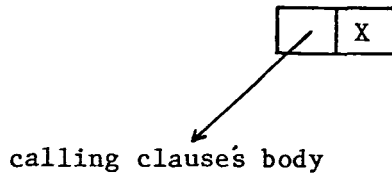
$$\begin{aligned} \mathbf{R} X E &\Rightarrow E X \\ \mathbf{R}' X E_1 E_2 &\Rightarrow E_1 (E_2 X) \end{aligned}$$

where E and E_2 represent the body of the calling clause, and E_1 is a combinator expression. The variable X denotes the clause's result. The **R** combinator is used to return a single result, and **R'** if there are several. For example if there are three results the first two will be returned using **R'** and the last be **R**. Both combinators always appear at the end of a clause and are applied to the result, returning it to the calling clause.

The graph for the reduction of **R** will be:



After the reduction of **R** has been performed the result will be:



As has already been mentioned the calling clause's body will contain the combinators necessary to distribute the result to those goals that require it. Applying the body to the result causes this distribution to take place.

The **R** combinator is in fact the graph reduction equivalent of the endc instruction (introduced in Chapter Eight), and as such must copy down the caller. In fact this occurs automatically because **S** and related combinators will peel off a copy of the caller's body as the substitution of the result is carried out.

The **R'** combinator is used if several results are to be returned from a clause, each result is returned using a separate **R'** combinator, except that the last one will be returned using an **R** combinator. If two results, **X** and **Y**, are to be returned the expression which will carry out the task will be:

$$\mathbf{R' X (R Y) E}$$

where **E** is the body of the calling clause. When **R'** is reduced the expression becomes:

$$\mathbf{R Y (E X)}$$

and after the reduction of **R** it becomes:

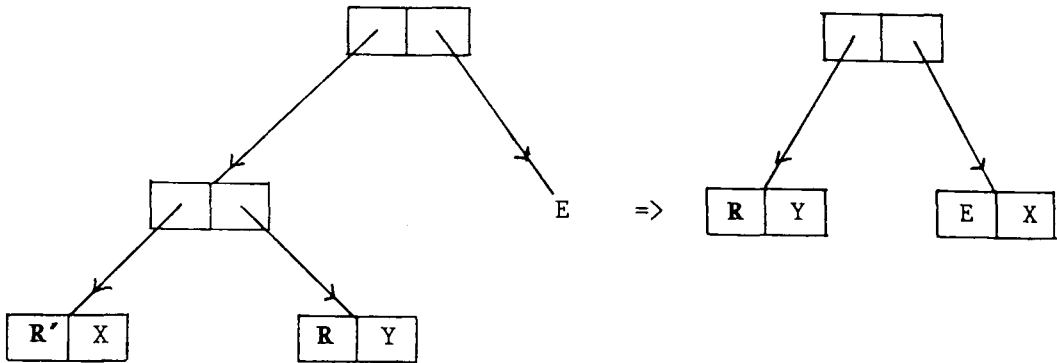
$$\mathbf{E X Y}$$

Thus **E** will be applied to both the result, and both results will

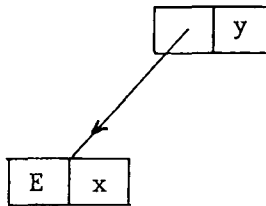
therefore be substituted into E. If the reduction of the expression

$R' X (R Y) E$

is drawn as a graph it will have the form:



The reduction of R will give the graph:



The introduction of the R combinator therefore allows the results of one clause to be returned to another, and the result substituted into the calling clause's body.

Abstraction

Each clause in a program must be compiled into combinators, this compilation is carried out by abstracting variables from the body of the clause. The abstraction process starts by dividing a clause into operator/operand pairs. This is achieved by taking the first goal as the operator and the remainder of the clause as the operand. The

operand is then divided in the same way giving the structure:

$$g_1, g_2, g_3 \Rightarrow g_1(g_2 g_3)$$

Each goal is also divided into operator/operand pairs, for example:

$$g(a,b,c) \Rightarrow ((g a) b) c$$

Having divided the clause into operator/operand pairs the clause may be compiled into combinators. The compilation is carried out by abstracting variables and has three sections. The simplest abstraction is that of input parameters which is explained first.

Each input parameter (one with mode + in a head) is abstracted from the body in turn, starting with the leftmost parameter. For example, given the clause below, the result of abstracting X will be (only S, K and I are used):

```
grandparent(X,T) :- parent(X,Z),parent(Z,T)
                    => (parent X Z)(parent Z Y)
[X] grandparent(X,Y) => S(S parent (K Z))(K (parent Z Y))
```

The same process will be repeated for Y if it also has a mode of +.

Another section of the compilation process is the abstraction of local variables. Any goal which produces a result will give rise to an abstraction of that result from the goals to the producer's left. The combinators introduced will be the ones which distributed the result throughout the clause. The local variables of the clause are abstracted by moving through the clause looking for a goal which has a mode of + for a local variable, and then abstracting the corresponding variable from the rest of the clause body. All consumers of the value will appear to the left of the producer in the clause body. For example, if the first goal of the grandparent clause produces a value for Z, the result of abstracting this variable from the clause body will be the

expression:

[Z] grandparent(X,Y) => parent X Z (S parent (K Y))

The execution of the first goal will produce a value to which the expression on the right of the goal is applied. The means by which this is achieved depends on of the way results of clauses are returned, which is the subject of a later section.

The final section of compilation to be described is the abstraction of results. The results of a clause are abstracted in much the same way as local variables. A result variable is selected from the head (the variable will have a mode of -) and the body of the clause is searched for the goal which has a mode of + for the same variable. This variable is then abstracted from the remainder of the clause so any goals *will receive a copy* which use the result. An R combinator is added to the end of the clause as if it were a goal, and the combinators generated so that R will be applied to the result. In this way R is applied to the result, and the result is returned to the calling clause. If X is the result of the grandparent clause its abstraction will produce the expression:

[X] grandparent(X,Y) => parent X Z (S (K (parent Z Y)) R)

The sections of the abstraction algorithm are not performed in the order in which they were described, they must be performed in the reverse order to substitutions. The first substitution is that of the input parameters, so these must be abstracted last. The order of abstractions of local variables and results depends on the position in the clause of the goals which produce them. The abstraction algorithm moves throughout the clause looking for goals with a mode of + for any variable. When it finds such a variable it decides if it is a local variable or a result, and performs the appropriate abstraction.

Calling a Relation

When a relation is called it will only be called by a goal using one of the acceptable modes. Suppose a goal $g(A,B,C)$ uses a mode of $+,+,-$, then the call will have the form:

$$g_{++-} A B$$

The subscript of the goal denotes the mode it uses. Each clause in the g relation will be represented by a set of different versions, one version for each acceptable mode. In the example the particular version of g for mode $+,+,-$ is called and passed the input parameters: A and B . The output parameter, C , is not passed because the combinators render it redundant, C need not be substituted into the clause, and is not required to pass the result out. The substitution of the parameters A and B will be not occur throughout the clauses of g , but will only be carried out as far as for the first goal of each. If the goal is successful the substitution will be pushed further down the each clause. At each stage any results returned by a goal are substituted into the rest of the clause's body. Eventually all the goals will have been obeyed, and so the result of the clause must be returned.

Assessment

The use of combinators to implement logic will reduce the complexity of a combined architecture compared to that described in Chapter Nine. Unfortunately using combinators has the same drawbacks as for functional languages, namely that the body of clause is copied for each use. Thus there will be a copy for each branch of the tree. The logic scheme proposed in Chapter Seven saved space by re-using activation records, but when using combinators there are no activation records.

However when one clause finishes and returns its result to the caller, the body of the called clause is discarded and the body of the caller copied down. The garbage collector will have reclaimed the cells of the discarded body, and will therefore allow them to be re-used to construct the body which is now being copied down. The space efficiency is still present, therefore, but now with the overhead of a garbage collector. Clearly additional work on this topic is needed to demonstrate that the ideas expressed above are practical, and attempt to simplify the abstraction algorithm for logic, perhaps by introducing more appropriate combinators.

10.4. Hybrid Languages

Hybrid languages offer some of the advantages of both functional and logic languages and are becoming an important research topic. The best known attempt so produce such a language has been made by Robinson and Sebert[64] when they produced LogLisp. This language allows logic to be called from Lisp, and the results returned in Lisp data structures. The results returned may be a list of all results, or just one. LogLisp does not, however, allow programs to be treated as data. It only allows a logic program to be consulted to obtain the desired results.

The logic scheme described in Chapter Seven allows programs to be treated as data, and also allows logic and functional code to be mixed freely. This permits goals to be curried in the same way as functions. For example:

```
f x y = z
      where
      z = goal(x,y,z)
```

The goal is in effect carried by enclosing it in a function, one may now write the following:

$$g = f \ l$$

The ability to pass programs as data opens the way to the use of higher order logic programs, for example:

$$\begin{aligned} f \ p = z \\ \text{where} \\ z = \text{goal}(x,y,z) \text{ using } p \end{aligned}$$

where z is produced by calling goal, the definition of which is held in the program p. This technique does not give the same power as higher order functions do in functional languages. In a functional language a function:

$$f \ g = g \ l$$

allows any other function to be supplied as an argument to f. When a function is passed as an argument its name is effectively changed to g, and is then applied to l. In a logic program the name of the goal to be called is fixed, in the example above it is "goal" so the relation in p to be called must always have the same name. This reduces the flexibility that the feature is able to provide. To achieve the power of higher order functions in logic one must make z point to a location in the activation record, and place the name of the goal to be called in that location. This will allow the function below to be written:

$$\begin{aligned} f \ \text{goal} = z \\ \text{where} \\ z = \text{goal}(x,y,z) \end{aligned}$$

where goal is now any relation. In other words one must pass relations, as well as programs, as data.

As was mentioned in Chapter Two higher orderness in logic can cause problems because one may write:

```
clause(GOAL,X) :- GOAL(X)
```

which asks which goal is true of X because the value of GOAL may be left undefined when clause is called. One cannot write this using a functional notation because the value of goal must be defined before it is used, as is the case with all objects used in functions. Thus so long as higher order relations are only used within functions the problem outlined in Chapter Two will not arise. Only allowing relations to be passed to functions does, however, limit the usefulness of the technique.

Any future work based on the ideas expressed above must devise an elegant set of features for a hybrid language which combine the useful features of functional and logic languages.

10.5. Hybrid Computer Architecture

Finally, we will discuss the design of (parallel) hybrid computer architectures. Such designs are attractive because they could efficiently support functional, logic and hybrid languages, all of which are likely to be important topics in future research. A computer architecture based on the ideas described in Chapter Nine, but now incorporating more than one processor, could be viewed as complementing other packet communication architectures such as the Manchester Data Flow computer, and the ALICE reduction machine being produced at Imperial College London.

The Manchester computer supports a pure data driven form of instruction execution, which although ideal for certain types of language (i.e. Single-Assignment) may present problems for logic, as was mentioned at the end of Chapter Seven. In contrast, ALICE is more general-purpose because it incorporates the possibility of controlling program execution using control driven mechanisms. The advantage of a parallel computer architecture based on the scheme described in Chapter Nine, should be its simplicity. The scheme is able to support both functional and logic languages with one mechanism, whereas ALICE may be viewed as needing two mechanisms. Such a hybrid architecture could reasonably claim to be a general-purpose alternative to Japan's so-called Fifth Generation computer.

CHAPTER ELEVEN

REFERENCES

- [1] Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time.", IFIP 77, pp.848-854 (1977).
- [2] J. Backus, "Reduction Languages and Variable Free Programming", IBM Research RJ1010 (7th April 1972).
- [3] J. Backus, "Can Programming be Liberated from the Von Neuman Style? A Functional Style and it's Algebra of Programs", CACM Vol. 21(8).
- [4] K.J. Berkling, "A Computing Machine Based on Tree Structures", IEEE Transactions on Computers (April 1971).
- [5] K.J. Berkling, "Reduction Languages for Reduction Machines", Proc. 2nd International Symposium on Computer Architecture, pp.133-140 (1975).
- [6] C. Bohm and W. Gross, "Introduction to the CUCH", in Automata Theory, ed. E.R. Cainiello, Academic Press (1966).
- [7] R.S. Boyer and J.S. Moore, "The Sharing of Structure in Theorem Proving Programs", in Machine Intelligence 7.
- [8] J.M. Brady, The Theory of Computer Science, A Programming Approach, Chapman and Hall.
- [9] M. Braynooghe, "An Interpreter for Predicate Logic Programs. Part 1.", Report CW 10, Applied Mathematics and Programming Division, Katholieke Universteit, Leuven, Belgium (1976).
- [10] W.H. Burge, Recursive Programming Techniques, Addison-Wesley.
- [11] R.M. Burstall, "Design Considerations for a Functional Programming Language", Proc. of Infotech State of the Art Conference (Copenhagen) (1977).
- [12] R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", JACM Vol. 24(1) (January 1977).
- [13] R.M. Burstall, D.B. MacQueen, and D.T. Sannella, HOPE: An Experimental Applicative Language, Dept. of Computer Science, University of Edinburgh (1980).
- [14] C. Chang and R.C. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press (1973).
- [15] A. Church, The Calculi of Lambda Conversion, Princeton University Press (1941, 1951).
- [16] A. Church and J.B. Rosser, "Some Properties of Conversion", Transactions of the American Mathematical Society Vol. 39, pp.472-482 (1936).

- [17] T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean, and A.C. Norman, "SKIM - The S, K, I Reduction Machine", in LISP-80.
- [18] K.L. Clark, F.G. McCabe, and S. Gregory, "IC-Prolog Language Features", Imperial College Report Doc 81/31 (October 1981).
- [19] J.J.W. Clark, P.J.S. Gladstone, C.D. MacLean, and A.C. Norman, "SKIM-The S K I reduction Machine", Lisp-80.
- [20] K.L. Clark and S. Gregory, A Relational Language for Parallel Programming, Imperial College, Dept. of Computing Science.
- [21] K.L. Clark, "Negation as Failure", pp. 2293-2324 in Logic and Data Bases, ed. Gallaire and Minker (1978).
- [22] K.L. Clark and S.A. Tarland, "Predicate Logic as a Language for Parallel Programming", in Logic Programming, Academic Press (1981).
- [23] W.F. Clocksin and C.S. Mellish, Programming in Logic, Springer Verlag.
- [24] J.S. Conery and D.F. Kibler, "Parallel Interpretation of Logic Programs", ACM Conference on Functional Programming (September 1981).
- [25] D. Comte and N. Hifdi, "LAU Multiprocessor: Micro-Functional Description and Technical Choices", 1st European Conference on Parallel Processing (14th February 1979).
- [26] H.B. Curry and R. Feys, Combinatory Logic, North Holland (1968).
- [27] J. Darlington and M. Reeve, "A Reduction Machine for the Parallel Evaluation of Applicative Languages", Imperial College, London (18th March 1981).
- [28] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor", Proc. 2nd Annual Symposium on Computer Architecture, pp.126-132 (January 1975).
- [29] J.B. Dennis, D.P. Misunas, and P.S. Thiagarajan, Data Flow Computer Architecture, MIT Project Mac Computational Structures Group Memo 104.
- [30] R.B.K. Dewar and E. Schonberg, Elements of SETL style.
- [31] D.P. Friedman and D.S. Wise, "Aspects of Applicative Programming for File Systems", Proc. ACM Conference for Reliable Software.
- [32] D.P. Friedman and D.S. Wise, "Aspects of Applicative Programming for Parallel Processing", IEEE Transactions on Computers Vol. c-27(4) (April 1978).
- [33] D.P. Friedman and D.S. Wise, "A Note on Conditional Expressions", CACM Vol. 21(11) (November 1978).
- [34] D.P. Friedman and D.S. Wise, "An Indeterminate Constructor for Applicative Programming", Conference Record of the 7th Annual ACM Symposium on the Principles of Programming Languages (January 1980).

