

# Deterministic Concurrency

David Paul Carter

September 1994

A thesis submitted to the University of Bristol in  
accordance with the requirements for the degree of  
Master of Science in the Faculty of Engineering,  
Department of Computer Science.

## **Abstract**

Existing functional languages appear not to be suitable for implementing systems which are inherently concurrent, such as operating system environments. Adaptations to functional languages developed to support such applications have in the past always involved the introduction of non-determinism.

This thesis proposes a purely declarative and deterministic model for concurrency that can be adopted by a standard non-strict functional language. The design and implementation of a simple but complete single-user concurrent working environment is presented in order to demonstrate the feasibility of deterministic design.

## Dedication

This thesis could only possibly be dedicated to Marianne Morrey (née Loo) and Joel Seymour, who made it happen. For all the wrong reasons, but that's part of the learning experience. Here's to a great pair of friends.

## Acknowledgements

Many of concepts and design points raised in this thesis were thrashed out with members of the functional programming group at Bristol University. I am particularly indebted to my supervisor, Ian Holyer for his near infinite patience and long years of gentle encouragement. The original concept of deterministic user environments belongs to him, and I hope that the full potential of the idea will eventually be realised. I would like to thank Neil Davies for being a good friend, and a fountain of common sense and interesting ideas. A special mention must go to Chris Dornan for his dedication and commitment to the field of functional programming.

I am extremely grateful for the support and patience on the part of my friends and family while completing this thesis. I would particularly like to thank Colin Trevor, Nick Burton and my fellow PhD students Dominic Binks and Nick Moffat.

On a technical note, the prototype system presented in this thesis relies on the existence of two major support systems:

1. Mark P. Jones ([mpj@cs.nott.ac.uk](mailto:mpj@cs.nott.ac.uk)) has provided an invaluable service to the functional programming community in the creation and continued support of his Haskell-like interpreter, Gofer [29].

This very powerful but freely distributable tool is available by anonymous FTP from <ftp.dcs.glasgow.ac.uk> in the directory `pub/haskell/gofer`.

2. A lightweight threads library produced by Stephen Crane ([jsc@doc.ic.ac.uk](mailto:jsc@doc.ic.ac.uk)), is available from [gummo.doc.ic.ac.uk](http://gummo.doc.ic.ac.uk) as `pub/rex/lwp.tar.Z`

## Declaration

An early overview of some parts of this work appeared as an internal report (CSTR-93-08), jointly authored with my supervisor Ian Holyer. Unless otherwise stated, all the research described is the author's work. No part of the work described in this thesis has been submitted in support of another degree qualification in this or any other university. The views expressed in this dissertation are those of the author and not of the University of Bristol.

David Carter

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Von Neumann Computing . . . . .	3
2.2	Declarative Languages . . . . .	4
2.2.1	Functional languages . . . . .	5
2.2.2	Logic Programming Languages . . . . .	5
2.2.3	Specification Languages . . . . .	6
2.3	More about the Functional Paradigm . . . . .	6
2.3.1	Higher Order Functions . . . . .	7
2.3.2	Lazy evaluation . . . . .	8
2.3.3	Type Checking . . . . .	9
2.4	A Brief History . . . . .	10
2.5	An Overview of Haskell . . . . .	10
2.6	Advantages and Disadvantages of the functional style . . . . .	11
<b>3</b>	<b>Operating Systems and Functional Languages</b>	<b>14</b>
3.1	Nondeterminism . . . . .	15
3.1.1	Approach suggested by Henderson . . . . .	17
3.1.2	Later Refinements to the model by Henderson and Jones . . . . .	20
3.1.3	Later Refinements to the model by Jones and Sinclair . . . . .	22
3.1.4	Stoye's Sorting Office . . . . .	25
3.1.5	Sorting Office as adapted by Cuppitt and Turner . . . . .	26
3.1.6	Nondeterminism as Data to a program . . . . .	28
3.1.7	Time Stamps . . . . .	28
3.1.8	Conclusions . . . . .	29
3.2	Interfacing . . . . .	29
3.2.1	Dialogue based I/O . . . . .	29
3.2.2	Continuation based I/O . . . . .	31
3.2.3	Monadic I/O . . . . .	33

<b>4</b>	<b>Implementing Functional Languages</b>	<b>34</b>
4.1	Simplification . . . . .	34
4.2	Lambda Lifting . . . . .	35
4.3	Manual Evaluation . . . . .	36
4.4	Representing expressions . . . . .	38
4.5	Template Instantiation . . . . .	41
4.6	The G machine . . . . .	42
4.6.1	The peculiarities of Gofer Gcode . . . . .	42
4.6.2	An example: The map function . . . . .	43
4.6.3	Optimisations to the G machine . . . . .	44
<b>5</b>	<b>Functional Concurrency</b>	<b>46</b>
5.1	Concurrency is not Simply Parallelism . . . . .	46
5.2	A Traditional View of Concurrency in Operating System Design . . . . .	47
5.3	Data Driven Design . . . . .	48
5.4	Demand Driven Design . . . . .	49
5.4.1	Network Structure . . . . .	49
5.4.2	Propagating Demand . . . . .	50
5.4.3	Concurrency in a Demand Driven System . . . . .	50
5.4.4	Multiple Output Channels . . . . .	53
5.4.5	Hyperstrict Communications . . . . .	53
5.4.6	Merge . . . . .	54
5.4.7	Conclusions . . . . .	55
5.5	Behaviour of Concurrent Demand Driven Systems . . . . .	55
5.5.1	Copy Transformation . . . . .	57
5.5.2	Processes can be Networks . . . . .	58
5.6	Network Configuration . . . . .	59
5.6.1	WriteChan Request . . . . .	59
5.6.2	MakeChannel . . . . .	61
5.6.3	Channel Passing . . . . .	62
5.6.4	Conclusions . . . . .	64
<b>6</b>	<b>Deterministic Design</b>	<b>65</b>
6.1	Nondeterminism and resources . . . . .	66
6.1.1	Bitmap Display Example . . . . .	67
6.1.2	FileSystem Resource . . . . .	69
6.2	Handling Interrupts . . . . .	71
6.3	A Design for a User Environment . . . . .	72
6.4	Overview of Prototype Implementation . . . . .	73
6.5	Structure of the Main Shell Loop . . . . .	76
6.6	Behaviour of the prototype system . . . . .	78
6.7	FileSystem Operations . . . . .	81

<b>7</b>	<b>Overview of Support System</b>	<b>84</b>
7.1	Implementing a concurrent interpreter . . . . .	85
7.1.1	Shared heap . . . . .	86
7.1.2	Types of Dialogue . . . . .	87
7.2	Lightweight Threads Library . . . . .	89
7.2.1	Features of the threads library . . . . .	89
7.2.2	Local state extension . . . . .	90
7.2.3	Asynchronous I/O . . . . .	91
7.3	Handling Input and Output . . . . .	92
7.3.1	Interaction between ServerDialogues and the I/O thread . . . . .	93
7.3.2	Providing Reliable Buffered Message Streams . . . . .	94
7.4	Garbage collection . . . . .	97
7.5	Server Processes . . . . .	98
<b>8</b>	<b>Conclusions</b>	<b>100</b>
<b>A</b>	<b>The Gofer G machine instruction set</b>	<b>103</b>
<b>B</b>	<b>Source code</b>	<b>107</b>
<b>C</b>	<b>Functions provided by the Concurrency library</b>	<b>119</b>

# Chapter 1

## Introduction

People like concurrent working environments. In such an environment, they can switch between applications as they like without having to physically shut down, or put to sleep, other activities that may be happily working along at their own pace. A simple example is the ability to continue using an editor while a long and involved compilation is in progress.

With suitable cooperation, applications can arrange to do this between themselves. However, it is much cleaner to hide the switching, so that each application believes that it is the sole custodian of its hardware. This is termed preemptive multitasking.

On the other hand, people do **not** enjoy nondeterministic behaviour. Many Unix programs do not lock files that are being altered, or at least they do not lock them in compatible ways. All too frequently someone will try to use a file without realising that it is being changed, quite possibly by another of their own processes. The results can be frustrating, to say the least.

We suggest the problem is that although concurrency and nondeterminism are two separate issues, the distinction is poorly understood. Traditionally, concurrent systems are implemented using some form of nondeterministic operator. In stateful, procedural, environments there is often little choice as communication between concurrent entities is handled using shared data buffers. This is inherently nondeterministic.

Previously, there has seemed little motivation in hiding this nondeterminism, despite its ill effects. Indeed it is often made easily available to casual application programmers, for example the `select()` system call in Unix. The easy availability of such an operator is dangerous, as programmers use it even when not strictly necessary. On an individual, application by application basis, it may seem that little harm can be done. However,

any system containing a large number of nondeterministic applications will prove to be unpredictable in ways unanticipated by the original authors. In general, the more points of nondeterminism in a system, the more unpredictable it becomes.

It is our thesis that nondeterminism should be hidden, rather than exposed, in a concurrent system. We suggest that a useful concurrent single user working environment can be built without any nondeterministic feature available to the applications programmer.

We further suggest that such a style is particularly suited to the stateless and mathematically pure functional paradigm, where the introduction of nondeterminism has long been a thorny issue, despite its apparent necessity. A simple, but interesting, demonstration system is presented and its features are discussed.

The remainder of this thesis is structured in the following way:

- Chapter 2 discusses the motivation behind the functional paradigm, and the rationale behind common language features. The problems associated with introducing standard models of concurrency in the functional style are described in chapter 3, together with some possible, although inelegant solutions. Chapter 4 describes the graph reduction process used to implement functional languages.
- In chapter 5 we consider a more appropriate model for functional concurrency using a demand driven network of processes. We proceed to demonstrate some ways in which networks of such processes can be configured and controlled completely by the functional programmer.
- Chapter 6 shows how the extra expressiveness provided by demand driven functional processes allows for purely deterministic design of concurrent systems. A prototype system is presented using a concurrent variation of the Gofer programming language. (Complete source code for the functional components of this system can be found in appendix B).
- Chapter 7 gives an overview of the support infrastructure required to implement concurrent graph reduction.
- Finally, in chapter 8, we review the progress that has been made and possible directions for future research.

# Chapter 2

## Background

In this chapter we discuss the need for functional programming languages. We start by considering the nature of traditional Von Neumann architectures and programming languages. The various families of declarative language are then introduced and their characteristics described. Important characteristics of functional programming languages, in particular lazy evaluation, higher order functions and strong polymorphic typing, are discussed in more detail. Finally we discuss advantages and disadvantages associated with the functional style.

### 2.1 Von Neumann Computing

The performance of digital computers has increased many thousand-fold over the last forty years. However, the underlying architecture remains unchanged: a single central processor executes a stream of primitive instructions that manipulate a large store of simple binary values.

Procedural languages developed directly from the assembly languages of such “Von Neumann” computers. The emphasis has always been to ensure that these languages map well to the underlying instruction set. As such they can be viewed as powerful macro-assemblers that contain support for common abstractions such as procedure calls and local variables.

The problem is that such languages remain very low level compared to the algorithms that they are used to implement. Many instructions are required to implement even simple algorithms, forcing the programmer to focus on the individual instructions rather than the

(far more important) whole. In his Turing Award lecture that popularised the functional style [2], Backus termed this “word at a time” programming.

The arbitrary manipulation of a single shared state introduces subtle dependencies between instructions. This means that it is difficult to assign any meaning to a program other than the effects that we observe by executing it. We say that procedural languages have well defined operational semantics, but no independent static or **declarative** meaning. This makes it very difficult to reason about a section of procedural code, or demonstrate that two pieces of code have equivalent effects.

The complexity and subtle dependencies inherent in procedural code create many software engineering problems. Computer scientists have introduced structured programming, abstract data types and more recently object orientated programming in an attempt to contain this complexity. These solutions remain heavily biased towards word-at-a-time thinking.

## 2.2 Declarative Languages

This grouping covers a number of very different programming languages and mathematical notations. The common theme is that programs are constructed by defining permanent relationships between values, giving some form of static meaning. The emphasis is on providing a very high level notation that is convenient for the programmer, rather than an abstract Von Neumann machine.

In purely declarative languages, the programmer has little or no control over the physical order of evaluation used to generate results. This depends on the particular evaluation or search strategy adopted by the language. The programmer is freed from worrying about a large number of mundane implementation issues, but must rely far more on the compiler to generate efficient code.

The motivation for such languages is as follows:

1. Programs are more concise, readable, and easier to maintain. Typically a procedural program will be five or ten times the length of its declarative counterpart. Declarative code also tends to be far more stylised, and predictable in nature.
2. Simple static semantics means that it is much simpler to reason about the behaviour of a program. It becomes possible to transform a program, for optimisation and proof

purposes.

3. Declarative programs are not inherently sequential in nature. Although extra language support must be added to describe concurrent systems, declarative notation maps well to parallel machines, especially where speculative parallelism is involved.

There are three separate groups of languages that have some claim to be declarative.

### 2.2.1 Functional languages

Functional, or applicative, languages are based on the composition of simple computable functions. Each function takes a number of parameters and generates a result. The important difference from the “functions” used by a language such as Pascal is that real mathematical functions are stateless and therefore predictable. While this might seem, at first sight, to **reduce** expressiveness, real functions are far more composable than their stateful counterparts.

The functional style encourages a toolkit mentality, where each function becomes a black box machine with a well defined effect. This works well with a top-down design, bottom up implementation strategy. More information about this paradigm is given in the next section.

### 2.2.2 Logic Programming Languages

Logic programming languages are based on first order logic. The programmer defines a series of logical constraints between a set of objects. Evaluation is powered by a search engine that generates all possible permutations of results that match those constraints. The vast majority of logic programming languages are derivatives of Prolog, a language originally developed as a practical tool in natural language processing.

