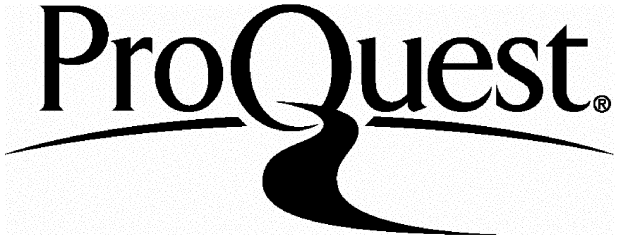
UNIVERSITY COLLEGE LONDON
DEPARTMENT OF COMPUTER SCIENCE

Developing and Measuring Parallel Rule-Based Systems in a Functional Programming Environment

Stuart Clayman

A thesis submitted for the degree of
Doctor of Philosophy
in the University of London

September 1993



Abstract

This thesis investigates the suitability of using functional programming for building parallel rule-based systems.

A functional version of the well known rule-based system OPS5 was implemented, and there is a discussion on the suitability of functional languages for both building compilers and manipulating state. Functional languages can be used to build compilers that reflect the structure of the original grammar of a language and are, therefore, very suitable. Particular attention is paid to the state requirements and the state manipulation structures of applications such as a rule-based system because, traditionally, functional languages have been considered unable to manipulate state.

From the implementation work, issues have arisen that are important for functional programming as a whole. They are in the areas of algorithms and data structures and development environments. There is a more general discussion of state and state manipulation in functional programs and how theoretical work, such as monads, can be used. Techniques for how descriptions of graph algorithms may be interpreted more abstractly to build functional graph algorithms are presented. Beyond the scope of programming, there are issues relating both to the functional language interaction with the operating system and to tools, such as debugging and measurement tools, which help programmers write efficient programs. In both of these areas functional systems are lacking.

To address the complete lack of measurement tools for functional languages, a profiling technique was designed which can accurately measure the number of calls to a function , the time spent in a function, and the amount of heap space used by a function. From this design, a profiler was developed for higher-order, lazy, functional languages which allows the programmer to measure and verify the behaviour of a program. This profiling technique is designed primarily for application programmers rather than functional language implementors, and the results presented by the profiler directly reflect the lexical scope of the original program rather than some run-time representation.

Finally, there is a discussion of generally available techniques for parallelizing functional programs in order that they may execute on a parallel machine. The techniques which are easier for the parallel systems builder to implement are shown to be least suitable for large functional applications. Those techniques that best suit functional programmers are not yet generally available and usable.

Acknowledgments

As much of the early research work for this thesis was undertaken at Kingston Polytechnic, I would first like to thank Prof. George Rzevski, formerly Head of the Department of Information Systems at Kingston, who had faith in me and let me start a PhD on a part-time basis and then employed me as a Research Lecturer. Thanks are also due to Prof. Pat Pearce, who as a supervisor and a friend helped and advised me through the years, and with whom I had many interesting discussions and exchange of views. Further thanks go to Ed Whitehead for his assistance in documentation, and to all the members of the Department for making it an enjoyable place to work.

This thesis work was completed at the Department of Computer Science of University College London with help from SERC in the form of a short grant. Thanks go to Chris Clack, my supervisor at UCL, and to all the members of the functional programming group. Special thanks go to Dave Parrott for his persistent efforts in building a distributed graph reduction system and adding a profiling mechanism, and for his patience in his discussions with me about the finer points of reduction systems, and for our chats in general. Also, Simon Courtenage read drafts of my thesis in detail, and both Derek Long and Russel Winder gave support and advice when times got difficult. I would like to thank all the members of the department who made my stay at UCL pleasant.

Finally, I would like to thank my wife Lizzy for her continual support, encouragement, and the time she spent reading drafts, and to my son Sam whose imminent arrival acted as a catalyst to finish this research work.

Introduction

Current research indicates that there is a need for parallelism in rule-based systems in order to increase their speed [Gupta86], [Hillyer86], and [Miranker87]. Functional programming is considered a technique well-suited for harnessing parallelism because functional programs decompose into independent tasks each of which can be evaluated concurrently [Hudak85], [Cripps87], and [Watson88]. Given that there is a need for parallelism and there is a tool that is well-suited for harnessing parallelism, it seems pertinent to ask the question:

> *Can functional programming be used for harnessing parallelism in rule-based systems?*

The need for extra resources in computer systems is being hampered by conventional software and hardware techniques [Turner80]. The software limitations are known as the *software crisis*, where the size and complexity of software systems is becoming unmanageable. This is combined with a proportional increase in both the number of bugs, and the cost of development and maintenance. One solution to the software crisis is the use of functional programming techniques which provide benefits through good design, powerful abstraction mechanisms, the lack of side-effects, and a strong mathematical basis [Turner84].

On a par with the software crisis is the *hardware crisis* in which the limits imposed by both the speed and size of hardware have begun to force designers into new areas. To overcome the hardware deficiencies, the use of parallelism is generally advocated. Large scale parallelism can be derived from machines with hundreds or thousands of

processors all executing programs at the same time [Hillis85]. Each processor does a small amount of the work, but the whole homogeneous machine does enormous amounts. The architecture of these parallel machines is a deviation from conventional machines, and the harnessing of the parallelism to the fullest capacity calls for novel techniques in software. Functional programming provides a method for approaching software design in a novel way [Hughes89] and, as functional languages are independent of any machine architecture [Henderson80] [Glaser84], they are amenable to execution on a wide range of machines.

A particular class of applications which imposes a heavy load on conventional architectures and would benefit from parallelism are rule-based systems [Stefik81]. Rule-based systems [Hayes-Roth85] are the use of artificial intelligence techniques applied to human understanding and reasoning [Winston81], [Rich83], [Charniak85]. They are particularly appropriate for many tasks, including requirements analysis, expert systems for analysis and synthesis, and for complex problems where the flow of control is unknown or the definition of the model is incomplete.

In the past, rule-based systems, which provide a powerful paradigm for problem solving, have been limited by their run-time performance. In an attempt to overcome this, several parties have written parallel versions of rule-based systems [Gupta84], [Hillyer86], [Gupta86], and [Oflazer87]. They all use specialized hardware for their implementations and their work provides comprehensive data concerning these specialized machines. Yet, although the behaviour of their algorithms are well understood, little work in this area has been done for general purpose hardware.

There is a need to build a parallel implementation of a rule-based system that is portable, flexible, and does not require specialized hardware. As a functional programming environment provides a mechanism which enables programs to be independent of any machine architecture, there is no need for the programmer to be concerned with the partitioning, scheduling, and synchronizing of parallel tasks as this can be done automatically by the compiler and the run-time system [Clack85], [Clack86] (Although some researchers advocate the use of annotations or skeletons to indicate parallelism and placement in addition to the automatic analysis provided by the

compiler [Hudak85], [Kelly87]). In parallel functional programming environments, the dynamic mapping of tasks onto machines occurs at run-time in contrast to some specialized environments, in which a static mapping of tasks onto machines is done in advance [May84]. This feature enables functional environments to have dynamic load balancing, which distributes work more evenly [Hudak84]; in other words, no machine need be idle if there is work to be done [Eager86]. The most important aspect from a programming viewpoint is that parallelism is implicit and no programmer intervention is needed to run the rule-based system on a selection of different parallel machines.

Goals of the research

The need for parallelism in rule-based systems has been ascertained. In [Stolfo86] and [Rosenthal85], both conclude that implicit parallelism, which is where the system finds the parallelism rather than the programmer stating where it is, is a promising area to investigate in order to obtain more parallelism in a rule-based system. This is because programmer specification of parallelism has reached its limits due to the complexity of the task. As a consequence of the findings of Stolfo and Rosenthal and because one of the many proposed benefits of functional languages is that parallelism is implicit, functional programming techniques seem well-suited for obtaining parallelism in a rule-based system. Therefore, functional programming was chosen as the vehicle for the implementation of a parallel rule-based system in this research.

The original goals of the research were:

i) to use functional programming techniques to implement a rule-based system.

ii) to analyse the functional rule-based system for inefficiencies and then to implement efficient new algorithms or to transform old algorithms into more efficient ones.

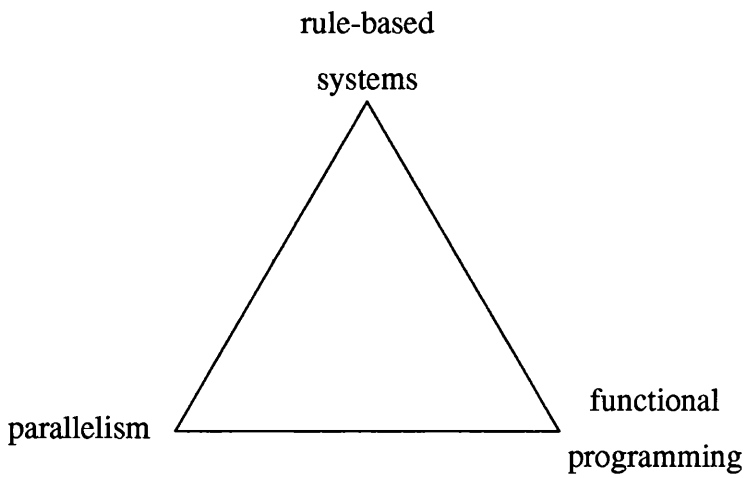
iii) to create a version of the functional rule-based system that is amenable to execution on a parallel machine.

iv) to analyse the functional parallel environment and gather data on the performance of the parallel functional rule-based system in order to remove any inefficiencies.

v) to compare the performance of the parallel functional rule-based system with an existing parallel rule-based system.

Only when these 5 aims have been addressed will it be possible to determine if functional programming techniques are suitable for harnessing parallelism in rule-based systems.

There are three main research areas in this thesis, namely: functional programming, rule-based systems, and parallelism. There has been previous research work in combinations of two of the three areas, but this thesis is new in combining all three. These main research areas are inter-related such that their combination can be viewed as a three way relationship:



There has been little work on large parallel functional applications as much of the work in the functional programming arena has been either theoretical or focused on implementing abstract machines and compilers. Although the many proposed benefits of functional programming appear to render it a well-suited method to use for both parallelism and rule-based systems, there is as yet no definitive answer indicating how useful functional programming techniques are for harnessing parallelism in general and

rule-based systems in particular.

The functional programming environments available are not as mature as imperative programming environments because practical functional programming environments are relatively new. There are few sources of functional interpreters and compilers, there are no known full development environments for functional languages, and there are no books on the design and development of large, functional applications. Furthermore, there are no design methodologies in general use for developing functional programs as there are for imperative programs. The lack of development environments and written material could limit the development of a functional application; this thesis will investigate if this is the case.

Furthermore, the formalisms which constitute the basis of functional languages are considered to be an advantage for functional programmers. These formalisms provide a rigid framework within which programs are built. However, this *advantage* could also be a *disadvantage* because operations that are simple to do in imperative languages could be difficult in a functional language due to this rigid framework. (For example, it is impossible to add a line of code to print the value of an object. In order to get this value, the code must be explicitly designed).

Contributions

The contributions of this thesis are:

- a critical assessment of the suitability of functional programming techniques for implementing large applications and rule-based systems in particular.
- a critical assessment of practical state manipulation techniques in functional programming.
- a large, working, application written in a lazy, higher-order functional programming language which does large amounts of state manipulation
- a critical assessment of the functional programming environment, with suggestions for how the environment can improve.

- the design, implementation and analysis of a tool for profiling lazy, higher-order functional programs. The tool measures function call count, time spent in a function, and the heap space used by a function.
- a critical assessment of techniques for parallelizing large functional programs.

Overview of the thesis

It is the aim of this research to investigate if functional programming techniques can be used to develop and build rule-based systems that are of an acceptable quality, if they are indeed beneficial for tasks that require parallelism, and if they can be used to harness parallelism in a rule-based system such that the resulting rule-based system executes at an acceptable speed. In addition, the available functional programming development environments will be considered in relation to these aims, and in particular to determine their suitability for writing a large application.

Chapter 1 provides a general background to the three main research areas, and the advantages of functional programming are discussed in more detail.

Chapter 2 provides a more detailed discussion of rule-based systems, why certain pattern matchers are more efficient than others, and discusses previous work in parallel rule-based systems.

Chapter 3 considers the design and implementation of a rule-based system written in a higher-order, lazy functional language, and discusses how different aspects of functional programming affected the design and the implementation of separate components of the rule-based system.

Chapter 4 discusses the issues arising from the implementation in chapter 3. Particular attention is paid to programming aspects, namely algorithms and data structures, and to the efficiency of programs. This chapter considers the functional programming environment in more general terms than chapter 3.

Chapter 5 addresses one of the issues arising in chapter 4 — the lack of measurement tools. The design and implementation of a profiler for higher-order, lazy

functional programs is described. This profiler measures the number of calls to a function, the amount of time spent in a function, and the amount of heap space used by a function.

Chapter 6 considers how parallelism can be harnessed in a functional program and shows the results of using a real parallel machine. It can be seen that the techniques advocated are not ready to be used for large functional programs.

In the final chapter the work is reviewed and conclusions drawn. Pointers to where further work needs to be done in order to develop functional programming into a more useful tool for harnessing parallelism in rule-based systems are discussed.

Chapter 1

1. General Background

This chapter presents a general background to the three main research areas in this thesis, namely functional programming, parallelism, and rule-based systems.

1.1. Functional Programming

A functional program is a program that consists entirely of functions. A program has a main function, which calls other functions to do work for it, and they in turn call yet more functions. The main function collects input from the user and prints the result which is calculated by its body. Functional programs have a mathematical basis which enforces a rigorous approach to the design and implementation of the program.

Functional programming is being investigated by many researchers because of its theoretical basis, and because functional programs are amenable to automatic machine-based reasoning. The areas being investigated include automatic program transformation [Darlington80] [Darlington90], automatic program proving [Turner82], and formal semantics [Stoy80] [Schmidt86], while others are investigating the efficient implementation of functional programs on conventional architectures ([Turner79], [Fairburn87], [Peyton-Jones87],) [Peyton-Jones89] . The area this thesis investigates is the use of functional programming for large applications.

The benefits gained from writing an application in a functional language are:

- there are expressions only, no commands. Functional programs express what to do as opposed to conventional programs, which express how to do it. This prevents programmers from worrying about small details, such as

incrementing a control variable of a loop, and leaves the programmer free to solve larger problems.

- there is no assignment to variables, just definitions; thus there can be no side-effects and the ability to state formally what is happening in a program is maintained. Obscure behaviour from variables being unexpectedly updated is eliminated.
- there is no explicit flow of control or sequencing due to there being no variables to change in a loop statement and no concept of a program counter to state where the next instruction is. Therefore, there are no confusing goto's. The programmer does not have to define a total ordering on operations; flow of control and sequencing is through function application, recursion, and data dependencies.
- there is no explicit memory management. The memory or heap space is managed transparently, with heap space being allocated and deallocated on demand. This avoids the problems of programs failing because of illegal pointers.
- there is no connection between the source language and the underlying machine architecture. Therefore, the code for the application need never be changed when a different sequential or parallel machine is available.
- potential parallelism in the code can be found by special compiler techniques because there are no inter-procedural dependencies between functions. As the parallelism is implicit, the programmer is saved from stating where parallelism occurs.
- functions are first class items within a functional language and are as important as data. This results in the same treatment for functions as for numbers and lists, thus presenting a level of uniformity not seen in conventional languages.
- higher-order functions are permitted. This enables functions to be passed or returned to or from other functions arbitrarily, thus allowing a high

degree of expressiveness.

- lazy evaluation is available in some run-time systems which allows infinite data structures to be defined. This means general solutions to problems can be defined rather than having a solution for an arbitrary number as is often the case in imperative programs, whereby a programmer will chose to evaluate a large number of solutions. This results in greater modularity [Hughes89].
- there are very few syntactic rules, thus enabling programmers to concentrate on the problem at hand and not on the syntax. Conventional languages often over-burden the programmer with syntactic rules [May83].
- the notation used in functional languages is very close to that used in formal methods, hence any system designed using these methods can be implemented very rapidly. Functional languages are often considered as executable specification languages [Turner84].

These benefits allow the development of more expressive and modular programs which are closer to the conceptual abstraction of a model. This contrasts with the conventional approach which requires a sequence of commands to be specified to fit with the traditional von-Neumann model of computation [Backus78]. This is a major benefit for functional programming because no time needs to be spent changing the conceptual model into the von-Neumann model so that an algorithm can be expressed in a conventional programming language.

With all these benefits forwarded to the functional programmer, he is free to concentrate on problem solving rather than fiddling with minor details. The high-level specification of functional languages means that program proving techniques and automatic program transformation techniques can be used. This is a further benefit for functional programmers. This is not the case for conventional languages where these techniques are not available to programmers.

1.1.1. Program proving

To determine if a function behaves correctly it is desirable to prove its correctness rather than running numerous and contrived tests of the function which may not find failure cases. Functional programs are amenable to program proving, which is much the same concept as a mathematical proof. The approach used to prove functional programs is based on equations and the properties of equality. Most of the facts one may wish to prove about a program may be expressed as equations. For example we may need to prove that:

$$map\ (f \cdot g) = map\ f \cdot map\ g$$

or that:

$$reverse\ (reverse\ l) = l$$

In [Bird88] there is a detailed presentation of proofs of both of these equations.

The attraction of this approach is that functional programs already consist of equations, so that the nature of proving a program involves deriving a new set of equations which have the same properties as the given set of equations. Reasoning with equations is a well established mathematical activity and thus presents no new undefined problems.

1.1.2. Program transformation

Program transformation is a technique for mapping an expression from one form to another using techniques similar to algebraic manipulation. For example, $n(x+1)$ can be transformed into $nx+n$, and vice-versa. Different transformers take the expression and rewrite the expression such that it is semantically equivalent but structurally changed. Transformation can be used to improve efficiency in programs by manipulating the text of a program while maintaining correctness. The set of transformations developed are [Burstall77]:

Definition — introduces a new definition.

Instantiation — introduces a substitution instance of another equation.

Unfolding — replaces a call of a function by its body, substituting the formal parameters.

Folding — replaces the body of a function by a call to the function with the parameters

Abstraction — introduces sub-definitions.

Program transformation can be used to convert well designed code into a more efficient form for execution. Table 1.1 shows the attributes of the *before* and *after* code. The *before* code is the style written by the programmer and has all of the desirable properties of a program from a human perspective. The *after* code is the code actually executed on a machine and has the desirable property of executable code, namely efficiency. Thus, program transformation does all the hard work of optimization and allows the programmer to concentrate on the important issues of good quality design and structured programming.

Before	After
clean	obscure
modular	tangled
short	long
simple	complex
inefficient	efficient

Table 1.1: Transforming functional programs for efficient execution

Program transformation techniques can also be used to generalize regularly used expressions into new function definitions. The following is a step-by-step example of how transformation is of benefit to functional programmers.

An Example of Program Transformation

This example relies on some proofs that are not shown here but are taken to be true. As an example of program transformation, consider a function that takes two lists and appends every element of the first list onto every element of the second list. This is similar to the cross product function, which is traditionally defined as:

$$\{(x, y) \mid x \in X,\ y \in Y\}$$

This function will be called *cp* [1].

The *cp* function can be used to create the cross product of multiple lists. The following expression creates the cross product of 4 lists:

$$cp\ list1\ (cp\ list2\ (cp\ list3\ (cp\ list4\ [[]]))) \qquad (A)$$

where the resulting list will have elements of the same length as the number of lists passed to the calls of *cp*, in this case 4. By using program transformation, it is possible to convert multiple calls of *cp* into a function that will take any number of lists, and produce their cross product. Step 1 uses the proof:

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

such that equation A can be transformed into:

$$(cp\ list1 \cdot cp\ list2 \cdot cp\ list3 \cdot cp\ list4)\ [[]] \qquad (B)$$

Step 2 uses the proof:

$$(f . g)\ x = compose\ [f, g]\ x$$

such that equation B can be transformed into:

$$compose\ [cp\ list1\ ,\ cp\ list2\ ,\ cp\ list3\ ,\ cp\ list4]\ [[]] \qquad (C)$$

Step 2 uses the proof:

$$map\ f\ [x_1, x_2, \ldots] = [f\ x_1, f\ x_2, \ldots]$$

---

[1] A version of cross product which can be composed with other cross product functions can be defined in Haskell as:

```
cp :: [a]->[[a]]->[[a]]
cp xs ys = [ (x:y) | x <- xs , y <- ys]
```

such that equation C can be transformed into:

$$compose\ (map\ cp\ [list1\ ,\ list2\ ,\ list3\ ,\ list4])\ [[]] \qquad (D)$$

where brackets have been added around the *map* expression for grouping. The final step involves the introduction of a new definition:

$$multicp\ l = compose\ (map\ cp\ l)\ [[]] \qquad (E)$$

Using the new function *multicp*, equation A can be written as the expression:

$$multicp\ [list1\ ,\ list2\ ,\ list3\ ,\ list4]$$

However, *multicp* can be passed any number of lists to generate the lists' cross product. The importance of transformation in large functional programs is discussed in [Kelly87]. Kelly's PhD thesis has an extensive description of program transformations used for a graphics processing system which takes a naive implementation and produces a program which is more amenable to a distributed, parallel architecture. In the cross product example, it took 5 transformation steps to go from a specific instance of function calls to a general purpose function.

1.1.3. Functional Applications

There are few large functional applications, and the creation of one normally is of enough interest to generate some research papers. Some examples of large functional applications are shown in table 1.2. At the start of this research there was little reference material for the functional applications builder. The current situation is that many more have been written and reported in recent times, showing how functional

[2] A collection of applications is being made by Partain for his work on benchmarking Haskell implementations [Partain92]. This suite of functional programs is intended to be a representative workload for a Haskell compiler and run-time system. The suite will be used for finding good features of different Haskell compilers.

applications are now coming to the fore [2].

Application	Author	Location	Reference
YACC in SASL	S. Peyton-Jones	UCL	[Peyton-Jones85]
Lexical Analyser Generator.	R. Jones	UKC	[Jones86]
Spreadsheet	S. Wray	Cambridge	[Wray86]
SML in SML compiler	A. Appel	Princeton	[Appel87]
Database	P. Trinder	Glasgow	[Trinder89]
Process Animation	K. Arya	Oxford	[Arya89]
Lazy ML in Lazy ML compiler	L. Augustsson	Chalmers	[Augustsson89]
Solid Modelling	D. Sinclair	Glasgow	[Sinclair90]
A terminal emulator	C. Runciman	York	[Runciman91]
Text Compression	P. Sanders	BT Labs	[Sanders92]
Quasi Linear Hyperbolic Partial Differential Equations	J. Boyle	Argonne National Laboratory	[Boyle92]
Oil Reservoir Modelling	R. Page	Amoco	[3]

Table 1.2: Examples of functional applications

Much of the earlier implementation work for this thesis was done using the functional language Miranda† [Turner85]. It was chosen at the beginning of the

[3] Personal communication

† Miranda is a trademark of Research Software Ltd.

research because it was the most effective lazy, higher-order functional language available. It had the latest features, it was commercially supported, and it was widely used in the functional programming community. Since that time Miranda has been superseded in the functional programming community by Haskell, a public domain language for which there are now many sources of compilers and interpreters [Hudak88]. As Haskell is the more modern and generally used functional language, and because any Miranda functions can be easily converted to Haskell, all code examples will be in Haskell even though they were originally implemented in Miranda. A brief introduction to Haskell is given in appendix B in order to clarify the features used in this thesis.

1.2. Parallelism

The main aim of parallelism is to execute a program on more than one processor in order to speed-up the execution time of that program. This technique is achieved by splitting the program into separate tasks and evaluating the tasks concurrently, or by applying the same operation to many data items concurrently [Uhr87]. The former approach is known as process parallelism and the latter is known as data parallelism.

Process parallelism consists of a number of independent threads of control engaged in concurrent computation. Each task does a small amount of the whole computation. Data dependencies between the tasks cause task synchronization. Data parallelism consists of multiple data structures which are processed at the same time by one operation.

Attempts to design and write parallel languages resulted in parallel features being added to existing languages. The method for programming in these languages relies on the programmer knowing which parts of the program can be executed in parallel and how data in different parts of the program interacts with the other data. This process introduces another level of complexity in software creation. It is more difficult to write a parallel program than to write a sequential program due to the complexity of parallel algorithms, side-effects causing unexpected interactions, and the enormous amount of time spent on finding the parallelism. When a program is large and complex, the task of

explicitly stating where the parallelism is can be difficult.

An alternative method is to use functional programming, in which the parallelism is implicit and can be found by a clever compiler [Clack85]. There are no side-effects in functional programming, so there are no obscure interactions and there is no global data store and, thus, no need to synchronize on global data. In addition, the semantics of the language are well defined and do not change when a parallel evaluation mechanism is used [Peyton-Jones89a]. Furthermore, there is no burden for humans in learning parallel features; they can involve themselves with expressing algorithms only.

Although parallelism is a way to improve the performance of complex applications, the parallelism harnessed has to be effective, i.e. a parallel version of a program must be more efficient than the best sequential version. Furthermore, some algorithms need to be rewritten and / or redesigned in order to work in parallel. (Experiments with old Fortran programs have demonstrated this [4]). Effective parallelism is not about keeping processors busy but about speed-up relative to the speed of the fastest sequential version. Schultz warns [Schultz88]:

i) a parallel algorithm can be made to achieve optimal cpu usage by increasing the complexity — that is, just because a parallel algorithm is keeping many cpu's busy does not mean that the algorithm is effective.

ii) a parallel algorithm can be made cpu bound either by making its complexity sufficiently bad or by using slower cpu's.

iii) a poor algorithm doing operations at a high rate does not necessarily finish before a good algorithm doing operations at a slow rate.

The important factor is speed-up over the best sequential version of a program. One can define speed-up to be:

$$speed-up = \frac{time\ of\ the\ best\ sequential\ algorithm}{time\ of\ the\ best\ parallel\ algorithm}$$

[4] The programs are known as *dusty deck* programs because they are so old they were originally entered into a computer via a *deck* of punched cards. The cards have been stored for so long that they have become *dusty*.

In [Padua87], which is predominantly about parallelism and Fortran, Padua discusses how parallelism is harnessed in imperative languages such as Fortran and how this may differ for functional languages. He observes that explicit parallelism, which forces the programmer to use parallel language constructs in order to harness parallelism, must be used. The constructs may be one of:

- fork/join
- microtasks
- parallel loops

and are needed due to the features of imperative languages, such as global store and side-effects.

Conversely, implicit parallelism occurs when a compiler or interpreter automatically extracts the parallelism. Due to the absence of side-effects in functional languages, Padua observes that there is no need for compile-time:

- inter-procedural analysis to compute dependencies
- array expansion
- variable renaming

which are all required for parallel versions of Fortran and other imperative languages.

Padua concludes that, although there are differences in parallel languages at present, future parallel systems will comprise program manipulation components, meta languages, and specification languages. The functional programming world is able to address of all these now.

1.2.1. Parallel hardware

Designers of parallel hardware can make choices regarding the style of the machine they build. The main issues affecting their decisions are:

- general versus fixed communication
- fine versus coarse granularity
- multiple versus single instruction streams
- shared versus distributed memory

Although each issue can be characterized by extreme schools of thought, each offers a spectrum of choices rather than a yes/no decision. Each choice is independent of the other, thus allowing for many styles of architecture [Hillis85].

General Versus Fixed Communication

Some portion of the computation in all parallel machines involves communication among the individual processors. General communication permits any processor to talk to any other, whereas fixed communication allows only a few specific patterns of communication which are defined by the hardware.

The main advantage of fixed communication is simplicity, and for certain applications this mechanism can be much faster. The general communications machines have the potential of being easier to program for a wider range of tasks, and the connection pattern can change dynamically for particular data. However, depending on how a general communications network is implemented, some pairs of processors may be able to communicate more quickly than others due to attributes of the underlying real architecture.

Fine Versus Coarse Granularity

In any parallel computer with multiple processors, there is a trade-off between the number of processors and the size of each processor. We can characterize machines with a handful of processors as being coarse grained and machines with thousands to millions of processors as fine grained. The conventional, single processor machine is an extreme case of a coarse grained machine.

The fine grained processors have the potential to be faster because of the larger degree of parallelism, but the potential speed-up may not always happen due to factors

such as a large communication overhead. The processors in a fine grained system are generally less powerful, so many small processors may be slower than one fast, large processor.

Multiple Versus Single Instruction Stream

A multiple instruction stream machine is a collection of autonomous computers, each capable of executing different code. A single instruction stream machine is a collection of identical computers, each executing the same code. As both types of machine operate on different data, this leads to the commonly used synonyms for parallel machines – MIMD (multiple instruction multiple data) and SIMD (single instruction multiple data)

The most common type of SIMD machines are vector or array processors. These fall into two categories, either general purpose machines such as the Cray supercomputer, or special purpose machines such as CLIP [Duff83], which is used specifically for image processing.

MIMD machines come in many different forms due to the different methods of design and construction used by the different research groups. Some of the better known forms include dataflow machines [Watson79], Transputers [Inmos85], hypercubes [Intel85], the Connection machine [Hillis85], and the graph reduction machine [Cripps87] [Clack86]. Graph reduction machines are commonly used for executing parallel functional programs.

The choice as to whether SIMD or MIMD is better is difficult to make as the SIMD machine can simulate the MIMD machine and vice-versa. For well structured problems with regular patterns of control, the SIMD machines have the edge. In applications in which the control flow required of each processor is complex and data dependent, the MIMD architecture has the advantage. There are many arguments to consider when choosing an architecture for a real application [Fox89].

Shared Versus Distributed Memory

When processors have to access memory there are generally two configurations for this memory, (a) shared memory, where there is one memory and every processor has access to that memory, and (b) distributed memory, where there are many memories. In the distributed memory case, either the memories are independent units whereby any processor can access the memory, or the memories are associated with one processor and only that processor can access the memory. Each layout has advantages and disadvantages for different applications and, again, there are many arguments to consider when choosing an architecture.

1.3. The Rule-Based System Approach

Rule-based techniques are appropriate for many tasks, including requirements analysis, expert systems for analysis and synthesis, and complex problems where either the flow of control is unknown or where there is an incomplete definition of the model [Hayes-Roth85] [Waterman86]. Because of their modularity, rules appear to be the most natural representation for systems that are in constant flux [Hayes-Roth83].

One of the major reasons for choosing rule-based systems is that humans usually find it intuitively appealing to express their knowledge in terms of condition / action pairs (i.e. if *condition* then *action*). Also, because rule-based systems tend to be built incrementally due to knowledge becoming available in a piecemeal fashion, it is not necessary to know the entire model in advance, but rather to gradually build towards it [Waterman86] [5]. The power of rule-based systems is most evident when they are applied to large ill structured problems for which it is difficult to provide a detailed specification, such as analysing complex laws and statutes.

---

[5] This process of acquiring knowledge in a piecemeal fashion is similar to the way a baby learns. It learns a few rules and has a few facts but it is still able to exhibit intelligent behaviour. As it learns more rules and facts the baby is capable of doing more. Babies are not born with a head full of rules and facts. They gradually acquire these, and there is no pre-determined path as to how the baby's life will develop – it develops and is shaped as needs arise. The same is true for rule-based systems.

A rule-based system is a tool which enables the builders of artificial intelligence applications to represent their knowledge of a domain through rules (see [McDermott78], [Hayes-Roth85], and [Waterman86]). Consider an example rule from a computer hardware configuration program given in [McDermott82]. This rule helps to assign power supplies to a bus of the computer:

IF the most current active context is assigning a power supply
and a bus module of any type has been put into the cabinet
and the position it occupies in the cabinet is already known
and space is available in the cabinet for a power supply at that position
and there is an available power supply
THEN put the power supply in the cabinet in the available space

This rule is part of a system that started with 300 rules, and grew over a period of 6 years to have approximately 3500 rules. As the rules were added, the program could configure new computers as they were manufactured and could perform many new tasks. It is this kind of evolutionary growth to which rule-based systems are most suited. The rules are specified in English by the rule-based system designer to be expressive. They are then encoded by the rule-based system designer into a particular rule-based system language when enough rules have been acquired to process the facts of the domain. In an implementation of a rule-based system, rules are encoded in the form of productions [6].

1.3.1. How A Rule-Based System Works

A rule-based system has three main components:

- production memory, which contains productions each in the form of condition / action

---

[6] Because rules are encoded in the form of productions, *rule-based systems* are also called *production systems*.

- working memory, which contains working memory elements, each one being a fact about the domain
- an inference engine, whose task is to initiate the recognize-act cycle

In a rule-based system, production memory and working memory are independent of one another. Both production memory and working memory are unstructured and elements within each are independent of the other elements. Only the recognize-act cycle can combine the contents of production memory and working memory, and on each iteration of the cycle may update working memory. This process is shown in figure 1.1.

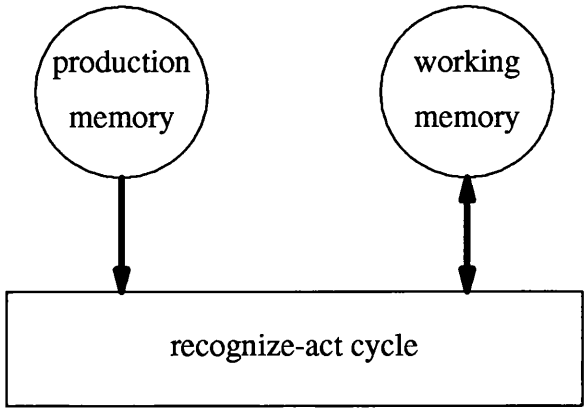


Figure 1.1: The components of a rule-based system

Production memory contains productions which are similar to a single conditional (`if-then`) statement in a conventional programming language. All productions are independent of one another, and there is no predefined order of production execution. A production contains n conditions C_1 to C_n and m actions A_1 to A_m. A production may be executed when working memory is in a state such that all conditions C_1 to C_n are simultaneously true. When the production is executed, then all actions A_1 to A_m are evaluated in the order in which they are written [7]. An action may add or delete an

[7] Some rule-based systems, such as SOAR [Rosenbloom85] have been modified to allow actions to occur in parallel.

object from the contents of working memory or do some input or output. Usually the condition parts of a production are called the left hand side (LHS) and the action parts are called the right hand side (RHS). This is because productions take the written form:

$$C_1\, C_2 \cdots C_n \rightarrow A_1\, A_2 \cdots A_m$$

where the conditions are to the left of the arrow and the actions are to the right.

Working memory contains objects called working memory elements. These objects represent either physical objects, relationships between objects, or statements about a particular domain. Working memory contains the "state of the world" for each rule-based system application, and its contents change continuously as the rule-based system executes productions. Production memory, by contrast, is stable [8]. Working memory and production memory are independent, and both have to be initialized at the beginning of exection for an application to work. Working memory is initialized with facts about the domain, that is, it contains the current "state of the world", and production memory is initialized with the rules of the domain.

The inference engine is the executor in a rule-based system. It determines which productions are appropriate to select by matching each production against contents of working memory. It then chooses one production to execute through conflict resolution. The execution of the production causes the actions to be evaluated, which then causes working memory to be updated, and hence the "state of the world" changes. This process of selection and execution is called the *recognize-act* cycle. Because of the continuous operation of the *recognize-act* cycle and because of changes in the "state of the world", new productions are selected on each iteration of this cycle. If it becomes impossible to select a production for execution, then the inference engine stops.

The *recognize-act* cycle takes the form:

1. *match* — evaluate the LHS's of the all the productions in production memory to determine which productions are satisfied given the current

[8] Some implementations of rule-based systems allow new productions to be built at run-time. This allows the rule-based application to display a learning behaviour. However, the most common implementation is for production memory to be static.

contents of working memory. The match process compares each condition of a production with every element of working memory.

2. *conflict resolution* — choose one production with a satisfied LHS. Often, more than one production is satisfied in the match phase; this is called a conflict. The conflict is resolved by selecting the best single production. If there are no satisfied productions, then the inference engine halts.

3. *act* — perform the actions specified in the RHS of the selected production. The actions may update working memory or do input or output.

The cycle iterates again by going back to matching, i.e. step 1.

The control flow and data flow of a rule-based system are presented in figure 1.2. Control flows from the matcher, to conflict resolution, to act, and back to the matcher again – this is the recognize-act cycle. Data flows from production memory and working memory into the matcher for matching, and into working memory when a production is acted upon.

Rule-based systems differ from conventional programs in two major respects. The first is that rule-based systems use a different method of encoding the state of a computation than conventional methods. A conventional program encodes state by updating values in variables. A rule-based system encodes state by placing objects into the system's working memory. The second difference is the way the flow of control is managed. A conventional program uses ordered statements together with control constructs such as loops and conditional branching. A rule-based system uses left hand side satisfaction. That is, each production's left hand side is a description of the states in which the production is applicable, such that the production is satisfied when objects in working memory cause each condition on the left hand side of the production to be true. When the rule-based system performs a match it is in effect searching for the best production to process the data in working memory. Once a production is chosen, the actions on the right hand side cause working memory to be updated.

The rule-based system model allows the programmer to concentrate on the essential problem solving strategies of a domain expert rather than complex data structures or control strategies [Brownston85]. Because of the relatively independent nature of the

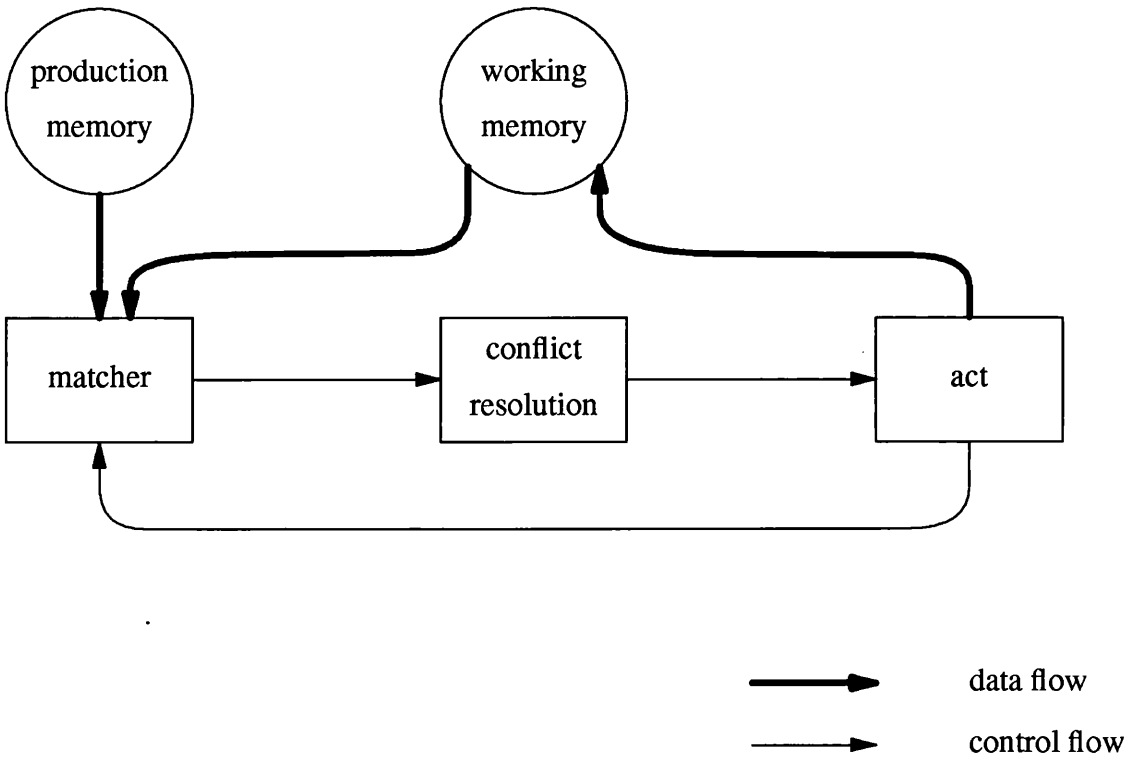


Figure 1.2: The control and data flow in a rule-based system

rules and the reduced amount of control information, a rule-based system specification does not prematurely determine the control strategy of the final solution. A rule-based system has neither a declarative model nor an imperative model. It requires an entirely different concept of program structure. The focus of attention using this technique is on non-formal solution strategies where knowledge elicitation is used to incrementally devise new rule sets to solve a small part of a problem. As more rule sets are created the system is able to perform more tasks. The interaction between the rules relies on the working memory elements that match the rules and the conflict resolution which choses a rule to execute. Different conflict resolution strategies allow identical rule sets to appear to behave differently. This often leads to unexpected behaviour as the flow of control may jump into an unexpected rule set, but it is this seemingly non-deterministic behaviour that makes rule-based systems appropriate for modeling intelligent behaviour where no known algorithms exist [McDermott78]. As Brownston observes,

this approach often leads to the discovery of algorithms and solutions to problems which may be missed when using conventional techniques.

Chapter 2

2. State-Saving in Rule-Based Systems

In this chapter there is a brief description of OPS5, which is a widely used rule-based system. There is a discussion and analysis of state-saving and non state-saving matching algorithms used in rule-based systems. This will show the benefit of saving state in a rule-based system matcher. Particular attention is paid to the Rete matching algorithm used in OPS5. This is an efficient algorithm for doing matching and is effective in sequential and parallel implementations. Then follows a discussion on research into parallelizing OPS5; this includes work done on the design of special hardware for executing rule-based systems in parallel, in particular OPS5, and on why the Rete matching algorithm is amenable to implementation on a parallel system. Finally, the issues arising from this research which lead onto considering why functional programming could be suitable for harnessing parallelism in OPS5 are reviewed.

2.1. A Language for Rule-Based Systems

The rule-based system chosen for further investigation in this thesis is OPS5 [Forgy81]. OPS5 is a system which allows the encoding of rules as a set of independent productions. Moreover, it is widely used and is the basis for some of the largest and best known expert systems (for example, a computer backplane configuration system [McDermott82] and an expert mainframe operator [Griesner84]). Due to its wide use, OPS5 is sometimes called the FORTRAN of artificial intelligence languages [Stolfo86]. Because of its wide use, its simplicity, the availability of a working rule-based system environment, and the availability of a formal grammar [Forgy81], OPS5 was

considered the best system to analyse and to make comparisons with the rule-based system devised in this thesis [9].

The representation of knowledge in OPS5 is contained in working memory. The representation is oriented towards objects and relations between objects. Each object and its attributes are represented through the use of working memory elements. For example, a working memory element may represent a block, which is named *block1* , is red, weighs 500 grammes, and measures 100 mm on each side. This block object can be represented in OPS5 as:

```
(block
        ^name    block1
        ^colour  red
        ^mass    500
        ^length  100
        ^height  100
        ^width   100)
```

In this example, the name `block` is the object class and is followed by a set of *attribute pairs* . The name of the attribute is preceded with a caret ^ and followed by the attribute value.

The specification of rules in OPS5 is simple yet sophisticated, allowing relatively easy encoding of knowledge into rules. The left hand side (LHS) of a production consists of one or more conditions. Each condition is a pattern that describes a working memory element. During the match phase of the recognize-act cycle, each condition of a production is compared with elements in working memory in order to determine if the condition matches any working memory elements. The condition is considered satisfied if it matches at least one working memory element, and the whole production is satisfied if every condition is satisfied.

---

[9] Full details of the syntax of OPS5 and how to program a rule-based system application can be found in [Brownston85], and reasons for choosing OPS5 as a language to build expert systems can be found in [Clayman87].

The patterns of each condition are abstract representations of working memory elements. These patterns may fully match every attribute pair of a working memory element, or may partially match a working memory element by matching a few attribute pairs. A pattern will match any working memory elements that contain the information in the pattern. For example, the condition pattern:

```
(block ^colour red)
```

would match any working memory element that described a red block, such as *block1* . However, the pattern

```
(block ^colour blue)
```

would not match *block1* because the colour attribute of *block1* is red. Patterns may contain variables which can match anything, but if the variable occurs again in the production, the value of the variable must be the same as before. In this way, OPS5 is able to the represent relationships between objects.

The right hand side (RHS) of a production consists of the actions. The actions can add, delete, or modify working memory elements and perform input or output. To create a working memory element, OPS5 defines the `make` action. This takes a description that looks like a pattern and creates a working memory element.

A production consists of a name, a set of conditions, and a set of actions. The `p` symbol is used to denote a production and the `-->` symbol is used to separate the LHS and RHS. The following example production prints a message if it finds a coloured block:

```
(p find-coloured-block
   (goal ^status active ^type find ^object block ^colour <c>)
   (block ^colour <c> ^name <n>)
 -->
   (write stdout Found a <c> block called <n>))
```

In this rule, if the first condition matches a relevant working memory element and the second condition matches the working memory element for *block1* , then the message:

Found a red block called block1

would be produced.

2.2. Alternatives to OPS5

The OPS5 rule-based system is freely available software which is reliable. It works on various platforms, as the source code has been written in many dialects of LISP, and there is detailed documentation and descriptions of how the inner parts of OPS5 work. This allows a functional OPS5 to be written and compared with an existing version. Furthermore, parallel versions of OPS5 have been built and documented. As OPS5 is used widely for research into rule-based systems, it was chosen for this research rather than any of the other options. In this section there is a brief overview of the alternative rule-based systems to OPS5 which were considered at the beginning of this research. The tools considered were large hybrid tools and small PC-based tools.

Large Hybrid Tools

The large hybrid tools that were considered were all commercial products; ART sold by Inference Corporation, KEE sold by Intellicorp, and Knowledge Craft sold by Carnegie Group. They are knowledge engineering environments rather than merely rule-based system shells. This is due to the fact that they each offer a variety of different ways to approach any given problem. They are complex systems with many options and considerable flexibility. The range of facilities these tools provide for knowledge-based system developers are:

- different methods of representing knowledge within each system
- inheritance of values by entities in the system
- alternative worlds or viewpoints, which allow hypothetical reasoning
- the support of truth maintenance mechanisms
- the selection of powerful inference and control mechanisms

The user interfaces of ART, KEE, and Knowledge Craft employ advanced man-machine interface techniques. All three tools provide natural language interface

mechanisms and explanation facilities, and allow full access to the underlying system and to other programming languages such as LISP or C. This enables developers to write critical code in a more efficient manner. These tools also allow access to commercial database systems for storing large amounts of data. With all these facilities available to the systems developer, any one of the three tools considered would be highly suitable to develop and implement a deliverable expert system. However, this is not the aim of this research.

The drawbacks of these development environments are the lack of a detailed description of their inner workings which is needed in order to make comparisons with the implementation in this thesis. There were no known parallel implementations of these tools and they were too big and complicated to emulate given the scope of the research. Furthermore, they consume enormous amounts of computing power, require machines with huge amounts of resources in order to execute, and need graphics hardware for their advanced user interfaces. In the light of these drawbacks, the large hybrid tools were not considered suitable for this research.

Small Tools

The small tools considered were taken from a collection of rule-based system tools which have proliferated recently on desktop PC's. They were considered because of the availability of the machinery for development and for end-users, and because the software was generally available. These were also commercial products but much smaller and cheaper than the hybrid systems. Those available were Expert Ease, Micro Expert, Micro Synics, and ES/P Advisor.

The power and flexibility of these tools is quite limited because they are specifically written for small machines. However, they were adequate for an initial investigation into rule-based systems. This investigation began by taking each tool individually and attempting to execute the demonstration programs. All four tools failed to execute for various reasons. Because of these execution failures and the lack of documentation, it was decided that the PC-based tools and the PC operating systems were either too unreliable or too unstable for this research.