- [35] Peter Henderson, Functional Programming, Prentice Hall.
- [36] P. Henderson and J.H. Morris, "A Lazy Evaluator", Proc. 3rd ACM Symposium on the Principles of Programming Languages, pp.95-103 (January 1976).
- [37] J.R. Hindley, B. Lercher, and J.P. Seldin, Introduction to Combinatory Logic, Cambridge University Press, Cambridge (1972).
- [38] C.J. Hogger, "Concurrent Logic Programming", Dept. of Civil Engineering, Imperial College, London (October 1981).
- [39] A. Horn, "On Sentences that are True of Direct Unions of Algebras", Journal of Symbolic Logic Vol. 16, pp.14-21 (1951).
- [40] R.P. Hopkins, P.W. Rautenback, and P.C. Treleaven, A Computer Supporting Data Flow, Control Flow and Updatable Memory.
- [41] S.B. Jones, "The Performance Evaluation of Interpreter Based Computer Systems", Ph.D. Thesis, University of Newcastle Upon Tyne (1981).
- [42] S.C. Kleene, Origins of Recursive Function Theory.
- [43] W.E. Kluge, Co-operating Reduction Machines, GMD Bonn.
- [44] W.E. Kluge, "The Architecture of a Reduction Language Hardware Model", GMD ISF-Report 79.03 (August 1979).
- [45] K. Furakawa, K. Nitta, and Y. Matsumoto, Prolog Interpreter Based on Concurrent Programming.
- [46] R. Kowalski, "Predicate Logic as a Programming Language", IFIPS 74, pp.569-574.
- [47] R. Kowalski, "Algorithm=Logic+Control", CACM Vol. 22(7) (July 1979).
- [48] R. Kowalski, Logic for Problem Solving, North Holland.
- [49] P.J. Landin, "The Mechanical Evaluation of Expressions", Computer Journal Vol. 6, p.308.
- [50] P.J. Landin, "The Next 700 Programming Languages", CACM Vol. 9(3) (March 1966).
- [51] P.J. Landin, "A Correspondence Between ALGOL 60 and Church's Lambda Notation, Part 1", CACM Vol. 8(2) (February 1965).
- [52] P.J. Landin, "A Correspondence Between ALGOL 60 and Church's Lambda Notation, Part 2", CACM Vol. 8(3) (March 1965).
- [53] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1", CACM Vol. 3(4), pp.184-195 (1960).

- [54] G.A. Mago, "A Cellular Computer Architecture for Functional Programming", Technical Report, University of North Carolina.
- [55] Z. Manna, Mathematical Theory of Computation, McGraw Hill.
- [56] C.S. Mellish, "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter", Research Paper 150, Dept of A.I., University of Edinburgh (1980).
- [57] N. Nilson, Problem Solving Methods in Artificial Intelligence, McGraw Hill.
- [58] E.S. Page and L.B. Wilson, Information Representation and Manipulation in a Computer, Cambridge University Press (1978).
- [59] G.H. Pollard, "Parallel Execution of Horn Clause Programs", Ph.D. Thesis, University of London.
- [60] J.A. Robinson, "Theorem Proving on the Computer", JACM Vol. 10, pp.163-174 (1963).
- [61] J.A. Robinson, "Computational Logic, The Unification Algorithm", Machine Intelligence 6, Edinburgh University Press (1971).
- [62] J.A. Robinson, Logic: Form and Function.
- [63] J.A. Robinson, "A Review of Automatic Theorem Proving Techniques", Proc. of the Symposia in Applied Maths Vol. 19, American Mathematical Society (1967).
- [64] J.A. Robinson and E.E. Sibert, LogLisp-An Alternative to Prolog, School of Computer and Information Science, Syracuse University (December 1980).
- [65] J. Schwartz, "An Introduction to the Set Theoretic Language SETL", in Computers and Mathematics with Applications, Pergeman Press.
- [66] M. Sconfinkel, "On the Building Blocks of Mathematical Logic", in From Frege to Godel, a Source Book in Mathematical Logic 1879-1931, ed. E.H. Maden, Harvard University Press.
- [67] M.R. Sleep, "Applicative Languages, Data Flow, and Pure Combinatory Code", Comcon 80.
- [68] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, "Data Driven and Demand Driven Computer Architecture", Computing Surveys Vol. 14(1) (March 1982).
- [69] D.A. Turner, "A New Implementation Technique for Applicative Languages", Software Practice and Experience Vol. 9, pp.31-49.
- [70] D.A. Turner, "Another Algorithm for Barcket Abstraction", Journal of Symbolic Logic Vol. 44(2) (June 1979).
- [71] M. Van Emden and R. Kowalski, "Semantics of Prolog as a Programming Language", JACM Vol. 7(3), pp.733-742 (1976).

- [72] W.W. Wadge, "Programming Constructs for Non Procedural Languages", University of Warwick: Theory of Computation Report #23.
- [73] W.W. Wadge, "An Extensional Treatment of Data Flow Deadlock", p. 285 in Semantics of Concurrent Computation., Springer Verlag.
- [74] D.H.D. Warren, "Implementing Prolog", 30, 40, Dept. of A.I., Univeristy of Edinburgh (1977).
- [75] D.H.D. Warren and L.M. Pereira, "Prolog-The Language and its Implementation Compared with Lisp", SIGPLAN Vol. 12(8) (August 1977).
- [76] D.H.D. Warren, "Logic Programming and Compiler Writing", Software Practice and Experience Vol. 10, pp.97-125 (1980).
- [77] I. Watson and J. Gurd, "A Prototype Data Flow Computer With Token Matching", Proc. AFIPS Conference Vol. 48 (1979).
- [78] W.T. Wilner, "Recursive Machines(I)", Xerox Palo Alto Research Centre.
- [79] W.T. Wilner, "Recursive Machines(II)", Xerox Palo Alto Research Centre.
- [80] M.J. Wise, "A Parallel Prolog: The Construction of a Data Driven Model", ACM Symposium on Lisp and Functional Languages (1982).

Appendices

APPENDIX ONE
MACHINE ARCHITECTURE IMPLEMENTATION

This appendix describes the use and implementation of the first version of the emulator. The purpose of this program is to emulate the packet communication architecture upon which control flow, data flow and reduction can be implemented as described in Chapter Four.

1.1. Instruction Format

The description of an instruction format given in Chapter Four is repeated below in more detail.

The following fields form an instruction:

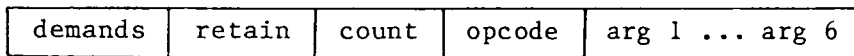


Figure 1.1: Instruction format.

- 1) demands: The flag is true if demands are expected by this instruction. If a demand is received and this flag is false the emulator will generate an error.

- 2) retain: This flag is true if the instruction is to be retained in AM once it has been executed.

- 3) count: This is the number of control and data tokens which this instruction must receive before it can be executed.
- 4) opcode: The operation code of the instruction, described in Chapter Four.
- 5) arg: Any number of arguments up to the maximum defined by the maxarg constant in the emulator, six at present. Arguments are described in Chapter Four.

1.2. Program Source Format

The source of the program consists of a sequence of procedures, the first one of which is the main body of the program; this is the code from which the initially executable instructions will be selected. Each of the subsequent procedures must have a "#" before the first instruction of the procedure. The "#" must be on its own line. Each procedure is referred to by a number, the number will be n if the procedure is the nth to be given, one for the first, two for the second and so on. The main body of the program has the number zero. All procedures start at a DM location whose address is an exact multiple of the maximum allowed size of a procedure. The maximum number of instructions a procedure may have is limited to five hundred. In this implementation the main program starts at location zero, the first procedure at location five hundred, the second at one thousand, and so on.

An instruction in source form has the format:

```
<instruction> = <control><arguments>
<control>    = [r][d]/count/opcode
<arguments>  = up to 6 of: argument type[/<value> or <address>]
<address>    = ([<process>]/[<location>]/[<slot>])
<value>     = integer
<process>    = natural number
<location>   = natural number
<slot>      = 1..6
```

default values

```
r          = false
d          = false
process    = -1
location   = 0
slot      = 1
```

r is the retain flag
d is the demands flag
[] means optional

Figure 1.2: Instruction source format.

1.3. Instruction Execution Cycle

A simple description of the instruction execution cycle is given in Chapter Four, the details of its implementation will be given here.

An instruction is copied into AM when it receives its first token, and will be obeyed by the processor when it becomes executable. An executable instruction will be one whose count is zero and that has at least one output argument, but there are five exceptions. These are the cond, call, print, ret and param instructions which may all be executed with no output arguments. A cond, call or print instruction can be executable with no output arguments only if the demands flag is false. All three of these instructions can be executed without producing a result, although they are all capable of doing so if required. The instructions will therefore be executable with no output arguments if no result will

ever be expected from them. This is the principle reason for the inclusion of the demands flag in the instruction format. If this is false no demands are expected by the instruction so no output arguments will ever be present in the instruction. The instruction may therefore be executed if the flag is false, even if there are no output arguments present. If demands flag is true the instruction expects a demand and will therefore not be executed until a demand has been received. A ret instruction never needs an output argument because it, in effect, uses those of the call instruction that its return address refers to. A param instruction has implicit output destinations, all slots in the instruction are used for input and so a param instruction may execute providing its count=0.

Once an instruction has become executable it is placed on a queue of instructions which the processor inspects whenever it needs a new task. The execution of the program stops when this queue becomes empty.

When the processor has selected an instruction for execution the instruction will be obeyed in the way described in Chapter Four. If the retain flag is set the instruction will be held in AM for future reference once its execution is complete. If the instruction produces a result it will be modified to become a dist1 instruction and the result will be placed in argument one. The value produced by the instruction may then be obtained by accessing it directly using an AM address, or by propagating a demand to it. If no result is produced the executed instruction is held in an its original form. The retain flag allows reduction to be implemented in two ways. Using a by-name mechanism, whenever a result is demanded it is recalculated, which will occur when the retain flag is false. Using lazy evaluation, the result is retained for future use by setting the retain flag.

1.4. Calling a Procedure

This section describes how procedures should be coded for each model of computation. The description also includes an explanation of how the various common parameter passing mechanisms can be implemented.

The call instruction should be executed according to the rules for the computation mechanism being used. For demand driven execution this will be when the result is demanded. When using a model of computation which relies on the availability of data, the call instruction should only be executed when the parameters are ready. For control flow this means sending the signals which indicate this to the call instruction. For data flow an additional instruction is used to collect the procedure's parameters. This is the the param instruction, which is placed immediately before the call in the program. Any data token which contains a parameter for the called procedure is sent to this instruction. The param instruction will only allow the call to proceed when all the parameters have arrived. Further details are given below.

The call instruction must send the return address to the called procedure. The return address is always the address of the call instruction itself.

Using the mechanism described above it is possible to implement any of the common type of parameter passing schemes.

Control Flow

The call will pass the return address to the called procedure. The call instruction is then retained as a dist1 instruction, with a first argument of type spare. When the modified call instruction is restarted, by a signal from the return instruction, it will signal those instructions specified by its output arguments. A call will therefore have the form

```
call p, sig , sig , ... ,sig
```

The following parameter mechanisms can be implemented:

Value: The parameter passing instruction in the calling code must be a dist instruction which should have a literal value for its first argument by the time the call is executed. This instruction will send the parameter to the procedure, where it will be stored in PM by a parameter distribution instruction.

call	procedure
call,p	dist1,n
dist,v	ret
	dist,unk,"PM address"

Result: The result will be copied from the procedure's area of PM into the calling code's area. The caller should pass the PM address of the location where the result is to be stored using a dist1 instruction. When the address arrives at the called procedure it should only be sent to the instruction that performs the final operation that produces the result. This will mean that the only the final version of the result will be returned to the caller.

call	procedure
call p	distl n
distl "PM address"	ret
	distl unk

Reference: The address in PM of the parameter is passed using a distl instruction as before, but this address is distributed though out the procedure body so that every reference to the parameter directly accesses the location which holds the value.

call	procedure
call,p	distl,n
distl,"PM address"	ret
	distl,unk,.....

Name: The value passed as a parameter should be the DM index of the procedure that will produce the required data. The procedure index will be an integer which will become the first argument of every call which produces the value of the parameter.

call	procedure
call,p	distl,n
dist,n	ret
	distl,unk,"IM address a"
	.
	.
	a:call,unk{procedure name}
	.
	.

Data Flow

The call instruction calls the procedure and is then deleted. There is no need to inform the calling code when the results are ready because the results are passed directly to the calling code in data tokens.

Input: The param instruction gathers all the parameters for the called procedure together, and then sends them to the parameter passing instructions. When all the parameters have arrived at the param instruction it also signals the call instruction to start executing. The parameter passing instructions send the parameter to the procedure in a data token, the parameter is then distributed through out the called procedure's body by the parameter handling instructions in the procedure's head. Using the param instruction imposes a limit on the number of input parameters that can be used for a procedure. There can only be as many input parameters as there are arguments in the instruction; this is the maximum number of parameters the param instruction can hold.

call	procedure
param,unk	
call,p	dist1,n
dist[1],value	ret
	dist[1],unk

output: The calling code must send the addresses of the all its instructions which will need the result. These are distributed in the procedure to those instructions which produce the results. The result will be send directly to the consuming instructions in the calling code which the result is produced.

call	procedure
param,unk	dist1,n
call,p	ret
dist1,address	dist1,unk,...

Reduction

By Availability

The call instruction will expect one signal for each parameter. The parameter passing instructions will load the parameter values from the instructions which produced them and pass the parameters into the called procedure. Having made the call, the call instruction is retained as a dist1 instruction which awaits the result. Upon receiving the result the dist1 (i.e. the old call) instruction signals the consumers of the result, which load the result for themselves.

input: The call will be as for data flow but the call instruction will have signal arguments.

call	procedure
call,p,sig ,...,sig	dist1,n
dist1,unk	ret
	dist1,unk ...

result: The result will be sent to the return instruction of the procedure which will pass it out to the modified call.

By Need

The call is executed when a demand is propagated to it. After having made the call, the call instruction is retained as a dist1 instruction so it may return its result to the consumers who demanded the result.

Input: To preserve the need driven scheme used in graph reduction a function argument must not be evaluated until its value is required. To achieve this the parameter sent must be an argument which will propagate the demand for the value when the time comes. The parameter must therefore be of type "prop". The parameter can be evaluated using either a by-name mechanism, or lazy evaluation.

call	procedure
call,p	dist1,n
dist1,"prop address"	ret
	dist1,unk ...

Result: The result will be sent to the return instruction of the procedure from where it will be sent to the caller for distribution though out the code.

The instructions used by the machine have a fixed format and therefore do not allow structures to be held, or passed as parameters. To overcome this pointers to the structure must be used instead.

1.5. Returning from a Procedure

In control flow all results are passed back via the procedure's parameters. Only the return address will therefore be present in the return instruction, which restarts the calling code by sending signal to this address.

In data flow the return instruction will not be executed because data tokens are used to pass results directly back to the instructions in the calling procedure which require them.

In reduction both arguments of the return instruction will be present. The result will be sent to the instruction specified by the return address which will then distribute the result in the calling code. This will be the modified call instruction.