Logic programming languages are not purely declarative. The main problems arise in the use of “cut” directives that prune the search space, and “assert” and “retract” predicates that manipulate a program representation directly and can lead to self modifying code. While logic programmers see these mechanisms as optimisations to be applied to a purely declarative program, they are frequently abused. In addition, the depth first search strategy that is most commonly used means that the termination of a search may depend operationally on the order in which the search is carried out, not just the declarative

existence of solutions. More recent languages such as Gödel [22] avoid most of the non declarative features, but continue to rely on side effects for I/O behaviour.

Logic programming languages are particularly well suited to knowledge representation and database query, and were consequently adopted, and popularised, by the Japanese in their fifth generation project.

### 2.2.3 Specification Languages

Specification languages are mathematical notations used to reason about the behaviour of a program and its subcomponents. As such, they can be seen as design tools that can help lay out the structure of a program and the interfaces between modules. Such languages are not practical programming languages because they frequently rely on infinite computation or sets of data. The properties of a system can be described by relationships between infinite sets without specifying any algorithm to derive one set from other.

Typically a programmer will write a top level specification for a module and then refine this description until a feasible implementation is identified. Because specifications are mathematically tractable, we can prove that each refinement step is consistent with the previous specification.

An executable subset of a specification language is effectively a functional language, and it is possible to **animate** suitable specifications to serve as rapid prototypes for a system.

## 2.3 More about the Functional Paradigm

We have introduced functional languages as a notation based on the composition of pure, stateless, functions. There is no shared global state. While functions may define local “variables” as a notation for shared computation, these values are not updateable. Consequently all instances of a variable or function call with fixed parameters in a given scope are guaranteed to have the same value. We say that the functions and variables are **referentially transparent**.

This concept underpins the simple static semantics of functional languages. It is important that this property is maintained, especially when interfacing to a **stateful** environment outside the functional programmer’s control.

An important consequence of referential transparency is the ability to replace a definition with its body and vice versa. Given a function definition:

```
square x = x * x
```

An instance of the expression “`square 4`” can safely be expanded to “`4*4`”. This mechanism, termed equational reasoning, yields both a simple mathematics for program transformation and an evaluation mechanism.

The simple lack of side effects and state remove many subtle dependencies in a program, a laudible effect in the engineering of software. However most modern functional languages build on this clean and simple base to provide a number of features that would not be possible or practical in a stateful language.

Hughes [28] suggests much of the extra expressive power comes from the provision of two new forms of program “glue”: higher order functions and lazy evaluation. In addition, most modern languages support automatic type inference and checking based on the Hindley-Milner system that transparently supports polymorphic functions and high level data types. These features greatly improve the modularity of a program and code reuse.

A good overview of functional languages and research topics is given by Hudak [27]. Bird and Wadler [4] and Holyer [23] present introductory texts.

### 2.3.1 Higher Order Functions

A higher order function manipulates other functions passed in as input parameters. One standard example is “`map`”, which takes a function of one argument and a list, and returns the corresponding list where each element has the given function applied to it (See Figure 2.1).

This is more powerful than the procedural concept of pointers to functions because a higher order function can generate and return new functions on the fly. For example, the function “`map (2*)`” is a new function that doubles all the elements in a list of numbers. It is very easy to combine toolkit functions in this way to solve specific one-off problems. In contrast, much of the shared code has to be repeated for similar but not identical cases in a procedural language.

An important auxiliary notation is the concept of curried functions and partial application. Because a function applied to one argument can be a function itself, it is possible

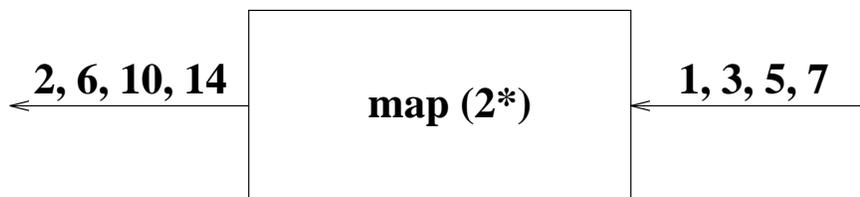


Figure 2.1: The map function

to write a function call “ $f\ x\ y\ z$ ” as “ $((f\ x)\ y)\ z$ ”. We say that “ $(f\ x)$ ” is a **partial application** of the function  $f$  that requires two further arguments. The expression “ $(2*)$ ” above is a shorthand notation for the partial application “ $(*)\ 2$ ”.

### 2.3.2 Lazy evaluation

At any point in the evaluation of an expression, there are typically several subexpressions that could be evaluated. In a stateful language such as LISP, the results of these expressions will depend on the order of evaluation. Therefore a “function” will only behave correctly when the expressions are evaluated in certain orders, typically the order in which they are given.

However in a functional language these subexpressions are guaranteed to be independent, and can be executed safely in any order, even in parallel if required. The only ordering forced is functional dependency, where an expression explicitly requires an input or shared value to continue its computation.

This allows a call-by-need, or lazy, evaluation strategy, where function arguments are evaluated only when (or if) they are required. In addition, functional languages support lazy data, where only the required parts of a structured data value are constructed. In contrast, most procedural languages implement a call-by-value scheme, where function arguments are evaluated before the function is called.

Lazy evaluation was first introduced to allow purely functional handling of I/O, about which more details will follow. It also provides an important modularisation tool. A programmer can define an abstraction that uses a large, possibly infinite, data set, and be confident that a client function will construct only the sections that it requires for its own computation. A simple example is the definition “**primes**” given below.

```

primes
  = sieve [2..]
  where
    sieve [] = []
    sieve (n:ns) = n : sieve [m | m<-ns, m `mod` n /= 0]

```

Hughes [28] expressed this by suggesting that lazy evaluation allows the separation of a program's logic from its control.

### 2.3.3 Type Checking

Many procedural languages claim to be strongly typed. In a language such as Pascal, the programmer declares the type of an object or function at its definition. The compiler checks that each application of the object is consistent with this definition. Unfortunately this involves a large contribution from the programmer, and it can be quite difficult for the compiler to pick up incorrect declarations. The primitive data types involved make it very difficult to check that a sequence of machine words is manipulated in a way according to their type definitions.

The Hindley-Milner type checking system can determine the type of all objects in a functional system at compile time. Type signatures are reduced to an auxiliary, documentary rôle, as any program that compiles is known to be type correct. It is surprising just how many programs will run correctly the first time that they pass the type checker.

Hindley-Milner gives us even more than this. The **most general** type of an function or object is derived. Consistent representation of structured objects allows us to introduce a polymorphic type system. At its simplest level this means that we can use a single utility function on a wide range of possible input, where most procedural languages would require a separate utility function for each type to be supported. For example:

```

reverse :: [a] -> [a]
reverse [ 1 , 2,  3  ] ==> [ 3 , 2 , 1  ]
reverse [ 'a', 'b', 'c' ] ==> [ 'c', 'b', 'a' ]

```

Modern procedural languages such as Ada support simple forms of polymorphism. However, the support in modern functional languages is particularly clean and well integrated, requiring very little intervention on the part of the programmer.

## 2.4 A Brief History

Functional notation developed from the theoretical foundations of the Lambda Calculus [8, 3] and ISWIM [42]. McCarthy's LISP [45] encouraged functional style, and more recent variants such as Scheme have a purely functional core. The functional ideal was popularised by John Backus in his 1978 Turing award lecture [2] (ironically given for his work on Fortran).

During the 1980's a wide range of functional notations were developed (e.g. ML, Hope, Orwell). Most notable was David Turner's series of languages, KRC and SASL culminating in Miranda [63, 64, 65]. These languages had very different syntax, but offered similar features.

In an attempt to resolve this unfortunate state of affairs, a standard language, Haskell, was developed by committee during the early 1990's. Haskell adopts most of the features introduced by these earlier languages, but adds the technical innovation of type classes [14] for the systematic treatment of overloaded functions. The language definition includes an extensive suite of Prelude modules.

## 2.5 An Overview of Haskell

Haskell is designed to be a powerful and complete language with support for large projects. The language definition is given by [26], and a more gentle introduction to the language can be found in [24]. Four genuine compilers are currently under development. Two of these are themselves written in Haskell and generate C code as a form of portable assembler. A (large) subset of the language is adopted by the Gofer interpreter [29], which allows quick development with fast turnaround. The anonymous FTP site `ftp.dcs.glasgow.ac.uk` contains copies of all these tools in the directory `/pub/haskell`.

The language syntax is largely derived from the Miranda series of languages. In particular, list comprehensions, extensive pattern matching facilities and the "offside rule" method of laying out nested scope are adopted. Most of these features are strictly "syntactic sugar", in that a program can be transformed into an equivalent program that does not contain these features. They do however greatly aid the structuring and readability of functional code.

## 2.6 Advantages and Disadvantages of the functional style

### Advantages

1. Referential transparency makes functional programs simple and well behaved. This is very important as software engineering is overloaded with complexity.
2. The black box nature of pure functions and enhanced modularity through H.O.Fs and lazy evaluation encourages very modular design and coding styles.
3. The high degree of composability and the availability of a large number of standard toolkit functions produce very compact and readable code. A procedural program is typically five to ten times longer than its functional counterpart.

### Disadvantages

Traditionally, functional languages are viewed as having the following shortcomings.

1. Compiler technology can be far more involved for functional languages, as there is no direct mapping to the underlying architecture. Many involved and esoteric schemes have been suggested, but it is only comparatively recently that the subject of functional implementation has matured.
2. While the run time **behaviour** of a functional program is deterministic and therefore predictable, its **performance** is not. High heap throughput with lazy evaluation “holding onto” many expressions at any one time frequently lead to large space leaks in the course of a computation. Intermittent garbage collection can lead to glitches in data throughput, especially noticeable in real time or interactive applications.
3. Functional languages are generally perceived to be weak at bulk data manipulation because of the lack of update-in-place mechanisms. A solution to this problem has been a long time in the making, with much investigation into various types of “single threaded analysis” as a compiler optimisation to find data structures that can be safely overwritten rather than copied when updating.

4. Currently functional implementations suffer from poor availability, and low credibility outside the academic establishment. Hopefully this situation will change soon. A number of powerful, but freely available, optimising Haskell compilers are under development at Glasgow, Chalmers, Yale and Bristol.
5. I/O behaviour causes a number of different problems. This is the subject of the next chapter, and indeed the remainder of this thesis.

A number of these problems have now been addressed. In particular, functional compiler technology has matured rather nicely in recent years. Compilers are now emerging that generate code surprising similar to the equivalent procedural code, and with very good efficiency. A very important optimisation is the removal of “unnecessary” laziness through strictness analysis. Unfortunately while laziness is a very powerful tool, it is inherently expensive to implement on a Von Neumann computer.

It is interesting to note that once a basic compiler technology, such as the G machine, has been invented, it is very easy to add a large number of optimisation passes, each addressing a particular need. The lack of shared data dependencies mean that such optimisations can be implemented as program transformations. This is a lot easier, and more reliable, than manipulations to a complex stream of primitive instructions. In contrast, procedural optimising compilers are frequently bug-ridden and unreliable.

Runciman and Wakeling [54] have demonstrated that, as in the procedural case, most of a process’s execution time is spent in key toolkit functions that should be hand optimised. The complex program trace caused by lazy evaluation can make it difficult to determine which functions, and more particularly which instances of those functions, are concerned. They suggest the solution is that a program should be written with little regard for low level efficiency and then hand optimised using information provided by a powerful graphical profiling package. Of course, the prime source of inefficiency remains the choice of poor algorithms rather than coding style.

It is now generally accepted that the solution to bulk data manipulation lies in the use of an abstract data type accessed using higher order functions to force single threaded access. These **monadic** ADTs were introduced in category theory by Moggi [47] and were championed in the functional arena by Philip Wadler [66, 67]. Monads have been the

biggest innovation in the functional universe of the 1990's, and their advocates think that they are the best thing since sliced bread. However, monadic style is not used in this thesis, as our model of concurrency is best expressed in terms of stream interactions.

## Chapter 3

# Operating Systems and Functional Languages

Most functional languages have a guest status in a procedural operating system. We can either link each functional process with a procedural subsystem controlled through a Dialogue type, such as that used by the Haskell language [26], or call operating system services directly using monadic C Call [39]. The objective in both cases is to transform a well behaved functional component into a stateful and side effecting procedural one.

In a operating system such as Unix, a single such process can be forced to be reasonably well behaved. For example, a purely functional “emulator” can be written that describes the effect of the entire system [25]. However, given a network of communicating processes, any procedural components can introduce non-functional behaviour.

Various attempts have been made to overcome this problem by creating completely functional operating systems, where all of the system software (but not the kernel) is written in a purely functional style, preferably completely in a single functional language.

Rather unfortunately, most of these projects have worked from the premise that functional operating systems should have very similar services and structure to their procedural counterparts. Consequently, these projects have adopted the design and a great deal of the conventional philosophy of procedural systems.

The most damaging of these philosophies is that **all** concurrent systems require some form of nondeterministic merge operator, similar to the `select()` system call in Unix. The majority of functional operating system research has concentrated on various ways of introducing such an operator while preserving the all important property of referential trans-

parency as far as possible.

It is our thesis that many of these problems can be avoided by taking an alternate view of the situation. However, let us first consider what has gone before.

This chapter splits into the following two sections:

- First we consider how we might manage a system that consists of a large number of communicating processes, rather than a single monolithic calculation.
- Then we consider the various ways that functional code can interface to the virtual hardware. This can involve both low level device drivers, and interfacing to services in the host operating system.

### 3.1 Nondeterminism

One of the major benefits of functional programming languages is that they demonstrate deterministic behaviour — a particular set of input values to a function guarantees a particular set of output values.