2.3. Different matching algorithms

A matching algorithm in a rule-based system computes the state of the match between the whole of working memory and all the productions. Its task is to select the productions in which every condition of the production matches an element from working memory. From the selected productions, just one is chosen by conflict resolution for further execution. There are two main techniques for doing this matching; they are non state-saving and state-saving.

2.3.1. Non state-saving matching algorithms

The non state-saving algorithm is the simpler. Every condition of every rule is matched with every working memory element to generate the state of the match. Conflict resolution chooses the one production for execution, and the state of the match is then forgotten. However, every iteration of the recognize-act cycle recomputes the state of the match, but because very little changes on each iteration, it is nearly the *same* state that gets recomputed and forgotten. Due to the matching behaviour of this approach, the algorithm is sometimes called the *dumb matcher*. As this algorithm keeps recomputing the same state, it can clearly be improved.

2.3.2. State-saving

Matching algorithms for rule-based systems can save some of the match state on each iteration of the recognize-act cycle. This is because each match state is similar to previous match states. By saving some state, the cost of the match is reduced. There are different matching algorithms for OPS5 that store different amounts of state. They are:

i) the TREAT algorithm [Miranker87], developed for the DADO machine at Columbia University [Stolfo83] [Gupta84]. TREAT saves working memory elements that match each condition but does not save anything that matches combinations of conditions. The match for the combinations is recomputed on each iteration of the recognize-act cycle.

ii) the Rete algorithm [Forgy82], developed at Carnegie-Mellon University. Rete saves working memory elements that match each condition and also saves data for some fixed combinations of conditions. It stores data for the combination of successive conditions in a rule. It stores the state of the match for condition 1, and then the state of the match of condition 1 combined with condition 2, and then the state of the match for a combination of condition 1 and condition 2 and condition 3, until all the conditions have been matched.

iii) Oflazer's algorithm [Oflazer87]. This algorithm saves working memory elements that match each condition and it also saves the combinations of matches for all conditions. It stores the state of the match for condition 1 and condition 2, for condition 1 and condition 2 and condition 3, etc. But it also stores the state of the match for condition 1 and condition 3.

The amount of state saved is different in each of these three algorithms. TREAT is at the low end of the state-saving spectrum; however, it has to recompute some fixed combinations on each cycle which increases its execution time. Oflazer's algorithm, which is at the high end of the state-saving spectrum, spends a lot of time computing state which may never be used and also stores huge amounts of state. Its execution time and memory usage are higher than both TREAT and Rete. Rete is in the middle of the state-saving spectrum and is the algorithm used in the sequential version of OPS5.

2.4. Analysis of Matching Algorithms

This section provides an analysis of the cost of using either non state-saving or state-saving algorithms. Data collected by Gupta in [Gupta86] shows the typical values in a range of real rule-based system applications for the average size of working memory, the average number of productions, and the average number of conditions for all productions. The data was collected from four OPS5 applications and two SOAR applications [10] [Rosenbloom85], and is shown in table 2.1.

[10] SOAR is another rule-based system that is similar to OPS5.

Attribute	OPS5	SOAR	Average
Average size of working memory	528	371	476
Average number of productions	955	191	700
Average number of conditions	3.39	9.29	5.36

Table 2.1: Data from Gupta's PhD thesis

2.4.1. Cost Analysis of a Non State-Saving Matcher

The cost of using a non state-saving matcher for real systems can be evaluated by using the data collected by Gupta in a set of equations which identifies the cost of the non state-saving matcher.

Let:

w = average size of working memory

p = average number of productions

l = average number of conditions

The average cost of a match for one production during one iteration of the recognize-act cycle involves choosing all the combinations of the size of the production's left hand side from working memory and then matching them with the production. This equates to:

$$^{w}C_{l} = \frac{w!}{l!\,(w-l)!}$$

The average cost of matching during one recognize-act cycle is the cost of one production multiplied by the total number of productions:

$$p \times {}^{w}C_{l} \qquad \text{(A)}$$

When using a non state-saving matcher, the average number of matches per iteration of the recognize-act cycle can be evaluated by instantiating the values of w, l and, p in equation A. The values from table 2.1 for OPS5 systems are:

$w = 528$

$l = 3.39$ (rounded to 3)

$p = 955$

The average number of matches per recognize-act cycle equates to:

$$= 955 \times {}^{528}C_3$$

$$= 955 \times \frac{528!}{3!\ 525!}$$

$$= 955 \times \frac{528 \times 527 \times 526}{1 \times 2 \times 3}$$

$$= 2.33 \times 10^{10}$$

Therefore, when using a non state-saving matcher for a large OPS5 application, there are 2.33×10^{10} matches performed on *every* cycle.

To calculate the algorithmic complexity of the non state-saving matcher some approximations are made. One can use the approximation [11]:

$$\frac{528 \times 527 \times 526}{1 \times 2 \times 3} \approx 528^3$$

so that the average number of matches is approximately:

$$955 \times 528^3$$

Therefore, equation A can be approximated by:

$$p \times w^l$$

The complexity of the match is approximately polynomial on the size of working memory, and is cubic ($l = 3$) for an average OPS5 program.

---

[11] It can be observed that:

$${}^{w}C_l \rightarrow w^l \text{ as } w \rightarrow \infty, l \rightarrow 1$$

2.4.2. Cost Analysis of the Rete State-Saving Matcher

The Rete algorithm uses a clever compiler which converts the left hand side of a production into a graph representation of that production, called a Rete network. Nodes in the graph are used either to test attribute pairs of working memory elements or to save the state of previously computed matches. The test nodes match an individual attribute value or test that variables are bound correctly across combinations of working memory elements. The state-saving (or memory) nodes are used to save working memory elements that have successfully matched tests in the network. When all the conditions have been satisfied, the terminal node becomes active and the production is put into the conflict set. Consider two example rules, such as:

```
(p p1
   (C1 ^attr1 <x> ^attr2 12)
   (C2 ^attr1 15  ^attr2 <x>)
   (C3 ^attr1 <x>)
 -->
   (remove 3))
```

and

```
(p p2
   (C2 ^attr1 15 ^attr2 <y>)
   (C4 ^attr1 <y>)
 -->
   (modify 1 ^attr1 12))
```

These two productions have the Rete networks as presented in figure 2.1.

For extra efficiency, Rete is able to share partial networks between productions. This further enhances the speed of matching because it eliminates matches and reduces the number of nodes in the network compared with the non-sharing networks. Both productions `p1` and `p2` have a clause that starts:

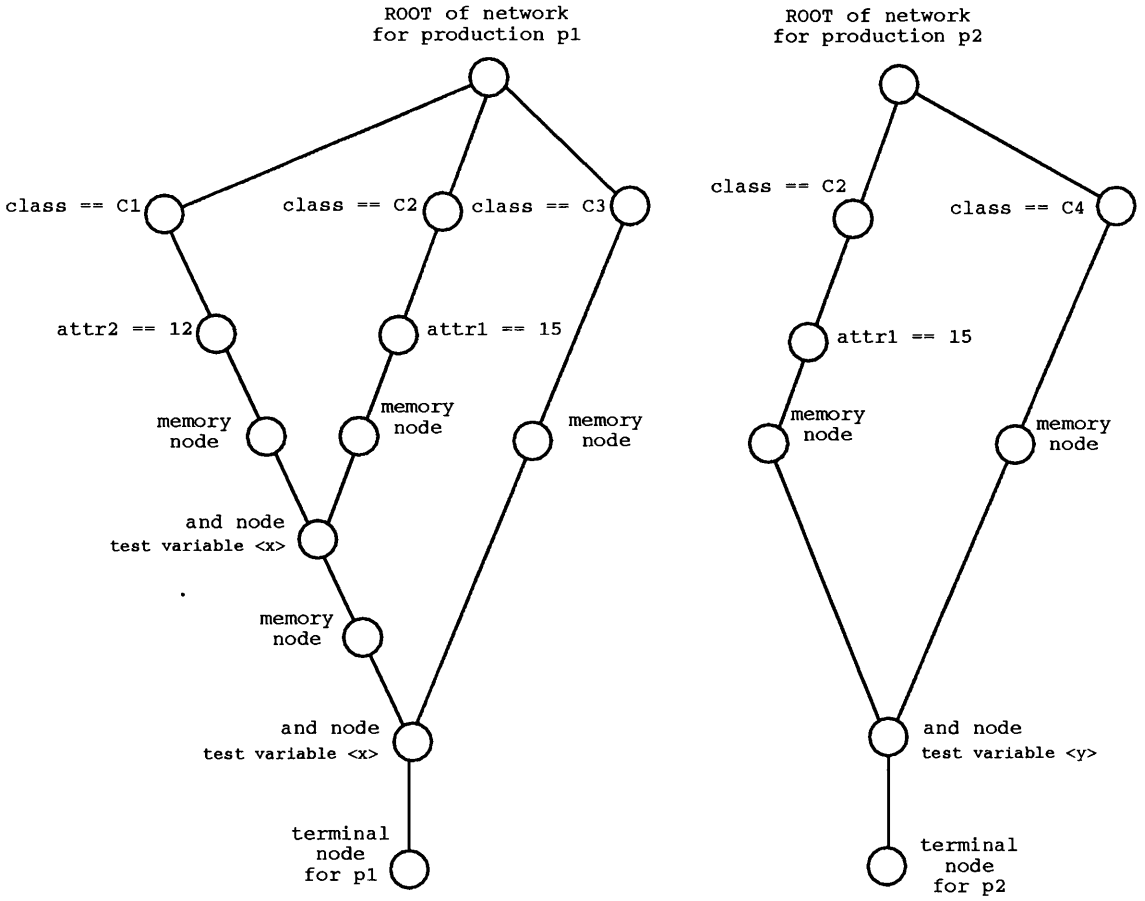


Figure 2.1: Rete networks for productions `p1` ***and*** `p2`

```
(C2 ^attr1 15 ...)
```

which can be shared. The network with sharing is shown in figure 2.2.

Gupta states that the behaviour of Rete is independent of both the number of productions in the rule-based system program and the size of working memory. He observes that the way production systems are currently written means that changes to working memory only affect a small fraction of productions. Gupta has calculated that each change to working memory will have an effect, on average, on 28 productions in the next iteration of the recognize-act cycle. This highlights how ineffective a non state-saving matching algorithm can be because if only 28 out of 955 productions are affected, then the non state-saving matcher does needless matching on 927 productions. That is, 97% of the matching work is unnecessary in a non state-saving matcher.

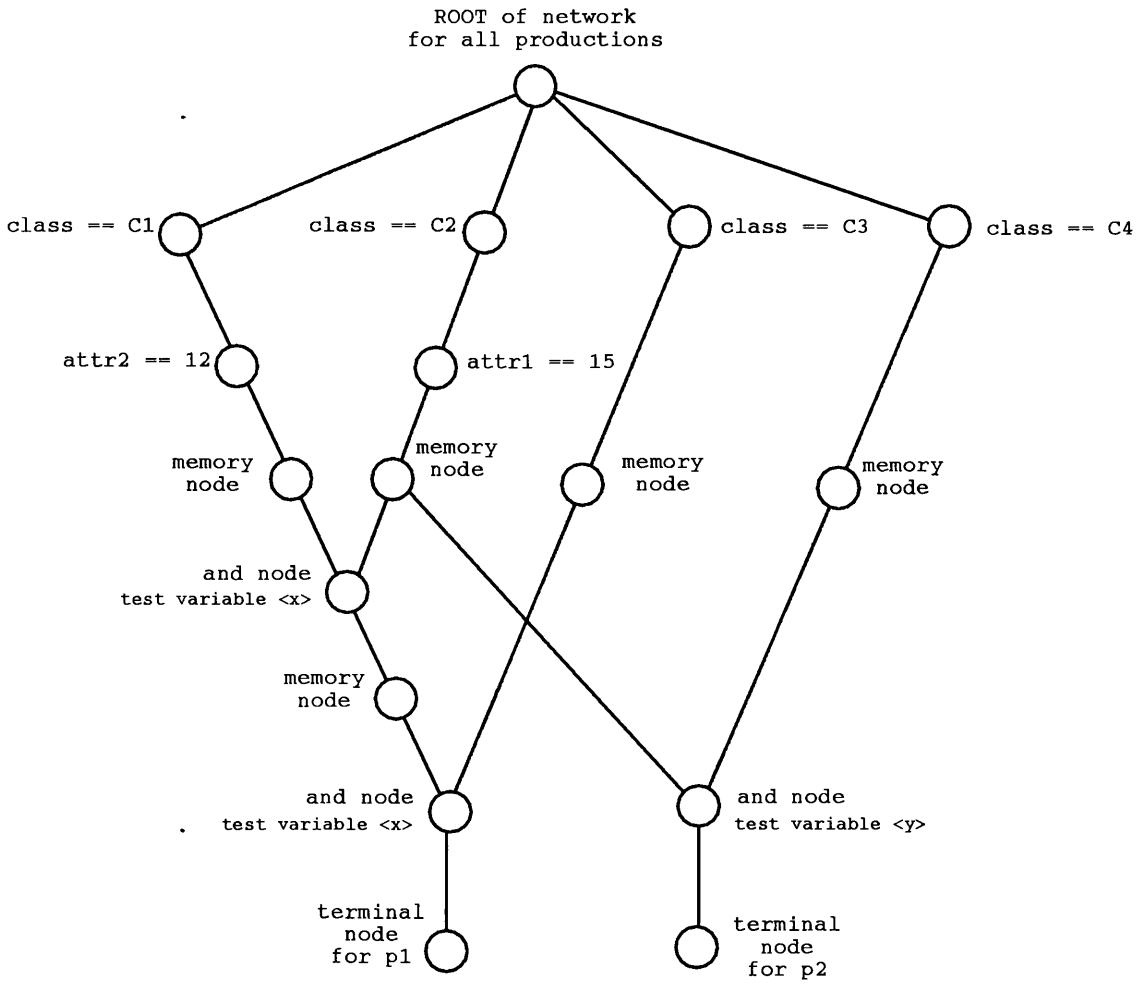


Figure 2.2: A Rete network with sharing

By saving state, it is possible for a matcher to discriminate between productions that need to be matched and those that do not. This means that the work done by a state-saving matcher is reduced to just 3% of the work done by a non state-saving matcher. However, Rete is better than this; it does not even match the 3% of affected productions but saves the state of the match from the previous recognize-act cycle. Data from Gupta indicates that there are, on average, 97 simple matches and 42 variable testing matches per change to working memory, and that there are 3.1 working memory changes per production firing. Therefore, there are:

$$3.1 \times (97 + 42) = 430$$

matches per cycle of the production system. In the non state-saving matcher there are 2.33×10^{10} matches per cycle, and with Rete there are 430. Therefore, the Rete state-saving matcher is algorithmically superior to a non state-saving matcher.

2.5. Parallel Rule-Based Systems

The desire for parallelism in rule-based systems is motivated by the observation that, although rule-based systems have been used extensively to build large expert systems, they are computationally expensive because of the matching required and, hence, run slowly. This slow execution time limits the use of rule-based systems to domains that are not time critical. For example, one study considered implementing an algorithm for real-time speech recognition using a rule-based system [Newell78]; it was found that present rule-based systems were between 5,000 and 20,000 times too slow for such a task. Rete was considered a suitable candidate for a parallel implementation of OPS5 because it is such a good algorithm for matching in sequential rule-based systems.

2.5.1. Parallelism and Rete

In his PhD thesis [Gupta86], Gupta states that the expected speed-up available from parallelism in Rete is between 100 and 1,000 times. However, the actual amount of speed-up is between 10 and 25 times. The main reasons for this are:

i) only a small number of productions are affected on each change to working memory (28 on average).

ii) a large variation exists in the processing required for each production. Furthermore, this variation can change on each cycle of the production system.

iii) the number of changes made to working memory per cycle is minimal (3.1 on average)

The consequences (of these reasons) are:

i) only a few processors would be busy even if there were a processor per production. This is due to the small number of productions that are affected on each recognize-act cycle.

ii) one is less certain of any speed-up due to parallelism because the stage after matching cannot begin until all productions have been matched. If the time to process one production is large, then the variation of processing time is large and the speed-up will be reduced. Figure 2.3 shows the time taken to process some productions in parallel. Matching all productions takes the same time as matching the most expensive production. Gupta states that it is desirable to eliminate the variation (for example, by using load balancing). The situation may change from cycle to cycle, but the important aspect is the time t_{max}, because this is the time taken for the whole match phase. The aim is to reduce t_{max} so that it is closer to t_{avg}. This situation is shown in Figure 2.4.

iii) the speed-up from processing multiple changes to working memory in parallel is minimal because only a small percentage of working memory is changed on each cycle and the amount of processing required to deal with these changes is also minimal.

The limited amount of speed-up available in OPS5 is mainly due to the way rule-based system programmers write their rules. To overcome this problem and to make effective use of a parallel machine, it is necessary to decrease the variation in the cost of processing each production. Gupta's method for harnessing parallelism involved designing a parallel version of Rete which exploits parallelism at a fine-grained level. The parallel Rete algorithm processes each node of the Rete network as a parallel task. However, Gupta observes that 75% - 95% of execution time is spent processing state-saving memory nodes and very little time is spent processing test nodes.

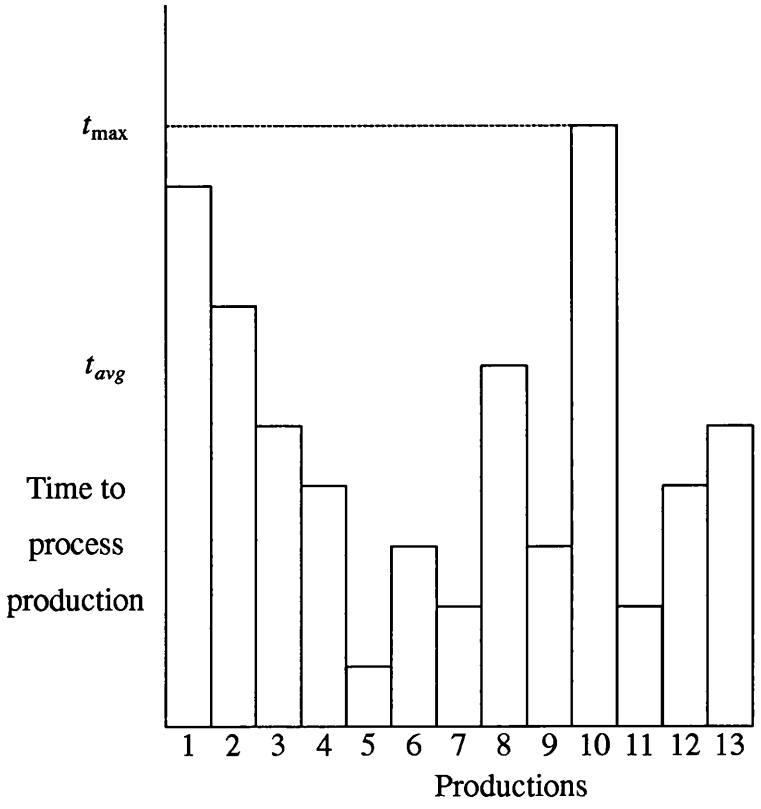


Figure 2.3 Actual situation — variation in processing time

2.5.2. Parallel Implementations of OPS5

Practical attempts at harnessing parallelism in the OPS5 rule-based system have all been successful to some degree. However, most can be characterized by three recurring features: the use of special hardware for the parallel machine, the use of different partitioning algorithms for each of the different architectures, and the static placement of tasks onto machines. For a detailed overview of much of this work see [Gupta86a] or [Gupta89]. As an example of the differences, consider the hardware chosen by these groups:

i) the Production System Machine project at Carnegie-Mellon [Gupta86] has 32-64 processors with shared-memory. Each processor has a hardware task scheduler.

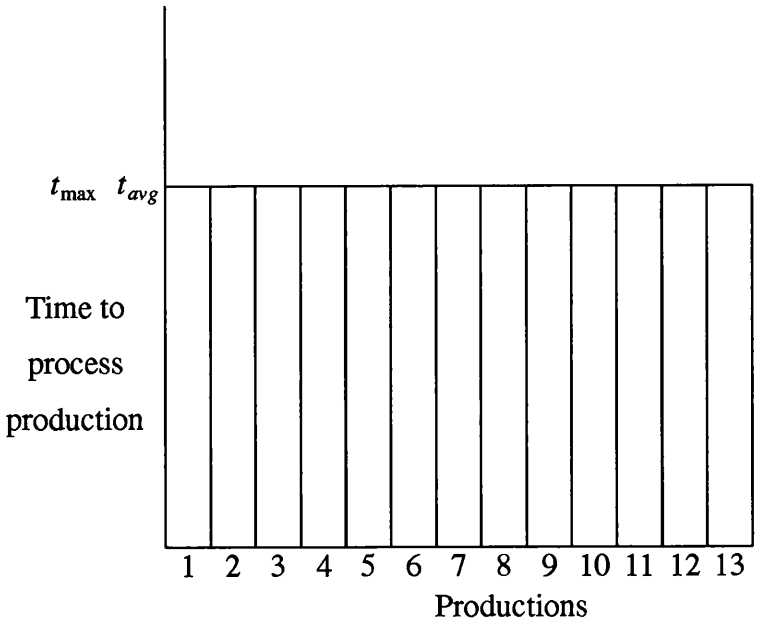


Figure 2.4 Ideal situation — no variation in processing time

ii) the DADO machine from Columbia University [Stolfo83] was a binary tree of 16,000 very small processors with distributed memory.

iii) the NON-VON machine, also from Columbia University, has between 16,000 and 1,000,000 very small processors connected to 32 larger processors [Hillyer86]. This is also connected as a binary tree.

iv) Oflazer's machine [Oflazer87] is a tree with 512 medium size processors at the leaves of the tree. These are combined with very simple processors at other nodes.

In order to drive these machines using conventional imperative techniques, significant parts of the application need to be rewritten in order to get the required parallelism. This thesis proposes that using functional programming techniques on a parallel machine will result in a portable implementation of a rule-based system.

2.6. Summary

A state-saving matching algorithm is far superior to the simple non state-saving matching algorithm as it needs to do only 3% of the work. The Rete state-saving matcher is very efficient, and for a typical OPS5 application with over 500 working memory elements and over 900 productions, it does 430 matches compared with a non-state-saving matcher which would do approximately 2.33×10^{10} matches.

Parallel versions of rule-based systems do not display as much speed-up as expected because of the way the programmers of rule-based systems write their rules. To overcome these limits, Gupta built a parallel version of Rete which processes each node of the Rete network as a separate task.

The challenge for functional programming is to emulate the efficiencies of Rete in both sequential and parallel environments. Gupta stated that 75% to 95% of the processing time is spent updating state-saving nodes. As functional languages have no concept of updatable store, recreating these efficiencies could be difficult. As there are no standard ways to manipulate state in a functional language, it is proposed that a prototype rule-based system be built using a non state-saving matcher in order to determine the effectiveness of manipulating state items such as production memory and working memory, before the manipulation of the extra state held by a state-saving algorithm is undertaken. Some researchers concluded that implicit parallelism techniques are the only way to improve the parallel performance of a rule-based system beyond that achieved by human intervention (see [Stolfo86], and [Rosenthal85]). By implementing a rule-based system in a functional language, one can expect to gain an efficient, automatically parallelized implementation.

Chapter 3

3. The Design and Implementation of a Functional Rule-Based System

This chapter describes the design and implementation of a version of functional OPS5 that has been created for this thesis. OPS5 is interesting from a functional programming viewpoint because it is an application that encompasses various computing disciplines. It has a compiler, a lexical analyser, and a pattern matcher and it requires a large amount of state which is accessed and regularly updated and does input and output from the environment. The literature has few reports of functional applications and the problems that arise, and as such this application highlights some of the issues that arise when building large applications.

This chapter contains a description of the design of each part of the functional rule-based system. Then the issue of state is discussed as this is a problem area in any functional application. This is followed by details of the actual implementation of the rule-based system with each component of the system considered separately. And, finally, an analysis of the working functional rule-based system which highlights both the problems and the solutions of using functional languages for large applications.

3.1. Design of a Functional Rule-Based System

One of the main investigations of this thesis was the analysis of the design and implementation of a large functional application. The process of building large applications in imperative languages is well known, but has its own problems. However, the process for functional languages is not well documented.

Early research indicated that there are problems which arise in functional programs that are not evident in imperative programs. These problems are the issues of (i) the manipulation of state and the related issue of store, and (ii) doing input and output. In imperative systems there are *variables* that hold values of state and which may be accessed or updated arbitrarily. Many imperative languages use lexical scoping to limit access to variables, but global variables are accessible everywhere. Each *variable* has a position in the computer's store and may be accessed and updated. A procedure in an imperative language may access and update the variables even if the variables have not been passed as an argument. Similarly, input and output in imperative languages can be done in arbitrary places. The input and output streams are part of a global environment that can be easily accessed without explicit mention of them if used in a function.

The functional rule-based system was designed with five main parts: there are three components that constitute the recognize-act cycle — the *matcher*, *conflict resolution*, and *act*; a *compiler*, which compiles the textual form into a form used by the matcher and the act process; and a *run-time system*, which provides the infra-structure to glue the previous four parts together.

Initially there seems to be a problem with retaining and updating state for both production memory and working memory. Functional languages do not provide updatable global variables, so how is it possible to implement a system which is inherently state-saving? The matcher needs access to both production and working memory, conflict resolution needs access to a selected subset of both production and working memory, and the act process needs to change the contents of working memory. An answer to this question will be seen in this chapter.

In the traditional imperative model, much of the global state is available in all parts of the system. In addition, any part of the state can be updated at any time, regardless of whether or not it is appropriate to that part of the system. This method of updating allows bugs to be easily introduced, although object-oriented techniques provide a discipline which reduces this problem [Stroustrup86]. Because functional systems do not have a global environment which can be accessed at any time, any items of state that are needed in a function have to be passed to it *explicitly*. By contrast, the imperative

system has *implicit* access to state.

In the functional implementation, the run-time system of the rule-based system passes state explicitly from one part of the system to another, removing the need for any global updatable state. Because no part of the system needs access to everything held in the state, the relevant items can be passed to any part of the system. For example, the match phase of the main cycle only needs access to the production memory and the working memory. No other items in the state are needed and no others are passed on.

Another aspect of passing explicit state in functional languages which is not seen in imperative languages is the need to *plumb* in the state. State has to be passed explicitly from function to function, just as water pipes are passed from room to room in a central heating system. Consider the example:

```
work :: (a->b) -> [a] -> [b]
work f l = [ f a | a <- l , test a ]
```

Suppose we wish to count the number of times `f` is applied to its argument. In an imperative language, it would be possible to add a line of code which updated the state of a global variable and the type of the function would not need to change. In a functional language this technique cannot be used. The state has to be made explicit, thereby changing the type of the function to:

```
type State = Int


workS :: (a->b) -> ([a], State) -> ([b], State)
workS f (l,s) = (list, s + sum statevals)
                where
                (list, statevals) = unzip [ (f a, 1) | a <- l , test a ]
```

This explicit change of the type and the extra code has to be done by design; it cannot be added as an afterthought. This is *plumbing*.

Figure 3.1 shows how the five main parts of the system fit together. The run-time system retains all the state and then passes the appropriate items to other parts of the system. The details of the items passed to each part are described in section 3.2, but

only some state items are needed in each part of the system. In figure 3.1, pm represents production memory and wm represents working memory.

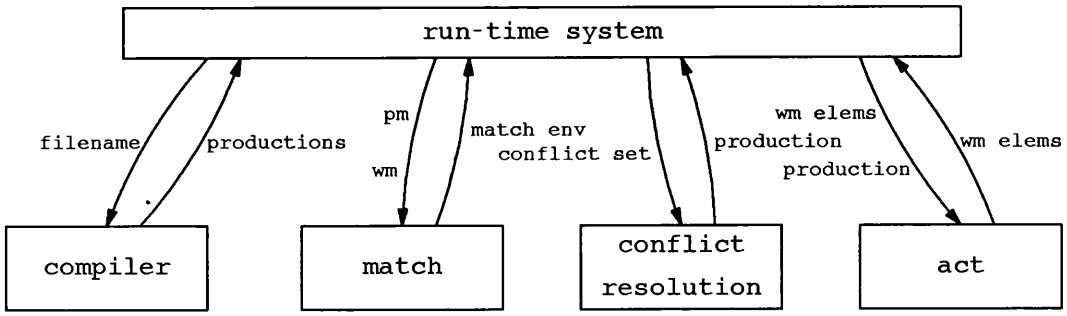


Figure 3.1: How the functional rule-based system fits together

The run-time system is the interface to the outside world, thus providing a mechanism for doing input and output. A large part of the design was a compiler that would recognize a language which specifies the rules for the rule-based system. Input to the compiler is in a textual form. Output from the compiler is in a form used by the match process, namely a list of productions which are saved in production memory. This requires interaction with the state-saving mechanism.

The match function takes the current working memory and current production memory and does an exhaustive match by matching every clause of every production against every working memory element. The result of this function is a conflict set, which is returned to the run-time system. The conflict set is passed through a conflict resolution function which selects one production to execute. The selected production together with the whole of working memory is passed to the act function, which executes the production and updates the working memory by either adding and deleting elements or doing input and output. The new working memory is passed back to the run-time system for the next iteration of the recognize-act cycle.

3.2. State Requirements of a Rule-Based System

The state required for a rule-based system is relatively large, comprising hundreds of productions and thousands of working memory elements. In the functional rule-based system there are twelve items of state to pass around, none of which can be avoided. The application runs from cycle to cycle, saving and updating different state items as it runs. The following sections will demonstrate how state handling need not be a problem, for the amount of plumbing which is required can be reduced and the access and update mechanisms can be streamlined, resulting in an elegant approach to state access and state update.

3.2.1. State Items in the Functional Rule-Based System

This section describes what items of state are saved in the implementation of the rule-based system. As previously stated, there are twelve items. These twelve items are briefly explained below:

Production memory — where all the productions are kept.

Working memory — a collection of independent data structures. This is the data that is matched with the productions.

Conflict set — the set of all matched productions which could possibly be acted on, together with their working memory instantiations.

Conflict resolution strategy — a function which takes the conflict set, and resolves down to the one production to be used on the next act. In OPS5 there is a choice of two conflict resolution functions. The choice is made at the start of execution of the rule-based system.

Conflict set history — a history of all productions and their working memory instantiations which have previously fired. This is kept because OPS5 disallows productions from firing twice with the same instantiation.

Current resolved production — the next production to fire, with full instantiation of working memory elements and bound variables [Forgy81].

Stream to file map — a map of stream name to file name. When OPS5 opens a file, a stream name is returned. From then, OPS5 can write to the stream. This is needed in order to output to the correct file.

Current firing cycle — a count of how many *recognize-act* cycles have occurred, in other words how many productions have been fired. Each cycle of the system increments this.

Current working memory timestamp — every working memory element has a unique timestamp which is used in various places. This timestamp is incremented every time an element is added to or deleted from working memory.

Debug output level — OPS5 does different amounts of debugging output depending on the value of the debug output level. 0 means none; 1 means indicate which productions are firing; 2 means indicate all of 1 and also which items are being added to or deleted from working memory.

System input — all input to the system is passed in the state. It is eaten by some actions.

System output — the output is incremented in many places, such as in actions and as part of debugging. Access to it is needed almost everywhere.

As previously stated, not all parts of the system need access to every item in the state. Because the run-time system is the infra-structure which holds the entire system together, it seemed better to have one big state and many access functions rather than having many small state structures which hold different parts of the state. Table 3.1 shows that different items in the state are used by many different parts of the system. Splitting a state structure into many small ones would introduce unwanted complexity. Table 3.1 also shows that different parts of the system interact through values in the state structure. Different sub-parts of the *recognize-act* cycle access or set the items within the state to be used for later processing.

Item in state	Set by	Accessed by
production memory	compiler	matcher compiler run-time system
working memory	run-time system act	matcher act run-time system
conflict set	matcher	conflict resolution run-time system
conflict set history	conflict resolution	conflict resolution
conflict resolution strategy	SET ONCE	conflict resolution
current resolved production	conflict resolution	act
stream to file map	act	act
current firing cycle	act	act
current working memory timestamp	act	act
current debug output level value	SET ONCE	compiler act
system input	SET ONCE	act
system output	ANYWHERE	run-time system

Table 3.1: State items in the functional rule-based system

3.3. Implementation of the Functional Rule-Based System

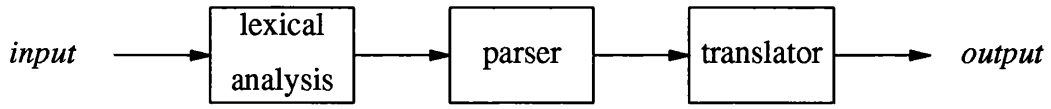
The implementation of the rule-based system is discussed, with each main element of the functional OPS5 considered separately. First the compiler for the OPS5 language

is discussed. From this it will be seen how suitable functional languages are for building compilers. Then follows a discussion on the recognize-act cycle and the run-time system for the functional OPS5.

3.3.1. The OPS5 Compiler

Functional programming languages are particularly well suited to writing compilers and compiler tools and have been used successfully to write compilers for functional languages. Some examples are the Lazy ML compiler written in Lazy ML [Augustsson89], the Standard ML compiler written in Standard ML [Appel87], and the Hope compiler written in Hope [Burstall80]. Compiler tools have also been successfully written; for example, a Yacc parser generator in SASL [Peyton-Jones85], a lexical analyser generator [Jones86], and a mechanism devised for integrating parser definitions into CAML [Mauny89]. Recent work in this area includes the Chalmers Haskell compiler, built on top of their Lazy ML compiler, and the Glasgow Haskell compiler. These programs form the largest body of working functional programs.

The compiler for the functional OPS5 has been designed with three main parts; a lexical analyser, a parser, and a back end translator.



Lexical analysis converts the input from a list of characters into a list of tokens which the parser then uses. The lexical analyser was hand-coded and explicitly matches fixed input sequences. No special lexical analyser generator tools were used.

The parser is a set of functions that represent the formal grammar of the input language as closely as possible, with one function for each grammar clause.

The translator converts the parsed data into the output form (in this case, a sequence of productions). The translation is achieved by using an action for each grammar clause.

The desire was to make the compiler as simple and easy to comprehend as possible. The compiler built is a recursive decent compiler [Aho86] which has rewrite rules for each clause parsed. It is important to note that this style of compiler will not parse left-recursive grammars (since the parser would recurse infinitely). For example, the grammar clause:

expr ::= *expr op expr*

is an example of a left-recursive clause. The source language for OPS5 has an LL(1) grammar [Forgy81]. Thus, it was not necessary to convert the grammar clauses from a left-recursive form to a non left-recursive form.

A Framework to Represent a Formal Grammar

This section discusses a functional framework for building a parser for the OPS5 input language. From this it can be shown why functional languages are so well suited to the task of writing compilers. The grammar used will be a simple arithmetic evaluator which is often used in the compiler literature as an example to highlight the features of a compiler. However, in this case the grammar is a part of the OPS5 grammar. The full grammar can be found in the OPS5 reference manual [Forgy81].

In the grammar:

expn = *term* + *term* |
term - *term* |
term

term = *factor* * *factor* |
factor / *factor* |
factor

factor = (*expn*) |
number

the *concatenation* of terms is implicit whilst the *alternation* of clauses is explicit using the | symbol. The grammar can be made more explicit by representing *concatenation* with the AND symbol and *alternation* with the OR symbol. The more explicit grammar looks like:

expn = *term* AND + AND *term* OR
term AND - AND *term* OR
term

term = *factor* AND * AND *factor* OR
factor AND / AND *factor* OR
factor

factor = (AND *expn* AND) OR
number

A framework for a parser is defined such that there are functions which represent the notation of the formal grammar. This enables the conversion of a formal grammar into a working parser. The framework for the parser has functions that represent the grammar symbols OR and AND. These, respectively, are called `por` and `pand`. Also defined is a `terminal` function for parsing the terminals of the grammar. The grammar can be converted to a functional form using these framework functions, and will result in [12]:

[12] The form `` `f` `` is Haskell syntax for the infix application of `f`.

```
expn =  (term `pand` plus `pandl` term)        `por`
        (term `pand` minus `pandl` term)       `por`
        term

term  = (factor `pand` times `pandl` factor)   `por`
        (factor `pand` divide `pandl` factor)  `por`
        factor

factor = (lpar `pand` expn `pandl` rpar)       `por`
          get_num
```

This is only the outline of a parser and it cannot do any parsing yet. The addition of the actions to enable it to work will be considered later.

When building a functional parser, it is necessary to remember that functional programming makes things more explicit. In particular, there is no global place from which input can be collected, and, therefore, it must be passed into the parsing functions as an argument. Input also has to be returned from the parsing functions, together with any other data, as the parsing functions may take values from the input.

An algebraic data type is defined to represent values returned from a parsing function. These functions may return a value of any type as well as the input, so the data type is defined as:

```
data Parser_value a i = ...
```

where `a` is the type of the parsed value and `i` is the type of the input. The actual constructors for this type are not of importance here. A parsing function which takes some input and returns a parser value has the type signature:

```
parser_function :: [input] -> Parser_value a input
```

The main parsing functions are those defined in the framework of the functional parser, namely `por`, `pand`, and `terminal`. Both `por` and `pand` are higher-order functions that apply parsing functions to some input.

The function `por` takes, as arguments, two parsing functions and some input and returns a parser value. Each parsing function may return a result which has eaten some input or failed to parse and eaten no input, (the mechanism for choosing a parsed value is discussed in Aho, Sethi, and Ullman [Aho86]). In this compiler, `por` takes the first successful parse, but it could easily be replaced by a function that selected the longest parse or that returned every parse as in [Wadler85]. The type of `por` is:

```
por :: ([i] -> Parser_value a i) ->
       ([i] -> Parser_value a i) ->
        [i]                      ->
        Parser_value a i
```

The first two arguments are the same type as other parsing functions, and when `por` is applied to its first two arguments it has type:

```
por fn1 fn2 :: [i] -> Parser_value a i
```

This is the same type as other parsing functions, and therefore it is a higher-order function which can be passed to another parsing function.

The function `pand` is similar to `por` except that it returns the result of both parses. It has type:

```
pand :: ([i] -> Parser_value a i) ->
        ([i] -> Parser_value b i) ->
         [i]                      ->
         Parser_value (a,b) i
```

When applied to its first two arguments, `pand` can also be used as a higher-order argument to other parsing functions, as in the grammar for the expressions.

Every parser needs a mechanism to collect terminals from the input, and one has been defined. The parsing function `terminal` takes as an argument a terminal symbol which it expects to find in the input stream. It then returns a parser value. The type of the terminal symbol and the input must be the same, giving `terminal` the type:

```
terminal :: i -> [i] -> Parser_value i i
```

When `terminal` is given its first argument, it can be passed to other parsing functions.

Once the input has been parsed, the compiler will take an appropriate action. Within the functional compiler framework an `action` function has been defined. The `action` function takes a higher-order parsing function which returns a parsed value, a function to do some action on the parsed value, and some input. A modified parsed value is then returned by the `action` function, which has type:

```
action :: ([i] -> Parser_value a i)              ->
          (Parser_value a i -> Parser_value b i) ->
           [i]                                   ->
           Parser_value b i
```

A support function for `action` is the `as` function. It checks to see whether a parser returned a failed parse value. If it did, then `as` returns the failed parse, otherwise it calls a function to process the successful parse. Using this mechanism, failed parse values can be propagated through the compiler to parsing functions that wish to catch errors and successful parse values can be processed in the place in which they are collected. The `as` function takes two arguments – a function to process the parsed value and the parsed value – and then returns a new parsed value. `as` has type:

```
as :: (Parser_value a i -> Parser_value b i) ->
       Parser_value a i                      ->
       Parser_value b i
```

To complement `as`, the function `return` is defined, which allows a new value to be returned as a parsed value. `return` takes the new value and an existing parser value, and it returns a new parser value. `return` is used as the first argument to `as`, and has type:

```
return :: v                  ->
          Parser_value a i ->
          Parser_value v i
```

Further functions used in the processing of the action are `dollar0`, `dollar1`, `dollar2`, etc. which pick the n^{th} element from a parsed value. The parsing functions, together with `action`, `as`, `return`, and the `dollar` functions are usually combined in the following way:

```
parsing_function
    `action` (\p ->
        return(f (dollarn p)) `as` p)
```

The parsing function does the parsing and `action` applies the action `(\p -> ...)` to the parsed value. The function `as` checks to see if p is a failed parse or not. If it is, then `as` returns the failed parse, otherwise it applies the function `return (f ...)` to the parsed value, which causes a new parser value to be returned

Using The Framework

The functional framework is used in the OPS5 compiler, where each grammar clause is represented with a unique algebraic type. This is a benefit when writing a compiler because each parser function and its associated action is strongly typed. This enables the functional language type-checker to test the type of every function in the parser for consistency. As a result, any errors that may have occurred in the writing of each action can be found at compile time rather than at run time. By using this technique in the OPS5 compiler and other parsers it has been found to reduce run-time errors in parsers quite substantially. The data types for the OPS5 grammar, with the arithmetic expression terms shown in particular, are:

```
data Production = Production PName LHS RHS

data LHS = LHS [Condition_elem]

...

data RHS = RHS [Action]

...

data Expn = PExpn Term Term |
            MExpn Term Term |
            SExpn Term

data Term = TTerm Factor Factor |
            DTerm Factor Factor |
            STerm Factor

data Factor = BktExpn Expn |
              NFactor Int
```

The grammar of the OPS5 compiler is now presented, with particular attention paid to the arithmetic expressions:

```
prod :: [OPS5_tok] -> Parse_val Production OPS5_tok
prod = ((lpar `pand` terminal "p" `pand` name `pand`
           lhs `pand` terminal "-->" `pand` rhs `pand` rpar)
             `action` (\p ->
                 return (Production (dollar3 p) (dollar4 p)
                                                  (dollar6 p)) `as` p))
        .
...


expn :: [OPS5_tok] -> Parse_val Expn OPS5_tok
expn =  ((term `pand` plus `pandl` term)
            `action`  (\p ->
                return (PExpn (dollar1 p) (dollar3 p)) `as` p))
        `por`
        ((term `pand` minus `pandl` term)
            `action` (\p ->
                return (MExpn (dollar1 p) (dollar3 p)) `as` p))
        `por`
        term `action` (\p ->
                return (SExpn (dollar0 p)) `as` p)
```

```
term :: [OPS5_tok] -> Parse_val Term OPS5_tok
term  = ((factor `pand` times `pandl` factor)
            `action` (\p ->
                return (TTerm (dollar1 p) (dollar3 p)) `as` p))
        `por`
        ((factor `pand` divide `pandl` factor)
            `action` (\p ->
                return (DTerm (dollar1 p) (dollar3 p)) `as` p))
        `por`
        factor `action` (\p ->
                return (STerm (dollar0 p)) `as` p)


factor :: [OPS5_tok] -> Parse_val Factor OPS5_tok
factor = ((lpar `pand` expn `pandl` rpar)
            `action` (\p ->
                return (BktExpn (dollar2 p)) `as` p))
         `por`
          get_num `action` (\p ->
                return (NFactor (picknum (dollar0 p))) `as` p)
```

The action for the parser function converts parsed values of one type into values of the type returned by the parser function. Consider the function `expr`, which returns the type `Expn`. It will parse using the function `term`, which returns the type `Term`. The actions in `expr` return results using the constructors `PExpn`, `MExpn`, or `SExpn`. If any data given to one of these constructors were not of type `Term`, then the type checker would complain. By having a new type for each grammar clause, the type checker of the functional language compiler can determine errors in the parser. If the parser had just one type throughout the code, it would be possible to introduce more errors at run time.

Work similar to the parser presented in this thesis has been done by Hutton [Hutton90]. Hutton uses the techniques devised by Wadler [Wadler85] to build parsers

which also use higher order functions to represent the formal grammar. Hutton's use of Wadler's techniques have not been designed for large compilers, although Hutton demonstrates that they can parse non-trivial grammars. In Hutton's parsers, the tasks of parsing and semantic action are merged together. That is, some of the semantic actions are done as part of the parsing and some are done as a rewrite rule. Hutton represents *concatenation* in the formal grammar as three functions, namely `then`, `xthen`, and `thenx`. The `then` function works in a similar way to the `pand` function defined in this thesis. However, the functions `xthen` and `thenx` throw away the first or second parse, respectively, after a parse has succeeded. This contrasts with the parser in this thesis, which explicitly has an *action* for each parse that is responsible for manipulating parsed values. In this thesis, the two issues of parsing and rewriting have been successfully separated. There is only one function for *concatenation*, namely `pand`. Consequently, the parser closely represents the specification of the formal grammar, having all the semantic actions in a separate rewrite rule. Furthermore, by maintaining the discipline of using a separate algebraic type for each grammar clause, many errors can be detected and identified at compile time (unlike Hutton, who may not detect these errors until run time).

The combination of regular higher-order functions for the parser and the use of strong typing provides a framework for building large parsers. It is simple to convert the grammar into a parser and then to construct the rewrite rules. If Hutton's technique is used, then the parser and the construction of the semantic actions need to be considered at the same time. This increases the scope for errors, rendering Hutton's technique less suitable for large parsers.

3.3.2. The Recognize-Act Cycle

The *recognize-act* cycle of a rule-based system is usually said to occur in the following order:

i) **match** — which evaluates the LHS's of the productions to determine which are satisfied given the current contents of working memory.

ii) **conflict resolution** — which selects one production with a satisfied LHS. If no productions have satisfied LHS's, then the system halts.

iii) **act** — which performs the actions specified in the RHS of the selected production.

The following sections show how match, conflict resolution, and act have been implemented in a functional language. The section "The Run-Time System" presents how each of these parts is used and called with the right arguments.

Match

During the match phase, the OPS5 interpreter determines every instantiation of every production. Furthermore, if any of the productions can be instantiated by more than one list of working memory elements, then the interpreter finds every valid list of elements and puts these instantiations into the conflict set.

The function `do_match` matches all productions with all of working memory. It uses a list comprehension to do a cross product over production memory and working memory. For each production, tuples of working memory elements are generated which have the same number of elements as the number of clauses in the rule's left hand side and are matched against that rule's clauses. Every rule that has all its clauses matched successfully will go into the conflict set. The working memory tuples are generated on each loop of the *recognize-act* cycle:

```
do_match::PM->WM->[Conflict]
do_match ps ws
    = select_conflict_set [(match_rule wm_el a_prod) |
                                a_prod <- prod_list ;
                                wm_el  <- wm_lists !! (index a_prod) ]

      where                                 -- !! is the list index function
      prod_list = pmget_as_list ps          -- get PM as one list
      wm_lists = wm_cross_product ws        -- all WM tuples of all sizes
      index prod = length (get_lhs prod)    -- index is no of LHS clauses
```

Conflict Resolution

In the OPS5 user manual [Forgy81], the conflict set is defined to be a set of pairs in which each pair contains a production name together with a list of working memory elements satisfying the production's LHS. Conflict resolution examines this set to determine which instance dominates all others. The method for determining which is dominant is called the resolution strategy, and OPS5 has two of them — LEX and MEA. Each strategy has an ordered list of rules to follow, and is described in the OPS5 User's Manual [Forgy81].