1.6. Emulator Errors

If an error occurs, either during the reading of a program or the program's execution, the user is informed and the activity of the emulator is stopped. If the error occurred during the execution of a program the emulator will ask the user if he wants a postmortem dump of the state of the emulator, or an dump of the last sixty four Pascal statements executed.

1.7. Emulator Commands

The emulator supports two features that can be invoked by the user, these are: the tracing of a program the emulator is executing, or obtaining a list of the last sixty four Pascal statements executed in the event of an emulator error. All commands are typed in reply to the

prompt "?<".

1) finish

The emulator will return to shell.

2) trace

Turns tracing on. A trace will print the state of the machine before each instruction is executed, but the contents of DM are omitted. The trace will include the instruction being executed and a dump of the contents of both AM and PM, listed in process number and location order. Each instruction will be listed in full and is preceded by its address. The latter will include a slot number of one that should be ignored. If an emulation error occurs a postmortem dump will also be produced. This will show the state of the emulator at the point during the execution cycle at which the error occurred. An example of a trace is given below.

```
{put "1" in pm location for the print instruction}
/0/dist1,litv/1,pm/(/0/),sig/(/1/)

{print the value in pm location 0}
/1/print,pm/(/0/)
```

Figure 1.3: Program to print "1"

```
trace of program to print "1"

***instruction being executed***
(0/0/1):/0/dist1,litv/1,pm(/0/1),sig(/1/1)

AM
(0/0/1):/0/dist1,litv/1,pm(/0/1),sig(/1/1)

PM

***instruction being executed***
(0/1/1):/0/print,pm(/0/1)

AM
(0/1/1):/0/print,pm(/0/1)

PM

(0/0/1):/0/dist1,litv/1

1
```

Figure 1.4: Program trace.

3) no trace

Turns tracing off

4) dump

This command may be issued when the emulator has returned to the user after it has detect an error during the execution of the program. It will print the state of the machine at the time the command is given.

5) edebug

The Pascal system used to implement the emulator supports a feature known as edebug which records, in a cyclic buffer, the line numbers of the last sixty four statements executed. Whenever a Pascal runtime error occurs the contents of this buffer are dumped into a file named eml_last. If an emulator error occurs while obeying a program a dump of the most recently used Pascal statements can be produced by deliberately

causing a Pascal runtime error, by taking the log of a negative number. If the edebug option of the emulator is turned on this is what will happen. The dumped information will include the statements which were used to cause the Pascal error, the user should bear this fact in mind when inspecting the dump. The lines are ordered so that the the most recent is placed last.

6) no edebug

Turn the edebug option off.

7) programs

If the user gives any reply other than those listed above it is assumed to be the name of a file, and an attempt is made to open it. If this fails the message "cannot open file" is printed, but otherwise the file is read and the program it contains is executed.

Input Required During Execution

If during the execution of a program it requires data form the user the prompt "integer?<" will be printed, to which the user may reply with an integer value.

1.8. Example Programs

The programs below illustrate the use of the emulator. The first three all implement a program which will find the factorial of a number read from the user. The instruction numbers on the right, the comments, and the blank lines must not be included in a program to be executed by the emulator. Each program is followed by an abbreviated trace: only the executing instructions are shown.

Control Flow

```
    {read a value, put it in PM and signal the data's user}
0  /0/read,pm(/0/),sig(/1/)

    {call procedure one, signal instruction 4 when return}
1  /1/call,litv/1,sig(/4/)

    {sent the first parameter to the procedure, parameter is n}
2  /1/dist,pm(/0/),unk

    {send the address of the location which is to hold the result}
3  /1/dist1,pm(/1/),unk

    {print the result}
4  /1/print,pm(/1/)

{the procedure factorial}
#

    {the number of parameters: 2}
500 /0/dist1,litv/2

    {return instruction, return address supplied by call}
501 /2/ret,unk

    {distribute the first parameter, n, into the procedure}
502 /1/dist,unk,pm(/0/),sig(/4/),sig(/7/),sig(/11/)

    {distribute the address for the result into the procedure}
503 /1/dist1,unk,am(/6/2),am(/11/3)

    {n=0?, put result in PM (location 1) and signal conditional}
504 /1/eq,pm(/0/),litv/0,pm(/1/),sig(/5/)

    {get result of n=0?. signal appropriate section of code according-
to result}
505 /1/cond,pm(/1/),sig(/6/),sig(/7/)

    {here if n=0. put "1" in result location and signal return instruct-
ion}
506 /2/dist,litv/1,unk,sig(/1/)

    {here if n<>0. calculate n-1, save it for call of factorial}
507 /2/sub,pm(/0/),litv/1,pm(/2/),sig(/8/)

    {factorial (n-1)}
508 /1/call,litv/1,sig(/11/)

    {parameter instruction for n-1}
509 /1/dist,pm(/2/),unk

    {parameter instruction for location to hold result}
510 /1/dist1,pm(/3/),unk

    {multiply result of factorial(n-1) by n. send signal to return}
511 /3/mul,pm(/0/),pm(/3/),unk,sig(/1/)
```

trace of control flow factorial

```
{read the value whose factorial is required}
(0/0/1):/0/read,pm(/0/1),sig(/1/1)
```

```
integer?<      1
```

```
{calculate factorial(1), first argument identifies factorial procedure}
(0/1/1):/0/call,litv/1,sig(/4/1)
```

```
{sent the parameter, 1, to factorial (the second instruction)}
(0/2/2):/0/dist,pm(/0/1),am(/1/502/1)
```

```
{send the address of the location which is to hold the result}
(0/3/2):/0/dist1,pm(/1/1),am(/1/503/1)
```

```
{first instruction of factorial, distribute n into the body}
(1/502/1):/0/dist,litv/1,pm(/500/1),sig(/504/1),sig(/507/1),sig(/511-
/1)
```

```
{distribute the address of the location which will hold the result}
(1/503/1):/0/dist1,pm(/0/1/1),am(/506/2),am(/511/3)
```

```
{is n=0? save result and signal conditional}
(1/504/1):/0/eq,pm(/500/1),litv/0,pm(/501/1),sig(/505/1)
```

```
{signal appropriate sections of code according to result of n=0}
(1/505/1):/0/cond,pm(/501/1),sig(/506/1),sig(/507/1)
```

```
{n<>0, therefore calculate factorial(n-1). first calculate n-1}
(1/507/1):/0/sub,pm(/500/1),litv/1,pm(/502/1),sig(/508/1)
```

```
{now make recursive call of factorial with 0 as parameter}
(1/508/1):/0/call,litv/1,sig(/511/1)
```

```
{send 0 to factorial}
(1/509/2):/0/dist,pm(/502/1),am(/2/502/1)
```

```
{send address of location for result of factorial(0)}
(1/510/2):/0/dist1,pm(/503/1),am(/2/503/1)
```

```
{distribute n'-1 (0) into body of new activation of factorial}
(2/502/1):/0/dist,litv/0,pm(/500/1),sig(/504/1),sig(/507/1),sig(/511-
/1)
```

```
{distribute the address of the location to hold the result of factorial-
(0)}
(2/503/1):/0/dist1,pm(/1/503/1),am(/506/2),am(/511/3)
```

```
{is n=0?}
(2/504/1):/0/eq,pm(/500/1),litv/0,pm(/501/1),sig(/505/1)
```

```
{signal appropriate sections of code according to the result of n=0}
(2/505/1):/0/cond,pm(/501/1),sig(/506/1),sig(/507/1)
```

```
{n'=0. use "1" as the result of factorial(0), signal return instruction}
(2/506/2):/0/dist,litv/1,pm(/1/503/1),sig(/501/1)
```

```
{signal caller that factorial(0) has been calculated}  
(2/501/1):/0/ret,am/(1/508/1),spare
```

```
{signal user of factorial(0)}  
(1/508/1):/0/dist1,spare,sig/(/511/1)
```

```
{calculate 1*factorial(0) and signal return for factorial(1)}  
(1/511/1):/0/mul,pm/(/500/1),pm/(/503/1),pm/(0/1/1),sig/(/501/1)
```

```
{signal caller that factorial(1) has been calculated}  
(1/501/1):/0/ret,am/(0/1/1),spare
```

```
{signal user of factorial(1) that it has been calculated}  
(0/1/1):/0/dist1,spare,sig/(/4/1)
```

```
{print factorial(1)}  
(0/4/1):/0/print,pm/(/1/1)
```

```
{factorial(1)}  
1
```

Data Flow

```
    {read the value whose factorial is required}
0  /0/read,am/(/1/1)

    {gather all the parameters for the call}
1  /1/param,unk

    {call factorial}
2  /1/call,litv/1

    {send the value of n to factorial}
3  /2/dist,unk,unk

    {send the address of the instruction which is to receive the resu-
    lt}
4  /1/dist1,am/(/5/1),unk

    {print the result}
5  /1/print,unk

{the factorial function}
#

    {there are two parameters}
500 /0/dist1,litv/2

    {the return instruction, which is never used}
501 /1/ret

    {distribute the value of n into the body}
502 /1/dist,unk,am/(/4/1),am/(/7/1),am/(/12/1)

    {distribute the address of the instruction to receive the result}
503 /1/dist1,unk,am/(/6/2),am/(/12/3)

    {is n=0?}
504 /1/eq,unk,litv/0,am/(/5/1)

    {signal appropriate sections of code according to the result of n=0}
505 /1/cond,unk,sig/(/6/),sig/(/7/)

    {here if n=0. "1" is the result so send to instruction requiring result}
506 /2/dist,litv/1,unk

    {here if n<>0. calculate n-1 for factorial(n-1)}
507 /2/sub,unk,litv/1,am/(/8/1)

    {gather the parameters for recursive call of factorial}
508 /1/param,unk

    {call factorial}
509 /1/call,litv/1

    {send n-1(from parameter) to factorial}
510 /2/dist,unk,unk

    {send address of instruction requiring factorial(n-51) to factorial}
511 /1/dist1,am/(/12/2),unk
```

{n*factorial(n-51), send to calling instruction which requires resu-
lt}
512 /3/mul,unk,unk,unk

trace of data flow factorial

{read the value whose factorial is required}
(0/0/1):/0/read,am(/1/1)

integer?< 1

{gathered the parameter, "1"}
(0/1/1):/0/param,litv/1,spare,spare,spare,spare,spare

{call factorial, the first arg identifies the function}
(0/2/1):/0/call,litv/1

{send n to factorial}
(0/3/1):/0/dist,litv/1,am(/1/502/1)

{send the address of the instruction which requires factorial(1)}
(0/4/2):/0/dist1,am(/5/1),am(/1/503/1)

{first instruction of factorial(n=1), distribute n into body}
(1/502/1):/0/dist,litv/1,am(/504/1),am(/507/1),am(/512/1)

{distribute the address of the instruction requiring the result}
(1/503/1):/0/dist1,am(/0/5/1),am(/506/2),am(/512/3)

{is n=0?}
(1/504/1):/0/eq,litv/1,litv/0,am(/505/1)

{signal appropriate section of code according to the result of n=0}
(1/505/1):/0/cond,litv/0,sig(/506/1),sig(/507/1)

{n<>0. calculate n-1 for factorial(n-1)}
(1/507/1):/0/sub,litv/1,litv/1,am(/508/1)

{gather parameters for recursive call}
(1/508/1):/0/param,litv/0,spare,spare,spare,spare,spare

{call factorial, first arg identifies function}
(1/509/1):/0/call,litv/1

{send n-1(0) to factorial}
(1/510/1):/0/dist,litv/0,am(/2/502/1)

{send the address of the instruction requiring factorial(0)}
(1/511/2):/0/dist1,am(/512/2),am(/2/503/1)

{first instruction of factorial(0), distribute n' into body}
(2/502/1):/0/dist,litv/0,am(/504/1),am(/507/1),am(/512/1)

{distribute the address of the instruction requiring factorial(0)}
(2/503/1):/0/dist1,am(/1/512/2),am(/506/2),am(/512/3)

{n'=0?}
(2/504/1):/0/eq,litv/0,litv/0,am(/505/1)

{signal the appropriate section of code according to the result of n'=0}
(2/505/1):/0/cond,litv/1,sig(/506/1),sig(/507/1)

```
{n'=0. "1" is the result, send it to the instruction which requires it}  
(2/506/2):/0/dist,litv/1,am/(1/512/2)
```

```
{1*factorial(0), send result to the calling instruction which requires it}  
(1/512/1):/0/mul,litv/1,litv/1,am/(0/5/1)
```

```
{print factorial(1)}  
(0/5/1):/0/print,litv/1
```

```
{factorial(1)}  
1
```


Reduction

```
    {print factorial(n)}
0  /0/print,prop(/1/)

    {call factorial, first arg identifies function}
1  dr/0/call,litv/1

    {the param is the instruction which will generate n}
2  /1/dist1,prop(/3/),unk

    {read n}
3  dr/0/read

{factorial function}
#

    {1 parameter}
500 /0/dist1,litv/1

    {demand result to be returned to caller}
501 /1/ret,unk,prop(/3/)

    {distribute parameter into the body of factorial}
502 /1/dist1,unk,am(/4/1),am(/5/1),am(/8/1)

    {get the result of factorial(n) depending on whether n=0 or not}
503 dr/0/cond,prop(/4/),litv/1,prop(/5/)

    {is n=0?}
504 dr/1/eq,unk,litv/0

    (n*factorial(n-1))
505 dr/1/mul,unk,prop(/6/)

    (factorial(n-1))
506 dr/0/call,litv/1

    {parameter is instruction which will calculate n-1}
507 dr/1/dist1,prop(/8/),unk

    {n-1}
508 dr/1/sub,unk,litv/1
```

trace of reduction factorial

```
{print factorial(n)}  
(0/0/1):/0/print,prop/(/1/1)
```

```
{call factorial, first argument identifies function}  
(0/1/1):dr/0/call,litv/1,am/(0/0/1)
```

```
{demand result of factorial from the body of the function}  
(1/501/1):/0/ret,am/(0/1/1),prop/(/503/1)
```

```
{pass prop arg which will produce the parameter of the function}  
(0/2/2):/0/dist1,prop/(/3/1),am/(1/502/1)
```

```
{demand result of n=0, and then demand the result of factorial}  
(1/503/1):dr/0/cond,prop/(/504/1),litv/1,prop/(/505/1),am/(1/501/2)
```

```
{distribute the parameter into the body of factorial}  
(1/502/1):/0/dist1,prop/(0/3/1),am/(/504/1),am/(/505/1),am/(/508/1)
```

```
{is n=0?}  
(1/504/1):dr/0/eq,prop/(0/3/1),litv/0,am/(1/503/1)
```

```
{need n, so read it}  
(0/3/1):dr/0/read,am/(1/504/1)
```

integer?< 1

```
{now can find out if n=0}  
(1/504/1):dr/0/eq,litv/1,litv/0,am/(1/503/1)
```

```
{n<>0, demand result of n*factorial(n-1)}  
(1/503/1):dr/0/cond,litv/0,litv/1,prop/(/505/1),am/(1/501/2)
```

```
{propagate demands for n and factorial(n-1)}  
(1/505/1):dr/0/mul,prop/(0/3/1),prop/(/506/1),am/(1/503/3)
```

```
{the old read instruction, now returns n}  
(0/3/1):dr/0/dist1,litv/1,am/(1/505/1)
```

```
{make recursive call of factorial}  
(1/506/1):dr/0/call,litv/1,am/(1/505/2)
```

```
{return for recursive call, demand result of factorial n' (=n-1)}  
(2/501/1):/0/ret,am/(1/506/1),prop/(/503/1)
```

```
{send parameter to recursive call; parameter propagates demand for n-1}  
(1/507/2):dr/0/dist1,prop/(/508/1),am/(2/502/1)
```

```
{propagate demand for n=0?}  
(2/503/1):dr/0/cond,prop/(/504/1),litv/1,prop/(/505/1),am/(2/501/2)
```

```
{distribute n into body}  
(2/502/1):/0/dist1,prop/(1/508/1),am/(/504/1),am/(/505/1),am/(/508/1)
```

```
{is n=0?}  
(2/504/1):dr/0/eq,prop/(1/508/1),litv/0,am/(2/503/1)
```

```
{n'=n-1}
(1/508/1):dr/0/sub,prop/(0/3/1),litv/1,am/(2/504/1)

{the old read instruction, returns n}
(0/3/1):dr/0/dist1,litv/1,am/(1/508/1)

{now calculate n-1}
(1/508/1):dr/0/sub,litv/1,litv/1,am/(2/504/1)

{now n'=0?}
(2/504/1):dr/0/eq,litv/0,litv/0,am/(2/503/1)

{propagate demand for factorial(n')}
(2/503/1):dr/0/cond,litv/1,litv/1,prop/(/505/1),am/(2/501/2)

{result of factorial(n') is 1}
(2/501/1):/0/ret,am/(1/506/1),litv/1

{return result of factorial(n') to instruction which demanded it}
(1/506/1):dr/0/dist1,litv/1,am/(1/505/2)

{calculate n*factorial(n-1)}
(1/505/1):dr/0/mul,litv/1,litv/1,am/(1/503/3)

{return result to instruction which demanded it from the conditional}
(1/503/1):dr/0/cond,litv/0,litv/1,litv/1,am/(1/501/2)

{return factorial(n) to caller}
(1/501/1):/0/ret,am/(0/1/1),litv/1

{old call for factorial(n), send result to instructions which demanded-
it}
(0/1/1):dr/0/dist1,litv/1,am/(0/0/1)

{print factorial(1)}
(0/0/1):/0/print,litv/1

{factorial(1)}
  1
```