However, most operating systems allow arbitrary networks of communication between a set of concurrent processes. If two separate processes are generating input to a third, we do not know which of the two processes will generate the next input event. In the worse case, one process may generate a large amount of output while the other generates none at all. If the receiver is suspended waiting for input from the wrong process, deadlock ensues (See figure 3.1).

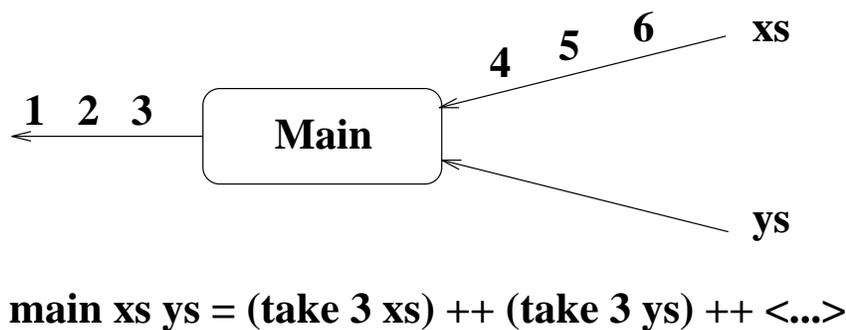


Figure 3.1: Deterministic Dead lock

The solution is to introduce some form of nondeterministic operator that can select the “first” item to arrive on a set of input channels. Most of these primitives are variations on each other. For the purpose of this section we use McCarthy’s “amb” operator [44], simply because it was the first to be introduced.

Informally, we consider “a amb b” to be an expression that arbitrarily returns “a” or “b”, typically the one which is available first. “amb” is an “angelic” operator [57] — given two arguments, only one of which terminates, “amb” will return the value of the converging computation.

In general, this requires amb to be a source of concurrency, starting separate threads of execution for the expressions and explicitly terminating the slower computation. Such an operator can obviously be used to overcome the deadlock situation outlined above.<sup>1</sup>

While this obviously increases the expressive power of a functional language, the problem with amb is that it destroys the all-important property of referential transparency (Clinger [9]). The following equality is no longer guaranteed to hold:

$$(4 \text{ amb } 5) = (4 \text{ amb } 5)$$

This means that we can no longer apply equational reasoning to manipulate programs.

$$(x + x) \text{ where } x = (4 \text{ amb } 5)$$

can no longer be replaced by:

$$(4 \text{ amb } 5) + (4 \text{ amb } 5)$$

as while the former can generate the values 8 and 10, the latter can also generate the value 9. This is symptomatic of the kind of race condition and resource contention that can exist in real operating systems.

Nondeterminism is generally poorly understood. Indeed nondeterminism and concurrency are often confused. Söndergaard and Sestoft [57] consider 3 separate variables that lead to 12 forms of nondeterministic system, each with subtly different semantics. Main and Benson [43] consider three distinct forms of nondeterminism and present semantics based on semiring models, a generalisation of vector spaces in linear algebra.

---

<sup>1</sup>However Note: This does not guarantee fairness. A series of “a amb b” expressions may continue to return “a” without “b” getting a look in.

### 3.1.1 Approach suggested by Henderson

Peter Henderson was one of the first to carry out research into functional operating systems. His 1980 paper “Purely Functional Operating Systems” [18] discussed a number of solutions based on process networks, using laziness and the stream model of I/O.

In its simplest presentation, Henderson’s system reserves the names “keyboard” and “screen” to define input and output streams for an interactive program. Using Haskell syntax in place of the original LISP:

```
screen
  = double (keyboard)
  where double x = 2 * head x : double (tail x)
```

This program takes a list of integers as input and prints out each value doubled. We presume that the terminal works in a line based mode, where each input line is echoed to the terminal, and can be edited before being entered as an input to the program. Lazy evaluation ensures that each entry immediately generates an answer with the correct interleaving of input and output.

A simple data type is provided that allows clients to maintain a simple flat filestore database, implemented as an association list held in heap memory:

```
put : Filename -> File -> DataBase -> DataBase
get : Filename          -> DataBase -> File
```

We can easily build a Dialogue style interface to such a datatype:

```
data Request = Put Filename File | Get FileName
data Reponse = Done              | Got File

dbf : [Request] -> [Response]

dbf reqs = dbf' reqs <initialFileStore>

dbf' [] _ = []
dbf' (req:rest) db = response : dbf' rest db'
  where
    (reponse, db') = dbstep req db

dbstep : Request          -> DataBase -> (Response, DataBase)
dbstep (Put name file) db      = (Done, put name file, db)
dbstep (Get name)          db      = (Got (get name db), db)
```

With a stream based interface, the filestore can be separated out as a separate process that a client must communicate with in order to access files.

Henderson takes the (now rather outdated)<sup>2</sup> view that a multiuser system consists of a central CPU with several glass teletype terminals. Each terminal offers a keyboard and a screen and allows its user to run a single foreground process. These processes are time-sliced on the CPU.

If two user processes both wish to access the filestore process, then the filestore requires some kind of nondeterministic operator, as discussed in the last section. The operator that Henderson chooses to use is a merge operation called “*interleave*”. This takes two input streams as arguments and outputs a single, merged, stream on a first come, first served basis. It has the advantage over “*amb*” in being a fair choice operation. Once an input message has been dispatched, we are guaranteed that it will, eventually, be processed by the first come, first served, nature of “*interleave*”.

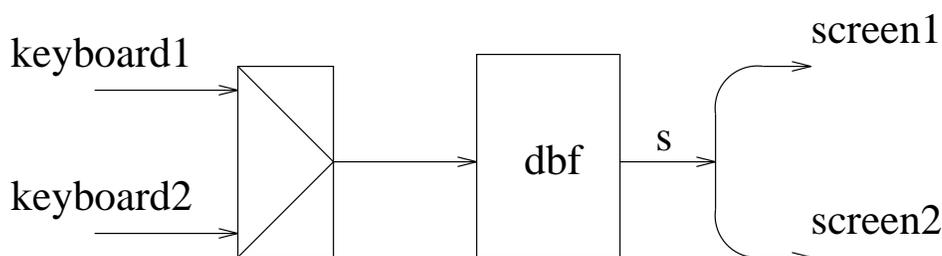


Figure 3.2: Process network for sys1

Figure 3.2 shows a simplified version of the following two user, one filesystem network:

```

interleave :: [a] -> [a] -> [a]

sys1 (keyboard1, keyboard2)
  = (screen1, screen2)
  where
    screen1 = <prettyprint> s
    screen2 = <prettyprint> s

    s      = dbf(interleave k1 k2)
  
```

---

<sup>2</sup>This outdated model is actually rather important to the design of our own system intended for rather more modern workstation environments.

```
(k1, k2) = (<parse> keyboard1, <parse> keyboard2)
```

This is not quite correct, as each screen reflects the responses to commands dispatched from the other keyboard, as well as its own. This problem is solved by tagging messages sent to and from the database.

If “ $x$ ” is a sequence of items, then “ $\text{tag } t \ x$ ” is the same list where each element is tagged with the atom “ $t$ ”. The function “ $\text{untag } t \ x$ ” takes a tagged list containing messages to different destinations, and filters out those with tag “ $t$ ”, removing the tag in the process.

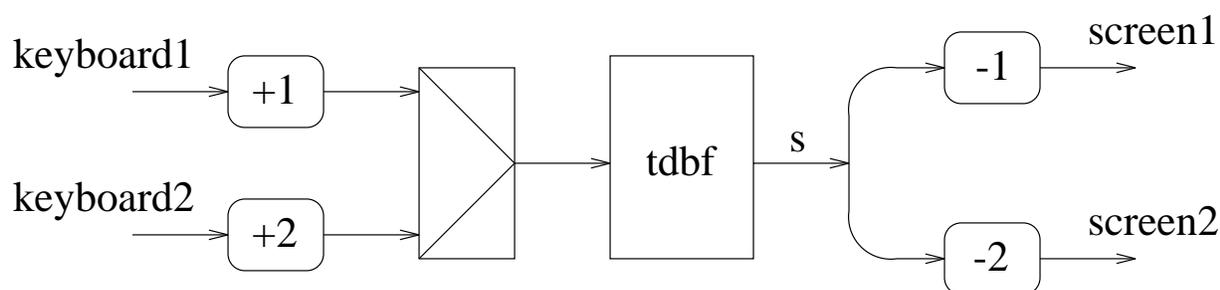


Figure 3.3: Process network for sys2

Figure 3.3 shows the tagged version, “ $\text{sys2}$ ”. “ $+t$ ” and “ $-t$ ” respectively denote the tagging and untagging operations.

```
interleave :: [a] -> [a] -> [a]
```

```
sys2 (keyboard1, keyboard2)
```

```
  = (screen1, screen2)
```

```
  where
```

```
  screen1 = <prettyprint> untag 1 s
```

```
  screen2 = <prettyprint> untag 2 s
```

```
  s      = dbf(interleave (tag 1 k1)
                (tag 2 k2))
```

```
(k1, k2) = (<parse> keyboard1, <parse> keyboard2)
```

In this way, each user is unaware of the others, other than the fact that the filestore may change unexpectedly. A number of interesting and useful operating systems can

be developed. The author suggests ways of introducing multiple file stores and editing processes (Figure 3.4).

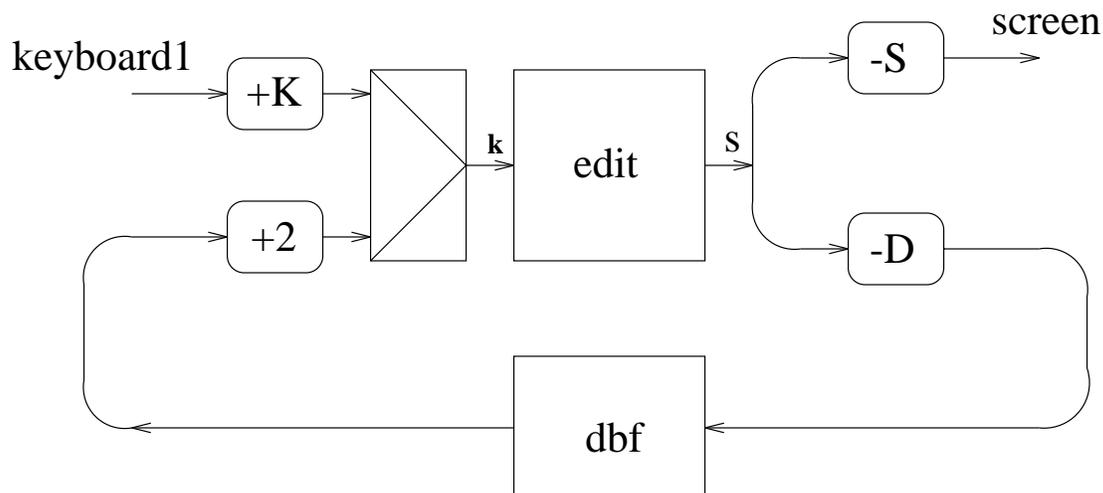


Figure 3.4: Integrating an editor into the system

`interleave` is an easier function than `amb` to implement when the arguments are I/O channels. Interprocess communications primitives implicitly introduce concurrency – `interleave` just has to wait for input to arrive. As such it can be considered to be a data driven operation in a demand driven environment. Looking at things the other way around, we believe that data driven thinking **forces** the use of a nondeterministic operator.

Henderson suggests indiscriminate use of “`interleave`” should not be encouraged because of the damaging effects of nondeterminism on referential transparency. He does however conclude: “Since we are trying to model nondeterministic behaviour it may be that a nondeterministic primitive is necessary”.

### 3.1.2 Later Refinements to the model by Henderson and Jones

Henderson’s original paper dealt with simple static networks of processes. He later went on to invent a graphical description of such networks [20] and described its equivalence to CSP notation in [19].

Together with Simon Jones [21] he discussed possible structures for “shell” programs: processes that involve and coordinate other processes. At about the same time, Shultis [56] developed a similar shell using Backus’s algebraic FP notation. Because FP is a strict,

data driven, functional language, multiple input and output channels were possible for user processes. Much later, in 1987, McDonald [46] reversed the process, designing a Unix shell with an (impure) functional “look and feel”.

Jones took up the mantle, and described the design and implementation of a complete multitasking, single user, operating system based on communicating processes [32, 31]. McCarthy’s “amb” primitive was used to implement “merge”.

The wide availability of Lispkit LISP at the time on a wide range of computers did a great deal to popularise the concept of a purely functional operating system using multiple lazy processes. The eventual system saw frequent service, and was “in its own limited way, . . . quite satisfying to use”.

The author concludes that while network structuring is an important modularisation, its indiscriminate use can lead to difficult “plumbing” problems in structuring the functional code that represents the network graph. The use of oracles to avoid instances of “interleave” is suggested and each of the original examples in [18] is rewritten in a completely deterministic style. The author concedes that this can greatly exacerbate the existing plumbing problems.

The most interesting contributions of this work is the introduction of a very simple Unix style shell with primitive I/O redirection features, and a discussion of various possible process network topologies. An operating system consisting of an editor, compiler, filesystem and pretty printer is presented (See figure 3.5)

The author suggests that each of these components can be moved into a separate process, running across a serial link to a remote processor if required. This is a powerful modularisation technique, as the behaviour of each component is defined by its I/O characteristics. Processes can therefore be replaced with more powerful subsystems that inherit the previous behaviour. They can also be used to implement procedural device drivers with an obvious [Request] -> [Response] style functional interface.

It is, however, conceded that attempts to modularise individual processes, such as the editor, in this way were not successful. This was due to the high degree of internal complexity. The author concludes that arbitrary networks of communicating processes become unmanageable in this scheme, and presents a fledgling ring network design.