The ability to create new types easily and to specify sequences of operations using function composition renders functional languages suitable for converting an ordered list of tasks into a functional definition. The method for converting a list of ordered rules into a function relies on a simple analysis of each rule. Consider the functions defined for the OPS5 conflict resolution strategy:

- *stage1* takes the conflict set history and the conflict set and removes all the instantiations from the conflict set that have fired already – this avoids unintentional (and potentially infinite) loops between rules, however the rule programmer is free to create his own explicit loops.

- stage2 sorts instantiations by the timestamps of the working memory elements in the instantiation tuple so that later stages can choose the instantiations with the newest timestamps. The process differs in the initial sort depending on whether the LEX or MEA conflict resolution strategy is being used and is split into 3 functions:
    - *lex_stage2_sort* sorts instantiations by the timestamps of the working memory element tuple, ordered with the newest timestamp first
    - *mea_stage2_sort* sorts instantiations by the timestamps of the working memory element tuple, with the timestamp of the first element of the tuple followed by the rest of the tuple which is sorted and ordered newest first
    - *stage2_order* sorts on two values. First on timestamp list, then by production name.
    - *stage2_select* selects only those instantiations whose timestamp lists are the same as the first member of the timestamp list.
- for *stage3*, if *stage2_select* returned one instantiation, then this is the selected item. If there is more than one item, then it is necessary to check the specificness of each production and choose the item which is most specific in the current context. The specificness of an instantiation is evaluated by counting the number of simple matches and variable matches in the original production. Instantiations are considered more specific if the count is higher; that is the production had a higher number of matches.
- in *stage4*, if there is still no obvious item, then an arbitrary item is chosen

The types for each function are given below:

```
stage1      :: [(Pname,[Timestamp])] -> [Conflict] -> [(Pname,[Timestamp])]
lex_stage2_sort :: [(Pname,[Timestamp])] -> [(Pname,[Timestamp])]
mea_stage2_sort :: [(Pname,[Timestamp])] -> [(Pname,[Timestamp])]
stage2_order    :: [(Pname,[Timestamp])] -> [(Pname,[Timestamp])]
stage2_select   :: [(Pname,[Timestamp])] -> [(Pname,[Timestamp])]
stage3          :: [Conflict] -> [(Pname,[Timestamp])] -> [Conflict]
stage4          :: [Conflict] -> Conflict_Resolution_State
```

These functions can be combined in a pipeline to generate the two functions needed for conflict resolution.

The LEX conflict resolution strategy:

```
lex :: [(Pname,[Timestamp])] -> [Conflict] -> Conflict_Resolution_State
lex cs_hist cs
    = (stage4. stage3 cs .
           stage2_select . stage2_order . lex_stage2_sort .
               stage1 cs_hist) cs
```

The MEA conflict resolution strategy:

```
mea :: [(Pname,[Timestamp])] -> [Conflict] -> Conflict_Resolution_State
mea cs_hist cs
    = (stage4.(stage3 cs).
           stage2_select.stage2_order.mea_stage2_sort.
               stage1 cs_hist) cs
```

Act

In the act phase of the cycle, the actions of the chosen production are executed one at a time in the order that they are written. These actions may add or delete elements from working memory, or they may do input and output.

3.3.3. The Run-Time System

The run-time system of the rule-based system is the infra-structure that binds the application together. Functions within the run-time system manipulate the items in the state, generating a new state after they are called. The *recognize-act* cycle is the main operation of any rule-based system. The form of this cycle is:

match → *conflict resolution* → *act*

Although previous sections demonstrated that no part of the system needs access to every item in the state, it is pertinent that the state be set correctly and in a predetermined order. It is necessary to represent the main *recognize-act* cycle in a functional way while still retaining its same operation. This can achieved by updating items of state as well as using the same ordering as in the original algorithm.

In the run-time system, equivalent functions are defined within the run-time system with the following types:

```
match     :: State -> State
conflict_resolve :: State -> State
act       :: State -> State
```

Each of the above is a function which does one part of the main cycle. When composed together, the result is:

```
act.conflict_resolve.match
```

From this it is evident that the ordering of operations can be achieved through function composition. The composition is of type *State* → *State*. The functions in this composition take the whole state, and then pass the relevant parts to a sub-function which does the real work. The value returned by the sub-function is set into the state, ready for when the next function begins.

By composing `match`, `conflict_resolve`, and `act`, each of which sets an item in the state, the behaviour of the original algorithm can be achieved. The result is a functional implementation equivalent to the recognize-act state-saving algorithm. Since the update of the state is hidden at this level, as opposed to having explicit manipulation

of the state, the result is a function written in a higher-order style which is elegant and hides the explicit plumbing.

The run-time system of the rule-based system has many other functions which are all of type *State* → *State*. This includes the compiler, which reads rules from a file, compiles them, and then puts them into production memory. There are many top-level functions which are equivalent to those in the original OPS5 run-time system. By having all the functions of the same type, new top level functions can be added easily. The problems of explicit plumbing are hidden through abstraction, enabling state-saving programs to be written in a functional style. There is no need for extra code for state at the top-level since the manipulation is done in the *State* → *State* functions. When using higher-order functions, an abstraction for manipulating state is created. This abstraction overcomes the issues of plumbing in the same way that `por` and `pand`, when used as higher-order functions, create an abstraction for building parsers.

An example of a state manipulation function is the function `match`. This takes production memory and working memory, and then returns an environment of matches for every production which could possibly fire next time. This environment is the conflict set. The match function used in the run-time system encapsulates the previously described matching function, called `do_match`, within a *State* → *State* framework. It can be written as:

```
match :: State -> State
match s = let conflict_set = do_match (get_pm  s) (get_wm  s)
          in
          set_conflict_set conflict_set s
```

The functions `set_conflict_set`, `get_pm`, and `get_wm` are defined in an abstract data type for state. `do_match` is the function that actually does the match. When `match` is completed, the state will have been updated with a new value for the conflict set.

Combining forms

For the functional OPS5 to interact with the user, a wrapper function is defined which retrieves the output of the system from the state for the user to see. This is achieved by applying the `get_output` function, which is of type *State* → *Output*, to the state of the system. Many functions are declared of type *State* → *State*. These are composed in order to operate on the state. To get the output to the user, the `get_output` function is applied to the state. The result is something like:

```
get_output ((fn. ... .f2.f1)  empty_state)
```

In the wrapper, functions are passed as a list in the order that they will be evaluated. The wrapper arranges for them to be composed so that they will then give the correct result. The wrapper can be written as:

```
execute :: [(State -> State)] -> output
execute = get_output.(foldl applys empty_state)
          where
          applys st command = command st
          applys :: State -> (State->State) -> State
```

The function `execute` composes all the functions in its argument list and gets the output at the end. A system can then be run with:

```
execute [load_productions "file",
         conflict_resolution_strategy mea,
         make "(make start)",
         run]
```

This produces the desired effect of executing a rule-based system.

Input and Output

Input and output are also considered to be problematic in functional programming because there are no side-effects and it is not obvious how to do both of them from the middle of a large application. By having the input and the output streams held as items

in the state, they both can be easy manipulated.

In a large application it may be necessary to do input or output at any time. In order to do this, access to both the input and the output stream is required. The particular functions required to do input and output may be buried deep in the application, so the streams must be passed down there. Since there is already a mechanism for passing items around, the state-saving structure is ideal for input and output streams. The alternative is to pass the streams around separately. This can lead to complicated control structures and to the loss of the previously seen functional cleanliness.

In the rule-based system output is produced in many places. Having the output stream in the state combined with the abstract data type functions makes it simple to do output. Any function that needs to do output can affect the output item in the state. Two functions for doing output are defined: `add_output`, which adds some new output to the end of the existing output, and `reset_output`, which sets the output to nil. The function `add_output` is the most common and the safest to use. This is because any function currently doing output is often unaware of the other output previously done. The function `reset_output` is only used in the top-level wrapper.

The section "Combining Forms" demonstrates how state update functions can be composed to affect multiple state items. If one of these state update functions is `add_output`, then it seems to behave like a print statement in an imperative language; for it is buried in a large expression and looks as though it is unrelated to the output stream. Consider the example:

```
(set_item1 new . add_output "hello world" . set_item2 val) state
```

in which the `add_output` expression is detached from any obvious plumbing (in fact, the plumbing is implicit in the function composition). Input to the rule-based system is handled in the same way. It is held in the state and manipulated by the abstract data type functions. If the application needs any input, it can get it from the state. Input is always eaten from the beginning of the input stream, and any input eaten is removed by rewriting the input stream.

Problems with I/O

As previously stated, input and output are considered problematic in functional programming. In the application in this thesis there is a problem with output. The observed behaviour is that no output is produced until the system stops. Then, once the system stops, all the output appears. This can be perturbing, especially since the more common result is for output to appear gradually. However, this behaviour can be explained, and a solution found which gives the behaviour we desire. By analysing the way the system was built, its operation can be predicted.

A solution to the problem of state being held up until the end of the program run relies on only *appending* new output to old output, because if new output is prepended anywhere, the results may be unpredictable (i.e. wrong). A top-level wrapper function can be used to ensure that new output is only appended to old output. If functions are not composed but, instead, the output from each of the functions f_1 to f_n is collected as each function is evaluated, then the result is the desired behaviour of output appearing as it is generated.

An alternative approach to the wrapper function is:

```
new_execute :: [(State -> State)] -> State -> Output
new_execute [] st = []
new_execute (command:rest) st
        = get_output this_run ++ new_execute rest new_state
          where
          new_state = reset_output this_run
          this_run = command st
          this_run :: State
```

The function `new_execute` could be started with an initial state of `empty_state`. The output for each update function is collected and prepended to the rest of the processing. Before continuing, the output is reset to avoid outputting the same thing twice. The functional simplicity of the former case has gone. Although both cases are semantically equivalent, they have a different run-time behaviour. The temporal behaviour of functional systems cannot be expressed as part of the program and can be

difficult to determine, especially in large programs. There have been undocumented reports of other attempts at large functional programs which have similar problems.

A second solution devised has a cleaner interface. A function is defined which generates a list of states that correspond to each state-update function which needs to be executed. From that list, the output can be collected and concatenated in order to provide output as it is generated. First, a function must be defined that generates a list of states. This list is generated in the order in which the state-update functions are called, thereby eliminating any hold-up of the output. This function is defined as:

```
statelist::[(State -> State)] -> State -> [State]
statelist [] st = []
statelist (command:rest) st
    = newstate : (statelist rest newstate')
      where
      newstate = command st
      newstate' = reset_output newstate
```

This function must also reset the output stream in order to avoid incorrect output. The wrapper can now be defined as:

```
newer_execute::[(State -> State)] -> State -> output
newer_execute fns st = concat.(map get_output).(statelist fns st)
```

If a clean functional interface to state manipulation is created, there is difficulty with output. However, this can be overcome, to a degree, by having a wrapper layer at the highest level which collects output as early as possible. It is not yet clear how this difficulty will manifest itself if output interleaves with input further into the bowels of the system. It may be that the simplicity completely disappears. Further work can investigate this issue. In particular the use of monads [Wadler90] and I/O combinators [Dwelly89] can be evaluated, as both allow an abstract framework to be created within the whole program structure. The current *State* → *State* functions do not provide a controllable method for collecting input from the input stream when input and output are interleaved within a single *State* → *State* function. Using either I/O combinators or

some specially designed monads could allow a framework to be built which overcomes the interleaving problem.

3.4. Executing OPS5

A working version of OPS5, written in Miranda, was built for this thesis. By executing some test programs that are present in the LISP source of OPS5, the functional OPS5 was evaluated. (Appendix C has an example of one of these programs). The functional OPS5 executed the productions of the test programs corectly. However, executing the functional OPS5 on a sequential machine was very slow because a non state-saving matcher was used. If a state-saving matcher, such as Rete, had been used one would expect to see a significant difference in the run-time performance, going from cubic on the size of working memory to independent of the size of working memory, as was discussed in chapter 2. In this section, a summary of the performance of different versions of OPS5 is shown in table 3.2.

In this research it has been discovered that it is difficult to run the functional OPS5 on a parallel machine due to problems such as:

i) availability — there are few machines built to execute parallel functional programs and even fewer with accessibility.

ii) different language — the only machine that was available was the GRIP machine [Clack85a] which has been installed at Glasgow University for general use [Hammond91]. The GRIP system uses either Lazy ML or a Haskell subset. There were no translation tools to convert Miranda into either Lazy ML or Haskell. Hand translation is possible but the need never arose as the matching algorithm used was too inefficient and the GRIP system was found to be unsuitable for the task required.

iii) non-trivial to use — GRIP requires hand annotations to harness parallelism. To do this for many thousands of lines of code is very time consuming and unlikely to be optimum or correct.

	Imperative	Functional
Sequential	A version of OPS5 written in Franz Lisp was executed in both interpreted and compiled form. It used the Rete matcher.	The Miranda version was executed in interpreted form. It used a non state-saving matcher and was therefore very slow.
Parallel	I have not personally tested any parallel versions of OPS5 due to both the hardware and software being unavailable. Some so-called parallel versions of OPS5 have only ever been tested on simulators. Further details of these parallel implementations can be found in the references shown in chapter 2.	The functional version has not been tested on any parallel machine. The reasons for this are discussed in chapter 6.

Table 3.2: Summary of performance of OPS5

These issues will be discussed more fully in chapter 6, in which parallel functional programming is considered and the problems encountered are discussed.

Furthermore, it was discovered that the measurement tools and techniques are thoroughly inadequate for observing the behaviour of an executing functional program. The number of graph reductions and the time in cpu seconds presented by functional run-time systems is of little use because it indicates nothing about the behaviour of the running program. A graph reduction on one machine may do substantially more work than a graph reduction on a different machine even though they are both required to do identical tasks. Graph reductions can be considered a similar measurement to MIPS in that they are not a reliable indication of real performance.

3.5. Analysis of the Functional Rule-Based System

The design, implementation, and execution of the functional rule-based system is a feasibility study of the practical use of functional programming from both a programming and an execution viewpoint. The programming viewpoint is a test to see if a large state-based application can be written effectively in a functional language. The execution viewpoint is a test to see if the functional application can be used on a day-to-day basis.

The functional rule-based system uses a simple non state-saving matching algorithm which has polynomial behaviour. This rule-based system consists of 5 main components:

i) *the compiler*

writing this re-enforced the view that recursive compilers are easy to build in functional languages. Much work has been done on functional languages and compilation as seen in section 3.3. This compiler is a simple recursive-decent compiler for a non-left recursive grammar.

ii) *the matcher*

this is the core of a rule-based system. This matcher uses a simple non state-saving algorithm whereby every clause of every rule is matched against every working memory element on every cycle. As stated, its behaviour is poor.

iii) *the run-time system*

this is the framework for the functional OPS5. It arranges for input and output to the program and binds the compiler and matcher together. It manipulates a large state object, which has all the data required by the program, such as production memory and working memory.

iv) *act*

this updates working memory and does input and output

v) *conflict resolution*

this uses a pipeline of functions to emulate an ordered list of instructions

There is a general misconception amongst imperative programmers that functional languages are unable to deal with state. This thesis refutes this claim with the proof of a working application. In this chapter a technique was demonstrated for writing applications which manipulate state. This technique combines using an abstract data type to represent state with a set of higher-order *State* → *State* functions. By using this technique, 90% of the rule-based system OPS5, which is an inherently state-saving application, has been successfully implemented.

Due to the desire to keep the compiler simple, the initial version of the compiler does not include error reporting or error recovery. Although this is sufficient for a prototype compiler, further work would be the implementation of a second version in which the lexical analyser and the parser support both error reporting and error recovery.

The fact that a functional language is being used to implement OPS5 presents both advantages and disadvantages. Depending on ones point of view the advantages for one person may be the disadvantages of another and we see that they are the same. The disadvantage is that *state must be represented explicitly and therefore the code must be redesigned.* As all state is explicit, the program code can look messy and thus lose the functional expressiveness that is expected. Imperative programs look much the same when state is added because state manipulation is implicit. The advantage is that *state must be represented explicitly and therefore the code must be redesigned.* There is explicit control over which parts of the state are passed and accessed, therefore implict state manipulation and generally accessible global store issues are overcome. Furthermore, an imperative implementation allows error reporting to be added as an afterthought. This has the disadvantage that error reporting and error recovery may suffer from incoherent design. In a functional system, error reporting and error recovery must be explicitly designed into the system.

Limitations of the Functional OPS5

The rule-based system created for this thesis is limited in comparison to the LISP version of OPS5. Only the main actions have been implemented, and the compiler is somewhat limited, giving few error messages and being unforgiving when errors do occur. These limitations have not been a problem because the original LISP system gives good error messages and can be used as a benchmark for any testing done. A group at Carnegie-Mellon University has implemented OPS5 in C. Their implementation has limitations which are similar to those found in the functional OPS5. This is because both Miranda and C treat programs and data differently, whereas LISP treats them the same.

In the functional OPS5 there are problems with:

- doing I/O, as seen previously.
- bugs buried deep in the system which were hard to find, because of a lack of debugging tools.
- measuring the performance of the system. Because neither the time spent in functions nor the space used can be measured, it is impossible to compare this system with other implementations of OPS5.

Benefits of Functional Programming

The features of strong typing, the creation of new data types, and higher-order functions in functional languages make applications such as compilers easier to write than in imperative languages. Pipelining (via function composition) aids in the building of algorithms, for example:

- in the compiler, the data types used for the simple OPS5 matcher were extended by adding new functions to convert the structure into a new form. The code for the original compiler was untouched.
- in the run-time system, numerous *State* → *State* functions were composed

- in the conflict resolution, each section of the definition was converted to its own function. These functions were given their own data type and then combined in a pipeline to form a working algorithm.

In general the use of pipelining and abstract data types are an effective way to write large programs. The term *pipelining* rather than *function composition* is used because it is possible to impose an abstract framework on the program which looks like function composition but is not. For example, monads may be used to pipeline functions, as will be seen in chapter 4. Expressions can be arbitrarily complex, and can be easily combined with one another. In imperative languages, there are commands and expressions which cannot be easily combined because expressions return values and commands do operations. Functional languages present a uniformity to the programmer.

3.6. Summary

The lessons learnt from writing a large application in a lazy, higher-order functional language are:

- pipelining combined with well considered data types can be used to do an ordered set of operations, e.g. conflict resolution.
- abstract data types can aid expressiveness, e.g. state manipulation or parser values.
- higher-order functions and laziness aid modularity. It is possible to write a general algorithm rather than selecting an arbitrary, yet large, number of instances of an algorithm. This begs the question "why aren't there more Haskell libraries around" ?
- functional programming is very good for writing compilers because the formal grammers can be easily encoded into a functional form.
- using functional programming for state means that there is explicit control over the state rather than implicit control. This can be used to great effect by:

a) limiting access by passing around only the parts required.

b) creating structures from a state. Stacks of state can be used for undoing operations. e.g. `save-excursion` in emacs.

- one can get natural looking code for many applications, e.g. compilers and graphics, as seen in [Henderson82] or [Arya89].
- one can get a prototype of a program working quickly. There is no fiddling with little things such as pointers to pointers.
- functional programming is easy to learn and easy to start building complex applications. This observation was supported by comparing the accomplishments of first year Unversity students learning Pascal with those learning Miranda. The students that learnt Miranda were able to solve much more complex problems than the students who learnt Pascal.
- the conversion of some well known algorithms can be difficult. Algorithms are usually defined in an imperative way. However, by reinterpreting the definition more abstractly, a functional implementation can be devised.
- easy to extend existing code by pipelining and function composition.

Chapter 4

4. Issues Arising in Functional Programming

One of the main aims of this thesis is to design and implement a rule-based system in a functional language and then to compare its performance with an existing rule-based system. This chapter discusses the software engineering issues that have arisen whilst writing such a large application in a lazy, functional programming language. Many of the issues discussed go some way to dispell various negative viewpoints held about functional programming. However, some re-enforce these negative viewpoints, and it is these issues which must be resolved if functional programming is to progress.

By writing a large functional program as part of this research some interesting aspects of functional programming have been discovered. These aspects are related to:

- algorithms and data structures
- development environments

This chapter discusses these two aspects and the issues that have arisen with respect to them. First, the issue of state is considered as this is an essential aspect of all programming systems. This leads into a discussion on monads, a theoretical concept that has been adapted as a possible way to deal with building a framework for state manipulation in functional programs. Then follows a section on vectors, a common data structure in imperative languages that is missing in functional languages. The section on graphs highlights the difficulties of functional languages in implementing certain algorithms that are easy to implement in imperative languages. The next section discusses the limited interaction of functional languages with the operating system which makes getting input and output into large applications non-trivial. Finally, there

is a discussion on the lack of measurement and debugging tools for functional programs, and how this hinders functional programmers from making their programs more effective.

4.1. State

One of the most important aspects for any programming system is that of state. Mechanisms for accessing, updating, and passing state are available in most languages. In imperative languages these mechanisms are usually so transparent that many programmers rarely give them much consideration. However, state manipulation has been a difficult issue in functional programming. As functional programs must make state explicit in all functions which need access to state, and because these functions return updated state objects, we seem to be *plumbing* in the state [13]. This plumbing can make programs look unwieldy and inelegant because the main operation of functions is obscured by the numerous details pertaining to state manipulation. This is especially true when the number of state items is large. Yet plumbing is essential in state-saving functional applications. It is not possible to write a set of functions and then add an extra argument which holds state. Items in the state are manipulated and the functions then return whole state objects.

Plumbing is not needed in imperative languages because the concept of state is different. The concept of state in an imperative language is a combination of three features – the value of the state, side-effect, and updatable store:

- the value of the state is the current condition of some structure, either a base type such as an integer or some composite type such as a tuple. The term *state* will be used to mean this within this thesis
- updatable store is a location in an environment that may have its value changed during the execution of a program. Functional systems do not

[13] This process is called *plumbing* because the state has to be passed into every function explicitly, and then explicitly passed back. The extra work required is similar to fitting piping that feeds water to different rooms in a house.

have updatable store.

- a side-effect is a procedure that causes some operation to occur secondary to the main operation of a function. This usually manifests itself as an update to some store or as some input or output. Functional systems do not have side-effects.

The combination of these three concepts allows imperative programmers to write programs such that state manipulation is transparent and plumbing is not needed. Although the above are sometimes considered to be related in imperative systems, in functional systems they are different.

Programmers who use imperative languages often fail to see how state can be represented in a functional language. This is because functional languages have single definitions, and imperative programmers are used to a computational model which encourages the use of updatable store through side-effects. Often what is overlooked is that the run-time binding of values to the formal parameters of a function provides a mechanism which is similar to the imperative model. In the functional model, state is expressed explicitly as an extra parameter to a function [14]. This differs from the imperative model, in which state can be expressed implicitly by using a global variable. Furthermore, in the imperative model, names are associated with locations in the updatable store, but in the functional model, names are associated with parameters to a function.

The state required for a rule-based system is relatively large, comprising hundreds of rules and thousands of working memory elements. In the rule-based system used in this thesis there are twelve items of state to pass around. The rule-based system runs from cycle to cycle, saving and updating different items of state as it executes. The following sections present the techniques discovered which show how state manipulation need not be a problem in functional languages. The amount of plumbing which is required can be reduced and the access and the update mechanisms

[14] The use of higher-order functions and currying can make extra parameters seem to disappear.

streamlined. The techniques presented allow functional programmers an elegant approach to state access and state update.

4.1.1. Manipulating State

Many functional programs contain small amounts of state and manipulate this state effectively. This can be demonstrated by considering a program which generates a set of stars in a pyramid. The pyramid starts with a single star in the first row. With each consecutive row one more star than the previous row is generated. The program builds rows of stars up to a given value. The state items required here are 1) the number of stars required per line and 2) the maximum number of stars required for the pyramid.

To code this in ANSI C, one could write:

```
stars(int max)
{
    int current;

    for(current = 1; current <= max; current++)
        generate_stars (current);
}
```

The variable `current` is state-saving. Its value is used to control a loop, and it is updated on every iteration of that loop.

To code this in a functional language such as Haskell, one could write:

```
stars :: Int -> [Char]
stars m = stars' 1 m

stars' :: Int -> Int -> [Char]
stars' current max = [], current > max
                   = generate_stars current
                             ++ stars' (current + 1) max, otherwise
```

Both the C and the Haskell programs produce the same result. In the C program the variable `current` is updated in place, but in the Haskell version `current` is passed to a new instantiation of `stars'` with a new value, namely `(current + 1)`.

To write code with a similar structure to the ANSI C, one could define a higher-order operator like `for`, to get:

```
stars :: Int -> [Char]
stars max = concat
                (for 1 (<=max) (+1)
                        generate_stars)


for :: a -> (a -> Bool) -> (a -> a) -> (a -> b) -> [b]
for value done next f
        = [] , if not (done value)
        = f value : for (next value) done next f , otherwise
```

In this example, the function `for` applies the argument function `done` to a current value in order to decide if the for loop has finished. `for` recurses with a new value which is created by applying the function `next` to the current value. The result is a list of values. The explicit control variable `current` of the imperative program has been eliminated from the `stars` function. The functional "for loop" produces a list of results which have to be concatenated to produce one list.

Another technique for manipulating state which is familiar to functional programmers is that of accumulating parameters. Using this technique, a parameter which accumulates a value acts as an updatable variable (i.e. the state). This is highlighted by the following commonly used function `rev`, which reverses a list. This can be written in Haskell as:

```
reverse::[a] -> [a]
reverse l = rev l []

rev::[a] -> [a] -> [a]
rev [] acc = acc
rev (h:t) acc = rev t (h:acc)
```

Here the first argument to `rev` changes on each instantiation. It has a new value *cons* ed onto it, this being the accumulating parameter.

From the previous examples it can be seen that functional programs treat state by passing it around explicitly. No problems are encountered here because each state held in the parameters `current` (in the `stars` example) and `acc` (in the `rev` example) is local to the recursive computation. When a single state item is needed beyond the scope of one function, or if more than one state item is required, new issues arise.

State-saving appears to be increasingly problematic when more than one value must be remembered. This is encountered, on a limited scale, with functions that take and return tuples of state values. The lexical analyser is a pertinent example of this. In a functional implementation one might have a function that takes some input, and then returns an eaten token combined with the remaining input. For example :

```
lex :: [Char] -> (Token, [Char])
lex input = (tok, rest)
            where
            tok  = get_a_token input
            rest = drop_a_token input
```

The function `lex` returns both the current state of the input and the current token. Thus, state has to be manipulated by the function that calls `lex`. Functional programmers are usually happy with, and capable of, this sort of processing.

The state-saving that is considered to be more difficult is when multiple items of state can be updated at any time. This type of state-saving is described briefly in [Hudak89]. Hudak's example is rather limited in that his state object is a 2-tuple. He

defines two update and two access functions for his state object:

```
x (xval,ival) xval' = (xval',ival)
i (xval,ival) ival' = (xval,ival')
x' (x,i) = x
i' (x,i) = i
```

Hudak uses these functions as an example to dispell the myth that state-saving cannot be done in a functional language. He uses an abstraction for state access through the functions `i'` and `x'`, and an abstraction for state update through the functions `i` and `x`.

More extensive use of state manipulation can be found in [Dwelly89]. Dwelly defines dialogue combinators, which are higher order functions used for manipulating I/O streams when defining user interfaces. This works successfully for the I/O streams, but he uses small tuples with pattern matching for state manipulation. In his system, state is just a 2-tuple containing a brush size and a colour. Consider an example which changes the colour of a pen:

```
ChangeColourRed (brush,colour) (input:rest_input)
        = ([], (brush,Red), rest_input)
```

Note how this function takes two arguments and returns a 3-tuple containing some output, the new state tuple, and the rest of the input. Although this technique for manipulating state is fine when dealing with two or three state items, it is impractical when there are many more state items.

In the following sections the difference between pattern matching directly and using an abstraction is presented. The effectiveness of each will be seen, particularly when the number of state items is large.

Pattern-Matching

As previously shown, one can implement state manipulation as a tuple and use pattern matching to access or set items. Dwelly uses this style of implementing state in his work [Dwelly89] but, as previously mentioned, his state tuple is very small. Hudak

also uses a tuple and pattern matching, but he hides this in an abstraction. The use of abstraction, which is important in functional programming, will be described in the next section.

The use of pattern matching for state is a technique that allows functional programmers to express state manipulations on many state items at once. However, this technique is only effective for small tuples. Consider an example from the functional rule-based system, described in chapter 3, which has 12 items of state. For access to one state item, such as the production memory, the code would be:

```
get_pm :: OPS5_State -> Production_Memory

get_pm (pm,wm,cs,cs_hist,res_strat,instantiation,stfm,cycle,timestamp,debug,input,output)

       = pm
```

To update an item of state such as the conflict set, which is generated by doing a match on the production memory and working memory, a function which uses pattern matching and tuples can be written as:

```
match :: OPS5_State -> OPS5_State

match (pm,wm,cs,cs_hist,res_strat,instantiation,stfm,cycle,timestamp,debug,input,output)

    = (pm,wm,new_cs,cs_hist,res_strat,instantiation,stfm,cycle,timestamp,debug,input,output)

      where

      new_cs = do_match pm wm
```

The long names of patterns can be replaced by shorter names, however the significance of these names would then be lost. When using pattern matching and tuples, functions become messy and the lucidity of the code is lost as attention is drawn to the pattern matching rather than to the body of the function. Therefore, this style is unreasonable for large state objects.

Haskell wildcarding can overcome some of these problems by eliminating names which do not appear in the body of the function [15]. Consider the previous match example, where the use of Haskell wildcarding would produce code such as:

---

[15] This can also be eliminated in SML by using records

```
match :: OPS5_State -> OPS5_State
match (pm,wm,cs,_,_,_,_,_,_,_,_,_)
    = (pm,wm,new_cs,_,_,_,_,_,_,_,_,_)
      where
      new_cs = do_match pm wm
```

Although the code is neater, this style is still unsatisfactory because attention is still drawn to the pattern matching rather than to the body of the function. If this style is used in every function that manipulates the state, a program becomes difficult to comprehend (See "Abstract Machine Specification in Functional Languages" [Koopman90] as an example of this). Therefore, using pattern matching and tuples for large state objects is unrealistic.

Abstract Data Types

If an abstraction is used through the use of an abstract data type, functions for setting items and functions for accessing items in the state type can be defined. Each item within the state abstract data type has its own functions for setting and accessing its value. Each function that sets a state item takes the old state and a new item, then returns the whole new state. Each function that accesses a state item takes the state, then returns the single item. The empty state value must be defined because there will be times when the state value has not been set and there must always be a valid state. All items in the empty state must be valid for that type and should be reasonable initializer values.

The implementation type of the state is any concrete data type that the programmer feels is suitable. Yet this is hidden by using abstract data types, thus sparing the programmer the burden of explicitly pattern-matching the concrete type in every function that accesses state. Access to the state is only through the abstract data type functions [16].

---

[16] This technique is similar to classes in object-oriented languages, where components of a class are accessed via accessor functions.

An advantage of using abstract data types for state is that all the implementation details are hidden beneath a layer of functions. These functions allow the access and setting of the items in state to be done in a clean functional way. Moreover, if these functions have meaningful names, then the code becomes readable and lucid. For example, to update one item in a state one could write:

```
set_item1 val state
```

This function would return a new state with `item1` changed to `val`. If pattern matching were used instead, the lucidity of the code would be lost and the containing functions would become messy, long-winded, and difficult to comprehend.

A further advantage of sensibly implemented abstract data types is that update functions can be composed in order to perform multiple updates. This composition can be performed for any updates needed. In order for this to work, all update functions with their arguments must be of the same type, such as *State* → *State*. To update two items, say `item1` and `item2`, this could be expressed as:

```
(set_item2 new . set_item1 val) state
```

which will return a new state with both `item1` and `item2` updated.

If more items are added to the state object, then the underlying implementation type must change but the functions that access the state may stay the same. That is, only the functions which need access to the new items require change and any changes to the program will probably be minor. This is very *important* when developing large applications; having the correct interface to state can avoid wasted time and effort. By contrast, the pattern-matching mechanism is particularly painful. If another item of state is added, then one must extend both the argument pattern and the resulting pattern. This must be done for all patterns in every function which pattern matches on the state, even though the added item may not be part of the function's operation.

As stated, multiple updates to the state can be performed by composing updates to an original state. Although this is clean and readable, it introduces some new and undesirable features.

First, when reasoning about updates one might surmise from the composition of the updates that their ordering is important; it rarely is. The order of composed updates is usually of no significance. The emphasis is on the resulting state, which remains the same even if the ordering of updates is different. For example:

```
(set_item2 new . set_item1 val) state
```

which gives the same result as:

```
(set_item1 val . set_item2 new) state
```

can be interchanged freely, even though they may appear to be different. This apparent difference may reduce the lucidity of any code.

Second, one may get the impression that state update is a divisible operation which can be broken down into its component parts, i.e. one item of state is updated, then another item of state, and so on until the updating is completed. If this is the case, one might assume that it is possible to examine the state between each update of the items. However, state update is not usually meant to be a divisible operation. The desire is to update all the items at once. Therefore, it must be clear that updates are neither ordered nor divisible.

Third, the state update is sequentialized on the update functions. All references to the state go through these update functions. This could cause problems if there are many composed updates. This is particularly important in a parallel system where sequentialization reduces the available parallelism.

One solution to these problems is to avoid abstract data types and revert to using explicit pattern matching. If pattern matching is used, direct access to multiple items can be achieved and updating multiple items can be accomplished in one operation. Furthermore, using pattern matching with multiple updates appears to be an indivisible operation, thus obliterating any concept of ordering, sequentialization, and divisible updates. The programmer has to decide which technique is most suitable for his program.

An alternate solution is to devise a set of higher-order operators which make it explicit that there is no ordering, no divisibility, and that state access may be

parallelised.

4.2. Monads

After much of this research had been undertaken, a new concept from the theoretical side of functional programming was presented. The concept is that of *monads* and was presented by Wadler in his paper "Comprehending Monads" [Wadler90]. Monads were originated by Moggi [Moggi89] to provide a way of structuring denotational specifications of imperative programming language features such as state, exceptions, and continuations. Wadler has adapted Moggi's work into a technique for structuring functional programs. This section has an extensive description of how monads can be specified in functional languages and then shows some examples of using monads to clarify their ability. Following this is a description of how monads can be used to build a framework for state manipulation in a functional program.

In his paper, Wadler shows how monads may be used for manipulating state, exception handling, non-determinism, and representing continuations within a functional language. The state manipulation functions he describes are for fetching and assigning values to a state value bound up within a state monad. He shows some simple example applications of the fetch and assign functions. The paper was very effective as many in the functional programming arena persuaded themselves that the issue of state manipulation had been solved and that monads were the *only* way to do state manipulation. This is not the case. The discussion on monads will show that monads are good for structuring functional programs, and particularly good for abstracting this structure, regardless of the concrete types being passed and returned by functions. Yet there is nothing obligatory in using monads for state, although it could be useful to use them for programs that have some state manipulation components.

Monads can be used to create an abstract structure within which small changes can be made to the functionality of a program without fundamental structural changes being made to the code. Imperative programmers can already do this as imperative languages have features which allow and encourage such changes. However, functional programmers often need to rewrite major parts of their code when some small changes

are introduced. Two common examples are the addition of a single state variable or the desire to add debugging output.

To demonstrate the power of monads, a simple arithmetic evaluator is examined. It uses monads to create structure within the program. From this initial example two further demonstrations will be derived. One will add some error handling to the program and the other will add limited state manipulation. From these examples it will be seen how the use of monads in a program can make seemingly complex changes simple. These examples are used to clarify the practical uses of monads. These practical uses were unclear to many as Wadler's paper, although impressive, is rather theoretical. These worked examples will provide a basis for the discussion on monads for state manipulation.

A monad is a triple that consists of a type constructor `M`, used to create the monadic type, plus the two operations:

```
unit :: a → M a
bind :: M a → (a → M b) → M b
```

`unit` creates a monadic version of a value when passed that value, and constitutes a monad creating identity function. `bind` applies a function to a monadic value; it is the monadic version of postfix function application.

All monadic functions also have to satisfy 3 laws which are discussed in [Wadler91]. The laws can be summarised as:

```
    unit ;; f = f
    f ;; unit = f
f ;; (g ;; h) = (f ;; g) ;; h
```

The symbol `;;` is a function that represents monad composition such that `f ;; g` does `f` followed by `g`, where `f` and `g` are both monadic functions. The function `;;` can be defined as:

```
(;;) :: (a -> M b) -> (b -> M c) -> (a -> M c)
f ;; g = \a -> let mb = f a
                          in bind mb g
```

An example arithmetic evaluator has the grammar:

expr ::= *expr op expr*
| *number*

op ::= + | - | * | /

This can be represented with the following data types:

```
data Expr = Expn Op Expr Expr |
          .
              Constant Int
data Op = Add | Sub | Mul | Div
```

The evaluator takes an expression and evaluates it. A function to do this could be written as:

```
eval :: Expr -> Int
eval (Constant c) = c
eval (Expn op e1 e2) = do_op op (eval e1) (eval e2)
```

Now consider a version of the evaluator written using monads. As the structure of the evaluator is bound with monads, the evaluator's type reflects this. Instead of it being of type *Expr* → *Int*, as in the previous evaluator, it is of type *Expr* → *M Int*. The evaluation of a constant involves taking its value and returning it as a unit monad. The evaluation of an expression involves applying monadic expressions in a specific order using the `bind` function. The second clause of `eval` can be read as: evaluate `e1`; bind `v1` to the result; evaluate `e2`; bind `v2` to the result; apply the operator to both `v1` and `v2`; and finally end. The operator is applied in the function `do_op`, which also returns a monadic result type. The code for the monadic evaluator is:

```
eval :: Expr -> M Int
eval (Constant c) = unit c
eval (Expn op e1 e2) = eval e1  `bind` (\v1 ->
        .              eval e2  `bind` (\v2 ->
                           do_op op v1 v2 `bind`
                       end))
```

A monadic version of `do_op` can be defined as:

```
do_op :: Op -> Int -> Int -> M Int
do_op Add a b = unit (a+b)
do_op Sub a b = unit (a-b)
do_op Mul a b = unit (a*b)
do_op Div a b = unit (a/b)
```

where all returned values are monadic.

The first version of the monadic evaluator uses the simplest definitions for the monad triple. They are:

```
type M a = a

unit :: a -> M a
unit a = a

bind :: M a -> (a -> M b) -> M b
a `bind` k = k a
```

The mondic type `M a` is a synonym for the original type, the `unit` function is the identity function, and `bind` applies its second argument to its first. Two support functions are also defined. They are the `end` function, which returns the unit monad, and the `display` function, which prints a monadic value:

```
end :: a -> M a
end = unit

display :: M a -> String
display a = show a
```

Notice the difference in clarity between the two versions of `eval`. This highlights one of the drawbacks of monads, namely that clarity of expression is lost and that expressions are sequentialized by the `bind` function. The monadic version of the evaluator seems unnecessarily complex, but this can be beneficial as will be seen later. To test the operation of the monadic evaluator some test expressions are defined:

```
text_expr0 = Constant 666
test_expr1 = Expn Div (Constant 123) (Constant 7)
test_expr2 = Expn Mul (Constant 123) (Constant 7)
test_expr3 = Expn Sub test_expr2 test_expr1
test_expr4 = Expn Div (Constant 1) (Constant 0)
```

The evaluator can be tested with an expression such as:

```
(display.eval) test_expr1
```

The results of the 5 test expressions are displayed in table 4.1.

Expression	**Result**
`test_expr0`	666
`test_expr1`	17
`test_expr2`	861
`test_expr3`	844
`test_expr4`	Program error: Division by 0

Table 4.1: Results of first monadic evaluator

This evaluator fails badly with the expression:

```
Expn Div (Constant 1) (Constant 0)
```

because of a division by zero and results in the program terminating in an uncontrolled manner. This is a common fault in many programming languages.

One can see how the problem of dealing with the division by zero can be dealt with easily when using monads. The traditional approach to solving this problem in functional programming is to define a new data type for results and then to rewrite all the functions that use the new data type [17]. When using monads, a new monadic data type is defined and the monadic functions `unit` and `bind` are redefined. A change is also made to the evaluator in the *divide* clause of the `do_op` function. The new type definition is:

```
data Result a = Failed String | Success a


type M a = Result a
```

and the new definitions for `unit` and `bind` are:

```
unit :: a -> M a
unit a = Success a


bind :: M a -> (a -> M b) -> M b
a `bind` k = case a of
             Failed s -> Failed s
             Success v -> k v
```

The divide clause of `do_op` is changed to:

---

[17] In imperative languages a global variable is used to raise an exception which is processed later. This cannot be done in side-effect free functional languages.

```
do_op Div a b = case b of
                  0 -> fail "cant divide by zero"
                  _ -> unit (a/b)
```

The display function has to be changed, and a function to return a failure, as used in `do_op`, is defined:

```
display a = case a of
              Failed s -> s
              Success v -> show v


fail::String -> M a
fail s = Failed s
```

These changes are all that is required to add a safety mechanism into the monadic evaluator. The test expressions can be re-evaluated to give the results in table 4.2, with the evaluation of 1/0 being processed in a controlled manner.

Expression	Result
`test_expr0`	666
`test_expr1`	17
`test_expr2`	861
`test_expr3`	844
`test_expr4`	cant divide by zero

Table 4.2: Results of second monadic evaluator

One of the reasons monads were devised was to allow the specification of state manipulation within programs. In the next example, it will be seen how a state monad can be added to the arithmetic evaluator with the use of a single state object to hold a count of the number of operations performed by the `eval` function. Again, the differences to the original monadic evaluator will be presented in order to clarify how

little has to be changed when using monads. A new definition for the monadic type is created, together with definitions for `unit` and `bind`. The definitions used are those described by Wadler in the section on state transformers in "Comprehending Monads". A state transformer is a function which takes a state object and a value and returns a tuple with the value and the state object. For the following example, the state object is a count and the monad type is the state transformer. The type definitions are:

```
type Transformer a = Count -> (a, Count)

type Count = Int


type M a = Transformer a
```

and the definitions for `unit` and `bind` are:

```
unit :: a -> M a
unit a = (\s -> (a,s))


bind :: M a -> (a -> M b) -> M b
a `bind` k = \s0 -> let (v1,s1) = a s0
                    in k v1 s1
```

Other changes required are to the `display` function and to the `end` function. In the two previous examples the `end` function has been the monadic identity function. The `end` function is now redefined to be the function that increments the number of operations undertaken:

```
display :: M a -> Count -> String
display f = \s -> let (v1,s1) = f s in
                  ("Value: " ++ show v1 ++
                                "  Operations: " ++ show s1)


end :: a -> M a
end = incr_ops


incr_ops :: a -> M a
incr_ops a = (\s -> (a,s+1))
```

The test expressions can be re-evaluated using the revised evaluator to give the results in table 4.3

Expression	**Result**
`test_expr0`	Value: 666 Operations:0
`test_expr1`	Value: 17 Operations:1
`test_expr2`	Value: 861 Operations:1
`test_expr3`	Value: 844 Operations:3
`test_expr4`	Program error: Division by 0

Table 4.3: Results of third monadic evaluator

The values presented in table 4.3 show that the state holding object has been added to the program without the problems of plumbing that are usually associated with adding state values to functional programs. The results returned have both the required value and the number of operations, however the problem of division by zero is still present. The combination of both state manipulation and error handling could be put into a monadic version of the arithmetic evaluator if desired. The changes to make to the original monadic evaluator would also be small. Techniques for doing the combination of monads can be found in [King92]. The structuring that monads provide

is powerful and flexible, but the code produced when using monads lacks the lucidity and elegance of code which does not use them.

4.2.1. Sequencing with monads

Having seen monads being used for a state value in a small program, this section reconsiders how state manipulation functions are combined using function composition (as seen in chapter 3) to form an ordered set of commands, as used in the rule-based system, and then considers if there is an equivalence with monadic functions.

Observe that there is a relationship between the imperative style of statement ordering and ordering through state manipulation functions in functional programs. Given some imperative code such as:

```
{
        one;
        two;
        three;
}
```

where `one`, `two`, and `three` represent statements in the program, this can be expressed in the functional style as:

```
do [
        one,
        two,
        three
] :: State → State
```

The `do` function is a function which allows this state manipulation to be expressed in a familiar style, namely that of an imperative, block structured language. Each of `one`, `two`, and `three` are of type *State* → *State*, and the function `do` has type *[State* → *State]* → *State* → *State*. The function `do` is passed a list of *State* → *State* functions, which are applied in the order given. This provides the familiar syntax and layout seen in imperative languages. Alternatively, the list of *State* → *State* functions can be

composed together as:

```
three · two · one
```

which produces a composed function [18] of type *State* → *State*.

When using monads, state manipulation functions cannot be composed together as each of the monadic functions are of type *State* → *M State*. Therefore, a function has to be defined to allow *State* → *M State* functions to be combined. The function defined by Wadler, which is not solely for state manipulation, is the `;;` function. Remember that it has type:

```
(;;) :: (a → M b) → (b → M c) → a → M c
```

where `(f ;; g)` does `f` followed by `g`.

Note that `;;` has a type that is similar to the normal compose function, which has type:

```
(·) :: (a → b) → (c → a) → (c → b)
```

and where `(g · f)` does `f` followed by `g`.

Given the monadic compose function `;;` and some monadic state manipulation functions `oneM`, `twoM`, and `threeM`, where each are of type *State* → *M State*, then one can express a state manipulation as:

```
oneM ;; twoM ;; threeM
```

This produces a combined function of type *State* → *M State*. Wadler observes that to access the state from within the state monad, one needs to define a state reader function which has type:

```
state_reader :: M State → State
```

Using the state manipulation monads, one can now express the *State* → *State* function as:

---

[18] It is not possible to do higher-order function composition in an imperative language as many statements work by side-effect and not as a functional form.

```
state_reader · (oneM ;; twoM ;; threeM)
```

This shows that there is a structural equivalence between the composition of *State* → *State* functions and the use of monadic state manipulation functions, as both express an ordering of functions within a *State* → *State* framework. Both forms are interchangeable within a program, and therefore monads model this kind of computation well.

One of the problems that arose in the state manipulation functions in the functional OPS5 was that of misunderstanding what constituted a *State* → *State* computation. It may be assumed that a composed set of *State* → *State* functions worked in a specific order, or that the state between *State* → *State* functions could be analysed for meaningful data, even though this was not meant to be the case. When using function composition, this misunderstanding could not be solved. When using the monadic `;;` function, this problem could also arise. As an example consider:

```
oneM ;; twoM
```

This reads as do `oneM` then do `twoM`. As stated, it is sometimes desirable that no order is implied in these state manipulation functions. The abstraction that monads provide allows a function to be defined to alleviate the problem of implied ordering in state manipulations. A function could be defined as:

```
(any_order) :: (a → M a) → (a → M a) → a → M a
```

which, when used in the following way:

```
oneM `any_order` twoM
```

would express that there is no particular order in which `oneM` and `twoM` are combined. This is not possible with function composition using the `(·)` operator. Here `g · f` always means do `f` followed by `g`, although the function `any_order` could be defined in such a way to abstract function composition.

The problem of analysing state between function compositions is also alleviated when using monads because the value returned by each monadic state manipulation function is of type *M State* rather than of type *State*. This is persuasive enough to

prevent anyone assuming intermediate states are meant to be analysed.

It is important to remember that monads do not address the issue of accessing and updating many items in a large state, such as that seen in the functional rule-based system in this thesis. These techniques are still needed even if the state manipulation framework is to be built using monads.