The following program illustrates reduction driven by the availability of data. It prints the result of $(1+2)*(3-4)$.

```
    {print the result of (1+2)*(3-4)}
0  r/1/print,am(/1/)

    {multiply (1+2) and (3-4)}
1  r/2/mul,am(/2/),am(/3/),sig(/0/)

    {1+2}
2  r/0/add,litv/1,litv/2,sig(/1/)

    {3-4}
3  r/0/sub,litv/3,litv/4,sig(/1/)
```

trace of availability reduction

```
{1+2}
(0/2/1):r/0/add,litv/1,litv/2,sig(/1/1)

{3-4}
(0/3/1):r/0/sub,litv/3,litv/4,sig(/1/1)

{multiply (1+2) and (3-4)}
(0/1/1):r/0/mul,am(/2/1),am(/3/1),sig(/0/1)

{print (1+2)*(3-4)}
(0/0/1):r/0/print,am(/1/1)

{(1+2)*(3-4)}
-3
```

The following program illustrates control flow iteration using a synchronisation token. It prints the values 1 and 2.

```
    {initialise counter and start loop}
0  /0/dist,litv/1,pm/(/0/),sig/(/1/)

    {the first instruction of the loop, print the counter}
1  /1/print,pm/(/0/),sig/(/2/)

    {increment the counter}
2  /1/add,litv/1,pm/(/0/),pm/(/0/),sig/(/3/)

    {is counter=3?}
3  /1/ne,litv/3,pm/(/0/),pm/(/1/),sig/(/4/)

    {start another iteration if counter<>3}
4  /1/cond,pm/(/1/),sig/(/1/)
```

trace of iteration

```
{set counter to 1}
(0/0/1):/0/dist,litv/1,pm/(/0/1),sig/(/1/1)

{print the counter}
(0/1/1):/0/print,pm/(/0/1),sig/(/2/1)

{the counter at start of first iteration}
1

{increment the counter}
(0/2/1):/0/add,litv/1,pm/(/0/1),pm/(/0/1),sig/(/3/1)

{is counter=3}
(0/3/1):/0/ne,litv/3,pm/(/0/1),pm/(/1/1),sig/(/4/1)

{start another iteration, counter<>3}
(0/4/1):/0/cond,pm/(/1/1),sig/(/1/1),spare

{next iteration, print the counter}
(0/1/1):/0/print,pm/(/0/1),sig/(/2/1)

{the counter at the start of the second iteration}
2

{increment the counter}
(0/2/1):/0/add,litv/1,pm/(/0/1),pm/(/0/1),sig/(/3/1)

{is the the counter = 3 now?}
(0/3/1):/0/ne,litv/3,pm/(/0/1),pm/(/1/1),sig/(/4/1)

{yes, don't start another iteration}
(0/4/1):/0/cond,pm/(/1/1),sig/(/1/1),spare
```

APPENDIX TWO

EXTENDED EXPLANATION OF COMBINATORS

This appendix gives a more precise description of the compilation of an function into combinators. The appendix also introduces additional combinators for which it provides the graph reduction rules.

2.1. Compilation to Combinators

The compilation process establishes a relationship between the original source code of a function, and the combinator expression which the compilation produces. To compile an expression into combinators the bound variable is abstracted from the body in much the same way as for Lambda Notation, but now the result is a combinator expression and not a lambda expression.

The abstraction process operates by dividing the outer-most function application in the source code into the operator and operand. The abstraction is then performed recursively on the inner function applications. As each division is made an **S** combinator is introduced. The first two arguments of this combinator are the operator and operand of the application just divided. Both the operator and operand will now have the bound variable abstracted from them in turn. If either the operator or the operand is a single identifier or constant, a **K** or **I** combinator must be introduced. If the identifier is the bound variable then it is replaced by an **I** to ensure that the function argument is accepted. If however, the identifier is not the bound variable it is

prefixed by a **K** so that the function argument will be rejected.

The abstraction algorithm for each combinator may be summarised as follows, where $[x]E$ means abstract x from the expression E .

1) **S**.

$$[x](E_1 E_2) \Rightarrow \mathbf{S}([x]E_1)([x]E_2)$$

Here E_1 is the operator and E_2 the operand. To abstract from the complete expression introduce an **S** and abstract x from the operator and the operand.

2) **I**.

$$[x]x \Rightarrow \mathbf{I}$$

Abstracting the bound variable from itself will require the introduction of an **I** so the function argument will be accepted.

3) **K**.

$$[x]y \Rightarrow \mathbf{K}y \quad (y \neq x)$$

Abstracting the bound variable from a different identifier, or from a constant, means that the symbol must be prefixed by a **K** to ensure that the function argument will be rejected.

In general the abstraction process is defined by the following rule:

$$E = ([x]E)x$$

Substituting the bound variable into an abstracted expression regenerates the original expression (the above equation applies equally well to Lambda Notation). Therefore applying a combinator expression to a

function argument will reverse the abstractions given above.

A compiler which will abstract a bound variable and produce a combinator expression is shown in Figure 2.1.

```
abstract x E = if id(E)
               then if E = x
                     then I
                     else K E
               else S(abstract x (operator E))
                   (abstract x (operand E))
```

Figure 2.1: Combinator compiler[10].

The functions operator and operand select the appropriate parts of the expression supplied as their argument, and the function id returns true if E is only one identifier long.

For example take the expression fgx, which is to have x abstracted from it. The compilation follows the steps below:

- a) Divide the expression into its operator and operand

operator = fg operand = x

- b) Introduce an S combinator, and abstract x from the operator and operand according to compilation rule 1):

S ([x]fg) ([x] x)

- c) [x] x => I according to rule 2).

- d) [x]fg. Divide the expression into its operator and operand:

operator = f operand = g

- e) Introduce an **S** combinator and abstract **x** from the operator and operand according to rule 1):

$$\mathbf{S} ([\mathbf{x}] f) ([\mathbf{x}] g)$$

- f) $[\mathbf{x}] f \Rightarrow \mathbf{K} f$ by rule 3).

- g) $[\mathbf{x}] g \Rightarrow \mathbf{K} g$ by rule 3).

- h) Now substitute results f) and g) back into expression e):

$$\mathbf{S} (\mathbf{K} f)(\mathbf{K} g)$$

- i) Now substitute results c) and h) back into expression b):

$$\mathbf{S} (\mathbf{S} (\mathbf{K} f)(\mathbf{K} g)) \mathbf{I}$$

which is the result of the abstraction.

To translate a multi-argument function such as $\lambda x.\lambda y.xy$, the abstraction algorithm must be applied several times, just as abstraction should be applied several times in the Lambda Notation. Each argument is abstracted in turn, starting with the innermost one, **x**. The result of one abstraction is the subject of the next abstraction. To compile the example, first abstract **x** and then abstract **y** from the result. Any combinators introduced in first abstraction are treated as constants in the second one. For example, the **S** combinators introduced in the first abstraction will have to be prefixed by **K** combinators in the second abstraction, to ensure that the **S** is kept for the substitution of the first bound variable.

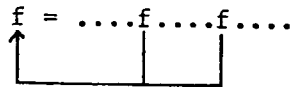
$$\begin{aligned} [\mathbf{y}]xy &\Rightarrow \mathbf{S}(\mathbf{K}\mathbf{x})\mathbf{I} \\ [\mathbf{x}]([\mathbf{y}]xy) &\Rightarrow [\mathbf{x}](\mathbf{S}(\mathbf{K}\mathbf{x})\mathbf{I}) \\ &\Rightarrow \mathbf{S}(\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}\mathbf{K}))\mathbf{I}) (\mathbf{K}\mathbf{I}) \end{aligned}$$

The substitution will be carried out in the reverse order to abstrac-

tion; outermost bound variable first, as with x in the example.

2.2. Recursion using Combinators

Recursion poses special problems for combinators, because in order to remove the recursive references from the function body, one must introduce a cyclic structure, which is difficult to encode.

$$f = \dots f \dots f \dots$$


Fortunately, Fixpoint Theory[55] provides a solution, but to follow it one must first understand the a meaning of a fixpoint. The fixpoint (p) of a function (f) is the value which is returned as the result, when the same value is given as an argument:

$$p = f p$$

For example the fixpoint of the function double is 0:

$$\begin{aligned} \text{double } x &= 2 * x \\ 0 &= \text{double } 0 \end{aligned}$$

Since we are going to use a fixpoint to represent a function, the fixpoint itself must be a function. For a recursive function f to have another function as its fixpoint, f must be a functional, denoted F. A functional is a function that has another function as its arguments and result. A function may have several, or indeed an infinite number of fixpoints, of which the least fixpoint is the most important here. The least fixpoint is the one which is least defined. In terms of functions this means the function which produces a result for the smallest section of its domain.

Fixpoint Theory provides a way of representing recursion because the least fixpoint of a functional derived from a recursive function f ; is equivalent to f . Since such a least fixpoint is not recursive, substituting it for the original function removes the recursion. Kleene (see [42]) has shown that every recursive function has such a least fixpoint, providing certain constraints are imposed upon the function, which need not concern us here.

The first stage in finding the least fixpoint is to convert the recursive function into a functional. This is done by abstracting the function name from its own body. In this way all the recursive references in the function body are replaced by one, the argument.

$$\begin{aligned} f &= \dots f \dots f \dots \\ &= ([f](\dots f \dots f \dots))f \\ &\text{where } [f](\dots f \dots f \dots) \text{ is the functional } F \end{aligned}$$

For example take the factorial function:

$$\begin{aligned} \text{fac } n &= \text{if } n=1 \text{ then } 1 \text{ else } n*\text{fac}(n-1) \\ &= ([\text{fac}] (\text{if } n=1 \text{ then } 1 \text{ else } n*\text{fac}(n-1))) \text{ fac} \end{aligned}$$

In order to find the least fixpoint of the functional, a new combinator, Y , is introduced, which when applied to the body of a functional returns its least fixpoint, lp .

$$\begin{aligned} lp &= Y([f](\dots f \dots f \dots)) \\ \text{or } lp &= Y F \end{aligned}$$

Since Y manipulates one function and returns another, it will appear to be a very sophisticated combinator indeed, but this need not be the case, as will be shown below.

The **Y** combinator is used to represent recursion without cyclic references, but semantically its result is equivalent to the original function. Thus when the function is compiled into combinators, **Y** is used to represent the recursion, and preserve its meaning, without introducing a cyclic structure. For example factorial will be represented as:

$$\text{fac } n = \mathbf{Y} ([\text{fac}](\text{if } n=1 \text{ then } 1 \text{ else } n*\text{fac}(n-1)))$$

When the function comes to be evaluated, finding the least fixpoint can be avoided by allowing the recursion to be unwound by replicating the function body. This replication is caused by the substitution rule for **Y**, which can be derived from the original definition of a fixpoint (or the least fixpoint (lp) in this case):

$$\text{lp} = F \text{ lp}$$

Since the $\text{lp} = \mathbf{Y}F$

$$\mathbf{Y}F = F(\mathbf{Y}F)$$

where F is the functional body

$$[f](\dots f \dots f \dots)$$

Thus if $\mathbf{Y}F$ is the least fixpoint of f , and $\mathbf{Y}F$ is evaluated we have $F(\mathbf{Y}F)$, which in terms of the functional body is:

$$(\dots(\mathbf{Y}F)\dots(\mathbf{Y}F)\dots)$$

because $\mathbf{Y}F$ will replace each occurrence of f in the F . We denote this by F' . Now both $\mathbf{Y}F$ s will be reduced, giving:

$$(\dots F(\mathbf{Y}F)\dots F(\mathbf{Y}F)\dots)$$

according to the reduction rule for Y , and if the substitution into F is carried out:

$$(\dots (\dots(YF)\dots(YF)\dots) \dots (\dots(YF)\dots(YF)\dots) \dots)$$

which we denote F'' . Now the inner Y s can be reduced and the whole process can be repeated. If F'' is represented in terms of the reduction rule for Y it will have the form:

$$F'' = F(F(YF))$$

The complete reduction of Y performed so far therefore is:

$$\begin{aligned} YF &= F(YF) \\ &= F(F(YF)) \end{aligned}$$

Now the innermost YF will be reduced giving:

$$F(F(F(YF)))$$

This expansion may continue infinitely, but it will usually stop because no recursive calls are made at a particular level. In this situation the reduction of YF is replaced by the result the function f would produce if f did not make a recursive call. For factorial this result is 1, which gives the expression:

$$F(F(\dots(1)\dots))$$

To produce the result of the recursion all substitutions in the this expression must be carried out, and the resulting expression reduced.

If a least fixpoint is applied to an argument x the expansion starts with $(YF)x$. The Y will be reduced first giving:

$$F(YF)x$$

and the least fixpoint is substituted into F gives F' as before. Now x will be substituted into F' . Each recursive call in F' will have the form $(YF)x'$, where x' is the parameter of the recursive call. If this

call is evaluated the whole process is repeated. First **Y** is reduced giving $F(\mathbf{YF})x'$, and then F'' is produced by substituting \mathbf{YF} into F , and so on.

2.2.1. Efficiency Considerations for Combinators

As each variable is abstracted from an expression, an exponential growth of the expression occurs compared to the original. This is due to the way combinators from one abstraction are treated by subsequent ones. Since the introduction of combinators obviously make the expression longer there will be more operator/operand pairs, and consequently more **Ss** will be needed to distribute the next abstraction's bound variable over the expression. In addition each combinator carried from one abstraction to the next will need a **K** to protect it from the latest bound variable. Combinators from one abstraction lead directly to extra combinators in the following one, which in turn lead to more in subsequent abstractions, producing an exponential growth overall.