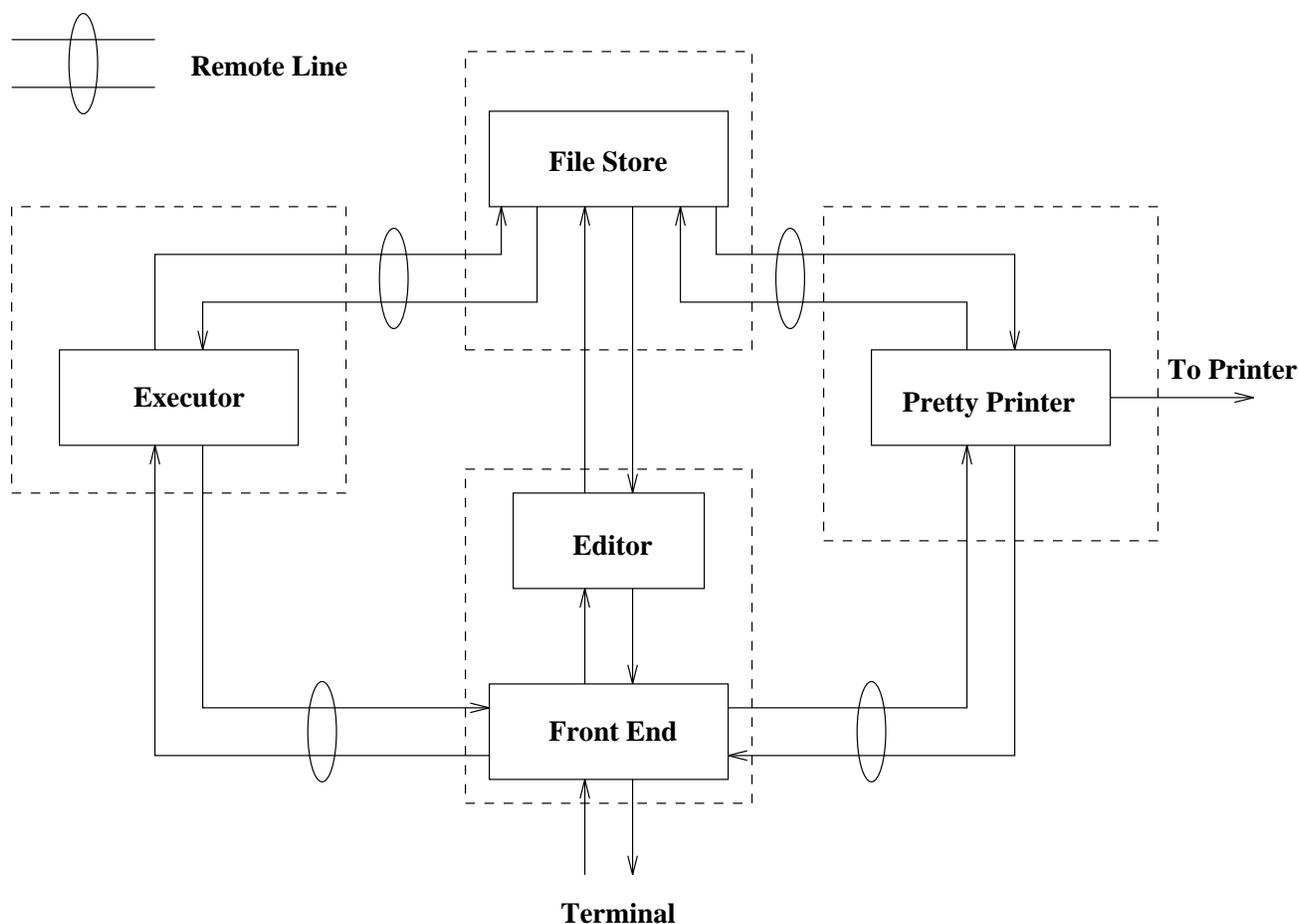


Figure 3.5: Simple single user model

### 3.1.3 Later Refinements to the model by Jones and Sinclair

Later in [33], Jones and Sinclair present a design for a complete single user multitasking operating system, containing an editor and a compiler. This design is based on a restricted use of an `interleave` style operator in a configurable message routing ring. The paper presents information about further experiments to write low level devices drivers with strong functional interfaces.

The authors demonstrate that programs can be composed into pipelines and networks in the same way as Henderson and Jones's earlier work. They further suggest that tagging allows each process to have a number of logical input and output channels, multiplexed onto single physical input and output channels. This is important: for reasons that we

shall see shortly, standard functional language implementations can only support a single output channel.

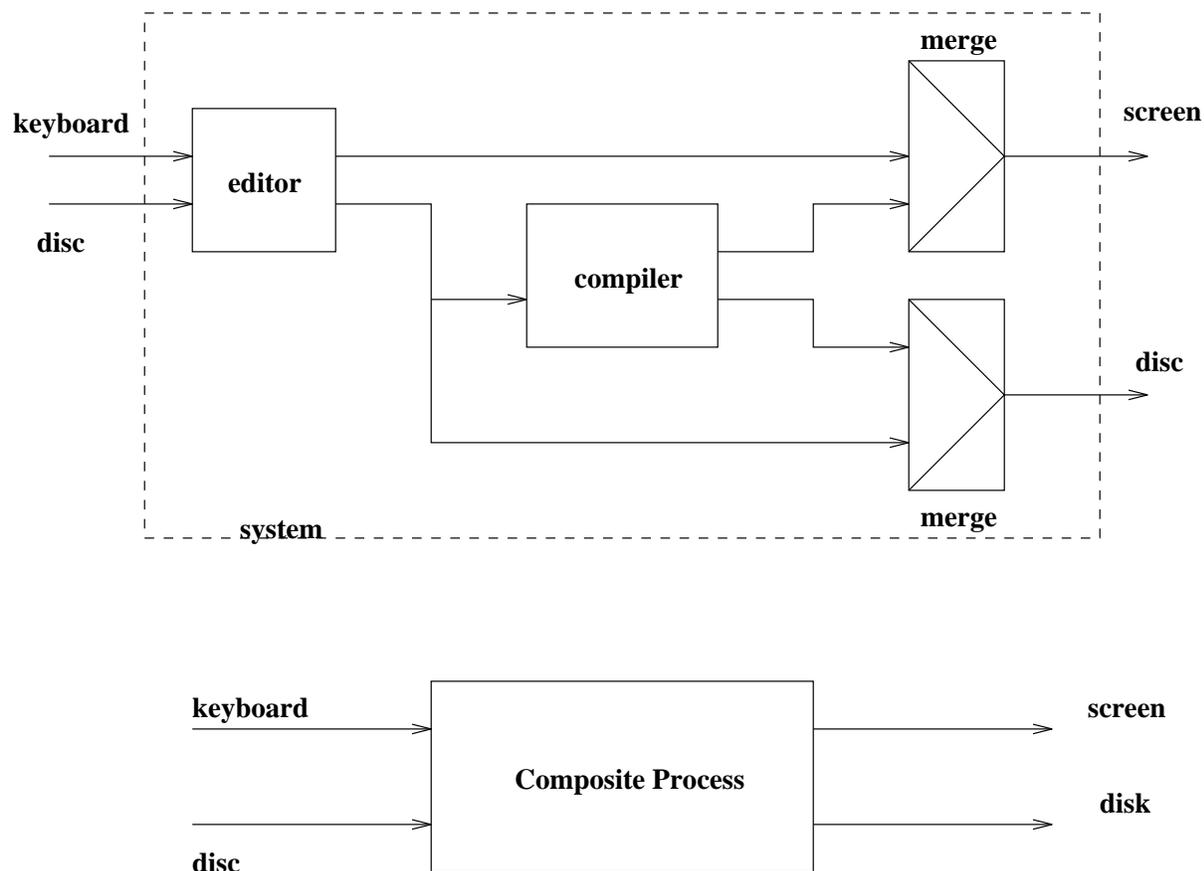


Figure 3.6: Functional Dependency

It is suggested that networks of such processes allow interesting functional dependencies to be formed. For example, the output from a compiler could be sent both to the filestore, and through a compiler to automatically generate a compiled object file and list of error messages (Figure 3.6). We can consider this composite network to be a single process in its own right.

Rather than using a primitive interleave operation, their “merge” operation is defined using overlapping pattern definitions. This is similar in concept to McCarthy’s `amb`, although restricting to stream inputs again allows various optimisations.

```
merge :: [a] -> [a] -> [a]
```

```

merge (x!xs) ys      = x ! merge xs ys
merge xs      (y!ys) = y ! merge xs ys

merge []      ys      = ys
merge xs      []      = xs

```

In the event that several patterns fire, the one that matches the first available input is used. Merge is a nondeterministic process, although what we actually have is implicit time determinism, as the choice is time dependent rather than purely random. ‘!’ is “head strict cons”. This is a similar operator to the normal list construction operator ‘:’, but it forces hyperstrict evaluation of its first argument. This is important in a network of communicating processes, where each message to be passed must have a simple concrete representation that can be passed between machines.

**Amb** can be written in this notation as:

```
a amb b = hd (merge (a![]) (b![]))
```

Use of the `merge` operation is restricted to the operating system programmer. The design uses a dynamically reconfigurable network in the shape of a ring, with all user processes and devices attached as shown in Figure 3.7.

Each process is allocated a unique process identifier of type `pid`, which is used for a variety of purposes including system calls and message passing. The process manager, `pm`, is responsible for allocating new `pids`, adding and removing processes from the ring. Each message in the system is tagged with the `pid` of both the sender and the receiver. The packages sent around the ring are therefore of type `(pid, pid, mesg)`. The system attaches source tags and strips destination tags from messages before they arrive at their destination. Consequently user processes have type:

```
process = stream(pid,mesg) -> stream(pid,mesg)
```

A privileged shell process manipulates the ring by sending special messages to the program manager.

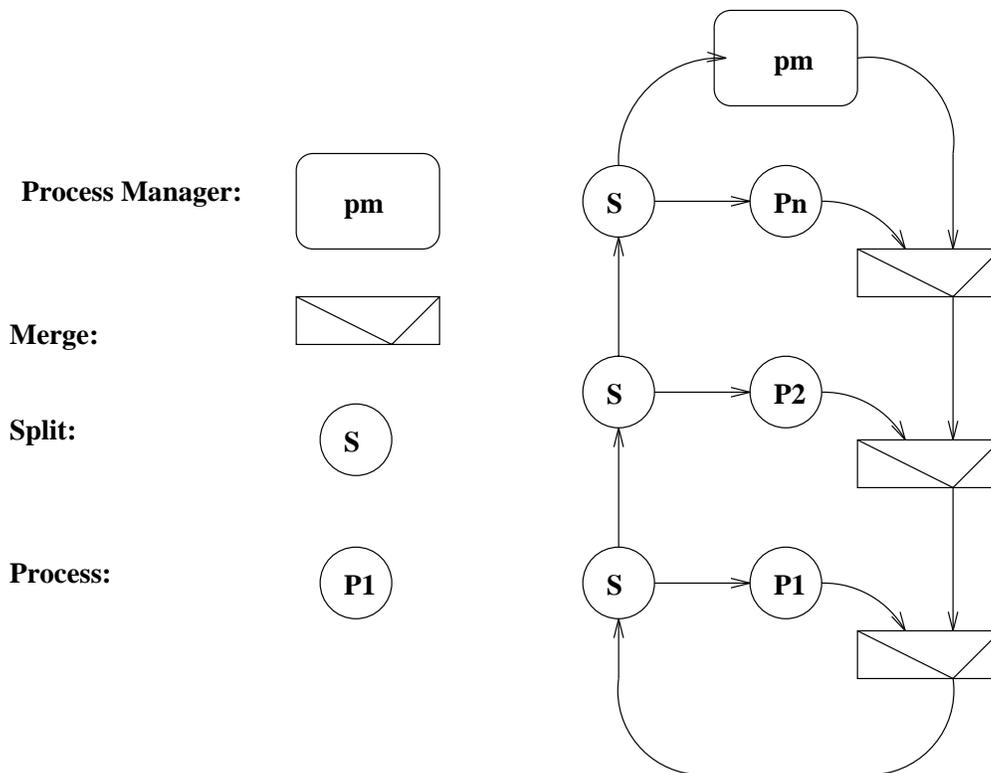


Figure 3.7: Dynamically configurable process network

### 3.1.4 Stoye's Sorting Office

Although developed before Jones and Sinclair's work, Stoye's sorting office [58] can be considered to be a natural refinement of it. Rather than have a ring of simple merge and split points, all messages are sent to a single point (the sorting office), where they are processed and sent on to their requested destinations. A first come, first served philosophy guarantees fairness. A special process management process creates and discards processes in the system.

Each process is considered to be under constant demand from the sorting office. This causes concurrent operation as each process proceeds as far as it can until suspending on input. We can consider the sorting office itself, and all the client processes to be data driven. Indeed, a strict rather than lazy functional language can be used.