4.2.2. Review of monads

The important thing to note is that the technique used in this thesis for state manipulation and that of monads are *complementary*:

- monads address the structure of a problem with state manipulation
- the technique used in this thesis addresses the issue of accessing and updating multiple state items

It is also important to note that both techniques can be combined in the same state manipulation parts of a program. State monads give a framework within which state can be passed around effectively by making it obvious that there is state. Monads would be just as cumbersome with 12 state items.

Wadler has stated that it is tedious to use monads, but it is easy to modify programs which have them when needing to change the behaviour of that program. This has been demonstrated in the arithmetic evaluator examples. Table 4.4 summarizes Wadler's view of using monads.

Good points	Bad points
More flexible than impure effects	Less efficient than impure effects
Makes obvious where effects occur	Makes it *painfully* obvious
Facilitates sequential style	May hinder parallelism
Pretty theory	Ugly syntax

Table 4.4: Wadler's view of monads

4.3. Vectors

In this section an argument is made for having vector manipulation as primitives within the functional language. A vector is a fixed-sized, same-type structure with fast access and fixed space usage. Vectors are present as arrays in imperative languages and have O(1) access time and O(n) space usage. One of the main reasons for having vectors in functional languages is that without them many applications will not achieve the speed required to match imperative programs and, therefore, functional languages will not be used for general-purpose programming. There is no need for this situation to persist. The definition of Haskell mentions *monolithic arrays* — these structures look like arrays but there is no guarantee that they have O(1) access or are of a fixed size. Haskell arrays can be generated lazily, and some of the array elements can have undefined values. Therefore, the name vectors is chosen to differentiate these structures from Haskell arrays.

Most functional systems implement data structures using *cons* cells. All compound types (such as TUPLES and PACK's in FLIC) can be implemented in this way. It provides a convenient and easy implementation technique such that allocating and garbage collecting cells becomes easier, therefore simplifying memory management. However, this simplicity can lead to inefficiencies with certain data structures or algorithms. Although small data structures can be pattern matched effectively, for example:

```
f (a, b, c)
```

or

```
g (Algebraic a b c)
```

large data structures are not effective to use if pattern matching is needed. Consider an example with a 26-tuple, which may look like:

```
f (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z)
  = ...
```

and is unwieldy. Now consider the need to use a 1000-tuple, where the effort required to manage these sorts of structures in a large program would be enormous.

If the elements are of the same type, then the data structure could be implemented as a list. If access is needed to any element, then the item can be accessed using the list index operator `!!`, such that:

```
list !! n
```

gets the n^{th} element of the list. This process involves O(n) pointer traversals which, if done continually on the same list, can be very inefficient compared to accessing O(1) structures.

In many cases, the overhead of building arbitrary length dynamic structures is not needed. There is often a case for fixed-sized, same-type data structures within programs. The power and flexibility of lists is not needed, it is the efficiency of vectors that is required. This is pertinent for this thesis as this requirement arises in the implementation of OPS5. Imperative languages have vectors in the form of arrays, which have O(1) access and O(n) space usage. Consider the `literalize` construct, which declares a data structure within working memory. The following `literalize` statement:

```
(literalize Class attr1 attr2 attr3)
```

asks OPS5 for a data structure with `Class` as the class name and 3 attribute pairs. These data structures are generated once, accessed many times, and never updated. It is desirable to have a data structure with a fixed size and O(1) access time available from

the functional implementation. This thesis argues that vectors are an essential data structure for functional languages.

Vector primitives are essential in functional languages for efficiency reasons. Without vector primitives, a functional version of any algorithm that requires O(1) access to fixed-type data structures can never match the speed of the imperative version using arrays. In particular, a functional OPS5 will always be slower than an imperative version regardless of any parallelism present.

There is little previous work in the area of lazy, higher-order functional languages and vectors. Many LISP systems have equivalent structures, but none occur regularly in lazy, higher-order functional languages. Recent work by Hartel and Vree analysed some case studies where vectors were added to their functional language [Hartel92]. They observed that lists accessed in order can be as efficient as accessing an array in the index order 0,1,... However, if the order of accessing the array is non-sequential, then arrays are more efficient. They deduce that this restricts the class of problems for which arrays are *better suited* than lists, namely to where the access order of the array is not 0,1,...

Hartel and Vree analyse some 1-dimensional fast fourier transform functions, some which use lists and others which use arrays. They conclude that:

- efficient implementation of arrays contributes significantly to the performance of functional languages
- the overhead of array construction can be too large in certain algorithms
- there is a distinction between array construction and array subscription

The last two points raise questions regarding the specification syntax of arrays. In their system, a Haskell-like notation is used which is clearly expensive at run-time.

Other work on vectors has been associated with parallel functional systems, but the main thread of this work has been the parallel systems themselves, and the vectors have been secondary. A notable exception is Jouret's work on data parallel functional programming [Jouret91]. An example where the vectors are secondary is the work by Robertson on evaluating some Hope+ test programs on the Flagship machine [Robertson89]. Robertson mentions that the Flagship instruction-set supports vectors.

After comparing programs which are written in both Hope+ and the Flagship assembler, he states that the Hope+ versions are much faster to write and modify. However, he observes that the assembler programs using vectors have the following benefits:

- they give impressive speed increases. nqueens is 10 times faster and a "triangle game", devised by Gabriel [Gabriel85], is 30 times faster with vectors than without vectors.
- there is fast access to data items
- they are more efficient than lists [19].
- the number of graph reductions was reduced, which in turn caused a small speed-up.

On evaluating a transaction processing benchmark, Robertson concluded that improvements in efficiency would have been possible if vector primitives had been available from Hope+.

Much work has been done in the parallel programming world using vectorizing compilers. Fortran compilers have been used in the numerical processing world and vector based hardware is often used to run vectorized Fortran programs [20]. This work seems to have been ignored in the functional world, perhaps because few researchers currently use functional programming systems for real work on parallel machines. An exception is Boyle and Harmer who use a functional language for vectorizing an application on a Cray [Boyle92]. In "Structured Parallel Functional Programming" [Darlington91], various points about vectors are presented:

- the Intel i860 is a parallel vector processor, but there is no suggestion that vectors be a built-in type which can be operated on in parallel by vector primitives.

---

[19] This was especially so when parallelism occurred, as the indivisible structure of vectors aided the locality of computation. Linked lists split across many machines can degrade performance significantly.

[20] Example suppliers of vectorizing compilers and hardware are: Alliant, IBM, Fujitsu, NEC, and other smaller niche manufacturers.

- there is some recognition that vectors are important. Some functions are presented using arrays and it is stated that the CM2 computer has built-in operations for arrays. However, these ideas are not elaborated into the functional programming arena.
- array operators are introduced. These are defined for moving data around the array of processors. However, array operators are not vectors as I propose, but are arrays of SIMD processors.

However, because no amount of parallelism can attain the speed-up lost by not having vectors, it is essential that vectors be included as a primitive within a functional language.

4.3.1. A Vector Data Type

Having considered why vectors are essential for functional languages, this section proposes a data type for these vectors and the next section proposes some primitive functions for vector manipulation.

The vectors proposed have type:

```
Vector α
```

where α is the type of the elements in the vector. Unlike Haskell arrays, the number of elements in the vector is not part of the data type. For vectors, the structure required has the following attributes:

i) it is of a fixed size, which is determined at run time

ii) the contents are created once, they are never updated

iii) fast access is guaranteed to all elements, namely O(1) access

This differs from arrays in imperative languages which have updatable store and allow the contents to be changed.

There are some alternatives to having vectors built in as primitives, but each one presents problems. The alternatives are:

a) Memoized functions. These are not so suitable because nothing is known about the structure of the memoised function at run time, this is up to the functional language implementation. There are no guarantees regarding the access time or space usage.

b) Vectored lists. The Miranda system will automatically and silently convert lists of a known size at compile time into a vector. This is a useful compile time enhancement, but if the size is unknown at compile time then this feature does not work.

c) Tuples. This is ineffective as one would need a new set of definitions for each size of a vector. For example:

```
type Vector_n = (elem1, elem2, ..., elemn)


getn 1 ...
getn i (elem1, elem2, ..., elemi, ..., elemn) = elemi
...
getn j (elem1, elem2, ..., elemj, ..., elemn) = elemj
getn n ...
```

Therefore there would be $O(n^2)$ access functions. These functions have to be written for all instances of i, from 0 to n. Furthermore, for each size of vector the same set of functions must be written.

4.3.2. Primitives for Vectors

This section describes the proposed set of primitives for manipulating *vectors*. The primitives have been designed to present a simple, yet flexible, interface to vectors. They should be easy to program with and easy to implement. The primitive functions have type:

```
vectorBuild   :: [a] -> Vector a
vectorGetElem  :: Vector a -> Int -> a
vectorSize :: Vector a -> Int
```

A vector is created using a `vectorBuild` primitive. The user does not supply the size of the vector, its size is determined from the size of the input list. The size is not an argument of the type constructor but is accessible through a primitive, namely `vectorSize`. To access an element of a vector, the `vectorGetElem` primitive is used.

The issue of changing a cell in a vector is addressed by having a vector copy primitive, which has the type:

```
vectorChange :: Int -> a -> Vector a -> Vector a
```

This creates a copy of the vector with one cell changed. This technique was chosen because:

(i) it can be executed quickly at run-time by doing a block copy plus an in-place update of one cell.

(ii) it saves converting the vector to a list, changing elements, and then revectorizing

(iii) it makes it possible to do the update of the one cell without doing a copy when suitable compilation and run-time techniques become available

Consider some simple uses of the vector primitives by worked examples. First, a vector can be created:

```
v = vectorBuild ['h', 'e', 'l', 'l', 'o']  :: Vector Char
```

which is a vector of characters. As stated, access to individual elements of the vector can be achieved with:

```
vectorGetElem v 1   => 'e' :: Char
```

which returns the 1st element of the vector. The size of the vector can be requested with:

```
vectorSize v   => 5 :: Int
```

which returns the number of elements in the vector. To make a copy of a vector with one element changed, the `vectorChange` primitive is used which creates a new vector. The issues relating to in-place update are current research in the functional programming world and, although they are not addressed directly by these primitives, they may be addressed in the future. An example of creating a new vector is:

```
vectorChange 0 'H' v  :: Vector Char
```

which creates a new vector whose 0^{th} element is the character 'H'. Multiple new vectors with multiple updated cells can be created by composing `vectorChange` functions. For example:

```
(vectorChange 0 'H' . vectorChange 1 'E' . vectorChange 2 'L') v   :: Vector Char
```

creates a vector with the characters 'H', 'E', 'L', 'l', and 'o'.

As stated, the primitives need to be simple yet flexible. Therefore, it is necessary to create useful functions using the primitives, such as a vector to list function. This does not need to be a primitive itself and can be written as:

```
vecToList :: Vector a -> [a]
vecToList v = map (vectorGetElem v) [0..(vectorSize v - 1)]
```

Another well known vector manipulation function is that of selecting part of a vector to generate a sub-vector. This can easily be expressed as:

```
subVec :: Int -> Int -> Vector a -> Vector a
subVec min max v = vectorBuild (map (vectorGetElem v) [min..max])
```

Once a functional language has primitive vectors, it is possible to have the run-time efficiencies required by an application such as OPS5. Again, consider the scenario in OPS5 where a working memory element is declared as:

```
(literalize Class attr1 attr2 attr3)
```

This could be represented as a vector of length 4, each element being some pre-defined OPS5 type, so that a working memory element could have type:

```
Vector OPS5cell
```

For this scenario, imagine that there also exists specific working memory elements within working memory expressed as:

```
(Class ^attr1 5 ^attr3 10)
```

This can be encoded using the vector primitives:

```
wme = vectorBuild [WMEstr "Class",
                   WMEnum 5,
                   WMEnil,
                   WMEnum 10]          :: Vector OPS5cell
```

Furthermore, if there were a production condition such as:

```
(Class ^attr1 = 5)
```

this could be converted into a matching function that uses vectors. A code segment to do this could be:

```
match_prod_clause wme = vectorGetElem wme 0 == WMEstr "Class" &&
                        vectorGetElem wme 1 == WMEnum 5
```

By using vectors, each match would execute more rapidly as each call of `vectorGetElem` has O(1) time complexity, as opposed to O(n) when using lists. This would speed up a functional implementation of OPS5 dramatically.

4.3.3. Other uses of vectors

Vectors can be used for other purposes in functional programs where $O(1)$ access is important. One obvious example is hash tables. Traditionally, hash tables are used in order to speed up access to large data spaces by using both a table of values and a hashing function that converts a value into a hash table index. Hash tables are generally faster to access than lists or trees, but in a functional programming system without vectors this may not be the case. To access the n^{th} element of a hash table in a language without vectors would require the use of some other data type together with that data type's accessor functions. For example, a hash table built using lists would require $O(n)$

time to access the n^{th} bucket. When using vectors this can be reduced to $O(1)$.

When updating a hash table it is usual for one bucket to be updated at a time. The `vectorChange` primitive is ideal for the situation in which a new copy of the vector is created with one item updated. A bucket can be represented using a list of values. In the Rete pattern matcher the memory nodes of the Rete network use lists to store working memory elements that have matched nodes in the network. Gupta [Gupta86] observed that by replacing the list by a hash table the Rete algorithm became more efficient. Without the ability to implement efficient hash tables in a functional language the speed of an imperative rule-based system could not be matched.

There are undoubtedly many other situations where vectors would be essential for an algorithm. The quicker they appear in functional languages the better.

4.4. Graphs

It has been found that there is little experience in using functional languages to solve a large set of well known algorithms. There are many books, journals, and papers on algorithms and data structures which express solutions to problems in an imperative style rather than a declarative style (for example Horowitz and Sahni [Horowitz76]). As a consequence of this, and because most functional programming research is either theoretical or focussed on abstract machine implementation, there are drawbacks for functional programming as a whole. They are that:

a) there is little well known experience to draw on

b) there are few well known solutions to problems

c) there are large gaps in the whole solution space

Although there are numerous books on the subject of graphs and their implementation in imperative languages, few documented solutions for building and manipulating real graphs in functional languages were found during this research. This is a prime example of the stated drawbacks. Initially, it seemed impossible to create a real cyclic graph in a lazy, higher-order functional language. Once this problem was solved and a cyclic graph was created, it then seemed impossible to visit this cyclic

structure in a controlled manner because functions that visited the cyclic structure executed infinitely when a cycle was reached. The solution to both of these problems highlighted that:

i) a new technique had been discovered which had not been documented before

ii) this technique required an interpretation of the standard definition which was more abstract than that stated in Horowitz & Sahni.

The differences in the approach to programming graphs are presented in the following sections. From this description it is possible to see why some algorithms, such as graph building, are relatively difficult in functional languages.

Traditionally, functional programmers build and manipulate *lists* of objects and often write polymorphic functions which perform generic operations on lists that are independent of the type of objects stored within them. Lists are simple data structures and arbitrary lists can be readily described using algebraic data types. Graphs, however, are more complex structures and, in their most general form, may contain cycles. The static construction of a graph is relatively straightforward, but it is the construction of arbitrary cyclic graphs "on the fly" that is more problematic. Figure 4.1 is a directed, cyclic graph which has a cycle between nodes 'A' and 'C'.

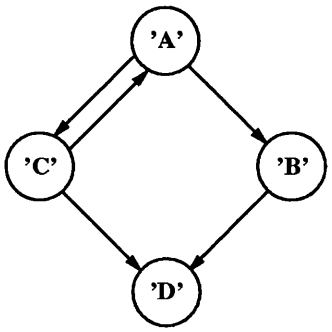


Figure 4.1: A small directed, cyclic graph

It is possible to represent a graph either by some form of adjacency matrix, by a function which will return a list of the successors of a given node, or by a form of

virtual heap using a list of nodes. However, none of these representations actually produce a real graph (that is, a truly cyclic data structure which has direct references between nodes along the arcs). The ability to build a truly cyclic data structure is important for reasons of efficiency. If a graph data structure has direct reference from node to node, the arcs can simply be followed to reach another node. By contrast, if a non-cyclic representation of the graph is used, then there is an interpretive overhead every time the program follows an arc. For many applications, truly cyclic structures are more concise, more expressive, and more elegant than representations of graphs.

Traditional functional programming solutions to the problem of creating graph data structures has involved the construction of a *representation* of a graph rather than building a truly cyclic structure. For example:

- a graph may be represented by a list of nodes and a separate list of arcs, thus circumventing the problem of physically connecting arcs to nodes.
- a graph may be represented as a function which maps from a given node to a list of the successors of that node
- a graph may be represented as a *virtual heap*, which uses a list as the heap, with list indexing being used to access individual nodes within the heap (successor nodes are identified by their index). The term *virtual heap* is used because the location of each cell in the list is used as a virtual address within the heap.

None of the above methods have arcs with direct access to nodes in the graph, all are subject to time and/or space overheads, and none are as elegant as a truly cyclic structure [21].

It is possible to create real cyclic graphs in a functional language by giving a name to each node and then referencing the node names explicitly from other nodes. The

[21] A vector representation has the potential to be the most efficient. If speed is the main requirement then it may be preferable to use a vector rather than a truly cyclic structure. In this case each graph node is an element of the vector and references to nodes in the graph are represented as indexes into the vector. Access to successor nodes can be achieved with O(1) lookup. At present, very few lazy functional languages currently provide vectors with guaranteed fast access.

cyclic graph of figure 4.1 can be represented as:

```
data Graph a = Node a [Graph a]


nodeA, nodeB, nodeC, nodeD :: Graph Char
nodeA = Node 'A' [nodeC, nodeB]
nodeB = Node 'B' [nodeD]
nodeC = Node 'C' [nodeA, nodeD]
nodeD = Node 'D' []


graph = nodeA
```

An alternate approach is to explicitly place the list indexing operator `!!` into a virtual heap representation in order to create a cyclic graph structure. The arcs of the graph are represented by direct references to nodes. The cyclic graph of figure 4.1 can now be represented as:

```
data Graph a =  Node a  [Graph a]
nodes :: [Graph [Char]]
nodes = [ Node 'A' [nodes !! 2, nodes !! 1],
          Node 'B' [nodes !! 3],
          Node 'C' [nodes !! 0, nodes !! 3],
          Node 'D' []
          ]


graph = nodes !! 0
```

Both of these representations build static graphs. If it is required to construct an *arbitrary* graphical structure to be specified at run-time by a textual description, then the construction of a truly cyclic structure is by no means as obvious. The first solution relies on the fact that one can name the nodes of the graph and refer directly to the names in the source code. Unfortunately, the static names that are bound to data objects at compile time are no longer available at run-time and it is certainly not possible to introduce new names to label new graph nodes as they are encountered. However, the

method of representing graphs using the virtual heap can be extended to be more general.

By representing a graph as an adjacency list and then converting this to a virtual heap, it is possible to create a truly cyclic data structure which has arcs with direct access to nodes of the graph and which can be built "on the fly". This approach makes heavy use of laziness (specifically lazy constructors) to achieve the desired goal. Both the adjacency list and virtual heap are intermediate tools for constructing the real graph.

The arcs of the resulting graph are direct references to the nodes, and the resulting data structure will be that shown in figure 4.2.

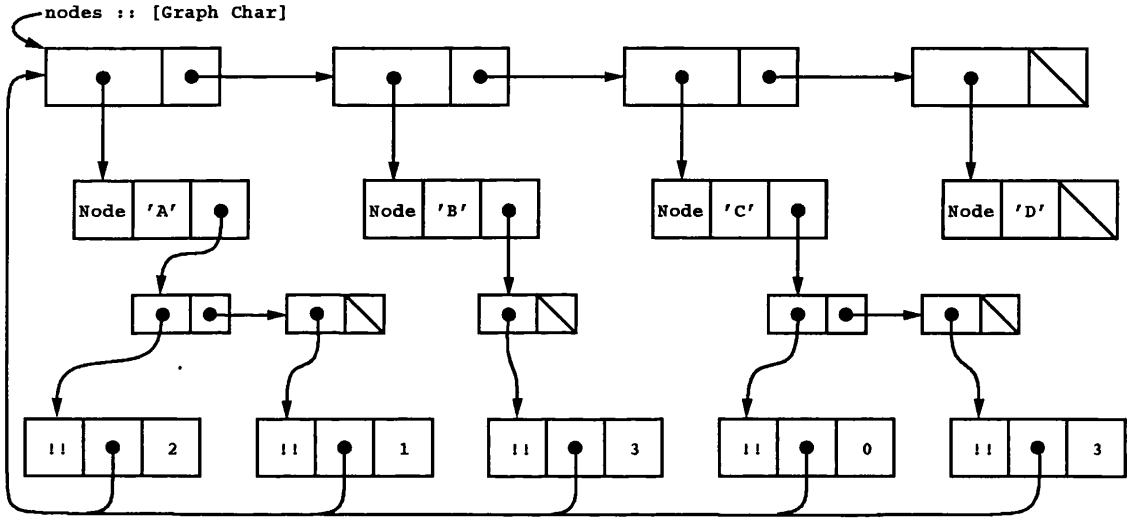


Figure 4.2: Representing a simple graph using embedded virtual addressing

Initially it appears that the space overhead of the top-level list `nodes` and the time overhead of the `!!` operator prevent a cyclic graph from being constructed. However, as the `!!` operator is embedded in the representation of the graph, when an arc is followed, the index expression is re-written by the functional run-time system to point directly at the relevant element of the list. Figure 4.3 shows the re-written data structure after the two arcs of the initial node have been visited. In Figure 4.3, there are fewer references to `nodes` than there are in Figure 4.2. When *all* of the references to `nodes` have been evaluated, then the list structure is no longer required and is garbage collected. At this point only the required data items are left, arranged as a graph with

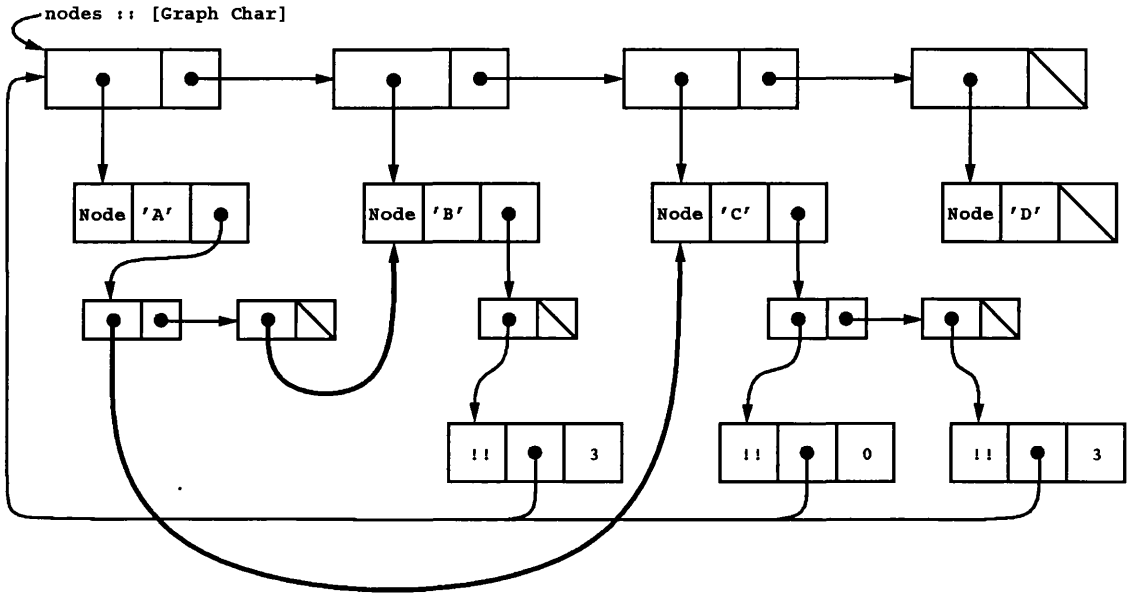


Figure 4.3: The list structure turning into a cyclic graph

direct access to nodes along the arcs. This is illustrated in Figure 4.4, in which the final data structure compares favourably with the original graph of Figure 4.1. The space overhead of the enclosing list has been eliminated, and future traversals of the graph are efficient in time because the pointers representing the arcs are followed directly.

When graphs are constructed in imperative programs, it is common to include with each node a bit that is set when the node is visited. This bit is used by graph traversal functions in order to ensure that nodes are not visited more than once; this avoids infinite loops due to cyclic pointers in the graph.

However, the functional paradigm prevents the in-place update of a visited bit. To overcome this problem each node of the graph is augmented with a unique tag. When visiting the graph, a list of tags is constructed to record the nodes that have already been visited. Prior to visiting a node, the list of tags is checked to see if it already contains the tag of the node to be visited; if it does, then the node is not revisited. Unfortunately, the list-of-tags technique introduces a searching overhead of $O(n^2)$ time-complexity where n is the number of nodes to be visited. This is expensive in comparison to the constant overhead of checking a single bit. However, more efficient structures than a

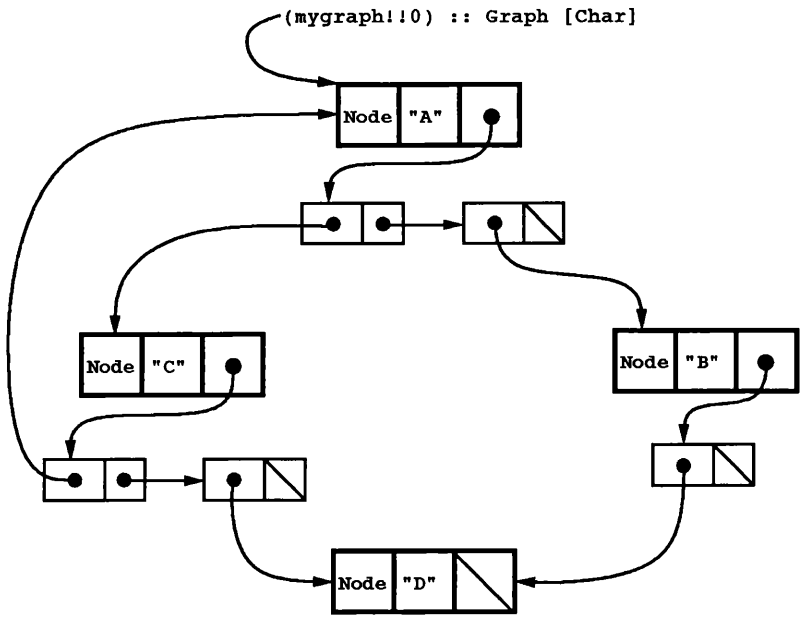


Figure 4.4: The adjacency list turned into a cyclic graph

list could be used if required.

The resulting graph structure generated by the functional program is immutable and this provides some advantages over mutable graphs generated using imperative techniques. The advantages are:

i) the same copy of the graph can be traversed multiple times without danger of unwelcome interaction

ii) there is no need to unmark the "visited" bits

iii) a single graph can be traversed by concurrent tasks

To facilitate use of these functional graphs, an abstract data type was designed and used in various case studies. The graphs can have both nodes and arcs labelled with separate types. This allows the implementation of doubly linked lists, rings, and solving the shortest path problem using a cyclic graph. A full description of the implementation details can be found in [Clayman93].

By taking a well known data structure and creating a functional form, and by taking its associated algorithms and reinterpreting the definitions of those algorithms in a more

declarative way, it is possible to implement the algorithms. The creation of the cyclic graph data structure was a non-trivial exercise, and a similar situation is likely to occur again for another data structure and its algorithms. It is issues such as this which hinder the general acceptibility of functional programming. Many programmers want to take well known data structures and algorithms and implement them directly in the language they know. Although this is not always possible with functional languages, the benefit of finding an implementation technique can bring many unforeseen advantages. The functional programming world needs a book on data structures, just as the imperative programming world has had for many years.

4.5. Interaction With The Operating System

This section discusses the two main mechanisms which enable a program to interact with the operating system.

4.5.1. Input and Output

As seen in chapter 3, input and output to functional programs is not as easy as in imperative programs. This is because the imperative model does input and output by side-effect, allowing both to occur anywhere in a program. It is not clear to imperative programmers how to do input and output if these side-effect procedures are removed.

The Miranda system and other functional interpreters provide a very simple model for doing input and output. The technique presented in Miranda allows a function of type *[Char] -> [Char]* to be applied to the input of the program rather than applying the function to some arbitrary string. The value returned by that function becomes the output of the program. This *stream* based model of I/O does work, but it is too simple to provide the flexibility required in a large application.

At the start of this research, the literature search and discussions with other researchers in the field of functional programming revealed little in the way of concrete experience for doing input and output in a large, functional application. There were some suggestions for how it may be done and of particular note is the work by

Thompson [Thompson86]. Thompson defines control structures, in which the operators allow the flow of control to be reflected in the syntactic form of a function. However, even in simple examples there are problems with the large number of types and support functions required. In a large application, such as a rule-based system, having such a large number of types and support functions would be problematic.

The approach chosen in this thesis has both input and output passed around the program as part of the state object. Both the input and output have to be plumbed into the program in order for there to be access to both streams anywhere in the program. As seen in chapter 3, having both input and output held in the state object can prevent input and output from behaving in a way the user would expect. This occurs when the state manipulation functions for I/O are not written with the operational behaviour of input and output in mind. The result is that all output is held up until the end of the program execution, then it all appears. This is perturbing, since the more usual behaviour is for output to appear gradually. This turns out to be a run-time issue rather than a semantic one, as the output is correct. By rewriting the state I/O functions, again seen in chapter 3, this odd run-time behaviour can be eliminated.

This work has highlighted the need for I/O control structures for use within functional applications. This is an important area of research for functional programming because input and output can be accessed anywhere in a program and because they have a run-time temporal behaviour as well as a semantic value. This temporal behaviour cannot be expressed as part of the program and, as discovered, can be difficult to determine, especially in a large application.

A newer model for manipulating input and output to functional programs is the *continuation* model. This model is used in the Haskell programming language. The Haskell I/O system allows for interactions with the environment provided for the program by the functional run-time system. In this environment, a program has a special type whereby the topmost function produces a list of `Request`. These *requests* for input and output are taken by the environment and, after execution, each one returns a `Response`. The Haskell I/O continuation system is layered on top of a stream based model of I/O, and both models can be mixed within the same program.

Unfortunately, the continuation model does not address the issue of building I/O control structures for use within a program although it allows the top level interactions to the run-time environment to be expressed with clarity and flexibility.

To address the need for I/O control structures, Dwelly suggests the use of *dialogue combinators* in his paper "Functions and Dynamic Interfaces" [Dwelly89]. He observes that their use is not well known, but then goes on to show how they can be used for systems with graphical interfaces. An area for further research would be to write a range of large applications using the dialogue combinators and to evaluate how they perform.

There is still much work to do in addressing input and output in functional programs. As in other areas, one can expect that the functional model will eventually be as expressive as the imperative model. Further work can be directed at building control structures for large applications. Some work was undertaken by Runciman to address the problem of input and output being held up. In [Runciman89], a special form of strictness analysis combined with some special transformation rules for a compiler are suggested. However, until these features are available in every functional language compiler, the run-time behaviour problems will persist.

4.5.2. Environment Interaction

The functional run-time system provides a mechanism which enables the functional program to interact with the operating system environment, such as doing input and output. In Haskell this mechanism is the `Request/ Response` system, where each `Request` to the Haskell run-time system has a corresponding `Response`. Full details of the `Request/ Response` system can be found in the Haskell report [Hudak88]. The advantage of this mechanism is that non-determinism is confined to the operating system and referential transparency is maintained within the Haskell program. The disadvantage is that all interactions to the operating system must come through the *main* function, thus limiting these interactions to one place. The program's interaction with the Haskell run-time system and the operating system environment can be viewed in Figure 4.5.

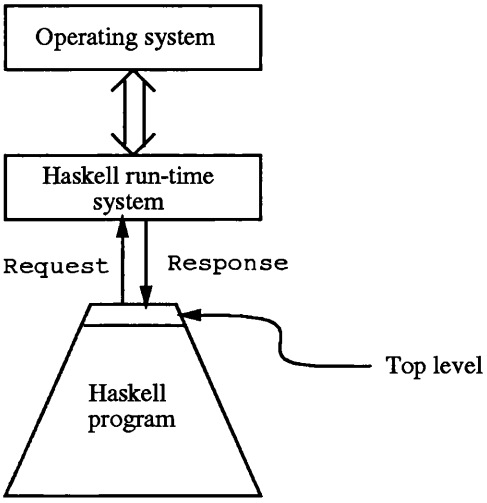


Figure 4.5 The interaction of a Haskell program and the operating system

Currently, the number of interactions with the operating system is very small. (The Lazy ML system [Augustsson92] provides a few more interactions than Haskell, but still not an extensive number). Those Haskell provides deal with file access and I/O stream access. The small number of operating system interactions limits the use of functional programming to either simple test programs or applications with a very limited form of input and output [22]. More complex applications cannot be written as there is no way to harness the operating system calls required.

As an example, consider the `Echo` request. It is a request to the functional language run-time system to turn on or to turn off echoing on the standard input stream to the program. The run-time system will make a call to the operating system to initiate this request, and finally a response is returned to the program. The problem in this case is that the facility to turn on or to turn off echoing is a single option in one operating system call. The other options of this operating system call are unavailable to the functional programmer, even though the programmer may deem them essential. The solution to this problem is for the functional language to provide an interface to each system call for the functional programmer to use at will [23].

---

[22] A former COBOL programmer, who is now a Haskell programmer, informs me that Haskell provides more operating system interactions than COBOL.

[23] This is a problem of having a limited manifesto, i.e. the design of lazy, functional

The lack of operating system interactions is a major obstacle for functional programming and hinders its general use. Until this issue has been addressed and resolved, functional programming is likely to remain in the realms of either a teaching language or a prototyping language. This is unfortunate because the compilers and run-time systems of functional languages are now of commercial quality, and large groups of programmers are not getting access to these functional programming environments.

4.6. Measurement

The techniques and tools available for observing and measuring the behaviour of functional programs are thoroughly inadequate. Given that one cannot measure the execution behaviour of a functional program effectively, it is impossible to make comparisons between programs, verify that algorithms display the expected behaviour, or observe degenerate behaviour.

At present there are no tools to help a programmer find a problem function and then to rewrite the function to make the program faster. A simple re-write of a function can make all the difference to a slow program. With well-defined measurement techniques, one can find these problem functions and also find where laziness has an effect on the program. With this knowledge, a better understanding of how functional programs actually work can be obtained, which in turn helps the programmer to write better programs. At present the measurement tools available to the functional programmer are:

a) counting the number of graph reductions performed

b) counting the number of cells used

These measurements are neither detailed enough nor do they express anything about the behaviour of the program. Furthermore, the information they provide is different in each functional run-time system.

---

languages, rather than the design of lazy, functional programming *environments*.

Three different simple test programs were executed on two different abstract machines, namely Miranda and Gofer, in order to gather figures for the number of graph reductions performed and cells used. The programs are:

1. `fac`, a program to generate one factorial:

```
fac n = 1 , n==0
      = n * fac (n-1) , otherwise


test n = fac n
```

2. `facs`, a program to generate a list of factorials:

```
facs = 1 : fact 1
          where
          fact n = n* facs !! (n-1) : fact (n+1)


test n = take n facs
```

3. `sfib`, a program that generates the fibonacci of a number together with the number of calls to sfib and the number of recursions from the original call to sfib:

```
empty:: (Int,Int)
empty = (0,0)


test n = sfib (n,empty)

sfib::(Int,(Int,Int)) -> (Int,(Int,Int))
```

```
sfib (0,(fc,pl)) = (1, (fc+1, pl+1))
sfib (1,(fc,pl)) = (1, (fc+1, pl+1))
sfib (n,(fc,pl)) = (fibm1+fibm2,
                        (1+fcm1+fcm2, max (plm1+1) (plm2+1) ))
                    where
                    (fibm1,(fcm1,plm1)) = sfib (n-1, (fc, pl))
                    (fibm2,(fcm2,plm2)) = sfib (n-2, (fc, pl))
```

Table 4.5 displays the number of graph reductions and table 4.6 displays the heap cells used for the expression `test n`.

program	fac		facs		sfib	
n	10	15	10	15	10	15
mira	124	184	223	363	11552	128590
gofer	45	63	116	127	2120	23672

Table 4.5: Graph reductions for expression `test n`

The table for the number of cells used is:

program	fac		facs		sfib	
n	10	15	10	15	10	15
mira	241	344	321	592	12774	141836
gofer	73	109	250	317	5846	65116

Table 4.6: Heap cells used for expression `test n`

For the programs `fac`, `facs`, and `sfib`, the values returned by the functions are the same on both the Miranda and the Gofer run-time system, but the number of graph

reductions and the number of cells used for each machine is different. Due to the different kinds of reductions used within each abstract machine, the grain size of a reduction is different in all of the abstract machines. These figures, which are easy for the run-time system implementor to produce, are of little benefit to the programmer.

Given that these measurement techniques and tools are inadequate, it is not possible to make any decisions as to the quality of a program. This is one of the reasons why the development of the functional OPS5 was limited. Without tools to compare its performance with exisiting versions of OPS5, it is impossible to state any concrete facts regarding its behaviour; for example, one cannot determine if the functional version of the matcher is faster than the imperative version. To overcome this problem, some measurement tools and techniques were designed and implemented for this PhD.

Examples of strange behaviour in functional programs are:

i) Wadler points out that some functions which are expected to be $O(n^2)$ may be less than this due to lazy evaluation.

ii) Simon Peyton-Jones describes functions in his SASL paper which seem to be cyclic functions. He observes that when these functions are written incorrectly they do not become cyclic and their space usage increases dramatically [Peyton-Jones85]. Hughes makes a similar point in his paper "Why Functional Programming Matters" [Hughes89].

With a measurement tool the strange behaviour of both (i) and (ii) can be verified.

The measurement tools and techniques will be fully explained in Chapter 5, where the design and implementation of a profiling tool for lazy, higher-order functional languages is presented.

4.7. Debugging

Another problem facing functional programmers is the lack of debugging tools. Debugging functional programs is much more difficult than imperative programs because referential transparency has to be maintained. Furthermore, as there are no side-effects and no ordering of statements, it is impossible to insert extraneous `print`

statements into a functional program. All output must be produced by the main function of the program. For any debugging output to appear, it must be returned as an extra value from the function that needs to be debugged in addition to *all* the functions up to the main function. The design and programming effort required to make these changes is non-trivial, especially in a large application. Most of this effort is wasted because the extra debugging code is thrown away when the debugging is finished. If monads were used in *every* function, then only the monadic type and the definitions for `unit` and `bind` need to be changed. However, the resulting code, particularly in a large application, would be inelegant.

During the development of the functional OPS5, no debugging tools for functional languages were discovered. In an attempt to address this issue, a simple debugging utility was designed by myself and Parrott as an extension to the FLIC language [Parrott90]. This extension is a function which prints some debug output by side-effect to a special output stream which is invisible to the program yet behaves in a referentially transparent way within the program. The function, called `debug`, takes as arguments a printing function and a value. The printing function is applied to the value and the returned string is sent to the special output stream. The value returned by `debug` itself is the value given; therefore `debug` behaves like the identity function within a program. A definition for debug could be:

```
debug :: (a -> [Char]) -> a -> a
debug show_fn a = a
```

The expression `show_fn a` was to be automatically inititiated by the run-time system.

This technique was discovered to have serious drawbacks. First, the value passed to the `debug` function may not have been evaluated at the time `debug` was called. To produce a result on the special output stream would require fully evaluating this value in order to apply the printing function to it. However, the `debug` function was meant to be invisible to the rest of the program and to behave like the identity function. If `debug` were to evaluate arbitary expressions, then the behaviour of the whole program might change. It was found that the `debug` function was strict in both arguments and, therefore, did not behave as desired. Second, the nature of lazy evaluation means that

the output produced by a program using `debug` would not necessarily be in the order that the programmer expects to see it. Given that the debug function was meant to be an aid to the programmer, this behaviour is not beneficial. The special debugging function was rejected as a debugging tool.

Recent attempts to define what constitutes debugging of a functional program have been addressed in "An Algorithmic and Semantic Approach to Debugging" by Hall et. al [Hall90]. The design and implementation of tools for doing debugging of functional programs using the algorithmic approach to debugging has been undertaken by Nilsson and Fritzson [Nilsson92]. Although algorithmic debugging is only one approach to solving the problem of debugging functional programs, the fact that someone is now addressing this issue is promising for all functional programmers.

Chapter 5

5. Profiling

One of the original aims of this thesis was to compare the performance of an existing rule-based system with a functional version, but this is impractical due to the present lack of measurement tools. Chapters 3 and 4 highlight that one of the major problems in developing applications in lazy, functional languages is the lack of tools which aid the programmer in debugging and analysing the run-time behaviour of the application. This chapter addresses the issue of analysing the run-time behaviour by describing the design and implementation of a profiler for lazy, functional languages.

The major issue when profiling programs is to enable the programmer to use the resulting information to determine whether parts of the program consume a disproportionate amount of resources. For many real-world applications it is not just desirable but essential for a programmer to be able to monitor and subsequently alter the time and space behaviour of the program. Without profiling information, it may be impossible to rectify a program which exhibits degenerate behaviour.

Lazy, higher-order functional languages provide a programming framework which is far removed from the details of instructing computer hardware. This high-level framework enables a programmer to express problem solutions in a way that closely resembles the problem specifications and which may exploit new software-engineering techniques [Hughes89]. Unfortunately, this high level of abstraction means that the executable form of a functional program is unrepresentative of the original source code. This poses two problems:

1. The source code is an unreliable indicator of a program's eventual run-time behaviour. It is therefore difficult for a programmer to use static analysis

techniques to reason about the time and space complexity of a functional program. This directly contrasts with imperative languages, in which the source code is a key factor in estimating a program's behaviour prior to execution.

2. It may be difficult for a programmer to interpret information on the run-time behaviour in order to reason about sections of the program which may need to be modified.

Most profilers address the second of these two problems.

In order to address the issues that have been highlighted in this thesis concerning the lack of measurement tools for functional languages, a profiler is proposed that is designed primarily for use by application programmers rather than functional language implementors. This profiler provides information that is related to the way the program is written rather than to how it is evaluated; this enables programmers to relate results back to the source program easily. The results directly reflect the *lexical* scoping of the source program, thus overcoming problems caused by compile-time program transformation, lazy evaluation, and higher order functions. I call this technique *lexical profiling*.

Using the lexical profiling technique, a *lexical profiler* was constructed, by Parrott and myself, to monitor programs as they run and to build detailed trace information for post-mortem analysis and debugging [Clayman91], [Clayman92]. This lexical profiler uses a mechanism which accurately profiles the call-count, time, and heap space used by lazy, higher-order functional programs. The results are similar in nature to, but more accurate than, the UNIX [24] imperative language profiler **gprof** [Graham82].

This chapter presents four different methods of profiling functional programs, followed by a discussion on two styles of profiling – inheritance and statistical. Then five existing profilers are reviewed in relation to how they each affected the design decisions for the lexical profiler. Various design issues of lexical profiling are presented followed by a discussion on the actual implementation techniques that were used to

[24] UNIX is a trademark of Bell Laboratories.

construct the lexical profiler. Finally, there is an analysis of the working lexical profiler with profiling data obtained from worked examples.

5.1. Different Kinds Of Profiling

There are four different kinds of profiling that can be undertaken in functional programming environments:

i) **Program profiling.** Measurements relate to the program's behaviour and are reported with respect to functions in the source code. This is *lexical profiling*.

ii) **Expression/Closure profiling.** This is similar to the earlier cost experiment at UCL [Parrott90] and the old Glasgow Cost Centres [Sansom92]. In expression/closure profiling, measurements are based on how the program executes and the results are reported when an expression is evaluated. This is dynamic profiling.

iii) **Abstract machine profiling.** This measures how effective an abstract machine is by examining the overheads of function calls, function returns, heap management, garbage collection, etc [Hammond91a].

iv) **Task profiling.** This is particularly relevant in parallel environments where programs are divided into tasks which execute on separate machines. The number and size of the tasks are reported [Parrott92].

In [Runciman90], Runciman and Wakeling provide a good overview of the problems associated with profiling functional programs. They make several suggestions regarding the sorts of information that would be useful to a programmer and provide a more detailed analysis of how such information might be collected. Later in the chapter there is a summary of the issues listed in [Runciman90] and how the lexical profiler addresses these issues.

The research in this thesis has indicated that it is not clear to everyone in the field of functional programming that these different kinds of profiling can be usefully measured separately. Many people in functional programming who are doing measurement are

implementors interested in low level details. They wish to measure *when* work is done and at what point an expression is evaluated and to observe the effect that lazy evaluation has had on a program. This gives very different results from lexical profiling, which is dissociated from *when* work happens. Lexical profiling measures *whether* work happens and how much happens, with results being presented with respect to the source code. The difference is mainly in the way in which lazy evaluation has an observable effect on the program.

The following examples show the difference in the results between dynamic and lexical profiling. Consider the following programs:

Program 1

```
f   = (g x) / 18
      where x = expression
g x = (h x) * 10
h x = x + 32
```

Program 2

```
f   = (g 10) / 18
g y = (h x) * y
      where x = expression
h x = x + 32
```

Program 3

```
f   = (g x) / 18
      where x = expression
g x = x * (h 10)
h y = y + 32
```

Although these three programs are similar, they differ where the expression x is declared and evaluated. Table 5.1 shows the number of primitive operations counted for the functions in each program using both lexical profiling and dynamic profiling. The term p_x equates to the number of primitive operations required to evaluate x. The results of the lexical profiler always show the cost of x being associated with the function in which x is lexically contained. The results of the dynamic profiler highlight the presence and effect of laziness, and the cost of x is associated with the function that required the value of x.

Program	Function in which x is		Number of primitive operations					
			lexical profile			dynamic profile		
	declared	reduced	f	g	h	f	g	h
1	f	h	$1+p_x$	1	1	1	1	$1+p_x$
2	g	h	1	$1+p_x$	1	1	1	$1+p_x$
3	f	g	$1+p_x$	1	1	1	$1+p_x$	1

p_x is the number of primitive operations

Table 5.1: How the cost of primitives is attributed by lexical and dynamic profiling

Although most profilers do not count primitive operations as a statistic, these examples highlight the differences in the two styles. Moreover, they indicate that in order to fully appreciate how a program is evaluating, *both* profilers can be used together to provide a comprehensive view.

Consider another example in which dynamic profiling may give differing results but lexical profiling will give a consistent result. In the program:

```
f   = (g x) + (h x)
      where x = expression
g x = h x * 10
h x = x + 32
```

the evaluation order of the primitive + is important. If the evaluation order of + is left to right, then a dynamic profiler will credit `g` with the evaluation of `x`, but if the evaluation order of + is right to left, then `h` will be credited with the evaluation of `x`. In a parallel system where the load balance and evaluation order are non-deterministic, a dynamic profiler may return different results on different occasions. Lexical profilers do not suffer from either of these problems as results are associated with lexical scope. This provides a static relationship between the source code and the run-time results.

A further advantage of lexical profiling is that because the results are dependent on the source code, it is possible to change the underlying evaluation mechanism and ALWAYS have meaningful results. As there is not a strong relationship between the source code of a functional program and its evaluation mechanism, one could, for example, replace a graph reducer with a Term Rewriting System [Glauert90]. The results of the lexical profiler would still be associated with the source code. A dynamic profiler for a Term Rewriting System may give very different results and may not fit the model of evaluation that the programmer has. Therefore, with lexical profiling the programmer gets meaningful profiling data for his program regardless of the evaluation mechanism, but data from dynamic profiling is always dependent on the evaluation mechanism.