In fact the length of the new expression is:

$$\begin{aligned} \text{newlen} &= b+2*(\text{len}-b)+(\text{len}-1) \\ &= 3*\text{len}-b-1 \end{aligned}$$

where b = the number of occurrences of the bound variable
and len = length of the expression before abstraction.

This can be explained by referring to the first equation. The first term (b) is the number of **Is** that will be introduced into the combinator expression, since every bound variable in the original must be replaced by an **I**. The second subexpression deals with the introduction of **Ks**. There will be $\text{len}-b$ free variables or constants; a **K** will added for each giving $2*(\text{len}-b)$ identifiers. Lastly an **S** is introduced for each operator/operand pair, there being $\text{len}-1$ in total.

The exponential expansion can cause efficiency problems because it will result in a large storage requirement. There are basically two ways to control this problem, the first is to recognise simplifications in the combinator expressions, and the second is to employ new combinators to represent commonly occurring subexpressions.

There are two simplifications:

$$1) \quad \mathbf{S}(\mathbf{K} E_1)(\mathbf{K} E_2) = \mathbf{K}(E_1 E_2)$$

Since E_1 and E_2 are constants which do not use the bound variable, the variable may be rejected from both E_1 and E_2 simultaneously, rather than distributed to each for individual rejection.

$$2) \quad \mathbf{S}(\mathbf{K} E_1) \mathbf{I} = E_1$$

The above expression results when the argument is abstracted from a simple function application, as in $[x](E_1 x)$, where E_1 is the function. The reason for E_1 replacing the usual combinator expression lies in the definition of abstraction:

$$([x] E)x = E$$

Given that $E = E_1 x$, it follows from the above that:

$$[x]E = E_1$$

because substituting E_1 for $([x] E)$ in the abstraction definition produces the original expression, $E_1 x$. So E_1 can replace the normal abstraction result of $\mathbf{S}(\mathbf{K} E_1) \mathbf{I}$.

There are two extra combinators which are useful:

- 1) **B**, Bracket, groups the last two operands together.

$$[x] (E_1 E_2) \Rightarrow B E_1 ([x] E_2)$$

This combinator is used if the operator (E_1) does not use x . Thus only E_2 need have it abstracted, and substituted back. It is the substitution rule that gives **B** its name since to substitute x back into E_2 only, one must bracket the last two operands of **B**, as shown below.

$$\begin{aligned} B E_1 E_2 x &= E_1 (E_2 x) \\ &= E_1 . E_2 x \end{aligned}$$

- 2) **C**, Converse, swaps its last two arguments.

$$[x] (E_1 E_2) \Rightarrow C([x] E_1) E_2$$

C is the opposite to **B**, only the operator uses the bound variable, so when substitution occurs only E_1 will need x , consequently the last two operands of **C** must be swapped to apply E_1 to x :

$$C E_1 E_2 x = E_1 x E_2$$

The two combinators above must only be used once the simplification rules described earlier have been applied. These rules apply only to expressions that contain **S**, **K** and **I**, so if **B** and **C** were introduced before the simplification rules were applied the opportunity to use them will be missed. This will mean that superfluous combinators will be retained, because they will have been converted to **B** and **C** before the simplification rules could have removed them.

2.2.2. Improved Abstraction Rules

Although the above techniques help control the size of abstraction results, they do not prevent an exponential growth occurring. Consider the expression $E_1 E_2$. As each abstraction is performed the expression grows because the combinators produced by one abstraction form the expression submitted to the next[70]:

$$\begin{array}{ll}
 S E_1' E_2' & \text{first abstraction} \\
 S(B S E_1'') E_2'' & \text{second abstraction} \\
 S(B S(B(B S) E_1''')) E_2''' & \text{third abstraction}
 \end{array}$$

The number of apostrophes denote the number of abstractions performed on E_1 and E_2 .

To overcome the problem of growth, Turner[70] has introduced three more combinators, which are slightly modified versions of S,B and C, denoted by S' , B' and C' . Their behaviour may be understood by studying just one, S' .

The problem with the standard abstraction rules is that they place a combinator in front of the expression being abstracted from. In the next abstraction additional combinators must be introduced in order to protect these combinators from the current bound variable when it is substituted, and to distribute the bound variable over the now larger expression. Turner's new combinators overcome this problem by introducing a new argument that does not have the bound variable substituted into it. This argument becomes the combinators introduced by earlier abstractions, but it can be any constant expression:

$$\begin{array}{l}
 S' k f g x = k (f x)(g x) \\
 \text{where } k \text{ is the constant expression}
 \end{array}$$

The combinator "reaches over" the constant expression k and then applies the normal S rule to the remaining arguments. The example sequence of abstractions now becomes:

$$\begin{aligned} & S E_1' E_2' \\ & S' S E_1'' E_2'' \\ & S' (S' S) E_1''' E_2''' \end{aligned}$$

Only one combinator is introduced into the expression for every abstraction, so the growth is now linear.

The definitions of C' and B' are:

$$C' k f g x = k (f x) g$$

$$B' k f g x = k f (g x)$$

Both combinators copy the first argument, and apply the usual C or B rule to the remaining three.

The abstraction rules for S' , C' and B' are the same as for their simpler counterparts, but with the constant expression added. To complete the description of the new combinators their abstraction rules are:

1) S' .

$$[x](E_1 E_2 E_3) \Rightarrow S' E_1 ([x] E_2)([x] E_3)$$

where E_1 is constant.

2) C' .

$$[x](E_1 E_2 E_3) \Rightarrow C' E_1 ([x] E_2) E_3$$

where both E_1 and E_3 are constant.

3) B' .

The abstraction rule for B' is similar to the ones above, but the situations in which it is used are complex, and require some explanation. Consider:

$$[x](E_1 E_2 E_3) \Rightarrow B' E_1 E_2 ([x] E_3)$$

where E_1 and E_2 are constant

The expression E_1 is the constant that must be stepped over while E_2 and E_3 form the usual abstraction rule for B . Although this is a valid use of B' , it is not the combinator expression Turner will produce. Instead he uses the original B , as shown below, preferring to use the new combinators only when absolutely necessary.

$$[x](E_1 E_2 E_3) \Rightarrow B (E_1 E_2) ([x] E_3)$$

Since E_1 and E_2 are constant they can be grouped together to form the constant expression in the abstraction rule for B . The B' combinator will only be used if the grouping of E_1 and E_2 does not occur. Such a situation will arise if a second abstraction were performed on the expression above, in which only E_2 uses the second bound variable. This has the effect of dividing E_1 and E_2 since only the latter needs the the new bound variable when it is substituted. Ignoring the combinators introduced for the moment, this means performing the abstractions below, in which the subscripts of x denote the order in which they are abstracted; x_1 first and x_2 second:

$$E_1 ([x_2] E_2)([x_1] E_3)$$

Of course E_1 and E_2 could be kept together by putting combinators

round the E_1 to make it reject x_2 when it was substituted, but this introduces more combinators, which is precisely what the new rules seek to avoid.

In order to reverse the abstraction correctly the appropriate combinators must be selected. The expression, including the values to be substituted, will have the form:

$$E_1 ([x_2] E_2)([x_1] E_3) x_2 x_1$$

The substitution of x_2 into E_2 will be accomplished using C' , (see its rule above):

$$\begin{aligned} C' E_1 ([x_2] E_2)([x_1] E_3) x_2 x_1 &= E_1 (([x_2] E_2)x_2) ([x_1] E_3) x_1 \\ &= E_1 E_2 ([x_1] E_3) x_1 \end{aligned}$$

Notice that the last expression is that which was originally given for B' , so the substitution of x_1 can be achieved using that combinator:

$$\begin{aligned} B' E_1 E_2 ([x_1] E_3) x_1 &= E_1 E_2 (([x_1] E_3) x_1) \\ &= E_1 E_2 E_3 \end{aligned}$$

The complete combinator expression will therefore be:

$$C' (B' E_1) ([x_2] E_2)([x_1] E_3)$$

resulting from the abstraction rule:

$$[x_2]([x_1](E_1 E_2 E_3)) \Rightarrow C'(B' E_1)([x_2] E_2)([x_1] E_3)$$

where E_1 is constant, E_2 constant with respect to x_1 and E_3 constant with respect to x_3 .

At present only E_2 uses x_2 , but it is quite possible that E_3 will use x_2 as well. Since it is C' that handles the substitution of x_2 this will need to be changed. The combinator S' would seem to be the correct replacement because it substitutes the variables into both the second and third arguments instead of just the second. Thus the abstraction rule will be:

$$[x_2]([x_1](E_1 E_2 E_3)) \Rightarrow S'(B' E_1)([x_2] E_2)([x_2]([x_1] E_3))$$

and substitution will be:

$$\begin{aligned} & S'(B' E_1)([x_2] E_2)([x_2]([x_1] E_3)) x_2 x_1 \\ &= B' E_1 (([x_2] E_2) x_2) (([x_2]([x_1] E_3)) x_2) x_1 \\ &= B' E_1 E_2 ([x_1] E_3) x_1 \\ &= E_1 E_2 (([x_1] E_3) x_1) \\ &= E_1 E_2 E_3 \end{aligned}$$

where E_1 is constant, E_2 constant with respect to x_1 and E_3 constant with respect to neither x_1 or x_2 .

What of the case when only E_3 used x_2 ; will S' be replaced by B' ? The answer to this is no, because now E_1 and E_2 are both totally constant and consequently only B will be used, as in the original rule:

$$[x_2]([x_1](E_1 E_2 E_3)) \Rightarrow B(B(E_1 E_2))([x_2]([x_1] E_3))$$

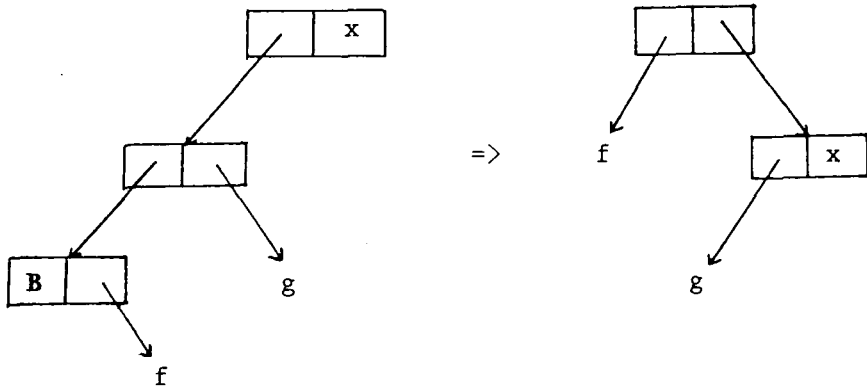
Both combinators introduced are B because the substitution of x_1 and x_2 only effects E_3 .

2.3. Graph Reduction

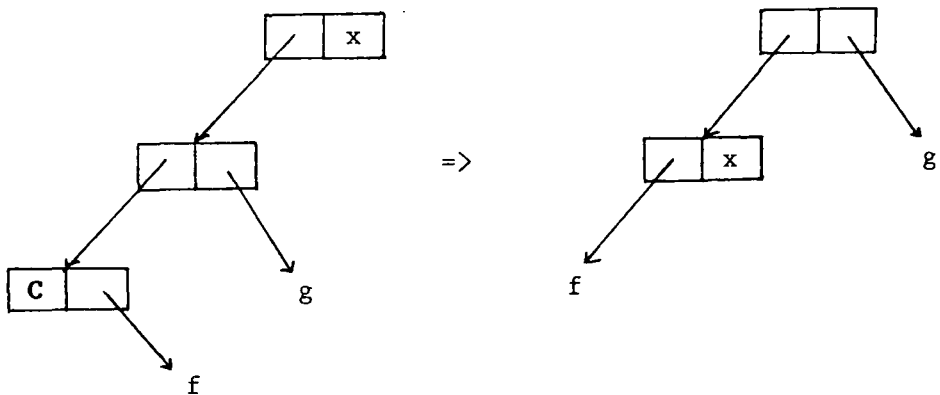
This section completes the description of graph reduction started in Chapter Five. The description includes all the graph manipulation rules used for the combinators mentioned above.

The B and C Combinators

The B and C combinators will have a similar result to S except that only f or g is applied to x. The operation of B and C are illustrated below:



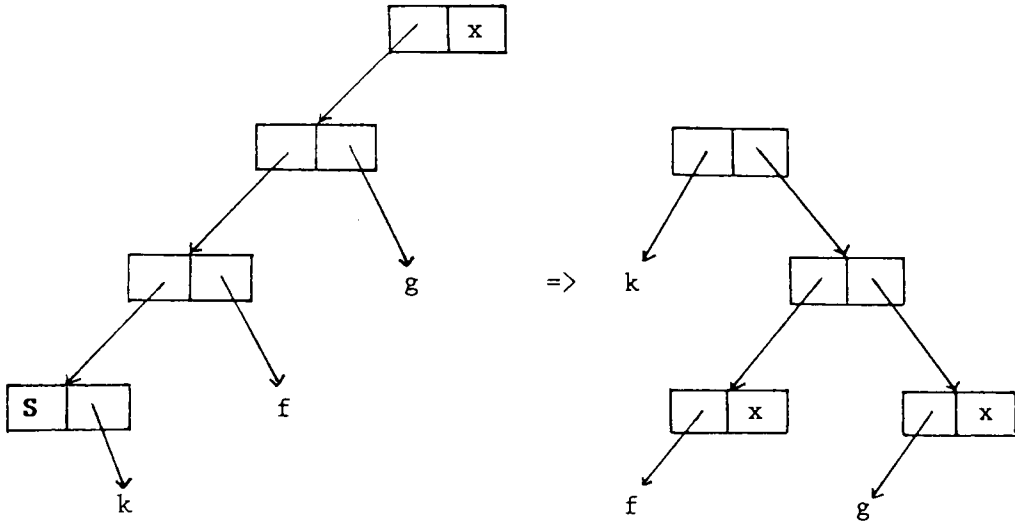
The operation of B.



The operation of C.

S' Combinator

The operation of **S'** may be illustrated by the graph:



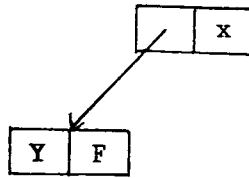
In the above graph of **S'** the top cell is modified to reflect to the reduction of **S'**, the result of which is $k(fx)(gx)$. The combinators **B'** and **C'** will have similar results.