While at first sight, this may seem more complex than the previous scheme, it maps well onto existing task and communication primitives on standard equipment such as Unix

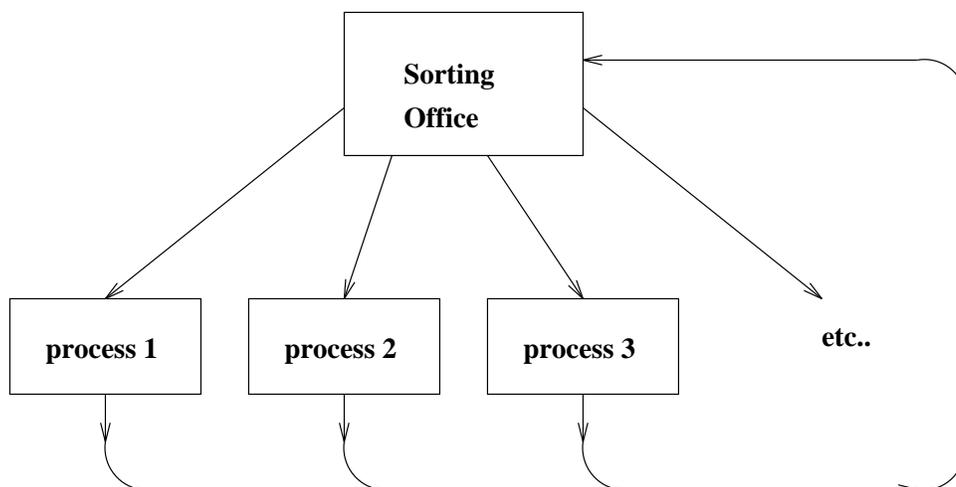


Figure 3.8: Stoye's Sorting Office

workstations. TCP/IP is designed for point to point virtual circuit connections rather than aesthetically pleasing rings.

If we consider the sorting office to be part of the virtual hardware that our “operating system” is functioning on, then the whole functional level of our system can be considered to be purely deterministic and untainted. This might be considered as avoiding the problem altogether, as the sorting office itself cannot be written functionally. However it is generally accepted that the single instance of a merge operation offers the cleanest solution to introducing nondeterminism into the functional context.

### 3.1.5 Sorting Office as adapted by Cuppitt and Turner

In [10], Cuppitt uses a **deterministic**, merge-free, sorting office to construct a Unix emulation system within a slightly adapted version of the Miranda interpreter. Features include:

- A purely functional sorting office that supplies communication between a set of user and support processes.
- Block based disk “device” implemented using fixed length lists. A hierarchical file system can be mounted onto such a device.

- Multiple stream based terminal devices. The implementation uses non-blocking keyboard input with hiaton characters to allow concurrent logons from different terminals. This is not a very efficient, or scalable mechanism.
- An event manager. Processes indicate interest in particular events, especially terminal interrupt characters.
- A simple (very Unix-like) shell, a number of small utilities mostly relating to filesystem manipulation and a full screen editor.

This project was largely concerned with demonstrating the **feasibility** of a Unix style operating system, rather than building a useful system. As such it is implemented as a single monolithic Miranda process of some 14,000 lines of code. A simpleminded, but highly inefficient, implementation of “merge” (based on busy waiting) is suggested but not implemented.

The lack of a merge operation in the extended Miranda interpreter caused a number of problems in the design and implementation. User processes must be very well behaved. Each receiving process must send out a “WAIT” message to announce the need for each input item, a primitive form of flow control. Each process must acknowledge the receipt of each message to the sender and all communication is strictly synchronous. Cuppitt suggests that such a system quickly becomes unmanageable unless a continuation style is adopted to ensure correct pairing of the input and output messages.

We have cooperative, rather than preemptive multitasking. This particular implementation requires a great deal of cooperation from the author of each user process. Indeed late in the implementation process, the author discovered that this model could not handle interrupts without causing deadlock, unless interrupt messages were given privileged status in the system.

The system is not secure: the kernel has no control over the space and time behaviour of each process. In addition, there is no provision for exception handling. For example, a divide by zero run time error in any process will cause the entire system to abort.

Turner [61] offers a number of refinements to this model. In particular, a “wrapper” mechanism that allows separate typing of each process’s message stream from the sorting office. Process creation is based on the Unix `fork()` and `exec()` pair of systems calls, splitting out each process into its own address space. Interprocess communication is restricted to hyperstrict messages, that can easily be converted to a network transparent representation.

### 3.1.6 Nondeterminism as Data to a program

Burton [5, 6] suggests an approach that preserves referential transparency in the presence of nondeterminism. A program takes an extra argument that is a lazy binary tree of values. This dynamically records nondeterministic choice – each collection of choices is a particular walk through the tree. In this way, a program called twice with the same choice tree will produce the same results.

Strictly speaking referential transparency is restored as nondeterministic choice is placed in the data, rather than the functions. It is not however a clean or elegant solution, as a large data structure has to be passed around to every point where a “*interleave*” style operation is to take place.

### 3.1.7 Time Stamps

We have seen that “*interleave*” style operators are nondeterministic because time dependences involved mean that two instances of the operator with the same input data do not guarantee the same result.

However if magic “time values” (monotonically increasing numbers) are generated by a special nondeterministic operator in the system and stamped onto messages as they pass, we can write a purely deterministic version of *interleave*, that orders messages according to their time stamps. In essence this is a variation on Burton’s approach of recording choice. Harrison [15] uses this concept in his programming language RUTH.

The problem with this approach is that, as well as adding complexity with support code that adds, manipulates and removes time stamps from messages, we suffer from flow problems. A message can only be forwarded onto the output stream if there is a “later” message on the other input stream. If no message is present, *interleave* must wait for an input event on that channel. No input message may be forthcoming, in which case we must timeout using special “hiaton” messages.

As well as adding to the complexity of the support code, such **busy waiting** is, by its very nature, wasteful and expensive. It seems unlikely that such a system would scale well.

### 3.1.8 Conclusions

Work on purely functional operating systems has been rather quiet in recent years. This is probably because of the perceived need for nondeterminism – in short no better solution than the sorting office is likely to emerge. The focus has switched rather onto ways of interfacing to existing operating systems in cleaner and more powerful ways.

## 3.2 Interfacing

Regardless of its efficiency, functional code is a very elegant notation for expressing simple, self-contained computation. Unfortunately, this is far less true of interactive applications, where a program must interface to some external entity, such as an operating system. This is largely because of the stateless nature of functional programs.

I/O behaviour is inherently stateful, as computers interact with peripherals such as displays and disk drives by changing bit patterns in memory. Procedural languages manipulate these machine words directly, and this approach was mimicked in early functional languages that included impure, side effecting, operations.

Three major strategies have been suggested for I/O interfacing in pure languages. The common theme is to separate out the stateful I/O functions and place this subsystem under rigorous control of a purely functional core.

### 3.2.1 Dialogue based I/O

Dialogue based I/O came first, and was largely responsible for the popularisation of lazy evaluation. The premise is quite simple: an interactive program is an expression that is a lazy list. The elements of the list are requests for I/O operations to be carried out in the given order. Some I/O requests will generate response values that are returned to the program by a symmetrical input list (See figure 3.9).

This is intuitively “the wrong way around” for the functional programmer as it is the view from the operating system.

In its simplest form, the Request and Response types can be simple characters: this is general enough to cover simple interactive applications that map keyboard input to screen output. However the Haskell standard defines a wider range of operations, sufficient

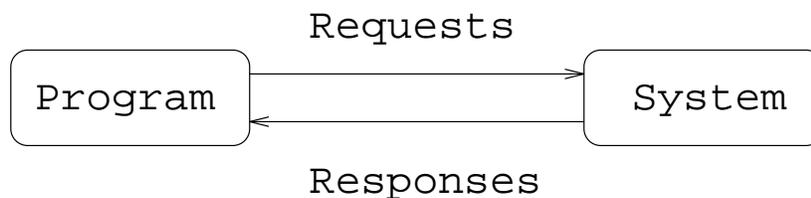


Figure 3.9: Dialogue Mechanism

```

type Dialogue = [Response] -> [Request]
main :: Dialogue

data Request =
  ReadFile    String
| WriteFile   String String
| AppendFile  String String
| ReadChan    String
| AppendChan  String String
| Echo        Bool

data Response = Success
              | Str String | Failure IOError

data IOError =
  WriteError String | ReadError String
| SearchError String
| FormatError  String
| OtherError  String
  
```

Figure 3.10: Haskell I/O Dialogue Data Types

for many simple, sequential, text based, applications. The various possible requests (See Figure 3.10) are based on the Unix I/O model with files and streams available.

While this set of operations is sufficient for simple applications, the approach suffers from the following problems:

- The code style involved is very messy because of the complex plumbing involved [33]. This is because associated request and response values are not logically paired in any fashion. It is up to the client programmer to structure his or her code. This normally involves a complex recursive loop that enumerates all the possible combinations of program state and possible input. Exception handling must be done explicitly.
- A consequence of this is that the programmer must be very careful to ensure that the input and output lists are interleaved correctly on the terminal [59]. A common problem, caused by the fact that pattern matching forces evaluation, is to have a

program request an input item before a prompt for it has been generated.

The Haskell committee introduced the lazy pattern ( $\sim$ ) to try and overcome some of the more grotesque structuring involved.

- The Dialogue style is not composable: you cannot take two small dialogues and combine them to form a larger whole with the desired effect. This is because any given Dialogue contains a great deal of internal state that is not visible at the [Request]  $\rightarrow$  [Response] boundary. Each Dialogue expects to interact with an external entity of known behaviour, not other dialogues that manipulate its well synchronised and behaved I/O lists.

O'Donnell [49] and Dwelly [11] introduce “Dialogue Combinators”, higher order functions that structure Dialogues as networks of communicating processes. While this can help with the layout and structuring of a program, Dwelly demonstrates that such networks have difficulty coping with dynamic user environments such as window and menu based systems.

- The list of I/O requests is fixed in the language definition. A functional programmer cannot add extra requests to cover new features such as an X windows interface or additional device drivers. Even if she could, the size of the request set would soon become unmanageable without some form of hierarchical structure to group related requests.

### 3.2.2 Continuation based I/O

A second school of thought tried to avoid expensive lazy evaluation and improve the request-response association using higher order functions to facilitate I/O “requests”. Each I/O request consists of a success and a failure continuation along with the request data. The appropriate continuation function is called with some I/O response data as input.

This is similar in concept to the procedural idea of call-back functions, used by the various X toolkits [48, 16]. We can view this as a thread of procedural execution dipping into functional territory. Nigel Perry [53] suggests a clear analogy to remote procedure call. See Figure 3.11.

We have successfully managed to delegate request-response pairing and error handling to the support system. This makes it far more pleasant to program applications with a

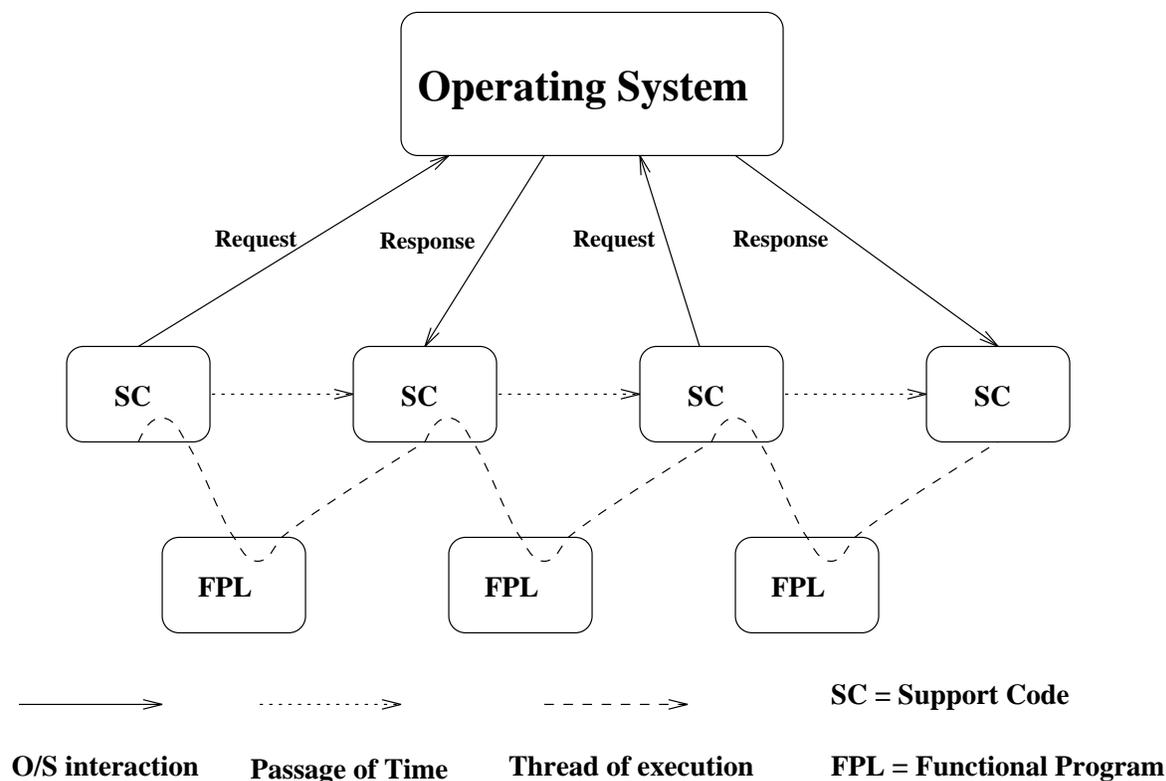


Figure 3.11: Continuation Style

wide variety of I/O behaviour using continuation rather than dialogue style. However the use of long chains of higher order continuations can be very intimidating to the novice reader and it is important to maintain strong coding discipline and style.

We have gone some way to solving the first of our three problems. However, the problems of composibility and a fixed “instruction set” remain. The lack of centralised state means that continuations cannot be directly combined, although the strict request-response pairing makes small scale grouping easier.

Continuation based I/O is probably only of historical interest, as part of the evaluation process of monadic I/O. However two influential functional operating system projects (Nebula [40] and Hope+C [52]) have involved the continuation style.

### 3.2.3 Monadic I/O

Monads are a very active research topic at the moment. As we have seen, they have been adopted from category theory to describe single threaded state using higher order functions.

From a purely I/O perspective we can view them as continuations that contain implicit state passed from function to function in the continuation chain. This state can be used to abstract out information about our world outside the I/O interface that would otherwise have to be carried explicitly through the client program. This makes the continuation functions much less stateful and therefore more composable.