5.2. Styles of Profiling

This section describes two styles of profiling, *statistical* and *inheritance*, which provide complementary views of the execution of the program. A technique for profiling lazy, higher-order functional programs is presented which uses both of these profiling styles. This technique is based on the lexical structure of the source code and therefore produces information that is meaningful to a programmer.

To be fully (100%) accurate a profiler needs to reconstruct the entire call-path for all function calls; however, in practice this is too costly. Therefore, the run-time log is restricted to information concerning the calls made by a function to its immediate children [Graham82]. Traditionally this causes problems because profilers are forced to estimate the execution time of more remote generations. Consider the call-graph segment shown in figure 5.1. Here the function i is called only from h, but h is called from both f and g.

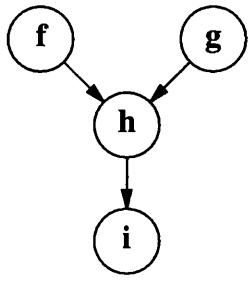


Figure 5.1: A typical call-graph segment

The following code outline represents this scenario:

```
f a = h a + 1
g b = h b - 1
h x = i x + i x
i x = x + 1
```

The function i is only called by the function h. The total time spent in or below f is uncertain because it includes the timings for i (which is called from h), of which unknown amounts are due to calls originating from g. The profiler will keep a log of calls f to h, and g to h, and h to i, but not f directly through to i, or g through to i.

One solution to this problem is to divide i's time according to the ratio of calls from f to h and from g to h; this is *statistical* profiling (e.g., if there are 6 calls from f to h and 4 calls from g to h, then f will get 60% of the time in i and g will get 40% of the time in i). However, statistical profiling is blatantly inaccurate as there need not be a linear correlation between the number of calls and the execution time; in fact, calls

originating at either f or g may not invoke i at all; this depends on the value of the parameters passed to h. Nevertheless, the information about the calls to immediate children is accurate.

An alternative solution is to allow the code for i to be subsumed by the code for h (i.e. as far as profiling is concerned i is then an integral part of h). This is *inheritance* profiling. Under inheritance profiling, the sub-function i is just an extension of its parent, and the total amount of time spent in or below h due to either f or g is determined absolutely. Although accurate over many generations, this style does not report a separate timing for i and it appears that profiled sub-functions of i are called directly from h — this may be confusing for the programmer as the function i seems to have no data relating to it. With *inheritance* profiling the code outline would be profiled as though it had been written as:

```
f a = h a + 1
g b = h b - 1
h x = i x + i x
      where
      i x = x + 1
        .
```

To provide comprehensive profiling, this PhD advocates the use of both statistical and inheritance profiling modes within the same profiler.

5.3. Existing Profilers

This Section describes existing profilers for both imperative and functional languages and considers how they motivated and affected the design decisions for the lexical profiler.

5.3.1. gprof - an existing imperative profiler

The UNIX profiling tool **gprof** [Graham82] produces a profile of a program based on the call graph of the programs execution. Results are presented with an entry for each function, together with its call graph parents and call graph children. The data for

child functions is propagated up the call graph to incorporate a measure of the expense of subchildren. The **gprof** mechanism is a great improvement over the simpler flat style of profiling which just reports how many times a function is called, the amount of time spent in that function, and the percentage of total running time spent in that function. As a result, **gprof** has been used successfully with imperative programs for many years.

The implementation of **gprof** is based on the assumption that code is statically placed in consecutive memory locations at load time. The execution time of each function is not measured exactly, but approximated by monitoring the location of the program counter every 1/60th of a second. A histogram of program counter values is constructed and the amount of time spent in each function is estimated by post-processing the histogram in conjunction with a map of code locations. One problem with **gprof** is that it does not monitor space utilisation and so cannot provide full information for programs which make extensive use of dynamic memory allocation (however the **mprof** profiler [Zorn88] does provide this facility). In addition, **gprof** does not provide useful information for mutually recursive functions because it collapses each strongly-connected component in the syntax graph to a single point.

Despite the faults and inaccuracies mentioned above, **gprof** has proved to be a useful tool for imperative programmers. This provides a motivation to develop similar profiling tools for functional languages.

5.3.2. The New Jersey SML Profiler

Most current implementations of functional programming languages provide only rudimentary profiling statistics, with information restricted to (for example) the number of garbage collections performed, the total number of reductions performed, and the total number of memory cells used. The New Jersey version of Standard ML is remarkable for the fact that it is supplied with a profiler which gives more extensive information related to function names.

The New Jersey SML profiler described in [Appel88] uses an inheritance profiling style but does not try to address the inaccuracies that are introduced (other than directing the programmer to experiment by using multiple profiles, choosing different

groups of functions each time in an attempt to get a more accurate picture of what really happened). It is limited to strict evaluation and neither profiles heap space usage nor provides a statistical profiling option.

The SML profiler is also inaccurate when profiling higher-order functions because it attributes execution times of higher-order arguments to special identifiers instead of to the real functions. The example in [Appel88] argues that the ambiguous results are of little consequence in short programs where a higher-order function is called just once and suggests that the programmer should be able to guess to which real function the special name refers. However, guessing is not so simple for large programs where higher-order functions, such as *map*, are called repeatedly with different higher-order arguments each time. The SML profiler coalesces all applications of a single higher-order parameter into a single timing, thus losing vital information. If timings are kept separate by inventing a new name for each call, the programmer will be swamped with too much information to decipher it sensibly.

5.3.3. UCL inline cost primitive

An early profiling technique investigated at UCL for measuring the cost of evaluating an expression was the use of inline cost functions [Parrott90]. This technique uses a cost function which has the equivalent semantic behaviour to the identity function. The cost of the evaluation is written to a special output stream which cannot be accessed by the program. For example:

```
g x = cost (f x) + 1
```

would report the cost of evaluating `f x`. There is no data for space usage or function call-counts. Due to problems with lazy evaluation and unevaluated arguments, the use of inline cost functions relies on evaluation transformers [Burn87] to enable the function to measure the cost of evaluating its argument by forcing the correct amount of evaluation to occur inside the cost function (i.e. the cost function evaluates its argument as far as the surrounding context demands and returns the result).

However, the use of inline cost functions has drawbacks because the information provided by a cost function is dependent upon its context at run-time. It is impossible to interpret the results without thoroughly understanding the effects of laziness on the evaluation of a program. When evaluation transformers are used, the results presented are for a program which is slightly different to the one the programmer wrote. Hence, the results are not very useful. In a *parallel* implementation, the order in which expressions are evaluated cannot be determined and the timings returned by `cost` will change from one program run to another. A fundamental problem with this profiling technique is that it takes a microscopic view of the program, whereas a macroscopic view would report its results at a level of abstraction understood by the functional programmer.

5.3.4. Glasgow Cost Centres

In [Sansom92], a profiler with a primitive similar to the UCL cost primitive is presented. Sansom and Peyton Jones introduce the named cost centre, which associates the cost of evaluating an expression with a given name. This concept is the same as the UCL cost primitive but has been extended to allow nested cost centres. The problems of lazy evaluation and unevaluated arguments also arise. To overcome some of these problems, Sansom and Peyton Jones suggest that code should be rewritten in certain instances in order to calculate the cost correctly. This may be a reasonable task for a short 10 line test program but is unsuitable for a 4000 line application.

As with the early UCL cost primitive, this solution to profiling requires the programmer to understand how a run-time system evaluates a functional program so that the programmer can then place the cost-centre primitives in the correct place.

5.3.5. Runciman and Wakeling Heap Profiler

In [Runciman92], Runciman and Wakeling describe a profiler that monitors heap usage of lazy, functional programs but does not measure call-counts or the time spent in functions. Their system relies on the user understanding how a run-time system works. This view of execution may be normal to a system implementor but is often alien to

applications programmers. In a worked example, data graphs are presented which show the producers of heap cells and the data types that are associated with those cells. The graphs are then analysed to determine the behaviour of the program with an aim to reduce heap usage. Although a significant reduction in heap usage was achieved, the authors were required to display a wider knowledge of the underlying implementation than would be expected of a typical applications programmer.

On two out of four occasions, Runciman and Wakeling observed problems with their compiler and run-time system; they then modified their compiler and run-time system in order to bring about the performance gain. For the ordinary applications programmer, with neither access to the source code nor knowledge of the internal workings of these systems, the changes made by Runciman and Wakeling would be infeasible.

The Runciman and Wakeling profiler measures heap space by visiting the whole graph at pre-determined intervals. For large heaps (as in their example), the pauses caused by these visits will be long. Thus, for practical reasons, an upper bound is imposed on the sample frequency but this can cause the presented data to be inaccurate.

5.4. Lexical Profiling

In this section the main aspects associated with the design decisions for lexical profiling are discussed.

5.4.1. Design Objectives

When an applications programmer uses a functional language to implement a system and then uses the lexical profiler to help him analyse the run-time behaviour, it is expected that he knows certain attributes of the languages he is using, the compiler and how it works, and the underlying abstract machine.

Language

On the language side, the programmer needs to know the basics of functional programming. However, in order to understand the results of the profiler and to use those results to improve a program, the programmer should know about substitutive equality/ referential transparency so that he can transform or re-write his code not only correctly but also more effectively.

Compiler

The programmer needs to know the following about the compiler:

- the flags that control the main/most useful options and what they do
- the compiler optimizations which may affect the running of the program.

  It would be useful if the compiler writers and abstract machine writers would document the optimizations, transformations, and features in their systems so that programmers realise their existence and can take account of them if necessary. For example, list comprehensions are often converted into other functions, e.g:

  ```
  fn g e = [g x | x <- [1..e]]
  ```

  gets converted silently into calls to built in functions.

The programmer does not need know:

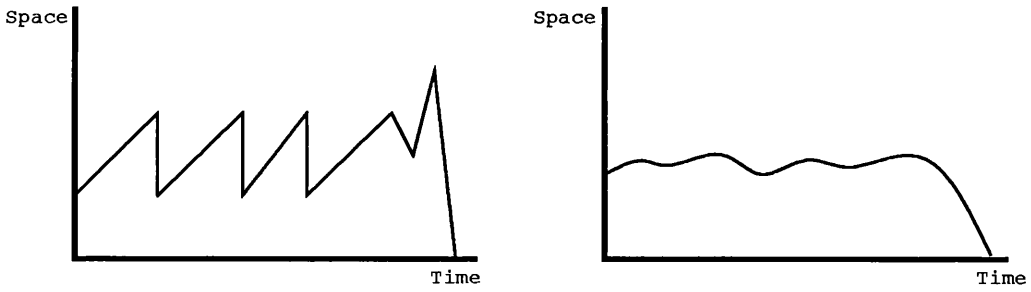
- if the compiler does dead code elimination. Dead code can be removed without affecting the program because it is never referenced and therefore never executed.

Abstract Machine Run-Time System

With regard to the abstract machine run-time system, the programmer needs to know:

- that functions and data are treated in the same way and that they both require space

- that function applications require space
- whether the system evaluates lazily or strictly. This means the programmer needs to know that computations can be delayed by the lazy evaluation mechanism, but he does not need to know how this happens. He should also know that with laziness he can save space and evaluation time by sharing expressions (and that using pipelining is an effective way to write functions) [Clayman93a].
- which sort of garbage collection technique is being used, as this may affect the results from the profiler. Results from the two main kinds of garbage collectors may look like those in figure 5.2. The mark and sweep and two-space copying collectors only run at certain intervals, so garbage builds up and is collected in a big mark and sweep for compacting space or the copying phase. Incremental garbage collectors collect garbage immediately.



Mark and sweep
Two-Space Copying

Incremental

Figure 5.2: Space usage with two different garbage collectors

The programmer does not need to know the following about the abstract machine run-time system:

- which abstract machine is being used, although some programs may behave well on one abstract machine and badly on another

- how a function is applied to an argument
- how laziness works and how the abstract machines' mechanisms provide laziness
- about stack space in an abstract machine, and that different machines use the stack differently. Some, for example the ABC machine [Koopman90], use more than one stack.

5.4.2. Program Size

This section considers the size of program to which the lexical profiler is best suited. Very short programs do not utilise the lexical profiler to its highest ability. The lexical profiler is more useful when monitoring programs that use more than a minimal amount of resources. One reason for this is that data is collected on every function call, every function return, and when cells are allocated and de-allocated. However, the system clock does not have a fine enough resolution for complete accuracy. On the Sun workstation used for the development of the profiler, the clock resolution is 20 ms (this resolution is built into many other machines). Therefore, all times attributed are in chunks of 20 ms. With very small programs, the whole program run may occur within 20 ms; this is neither the fault of the profiling technique nor the implementation of the lexical profiler, but is a limit of the hardware. If access to a real-time clock were available *all* timings would be 100% accurate.

For example, consider a function that converts a string to an integer:

```
string_to_int1 :: String -> Int
string_to_int1 s = string_to_int' s 0
    where
    string_to_int' [] v = v
    string_to_int' (h:t) v = string_to_int' t (10*v + (ord h - zero))
    zero = ord '0'
```

If one only needs to compare the function's performance with another string to integer conversion function, the lexical profiler would be an overkill solution - a sledge hammer

to crack a nut! When using this function to convert a string to an integer, the execution time would be less than the clock resolution of the machine. It is in this situation that the number of cells and the number of reductions is useful. Consider another string to integer function:

```
string_to_int2 :: String -> Int
string_to_int2 s = sum [ x*y | (x,y) <- scale_factors ]
    where
    digits = map ((\v -> v - ord '0').ord) s
    scale_factors = zip (reverse digits) (iterate (*10) 1)
```

In order to determine which string to integer function is the most efficient, one can compare their run-time behaviours for a given input. Table 5.2 gives the number of cells used and the number of reductions performed for the given input (this experiment was done using the Haskell interpreter, Gofer).

Input	`string_to_int1`		`string_to_int2`	
	cells	**reductions**	**cells**	**reductions**
""	10	3	19	7
"1"	19	9	47	23
"12"	29	14	77	41
"123"	39	19	107	59
"1234"	49	24	137	77
"12345"	59	29	167	95
"123456"	69	34	197	113
...	...	...	...	...

Table 5.2: The number of cells used and reductions performed for 2 string to int functions

The data in table 5.2 shows that, although both functions display linear behaviour, one function is more efficient than the other. One can see for `string_to_int1` that:

no of cells = 10 × length input + 9

no of reds = 5 × length input + 4

and for `string_to_int2` that:

no of cells = 30 × length input + 17

no of reds = 18 × length input + 5

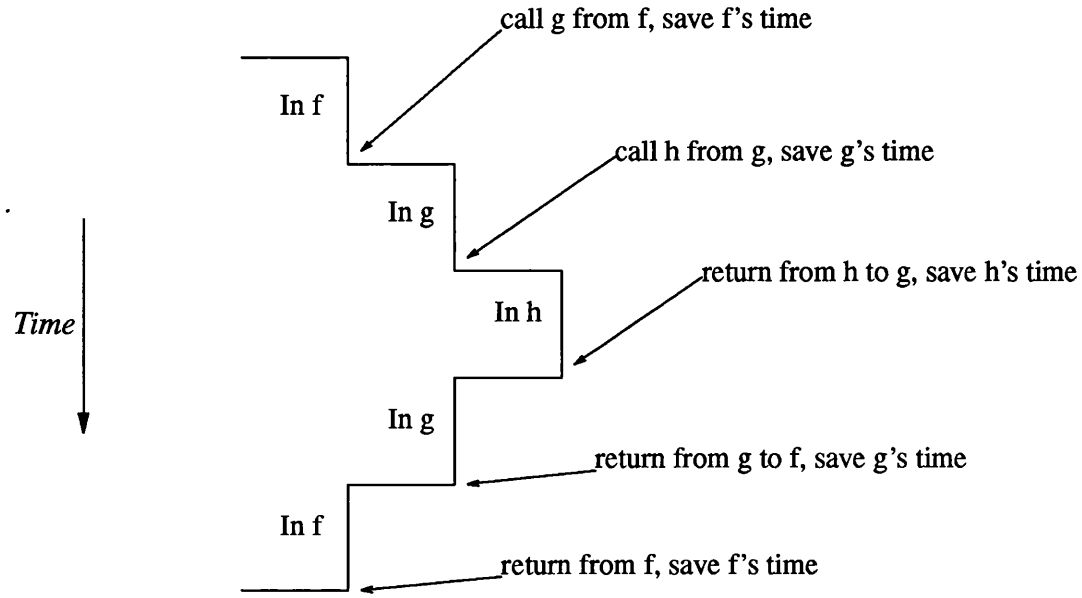
A programmer would choose `string_to_int1` to convert strings to integers in an application because it is the more efficient function.

5.4.3. Requirements for Lexical Profiling

Lexical profiling requires the compiler to record the lexical scope of functions so that the run-time system can monitor the functions and attribute measurements correctly in the presence of higher-order functions and lazy evaluation. The compiler needs to access the source program early in compilation and is responsible for maintaining the lexical affinities throughout all subsequent program transformations.

The run-time system is responsible for measuring the time spent in a function, the number of calls to a function, and the amount of space used by a function. The space used by a function equates to the number of cells allocated during the evaluation of that function. The number of calls to a function denotes the number of times that function is applied to some arguments. The time spent in a function is the accumulation of small amounts of time in different parts of that function. This is illustrated in figure 5.3 where times are incremented at relevant points during evaluation, i.e. when a call is made to another function and when a return is made from a function.

In order to retain time data for a call graph, it is necessary to remember when one function was called from another and how long this took. To enable this, a *from table* is built for every profiled function. It records the current profiled function, the function *from* which it was called, and the number of calls and amount of time associated with the function from which it was called. An example *from table* of calls to *k* will be:



Vertical lines represent time spent in a function.

Horizontal lines represent function calls and returns.

Figure 5.3: Timings saved when calling or returning from functions

In function	From	No. of calls	Accumulated time
	f	...	...
k	g	...	...
	h	...	...

Each *from table* is a representation of calls from multiple parents to a single child. To generate the full call graph, the data for a single parent to multiple children is needed. This data can be generated by inverting every *from table*. All the data relating to "from g to fn" can be generated by collecting all the "in fn from g" data in every *from table*, as in:

From table data	Inverted table
f *from* g	g *to* f
h *from* g	g *to* h
k *from* g	g *to* k

When the full call graph is generated, the profiling results are evaluated and returned to the user.

Both *inheritance* and *statistical* profiling styles require that output is restricted to functions whose profile was requested by the programmer. For each profiled function the following is reported:

- the time consumed by the function
- the space consumed by the function
- the number of times the function was called (and from whom)
- the number of calls the function made (and to whom)

The programmer can then compare and contrast the output of both styles to obtain a clearer overall picture.

5.4.4. Lexical Profiling

The innovation of the lexical profiling technique for profiling lazy, higher-order functional programs is the *combination* of:

1. profiling function definitions rather than expressions.
2. attributing the costs of all, and only, those expressions *textually* contained within each profiled function to that function.
3. taking a macroscopic view of the program and collecting statistics over a whole program run.

This thesis defines *lexical profiling* as a technique with the above three properties; it differs from dynamic profiling, which associates measurements with the run-time

representation of the program (as in [Clayman91] and [Sansom92]). The advantage of lexical profiling is that it provides information that is related to the way the program is written rather than to the way it is evaluated.

Consider the following example:

```
f  = g (sum [1..1000])
f' = g 500500
```

where `g` is non-strict in its argument. Lexically, the sub-expression `sum[1..1000]` appears within the body of `f`. Therefore, it is reasonable for the programmer to expect the cost of executing this sub-expression to be attributed to `f`. Many implementors disagree with this approach because the evaluation of the sub-expression actually occurs when `g` is executed. However, lexical profiling is designed to be used by application programmers who may know nothing about the run-time system. When using the lexical profiling style for the second expression, the cost of applying `g` to the atomic value `500500` is attributed to `f'` and is lower than the cost previously attributed to `f`. The programmers attention is therefore drawn immediately to the differences in the definitions of `f` and `f'`.

The lexical profiler collects statistics for user defined functions for either *all* top-level functions or just those which the programmer requests [25]. The restriction to top-level functions greatly simplifies the profiler at minimal cost to the programmer. The profiler should measure the time and space used at run-time by profiled functions and report the number of calls made to(from) profiled functions and from(to) whom. For lexical profiling, the profiler must recognise when lazy arguments are being evaluated and switch context so that the time and space required for the evaluation are attributed to the function whose definition lexically contains the associated expression. The context switch does not constitute a full function call so the number of calls made must not be incremented.

---

[25] This is achieved by compiler options rather than inline program annotations.

5.5. Implementation Techniques for Lexical Profiling

The lexical profiling technique is amenable to implementation in both compiled and interpreted abstract machines. This section demonstrates the general principles of the profiler's design and implementation and uses a sequential, interpreted model of graph-reduction in order to simplify the presentation [26]. The modifications in both the compilation phase and the execution phase are examined and details presented of call-count profiling, time profiling, and space profiling [27]. Then follows a discussion on how the lexical profiling technique applies to compiled abstract machines.

5.5.1. Compilation phase

The first pass of the compiler builds a graphical representation of the program, called CGF [Parrott91], marks the root of every function to be profiled with a one bit root-tag which is used for the call-count data, and also assigns a unique profiling colour to each function. The colour is used when function time and heap space usage are recorded. A second pass propagates the profiling colours from the root node to descendant nodes which are not themselves marked as roots. Two passes are required because all root tags need to be in place before propagation occurs in order that colours are propagated to the correct graph nodes.

By using the CGF notation to show the compiler's representation of a short program segment, the placement of root tags and profiling colours in the two passes of the profiling phase of the compiler can be seen. Consider the program:

---

[26] Full implementation details for both compiled and interpreted abstract machines can be found in [Clayman91].

[27] The design of the profiling technique was done by myself assisted by David Parrott, and the implementation of the profiling technique was done David Parrott assisted by myself. For full details of the implementation changes in the UCL experimental reducer, the reader is directed to Parrott's PhD thesis [Parrott93].

```
main  = f 10
f x   = h 1 (g x [1..1000])
g a b = a : reverse b
```

This program can be represented graphically and seen in figure 5.4, which shows the CGF form of the program. On the right hand side of the cells is the expression which each cell represents.

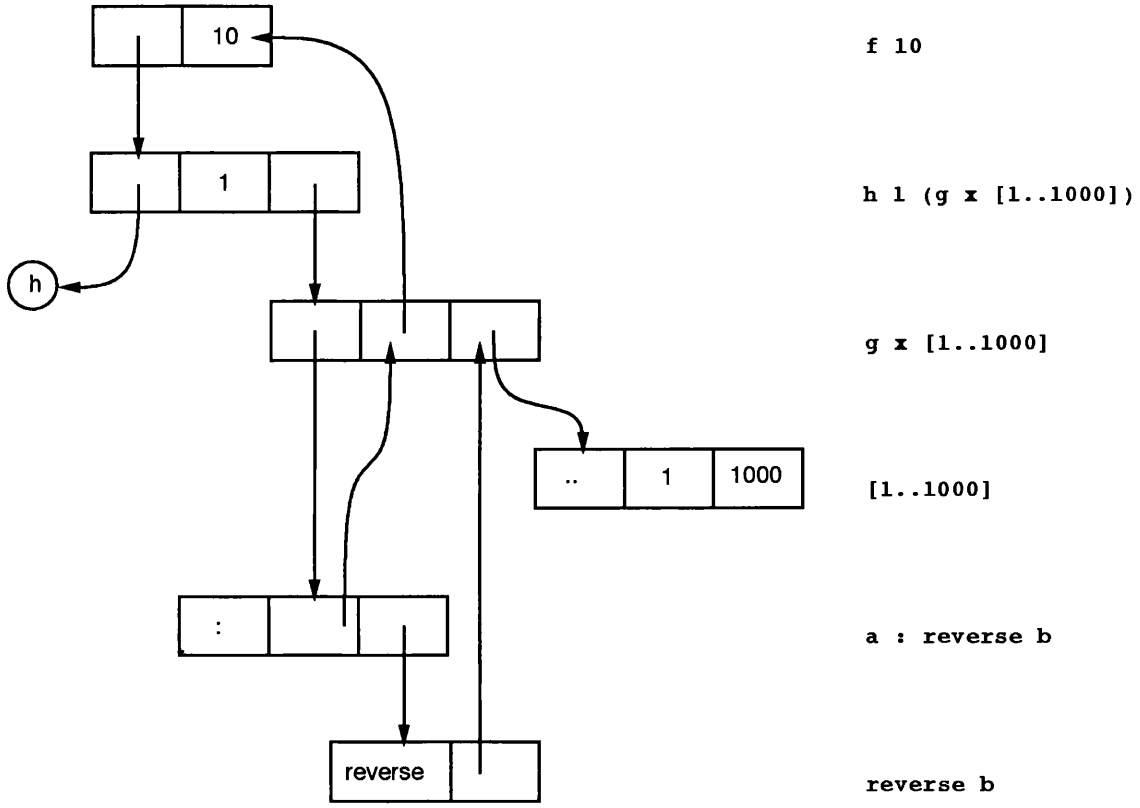


Figure 5.4: The CGF for program

The first pass of the profiling phase places the root markers on the relevant cells. Figure 5.5 shows which cells have root tags associated with them. Figure 5.6 shows all the cells after the profiling colour has been propagated to them. During the propagation pass of the profiling phase, the presence of a root tag forces the current colour not to be propagated further.

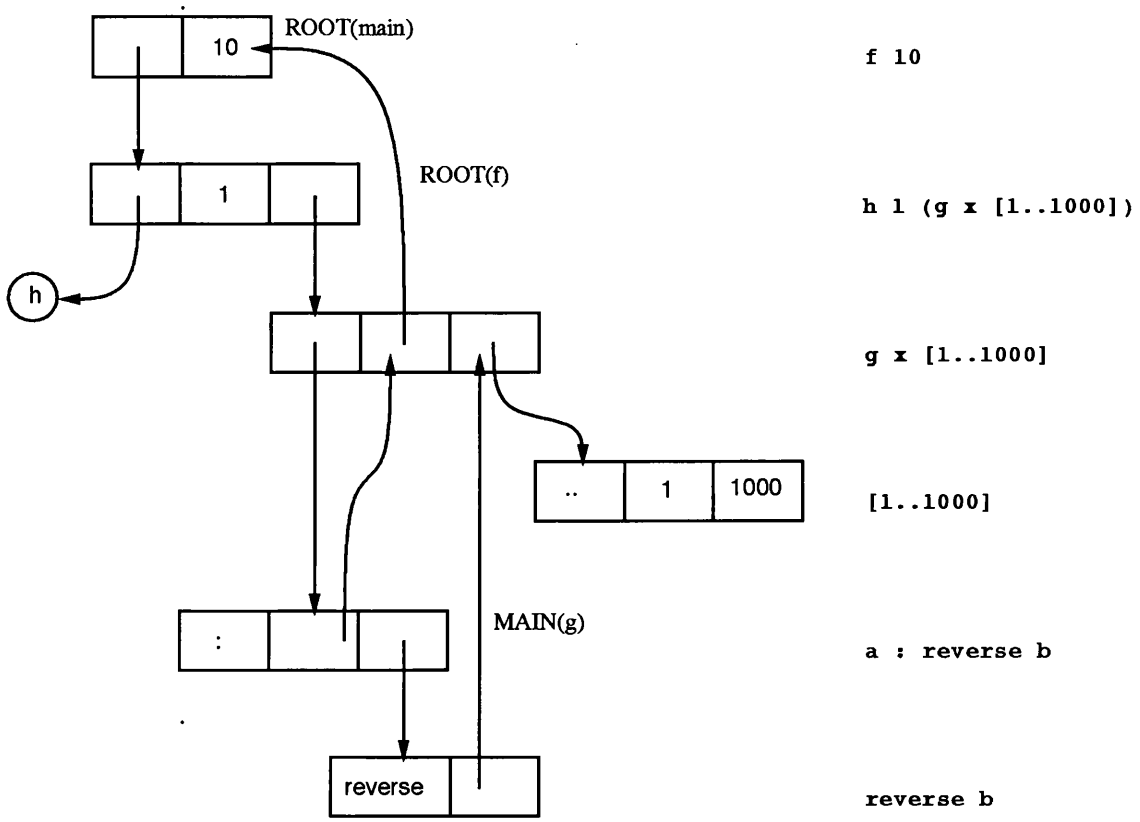


Figure 5.5: The CGF for program, plus root markers

Unprofiled Functions

When propagating profiling colours to shared unprofiled functions, it is observed that the order of doing the propagation can lead to different run-time results [28]. The shared function will inherit the profiling tag of the first function that has its tag propagated to the shared function. When another profiled function has its tag propagated to this shared function, the propagation will stop as the shared function will already have a profiling tag. However, this approach does not produce the correct results, as will be demonstrated shortly. Unprofiled functions should not be shared by more than one profiled function. Contravention of this rule is detected during the colour propagation phase when an attempt is made to paint a non-root node which already

[28] Although sharing can cause problems in the profiling stage, sharing is known to cause problems in other areas of functional programming so this is nothing unique.

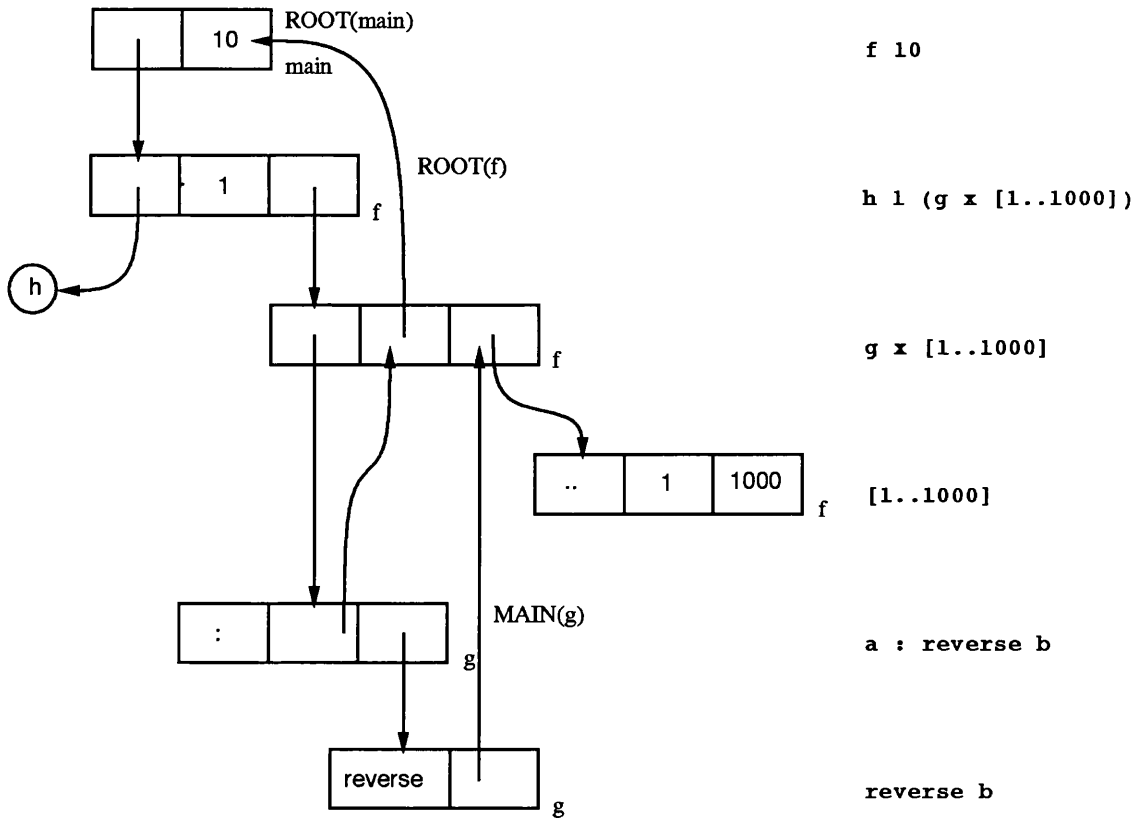


Figure 5.6: The CGF for program, plus root markers and colours

possesses another colour. When faced with the issue of shared functions, there are 3 methods the user can choose to deal with this. He can either:

a) the user can chose to profile the shared function separately. In the lexical profiler, the user is warned where there is sharing and he can chose to recompile the program with the shared function explicitly profiled.

b) let the profiler force a profiling colour onto a shared function. In the lexical profiler, all shared unprofiled functions are given a special `shared` profile colour.

c) let the profiler make a unique copy of the shared function. In this case, functions which share other functions will have their own local copy of the shared function. This has serious consequences in a lazy, functional system where sharing is used to reduce the amount of work undertaken. This

approach can be investigated as further work.

As an example of the sharing problem consider the following code:

```
f x = x + value

g x = x * value

value = [1..1000]
```

This code can be represented in CGF as figure 5.7.

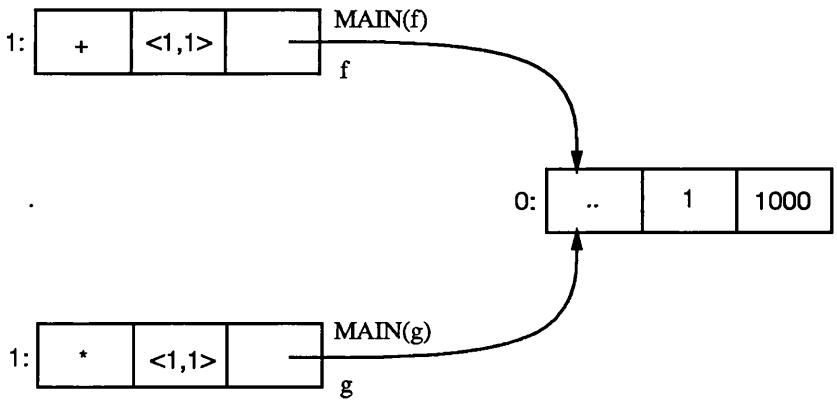


Figure 5.7: The CGF for program, showing shared expression

Note that if `f` and `g` are profiled and `value` is not, then the order in which the compiler propagates the profiling colours will determine which function gets attributed the cost of `value`. In this example, if the compiler propagates the colour for `f` first, then `value` will be attributed to `f`, but if the compiler propagates the colour for `g` first, then `value` will be attributed to `g`. Clearly this is not what is expected, but as previously described there are 3 methods to overcome this.

Once every profiled function has been coloured, transformations performed on the graph must preserve the colours so that knowledge of the lexical scoping of the original program is retained. In this way, the profiling is correct even if colours become fragmented during compile-time program transformation.

5.5.2. Execution phase - call, time, and space profiling

The run-time system monitors the call-count, time, and space data during the execution phase. Parrott's interpreter represents function definitions by graphical templates constructed and coloured at compile-time [29]. At run-time, the profile colour is copied from the appropriate template whenever a user function is instantiated. Each node of an instance is also tagged with the profile colour of the calling function. When laziness or calls to higher-order functions cause the node to be passed into another function, the tagging enables the profiler to identify not only the function from which the node originated but also its parent function.

Retaining the original lexical affiliations of nodes is of utmost importance when we come to promote execution times up the call graph to obtain final profiling statistics. The reader should note that profile colours and profile root-tags are properties of the profiling mechanism and not of the reducer. Once assigned, the colour of a node cannot be changed by the reduction process; overwriting a node's function or argument cells has no effect on its colouring.

Call-count profiling

Counting the number of calls made to a function is very simple. Each time a call is made to a function which possesses a profile root-tag, the call count for that function is incremented. This mechanism works for simple, recursive, and mutually recursive function calls.

Time profiling

The expected behaviour for time-profiling is shown in figure 5.3. Work may be done by a function both before and after a subsidiary function is invoked, hence the appropriate timing must be updated with the cost of work performed whenever control is transferred either by a function call or return.

---

[29] For a detailed introduction to the fundamentals of graph reduction, see [Peyton-Jones87]

Space profiling

Space profiling is quite simple. The colour of a newly allocated node is always set by the heap manager, which is responsible for incrementing the corresponding space-profile counter. Therefore space usage is monitored in real time rather than having visits to the whole graph at discrete intervals, as in [Runciman92]. More comprehensive data can be built if the colour of the calling function is also recorded.

5.5.3. Lexical profiling and compiled graph reduction

The techniques presented are illustrated using an interpreted model of graph reduction but they can also be implemented as part of a fully compiled abstract machine. Compiled graph reduction typically makes much more use of the stack for calculations which do not need to be written out to the heap. The heap is used when closures and shared data structures are built (e.g. see [Peyton-Jones89]). To implement the lexical profiling technique for compiled abstract machines such as the Spineless Tagless G-Machine, abstract machine instructions should be extended to carry profile-colour parameters. This would allow heap nodes to be built with profile colour tags, code sequences to pass their colours onto child sequences, and special profile markers to be constructed on the stack. The last of these extensions works in much the same way as update markers which force shared value updates in the heap (see [Fairburn87] or [Peyton-Jones89]). Code for examining the extra parameters, node colours, stack markers, and also for incrementing the relevant profile counters would be included in the executable binary.

5.5.4. Extending the technique to parallel graph reduction

Lexical profiling can be extended to parallel graph reduction by distributing the *from tables*. Each processing element will have its own *from table* which will be updated in the usual manner. At the end of a program run, a new *from table* is generated from the sum of the data in every *from table*. This distribution and accumulation can be accomplished due to the properties that allow a functional program to run in parallel and because of the way lexical profiling colours the program before the

execution stage. Once a program is coloured, it is not significant where a function executes.

5.6. Analysis of the Lexical Profiler

This section shows the output from a profiling session in order to illustrate the information given by the lexical profiler. The results presented here were obtained using the UCL experimental interpreting reducer rather than an optimized compiled reducer. The space usage is presented in cells rather than bytes. This is significant because the number of cells remains the same when executing the program either with profiling or without profiling. The profiling data for time and call-count is presented separately from the space usage data. Execution times are accumulated and reported for every profiled function. The time and call-count for each function is subdivided according to the functions that called it. These times denote the actual execution time rather than the elapsed wall-clock time. The time for garbage collection is presented separately and is *not* included in the time for any function.

At present the timings for each function are at a resolution of 20ms. As stated earlier, this is a limit of the current hardware rather than the lexical profiling technique. On hardware with a real-time clock the results would be more accurate.

5.6.1. Observing Program Behaviour

In this section the profiler is used to observe the behaviour of two programs. The results of profiling the well known functional program nqueens and a small relational database are presented.

the nqueens program

The nqueens program tries to put n queens on a chess board such that they are all in a *safe* position. The program can attempt to put from 1 queen up to 8 queens on the board, and it returns all the valid results. In the following test the first 10 valid results, with 7 queens on the board, is profiled. The code of the nqueens program is:

```
queens :: Int -> [[Int]]
queens 0          = [[]]
queens (m+1)      = [ p++[n] | p<-queens m, n<-[1..8], safe p n ]


safe :: [Int] -> Int -> Bool
safe p n          = all not [ check (i,j) (m,n) | (i,j) <- zip [1..] p ]
                    where m = 1 + length p


check :: (Int,Int) -> (Int,Int) -> Bool
check (i,j) (m,n) = j==n || (i+j==m+n) || (i-j==m-n)


main = take 10 (queens 7)
```

The nqueens program was compiled for profiling and results were asked for the functions `queens`, `safe`, `check`, and `main`. The space usage of nqueens is presented in figure 5.8. Each line represents the number of cells used by a function over time. The spikes in the lines represent where cells are used by a function and then garbage collected when they are not needed. The space results presented give similar information to the Runciman and Wakeling heap profiler but are in a different form. Runciman and Wakeling present their data as cumulative strata, whereas the lexical profiler presents the data for each function absolutely [30]. With the space usage data alone, attention is drawn to the `queens` function as it uses the most space. *Shared* code is presented by the lexical profiler because some functions are shared but, as can be seen, these shared functions are an insignificant factor in the compuation.

[30] It would not be difficult to post-process the space usage data to generate a report in the style of Runciman and Wakeling.

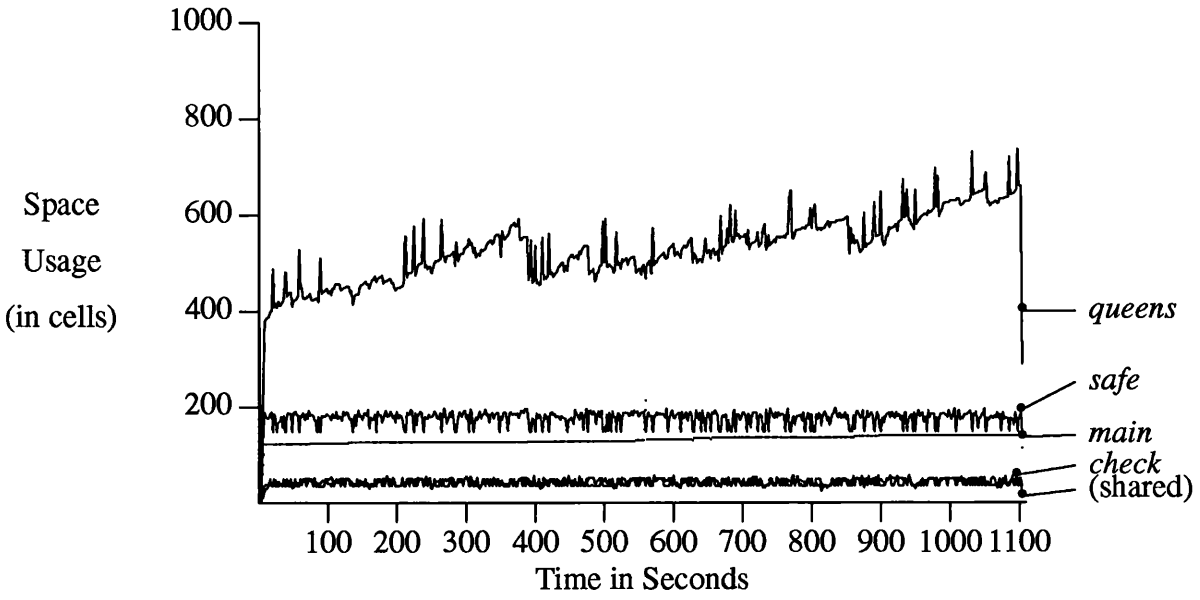


Figure 5.8: Heap usage results for nqueens program

In	From	No of Calls	Time in seconds
main		1	1.12
queens	main queens	1 7	22.48 69.82
safe	queens	742	520.76
check	safe	2003	103.06
Garbage collection time in seconds			197.86

Table 5.3: Call-count and timing results for nqueens

Lexical profiling also produces both call-count and timing data. By analysing the data in table 5.3, it is possible to gain further insight into the behaviour of the program. From the data in table 5.3, attention is not drawn to the function `queens` but to the

function `safe`. The `safe` function has 742 calls to it and the accumulated time is 520 seconds, which is 70% of the program execution time. Clearly, it is the `safe` function which could benefit from some optimisation. If only a heap profiler were available, it would be impossible to determine that this behaviour arises. The function `safe`, when given a list of current queen positions on the board and a possible new queen position, evaluates whether the new queen can be safely placed on the board. The list comprehension does the arranging of the checks to see if the queen can be taken in the new position, and the results are processed using the term `all not`, which determines if each element of the list comprehension is false. By looking at the definition of `all`, it becomes apparent that a more efficient function can be written to determine if every element of a list is false. The code for `all` is:

```
all :: (a -> Bool) -> [a] -> Bool
all p           = and . map p


and :: [Bool] -> Bool
and             = foldr (&&) True
```

This code is inefficient in this case because `all not` inverts every element of the list before evaluating the `and` term. This is unnecessary, and a new all-false function can be defined as:

```
allFalse :: [Bool] -> Bool
allFalse []         = True
allFalse (True:r)   = False
allFalse (False:r) = allFalse r
```

and `safe` can be redefined as:

```
safe p n          = allFalse [ check (i,j) (m,n) | (i,j) <- zip [1..] p ]
                    where m = 1 + length p
```

The call-count and time data of the new version of the program are presented in table 5.4.

In	From	No of Calls	Time in seconds
main		1	1.20
queens	main queens	1 7	22.58 66.34
safe	queens	742	424.64
check	safe	2003	98.38
Garbage collection time in seconds			174.60

Table 5.4: Call-count and timing results for new nqueens

`safe` now executes in 80% of the time that it used to and the whole program is 15% faster. This shows the benefit of having call-count and time profiling data. The space profile is very similar to the previous one, and is not shown here.

simple database program

Here profiling data is presented for an example program that is a demonstration of a simple relational database written in the functional style. The program is written in Haskell and contains approximately 350 lines of Haskell source code. The database program provides the functionality to display a table, to select rows from a table, to project columns from a table, to generate the union of two tables, and to join two tables to produce a new one.

The profile shown in figure 5.9 is for a run of the database program that displays a table generated by joining two existing tables. The heap space usage of this program is presented in figure 5.8 and was gathered using the *inheritance* style of profiling.

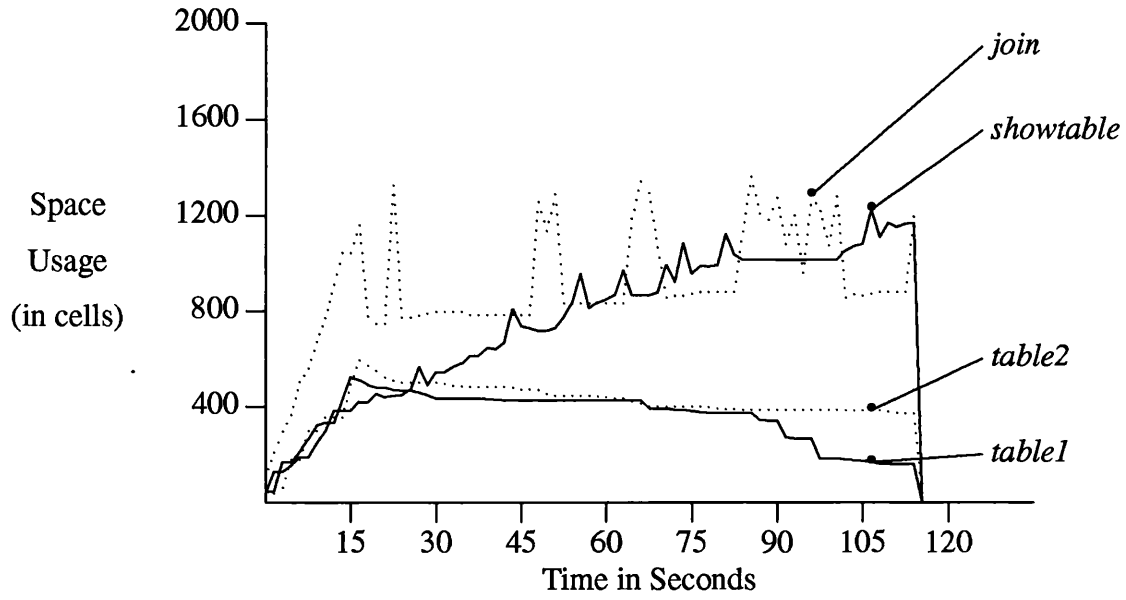


Figure 5.9: Heap usage results for database program

In figure 5.9, the line for the function *showtable*, which displays the resulting table, rises continuously throughout the program run. This continuous rise draws attention to the possibility of *showtable* having degenerate behaviour. For the function *join*, the line rises steadily as cells are allocated and after 10 seconds of execution time fluctuates from the 800 cells level. For both *table1* and *table2* there is a rise as cells are allocated, then their usage of cells reduces slowly.