Y Combinator

The reader will recall that the **Y** combinator is used to represent recursion because it returns the fixpoint of the recursive function. A fixpoint represents recursion without introducing a cyclic structure. The most obvious way to implement recursion is to use graphs whose structure reflects the expanding fixpoint expression given earlier:

$$\begin{aligned}
 \mathbf{YF} &= \mathbf{F(YF)} \\
 &= \mathbf{F(F(YF))} \\
 &\text{etc.}
 \end{aligned}$$

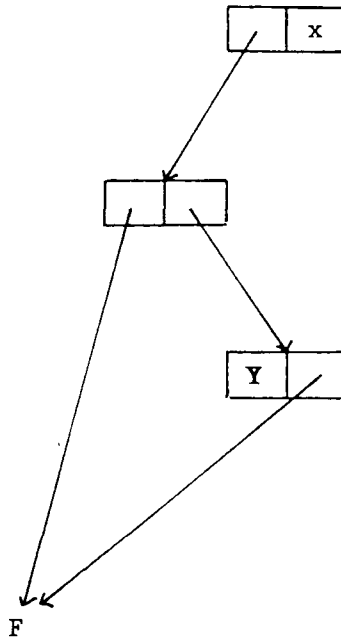
The graph will start by applying the fixpoint of a function to its argument, $(\mathbf{YF})x$, because the fixpoint will produce the desired result from **x**:



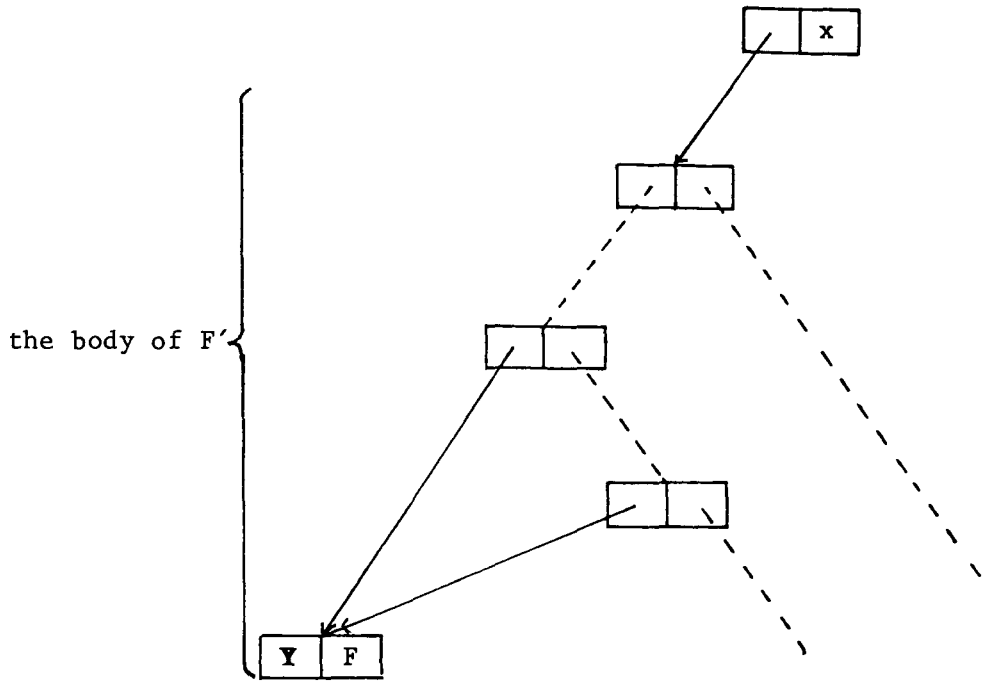
First **Y** is reduced using the reduction rule:

$$\mathbf{YF} = F(\mathbf{YF})$$

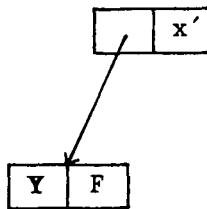
giving the graph which represents the expression $(F(\mathbf{YF}))x$:



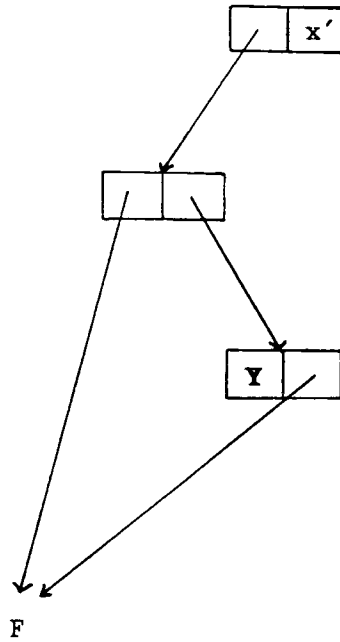
Next the pointer to **YF** is distributed into **F**'s body using the combinators generated when the function **f** was turned into the functional **F**. The combinators are generated by abstracting **f** from its own body. This generates **F'**, the function referred to in Section 2.2.



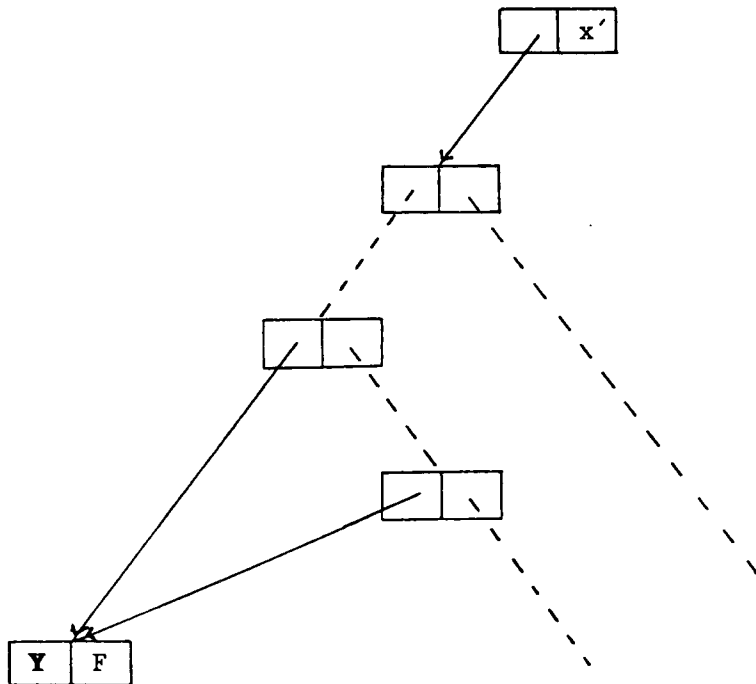
Now F' is applied to x , and x is bound into the function body. During the reduction of F' a recursive call could be made. The recursive call will have the format:



The variable x' is the argument of the recursive call. When the recursive call is made will Y be reduced, giving:

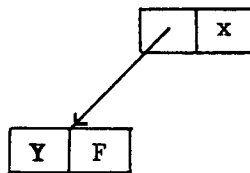


So YF is distributed into F again, and the combinators will therefore take a new copy of F .



Now the parameter of the recursive call is distributed and another recursive call made if necessary. The whole process is repeated, gradually building up a tree of recursively call Fs, until no more recursive calls are necessary. The graph will than be reduced to give the final result.

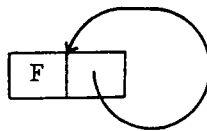
An alternative and more efficient method of implementing recursion involves the use of cyclic structures[69]. This method recreates the original cyclic references of the recursive function f. Initially the graph will have the same structure as before



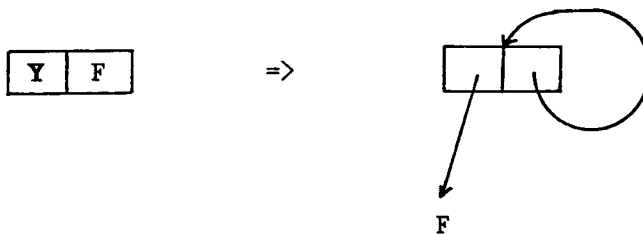
but its reduction will produce a different result. The result of reducing **YF** reflects the reduction rule for **Y**, namely:

$$YF = F(YF)$$

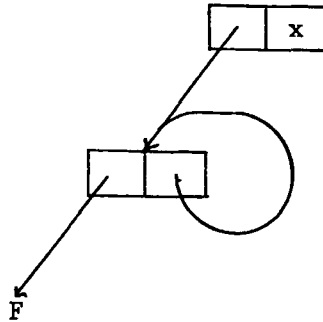
If the cyclic reference is replaced by a cyclic pointer, we have:



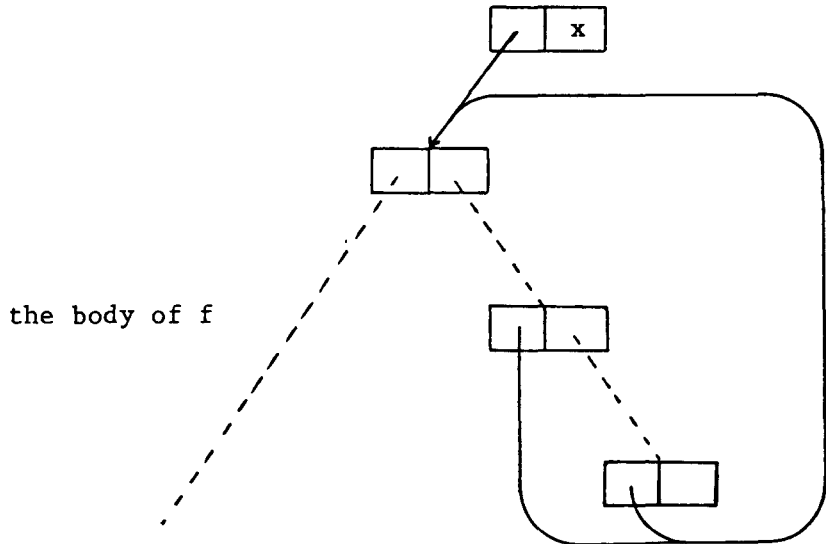
The reduction of **YF** is therefore:



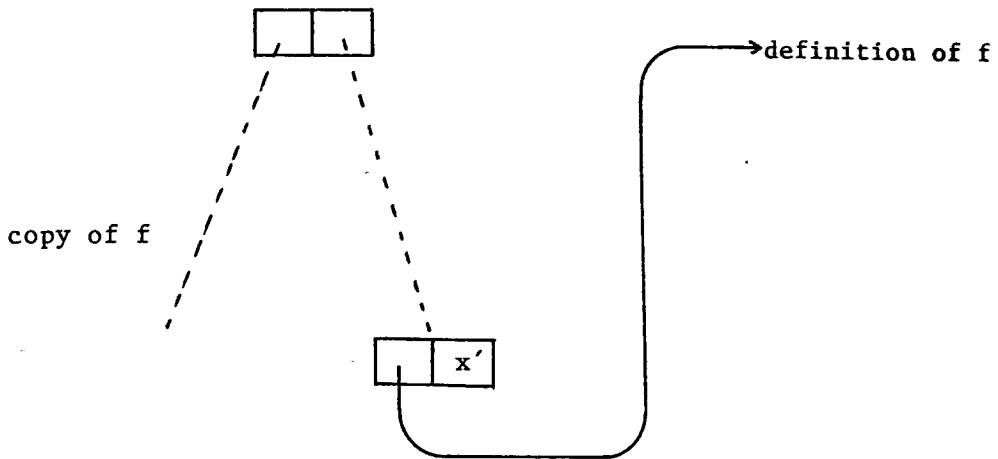
Thus the reduction of $(YF)x$ produces:



The combinators produced when f was turned into the functional F will now distribute the cyclic pointer to the cells where the recursive calls were made. The result is f , the original recursive function:



Whenever f is applied to an argument, a copy of the body will be taken by the S combinators which distribute the argument value. Each copy of f produced will retain the pointer back to the definition of f .



A recursive call will point back to the definition of f and the reduction of the call will result in another copy of f being taken; the whole process is repeated as many times as necessary.

The cyclic representation of recursion is more efficient because it does not repeatedly bind YF into the function body. In addition the cyclic structure allows infinite data structures to be represented in finite space. For example:

`ones = cons(1,ones)`

will be represented by the graph:

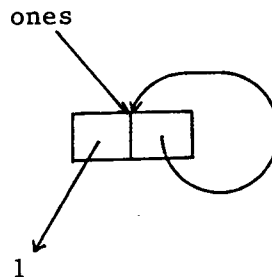


Figure 2.2: Cyclic representation of the function `ones`.

This form of graph however does introduce the problem of garbage collecting a cyclic list. This can only be accomplished by the mark-scan technique, which may not be ideal for a parallel architecture. The

first scheme for implementing recursion does not suffer from this drawback as it does not rely on cyclic graph structures. A full discussion of this topic is beyond the scope of the work reported here.

APPENDIX THREE
FUNCTIONAL LANGUAGE IMPLEMENTATION

This appendix describes the modifications made to the emulator to enable it to support graph reduction.

3.1. Instruction Format

The following combinators are implemented by the emulator:

I, K, S, B, C, S1, B1, C1

The last three correspond to **S'**, **B'** and **C'**.

Each of the combinators operates as described in Chapter Five, and uses the pure reduction scheme outlined in Chapter Six. As before the lower numbered slots are used for input arguments, and the higher numbered slots for output arguments.

The format of all the other instructions used by reduction remain the unchanged form that described in Chapter Four. Instructions are, however, allowed to have input arguments of type spare, although these need not be specified in the program's source. Any missing arguments in the source of an instruction are assumed to be of type spare. This allows the graph structure to be built using instructions with the format shown in Chapter Six. For example:

```
1: apply, prop 2, 1
2: add, 2
```

3.2. Program Format

The program format is the same as described in Appendix One, but four rules must be followed when writing combinator programs. The first rule concerns the values of an instruction's flags: all instructions are retained after their execution in order to implement lazy evaluation. All instructions must therefore have the "r" flag set. All instructions will receive demands, except the instruction which propagates the demand that starts the program's execution. All but one instruction therefore have the "d" flag set.

The second rule concerns the count field. Since no "unk" arguments should be specified by the program, and no instructions receive control tokens, the count field of every instruction should be zero.

The third rule restricts argument to be of type "spare", "prop", "pm" or "litv". These are the only types that should be necessary to write a combinator program.

The fourth rule concerns the use of instructions such as "mul" as operands. In the example of factorial given in Chapter Five there were several nodes of the form:

S	*
---	---

These nodes cannot be used in a program for the emulated architecture because it is not possible to have an argument of type opcode. Instead if the argument of an instruction should be of type "opcode", then the argument must propagate a demand to an instruction containing this opcode. The node above will therefore be written:

0: dr/0/S,prop(/1/)

1: dr/0/mul

3.3. Instruction Execution Cycle

When an instruction executes it inspects its input arguments to see if they all have values. If some input arguments are of type spare the instruction returns its own address as the result of the demand. The source of the demand should be an apply instruction which will be used as explained in Chapter Six. If all the instruction's arguments have values the instruction will be reduced to its result, which is returned to the source of the demand.

3.4. Garbage Collection

A mark-scan garbage collector has been added to the emulator, in which the mark phase of the algorithm starts with the execution queue. The mark phase proceeds by marking every instruction on the queue, and then follows each pointer from the instructions on the queue and marks the locations addressed. The pointers from these instructions are then followed and the locations referred to are marked, and so on. If an argument is of type "am", "sig" or "prop" then although the address in the argument points to an AM location, it also refers to a DM location, the one that will be used if AM does not contain the instruction addressed. If an AM location referred to by any of these types is marked, the the corresponding DM location is also marked.

The scan phase passes through both AM and PM and places all unmarked cells of the free chain.