The recent introduction of monadic C call interfaces in the ghc compiler [39] means that low level support libraries can be written directly in (rather stylised) Haskell. This means that the set of I/O facilities can be extended directly by the Haskell systems programmer. Whether this is easier than writing the support code directly in a more suitable procedural language is, of course, a matter for debate.

## Summary

While the programming style of the client programmer is very different in each of these schemes, the external interface is consistent, and indeed, each of the styles can be written using another as the primitive builtin system. Hudak and Sundaresh [25] demonstrate that the dialogue, continuation and their own “system” style have equal expressiveness.

Traditionally, Dialogue I/O is the primitive simply because it is the easiest to implement cleanly in a foreign (i.e. procedural) support system. For this reason the demonstration system and its intricate support code for the Gofer interpreter is based on Dialogue style in this project. This does not represent any comment on the appropriateness of the various styles for **large scale programming**.

The recent vogue for monadic style suggests that monadic I/O will predominate in the future. The availability of monadic C call and bulk data support libraries suggest that monadic I/O will be the primitive system.

# Chapter 4

## Implementing Functional Languages

Procedural languages are based on the execution of instructions that manipulate a centralised state. This corresponds directly to the underlying architecture of a conventional computer.

In contrast, functional languages are based on the evaluation of expressions, without enforcing a specific execution order. Increasingly involved and esoteric abstract machines have been proposed, but it has only been comparatively recently that functional language implementations have reached efficiency comparable to their procedural counterparts.

A full description of implementation techniques is outside the scope of this thesis. This section exists to provide an overview of the techniques involved, especially those relevant to the Gofer interpreter used by our demonstration system.

The seminal work on functional language implementation is Peyton Jones [37]. Sections of this work are elaborated in a simpler, tutorial, style in Peyton Jones and Lester [35]. Further information is presented by Field and Harrison [13]. Information about more recent developments is given in section 4.6.3.

### 4.1 Simplification

The first stage in compiling a functional program (after ensuring that the program is type correct) is to simplify the program to a core language by program transformation. Such transformations typically include:

- Replacing syntactic sugar such as list comprehension with equivalent function calls

- Replacing list and tuple notation with primitive constructors such as Cons and Nil.
- Replacing guards with a primitive test function.
- Replacing infix operators and sections with conventional function calls.
- Simplifying complex, nested, patterns to a succession of single level forms, using case statements.

## 4.2 Lambda Lifting

All lambda expressions and locally defined functions are transformed out to the top level of the program source. This transformation removes implicit dependencies on variables, defined in the surrounding environment that are not passed as parameters into a function, the so called **free variables**.

In the following,  $y$  is a free variable in the definition of  $g$ .

$$f\ x\ y = g\ x \quad \text{where} \quad g\ x = x + y$$

$y$  has been acquired from the surrounding environment  $f$ . This code can be transformed to the following pair of definitions with no free variables:

$$\begin{aligned} f\ x\ y &= g\ x\ y \\ g\ x\ y &= x + y \end{aligned}$$

A function or lambda expression that does not contain any free variables is termed a **supercombinator**. The process of converting a program into a set of supercombinators is called lambda lifting.

This is an important transformation because it is a lot simpler to execute programs that consist solely of supercombinator functions. Without such a guarantee, the evaluator has to pass around an environment of free variables that are currently active at a particular point in a program's execution. This is similar to the need for complex stack frames in procedural languages like Pascal, that allow for the nested definition of functions.

Although the early substitution based SECD machine (Landin [42], Henderson [17]) incorporated such an environment, the conventional wisdom has been to try and factor out such complexity. This view may change as the design of the STG machine [38] cleanly

integrates nested environments. However, the Gofer run time system is less advanced, and does require such a program transformation.

The lambda lifting process does not remove all local definitions. Many variables are left to describe shared expressions in a function's definition. This does not cause problems because each function remains self contained and self-sufficient.

### 4.3 Manual Evaluation

Evaluation in functional languages is based on the substitution of expressions, rather than in the execution of an instruction stream. This is complicated by the need to support lazy evaluation. Let us see what this entails. Assume the following definitions:

```
double x = x + x
square x = x * x
```

Equational reasoning tells us that wherever we see an instance of the left side of one of these definitions, we can replace it with the right side. For example:

```
? double (square 4)
--> double (4 * 4)
```

This process is called reduction, and the expressions involved are termed **reducible expressions** or **redexes** for short. Primitive operators such as (\*) and (+) are handled as special builtin functions. Intuitively, the expression (2 \* 3) can be reduced to the value 6. However, the expression (2 \* id 3) cannot be reduced until the (id 3) subexpression is reduced to a number. We can view user defined functions as macros that must be expanded before primitive functions can evaluate.

Different evaluation orders are feasible when an expression contains more than one redex:

```
? double ( (square 4) + 5)
--> double ((4 * 4) + 5 )
--> double (16 + 5)
--> double 21
--> 42
```

or

```

? double ( (square 4) + 5)

--> ( (square 4) + 5) + ( (square 4) + 5)
--> ( (4 * 4  ) + 5) + ( (square 4) + 5)
--> ( (4 * 4  ) + 5) + ( (4 * 4  ) + 5)
--> 42

```

The first of these two schemes involved evaluating the innermost redexes first, working left to right when necessary. This evaluation strategy is termed applicative order reduction and has similar semantics to call-by-value in the imperative world.

The second example shows simple normal order evaluation, where the outermost redex is evaluated first. This is important to us because the parameters to a function are only evaluated as they are required. This is fundamental to the concept of lazy evaluation.

## Normal Forms

An expression that contains no remaining redexes is said to be in **normal form**. At this point, no further evaluation can take place. Not all expressions can be reduced to a normal form. Such expressions are said to have a semantic value of  $\perp$ , pronounced bottom, and are analogous to infinite loops in a procedural language. Worse, some expressions will reduce successfully under one evaluation strategy, but not under another.

Fortunately for us, the Church Rosser Theorems I and II (Peyton-Jones [37], Church [8]) state that all reduction sequences that terminate have the same result. Furthermore, if an expression can be reduced by any particular sequence of reductions to normal form, then the **normal order** reduction scheme is guaranteed to do so.

An expression in Weak Head Normal Form (WHNF) has no top level function redexes, but may contain subexpressions that can be evaluated.

## Sharing

The normal order example given is not particularly efficient, because common subexpressions are not shared. Rather they are substituted wherever required, and evaluation is repeated as necessary. Inventing suitable notation for sharing, a more efficient approach might be:



```

      / \
PRIMFN:(*) y

```

Each @ node in the example above represents an application. A function call such as “\* y z” consists of several curried applications, chained together into a **spine** along the left branch of its subtree.

There is more than one way to encode an expression as a graph. The representation above is particularly convenient for normal order reduction because the outermost function call can be found simply by following the spine nodes. Typically pointers to these nodes are pushed onto a stack to simplify later access.

```

Stack
-----
| ---|-----> @ <-- Root
-----          / \
| ---|-----> @   \
-----          / \ \
                FN:(f) x <expr>

```

The leftmost node of each spine contains information about the function to be applied. This includes the number of argument expressions (n) that the function expects — these can then be accessed as the top n pointers on the stack. It also contains a tag field that allows us to distinguish between user defined supercombinators and primitive functions.

## Representing Data

Structured data in functional languages is based on the composition of (non-reducible) constructor functions. These can be represented in the same way as normal functions, given a third type of tag field. Primitive data types such as integers and characters have their own special tag values. “Cons 42 xs” has the following representation:

```

      @
      / \
      @  xs
      / \
DATA:Cons @
          / \
          INT 42

```

## Heap storage

Each vertebra node corresponds to a cell that contains pair of pointers (left, right) to subexpressions, that may be further vertebrae or encoded data. These cells are maintained within a large array called a **heap**.

It is important that heap allocation is very fast at all times, as each evaluation step may request fresh cells. The use of fixed size cells that can contain two pointers makes it particularly simple to automatically gather or **garbage collect** discarded cells. You simply mark all the cells that are reachable from a collection of root cells, and then scan the heap, linking all unmarked cells together into a new list of free cells.

## Updating and Indirection Nodes

Normal order reduction can introduce multiple instances of the same expression. These expressions must be shared rather than copied in order to avoid repeated evaluation. This is quite simple to achieve with a heap representation – an object is shared simply by having multiple pointers to it.

This has a slight complication in the evaluation process. Typically, an expression is evaluated when its value is first required. A result expression is built and a pointer returned to the parent expression without the original expression being altered. However when one of the other “copies” of the expression is forced, there is no evidence of the previous result. Evaluation must be repeated.

The solution to this problem is to replace the graph of the original expression with the graph of the result. While an expensive extra overhead for the original caller, this means that any further attempts to evaluate the expression will access the evaluated form directly.

The nature of heap allocation means that it is difficult to overwrite a lump of graph directly. It is far easier to construct a result expression separately, and then overwrite the root of the original graph in some way.

As all heap nodes in our model are pairs of pointers, this can be achieved by simply overwriting the root node of the original expression with the root node of the result expression. However if this node is then copied we can end up with multiple copies of parts of an expression, which again can lead to unnecessary repeated evaluation.

It is better to have special indirection nodes that point to the result expression. This adds a slight overhead as the evaluator must transparently step through indirection nodes,

but they can be discarded by the garbage collection process.

## 4.5 Template Instantiation

It is quite simple to build a simple minded graph reduction system using the machinery introduced in previous sections.

The basic premise is that we consider the body of each supercombinator definition to be an expression template. To reduce a function call “f x y” where x and y are pointers to arbitrary lumps of graph, we simply construct the graph of the template, substituting these pointers wherever references to the function parameters appear. For example:

<pre>       @      / \     @   e2    / \   f   e1  f e1 e2 where f x y = x * y + 2 </pre>	instantiates to	<pre>       @      / \     @   2    / \   PRIM:(+) @            / \            @   e2           / \           PRIM:(*) e1 </pre>
-------------------------------------------------------------------------------------------	-----------------	----------------------------------------------------------------------------------------------------------------------------------

To evaluate an expression in this scheme, we simply repeat the following process until no top level redexes remain.

1. Find the next function application by unwinding the spine of the current expression.
2. If we are looking at a constructor function or primitive data type, then the expression is in WHNF.

If we have a primitive function, then evaluate its subexpressions (recursively as required), before performing the primitive action on the simple data values that should result.

Otherwise the expression should be a supercombinator redex. Just instantiate it as shown above.

3. Update the root of the redex with the result.

Template Instantiation is not very efficient. The problem is that the function template must be transversed on each separate instantiation, before the graph that it represents can be constructed.

## 4.6 The G machine

The conceptual leap to compiled implementations, such as the G machine, is to translate each supercombinator into a sequence of machine instructions before running the program. Each of these code sequences is tailored to building an instance of their supercombinator's body with none of the overhead associated with templates. The reduction process is the same as that described in the previous section, but the instantiation function is replaced by this new, specialised code.

To keep the model suitably abstract and machine independent, we introduce an abstract machine language. Gcode is similar in concept to the intermediate codes used by imperative languages such as Pascal. It is distinctive in being a stack rather than register based language with a single addressing mode and integrated heap support.

The G machine was originally invented by Auggustsson and Johnsson [1], although variants have been adopted by many others.

### 4.6.1 The peculiarities of Gofer Gcode

The particular Gcode variant described here is that used by the Gofer interpreter. The main differences to the G machine described by Peyton Jones and Lester [35] are as follows:

1. The Gofer interpreter has a very involved encoding system that allows a tag field and pointer to be combined into a single machine word. This includes a special unboxed representation for characters and small integers to reduce heap throughput and increase performance.
2. Stack elements are counted from the current root rather than the top of the stack. This means that a functions parameters and important local state can be accessed as constant offsets for LOAD and UPDATE instructions. This makes the Gcode compiler's life easier, and makes for more readable machine code.

3. An extra instruction “ROOT” creates partial applications efficiently. i.e. “f x y” can be generated directly from “f x y z”. This instruction is strictly redundant and can be replaced by a sequence of LOAD/MKAP calls. However combined with the MKAP instruction, this is particularly useful for recursive functions where only the last few parameters change.
4. Gofer has an interesting way of decomposing structured data. When a structured data item (say `Cons x xs`) is evaluated to WHNF by an EVAL instruction, the spine nodes are pushed on the stack as extra local state for the caller. This information can be used to build a result expression directly. See below for an example of this process in action.

As code is generated for each function, the compiler must mimic the evolving stack frames in order to generate correct offsets into the local state information. This is, however, a lot more direct than the selector functions used by Peyton Jones [37] et al.

The instruction set of the Gofer G-machine is given in appendix A. In typical fashion, the information presented here was reverse engineered from the source code a few months before Mark Jones published definitive information about the Gofer implementation in [30]. C’est la vie!