In	From	No of Calls	Time in seconds
main		1	0.02
showtable	main	1	28.74
join	main	1	24.78
table1	main	1	0.66
table2	main	1	0.94
shared code	main	0	31.22
Garbage collection time			22.06

Table 5.5: Call-count and timing results for database program

The timing data for the database program is displayed in table 5.5. Notice that one third of the program execution time was spent in shared code. This indicates to the programmer that much of the execution time was spent in functions that were not profiled explicitly. In order to gather more detailed information, the program should be compiled with more functions being profiled. It is beneficial for the compiler to warn the user when a function is being shared and by which functions it is being shared. The compiler used in this PhD does this. In appendix A, there is an example of the call-count and time data for this program which was gathered by profiling *every* function in the database program.

By analysing the space and time data for this program, one can see that the `showtable` function hangs onto the space it uses until the end of the program run. Therefore, there needs to be a further investigation of this function in order to determine the cause of the observed behaviour. The code for `showtable` is:

```
showtable::Table->[Char]
showtable table
        = disp_title ++ "\n" ++ disp_colhdr ++ "\n" ++ disp_row el
          where
          disp_title = concat ["Table name: " , name ,
                               "\tPrimary key: " , pk ,
                               "\tForeign key: " ,fk]
          disp_colhdr = concat (map ((ljustify 10).fst) colhdr)
          disp_row [] = []
          disp_row (r:rs) = concat (map (ljustify 10) r) ++
                            "\n" ++ disp_row rs
          Table name pk fk colhdr el = table
```

`showtable` makes use of `concat` and ++ to generate output when given a table. A test profile of both `concat` and ++ is undertaken. To focus the test, `showtable` is applied to just one table. The space usage for this test is displayed in figure 5.10 and the timing data is displayed in table 5.6.

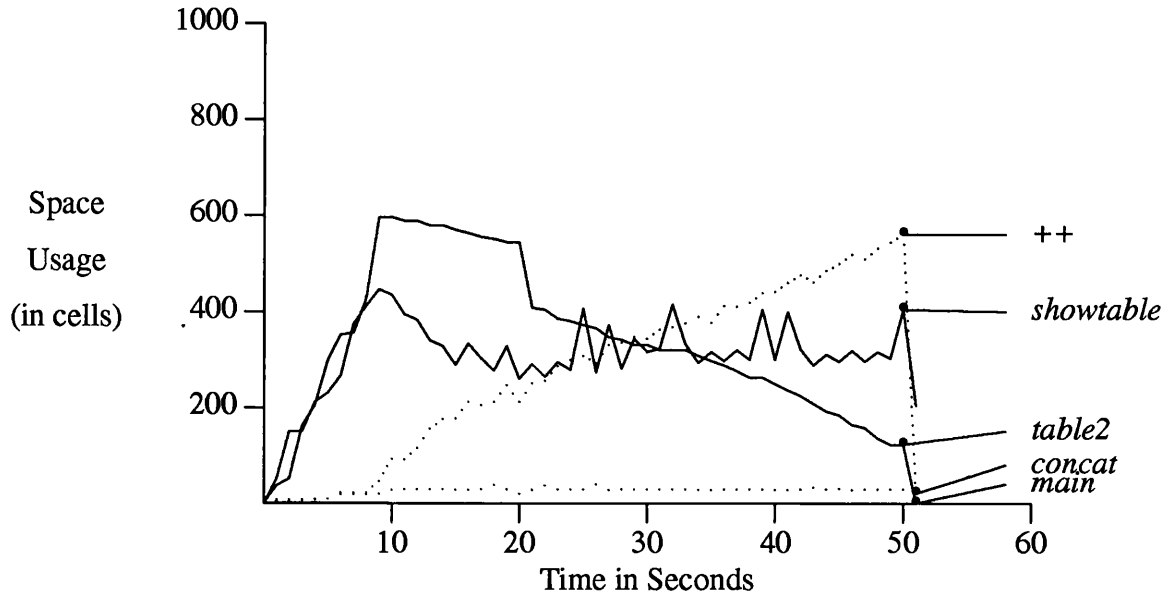


Figure 5.10: Heap usage results for `showtable`

In	From	No of Calls	Time in seconds
main		1	0.00
showtable	main	1	14.88
table2	main	1	1.10
concat	showtable concat	7 24	0.24 0.46
++	showtable concat	14 24	6.56 7.06
Garbage collection time in seconds			7.01

Table 5.6: Call-count and timing results for `showtable`

It can be seen that the append function ++ is actually the cause of the problem. It hangs onto cells until the end of the program run. Further tests were run to try to eliminate this problem, and many definitions of ++ were tried without success.

5.6.2. Verifying Program Behaviour

In this section the profiler is used to verify the behaviour of two programs. The first case-study uses the profiler to verify whether or not the function `foldr` is tail strict and the second case-study uses the profiler to verify if a hand-coded function performs better than a pipeline which does the same job.

foldr

In an example from Runciman and Wakeling's heap profiling paper [Runciman92], they observe that "certain functions can cause `foldr` to be *tail strict* ". To verify this belief, a case-study was constructed which passes a simple function to `foldr`. One version of the simple function forces `foldr` to become strict and another version uses

`foldr` lazily. In this case-study, a *cons* function is passed to `foldr` and then the head of the resulting list is taken. The first version of cons does pattern matching on its second argument. The code used in this example is:

```
pmCons :: a -> [a] -> [a]
pmCons v [] = v : []
pmCons v (h:t) = v : h : t

list :: [Int]
list = foldr pmCons [] [1..100]

main :: Int
main = head list
```

By profiling this program it can be seen that `foldr` has become strict. There are 100 calls to `pmCons` when only 1 is expected, and the heap usage is large when it is expected to be small. The call-count and timing data for this example is displayed in table 5.7 and the heap usage is displayed in figure 5.11.

In	**From**	**No of Calls**	**Time in seconds**
main		1	0.08
list	main	1	7.66
pmCons	list	100	1.20
foldr	list foldr	1 100	0.02 2.14
Garbage collection time in seconds			2.76

Table 5.7: Call-count and timing results for `pmCons`

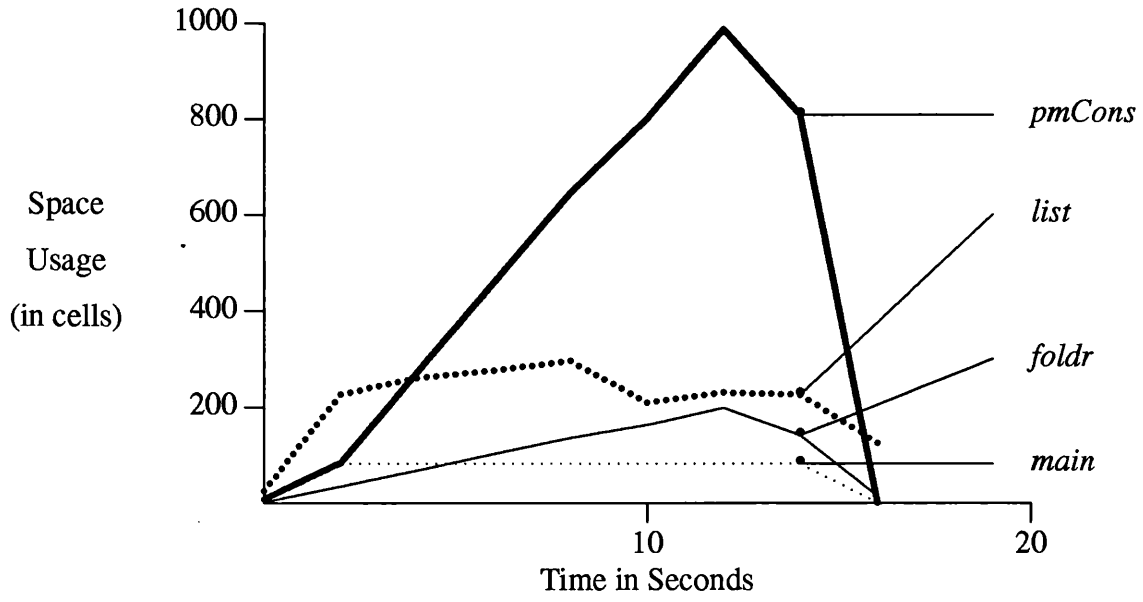


Figure 5.11: Heap usage results for `pmCons` ***program***

The *cons* function can be rewritten without pattern matching:

```
gdcons :: a -> [a] -> [a]
gdcons v l = v : l


list :: [Int]
list = foldr gdcons [] [1..100]


main :: Int
main = head list
```

As can be seen in table 5.8, the program goes much faster and the number of calls to the cons function is the expected number, i.e. 1. Clearly, the pattern matching is a problem when combined with `foldr`.

In	From	No of Calls	Time in seconds
main		1	0.02
list	main	1	0.10
gdcons	list	1	0.00
foldr	list	1	0.00
Garbage collection time in seconds			0.06

Table 5.8: Call-count and timing results for `gdcons`

sum of squares

The second example is a program to sum the squares of a list of numbers. In [Ferguson88], Ferguson suggests that the pipelining style of programming (through the use of function composition), which is common in functional languages, is inefficient as there is a need to build and immediately destroy intermediate list elements. A more efficient version can be written which has the same semantics and operational behaviour as the pipelining version. However, this efficient version has the disadvantage that it is considerably less clear than the pipelining version. In this section, the profiler is used to verify Ferguson's statment. Ferguson defines the sum of the squares to be:

```
(sum . map square . upto 1) n
```

A program to evaluate this expression is:

```
sumSquares :: Int -> Int
sumSquares n = (sum . map square . upto 1) n

upto :: Int -> Int -> [Int]
upto n m = if n > m then []
           else n : upto (n+1) m

square :: Int -> Int
square x = x*x

main = sumSquares 400
```

By profiling this program, the results obtained for call-count and function times are displayed in table 5.9 and the heap usage results are displayed in figure 5.12.

In	From	No of Calls	Time in seconds
main		1	0.00
sumSquares	main	1	0.04
sum	sumSquares	1	8.34
map	sumSquares map	1 400	0.02 8.64
upto	sumSquares upto	1 400	0.00 9.52
square	sumSquares	400	2.04
Garbage collection time in seconds			7.30

Table 5.9: Call-count and timing results for `sumSquares`

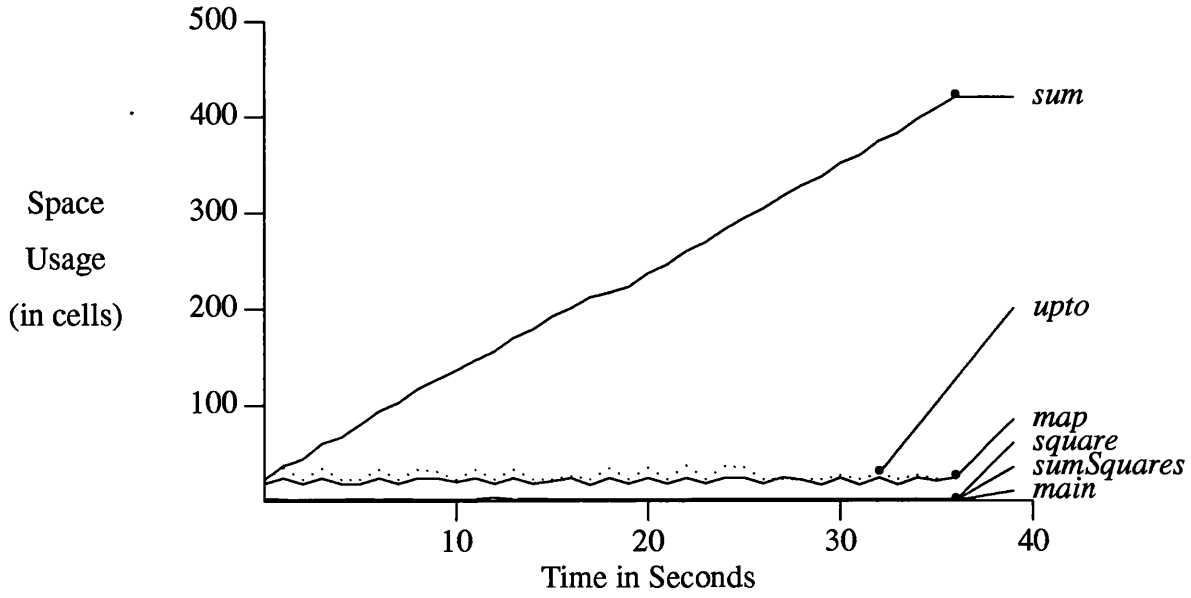


Figure 5.12: Heap usage results for `sumSquares` ***program***

A second version of the sum of squares program, which Ferguson says is more efficient, is:

```
sumNsquares n = sumNsquares' 0 1 n

sumNsquares' res m n = if m > n then res
                       else sumNsquares' (res + square m) (m+1) n

main = sumNsquares 400
```

The results of profiling this program are displayed in table 5.10 and figure 5.13.

In	From	No of Calls	Time in seconds
main		1	0.00
sumNsquares	main	1	0.00
sumNsquares'	sumNsquares sumNsquares'	1 400	0.04 9.18
square	sumNsquares'	400	1.94
Garbage collection time in seconds			2.98

Table 5.10: Call-count and timing results for `sumNsquares`

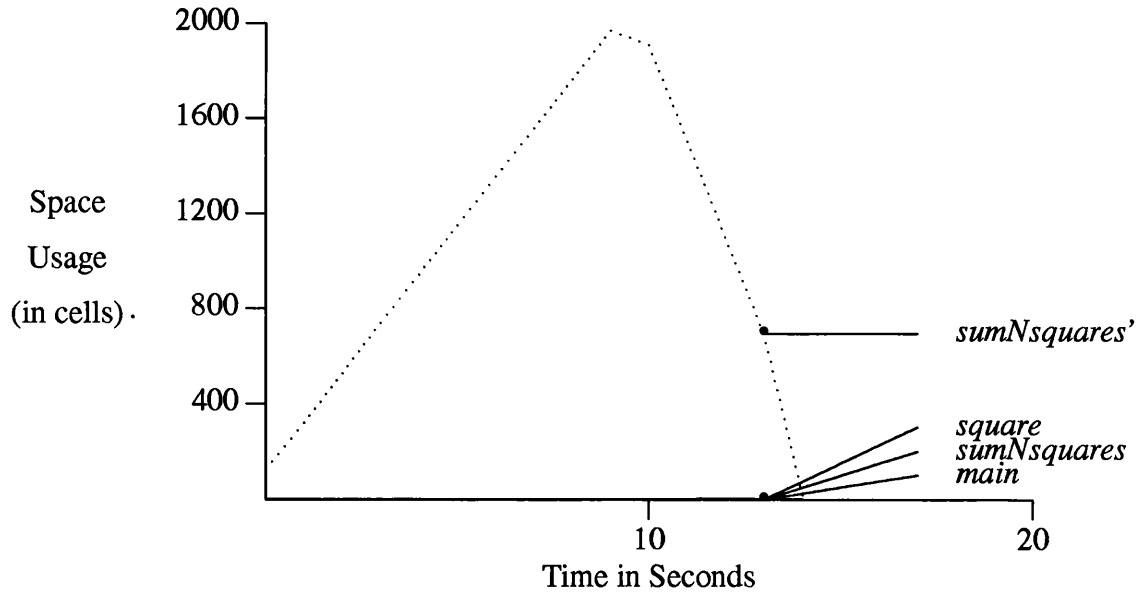


Figure 5.13: Heap usage results for `sumNsquares` *program*

Therefore, Ferguson is correct in stating that the second version of sum of the squares is faster, because the second version executes in 14 seconds whereas the first version took 36 seconds. However, in the second version, the space usage has a larger peak than the first version.

5.6.3. Achievements of Lexical Profiling

The achievements of the lexical profiling technique for lazy, higher-order functional languages are reviewed with respect to suggestions made by Runciman and Wakeling regarding problems that profilers for functional languages might have. In [Runciman90], Runciman and Wakeling suggest that profiling tools such as **gprof** are of limited use for profiling functional programs. The reasons they give are:

1. *The semantic gap*: they comment that functional programs do not map directly into a machine representation and require much transformation. They claim that measurements of a run-time profiler may be difficult to associate with structural units of source code.

   This is one of the motivations for lexical profiling. The structural unit is the function definition and is independent of any later transformations. Results are associated directly with the textual function definition, which the programmer understands, and do not depend on the run-time representation.

2. *Hidden routines*: they claim that routines in the functional run-time system may carry out a significant proportion of the computational work. For example, the full cost of garbage collection would go to the function that needed some memory and induced the garbage collection.

   During garbage collection, a lexical profiler can stop measuring execution time and start measuring garbage collection time. When returning to the evaluation, the profiler can continue measuring for the correct function according to its profile colour.

3. *Global laziness*: they maintain that lazy evaluation makes it difficult to assess the cost of isolated program parts and claim that changing a program may change the point of evaluation.

   The lexical profiler counteracts this dilemma because profiling information is always reported with respect to the lexical scoping of the source. Therefore, the results are insulated from the effects of laziness. It is possible to differentiate between real function calls and changes of context due to delayed evaluation.

4. *Space leaks*: they observe that having laziness means expressions may be held unevaluated for later use and that the lazy evaluation strategy can cause large demands on memory usage.

   In the lexical profiler, space usage can be observed and measured. As every cell is tagged, it is possible to measure cell usage separately for every function.

5. *Recursion and cycles*: they observe that **gprof** is poor at handling the recursive functions which functional programs rely on.

   In a lexical profiler the root of every profiled function is tagged. Therefore, recursive functions can be handled correctly.

This PhD concludes that lexical profiling overcomes apparent obstacles in building an effective profiler for lazy, higher-order functional languages.

5.7. Summary

One of the major problems in developing applications in lazy, functional languages is the lack of tools which aid the programmer in debugging and analysing the run-time behaviour of the application. This chapter addressed this issue and presented the design and implementation of a profiler which measures call-count, time, and heap space usage of lazy, higher-order functional languages using a technique called *lexical profiling*. This is of benefit to the applications programmer because results can be directly related to the source code and no knowledge of the underlying run-time system is required.

Furthermore, neither profiling annotations nor primitives need to be learned as lexical profiling allows the program to be executed unchanged. The lexical profiling technique is the only one that can present results for all 3 types of data, namely call-count, space, and time.

The programmer can run his program with both statistical and inheritance profiling in separate runs and compare the output to determine how his program behaves. When the programmer is comparing the graph from the inheritance profile with the graph from the statistical profile, he can determine whether an inherited function is causing a lot of resource usage.

The use of lexical profiling was demonstrated by examining example Haskell programs. From these examples, it was shown how the profiler presents data on the execution of the program and allows problem areas in the code to be identified. A lexical profiler allows the programmer to observe the execution of functional programs by observing *where* events occur and *what* they signify.

The task of profiling functional languages relies on two tenets:

(i) using lexical function definitions rather than a run-time representation.

(ii) ensuring that the compiler preserves the lexical affinities irrespective of program transformations.

Existing approaches have had limited success in profiling lazy, higher-order functional languages. In order to overcome these limitations, one can use the lexical profiling technique to build a working profiler for functional languages.

One of the benefits of lexical profiling over the annotation style of profiling is that the programmer does not have to change any code to do lexical profiling. The compiler and the run-time system will do all the work. With cost annotations, the programmer has to decide where to place the annotations and which expressions will give meaningful results. There are many problems with this technique, and they are discussed in [Sansom92]. Sansom also reviews the paper [Clayman91] which is an early description of the current work. He comments that:

- unprofiled functions cannot be shared by more than one profiled function

  As stated sharing unprofiled functions is undesirable because when a profiled function shares unprofiled functions the results produced will be incorrect. Section "Unprofiled Functions" these arguments in more detail.

- separate module compilation is not possible

  In the current, experimental version there is no separate module compilation. However, in a production version this limitation can be overcome. It is possible to design a system which allows modules to be compiled separately by keeping the colours of all profiled functions in a special profiling symbol table. At link time, the profiling symbol tables can be combined by a phase of the linker to produce the full colouring of the program.

Recently Sansom has adapted his work to encapsulate techniques from lexical profiling in his cost centers [31].

Therefore, lexical profiling is a fundamental development in run-time analysis tools for lazy, functional languages. Its results are reported with respect to the source code, which every programmer understands.

A conclusion is that the profiler produces data which has not been seen before, and therefore work needs to be done to understand the graphs that are produced. One obvious result is that a higher line on a space usage graph indicates that more space is being used by a function. However, more exposure to the results of lexical profiling are needed in order to provide more comprehensive knowledge of the meaning of profiling results. The further work required is:

i) the analysis of space usage graphs

ii) definition of what peaks and troughs mean in space usage graphs

---

[31] Personal communication.

iii) to provide the programmer with a list of changes that he can make to programs when presented with certain patterns of data from the profiler.

In the current version of the profiler statistical profiling has not been implemented. If statistical profiling data is required it can be generated by post-processing the run-time results of the existing profiler. No extra changes need to be made to the compiler or the run-time system of the functional language. The post-processor can collect the function call counts and the time spent in functions in order to generate the percentage of time spent in each profiled function.

The extensions to be made to the profiler are:

i) to add constructor profiling. Runciman and Wakeling do this in their profiler. With constructor profiling, the space used by each function is not presented as one homogeneous amount but is presented per constructor allocated by that function. The space results will have a report for each constructor. This will give the programmer both more information and clearer details as to how a function is allocating space.

   At present the lexical profiler does not do constructor profiling. It uses FLIC as its input language and any indication of the names of constructors have been stripped by the Haskell compiler. In the FLIC source, only `PACK` 's are seen.

ii) to allow copying of the body of shared functions. At present, shared functions can be profiled either individually or together using the `shared` profile colour. The desire is to make a copy of each shared function, thus making the copy local to each function that needs it.

Chapter 6

6. Parallelism and Functional Programs

This chapter reports the discoveries from attempts to find the best technique to parallelize a large functional application and considers the advantages and disadvantages of annotations, skeletons, and compiler detected parallelism when parallelizing a large application. The work is reported from the view of the programmer trying to evaluate the available tools and techniques; the views of parallel system implementors may be very different.

Parallelism in functional programming is appealing because expressions within a program are independent and the lack of data dependencies within a program permits the concurrent evaluation of these expressions. The functional program which executes on a sequential machine can just as easily execute on a parallel machine. In [Peyton-Jones89a], Peyton-Jones indicates that parallel functional languages have advantages over parallel imperative languages. These advantages are:

- no new language constructs are required to express parallelism, nor are there any sychronization or inter-task communication constructs. This is because all parallelism can be implicit.
- no special techniques are needed to protect shared data from concurrent tasks. This is because there is no updatable store and no side-effects.
- it is no more complicated to reason about the correctness of a parallel functional program than a sequential program. This is because no new constructs have been added, so all the same techniques still work.

- the results of a program are determinate. This is because the model of computation has not changed for the parallel environment, therefore, any variance in processing and communication speeds is irrelevant.

Although functional programs use implicit parallelism to achieve a reasonable speed-up, functional algorithms must be designed with parallelism in mind. For example, the function *sum*, which generates the sum of the numbers 1 to n, can be written as:

```
sum 1 = 1
sum n = n + sum (n-1)
```

However, this function can only be executed sequentially because the data dependencies for the additions occur one after another. Peyton-Jones shows how a parallel version of sum may be written:

```
sum = psum 1 n

psum lo hi = hi, hi == lo
           = psum lo mid + psum (mid+1) hi, otherwise
             where
             mid = (lo + hi) / 2
```

This version of *sum* decomposes the workload into two separate parts, (i) the sum from 1 to a mid-point and, (ii) the sum from the mid-point to n. The workload is recursively decomposed, with each task evaluating its part of the sum.

Many techniques for identifying and extracting parallelism in functional programs have been devised; they are annotations, skeletons, and compiler detected parallelism. Once the program has been parallelized, the individual tasks have to be mapped onto processors to make effective use of the machine. To do this requires some management of the parallel environment. Techniques for task management include load balancing, scheduling, and partitioning.

This chapter discusses three ways to harness parallelism in functional programming, namely the use of annotations, the use of skeletons, and the of compiler detected

parallelism. The investigation into techniques currently available for parallelizing the functional rule-based system led to the use of the GRIP parallel processor [Clack85a] as it was the only one available during this research. An experiment was devised to test the suitability of running a program in parallel on GRIP, and the results obtained from this experiment are presented. Then follows a discussion on the use of annotations, skeletons, and compiler detected parallelism in other parallel systems. This leads into a review of parallelism in functional programming, and in particular the advantages and disadvantages of these three techniques. There is a brief section on current parallel applications and, finally, conclusions are presented concerning the best method for harnessing parallelism in a functional program.

The requirements for parallelizing small programs are often different from large programs and thus the arguments presented in this chapter may not be relevant for small programs.

6.1. Parallelism in Functional Programming

This section discusses currently available technology for identifying parallelism in functional programs and for managing that parallelism in a parallel environment. There are currently three ways to identify parallelism in functional programs – compiler detected techniques, skeletons, and annotations.

6.1.1. Compiler Detected Parallelism

Compiler detected parallelism is a technique in which a phase of a compiler analyses the source code to determine which parts of the program may run in parallel. This is most commonly done through the use of strictness analysis [Clack85]. Strictness analysis determines if the value of expressions will be needed at some time in the future. If they are needed, then the expressions may become new parallel tasks. To enhance compile-time strictness analysis, Burn has proposed *evaluation transformers* which allow the strictness data to be modified at run-time when more information becomes available [Burn87].

6.1.2. Skeletons

Skeletons embody general structures of computation within a functional framework [Cole90] [Darlington91]. Skeletons are higher-order functions which provide building blocks for the specification of parallel algorithms. The programmer uses a skeleton function within a program to denote the kind of structure an algorithm has, but the skeletons do not change the meaning of the code. The algorithm can then be run efficiently on a parallel machine.

Consider an example where the programmer knows that a set of functions are to be combined into a pipeline. Pipelines are commonly written as compositions, so the programmer may write code such as:

```
(f1 · f2 · f3 · ... fn) data
```

Yet this forces the functions to be composed and, as a consequence, little parallelism may occur. However, by using skeletons, the programmer may express the pipeline as:

```
pipeline [f1, f2, f3, ... fn] data
```

With this construct, the parallelism may be generated in different ways on different parallel machines depending on which is the most efficient. The skeleton allows the programmer to express his knowledge of how the functions are to be combined in an abstract way.

6.1.3. Annotations

Annotations are declarations which the programmer hand-places into programs in order to specify where the parallelism should occur [Hudak85] [Hammond91b]. The annotations do not change the semantics of the program and, therefore, the program will give the same results when annotations are not used. Annotations are used because it is sometimes difficult for a compiler to determine where all the parallelism is. Reconsider the psum example. If the programmer uses annotations to harness parallelism, then the code for the parallel sum could be written as:

```
sum = psum 1 n


psum lo hi = hi, hi == lo
           = PAR (psum lo mid) + PAR (psum (mid+1) lo), otherwise
             where
             mid = (lo + hi) / 2
```

Here the annotation PAR indicates that both arguments to the addition operator are to be executed in parallel.

6.1.4. Managing Parallelism

The management of the parallel environment aims to ensure that the machine is being used effectively. Load balancing is a mechanism which tries to give every processor of a parallel machine an equal amount of work to perform [Hudak84]. This may involve moving tasks from busy processors to idle processors in order to attain the balanced load. Partitioning is a mechanism which splits programs into tasks and then splits these tasks into smaller sub-tasks [Hudak85a] [Goldberg88]. Each task can be executed concurrently with other tasks. Once new tasks have been created, it is the scheduling mechanism which decides which ones to execute [Goldberg88] [Hammond91a]. If no tasks are available to schedule on the current processor, then some load balancing is required to migrate tasks to that processor. If there are no tasks available anywhere in the parallel environment, then some existing tasks need to be partitioned in order to create new tasks.

These three mechanisms (partitioning, scheduling, and load balancing) are closely related and each can be done either statically at compile-time or dynamically at run-time.

6.2. Use of parallel systems

In this section there is a discussion on the use of functional programming systems which have parallelism. Many of the earlier systems which have been reported, such as Hudak's early work (see [Hudak84], [Hudak85], and [Hudak85a]) or ALICE [Cripps87], no longer seem to exist. Recent systems, such as the FAST project which aims to build an implementation of Haskell on a machine consisting of Transputers [Glaser90], are still under development and are not available. Some of the reported systems, which at first seemed promising, only ran on sequential machines or on simulators of parallel machines [Eekelen89].

Attempts to use other systems that have been reported have varied. Access to both the ALICE parallel machine and the FLAGSHIP parallel machine was unavailable during this research. Access to the GRIP parallel machine was encouraged by its administrator and he supplied many documents on how to access and use the system [Hammond91]. The following section describes the use of the GRIP parallel machine in order to investigate its suitability for executing a parallel version of the functional rule-based system written for this thesis.

6.2.1. Use of GRIP

The GRIP parallel machine is now publicly accessible over the Internet as a mail server. Either Lazy ML [Augustsson89] or Haskell [Hudak88] programs can be sent to GRIP for execution in a parallel environment. The document [Hammond91] describes how this is done. The GRIP environment provides a subset of the Haskell prelude plus support for some annotations to harness parallelism. Programs are executed on GRIP and results are mailed back to the originator for analysis. Having GRIP set up as a mail server allows wider access to the machine. As the GRIP machine is accessible over the Internet, it allows some experiments to be undertaken in parallel functional programming. Without access to GRIP, no parallel functional programming could have been done for this PhD. This section discusses how the parallelism is harnessed and describes some of the results obtained back from GRIP.

An experiment was devised to evaluate the strictness analysis technique used on GRIP by defining a small program to calculate a fibonacci number. The fibonacci program is used because it is a simple program that is a well understood and often used test case in functional programming circles. Its use here is to highlight the parallelism available in GRIP and it is not meant to be a representative functional application. Once the feasibility of executing a parallel program on GRIP has been established, then larger examples can be used for further evaluation of the parallel machine.

In [Clack85], the strictness analysis technique was proposed as a mechanism for determining where parallelism is available in a program. Run-time task management is used to manipulate that parallelism in the GRIP machine. The following program, which is a standard fibonacci function, was sent to GRIP to evaluate the use of strictness analysis in the compiler:

```
fib :: Int -> Int

fib 0 = 1

fib 1 = 1

fib n = fib (n-2) + fib (n-1)


main _ = show (fib 15)
```

The results from GRIP can be seen in figure 6.1. This activity report shows that the processor called "14.1" was 100% busy most of the time, as indicated by the solid line.

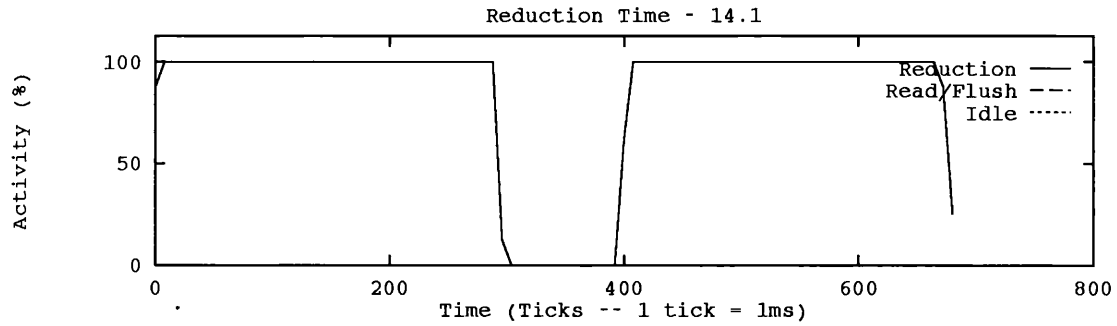


Figure 6.1: GRIP activity chart for `fib` ***program***

The activity report for the other GRIP processors is the same as that seen in figure 6.2 and shows that these processors were idle.

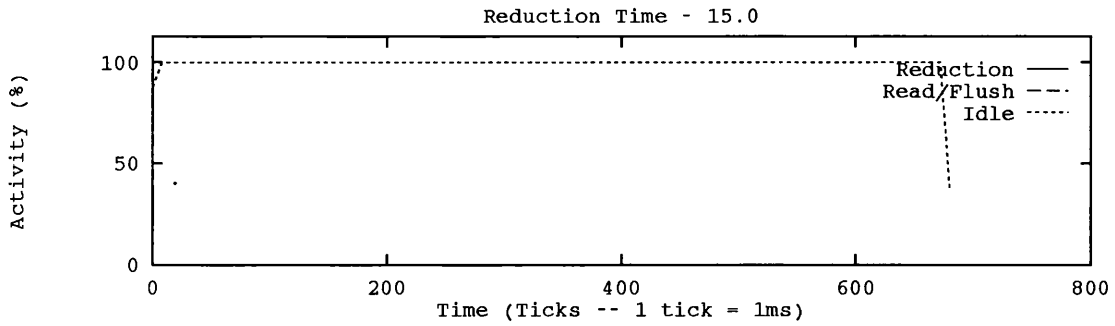


Figure 6.2: GRIP activity chart for `fib` ***program***

To confirm that only one processor was busy and that no parallel tasks were being created, the task creation report for processor "14.1" was analysed and the results seen in figure 6.3. This confirms that no parallel tasks were being created by GRIP.

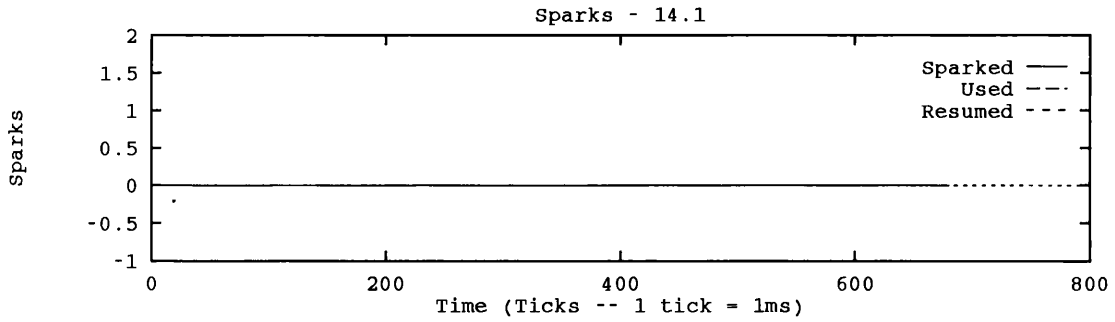


Figure 6.3: GRIP activity chart for `fib` ***program***

Finally, the aggregated report for all processor activity in GRIP is analysed and seen in figure 6.4. Figure 6.4 shows that GRIP was 5% busy and 95% idle during the execution of this program.

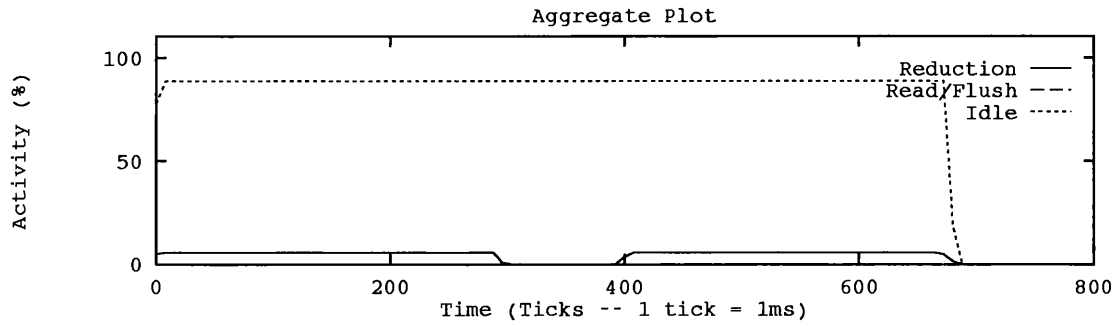


Figure 6.4: GRIP activity chart for `fib` ***program***

The results of this experiment show that the current GRIP environment does not use the strictness analysis technique, but *only* uses annotations embedded in the program to create new parallel tasks. The program sent to GRIP executed on a single processor only, rather than on many processors as would be expected when strictness analysis is used. There have been no reports that the strictness analysis technique does not work, so it is suprising that the method for harnessing parallelism seems to have changed since GRIP was first envisaged. Peyton-Jones considers that strictness analysis is still the best way forward but, in the short term, annotations are an easier way to harness parallelism [32]

For a second experiment, another version of fibonacci was created using GRIP annotations in order to harness some parallelism. In [Hammond91b], Hammond and Peyton-Jones describe some of their early work using GRIP for executing parallel programs and show the results of some simple experimental programs such as a parallel fibonacci program and a parallel 8-queens program. The results are somewhat erratic and they conclude that "some kind of dynamic thread control is necessary to control excess parallelism in the fine-grained case". In [Hammond91a], Hammond and Peyton-Jones address some of the issues raised in their early work. Neither [Hammond91b] nor [Hammond91a] suggest how the annotations are used or how tasks should be created at the source program level.

[32] Personal communication from Peyton-Jones

To harness parallelism on GRIP, the annotation `par` must be used, such that:

```
par new expression  =  expression
```

The `par` annotation causes the expression `new` to become a new parallel task which is then sent to a task pool. The expression `expression` is evaluated on the current processor. The returned value is `expression`; the annotation `par` is only used to create new parallel tasks. For example:

```
par small big
```

would send a small task to the task pool and evaluate a big task on the current processor. If the big task needed to be split into smaller tasks, then it would need more `par` annotations to create the new tasks.

The aim of the second experiment was to devise some annotations that would create many new tasks in the task pool. This approach was taken in order to maximise the amount of parallelism available in the fibonacci program. Using annotations, the following function was devised:

```
twopar f a b = par a (par b (f a b))
```

The `twopar` function creates two tasks, `a` and `b`, to run in parallel and applies `f` to the results on the local processor. Using the `twopar` function, the new version of fibonacci is:

```
fibTP :: Int -> Int
fibTP 0 = 1
fibTP 1 = 1
fibTP n =  (fibTP (n-1)) `padd` (fibTP (n-2))

        .
padd = twopar (+)


main _ = show (fibTP 30)
```

The results of processor activity from GRIP are seen in figure 6.5.

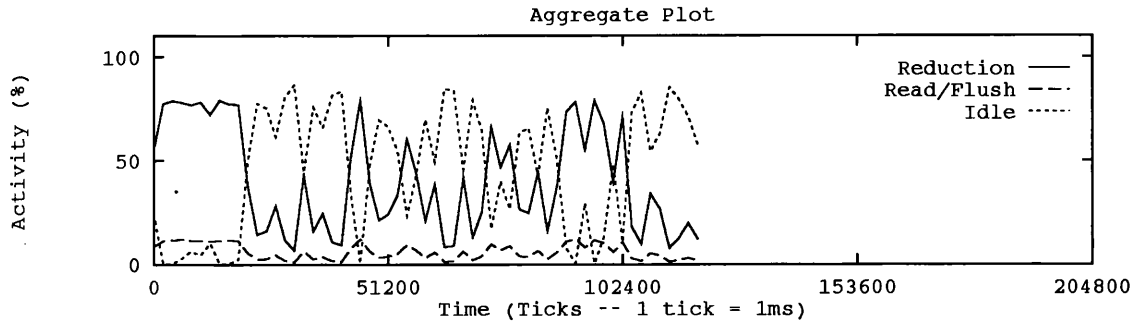


Figure 6.5: GRIP activity chart for `fibTP` ***program***

Figure 6.5 shows that out of the 120 seconds of execution time, 43% was spent evaluating, 49% was spent idling, and the rest spent in system management (doing tasks such as garbage collection). The results of the parallel task creation are seen in figure 6.6.

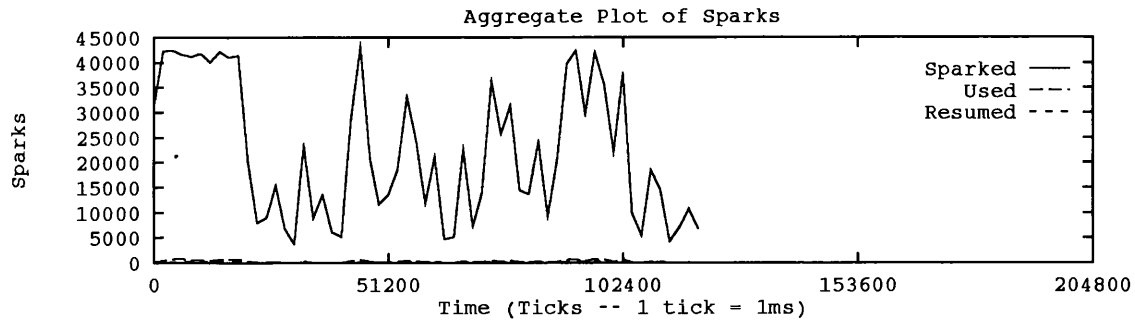


Figure 6.6: GRIP task creation chart for `fibTP` ***program***

The reports from GRIP show that thousands of new parallel tasks were created. Although the number of new tasks and the time spent creating these tasks was very high, the percentage of time evaluating the tasks was relatively low. Therefore, having too many small parallel tasks caused GRIP to spend a disproportionate amount of time in task creation thus leaving less time for task evaluation.

These results led to the next experiment, in which two functions were devised to create fewer parallel tasks. First:

```
onepar' f a b  =  par a (f a b)
```

which spawns a parallel task `a` and returns `f a b` as a result. Second:

```
onepar'' f a b  =  par b (f a b)
```

which spawns `b` as a parallel task and returns `f a b` as a result. Using just the `onepar'` function, a new version of fibonacci was written:

```
fibOP ::'Int -> Int
fibOP 0 = 1
fibOP 1 = 1
fibOP n = onepar' + n2 (fibOP (n-1))
          where
          n2 = fibOP (n-2)

main _ = show (fibOP 30)
```

The aggregated results for the processor activity from GRIP are seen in figure 6.7.

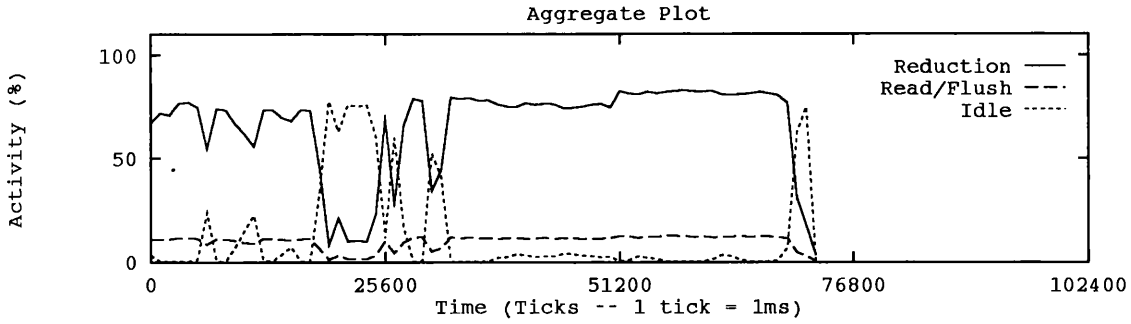


Figure 6.7: GRIP activity chart for `fibOP` ***program***

Figure 6.7 shows that the execution time was 73 seconds. Of this, 72% was spent evaluating, 14% was spent idling, and the rest spent in system management. The results of the parallel task creation are seen in figure 6.8.

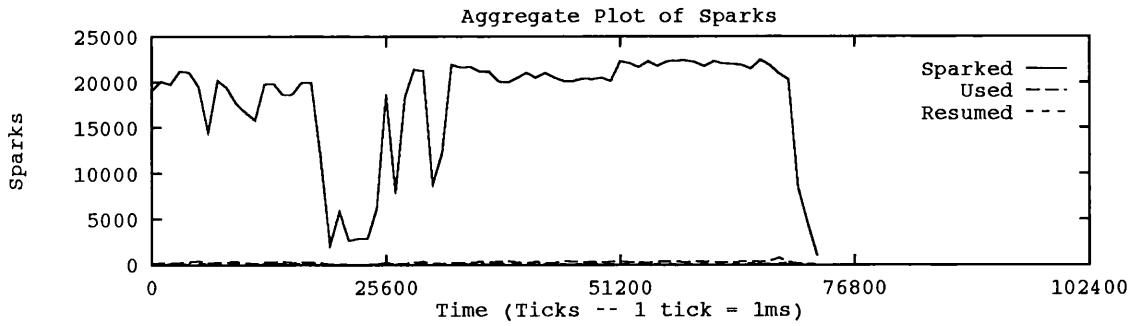


Figure 6.8: GRIP task creation chart for `fibOP` *program*

The results of the experiment using the `fibOP` definitions showed that less tasks were being created. The effect of this is that the `fibOP` program executed faster as a larger percentage of time was spent evaluating rather than creating tasks. A comparison of the results of the last two experiments can be seen in table 6.1.

Program	Total time	Evaluation time		Idle time		Other time	
	seconds	%age	seconds	%age	seconds	%age	seconds
`fibTP`	120	43	51.60	49	58.80	8	9.60
`fibOP`	73	72	52.56	14	10.22	14	10.22

Table 6.1: Comparison of speed between `fibTP` *and* `fibOP`

In table 6.1, attention is drawn to the number of seconds spent evaluating. The program that created more parallel tasks took 120 seconds to complete but spent only 51.60 seconds evaluating. However, the program that created less parallel tasks took 73 seconds to complete and spent 52.56 seconds evaluating. This shows what a large overhead creating tasks can be.

Once it was established that the parallel annotations had to be used with care, it was then possible to write some parallel versions of well known functions. For example, a parallel version of `map` can be written as:

```
pcons = onepar'' (:)


pmap f []     = []
pmap f (h:t) = (f h) `pcons` (pmap f t)
```

Using this definition, `(f h)` executes on the current processor with `pmap f t` being sent to the task pool for further parallel evaluation. This parallel version of map was put into a bigger test program. The results of this experiment were that the GRIP run-time system failed. The assumption was that the annotations had been used incorrectly. The actual problem (according to Kevin Hammond, the GRIP system administrator) was due to some bugs in the GRIP system garbage collector which were being tracked down at that time. According to Clack, one of the original designers of GRIP, this version of `pmap` causes *speculative* parallelism because every new task with a call to `pmap` causes yet another new task to be created. Any task placed in the task pool is a guarantee to the GRIP system that the task needs to be evaluated. Therefore, the concept of lazy evaluation does not apply to tasks, even though one may expect laziness in a system that evaluates lazy, functional languages. Clack states that the expression:

```
head (pmap id [1..])
```

will cause an infinite computation.

Although the fibonacci experiment highlights the pitfalls of annotations, it does not reveal much about the behaviour of large applications. Further medium-sized test programs were sent to GRIP, but these too failed to execute. During this research, the GRIP run-time system was being developed to use annotations and to utilize a different abstract machine from the one originally documented. This meant that GRIP was unstable at times.

6.3. Other Reported Experience

This section summarises other reported use of parallel identification techniques.

6.3.1. Use of Compiler Detetected Parallelism

Some early work on using compiler detetected parallelism through strictness analysis was presented by Goldberg in [Goldberg88]. This paper describes the Buckwheat system, which is a working implementation of a functional language on a commercially available multi-processor machine. By using strictness analysis at compile-time and various scheduling strategies at run-time, Goldberg showed impressive speed-ups as extra processors were added to the system.