3.5. Example Program

The program shown below is the factorial example given in Chapter Five. The program is followed by an abbreviated trace; only the executing instructions are shown.

```
    {print factorial(n)}
0  r/0/print,prop(/2/)

    {read n when n is demanded}
1  dr/0/read

    {apply factorial to n, cell forms the root of the outer-most S}
2  dr/0/apply,prop(/3/),prop(/1/)

    {second cell of the outer-most S}
3  dr/0/apply,prop(/4/),prop(/11/)

    {the outer-most S}
4  dr/0/S,prop(/5/)

    {second cell of C, contains 'then' arm of the conditional}
5  dr/0/apply,prop(/6/),litv/1

    {distributes n to the conditional}
6  dr/0/C,prop(/7/)

    {second cell of the conditional}
7  dr/0/apply,prop(/8/),prop(/10/)

    {distributes n to comparison with 0}
8  dr/0/B,prop(/9/)

9  dr/0/cond

    {is n=0?}
10 dr/0/eq,litv/0

    {the else arm of the conditional}
11 dr/0/apply,prop(/12/),prop(/14/)

    {distributes n into the else arm}
12 dr/0/S,prop(/13/)

13 dr/0/mul

    {second cell of B}
14 dr/0/apply,prop(/15/),prop(/16/)

    {distributes n and contains the cyclic pointer to factorial}
15 dr/0/B,prop(/3/)

    {calculate n-1}
16 dr/0/apply,prop(/17/),litv/1
```

{distribute n into 'n-1'}
17 dr/0/C,prop/(/18/)

18 dr/0/sub

trace of combinator factorial

```
{print factorial(n)}
(0/0/1):dr/0/print,prop/(/2/1)

{apply factorial to n}
(0/2/1):dr/0/apply,prop/(/3/1),prop/(/1/1),am/(0/0/1)

{bind n into factorial}
(0/3/1):dr/0/apply,prop/(/4/1),prop/(/11/1),am/(0/2/1)
(0/4/1):dr/0/S,prop/(/5/1),spare,spare,am/(0/3/1)
(0/3/1):dr/0/apply,am/(0/4/1),prop/(/11/1),am/(0/2/1)
(0/2/1):dr/0/apply,am/(0/3/1),prop/(/1/1),am/(0/0/1)
(0/19/1):dr/0/apply,prop/(/5/1),prop/(/1/1),am/(0/2/1)
(0/5/1):dr/0/apply,prop/(/6/1),litv/1,am/(0/19/1)
(0/6/1):dr/0/C,prop/(/7/1),spare,spare,am/(0/5/1)
(0/5/1):dr/0/apply,am/(0/6/1),litv/1,am/(0/19/1)
(0/19/1):dr/0/apply,am/(0/5/1),prop/(/1/1),am/(0/2/1)
(0/21/1):dr/0/apply,prop/(/7/1),prop/(/1/1),am/(0/19/1)
(0/7/1):dr/0/apply,prop/(/8/1),prop/(/10/1),am/(0/21/1)
(0/8/1):dr/0/B,prop/(/9/1),spare,spare,am/(0/7/1)
(0/7/1):dr/0/apply,am/(0/8/1),prop/(/10/1),am/(0/21/1)
(0/21/1):dr/0/apply,am/(0/7/1),prop/(/1/1),am/(0/19/1)

{select result of factorial according to the result of n=0?}
(0/9/1):dr/0/cond,spare,spare,spare,am/(0/21/1)
(0/21/1):dr/0/apply,am/(0/9/1),prop/(0/22/1),am/(0/19/1)
(0/19/1):dr/0/apply,am/(0/21/1),litv/1,am/(0/2/1)
(0/2/1):dr/0/apply,am/(0/19/1),prop/(0/20/1),am/(0/0/1)
(0/22/1):dr/0/apply,prop/(/10/1),prop/(/1/1),am/(0/2/1)

{is n=0?}
(0/10/1):dr/0/eq,litv/0,spare,am/(0/22/1)
(0/22/1):dr/0/apply,am/(0/10/1),prop/(/1/1),am/(0/2/1)

{need n so read it}
(0/1/1):dr/0/read,am/(0/22/2)
```

```
integer?<      1

{is n=0?}
(0/22/1):dr/0/eq,litv/0,litv/1,am/(0/2/1)

{select factorial result. n=1 so result = n*factorial(n-1)}
(0/2/1):dr/0/cond,litv/0,litv/1,prop/(0/20/1),am/(0/0/1)

(0/20/1):dr/0/apply,prop/(/11/1),prop/(/1/1),am/(0/2/3)

(0/11/1):dr/0/apply,prop/(/12/1),prop/(/14/1),am/(0/20/1)

{distribute n into code for n*factorial(n-1)}
(0/12/1):dr/0/S,prop/(/13/1),spare,spare,am/(0/11/1)

(0/11/1):dr/0/apply,am/(0/12/1),prop/(/14/1),am/(0/20/1)

(0/20/1):dr/0/apply,am/(0/11/1),prop/(/1/1),am/(0/2/3)

(0/23/1):dr/0/apply,prop/(/13/1),prop/(/1/1),am/(0/20/1)

{n*factorial(n-1)}
(0/13/1):dr/0/mul,spare,spare,am/(0/23/1)

(0/23/1):dr/0/apply,am/(0/13/1),prop/(/1/1),am/(0/20/1)

(0/20/1):dr/0/apply,am/(0/23/1),prop/(0/24/1),am/(0/2/3)

(0/1/1):dr/0/I,litv/1,am/(0/20/1)

(0/24/1):dr/0/apply,prop/(/14/1),prop/(/1/1),am/(0/20/2)

(0/14/1):dr/0/apply,prop/(/15/1),prop/(/16/1),am/(0/24/1)

(0/15/1):dr/0/B,prop/(/3/1),spare,spare,am/(0/14/1)

(0/14/1):dr/0/apply,am/(0/15/1),prop/(/16/1),am/(0/24/1)

(0/24/1):dr/0/apply,am/(0/14/1),prop/(/1/1),am/(0/20/2)

{distribute n' (=n-1) into recursively called factorial}
(0/3/1):dr/0/apply,prop/(/4/1),prop/(/11/1),am/(0/24/1)

(0/4/1):dr/0/S,prop/(/5/1),spare,spare,am/(0/3/1)

(0/3/1):dr/0/apply,am/(0/4/1),prop/(/11/1),am/(0/24/1)

(0/24/1):dr/0/apply,am/(0/3/1),prop/(0/25/1),am/(0/20/2)

(0/26/1):dr/0/apply,prop/(/5/1),prop/(0/25/1),am/(0/24/1)

(0/5/1):dr/0/apply,prop/(/6/1),litv/1,am/(0/26/1)

(0/6/1):dr/0/C,prop/(/7/1),spare,spare,am/(0/5/1)

(0/5/1):dr/0/apply,am/(0/6/1),litv/1,am/(0/26/1)
```

```
(0/26/1):dr/0/apply,am/(0/5/1),prop/(0/25/1),am/(0/24/1)
(0/28/1):dr/0/apply,prop/(/7/1),prop/(0/25/1),am/(0/26/1)
(0/7/1):dr/0/apply,prop/(/8/1),prop/(/10/1),am/(0/28/1)
(0/8/1):dr/0/B,prop/(/9/1),spare,spare,am/(0/7/1)
(0/7/1):dr/0/apply,am/(0/8/1),prop/(/10/1),am/(0/28/1)
(0/28/1):dr/0/apply,am/(0/7/1),prop/(0/25/1),am/(0/26/1)
{select result of factorial(n') according to result of n'=0?}
(0/9/1):dr/0/cond,spare,spare,spare,am/(0/28/1)
(0/28/1):dr/0/apply,am/(0/9/1),prop/(0/29/1),am/(0/26/1)
(0/26/1):dr/0/apply,am/(0/28/1),litv/1,am/(0/24/1)
(0/24/1):dr/0/apply,am/(0/26/1),prop/(0/27/1),am/(0/20/2)
(0/29/1):dr/0/apply,prop/(/10/1),prop/(0/25/1),am/(0/24/1)
{is n'=0?, now must calculate n' (=n-1)}
(0/10/1):dr/0/eq,litv/0,spare,am/(0/29/1)
(0/29/1):dr/0/apply,am/(0/10/1),prop/(0/25/1),am/(0/24/1)
(0/25/1):dr/0/apply,prop/(/16/1),prop/(/1/1),am/(0/29/2)
(0/16/1):dr/0/apply,prop/(/17/1),litv/1,am/(0/25/1)
(0/17/1):dr/0/C,prop/(/18/1),spare,spare,am/(0/16/1)
(0/16/1):dr/0/apply,am/(0/17/1),litv/1,am/(0/25/1)
(0/25/1):dr/0/apply,am/(0/16/1),prop/(/1/1),am/(0/29/2)
(0/30/1):dr/0/apply,prop/(/18/1),prop/(/1/1),am/(0/25/1)
(0/18/1):dr/0/sub,spare,spare,am/(0/30/1)
(0/30/1):dr/0/apply,am/(0/18/1),prop/(/1/1),am/(0/25/1)
(0/25/1):dr/0/apply,am/(0/30/1),litv/1,am/(0/29/2)
(0/1/1):dr/0/I,litv/1,am/(0/25/1)
(0/25/1):dr/0/sub,litv/1,litv/1,am/(0/29/2)
{have calculated n', so is the value=0?}
(0/29/1):dr/0/eq,litv/0,litv/0,am/(0/24/1)
{select result, n'=0 so result =1}
(0/24/1):dr/0/cond,litv/1,litv/1,prop/(0/27/1),am/(0/20/2)
```

```
{factorial(n)=n*factorial(n-1)}  
(0/20/1):dr/0/mul,litv/1,litv/1,am/(0/2/3)  
  
{now return result selected by the first call of factorial,-  
ie n*factorial(n-1)}  
(0/2/1):dr/0/cond,litv/0,litv/1,litv/1,am/(0/0/1)  
  
{print factorial(n)}  
(0/0/1):r/0/print,litv/1  
  
{factorial(1)}  
  1
```


APPENDIX FOUR
LOGIC LANGUAGE IMPLEMENTATION

In Chapter Eight an outline of the implementation of OR-parallelism was given, here the implementation details are described.

4.1. Activation Record Format

An activation record has the format shown below, each line represents the contents of one location:

negated flag
activation record length
next goal in this clause
calling clause's process
· · variables · ·
· · actual parameters · ·

Figure 4.1: Activation record for logic.

negated flag

The value of the first argument slot in this location is set to one if the clause is called by a negated goal (an ngoal instruction), and zero otherwise. The second argument slot contains the process number of the activation record belonging the nearest clause in the branch which executed a negated goal (see Chapter Eight for a description of the implementation of negation).

activation record length

The first argument of this location is a literal whose value is the number of locations in the activation record, excluding the four locations which form the head.

next goal in this clause

The first argument slot in this location holds the address of the next instruction to be obeyed in the clause to which the activation record belongs.

calling clause's process

This location holds the process number of the activation record of the calling goal.

variables

These are the variables of the clause. Each value is held in the first argument of the location, if the value is undefined the argument type is unk.

actual parameters

These are the actual parameters copied into the activation record by the unification algorithm. An actual parameter will have the parameter's value as its first argument, and the address of the parameter in the caller's activation record as its second argument.

4.2. Instruction Source Format

Instructions have the same format as described in Appendix One. Each instruction specifies the flags "r" and "d", neither of which are set for logic instructions, followed by the count, which will be one because each instruction expects one signal. The next item on the line is the instruction mnemonic, followed by the instruction's arguments.

In the case of a goal instruction the first argument holds a literal value which is the "procedure" index for the relation to be called. The following arguments of the instruction are the actual parameters of the goal. Literal values are specified in the usual way while indexes into the activation record are specified as PM addresses, where the location is equal to the index. For example, a goal instruction which calls a clause whose "procedure" index is 2, and with three parameters the first two of which are literals, and the last an index of two into the activation record, is be written:

/1/goal,litv/2,litv/1,litv/10,PM(/2/)

A clause instruction will hold the length (excluding the head) of the activation record it needs in its first argument. The remaining arguments will be the formal parameters, and have the same format as the parameters of a goal. For example a clause whose activation record is four locations long and with two formal parameters, one a PM address and the other a literal value, will have the form:

/1/claus,litv/4,pm(/5/),litv/0

The comparisons and arithmetic instructions may have either literal values or activation record indexes as their arguments.

4.3. Program Source Format

The format of a logic program has already been described in Chapter Eight, but in addition the first line of the program is a number which specifies the length of the activation record for the process which will obey the user's question.

4.4. Instruction Execution Cycle

The arithmetic and comparative instructions operate by carrying out their functions and passing control to the next instruction in the clause if they were successful. Failure is used to provide negation in the way described in Chapter Eight. Should too few parameters be present the instruction will abort, which will in turn cause the processor to stop obeying the program.

The five remaining instructions implement relation calling and are described in detail below:

goal

This instruction calls the relation which corresponds to the index held by its first argument. It sets up a process for each clause in the called relation and creates the corresponding activation record. Next the goal instruction sets up the four locations of the head in each new environment, and then signals the clause instruction to execute by sending it a control token. All clauses of the relation will therefore execute in parallel. At this point the head of each will be:

negated = 0

length = length of activation record specified by the first argument
of the clause instruction in the called clause +
the number of parameters passed to it.

next goal for clause = the address of the first goal in clause

calling clause's process = the process number of the goal

ngoal

Performs in the same way as the goal instruction but sets the value of the negated flag to one and stores its own address in the second argument.

fail

This instruction is executed as part of the implementation of negation and fails the clause it belongs to. The implementation of negation is explained in Chapter Eight.

claus

The `claus` instruction is the first to be executed in a clause. Its prime function is to perform the unification of the formal and actual parameters. To achieve this it looks at its own parameters and those of the calling goal. These may be found by referring to the "calling clause's process" entry in the head of the new clause's activation record. When performing the unification a copy of all values passed as actual parameters are placed the activation record.

endc

This instructions primary task is to copy the calling clause's activation record into current clause's activation record. This task will only be carried out if the clause which has terminated has a parent, signified by the value field of the "next goal in caller" location being nonzero. If there is no parent all the variables of the activation record are printed since they will contain the answer to the users question. The format of the output is:

```
proved
      1:v
      1:v
```

where "1" is the location in the activation record where the result resides, and v is the result value.

If the clause is negated, because the negated flag is set, then the `endc` instruction will force the fail instruction in the caller's clause to execute. This is done by moving the "next goal in clause" pointer of the caller back one location so that it now points to the fail instruction instead of the goal it would normally execute. See Chapter Eight

for a description of the implementation of negation.

Once the activation record is set up the next goal in the clause is signaled, and the "next goal in clause" pointer moved on.

4.5. Program Execution

Program execution is started by the first goal in the users question, it is the only instruction in the program which has a count of zero, and therefore is immediately executable. Execution continues until there are no instructions on the processor queue or an instruction is aborted.

4.6. Garbage Collection

Garbage collection uses the reference count strategy outlined in Chapter Seven. As each clause terminates the garbage collector is invoked and passes up the branch of the tree to which the clause belongs, as it does so it deletes all redundant activation records.