### 4.6.2 An example: The map function

Let us see how the standard function “map” compiles into Gofer Gcode instructions:

```
map           :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

The references [x]: refer to newly created stack frame slots:

Initial Stack Frame slots are function parameters:

```
1: (x:rxs)
2: f
```

name=map

```
0x00E8  LOAD    1           Duplicate the second function argument
```

```

0x00EA  EVAL          Evaluate it
0x00EB  TEST      [] 0x00F3  Is it a [] Cell
0x00EE  CELL      []          If so:
0x00F0  UPDATE    0          Copy the [] over root cell and return
0x00F2  RETURN
0x00F3  TEST      : 0x0000  Otherwise, it should be a (x:rxs) cell.
                                Extra stack frame slots from last WHNF:
                                3 -> rxs
                                4 -> x

0x00F6  LOAD      3          5: Duplicate (rxs)
0x00F8  ROOT      1          6: Form Partial Root (map f)
0x00FA  MKAP      1          5: Glue on rxs -> (map f rxs)
0x00FC  LOAD      4          6: Duplicate x
0x00FE  LOAD      2          7: Duplicate f
0x0100  MKAP      1          6: Build app (f x)
0x0102  CELL      :          7: Make (:) cell
0x0104  MKAP      1          8: Complete graph (f x) : (map f rxs)