A recent system that uses compiler detected parallelism is the DIGRESS system [Clack92]. DIGRESS is an architecture for executing parallel functional programs on a network of workstations. Each workstation has one (or more [33]) processing element, which communicates via a purpose built communications sub-system [Ghosh91]. DIGRESS is intended for coarse-grained parallelism and its expected use is for large functional applications. Because the load on workstations can vary dramatically and because DIGRESS does not expect sole use of the workstation, various strategies for run-time scheduling, load balancing, task size evaluation, and task partitioning have been devised. No results have been reported for DIGRESS, but a workload synthesizer and simulator have successfully utilized the communications sub-system.

Boyle and Harmer recently presented work which uses a functional language to harness parallelism on a CRAY X-MP vector processor [Boyle92]. The program was used to solve some problems using partial differential equations. The language they used for the program was pure LISP. The LISP program was automatically transformed into CRAY Fortran using the TAMPR transformation system. The TAMPR system used domain dependent, domain independent, and hardware dependent phases to produce the Fortran. The Fortran that was generated was not intended to be human readable, but was produced as a notation to inform the hardware how to perform. This is because the Fortran compiler produces very efficient vectorizing code on the CRAY. The results of Boyle and Harmer's work show that their functional program was faster than a hand coded Fortran program written to solve the same problem. This highlights how

[33] The reasons for running more than one processing element on a single processor workstation are discussed in Parrott's thesis [Parrott93].

compiler detected techniques can be beneficial for data parallelism.

6.3.2. Use of skeletons

Skeletons address the issue of mapping common algorithmic structures onto an underlying machine without the programmer having to know the details of that machine. The programmer may imagine that a set of composed functions, such as:

```
(f · g · h · i · j) data
```

could execute on a parallel machine, with each function on a separate processor creating a pipeline of functions with data flowing from one processor to another. In reality this depends on the complexity of the functions and the type of the data, but it does not stop the programmer thinking about the composition as a pipeline. Therefore, the programmer may imagine that each of the composed functions is placed on a different processor. This is seen in figure 6.9.

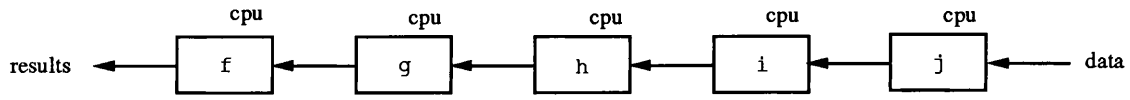


Figure 6.9: A pipeline of functions

Skeletons can address the expression of pipelines in an abstract way. The composed functions may need to be expressed as:

```
pipeline [f, g, h, i, j] data
```

to get the desired behaviour on a particular machine. In this way, the programmer can express the ideas in an abstract notation without relying on low-level annotations.

For each environment, the skeleton for `pipeline` may create code that is amenable to compiler detected parallelism or the skeleton may be written using the annotations for that environment. Therefore, the programmer is insulated from using the annotations. It may be that the compiler makes decisions about which actual skeleton code to use depending on the size of the functions and the type of the data.

This is possible even when the programmer uses the same skeleton in the same program. In the paper "Structured Parallel Functional Programming" [Darlington91], Darlington et. al. suggest that other common structures for programs (such as divide and conquer, meshes, lattices, and farms) are useful. They state that one of the advantages of using skeletons is that they may be transformed using standard functional programming transformation techniques. Using this mechanism, they demonstrate how to transform a program which uses one skeleton into an equivalent program which uses a different skeleton. The example shown converts a mergesort which uses a divide and conquer skeleton into a mergesort which uses a pipeline skeleton. This technique further enhances the power of parallel functional programming as using skeletons frees the programmer from the burden of understanding the underlying machine. These transformations can be done silently by the compiler and improve the performance of the program.

In [Cole90], Cole presents a skeleton for divide and conquer. He also defines the iterative combination skeleton, which combines elements in a set of objects if the elements are considered to be good partners. Each iteration over the set reduces the number of set members until there is just one member. Cole shows how this can be used to describe a minimum spanning tree algorithm. Cole also defines the cluster skeleton which, by his own admission, is a solution in search of a problem. This arose because it was designed from the hardware up. Once the cluster skeleton was designed, there were no obvious algorithms in which to use it.

In [Darlington91], skeletons are presented in which the aspects of process granularity, inter-connectivity of processes, and process placement are made explicit. These skeletons are low-level specifications of parallelism but are still more abstract than annotations because they can still express whole parallel structures. Darlington et al. state that a low-level skeleton may assume that each function supplied as an argument corresponds to a distinct process, and that each process may be allocated to adjacent processors with a single communication link between each stage. These low-level skeletons are an attempt at efficiency on certain machine architectures. However, they lose the flexibility of the more abstract skeletons, which just address structures of computation, and leave the choice of efficient execution techniques to the compiler. If

the programmer does not fully understand the underlying machine, he may use the low-level skeletons ineffectively.

Darlington et al. also present more abstract skeletons which can create as many processes as required. They observe that for many algorithms neither the inter-connection between the processes nor the placement of tasks can be determined at compile-time. They, too, present a set of skeletons including divide and conquer. Some skeletons, such as Kelly's Caliban notation [Kelly90], are used to express process networks which can then be parallelized. However ZAPP, which uses a divide and conquer strategy for parallelism, only has a divide and conquer skeleton [McBurnley90].

Most skeletons have been devised for creating process parallelism; that is, separate tasks execute concurrently to solve a problem. However, Jouret has suggested skeletons for data parallelism [Jouret91] which express parallel computation over large data sets, such that one operation is applied to every data item at once. Jouret shows the benefits of functional programming for data parallel computation and how his skeletons allow an abstract expression of this kind of parallelism.

If a system needs parallelism to be indicated by the programmer, then skeletons seem very suitable. They express abstract structures of computation which the programmer may already have in mind. Another benefit of skeletons is the ability to transform from one skeleton to another in order to achieve the most efficient implementation. However, as skeletons have only been produced for a few well known sets of solution strategies, when a new solution to a problem is found, there may be no suitable skeleton and, consequently, no parallelism may be harnessed.

6.3.3. Use of Annotations

In this section there is a discussion of the results from parallel functional systems that use annotations to harness parallelism.

Some of the earliest uses of annotations in parallel functional languages were seen in [Hudak85]. His paper addresses the issue of explicitly stating the mapping from program to machine using annotations that indicate on which processor to place a task.

This static mapping was considered uninteresting, so a function was devised which returned the current processor id and allowed the id to be manipulated in order to create new processor id's. This function could then be used in other functions in order to determine on which processor to place a newly created task and to create programs that could execute on machines where the processors were either tree structured or in a mesh. The annotations were used to create parallel versions of factorial and a matrix multiplication. However, the programmer has the burden of stating on which processor a task must execute.

Using annotations for more than just task creation is considered by Roe in [Roe89]. This paper presents a quicksort program which can be expressed as:

```
qsort []       = []
qsort (s:rest) = qsort [e | e<-rest, e<s]
                 ++ [s] ++
                 qsort [e | e<-rest, e>=s]
```

When successive changes are made to the quicksort program by adding annotations and rewriting sub-functions, different parallel behaviour is achieved. To achieve task partitioning, the same `par` annotation as the one used in the GRIP system is placed in the program. This produces parallel version of quicksort:

```
psort []       = []
psort (s:rest) = (par qlo . par qhi) (qlo ++ [s] ++ qhi)
                 where
                 qlo = psort [e | e<-rest, e<s]
                 qhi = psort [e | e<-rest, e>=s]
```

This parallel version loses the clarity of the original but is still recognizable as quicksort.

Roe observes that this version of quicksort will create parallel tasks which attempt to evaluate the expression `psort []`. The creation of these tasks causes inefficient execution. To avoid this, mechanisms to control the size of a task or to delay creating new tasks are presented. The size of a task is determined by counting the size of the list passed as an argument. If the list has *enough* elements, then tasks are created;

otherwise, the current task evaluates the list. To delay the creation of a task, some heuristics are suggested which are attributed to Hughes (but no reference is given). Using these heuristics, the following version of quicksort is presented:

```
qsort l = hsort l []


hsort []       l = []
hsort (s:rest) l = seq sl (slo ++ [s] ++ shi),            length l < k
                   where
                   slo = hsort [e | e<-rest, e<s]  (l++[shi])
                   shi = hsort [e | e<-rest, e>=s] []

                 = (par p . seq slo) (slo ++ [s] ++ shi), length l == k
                   where
                   (p:ps) = l
                   slo = hsort [e | e<-rest, e<s]  (ps++[shi])
                   shi = hsort [e | e<-rest, e>=s] []
```

This version is rather contrived, making the final result a program in which the clarity of the original and the essence of quicksort is lost [34]. It would be difficult for the average programmer to write programs in the resulting style on a regular basis. By writing functions such that their meaning is obscured, there is a good chance that the maintenance costs will be higher. Furthermore, it may be practical to spend this amount of time on a 2 line program, but not on a many thousand line program.

In his conclusions, Roe states that in order to achieve speed-up, a parallel program must make efficient use of a parallel machine and that in order for this efficiency to occur, parallel programs must explicitly control certain aspects of parallelism, notably task size and the re-evaluation of expressions. Although this may be true, it is debatable

---

[34] This program also uses the `seq` annotation, which is defined as:

```
seq a b = b
```

and evalutes `a` and then returns `b`.
Neither a definition for `k` nor an expression for the case when `length l > k` was given.

if programmer intervention and hand-placed annotations are the best method for achieving such efficiencies. Roe does observe that the embarrassing lack of empirical studies using real programs and data prevents one from identifying the real efficiency issues in parallel functional programming. However, to conduct empirical studies on these yet-to-be-written programs requires decent measuring tools, of which few exist.

Work done at the University of Nijmegen revolves around a technique called Communicating Functional Processes and the language Concurrent Clean, which is an intermediate language between functional languages and parallel machines [Eekelen89]. In [Eekelen90], the use of annotations is described and there is a discussion on parallel functional programming which is similar to that in [Roe89]. Eekelen observes that using annotations for parallel partitioning can create tasks which do little work and he suggests techniques that are similar to Roe's in order improve the task's workload. A method similar to Roe's is also devised for controlling the size of a task by limiting parallelism if the amount of data passed to a function is small. Eekelen also proposes a technique called interleaved processes, in which a process spawns a new task for execution on another processor, evaluates some expressions on the current processor, and then combines the results.

Eekelen presents two annotations. One is for creating a parallel task, such that:

```
PAR expr
```

causes `expr` to be evaluated in parallel. The other is for evaluating an expression on the current processor. This is achieved by:

```
SELF expr
```

By combining both annotations, interleaved processes can be created. Eekelen suggests that a construct, such as:

```
SELF expr1 `op` PAR expr2
```

will do the interleaving.

Although the annotations of Concurrent Clean and GRIP are different, there is an equivalence between them. Eekelen's interleaving construct can be written using GRIP

annotations, such that:

```
SELF a `f` PAR b = par b (f a b)
```

and:

```
PAR a `f` SELF b = par a (f a b)
```

These GRIP definitions may look similar because the first equivalence is the same as the definition `onepar''` and the second equivalence is the same as the definition `onepar'`.

In both Roe and Eekelen, there is a discussion on how annotations can be used to write skeletons such as divide and conquer or pipelines.

6.4. Review of Parallelism in Functional Programmming

Much of the work in parallel functional programming is experimentation with small parallel programs and little is being done with large parallel programs. I believe that, in general, parallelizing should be done on a macroscopic scale rather than on a microscopic scale. This is because machines are becoming smaller, faster, and cheaper with larger, faster, and cheaper memory and higher comms bandwidth (no one will really parallelize a program to sort 50 numbers on new machines; this is a hangover from the past).

Different institutions are using different languages which causes fragmentation of research. Furthermore, each institution has its own specialized hardware which merely exacerbates the situation. This means that any work done at one institution cannot be consolidated easily because the programs have to be rewritten either in a different language or with different annotations / skeletons. Haskell [Hudak88] is an attempt to address the language issue, but this is happening slowly and high quality compilers are only just appearing. There seems to be no common ground for the specification of annotations / skeletons. Having few machines available probably limits the growth of parallel functional programming. In addition, the consequence of relying on *specialized* hardware limits this growth even more (Vranken [Vranken90] reviews hardware for parallel functional programming).

Data parallelism through the use of vectorization is being successfully used in the Fortran world to get data parallelism easily. This has been seized upon by Boyle and Harmer, who have written functional programs which execute faster on a CRAY vector processor than hand written Fortran programs which do the same task. Functional programming for data parallelism has been addressed by Hill in his work on Data Parallel Haskell [Hill92], and by Jouret in his work on skeletons for data parallelism [Jouret91].

In this section there is a review of the investigation into parallel functional programming. Much of the review regards GRIP because it was on this system that most exposure was gained to parallel systems. The GRIP system is the only accessible parallel environment available for experimenting with functional programs, so the original designers at UCL and the current developers at Glasgow are to be congratulated for this acheivement.

6.4.1. Review of GRIP

The results of using the GRIP machine to experiment with annotations in various programs shows that:

(i) too many parallel tasks slow down the computation because too much time is spent managing tasks rather than evaluating tasks. For example:

```
f parallel_task parallel_task
```

does not seem to be an effective use of parallelism, whereas:

```
f local_task parallel_task
```

proves better. Therefore, more parallelism does not bring more speed.

(ii) annotations seem to be an ineffective way to harness parallelism. In the fibonacci experiment, it was possible to test various placements for annotations to get the best results, but in a large application this would not be feasible. Without a thorough understanding of the whole program and the environment in which parallelism is to be used, it is difficult to decide where the annotations for parallelism should be placed. As in other areas,

experience can be gained by building a body of knowledge. But for general use, a programmer should not need to be an expert in order to get parallelism.

(iii) without an understanding of the run-time system and how the annotations work, any annotations that are used to optimise some parallel performance could be undone by changes to the run-time system. That is, annotations are very specific to a machine and run-time system; they are not portable to other systems and may not be effective if a change is made to improve a feature of the run-time system. Therefore, annotations are only useful on one machine and one version of its run-time system.

From these simple yet enlightening experiments, the question has to be asked "what is the result of 6-7 years of research into implicit parallelism?". If annotations can go so wrong, why should any other technique be rejected without a thorough investigation? As stated, Peyton-Jones considers that strictness analysis is still the best way forward, but in the short term annotations are an easier way to harness parallelism. The conclusions drawn from this investigation are that, in order to use annotations effectively, one must be restricted to one machine and one run-time system. One must also learn the peculiarities of the annotations, because annotations are not necessarily portable to other machines. However, this machine dependence runs counter to one of the main arguments for using functional programming, namely that functional programs are independent of any machine architecture. Functions are a declaration of work to be done rather than a sequence of instructions for a machine. So if there is a machine independent program, why add machine specific annotations? On the evidence of both Goldberg's work on Buckwheat and the use of GRIP, the only way forward is to do further research into implicit parallelism by finding ways to improve task management techniques.

Relying on Purpose-Built Hardware

In the paper "Some Early Experiments on GRIP" [Hammond91b], Hammond and Peyton-Jones present a table of timings for the program nfib:

```
let
nfib n = if (n<2) then 1
         else
             let n1 = nfib (n-1)
             in
                par n1 (n1 + nfib (n-2) + 1)
in
    show(nfib 30)
```

In table 6.2, the speed of GRIP with various configurations of processors is compared with the speed of some Sun workstations.

Configuration	Time (in secs)	Speedup
Sun 3/50	76.3	1.00
Sun 3/60	59.6	1.28
Sun 3/260	47.9	1.59
GRIP (1 proc)	75.0	1.02
GRIP (3 procs)	27.3	2.79
GRIP (6 procs)	14.3	5.34

Table 6.2: The speed of GRIP compared with the speed of Sun workstations

The GRIP processing elements use the same Motorola MC68020 microprocessor as a Sun 3/50, which explains why a GRIP with 1 processor is about the same speed as a Sun 3/50. Hammond and Peyton-Jones observe that there is a near linear speed-up when using multiple processors on GRIP. They state that `nfib 30` does 2,692,537 function calls, which means a GRIP machine with 6 processors did about 188,000 function calls a second [35]. They also state that, with a hand-tuned version of nfib,

[35] This is evaluated by dividing the number of function calls by the total execution time, i.e. 2,692,537 / 14.3 = 188,289.3

they managed to speed up from 17 times using 20 processors. It can be calculated that the program took 4.41 seconds and that GRIP did 610,552 function calls a second. This may seem impressive for a multi-processor Motorola 68020 machine, but the latest workstations from Sun, such as the Sun 10 workstation, are 60 times faster than a Sun 3/50. If one were to extrapolate the figures to a Sun 10, one can calculate that the program would take 1.27 seconds to execute and that there would be 2,120,108 function calls per second. Therefore, the latest workstations, which cost about 6,000 pounds, could run programs faster than a 20 processor GRIP.

This highlights the problem of using purpose-built hardware as opposed to "off-the-shelf" technology. The techniques devised for GRIP are reliant on GRIP being available and working. If GRIP is not available, or fails to work, the experiments and investigation are seriously held up until a new machine arrives. This may take years if the machine is purpose-built. If the investigation into GRIP had used "off-the-shelf" technology, then the techniques could be moved over easily when newer and faster machines arrive.

The gestation period of GRIP has been so long that it has been superseded by a single workstation. This leads us to question whether the techniques discovered (and those yet to be discovered) are suitable for:

i) only GRIP

ii) single bus machines with multiple processors

iii) any multi-processor system

In the past, special purpose machines, for example Lisp machines, have been superseded by general purpose workstations as the workstations became increasingly faster. The lesson learned is that a number of large hardware manufacturers can build faster machines more quickly than a few small manufacturers.

In [Hammond91b], Hammond and Peyton-Jones discuss fine control of the machine. It seems some of their problems are related to the way the task pool is managed on GRIP. If a new task is created (sparks in GRIP-speak), then it is forcibly sent to a global task pool. This clearly has severe overheads if the new tasks are small,

and their paper discusses some of the issues regarding small tasks. For example, one scenario they consider is the evaluation of $E_1 + E_2$. Suppose the parent sparks E_1 and then discovers that E_2 is quick to evaluate or needs no evaluation. The parent then goes to evaluate E_1 but discovers that it is unable to because another processor is currently evaluating E_1 or that, even though no processor is evaluating E_1, there was a considerable cost in sending E_1 to the global task pool. In other systems, such as DIGRESS, when tasks are created they are sent to the local task pool, which has less overhead.

In the scenario of $E_1 + E_2$, if E_1 were made into a new task on DIGRESS, it would be placed in the local task pool at minimal cost. If E_2 were evaluated quickly, then E_1 would be available from the local task pool. E_1 would only be evaluated by another processor if, at the time E_2 was being evaluated, another processor had requested some work and E_1 happened to be the first task in the local task pool and the local processor was the heaviest loaded. Parrott observes that DIGRESS can be swamped with new tasks with little degradation in performance because new tasks go onto a local task pool [Parrott93].

If there needs to be more control of the machine,as Hammond and Peyton-Jones suggest, why does GRIP only provide two annotations, namely par and seq? This only allows the spawning of new processes. Why not devise a declarative framework for task management?

6.4.2. A Question of Maintenance

Annotations force the programmer to change his code to indicate where new parallel tasks should be. Under the UNIX system, the programmer has no say in matters which were (and still are), in some systems, considered essential for the programmer to control. This has not been detrimental to the effectiveness of UNIX as an operating system. Some (usually mainframe) systems require the programmer to state how much memory their program will use, how much I/O it will perform, and how long it will run. If these limits are exceeded, the program stops. UNIX has clearly demonstrated that it is not essential to specify these limits, rather, it is desirable from the programmer's point

of view not to specify them. Better systems, not programmer intervention, will solve the issue of the effective harnessing of parallelism.

One of the biggest issues in the software industry at present is that of maintenance and debugging. As programs get larger and more complex the cost of maintenance rises sharply. Functional programming is of benefit here as programs can genuinely be built as machine independent black boxes, whereby the complexity of the source code can be reduced and therefore maintenance can be made simpler. If the complexity of a program is increased by adding parallel annotations, the maintenance task will not become easier but more difficult and costly. I believe that using a language where there are no annotations and where the code is more readable is beneficial.

In the long term, I think one should contrast / weigh-up the cost of maintenance with the cost of absolute speed. That is, if one adds annotations to a program in order to make it more efficient on a parallel machine, what is the cost of adding these annotations and what is the cost of the extra maintenance? Will the extra speed-up be worth the extra incurred costs? Is it cheaper to buy a faster machine? If optimum performance is not the ultimate goal, then there is not the cost of adding the efficiency enabling annotations and the maintenance costs are lower. How much slower is a program compiled with strictness analysis than a program with annotations? Is the difference worth the extra cost of maintenance? The elegance and correctness of functional programs could outweigh any run-time difference.

6.4.3. Advantages and Disadvantages of Compiler Detected Parallelism

The advantages of compiler detected parallelism are:

i) code is portable as there are no environment specific dependencies

ii) no changes to the code are required and so there is no human intervention in the parallelizing process.

iii) parallelism is found automatically

iv) parallelism is dependent neither on the programmer knowing fixed parallel structures (as in skeletons) nor on the programmer thinking he knows where the parallelism is (as in annotations)

The disadvantages are:

i) compiler detected parallelism through strictness analysis can be very slow

ii) the resulting program may not utilize the machine as much as programs with hand written annotations in which the programmer may know a significant amount about the parallel properties of the algorithm. With compiler detected techniques, the programmer is unable to specify where parallelism should occur

Implicit parallelism using compiler detected techniques and task management can be beneficial as shown by Goldberg in the Buckwheat experiments. However, it is not clear if compiler detected techniques alone are sufficient.

6.4.4. Advantages and Disadvantages of Skeletons

Using skeletons, general structures of computation can be specified which cause tasks to evaluate in parallel. It has been seen that parallelism has to be designed into a system because parallelism is not easy to harness in all algorithms due to data dependencies. Skeletons allow the programmer to express where parallelism occurs in an algorithm through abstract ideas such as pipelines, lattices, divide and conquer, etc. The skeletons are compiled into efficient code depending on the actual type of hardware used.

Skeletons are amenable to automatic program transformation techniques which can convert one skeleton type into another. This allows the programmer to specify a skeleton type and to have the skeleton transformed into one that can be executed efficiently on a machine. Some skeletons are independent of any machine environment while others are tied to a machine such that they become non-portable. Some researchers use skeletons that make explicit the issues of process granularity, inter-connectivity, and process placement [Darlington91]. Their technique is more abstract

than annotations but still too low-level for the applications programmer. Presenting the programmer with high-level skeletons which hide low-level features is required in order to provide machine independent parallelism. These high-level skeletons are portable because they can be written in terms of low-level skeletons or annotations on each different machine. It is important to remember that the programmer may not fully understand the workings of the underlying machine. If he is to program using low-level skeletons, then he may make mistakes which cause poor behaviour from the program.

Skeletons exist for only a few well known algorithmic structures. Consider the graph manipulation functions presented in chapter 4. As there is no standard functional solution to creating and visiting cyclic graphs, there is no skeleton by which graph manipulation algorithms can be parallelized. Are we to tell functional programmers who have large graphs that they cannot have parallelism? Furthermore, as the graph manipulation techniques are complex and have not yet been fully analysed, the source of parallelism is not yet obvious. Therefore, implicit techniques for harnessing parallelism still seem the best way forward for many algorithms.

Skeletons have the disadvantage that changes have to be made to the source code in order for parallelism to be harnessed, unless they are used from the design stage.

6.4.5. Advantages and Disadvantages of Annotations

Although the annotations of Nijmegen are different from those on the GRIP system, both systems use annotations that have basically the same attributes.

The advantages of using hand-coded annotations are:

i) it is possible to get effective parallelism by matching the granularity of the tasks with the granularity of the machine.

ii) it is possible to acquire a deeper understanding of how an algorithm parallelizes by experimenting with the placement of annotations and analysing the run-time behaviour in detail.

The disadvantages of annotations are:

i) they are not portable; each machine/system may have different annotations. The annotations will be written for only one machine and may have to be rewritten when porting to a different machine. This is because annotations are associated with some underlying environment, for example GRIP. Therefore, a program which has the parallel annotations of GRIP will not run in a different parallel environment, such as the one suggested by Nijmegen, and vice-versa, unless all the annotations are changed.

ii) they can cause the program to have poor parallel behaviour; the programmer may misunderstand how they work or not understand enough in order to use them correctly. The programmer needs to understand the underlying machine, the run-time system, and how the program will be evaluated by that system in order for the annotations to be effective. The model of functioning programming and the rhetoric behind it generally discourage this knowledge as it is meant to be unnecessary.

iii) their effectiveness may be reduced if the abstract machine implementation changes.

iv) they go against the grain of functional programming because they (a) tie you to a particular machine and (b) involve changing the source code.

It is assumed that the programmer (a) knows where the parallelism is, and (b) knows how to add annotations to harness the parallelism. For there to be effective parallelism using annotations, both (a) and (b) have to be true. There is no evidence that one follows the other. If the programmer does not know where the parallelism is, then one must consider how effective he will be at accurately placing annotations in the program. For small programs it may be easier to have a full understanding of the whole system, but for large programs where there may be only minimal knowledge of the system, and placement could be difficult. If a small set of hand placed annotations seem to harness enough parallelism, why not place them automatically after a small amount of analysis?

Peyton-Jones has commented that if one uses annotations in a program and then transforms this program, then the transformer doesn't know what to do with the

annotations. I believe the use of annotations together with program transformation is contradictory. One would only use annotations for explicit speed-up on a certain machine. If a program transformer could transform away any usefulness of the annotations, as Peyton-Jones suggests, then the use of annotations and program transformation together is of no value. Peyton-Jones recent idea is to have special combinators to affect run-time behaviour [36]. Irrespective of the advantages and disadvantages of annotations, it does not seem beneficial to hand place annotations in a many-thousand line program. Therefore, annotations may be effective for understanding the nature of how a certain reduction machine's run-time system behaves with small programs, but they are not effective for large programs. Thus, a big gap exists between the use of annotations for harnessing parallelism and the requirements of application programmers.

The techniques of compiler detected parallelism, skeletons, and annotations have the same goal but use different approaches. In the parallel functional programming world at present just one technique is chosen for harnessing parallelism. It may be that future systems use combinations of them, and so there seems no need to reject any of them as unsuitable. Figure 6.10 shows how these three techniques are related.

6.5. Parallel Applications

Parallel computers are now in regular use at Caltech (The California Institute of Technology) for several major scientific calculations [Fox89]. This section contains a list of the types of applications undertaken at Caltech – first, because they are typical of the sorts of problems that are run on a parallel machine and second, to contrast with the programs that have been run and reported on parallel, graph reduction machines. The types of applications reported at Caltech are:

- lattice monte carlo simulations

---

[36] Personal communication

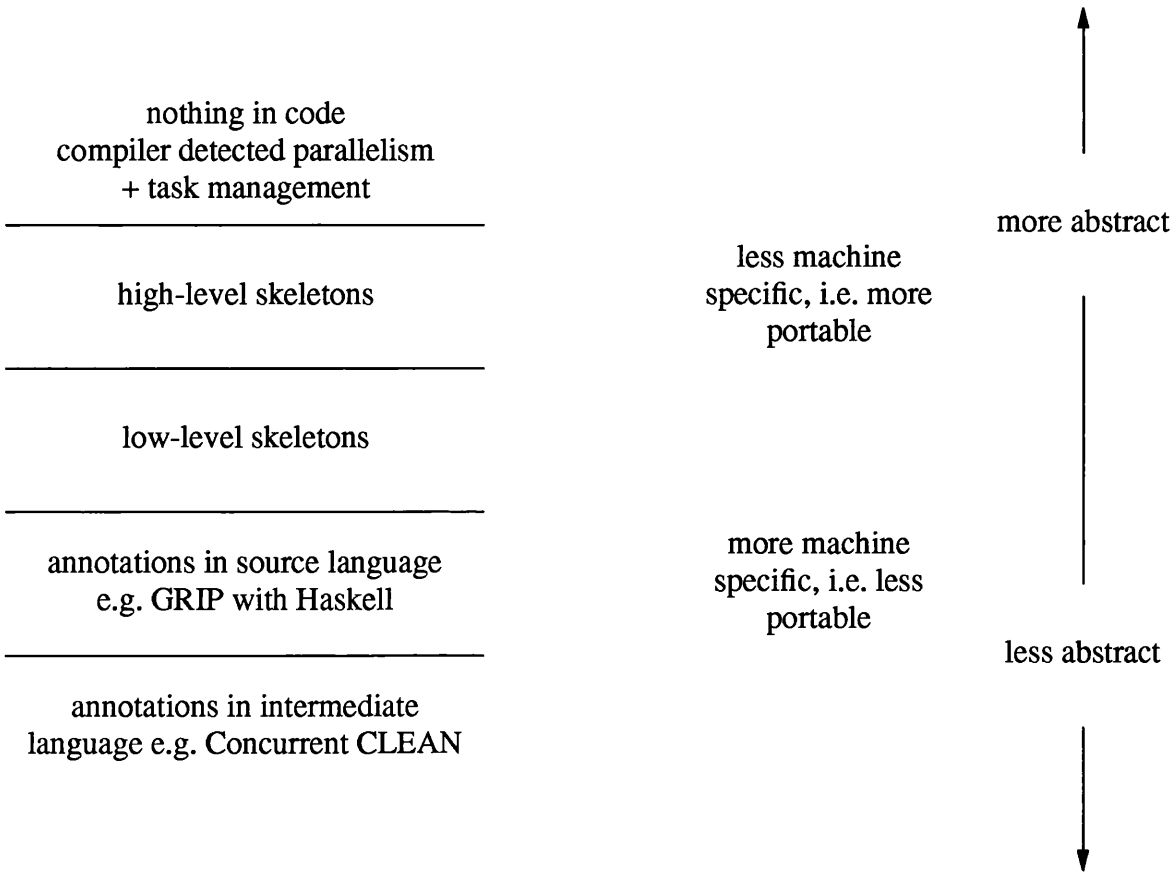


Figure 6.10: Layers of abstraction in parallel functional code

- subatomic string dynamics
- high T_c superconductivity
- exchange energies in He^3 at a temperature of 0.1mK
- astrophysical partical dynamics
- astronomical data analysis
- quantum chemistry reaction dynamics
- grain dynamics by lattice gas techniques
- computer chess
- ray tracing in computer graphics

- kalman filters
- plasma physics

Work on adding annotations to a `quicksort` or `fibonacci` program will not impress the people that need high volume parallel machines. It is clear that the implementors of programs on parallel graph reduction machines are not addressing the issues that need to be addressed in order to render these machines acceptable in parallel programming environments.

Boyle and Harmer are unique in reporting a large parallel application written in a functional language [Boyle92]. Their application to solve partial differential equations ran faster than a hand-coded version written in Fortran. For parallel functional programming to advance, more serious parallel applications have to be written. A good start may be a high T_c superconductivity program, which Fox says is *embarrassingly parallel*. Surely, parallel functional programming techniques can do well here.

6.6. Summary

There seems to be little evidence that compiler detected parallelism and task management does not work even though it is accepted that the strictness analysis can be very slow and that task management may not utilise the machine to its fullest capacity when compared with handcrafted annotations. However, this approach is appealing for large applications which may need to be portable because the programmer does not need to know anything about the underlying machine. Furthermore, it seems contradictory to have a high level declarative language and then add low-level machine specific annotations. It has been seen that both Boyle and Harmer and Goldberg successfully used compiler detected parallelism.

If the programmer needs or wishes to intervene, skeletons seem a better choice than annotations because skeletons allow the programmer to express structures within a program without knowing much about the underlying system. Skeletons can be made portable by writing the code for them on each parallel machine. Skeletons have an advantage over strictness analysis in that the programmer can say where some

parallelism occurs, which is not possible with strictness analysis. Annotations seem the worst approach to harnessing parallelism from the view point of the large applications programmer. They are specific to each machine and, therefore, not portable. They must be hand placed and the programmer must understand the specifics of the behaviour of each placed annotation. The advantages of annotations are that they can be manipulated to obtain high performance from a parallel machine. As stated, this is fine for short test programs but unsuitable for large applications.

This thesis proposes that strictness analysis at compile time and task management at run time is the most appealing solution in general. As machines get faster and cheaper and memory gets larger and cheaper, the cost of processor time for task management will become insignificant. Researching strategies for task management seems to be the way forward [Parrott93]. Furthermore, it is my belief that the programmer should accept a bit of inefficiency in the system and not try to manipulate the program until it is perfect. Fox observes that parallelism is easier to harness when a problem has a regular decomposition; Boyle and Harmer show that this is even true for functional languages. However, for problems with an irregular decomposition he notes that the efficiency of parallel machines is low, with an N-CUBE achieving 50% efficiency and a CRAY only 5% efficiency. He states that, on average, the CRAY X-MP at Caltech achieves an efficiency of just 12% for all problems. He concludes that there is too high of an expectation of efficiency on parallel machines. As it is clear that no parallel technology is achieving near-100% efficiency, the benefits of parallel functional programming can be evaluated without expecting a linear speed-up for all applications.

Limiting factors in the consolidation of research into parallel functional programming have been:

i) the reliance on special hardware

ii) the lack of proliferation of the special hardware

iii) the lack of tools for generalized hardware

To overcome these problems, systems such as DIGRESS have been designed. DIGRESS is an experimental system for running functional programs in parallel on a network of workstations (although this is not the only architecture). With DIGRESS,

more exposure to writing and executing parallel functional programs can be obtained.

The criticisms in this chapter are based on the usability of parallelism harnessing techniques from the view point of the applications programmer. The recommendation for further work is that more effort needs to be expended on skeletons and strictness analysis, which both retain the high level properties of programs, rather than fiddling with annotations, and that more real applications be used as test cases when testing the techniques for parallel functional programming.

It is interesting to note that in [Hudak84], Hudak states that DAPS (Distributed Applicative Processing System) is aimed at AI systems. Hudak states that AI programs do not execute efficiently on super-computers and DAPS would be tailored especially for AI programs. No generally available working system has appeared. It is clear that a parallel version of OPS5 could not have been executed for this thesis, irrespective of the kind of matcher used. The only parallel machine available, GRIP, would not successfully execute much more than a simple test program. Although Stolfo [Stolfo86] and Rosenthal [Rosenthal85] both concluded that implicit parallelism is promising for obtaining more parallelism in a rule-based system than the previously used approaches, obtaining such parallelism in a functional environment is not yet feasible. Until techniques for harnessing parallelism in large, functional programs have been developed and tested it is not possible to determine how best to parallelize a functional version of OPS5.

In the report [Johnsson90], John Hughes asked the question: "Implementors, what analysis would you want from us analysis designers to make your parallel functional implementation run faster?" After this, Johnsson prints a list of items requested for analysers. Clearly the time is right for the implementors to take their turn and ask application builders: "What features do you want in a language or system in order to harness parallelism in your application?"

Chapter 7

7. Conclusions

In this chapter the discoveries of this work are presented together with a review of the original goals of the research and a summary of the research issues. Suggestions for continuing this research are contained in the section on further work.

At the start of this research it seemed that the need for parallelism in rule-based systems could be met by the apparent suitability of functional languages for harnessing parallelism. However, this research has indicated that this need cannot be met at present. The conclusion of the work in this thesis is that:

> *There are no fundamental limitations that prevent functional programming from being used for large applications such as rule-based systems. However, the environment for building and executing functional programs needs to be improved in order to address the limitations imposed by the immaturity of current functional programming environments.*

The contributions of this thesis are:

- a critical assessment of the suitability of functional programming techniques for implementing large applications and rule-based systems in particular.
- a critical assessment of practical state manipulation techniques in functional programming.
- a large, working, application written in a lazy, higher-order functional programming language which does large amounts of state manipulation

- a critical assessment of the functional programming environment, with suggestions for how the environment can improve.
- the design, implementation and analysis of a tool for profiling lazy, higher-order functional programs. The tool measures function call count, time spent in a function, and the heap space used by a function.
- a critical assessment of techniques for parallelizing large functional programs.

This thesis has discovered that:

i) it is a non-trivial task to design and develop a rule-based system in a functional language because of the requirements for state manipulation, input and output, sequencing of operations, and complex data structures and algorithms. Techniques had to be specially devised for the functional rule-based system in order to deal with these specific requirements. This differs from an imperative environment which has some of these techniques already built-in. The power and flexibility of the functional approach allows the design and implementation of these techniques to be approached in an organised and modular fashion.

ii) some aspects of functional languages and their associated environments are not always suited to large applications. Certain aspects are difficult to do in a functional language, such as representing data structures such as graphs or doing input and output from deep in an application. Other aspects require either support from the functional language, such as a vector data type needed to execute a rule-based system efficiently, or better interaction with the operating system.

iii) the lack of measurement tools is a hindrance for the developer of large applications. Without these tools it is impossible to observe or verify the behaviour of algorithms and programs. Furthermore, the lack of debugging tools makes it impossible to fix a range of bugs which occur inside large programs.

iv) the current facilities for executing functional programs in parallel environments are not effective for large applications. The use of hand-coded annotations may be fine for small programs but it is unsuitable for large programs. Furthermore, there is a lack of parallel functional machines on which programs can be executed.

7.1. Review of the Goals of the Research

In the introduction it was stated that when the 5 original aims have been addressed it will be possible to determine if functional programming techniques are suitable for harnessing parallelism in rule-based systems. In this section, these original aims are reviewed in the light of the discoveries of this research.

Goal (i)

To use functional programming techniques to implement a rule-based system.

In chapter 3 the design and implementation of a rule-based system was discussed, and it can be concluded that a functional language can be used to implement such an application. The power and expressiveness of functional programming is an aid in the development of large applications. The ability to build abstractions and to use higher-order functions is a benefit to the programmer.

A compiler for the OPS5 language was built using a framework of higher-order functions that closely represents a formal grammar. With this framework, a parser for any LL(1) grammar can be built and such a parser was built for OPS5 in this research. The framework has also had extensive use in other applications.

An algorithm which is comprised of a description as an ordered list of statements can be converted into a functional algorithm by converting each item of the description into its own function. These functions are given their own data type and then combined in a pipeline to form an algorithm expressed in a functional style.

The implicit state manipulation in imperative languages has the advantage that it can be undertaken with relative ease. This is not the case in functional languages. The

advantage of state in functional languages is that *state must be represented explicitly and therefore the code must be designed.* As there is explicit control over which parts of the state are passed and accessed, the negative issues of implicit state manipulation and generally accessible global store are overcome. The disadvantage of state in functional languages is that *state must be represented explicitly and therefore the code must be designed* to accommodate it. As all state is explicit, the program code can look messy if an inappropriate implementation technique is chosen. Imperative programs look much the same when state is added because the state manipulation is implicit.

The requirement to store large amounts of state in a functional application can be achieved by using an abstract data type for the state object. Access and update functions are defined, which are of type *State* $\rightarrow$ *State*. These are combined in a pipeline to facilitate state manipulation throughout the application. A top-level function can control the application of each *State* $\rightarrow$ *State* function to get the desired behaviour from the application. The misconception that functional languages are unable to deal with state is held by many imperative programmers; the state manipulation undertaken in this application is enough to prove them wrong.

The manipulation of both input and output has to be done with care. It is possible to write programs that hold onto all of the output until the end of a program run. This behaviour can be perturbing to the user, who would expect output to occur gradually. However, the semantics of the program remain correct.

Goal (ii)

To analyse the functional rule-based system for inefficiencies and to then implement efficient new algorithms or to transform old algorithms into more efficient ones.

In chapter 4 it was seen that the tools available for analysing the behaviour of functional algorithms and programs were non-existent. Most functional environments report the behaviour of a program as the number of reductions and the number of cells used. This tells the programmer very little about the real behaviour of a program [37].

[37] This information is equivalent to driving a car that has no dashboard equipment. At the end of a journey the car reports that there were 487,000 engine revolutions and that 690 litres of

Furthermore, the number of reductions and the number of cells used differs on each abstract machine.

To address the need for a tool to analyse functional programs, a profiler was designed and built. This profiler measures the number of calls to a function, the amount of time spent in a function, and the number of cells used by a function. The profiler is aimed at application programmers rather than abstract machine builders and, consequently, the results presented are amenable to the programmer. The results are reported with respect to the lexical scope of the program rather than some run-time representation. This new technique is called *lexical profiling*.

In chapter 5 both the design, implementation, and usefulness of the lexical profiler were presented. In order for results to be associated with the lexical scope of a program, it is necessary for both the functional language compiler and the run-time system to be modified. The compiler colours a representation of the program to attribute lexical scope and the run-time system collects data continuously throughout execution. The collected data can be reported to the programmer as execution occurs.

The benefits of tools that allow program behaviour to be monitored was also shown in chapter 5. The lexical profiler makes it possible to ascertain (i) if functions are inefficient, as seen in the nqueens program; (ii) if functions have space behaviour problems, as seen in the database program; (iii) if functions have strictness problems, as seen in the foldr program; and (iv) if one function is more efficient than another, as seen in the sum of squares programs.

Due to the limits of the functional compilers available during this research, it was not possible to profile the functional rule-based system. The Haskell compiler used in this research, which was the original Glasgow Haskell compiler, was not able to compile such a large application. It was too slow, used too much heap space, and had quite a few bugs [38]. The UCL experimental reducer uses FLIC as its input language,

---

exhaust fumes were expelled. As most people are aware, the behaviour of the car needs to be fed to the driver in units the driver understands and at continuous intervals in order for the driver to gain a useful assessment of the car's current performance.

[38] The original Glasgow Haskell compiler sometimes had degenerate behaviour. For example, it once took 45 minutes to compile a 243 character program on a Sun 3.

however newer, more robust, Haskell compilers cannot be used with this reducer. The new Glasgow Haskell compiler does not produce FLIC, and the Chalmers Haskell B compiler produces illegal FLIC.

An investigation into program transformation tools for use by application programmers was never undertaken as the need never arose.

Goal (iii)

To create a version of the functional rule-based system that is amenable to execution on a parallel machine.

In chapter 2 an analysis of matching algorithms in rule-based systems was presented. This showed how Rete is a good algorithm for both sequential and parallel rule-based systems. It was clear that matching algorithms that saved state were far more efficient than those that did not. However, the functional rule-based system was implemented using a non state-saving algorithm because it was necessary to determine if a large, functional application could manipulate the large amounts of state required irrespective of the extra state required in the Rete matcher. As was shown in chapter 3, the functional rule-based system was a success.

To write a version of Rete requires the manipulation of graphs that have state saving nodes. In Rete, each production is converted into a graph representation, with attribute pairs in each condition being converted to nodes having different behaviour. Some nodes do simple tests, some do variable instantiation tests, but most significantly for this thesis, some nodes are state-saving. Chapter 4 showed that it is possible to create and visit graphs in a functional language. However, this was a non-trivial problem to solve. Although it is possible to create and visit graphs and to manipulate state in a functional language, the ability to have graphs with state-saving nodes is a requirement of the Rete matcher. The non-existence of such generally available algorithms limits the development of a functional Rete. The development of graphs with state-saving nodes is still outstanding and is, therefore, an area for further research.

The use of state-saving algorithms and data structures is an issue that has not been addressed within a state manipulation framework. The research in this thesis found

effective ways to manipulate state. However, this was only used as a framework for *State* → *State* functions. A functional Rete would require a mechanism where individual state-saving nodes could be updated. Although the current framework does not allow this, the use of linear types [Wadler90a] may be of benefit. When using linear types the programmer specifies which values have single-threaded access through a program. This allows the run-time system to do in-place updates because a value is guaranteed not to be needed by other functions. The use of linear types can be investigated as further research, however, as most functional environments do not have linear types their applicability may be limited.

Once the required data structures for a functional Rete can be built, it will be possible to determine if a state-saving algorithm in a functional language is effective. Although the functional version may not have the efficiency of updatable store as in the imperative version, it will be possible to observe the algorithmic improvements of state-saving over non state-saving algorithms. Even if it were possible to build a version of Rete *today*, the facilities and techniques available for running it in parallel are not suitable. In chapter 6, the facilities and techniques presented were more suitable for small programs rather than large applications.

It can be concluded that parallel functional programming environments are not quite ready to execute applications such as a rule-based system. This research has highlighted this and shown issues that need to be addressed in order for functional programming to be usable on a day-to-day basis for parallel applications.

Goal (iv)

To analyse the functional parallel environment to gather data on the performance of the parallel functional rule-based system in order to remove any inefficiencies.

As no parallel version of the functional rule-based system was created and executed, it was not possible to analyse one. To do such an analysis requires suitable tools. In chapter 6, the results presented by GRIP were shown. These results indicate to the programmer the effect of his program on the machine. The amount of cpu busy / idle time is presented, together with the number of new tasks created. However, there is

no information on either which functions were executed or which functions created the tasks.

The results presented by GRIP are more useful to the programmer who understands what the underlying machine is doing. However, functional languages are independent of any execution environment, therefore it is reasonable for a programmer not to have such an understanding. It is beneficial to have reports that the programmer can understand. Having lexical profiling on a parallel machine would be an additional aid for the programmer. To address this need, the DIGRESS project has implemented lexical profiling on its parallel environment, but reports from executing programs are unavailable at present.

It is clear that the reports from a parallel environment need to present more facets of execution than for programs on a sequential machine, but they still need to direct the programmer to the cause of the observed behaviour.

Goal (v)

To compare the performance of the parallel functional rule-based system with an existing parallel rule-based system.

The research never came this far. However, to determine relative performance requires comparison of like with like. The behavioral indicators for a parallel rule-based system can be collated from the work reported for these applications. It is not clear which of these indicators can be retrieved from a functional rule-based system in order for the comparison to occur.

7.2. Summary of the Research Issues

This section reviews the main research areas investigated for this thesis.

Developing Functional Applications

The majority of research in the functional programming arena is aimed at the theoretical aspects of functional programming and the implementation of abstract machines rather than developing applications. Although the theoretical research is not misplaced, this thesis proposes that by focusing on the practical issues of functional programming and attempting to proliferate the technology via general purpose programming, the required development and, hence, maturity will be forthcoming.

Functional programming has been around for a shorter time than imperative programming and the difference in the number of man-hours devoted to providing / discovering well known solutions in each is apparent. In functional programming there are few well known solutions to problems that are considered non-difficult for imperative programmers. For example, the algorithms for creating and visiting graphs need to be interpreted in a more abstract way than the traditional, imperative description.

It has been seen that there are some efficiency issues that need to be addressed in functional languages. The efficiencies of having O(1) access to data structures in a rule-based system can not be overcome by using parallelism. This thesis proposes that vectors can be added to functional languages without compromising the integrity of the functional model.

Recent work in functional programming has addressed the issues of high quality compilers, such as the Haskell B compiler from Chalmers [Augustsson92a] and the Glasgow Haskell compiler written in Haskell [Hall92]. It is promising that work is being undertaken in the areas of state manipulation [Hudak93], input and output [Achten92], and sequencing [Hall92].

Functional Programming Environments

These need to be improved. There are too few interactions with the operating system. This means that functional languages cannot be effectively used for general purpose programming because the speed of the current interactions are too slow.