4.7. Example Program

```
parent(1,2).
parent(2,3).

descendant(X,Y) :- parent(X,Y).
descendant(X,Y) :- parent(X,Z),descendant(Z,Y).

question = descendant(A,B).
```

The program above is coded as shown in the example below. Numbers are used to represent the names in the example as they are the only type of literal permitted by the implementation. The program is followed by

an abbreviated trace in which only the executing instructions are shown.

```
    {two locations in question's activation record}
    2

    {call descendant relation}
0    /0/goal,litv/2,pm/(/5/),pm/(/6/)

    {end of question}
1    /1/encd,litv/0

{parent relation}
#

    {'1' is the parent of '2'}
500  /1/claus,litv/2,litv/1,litv/2

    {end of the first clause of the descendant relation}
501  /1/encd

    {'2' is the parent of '3'}
502  /1/claus,litv/2,litv/2,litv/3

    {end of the second clause of the descendant relation}
503  /1/encd,litv/0

{the descendant relation}
#

    {the first clause descendant(X,Y):-parent(X,Y)}
1000 /1/claus,litv/4,pm/(/5/),pm/(/6/)

    {call parent relation}
1001 /1/goal,litv/1,pm/(/5/),pm/(/6/)

    {end of the first clause}
1002 /1/encd

    { {second clause: descendant(X,Y):-parent(X,Z),descendant(Z,Y)}
1003 /1/claus,litv/5,pm/(/5/),pm/(/6/)

    {call parent relation}
1004 /1/goal,litv/1,pm/(/5/),pm/(/7/)

    {call descendant relation}
1005 /1/goal,litv/2,pm/(/7/),pm/(/6/)

    {end of descendant relation}
1006 /1/encd,litv/0
```


trace of family tree

{the question}

(0/0/1):/0/goal,litv/2,pm(/5/1),pm(/6/1)

{the first clause in the descendant relation}

(1/1000/1):/0/claus,litv/4,pm(/5/1),pm(/6/1)

{the second clause in the descendant relation}

(2/1003/1):/0/claus,litv/5,pm(/5/1),pm(/6/1)

{the call of the parent relation in the first descendant clause}

(1/1001/1):/0/goal,litv/1,pm(/5/1),pm(/6/1)

{the call of the parent clause in the second descendant clause}

(2/1004/1):/0/goal,litv/1,pm(/5/1),pm(/7/1)

{the first clause from the parent relation}

(3/500/1):/0/claus,litv/2,litv/1,litv/2

{the second clause of the parent relation}

(4/502/1):/0/claus,litv/2,litv/2,litv/3

{the first clause of the parent relation}

(5/500/1):/0/claus,litv/2,litv/1,litv/2

{the second clause of the parent relation}

(6/502/1):/0/claus,litv/2,litv/2,litv/3

{the end of one clause of parent, copy down caller}

(3/501/1):/0/encd

{the end of the other clause of parent, copy down caller}

(4/503/1):/0/encd,litv/0

{the end of clause one of parent, copy down caller}

(5/501/1):/0/encd

{the end of clause two of parent, copy down caller}

(6/503/1):/0/encd,litv/0

{the end of the first clause of descendant, copy down caller}

(3/1002/1):/0/encd

{the end of the first clause in descendant, copy down caller}

(4/1002/1):/0/encd

{the second goal of the second clause of descendant}

(5/1005/1):/0/goal,litv/2,pm(/7/1),pm(/6/1)

{the second goal of the second clause of descendant}

(6/1005/1):/0/goal,litv/2,pm(/7/1),pm(/6/1)

{the end of the question, print result}
(3/1/1):/0/encd,litv/0

{a result: '2' is the descendant of '1'}
proved
 5:litv/1
 6:litv/2

{the end of the question, print a result}
(4/1/1):/0/encd,litv/0

{a result: '3' is the descendant of '2'}
proved
 5:litv/2
 6:litv/3

{the first clause of descendants, started by recursive call}
(8/1000/1):/0/claus,litv/4,pm(/5/1),pm(/6/1)

{the second clause of descendant started by recursive call}
(9/1003/1):/0/claus,litv/5,pm(/5/1),pm(/6/1)

{the first clause of descendant started by recursive call}
(10/1000/1):/0/claus,litv/4,pm(/5/1),pm(/6/1)

{the second clause of descendant started by the recursive call}
(11/1003/1):/0/claus,litv/5,pm(/5/1),pm(/6/1)

{the call of the parent relation made in clause 1 of descendant}
(8/1001/1):/0/goal,litv/1,pm(/5/1),pm(/6/1)

{the call of the parent relation made in clause 2 of descendant}
(9/1004/1):/0/goal,litv/1,pm(/5/1),pm(/7/1)

{the call on parent made by the clause 1 of descendant}
(10/1001/1):/0/goal,litv/1,pm(/5/1),pm(/6/1)

{the call on parent made by clause 1 of descendants}
(11/1004/1):/0/goal,litv/1,pm(/5/1),pm(/7/1)

{the first clause of the parent relation}
(12/500/1):/0/claus,litv/2,litv/1,litv/2

{the second clause of the parent relation}
(13/502/1):/0/claus,litv/2,litv/2,litv/3

{the first clause of the parent relation}
(14/500/1):/0/claus,litv/2,litv/1,litv/2

{the second clause of the parent relation}
(15/502/1):/0/claus,litv/2,litv/2,litv/3

```
{the first clause of the parent relation}
(16/500/1):/0/claus,litv/2,litv/1,litv/2

{the second clause of the parent relation}
(17/502/1):/0/claus,litv/2,litv/2,litv/3

{the first clause of the parent relation}
(18/500/1):/0/claus,litv/2,litv/1,litv/2

{the send clause of the parent relation}
(19/502/1):/0/claus,litv/2,litv/2,litv/3

{the end of the clause 2 of parent, copy down the caller}
(13/503/1):/0/encd,litv/0

{the end of the second clause of parent, copy down the caller}
(15/503/1):/0/encd,litv/0

{the end of the first clause of descendant, copy down the caller}
(13/1002/1):/0/encd

{the second goal of clause 2 of descendant}
(15/1005/1):/0/goal,litv/2,pm(/7/1),pm(/6/1)

{the end of the second clause of descendant}
(13/1006/1):/0/encd,litv/0

{first clause of descendant started by second recursive call}
(20/1000/1):/0/claus,litv/4,pm(/5/1),pm(/6/1)

{the second clause of the descendant clause called by the second-
recursive call}
(21/1003/1):/0/claus,litv/5,pm(/5/1),pm(/6/1)

{end of the question, print a result}
(13/1/1):/0/encd,litv/0

{a result: '3' is the descendant of '1'}
proved
    5:litv/1
    6:litv/3

{the first goal of the first clause of the descendant relation}
(20/1001/1):/0/goal,litv/1,pm(/5/1),pm(/6/1)

{the call on parent made form the second clause of descendant}
(21/1004/1):/0/goal,litv/1,pm(/5/1),pm(/7/1)

{the first clause of the parent relation}
(22/500/1):/0/claus,litv/2,litv/1,litv/2

{the second clause of the parent relation}
(23/502/1):/0/claus,litv/2,litv/2,litv/3
```

{the first clause of the parent relation}
(24/500/1):/0/claus,litv/2,litv/1,litv/2

{the second clause of the parent relation}
(25/502/1):/0/claus,litv/2,litv/2,litv/3

{none of the unifications with the parent relation succeeded, so the-
program's executions stops}

APPENDIX FIVE
COMBINED FUNCTIONAL AND LOGIC LANGUAGE IMPLEMENTATION

This appendix describes some of the details of the proposed implementation of the combined functional and logic scheme.

5.1. Structure of Activation Records

Both logic programs and functional programs will share the same form of activation record:

Flags
Number of Arguments
Number of Locals
Number of Descendants
Negated Pointer
Arguments
Local Variables

Figure 5.1: Combined logic/functional activation record.

The flags consist of the following:

- l: true if this is a logic activation record. Causes a clause or function which has just terminated to copy down its caller.
- n: true if the call which created this activation record is negated.
- a: true if the activation record belongs to a function which is obeying an "all" instruction.
- i: true if the activation record is to inherit values from the expression it qualifies.

The other enties in the activation record are explained when necessary.

5.2. Format of Instructions

Each instruction has the format:

f	r	p	o	c	opcode	... arguments ...
---	---	---	---	---	--------	-------------------

- f: true if the instruction is part of a function. Used to control the behaviour of arithmetic instructions.
- r: If true the instruction is retained after it has been executed.
- p: true if the instruction is to be obeyed in parallel. Each demand starts a parallel execution of the instruction. This allows the instruction to deal with several demands in parallel, one per copy.
- o: if true the instruction remains in its defining process when executed. This is so that the instruction can refer to its defining process during its execution.

c: the count of arguments in the instruction

opcode: the operation code of the instruction

arguments: provide the values upon which the instruction will operate.

Argument Format

There are two types of argument, a basic argument and an instruction argument. The first has the format:

type	address or value
------	------------------

An address has the same format as for the general-purpose architecture described in Chapter Four, it comprises of a process number, a location number and an argument slot number. The values of all of these are integers.

The type of a basic argument may be:

spare: the argument is not in use.

unk: the argument has an unknown value. This is used to receive the results of a demand.

closure: The address of a closure. Gives the address (process and location) for the instruction at the root of the function and the process number of the activation record.

forward: forwards all demand tokens received by the instruction to the instruction whose address in this argument.

- im: instruction memory address.
- prop: gives the address to which a demand must be propagated.
- name: the name of a function or relation. Consists of the actual name, an integer, and a pointer to the activation record location that points to the program which contains the named object.
- dm: data memory address
- sm: structure memory address

An instruction argument may be a basic argument, or a more complex one. They have the format shown below.

type	instruction argument's data	basic argument
------	-----------------------------	----------------

The types provided are:

- litv: a literal value. The basic argument is the value.
- basic: the argument is a basic argument. The type of the basic argument is to be used to determine which action is to be carried out for this instruction argument.
- lpar: the argument is a formal logic parameter. The basic argument holds the value of the logic parameter. The remainder of the instruction argument points to the actual parameter which this parameter is unified with.

5.3. Token Format

A token will contain the fields:

DM address of the sender	basic argument
--------------------------	----------------

A token is therefore able to pass values between instructions, no just signify that a demand has been propagated.

5.4. Instruction Execution

An instruction is only executed when it has received at least one demand. When the demand arrives the instruction will be placed on the queue for the processor. When obeying an instruction the processor will attempt to obtain values for all the arguments necessary for its execution. The operations necessary to deal with each argument type are described below. The instruction argument type will be "basic" so the types listed below are those of the basic argument held by the instruction argument:

DM address: The address gives the process and location for a value. The contents of the location will be an argument which should be dealt with in the manner appropriate to its type.

prop: This will result in a demand being propagated to the instruction specified by the code address. The process number of the address will identify the definition process for the code, and the location will identify the particular instruction. The destination can therefore reside in a different process to the instruction that propagated the

demand. When the demand reaches the destination a copy of the instruction is placed in the same process as the sender providing the "o" flag is not set. The return address for the demand will an argument of type im whose address points to the caller. The argument is placed in the first spare argument slot.

forward: When the instruction is executed it performs whatever task is required of it, but also sends a copy of the demand it received to the instruction whose address is given in the forward argument. In all other respects the demand is like that produced by an argument of type prop. The demand therefore appears to have come from the originator, not the current instruction.

closure: this argument type will usually be used in a call instruction and identifies the function to be called together with its activation record.

IM address: This is used as the return address for a demand and may point to any instruction in any process.

name: The name argument type operates in much the same way as an closure type does. The difference is that an closure argument points directly at an function whereas a name refers to it by name. This name must be dereferenced to give the relation or function. The name section of the argument identifies the function or relation in the program. A demand is propagated to the program containing the name of the function or relation.

litv: This argument type simply holds a value. The value is itself an argument.

Having obtained all necessary arguments the instruction will carry out its task and dispatch the result to the instructions which appear in its output arguments. When completed it will be converted to hold the result and retained if the "r" (retain) flag is set, or otherwise deleted.

5.5. Instruction Opcodes

The following opcodes are provided by the architecture, each description indicates what flags will usually be set. The "f" flag is always set if the instruction is part of a function.

prog: program instruction. This instruction is the first in the process which represents a collection of functions or relations. It has two arguments.

- 1) the address of the last instruction in the process.
- 2) the address of the first link instruction in the program held in a forward typed argument.

Flags: p

When it receives a demand it forwards it to the instruction pointed to by its second argument.

rel: The first instruction in a process which represents a relation.
It has two arguments.

- 1) the address of the last instruction in the process.
- 2) the address of the first rlink instruction in the relation.
held in a forward type argument.

Flags: p,o

When it receives a demand it forwards it to rlink instruction pointed to by the second argument.

link: A member of the sequence of pointers to relations or functions which form a program, it has three arguments.

- 1) the closure for the function or relation held in an argument of type forward.
- 2) the name of the function, relation or clause
- 3) the address of to the next link instruction, held in argument of type forward.

Flags: p,o

rlink: a relation link, a member of the sequence of links which make a relation. It has two arguments.

- 1) the closure for a clause, held in a forward type argument.
- 2) the address of the next rlink instruction held in a forward type argument

Flags: p,o

The instruction forwards any demand it receives to both arguments.

clause: The first instruction of a clause. It controls the execution of the goals in the clause and copies down its caller if all the goals are successful. Each argument points to a goal in the clause except the last one which holds the return address for the clause.

Flags: o

It creates a copy of its defining process and transfers its execution to that. The o flag is set so that the instruction initially starts executing in the defining process, but later transfers itself to the new copy.

func: The first instruction of a function. It demands a result from the instructions which form the body of the function and return it to the caller. The instruction has three arguments

1) the address of the instruction to which the demand for the result must be propagated.

2) the number of arguments expected by the function

3) the return address

Flags: o

call: This instruction calls a function or clause. It propagates a demand token containing the name of the function or relation to the prog instruction at the head of the appropriate program. The instruction has the following arguments.

1) the name of the function or clause to be called, or a closure for either.

2) the arguments of the call.

ncall: this instruction is the same as the call instruction but is intended to be used for negated goals in a logic program. It therefore sets the "n" flag in the called clauses activation record.

The following instruction are intended to provide an interface between a functional program and a logic program:

get: This instruction allows a functional program to call a goal. It has two arguments.

1) a prop argument which points to the call instruction for the goal.

2) the DM address of the location in which the goal will leave the desired result.

The instruction checks the contents of the DM location and if the value is unknown it demands the result of the goal(success or failure). When this is received the instruction gets the value returned by the goal from DM and passes it back as if it were its own result.

store: This instruction allows functional expressions to made a goal in a clause. It has two arguments:

1) a prop argument pointing to the root instruction for the functional expression.

2) the location in DM where the result is to be stored

When the store instruction is executed it propagates a demand to the root of the functional expression and awaits its result. When it is received it stores it in the location specified by its second argument. This may be accessed by the other goals in the clause.

all: This instruction is like call but is intended to be used by a functional program when it wishes to have a list of the results produced by a call on a goal. The results are represented as a relation which consists entirely of assertions.

Arithmetic and Related Opcodes

The usual collection of arithmetic operations are provided, but each has two versions. The first is intended for functional programs and demands its operands in parallel. The second is intended for use in hybrid programs and demands its arguments one at a time. Both types of instruction use the first two arguments as input values and put the result in the place specified by the third. This may either be a DM address, in which case the value is stored in the activation record, or an IM address in which case the result is placed in the specified instruction at the specified argument position. All arithmetic operations are capable of dealing with up to one unknown argument, and pro-

duce the value for it from the other two. An unknown value may either be represented by an unknown argument, or a DM location whose contents are an argument whose type is "unk". If the "f" flag is set the instruction is obeyed when it has three arguments, if it is not set the count must be four before it will be executed.

The Conditional Instruction

This instruction is only used in a functional program, it demands the result of its first argument, the predicate, and then demands the result of the second or third argument depending on its value.

Constructors

The way constructors are implemented depends on the parameter passing mechanism. If the copying pure code technique is used the constructor will reside in SM and will have the form shown below.

name	...arguments...
------	-----------------

Each argument has the same format as an instruction argument.

5.6. Assessment

The proposed scheme for implementing functional and logic languages is likely to be simpler than the architecture described in Chapter Four because it is based around one computation mechanism; demand propagation with multiple results. There will of course be more instruction opcodes to implement than on an architecture which supports only one type of language, but not twice as many. Both functional and logic languages

use the same form of code for a call, both share the arithmetic operations, and both share some instructions which are used in a function or relation definition, for example the clo instruction. The architecture described in this appendix does have the additional complexity of dereferencing names, but that complexity provides extra capabilities, and so is justified. The architecture described will therefore provide a simple unified scheme for functional and logic languages.