Measurement

By implementing a large application in a functional language it has become apparent that the support tools needed for such large undertakings are not available. In order to overcome this, a technique called lexical profiling was designed and implemented so that higher-order, lazy functional languages could be measured. A tool has been built that presents the number of calls to a function, the time spent in a function, and the heap space used by a function. This technique not only gives more information but also is more accurate than the traditional, well known imperative profiler "gprof". Other approaches to profiling have appeared recently, and they too address the lack of tools for programmers. The availability of measurement tools will allow functional programmers to observe and verify the behaviour of their programs.

Parallelism and Functional Programming

Much of the research in this area uses small test programs, such as nqueens and factorial, as reference cases rather than large applications, which is where parallelism is really needed. Fox [Fox89] has shown the types of applications being parallelized at Caltech and has dubbed these applications "grand challenge" problems. Research into parallel functional programming must be directed at these "grand challenge" problems in order to prove its effectiveness. Boyle and Harmer have successfully shown that functional languages are capable of addressing the real needs of parallel systems, as their functional version of a partial differential equation program executed faster than a hand-coded Fortran program to do the same job [Boyle92].

Current arguments that functional programs execute too slowly is only relative to today's hardware and compilation techniques. The fate of specialized LISP machines was sealed when general workstations outperformed them in terms of cost and performance within the space of a few years. This raises the question of the benefit of building specialized hardware for executing functional programs in parallel. In chapter 6, it was seen that a Sun 10 workstation is faster than GRIP with 20 processors. Furthermore, the bandwidth of current networking technology for workstations is 100Mb/s when using FDDI. The combination of new workstations and new networking

technology can provide an environment with computing power many times greater than GRIP. Future hardware will make an even bigger difference to execution times, even for functional programs. [Article91] reports that AT&T scientists managed to get a laser chip to pulse at a rate of *600 femtoseconds*. Future networking technology is aimed at gigabit bandwidths. When machines are built out of devices that are this fast, the execution speed of functional programs will not be an issue. Therefore, it would be better to direct effort at executing functional programs efficiently on existing hardware and in parallel on networks of general purpose machines so that the techniques are transferable to new machines when these machines become available.

Work in this area has begun with the DIGRESS project [Clack92], which aims to make parallel abstract machine technology more generally available without requiring special purpose hardware but, instead, using networks of workstations. The promise of this approach has led to the commercial development of the technologies required.

The observation that parallel functional programs are unable to use a parallel machine efficiently now seems to be an irrelevant diversion. Fox observes that a Cray at Caltech only achieves 5% efficiency on irregular problems, and 12% efficiency on average. Having collated data from many experiences of parallel systems in 12 application domains, he states that, in general, there is too high an expectation of efficiency in parallel programs. In light of this, parallel functional programming may yet flourish.

Parallel Rule-Based Systems

Although Rete is considered a good matching algorithm for both sequential and parallel rule-based systems, it may be difficult to attain its efficiencies in a functional language because Rete has many pragmatic design decisions. It might be more suitable to implement a matcher with a more theoretical basis in a functional language than the Rete matcher.

7.3. Further Work

There needs to be a large body of solutions to well known algorithms written in functional languages. This can only come about by implementing the algorithms and it is pertinent to suggest that this work starts soon. This thesis found both problems and solutions in data structures and algorithms that were specifically suited to rule-based systems. Further work will reveal solutions to problems that are more general in nature. For example, there was a need for graphs with state saving nodes. Although these are needed for the Rete algorithm, they may also be of use in many other algorithms.

State manipulation, input and output, and sequencing need to be addressed in the future. They are essential for large applications, and this has been recognized. This research presented mechanisms for doing these three issues, however further work can extend these ideas.

Further work needs to be done to improve functional language interaction with the operating system in order for functional programming to be of use for a wider range of applications. At present there is just simple input and output to file streams. An operating system supports much more than file I/O, however it could be difficult to integrate all of the operating system functions into a functional environment. Those functions that are selected for inclusion into the functional environment need to be implemented as efficiently as possible.

By addressing all of these areas, functional languages could then be realisticly used for general purpose programming. To aid functional programmers, I look forward to the day when someone writes a book, using a higher-order, lazy functional language, as an equivalent of Knuth's "The Fundamentals of Computer Algorithms" [Knuth68] or Kernighan and Plauger's "The Elements of Programming Style" [Kernighan78], or when someone writes "Numerical Recipes in Haskell" to join "Numerical Recipes in C" [Press88].

The lexical profiler designed and implemented for this thesis executed on an experimental reducer which uses the FLIC language as its input. Runciman and Wakeling make changes to the Chalmers Lazy ML compiler to accommodate their heap profiler. For more general use and usability, the lexical profiler needs to be included in a

Haskell compiler and into various run-time systems. Furthermore, the space usage results need to be extended in order to report in as much detail as the Runciman and Wakeling heap profiler; that is, to report each constructor separately. This is impossible in the current FLIC implementation where this information has been lost, but in a Haskell implementation this information can be collected. The reports given by the lexical profiler are based on the inheritance profiling style. For statistical profiling, the results have to be post-processed. Further work is to design and implement this post-processor.

Debugging tools are still lacking in the functional programming world, although some suggestions for their implementation are now being made [Nilsson92]. Functional programs tend to be more bug free than their imperative counterparts because the computations are expressed at a higher level and because the strong type system forces programs to be type correct. However, bugs do still occur and further work can provide the required debugging tools.

Further work in the area of parallelism and functional programming needs to be directed at techniques for harnessing parallelism that are amenable to the builder of large applications. Parallel functional environments need to have greater availability, and this can be achieved through implementing such environments on networks of general purpose machines. Once these parallel environments are available, they need to produce reports of activity that allow the programmer to analyse the parallel behaviour. Further work in this area is to design reporting and analysis tools that are independent of the underlying machine yet reflect some common parallel environment.

Once all this further work has been done, it will be possible to design and build a parallel rule-based system. Further work in this area is to investigate the new Match Box algorithm [Perlin89]. Match Box, which has been specifically designed for matching in a parallel environment and has a formal basis, could be implemented in a functional language easier than other more pragmatic matching algorithms such as Rete because functional programming is amenable to implementing formal algorithms.

Appendix A

Database Profiling Data

This appendix shows call-count and time data gathered from a run of the functional database program where *every* function in the program was profiled. As every function is profiled, no function is inherited by another.

In	From	No of Calls	Time in seconds
main		1	0.00
showtable	main	1	0.42
join	main	1	0.04
table1	main	1	0.32
table2	main	1	0.48
space	ljustify	35	0.04
copy	space	35	1.50
take	copy take	35 206	1.98 10.92
length	ljustify	35	0.42
strict	foldl'	144	0.20
foldl'	length foldl'	35 144	0.46 1.42
createEquiJoin	join	1	1.66
equiJoinRow	createEquiJoin	29	1.48

In	From	No of Calls	Time in seconds
getEntityFromBindingList	equiJoinRow	58	6.42
typeForeignKey	join	1	0.06
typePrimaryKey	join	1	0.06
==	join	1	0.00
	equiJoinRow	29	0.02
	getEntityFromBindingList	116	0.26
	typeForeignKey	3	0.00
	typePrimaryKey	1	0.00
/=	createEquiJoin	25	0.06
	getEntityFromBindingList	116	0.04
foldr	foldr	46	0.42
	flatten	7	0.10
flatten	showtable	7	0.04
filter	getEntityFromBindingList	58	1.48
	typeForeignKey	1	0.04
	typePrimaryKey	1	0.02
	filter	60	1.60
getEntityFromBinding	getEntityFromBindingList	116	1.02
snd	typeForeignKey	1	0.00
	typePrimaryKey	1	0.00
	bindingSetToTable	28	0.04
head	getEntityFromBindingList	58	0.22
	typeForeignKey	1	0.00
	typePrimaryKey	1	0.00

In	From	No of Calls	Time in seconds
	bindingSetToTable	1	0.00
tableToBindingSet	join	2	0.10
colhdrToBinding	tableToBindingSet	29	0.16
multi	tableToBindingSet	10	0.40
	multi	26	0.82
fst	showtable	7	0.00
	typeForeignKey	3	0.00
	typePrimaryKey	1	0.00
	bindingSetToTable	7	0.00
ljustify	showtable	35	0.26
.	showtable	7	0.02
	typeForeignKey	3	0.02
	typePrimaryKey	1	0.00
	bindingSetToTable	1	0.00
map	showtable	6	0.06
	getEntityFromBindingList	58	0.72
	bindingSetToTable	14	0.20
	tableToBindingSet	4	0.08
	map	190	2.36
++	showtable	7	0.06
	createEquiJoin	2	0.02
	equiJoinRow	8	0.06
	ljustify	35	0.22
	++	1396	14.18
	flatten	46	0.54

In	From	No of Calls	Time in seconds
nullEntity	getEntityFromBindingList	116	0.30
	getEntityFromBinding	58	0.06

Appendix B

Introduction to Haskell

In this appendix a brief introduction to the lazy, higher-order functional language Haskell [Hudak88] is given in order to clarify the features used in the functional rule-based system. Many of the features presented here will are used in examples in this PhD. As this introduction will be brief, further details on functional programming can be found in the many tutorial guides to programming in functional languages. For lazy languages see Bird and Wadler [Bird88] or Glaser, Hankin, and Till [Glaser84]. For strict functional languages see Henderson [Henderson80] or Ableson and Sussman [Ableson85].

As with most languages Haskell has values and types. In the numeric domain there are `Int`'s for integers and `Float`'s for floating point numbers. In Haskell the symbol `::` can be read as *is of type* , where:

```
   1 :: Int
3.14 :: Float
```

There are characters and strings:

```
    'a' :: Char
"hello" :: String
```

and lists:

```
[1,2,3,4] :: [Int]
```

When a type value is in square brackets it is read as *list of* type. The type *String* is the same as *[Char]* such that `"hello"` is *shorthand* for `['h', 'e', 'l', 'l', 'o']`. The use of *[Char]* is so common that the shorthand for it is deemed essential.

Lists in Haskell are polymorphic and therefore may be of *any* type, but not of mixed types. If more than one object needs to be mixed, Haskell provides tuples. They can contain similar or mixed polymorphic types. For example:

```
(0, 0) :: (Int, Int)
```

is a 2-tuple with the same type, and

```
(1, 3.14, 'a') :: (Int, Float, Char)
```

is a 3-tuple with mixed types. Tuples can be constructed of any *arity* [39].

Functional programs are made up of *expressions*. Some examples are:

```
take 2 [1,2,3,4]
```

returns

```
[1,2] :: [Int]
```

or

```
filter even [1,2,3,4]
```

returns

```
[2,4] :: [Int]
```

or

```
map add1 [1,2,3,4]
```

returns

```
[2,3,4,5] :: [Int]
```

Expressions can be arbitrarily complex and can be combined easily with one another. In imperative languages there are commands and expressions which cannot be easily combined because expressions return values and commands do operations. Functional languages present a uniformity to the programmer.

One of the features of modern functional programming languages is laziness. This is a technique whereby values are evaluated when they are needed. This allows the programmer to create very general algorithms without worrying about the resources that

[39] 1-tuples are not allowed because they are syntactically the same as bracketed expressions. 0-tuples, which contain no value, are written as ().

are consumed. In the following example, the expression `[1..]` reads as 1 to infinity. In a strict language `[1..]` would be evaluated before any other computation is started. Therefore, such a program would never terminate. In a lazy language, the amount of computation is dependent on the context, so:

```
take 10 [1..]
```

returns

```
[1,2,3,4,5,6,7,8,9,10] :: [Int]
```

without entering a non-terminating condition.

In Haskell, new names are introduced with function definitions. The programmer is encouraged to state the type of the definition, even though this is not essential as Haskell can derive the type for any expression. Consider a function definition to add 1 to an integer:

```
add1 :: Int -> Int
add1 x = x + 1
```

The first line is called the *type signature*. It states that `add1` takes an *Int* as an argument and returns an *Int* as a result. The type signature for + is [40]:

```
(+) :: Int -> Int -> Int
```

It takes 2 *Int*'s as arguments and returns an *Int*.

Another feature of modern functional programming languages is higher-order functions. Functions which manipulate other functions, either by taking functions as arguments or by returning functions as results, are said to be higher-order. Functions are treated in the same way as values such as *Int* and *Char*. Higher-order functions encourage the use of the building blocks approach to software development. In particular, named arguments to functions can be dropped so that `add1` can be defined as:

---

[40] Any function with a non-alphabetic name is surrounded by parentheses. Full details are in the Haskell report.

```
add1 = (+ 1)
```

Expressions, laziness, and higher-order functions can be combined, such as:

```
map add1 (take 10 [1..])
```

which returns:

```
[2,3,4,5,6,7,8,9,10,11]
```

In fact the name of the function `add1` is not needed, and the expression can be written as:

```
map (+1) (take 10 [1..])
```

The function `map` is one of those that is polymorphic. It can be used on many types of objects, so there is no need for a mapping function for each type. The type signature for `map` is:

```
map :: (a -> b) -> [a] -> [b]
```

The letters *a* and *b* represent arbitrary and potentially different types. `map` has 2 arguments, the first is a function of type *(a -> b)*, the second is a list of *a*'s. `map` applies the function to every element of the argument list, and returns a list of *b*'s. In the previous function the first argument was the function `(+1)` which is of type *Int -> Int*. Both the input list and returned list were of type *[Int]*. Consider another example using map, in which the function `even` is passed as an argument. `even` returns whether or not the argument passed to it is an even integer. The function `even` has type signature:

```
even :: Int -> Bool
```

We can use `even` in:

```
map even (take 8 [1..])
```

which has the result:

```
[False, True, False, True, False, True, False, True] :: [Bool]
```

The theoretical basis for functional languages is lambda calculus, and many programmers wish to manipulate lambda expressions in their programs. Consider an example function which doubles an integer:

```
double :: Int -> Int
double x = x + x
```

This can be used as follows:

```
map double [1..5]
```

which returns:

```
[2,4,6,8,10]
```

However, the function `double` can be replaced with a Haskell lambda expression. The lambda calculus term $\lambda x \,.\, x + x$ can be written in Haskell as `\x -> x+x`. This can be used in the following way:

```
map (\\x -> x+x) [1..5]
```

which returns:

```
[2,4,6,8;10]
```

This leads into two more features of modern functional programming languages, sharing and referential transparency. When sharing occurs, in the expression:

```
double (complicated 10)
```

the term `(complicated 10)` only gets evaluated *once*. In languages without sharing, given a similar definition of double, `complicated` would be evaluated twice. With referential transparency, the expression:

```
double (complicated 10) == complicated 10 + complicated 10
```

is *always* true. One of these terms can be replaced by the other term at any time without altering the meaning and value of the program.

Data Types In Haskell

Haskell allows the definition of new data types to complement the set of built-in types. For example:

```
data Temperature = Farenheight Float |
                   Celcius Float
```

defines a new type *Temperature* which is a union of two type constructors. *Celcius* and *Farenheight* are the type constructors, and they both have type *Float -> Temperature*.

The new types then can be used in function definitions. Haskell allows the new type to be pattern matched in the definition of a function. Consider a function that takes an arbitrary value of type *Temperature* and returns a *Temperature* which always uses the *Celcius* constructor:

```
t_to_c :: Temperature -> Temperature
t_to_c (Farenheight f) = Celcius ((f-32)*9/5)
t_to_c (Celcius c)     = Celcius c
```

This new function `t_to_c` can be used anywhere that *Temperature*'s are needed. This can be seen in the following function which returns the number of degrees to absolute zero (0 Kelvin):

```
degrees_to_abs_zero :: Temperature -> Float
degrees_to_abs_zero t = temp - abs_zero
                        where
                        abs_zero = -273.05
                        Celcius temp = t_to_c t
```

Another feature of Haskell is the ability to have local definitions which are only in the scope of the lexically containing function. Modern functional languages, including Haskell, determine if an expression is local by using what is called the *offside* rule. Any definitions indented to the right of, or equal to, the first symbol after the *where* symbol are considered to be onside and local to the current definition. Expressions with less indentation are considered to be offside of the *where* symbol and therefore not a

local definition. The *where* symbol has to be to the right of the first character of the function definition. The Haskell manual gives a full description of its treatment of layout control and indentation.

Haskell also allows the definition of types which are parameterized and polymorphic. Consider the type *Finder* in:

```
data Finder a = Found a |
              Fail
```

Finder may be instantiated over arbitrary types because it is polymorphic. There may be instances of *Finder Int*, *Finder Float*, *Finder Temperature*, and so on.

The following function, `is_in`, checks to to see if a value is in a list. The returned value is of type *Finder a*. In languages such as C [Kernighan78a], an error value returned by a function can be:

a) part of the domain, e.g. functions returning integers often return `-1` to mean failure even though `-1` is part of the integer domain

b) set in a global variable, which has to be checked after the function has returned.

Because new types can be easily created and manipulated in Haskell, these programming *styles* can be avoided and more correct functions written. The `is_in` function can be written as:

```
is_in :: a -> [a] -> Finder a
is_in value []     = Fail
is_in value (h:t) = Found value,    value == h
                  = is_in value t, otherwise
```

Within `is_in` there is more pattern matching, with the list structure which is the second argument represented by `(h:t)`. Here `h` is the head of the list and `t` is the tail. We also see the equational style of programming, with the symbol comma (i.e. a `,`) separating the returned expression on its left from the guard on the right, such that `expr, guard` can be read as "if guard then expr". This style is similar to

mathematical notation and is popular in the functional programming community. The function `is_in` can be used thus:

```
is_in 3 [1,2,3,4]
```

which returns:

```
Found 3 :: Finder Int
```

Haskell allows functions to be put in an infix position by surrounding the function name in backquotes ``` ` ```. This makes expressions more readable, so:

```
5 `is_in` [1,2,3,4]
```

returns:

```
Fail :: Finder Int
```

or:

```
1.5 `is_in` [0.5, 1.0, 1.5, 2.0]
```

returns:

```
Found 1.5 :: Finder Float
```

This brief introduction highlights the main features of the modern functional programming language Haskell, many of which will be used in this PhD. Full details of Haskell can be found in the Haskell defintion report [Hudak88].

Appendix C

An OPS5 test program

In this appendix, the source of an example program for testing OPS5 is presented. This program solves the "Monkey and Bananas Problem" in which there is a monkey in a room and some bananas attached to the ceiling. The example program uses a set of goals which enables the monkey to reach the bananas. A full description of the design and implementation of this program can be found in [Brownston85].

```
(p mb1
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on ceiling)
    -->
     (make goal ^status active ^type move ^object ladder ^to <p>))

(p mb2
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on ceiling)
     (object ^name ladder ^at <p>)
    -->
     (make goal ^status active ^type on ^object ladder))

(p mb3
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on ceiling)
     (object ^name ladder ^at <p>)
     (monkey ^on ladder)
    -->
     (make goal ^status active ^type holds ^object nil))
```

```
(p mb4
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on ceiling)
     (object ^name ladder ^at <p>)
     (monkey ^on ladder ^holds nil)
    -->
     (write (crlf) grab <w>)
     (modify 4 ^holds <w>)
     (modify 1 ^status satisfied))

(p mb5
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on floor)
    -->
     (make goal ^status active ^type walk-to ^object <p>))

(p mb6
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on floor)
     (monkey ^at <p>)
    -->
     (make goal ^status active ^type holds ^object nil))

(p mb7
     (goal ^status active ^type holds ^object <w>)
     (object ^name <w> ^at <p> ^on floor)
     (monkey ^at <p> ^holds nil)
    -->
     (write (crlf) grab <w>)
     (modify 3 ^holds <w>)
     (modify 1 ^status satisfied))
```

```
(p mb8
     (goal ^status active ^type move ^object <o> ^to <p>)
     (object ^name <o> ^weight light ^at <> <p>)
    -->
     (make goal ^status active ^type holds ^object <o>))

(p mb9
     (goal ^status active ^type move ^object <o> ^to <p>)
     (object ^name <o> ^weight light ^at <> <p>)
     (monkey ^holds <o>)
    -->
     (make goal ^status active ^type walk-to ^object <p>))

(p mb10
     (goal ^status active ^type move ^object <o> ^to <p>)
     (object ^name <o> ^weight light ^at <p>)
    -->
     (modify 1 ^status satisfied))

(p mb11
     (goal ^status active ^type walk-to ^object <p>)
    -->
     (make goal ^status active ^type on ^object floor))

(p mb12
     (goal ^status active ^type walk-to ^object <p>)
     (monkey ^on floor ^at { <c> <> <p> } ^holds nil)
    -->
     (write (crlf) walk to <p>)
     (modify 2 ^at <p>)
     (modify 1 ^status satisfied))
```

```
(p mb13
     (goal ^status active ^type walk-to ^object <p>)
     (monkey ^on floor ^at { <c> <> <p> } ^holds <w> <> nil)
     (object ^name <w>)
    --> .
     (write (crlf) walk to <p>)
     (modify 2 ^at <p>)
     (modify 3 ^at <p>)
     (modify 1 ^status satisfied))

(p mb14
     (goal ^status active ^type on ^object floor)
     (monkey ^on { <x> <> floor } )
    -->
     (write (crlf) jump onto the floor)
     (modify 2 ^on floor)
     (modify 1 ^status satisfied))

(p mb15
     (goal ^status active ^type on ^object <o>)
     (object ^name <o> ^at <p>)
    -->
     (make goal ^status active ^type walk-to ^object <p>))

(p mb16
     (goal ^status active ^type on ^object <o>)
     (object ^name <o> ^at <p>)
     (monkey ^at <p>)
    -->
     (make goal ^status active ^type holds ^object nil))
```

```
(p mb17
     (goal ^status active ^type on ^object <o>)
     (object ^name <o> ^at <p>)
     (monkey ^at <p> ^holds nil)
    -->
     (write (crlf) climb onto <o>)
     (modify 3 ^on <o>)
     (modify 1 ^status satisfied))

(p mb18
     (goal ^status active ^type holds ^object nil)
     (monkey ^holds { <x> <> nil } )
    -->
     (write (crlf) drop <x>)
     (modify 2 ^holds nil)
     (modify 1 ^status satisfied))

(p t1
     (start)
    -->
     (make monkey ^at _5_7 ^on couch)
     (make object ^name couch ^at _5_7 ^weight heavy)
     (make object ^name bananas ^on ceiling ^at _2_2)
     (make object ^name ladder ^on floor ^at _9_5 ^weight light)
     (make goal ^status active ^type holds ^object bananas)
     (remove 1))
```

References

Ableson85. H. ABLESON, G.J. SUSSMAN, AND J. SUSSMAN, *Structure and Interpretation of Computer Programs,* MIT Press, 1985.

Achten92. P. ACHTEN, J. VAN GRONONGEN, AND R. PLASMEIJER, "High Level Specification of I/O in Functional Languages," Workshop on Functional Programming, pp. 1-17, University of Glasgow, 1992.

Aho86. A. AHO, R. SETHI, AND J.D. ULLMAN, *Compilers Principles, Techniques and Tools,* Addison Wesley, 1986.

Appel87. A.W. APPEL AND D.B. MACQUEEN, "A Standard ML Compiler," LNCS 274, Proc. FPCA, pp. 301-324, Springer Verlag, 1987.

Appel88. A.W. APPEL, B.F. DUBA, AND D.B. MACQUEEN, "Profiling in the Presence of Optimization and Garbage Collection," Part of the New Jersey SML distributed documentation, 1988.

Article91. ARTICLE, "Laser Chips," Communications Week International, 29th January, p. 21, 1991.

Arya89. K. ARYA, "Processes in a Functional Animation System," Proc. FPCA, pp. 382-395, 1989.

Augustsson89. L. AUGUSTSSON AND T. JOHNSSON, "The Chalmers Lazy-ML Compiler," *The Computer Journal*, vol. 32, no. 2, pp. 127-141, 1989.

Augustsson92. L. AUGUSTSSON AND T. JOHNSSON, *The Lazy ML User's Manual,* Chalmers University, 1992.

Augustsson92a. L. AUGUSTSSON AND T. JOHNSSON, *The Haskell B User's Manual (Draft),* Chalmers University, 1992.

Backus78. J. BACKUS, "Can Programming Be Liberated from the von-Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, vol. 21, no. 8, pp. 613-641, 1978.

Bird88. R. BIRD AND P. WADLER, *Introduction to Functional Programming,* Prentice-Hall, 1988.

Boyle92. J.M. BOYLE AND T.J. HARMER, "A practical functional program for the CRAY X-MP," *Journal of Functional Programming*, vol. 2, no. 1, pp. 81-126, 1992.

Brownston85. L. BROWNSTON, R. FARRELL, E. KANT, AND N. MARTIN, *Programming Expert Systems in OPS5,* Addison-Wesley, 1985.

Burn87. G. BURN, "Evaluation Transformers - A Model for the Parallel Evaluation of Functional Languages," LNCS 274, Proc. FPCA, pp. 446-470, Springer Verlag, 1987.

Burstall80. R. BURSTALL AND D. MCQUEEN, "HOPE: An Experimental Applicative Language," Proc. 1st International LISP conference, 1980.

Burstall77. R.M. BURSTALL AND J. DARLINGTON, "A transformation system for developing recursive programs," *Journal of the ACM*, vol. 24, no. 1, pp. 44-67, 1977.

Charniak85. E. CHARNIAK AND D. MCDERMOTT, *Introduction to Artificial Intelligence,* Addison-Wesley, 1985.

Clack85. C. CLACK AND S.L. PEYTON-JONES, "Generating Parallelism from Strictness Analysis," Proc. Workshop on Implementation of Functional Languages, Report 17, pp. 92-131, Chalmers University of Technology and University of Goteborg, 1985.

Clack85a. C. CLACK, S.L. PEYTON-JONES, AND J. SALKILD, *A Brief Overview of GRIP - A Parallel Graph Reduction Machine,* Dept. of Comp. Sci., UCL,

1985.

Clack86. C.D. CLACK AND S.L PEYTON-JONES, "The Four-Stroke Reduction Engine," Proc. ACM Symposium On Lisp and Functional Programming, 1986.

Clack92. C.D. CLACK, *DIGRESS Project - Technical Overview,* Athena Systems Design Ltd , 1992.

Clayman87. S. CLAYMAN, "Exploiting Parallelism in Knowledge-Based Systems," KP-IS-87-1, Dept. Information Systems, Kingston Polytechnic, 1987.

Clayman91. S. CLAYMAN, D. PARROTT, AND C. CLACK, "A Profiling Technique For Lazy, Higher-Order Functional Programs," RN/92/24, Dept. of Computer Science, University College London, 1991.

Clayman92. S. CLAYMAN, D. PARROTT, AND C. CLACK, "Analysing the Behaviour of Lazy, Higher-Order Functional Programs," RN/92/82, Dept. of Computer Science, University College London, 1992.

Clayman93. S. CLAYMAN, D. PARROTT, AND C. CLACK, "An Abstract Data Type for Functional Graphs," RN/93/??, Dept. of Computer Science, University College London, 1993.

Clayman93a. S. CLAYMAN, D. PARROTT, AND C. CLACK, "Lexical Profiling: Theory and Practice," RN/93/23, Dept. of Computer Science, University College London, 1993.

Cole90. M. COLE, *Higher-Order Functions for Parallel Evaluation,* University of Glasgow, 1990.

Cripps87. M.D. CRIPPS, A.J. FIELD, AND M.J. REEVE, "An Introduction to ALICE: a Multiprocessor Graph Reduction Machine," in *Functional Programming: Languages, Architectures and Tools*, ed. S. Eisenbach, Ellis Horwood, 1987.

Darlington80. J. DARLINGTON, "Program Transformation," in *Functional Programming and its Applications*, ed. D.A. Turner, Cambridge University Press, 1980.

Darlington90. J. DARLINGTON AND , ET AL, "A Functional Programming Environment Supporting Execution, Partial Execution and Transformation," Proc. 2nd Joint ICOT/DTI-SERC Workshop on Decomposition of Parallel Applications and Benchmarking Evaluation of Parallel Systems, 1990.

Darlington91. J. DARLINGTON AND , ET AL, *Structured Parallel Functional Programming,* Dept. of Computing, Imperial College London, 1991.

Duff83. M. DUFF, "The CLIP-4 Image Processor," personal communication, Dept. of Image Processing, UCL., 1983.

Dwelly89. A. DWELLY, "Functions and Dynamic User Interfaces," Proc. FPCA, pp. 371-381, 1989.

Eager86. D.L. EAGER, J. ZAHORJAN, AND E.D. LAZOWSKA, "Speedup versus efficiency in parallel systems," Tech Report 86-08-01, University of Sasketchewan, 1986.

Eekelen89. M.C.J.D. VAN EEKELEN, M.J. PLASMEIJER, AND J.E.W. SMETSERS, "Communicating Functional Processes," Technical Report no. 89-3, University of Nijmegen, Dept. of Infomatics, 1989.

Eekelen90. M.C.J.D. VAN EEKELEN AND M.J. PLASMEIJER, "Concurrent Programming in a Functional Language," Proc. 2nd International Workshop on. Implementation of Functional Languages on Parallel Architectures, Technical Report no. 90-16, pp. 131-253, University of Nijmegen, Dept. of Infomatics, 1990.

Fairburn87. J. FAIRBURN AND S.C. WRAY, "TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators," LNCS 274, Proc. FPCA, Springer Verlag, 1987.

Ferguson88. A.B. FERGUSON AND P. WADLER, *When Will Deforestation Stop,* University of Glasgow, 1988.

Forgy81. C.L. FORGY, "The OPS5 User's Manual," CMU-CS-81-135, Dept. Comp. Sci., CMU, PA., 1981.

Forgy82. C.L. FORGY, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, pp. 17-37, 1982.

Fox89. G.C. FOX, "Parallel computing comes of age: supercomputer level parallel computations at Caltech," *Concurrency: Practice and Experience*, vol. 1, no. 1, pp. 63-103, 1989.

Gabriel85. R.P. GABRIEL, *Performance and Evaluation of LISP Systems,* MIT Press, 1985.

Ghosh91. A. GHOSH AND B. BORTZ, *Project DIGRESS - Distributed Graph Reduction Experimental Support System,* Dept. of Computer Science, University College London, 1991.

Glaser84. H. GLASER, C. HANKIN, AND D. TILL, *Principles of Functional Programming,* Prentice Hall, 1984.

Glaser90. H. GLASER, P. HARTEL, AND J. WILD, "A Pragmatic Approach to the Analysis and Compilation of Lazy Functional Languages," Proc. 2nd International Workshop on Implementation of Functional Languages on Parallel Architectures, Technical Report no. 90-16, pp. 203-222, University of Nijmegen, Dept. of Infomatics, 1990.

Glauert90. J. GLAUERT, "Compiling Functional Languages Based On Graph Rewriting," Proc. 2nd International Workshop on Implementation of Functional Languages on Parallel Architectures, Technical Report no. 90-16, pp. 155-169, University of Nijmegen, Dept. of Infomatics, 1990.

Goldberg88. B. GOLDBERG, "Buckwheat: Graph Reduction on a Shared Memory Multiprocessor," Proc. ACM Symposium On Lisp and Functional Programming, 1988.

Graham82. S.L. GRAHAM, P.B. KESSLER, AND M.K. MCKUSICK, "gprof: a Call Graph Execution Profiler," *SIGPLAN*, vol. 17, no. 6, pp. 120-126, 1982.

Griesner84. J.H. GRIESNER, "YES/MVS: A Continuous Real Time Expert System," Proc. AAAI-84, 1984.

Gupta84. A. GUPTA, "Implementing OPS5 Production Systems on DADO," CMU-CS-84-115, Dept. Comp. Sci., CMU, PA., 1984.

Gupta86. A. GUPTA, "Parallelism in Production Systems," PhD thesis, CMU-CS-86-122, Dept. Comp. Sci., CMU, PA, 1986.

Gupta86a. A. GUPTA, C. FORGY, A. NEWELL, AND R. WEDIG, *Parallel Algorithms and Architectures for Rule-Based Systems*, 1986.

Gupta89. A. GUPTA, C. FORGY, AND A. NEWELL, "High-Speed Implementations of Rule-Based Systems," *ACM Transactions on Computer Systems*, vol. 7, no. 2, pp. 119-146, 1989.

Hall90. C.V. HALL, K. HAMMOND, AND J. O'DONNELL, "An Algorithmic and Semantic Approach to Debugging," Workshop on Functional Programming, pp. 44-53, University of Glasgow, 1990.

Hall92. C.V. HALL, K. HAMMOND, W. PARTAIN, S.L. PEYTON-JONES, AND P. WADLER, "The Glasgow Haskell Compiler: A Retrospective," Workshop on Functional Programming, pp. 62-71, University of Glasgow, 1992.

Hammond91a. K. HAMMOND AND S.L. PEYTON-JONES, *Profiling Scheduling Strategies on the GRIP Parallel Reducer,* University of Glasgow, 1991.

Hammond91b. K. HAMMOND AND S.L. PEYTON-JONES, *Some Early Experiments on the GRIP Parallel Reducer,* University of Glasgow, 1991.

Hammond91. K. HAMMOND AND S.L. PEYTON-JONES, *How to use the GRIP Mail Server Revised Report,* University of Glasgow, 1991.

Hartel92. P. HARTEL AND W.G. VREE, "Arrays in a Lazy Functional Language - a case study: the Fast Fourier Transform," TR CS-92-02, Dept. of Computer Systems, University of Amsterdam, 1992.

Hayes-Roth83. F. HAYES-ROTH, D.A. WATERMAN, AND D.B. LENAT, *Building Expert Systems,* Addison Wesley, 1983.

Hayes-Roth85. F. HAYES-ROTH, "Rule-Based Systems," *CACM*, vol. 28, no. 9, pp. 921-932, 1985.

Henderson80. P. HENDERSON, *Functional Programming - Application and Implementation,* Prentice Hall, 1980.

Henderson82. P. HENDERSON, "Functional Geometry," Proc. ACM Symposium On Lisp and Functional Programming, 1982.

Hill92. J.M.D. HILL, *Data Parallel Haskell: Mixing old glue and new glue,* Dept. of Computer Science, Queen Mary and Westfield College, University of London, 1992.

Hillis85. W.D. HILLIS, *The Connection Machine,* MIT Press, 1985.

Hillyer86. B.K. HILLYER AND D.E. SHAW, "Execution of OPS5 Production Systems on a Massively Parallel Machine," *Journal of Parallel and Distributed Computing*, vol. 3, pp. 236-268, 1986.

Horowitz76. E. HOROWITZ AND S. SAHNI, *Fundamentals of Data Structures,* Pitman, 1976.

Hudak84. P. HUDAK, "Distributed Applicative Processing Systems: Project Goals, Motivation, and Status Report," YALEU/DCS/TR-317, Dept. Comp. Sci., Yale University, 1984.

Hudak85a. P. HUDAK AND B. GOLDBERG, "Serial Combinators: "Optimal" Grains of Parallelism," Proc. FPCA, 1985.

Hudak85. P. HUDAK AND L. SMITH, "Para-Functional Programming - A Paradigm for Programming Multiprocessor Systems," YALEU/DCS/RR-390, Dept. Comp. Sci., Yale University, 1985.

Hudak88. P. HUDAK AND P. WADLER, *Report on the Functional Programming Language Haskell,* Dept. Comp. Sci., University of Glasgow, 1988.

Hudak89. P. HUDAK, "Conception, Evolution and Application of Functional Programming Languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359-411, 1989.

Hudak93. P. HUDAK AND D. RABIN, *State in Functional Languages: An Annotated Bibliography,* Dept of Computer Science, Yale University, 1993.

Hughes89. J. HUGHES, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, no. 2, pp. 98-107, 1989.

Hutton90. G. HUTTON, *Parsing Using Combinators,* University of Glasgow, 1990.

Inmos85. INMOS, *The Transputer Reference Manual,* Inmos Ltd, 1985.

Intel85. INTEL, *iPSC User's Guide,* Intel Corporation, 1985.

Johnsson90. T. JOHNSSON, "Discussion summary: which analysis?," in *Functional Languages: Optimization for Parallelism*, ed. R. Wilhelm, 1990.

Jones86. R. JONES, "Flex: An experience of Miranda," UKC Computing Laboratory Report No. 38, 1986.

Jouret91. G.K. JOURET, "Exploiting Data-Parallelism in Functional Languages," PhD thesis, Dept. of Computing, Imperial College London, 1991.

Kelly90. P. KELLY, "Work in Progress on Compiling Caliban," in *Functional Languages: Optimization for Parallelism*, ed. R. Wilhelm, 1990.

Kelly87. P.H.J. KELLY, "Functional Programming for Loosely-Coupled Multiprocessors," PhD thesis, Univ. of London, 1987.

Kernighan78. B.W. KERNIGHAN AND P.J PLAUGER, *The Elements of Programming Style*, McGraw Hill, 1978.

Kernighan78a. B.W. KERNIGHAN AND D.M. RITCHIE, *The C Programming Language*, Prentice-Hall, 1978.

King92. D.J. KING AND P. WADLER, "Combining Monads," Workshop on Functional Programming, pp. 134-143, University of Glasgow, 1992.

Knuth68. D.E. KNUTH, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1968.

Koopman90. P.W.M. KOOPMAN, M.C.J.D. VAN EEKELEN, AND M.J. PLASMEIJER, "Abstract Machine Specification in Functional Languages," Proc. 2nd International Workshop on Implementation of Functional Languages on Parallel Architectures, Technical Report no. 90-16, pp. 297-320, University of Nijmegen, Dept. of Infomatics, 1990.

Mauny89. M. MAUNY, "Parsers and Printers as Stream Destructors and Constructors Embedded in Functional Languages," Proc. FPCA, pp. 360-370, 1989.

May83. D. MAY, "Large Languages versus Small Languages," *IFIP Panel*, 1983.

May84. D. MAY, *OCCAM*, Inmos Ltd, 1984.

McBurnley90. D.L. MCBURNLEY AND M.R. SLEEP, "Concurrent Clean on ZAPP," Proc. 2nd International Workshop on Implementation of Functional Languages on Parallel Architectures, Technical Report no. 90-16, pp. 73-113, University of Nijmegen, Dept. of Infomatics, 1990.

McDermott82. JOHN MCDERMOTT, "R1: A Rule-Based Configurer of Computer Systems," *Artificial Intelligence*, vol. 19, pp. 39-88, 1982.

McDermott78. J. MCDERMOTT, A. NEWELL, AND J. MOORE, "The Efficiency of Certain Production System Implementations," in *Pattern-Directed Inference Systems*, ed. F. Hayes-Roth, Academic Press, 1978.

Miranker87. D.P. MIRANKER, "Treat: A New and Efficient Match Algorithm for AI Production Systems," PhD thesis, Columbia University, 1987.

Moggi89. E. MOGGI, "Computational lambda calculus and monads," IEEE Symposium on Logic in Computer Science, 1989.

Newell78. A. NEWELL, "HARPY - Production Systems and Human Cognition," CMU-CS-78-140, Dept. Comp. Sci., CMU, PA., 1978.

Nilsson92. H. NILSSON AND P. FRITZSON, "Algorithmic Debugging for Lazy Functional Languages," PLILP'92, 1992.

Oflazer87. K. OFLAZER, "Partitioning in Parallel Processing of Production Systems," PhD thesis, CMU-CS-87-114, Dept. Comp. Sci., CMU, PA., 1987.

Padua87. D. PADUA, *Languages and Compilers for Parallel Programming*, Centre for Supercomputing, University of Illinois, 1987.

Parrott90. D. PARROTT AND S. CLAYMAN, "Report on 'Cost' and 'Debug' primitive extentions to FLIC," RN/91/79, Dept. of Computer Science, University College London, 1990.

Parrott91. D. PARROTT AND C. CLACK, "A Common Graphical Form," RN/91/27, Dept. of Computer Science, University College London, 1991.

Parrott92. D. PARROTT AND C. CLACK, "Paragon - A Language for Modelling Lazy, Functional Workloads on Distributed Processors," RN/92/72, Dept. of Computer Science, University College London, 1992.

Parrott93. D. PARROTT, "Synthesising Lazy Functional Workloads for Distributed Environments," PhD thesis, Dept. of Computer Science, University College London, 1993.

Partain92. W. PARTAIN, "The nofib Benchmark Suite of Haskell Programs," Workshop on Functional Programming, pp. 195-202, University of Glasgow, 1992.

Perlin89. M.W. PERLIN, "The Match Box Algorithm for Parallel Production System Match," CMU-CS-89-163, Dept. Comp. Sci., CMU, PA, 1989.

Peyton-Jones85. S.L. PEYTON-JONES, "Yacc in Sasl - an Exercise in Functional Programming," *Software - Practice and Experience*, vol. 15, no. 8, pp. 807-820, 1985.

Peyton-Jones87. S.L. PEYTON-JONES, *The Implementation of Functional Programming Languages,* Prentice-Hall, 1987.

Peyton-Jones89. S.L. PEYTON-JONES AND J. SALKILD, "The Spineless Tagless G-machine," Proc. FPCA, pp. 184-201, 1989.

Peyton-Jones89a. S.L. PEYTON-JONES, "Parallel Implementations of Functional Programming Languages," *The Computer Journal*, vol. 32, no. 2, pp. 98-107, 1989.

Press88. W.H. PRESS, B.P. FLANNERY, S.A. TEULOLSKY, AND W.T. VETTTERLING, *Numerical Recipes in C,* Cambridge University Press, 1988.

Rich83. E. RICH, *Artificial Intelligence,* McGraw Hill, 1983.

Robertson89. I.B. ROBERTSON, "Hope+ on Flagship," Workshop on Functional Programming, pp. 296-307, University of Glasgow, 1989.

Roe89. P. ROE, "Some Ideas On Parallel Functional Programming," Workshop on Functional Programming, pp. 338-352, University of Glasgow, 1989.

Rosenbloom85. P.S. ROSENBLOOM, J.E. LAIRD, J. MCDERMOTT, A. NEWELL, AND E. ORCIUCH, "R1-Soar: An Experiment in Knowledge Intensive Programming in a Problem-Solving Architecture," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 5, pp. 561-569, 1985.

Rosenthal85. D. ROSENTHAL, "Adding Meta Rules to OPS5: A Proposed Extension," *SIGPLAN*, vol. 20, no. 10, pp. 79-86, 1985.

Runciman89. C. RUNCIMAN, M. FIRTH, AND N. JAGGER, "Transformation in a Non-Strict Language: An Approach to Instantiation," Workshop on Functional Programming, pp. 133-141, University of Glasgow, 1989.

Runciman90. C. RUNCIMAN AND D. WAKELING, "Problems & Proposals for Time & Space Profiling of Functional Programs," Workshop on Functional Programming, pp. 237-245, University of Glasgow, 1990.

Runciman91. C. RUNCIMAN, "TIP in Haskell - Another Exercise in Functional Programming," Workshop on Functional Programming, pp. 278-292, University of Glasgow, 1991.

Runciman92. C. RUNCIMAN AND D. WAKELING, "Heap Profiling for Lazy Functional Languages," Technical Report No. 172, University of York, 1992.

Sanders92. P. SANDERS AND C. RUNCIMAN, "LZW Text Compression in Haskell," Workshop on Functional Programming, pp. 215-226, University of Glasgow, 1992.

Sansom92. P.M. SANSOM AND S.L. PEYTON-JONES, *Profiling Lazy Functional Languages,* University of Glasgow, 1992.

Schmidt86. D.A. SCHMIDT, *Denotational Semantics - A Methodology for Language Development,* Allyn and Bacon, 1986.

Schultz88. M.H. SCHULTZ, *Lecture Notes on Parallelism,* Dept of Computer Science, Yale University, 1988.

Sinclair90. D.C. SINCLAIR, "Solid Modelling in Haskell," Workshop on Functional Programming, pp. 247-263, University of Glasgow, 1990.

Stefik81. M.J. STEFIK, "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence*, vol. 16, pp. 111-140, 1981.

Stolfo83. S.J. STOLFO, D. MIRANKER, AND D.E. SHAW, "Architecture and Applications of DADO: A Large Scale Parallel Computer for Artificial Intelligence," Proc. International Joint Conference on Artificial Intelligence, 1983.

Stolfo86. S.J. STOLFO, "Let's Stop the Dust from Collecting on OPS5," Workshop on Performance Efficient Parallel Programming, CMU-CS-86-181, pp. 93-95, Dept. Comp. Sci., CMU, PA., 1986.

Stoy80. J.E. STOY, "Some Mathematical Aspects of Functional Programming," in *Functional Programming and its Applications*, ed. D.A. Turner, Cambridge University Press, 1980.

Stroustrup86. B. STROUSTRUP, *The C++ Programming Language,* Addison-Wesley, 1986.

Thompson86. S. THOMPSON, "Writing Interactive Programs in Miranda," UKC Computing Laboratory Report No. 40, 1986.

Trinder89. P. TRINDER, "Referentially Transparent Database Languages," Workshop on Functional Programming, pp. 142-156, University of Glasgow, 1989.

Turner79. D.A. TURNER, "A New Implementation Technique For Applicative Languages," *Software - Practice & Experience*, vol. 9, pp. 31-49, John Wiley and Sons, 1979.

Turner80. D.A. TURNER, "Recursion Equations as a Programming Language," in *Functional Programming and its Applications*, ed. D.A. Turner, Cambridge University Press, 1980.

Turner82. D.A. TURNER, "Functional programming and proofs of program correctness," in *Tools and Notions for Program Construction*, ed. D. Neel, Cambridge University Press, 1982.

Turner84. D.A. TURNER, "Functional Programs as Executable Specifications," in *Mathematical Logic and Programming Languages*, ed. P. Sheperdson, Prentice Hall, 1984.

Turner85. D.A. TURNER, "Miranda: A non-strict functional language with polymorphic types," LNCS 201, Proc. FPCA, pp. 1-16, Springer Verlag, 1985.

Uhr87. L. UHR, *Multi-computer Architectures for Artificial Intelligence,* Wiley-Interscience, 1987.

Vranken90. J.L.M VRANKEN, "Reflections on Parallel Functional Languages," Proc. 2nd International Workshop on Implementation of Functional Languages on Parallel Architectures, Technical Report no. 90-16, pp. 9-49, University of Nijmegen, Dept. of Infomatics, 1990.

Wadler85. P. WADLER, "How to Replace Failure by a List of Successes," LNCS 201, Proc. FPCA, pp. 113-128, Springer Verlag, 1985.

Wadler90a. P. WADLER, "Linear types can change the world!," IFIP Working Conference on Programming Concepts and Methods, 1990.

Wadler90. P. WADLER, "Comprehending Monads," ACM Conference on Lisp and Functional Programming, 1990.

Wadler91. P. WADLER, *Continuing Monads (Technical Summary),* University of Glasgow, 1991.

Waterman86. D.A. WATERMAN, *A Guide to Expert Systems,* Addison-Wesley, 1986.

Watson79. I. WATSON AND J.R. GURD, "A Prototype Dataflow Computer with Token Labling," Proc. of 1979 National Computer Conference, 1979.

Watson88. I. WATSON, V. WOODS, P. WATSON, R. BANACH, M. GREENBERG, AND J. SARGEANT, "Flagship: A Parallel Architecture for Declarative Programming," FS/MU/IW/017-88, Dept. of Comp. Sci., Univ. of Manchester, 1988.

Winston81. P. WINSTON, *Artificial Intelligence,* Addison Wesley, 1981.

Wray86. S. WRAY, "Implementation and Programming Techniques for Functional Languages," Report No. 92, University of Cambridge Computer Laboratory, 1986.

Zorn88. B. ZORN AND P. HILFINGER, "A Memory Allocation Profiler for C and LISP Programs," Proc. USENIX, pp. 223-237, 1988.