0x0106  UPDAP     0          Update and return
0x0108  RETURN

```

### 4.6.3 Optimisations to the G machine

Gofer is not a particularly fast implementation because it interprets the Gcode that it generates, rather than translating it to native machine code. However, the **model** contains two major inefficiencies that are addressed by a refined model, the Spineless Tagless G machine (Peyton Jones [36]).

1. The use of a spine doubles the number of nodes (and therefore accesses) required to store an expression. We need a spine to represent function applications with an arbitrary number of arguments in a heap that consists of simple fixed size cells.

Given a heap that provides for variable size nodes, we can store all the pointers for an application into a single node, and therefore discard the spine. Unfortunately, this complicates the garbage collection process, and requires special code to handle partial applications properly.

2. The tag value must be extracted and processed to determine the type of an expression, and therefore the code to use for it. It is faster to replace the tag with a function

pointer that contains special case code for a particular type of expression.

This idea has been developed in the STG intermediate code (Peyton Jones [38]), which is a purely functional notation with both denotational and operational meanings. This code has a well defined mapping to C code.

Peyton Jones and Launchbury [34] consider the well integrated, rather than ad hoc, use of unboxed values. A different compilation scheme called the Three Instruction Machine, or TIM, addresses the problem of only building the expression trees for function parameters as their values are required (Fairbairn and Wray [12], Peyton Jones and Lester [35])

Many other optimisations are possible using the type and strictness information generated by a compiler.

# Chapter 5

## Functional Concurrency

In this chapter, we consider the behaviour of concurrent functional systems. We suggest that applying traditional stateful models of concurrency to the functional world leads us into ways of thinking that force the use of nondeterministic operators. As an alternative, we present a demand driven and lazy view of concurrency that far better matches the requirements of the functional programmer.

### 5.1 Concurrency is not Simply Parallelism

Very often in the computing literature, no sharp distinction is drawn between parallelism and concurrency, indeed the two terms are often used interchangeably. A parallel system is one that contains multiple threads of execution working towards a common goal. These threads may be timesliced onto a single processor, or distributed onto a set of processors to give genuine rather than pseudo-parallelism.

As functional expressions have no side effects, the value of an expression depends only on the values of its subexpressions. These subexpressions can be evaluated simultaneously, subject only to data dependencies i.e. situations where one expression must be evaluated before another because the second requires the value of the first. A parallel algorithm is simply one where data dependencies are kept to a minimum. Although many activities may be carried out at the same time, they are coordinated in such a way that the results are the same as if the program was carried out sequentially. Thus parallelism alone does not increase the expressive power of languages beyond sequential execution.

In contrast, concurrency is a property of the externally visible behaviour of a program.

A program is concurrent if it is able to do several things at once in an unsynchronised way. Of course, a concurrent program requires some measure of parallelism in its implementation.

## 5.2 A Traditional View of Concurrency in Operating System Design

If a single state was shared by all threads of execution, each thread could arbitrarily alter the private state of another thread at any time. This would be a large source of nondeterminism and could cause many subtle bugs, as misbehaving threads could have far reaching effects.

Unix style operating systems deal with this problem by providing heavyweight processes. Each process has its own independent address space and therefore cannot interfere, directly, with another process's state. Processes normally interact by sending messages along communication channels routed via the kernel. Process execution passes into kernel space through carefully controlled entry points. This places a great deal of complexity inside the kernel which must deal with a large amount of sharing between threads as well as all thread scheduling decisions.

Typically each process is host to a single thread of execution. Therefore, each process is purely sequential, and it is only when the network of processes is considered as a whole that concurrency becomes visible. The flow of messages into and out of each process controls its rate of execution.

Most operating system environments consist of a relatively small number of large processes that are weakly coupled: that is most state is private, rather than shared with other processes. Each process can be deceived into thinking that it has complete control of the computer. It endeavours, with varying degrees of success, to ignore the nondeterministic effects caused by other processes intruding into its private little world.

The overheads of interprocess communication mean that networks of heavyweight processes are much less effective solutions when large amounts of shared data is involved in a concurrent system. System V based Unix systems allow user processes to define shared memory between two or more processes. A more generally available solution is to introduce multiple threads of execution into a single heavyweight process. Programming using

“light-weight threads” is not for the faint of heart. It is notoriously complex to manage stateful processes that contain **arbitrary** internal concurrency. Nondeterministic artifacts quickly emerge. A whole area of computer science is dedicated to the study of semaphores, monitors and coroutines as methods of serialising concurrent access to vulnerable code that would otherwise lead to subtle bugs or deadlock situations.

The structure of the existing Gofer interpreter necessitates the use of light-weight threads in our own prototype. However the well behaved nature of the shared state (the Gofer heap) and large amount of private thread state means it far easier to develop a clean design than in the general case.

### 5.3 Data Driven Design

Evaluation in procedural systems is normally controlled by input events. Each thread of execution independently waits for data to be received from another thread or device driver, and then produces as many output events as possible before repeating the cycle. We say that evaluation is data driven. Existing functional operating system projects all presume data driven processes, even when lazy languages are used.

Typically, each process can negotiate arbitrary connections to other processes on both temporary and permanent basis. Each output event can be directed along any current output stream. The lack of any enforced discipline on the part of sending processes makes it very difficult to predict I/O behaviour. Worse, it enforces merge based thinking.

Consider a situation with a process “**scatter**” that has a single input stream and a large number of output streams. The process dispatches each input event that arrives to an output stream based on some complex heuristic that is unknown to the receiving process “**sink**”. To all intents and purposes the distribution of messages is random to the receiver. If a number of distinct connections between “**scatter**” and “**sink**” exist, the receiver cannot predict which channel the next message will arrive on (See Figure 5.1).

The only options are to poll between channels checking for a message, or to use a nondeterministic operator such as the `select()` system call to implement a logical merge of input streams.<sup>1</sup> This rather contrived example demonstrates that data driven thinking can force the use of a nondeterministic operator even when no concurrency is involved.

---

<sup>1</sup>An even simpler scheme on packet based networks such as the UDP/IP is to only advertise a single input port to each process. The communications protocol does the merge for you.

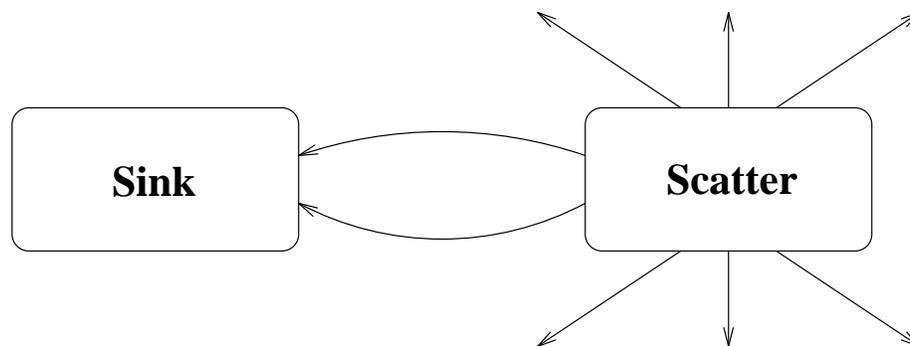


Figure 5.1: Scatter causes nondeterminism

With more than one independent sending process, there is little choice in the matter. Almost all the Unix networking tools feature an instance of the `select()` system call. Easy availability means that there is a tendency to adopt `select()` even when a system could be redesigned to avoid its use.

## 5.4 Demand Driven Design

In contrast, evaluation in lazy function languages is driven by the explicit knowledge that the value of a subexpression is definitely required to evaluate the current expression. This leads to the intriguing question as to whether a “demand driven” philosophy of network communication might be more suitable to a functional system.

### 5.4.1 Network Structure

It is rather less obvious as to the form such a network might take. Let us work from the premise that a demand driven network is one where an output event is only generated and sent when it is specifically requested by the other end of the channel. The reader will note a close analogy between the use of the word “demand” in this section compared to that of “event” in the last. Some points to note are:

1. Such a network is rather more restrictive, but much better behaved than a data driven one. Each process is forced to respond to specific demands for output events, rather than transmitting output data as it sees fit. While a request based system

can be implemented in a data driven environment, there is no way of **forcing** the sending end into such behaviour.

2. While the receiving process initiates evaluation it cannot influence the result without some kind of feedback circuitry. This is consistent with functional philosophy.
3. Laziness is preserved across process boundaries. A network of lazy processes linked by demand driven network connections is bottom avoiding.

### 5.4.2 Propagating Demand

Existing communication mechanisms are data driven in nature and their flow control mechanisms are insufficiently lazy to support demand driven networks directly. The simplest unbuffered channel protocol which is semantically sound can be described as follows. When the consumer process reaches a point where a value is required from the channel, it creates a ‘demand token’ for the next item from the producer and is suspended. The producer is woken, evaluates the item, sends it and is suspended again. This design has quite a high overhead compared to single direction data transfer, but strictness analysis might allow us to send several demand tokens together as an optimisation.

A data driven process can be integrated into a demand driven network by making it a source of demand, so that every time it receives an input event, it immediately requests the next. Existing procedural services can be added to our network using a functional ‘wrapper’ in this way. Of course, if a producer knows that it is attached to a data driven consumer there is no need to actually send demand tokens: the producer can consider itself to be under perpetual demand (See figure 5.2). This is a useful optimisation adopted by our prototype system. It is also the standard model for interfacing a lazy functional program to system resources using an I/O Dialogue.

### 5.4.3 Concurrency in a Demand Driven System

In a data driven system, concurrent evaluation is possible when we have more than one process with outstanding or unprocessed input events. Each output event generated will allow another process to evaluate. As each process may generate an arbitrary number of output events before suspending on input, it is easy to see how concurrency is self-propagating in a data driven environment.

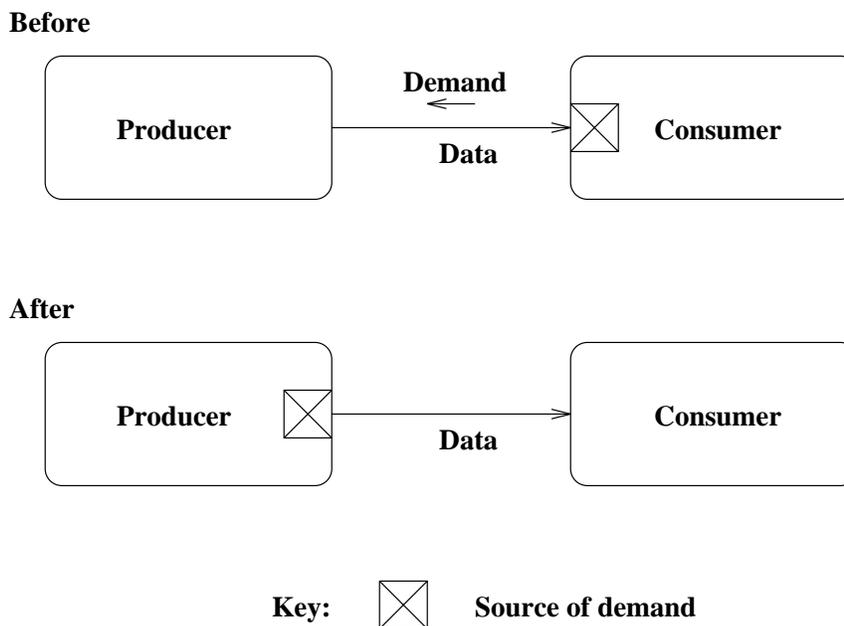


Figure 5.2: Data driven processes cause perpetual demand

All that has been introduced to date is a mechanism for preserving lazy evaluation across a communication mechanism. An obvious question is: how does concurrency manifest itself in a demand driven network? We believe that the answer is quite beautifully elegant and simple.

If each demand causes a thread of execution to evaluate an output event then a concurrent demand driven system is simply one that allows several independent demands to be outstanding and processed simultaneously.

The keyword in the last sentence was “independent”. In the case of speculative parallelism we can have multiple demands without concurrency. We have already seen that data driven processes act as sources of demand in a demand driven network. Therefore a mixed network with several data driven elements is naturally concurrent. Later we shall see that this demand is caused by naturally concurrent objects such as individual X windows and network device drivers. These devices will normally be controlled using an I/O dialogue by the functional subsystem.

Conversely, a function that supports multiple independent demands must be capable of concurrent evaluation (see figure 5.3). Typically this is implemented using a lightweight

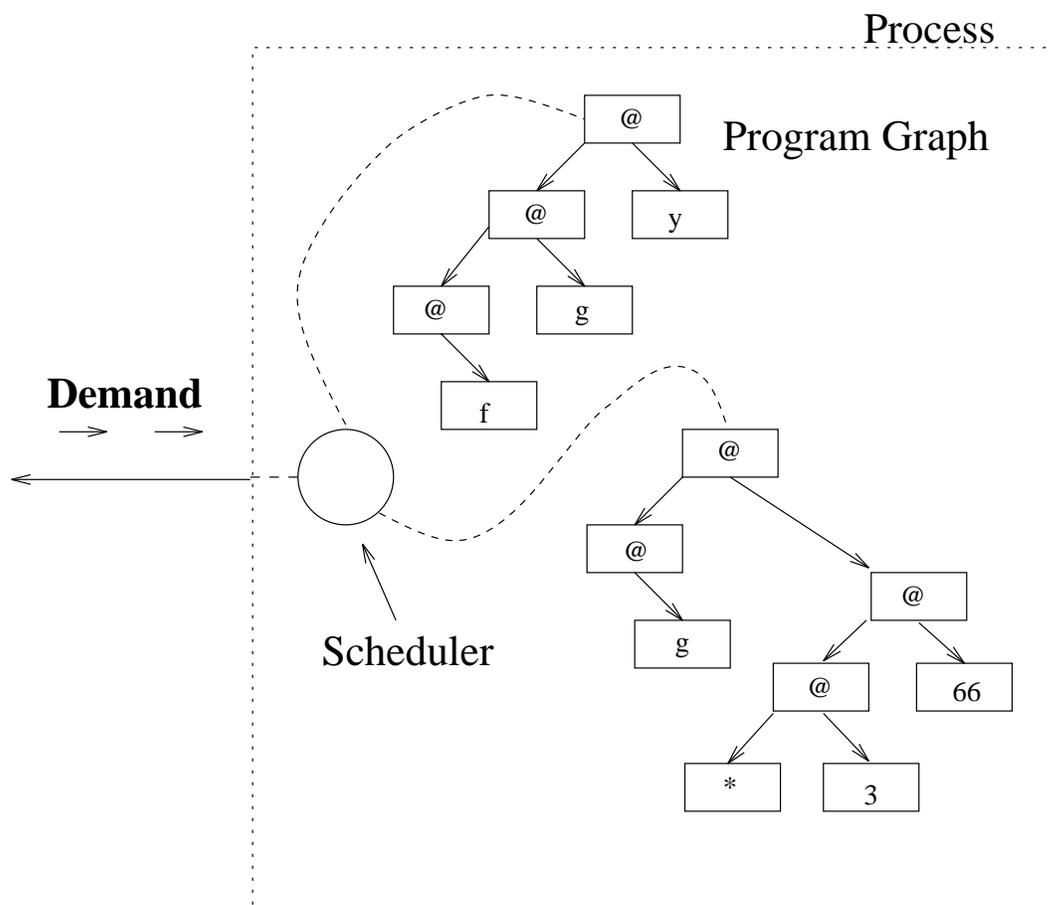


Figure 5.3: Multiple Demands

threads mechanism. It was suggested in section 5.2 that this is not a very pleasant programming activity. What is quite remarkable is that we can build a multithreaded Gofer interpreter, containing a great deal of complexity, in such a way that the functional programmer is hardly aware of the difference, even when writing concurrent code.

The pipelining effect of lazy evaluation means that there are often much fewer active threads of execution than in a data driven system. This can lead to poor performance using traditional communications channels. If great care is taken to preserve laziness, the problem can be overcome to a certain extent using speculative parallelism powered by preemptive demand.

### 5.4.4 Multiple Output Channels

Typically a process needs to be concurrent if it is supplying data to two independent data driven servers, neither of which is aware of, or wishes to be distracted by the other. We are now in a position to introduce a very convenient notation that simplifies both the presentation of our model and its implementation.

Typically a data driven functional process will control separate devices by sending a single stream of tagged output requests to a sorting office. We are restricted to a single output stream because each process is sequential and must commit to computing a single expression at a time<sup>2</sup>. However with multiple demands we produce several output values concurrently. There is therefore nothing stopping us from having several output channels, and associating one or more threads of execution to each channel (See figure 5.4).

This is a very powerful mechanism for structuring concurrent applications, as the logically distinct output channels of a sorting office model are separated into physically distinct channels that do not require support infrastructure to tag and untag messages. We suggest that a process with several output streams is more expressive than one restricted to a single stream.

### 5.4.5 Hyperstrict Communications

When using the hyperstrict communication channels provided by standard interprocess communication primitives, it is only possible to deal with a single outstanding demand on each output stream (although others may be queued). Using a suitable low level protocol, communication streams can be designed to be lazy in the sequencing of elements but hyperstrict in their contents. This allows out of sequence evaluation and concurrent demand on a single output channel. Trinder [60] demonstrates that such concurrency allows independent requests to overtake one another in the design of a functional database system.

However, in general, demand is caused by device drivers asking for the “next” event from a process. As device drivers themselves do not feature internal concurrency, events are processed, and therefore requested, sequentially. Therefore, generally only a single outstanding demand per channel is required. This is a useful simplification as we can

---

<sup>2</sup>If multiple output channels were provided we would have to commit to one of a number of expressions at random. This is unsafe, as many orderings will lead to deadlock

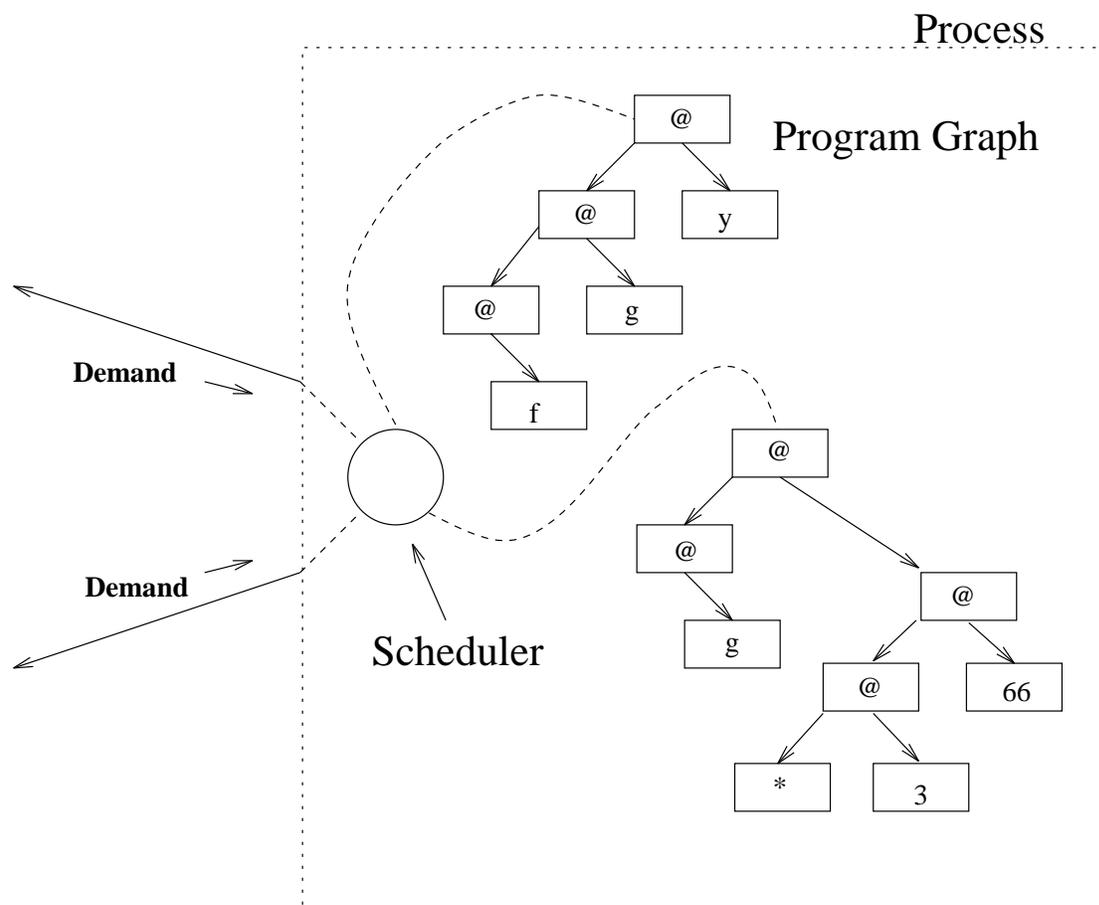


Figure 5.4: Multiple Channels

dedicate a separate thread to each output stream, rather than sparking off a new thread for each demand request. This thread will wait for an incoming demand before attempting to evaluate the next item on the output channel.

### 5.4.6 Merge

The astute reader will have noticed a careful lack of any mention about nondeterministic “merge” style primitives in the preceding discussion.

Merge is particular easy and cheap to implement in a data driven environment. An operator such as the `select()` system call in Unix will fire when input arrives on one of a number of input channels.

The situation is rather different in the realm of concurrent demand driven systems. Evaluation only occurs when a value is required. In order to select the first of two values to be available we must begin concurrent evaluation of two expressions using an operator such as McCarthy’s “`amb`”. The first thread to return a value causes the `amb` node to terminate, pruning the other branch of execution. This requires significantly more support infrastructure than the data driven case. Note that unlike `merge`, this operation does not, in itself, guarantee fairness.

### 5.4.7 Conclusions

We suggest that data driven systems encourage merge based thinking that is significantly more difficult to implement in a demand driven world. We have suggested that a concurrent, demand driven, functional system offers additional expressive power over the equivalent data driven version. The question is whether or not this extra expressiveness is enough to implement useful systems without resorting to `merge`. We will show that several key components of a single user workstation environment can be redesigned in a purely deterministic fashion. While less general, these deterministic components are much better behaved.

## 5.5 Behaviour of Concurrent Demand Driven Systems

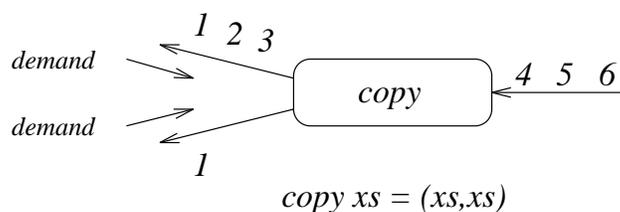
The behaviour of a computing system can be described using some form of formal semantics notation. One of the stated benefits of functional languages is that their simple operational semantics match their static (or denotational) reading. This will be a very informal discussion: a more detailed discussion about denotational semantics can be found in [55].

Let us briefly consider how a process supporting concurrent demand through multiple output streams should behave. To explain lazy evaluation we need to extend strict semantics to demonstrate how non-terminating computation ( $\perp$ ) is handled. In contrast we do not need to change either the syntax or semantics of a lazy functional language in order to explain concurrency. The denotational semantics is just the usual lazy semantics for functional languages; it is simply the fact that functions are evaluated in the context of multiple demands which increases their expressive power. The operational and denota-

tional points of view agree; nondeterminism is used in the implementation in a restricted way which provides a purely deterministic increase in expressive power.

The fact that the denotational semantics is unchanged has powerful consequences. It means that traditional declarative methods can be used to determine program properties. For example, a true deadlock, ie one caused by the accidental timings of events or communications, is impossible. If output on a particular stream is suspended indefinitely, it can only be because the remainder of the stream has value  $\perp$ , which means that the effect is repeatable and can be investigated easily.

As a concrete illustration of the relationship between the operational and denotational points of view, consider a function `copy` which takes a single stream as input and returns two copies of that stream as outputs:



The denotational semantics of this is very simple; for every possible value which `xs` can take, including partial values, the result must consist of two copies of that value. If the two output streams are evaluated independently by separate tasks, the results seen by those two tasks must be completely determined by the value of `xs`. If `xs` has the value  $[1, 2, 3, \dots]$ , the first task demands three items from the first output stream, and the second task demands one item from the second output stream, then the situation will be as shown above. Each task sees the items it has demanded, and the values 2 and 3 are ‘buffered’ until demanded by the second task.

A very important point is that our `copy` process must be able to respond to demands in any order. In particular if one thread enters a non-terminating computation (i.e. it has value  $\perp$ ), we need some way of ensuring that other outstanding requests continue to be computed. In general this requires some form of time-slicing mechanism between threads. Also note that hyperstrict channels are not bottom avoiding [61], although section 5.4.5 suggests a mechanism for making them so.

### 5.5.1 Copy Transformation

We have observed that a demand driven process with multiple output streams can be implemented using a single thread of execution for each channel. This means that it is possible to rewrite a process with  $n$  output streams as a network of  $n$  sequential processes that filter inputs values from an  $n$  way copy operator (one for each input stream). See figure 5.5.

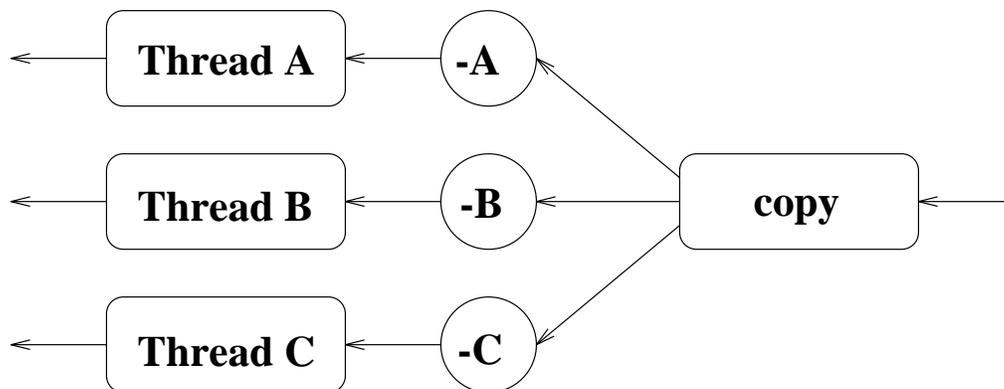


Figure 5.5: Copy Transformation

Note that we have not **removed** multiple output channels, just concentrated the functionality on the input side. Demand driven semantics must be maintained in the network communications.

In general a set of processes using multiple output channels can be transformed to a set of processes using single input and output channels, linked by `copy` operators. This gives us a useful handle on the behaviour of our concurrent systems. Unfortunately, the transformed networks are not very efficient, as the filter operations that are introduced hold onto a great deal of shared state. Consider a situation where one output channel demands continuously, while a second does not demand at all. All of the computed output values must be buffered and cannot be garbage collected until the second output channel is closed.

### 5.5.2 Processes can be Networks

We have attempted to present a network layer that preserves lazy functional evaluation. A very pleasant feature of this model is that a network of lazy processes can be replaced by a single lazy process, hiding internal communications.

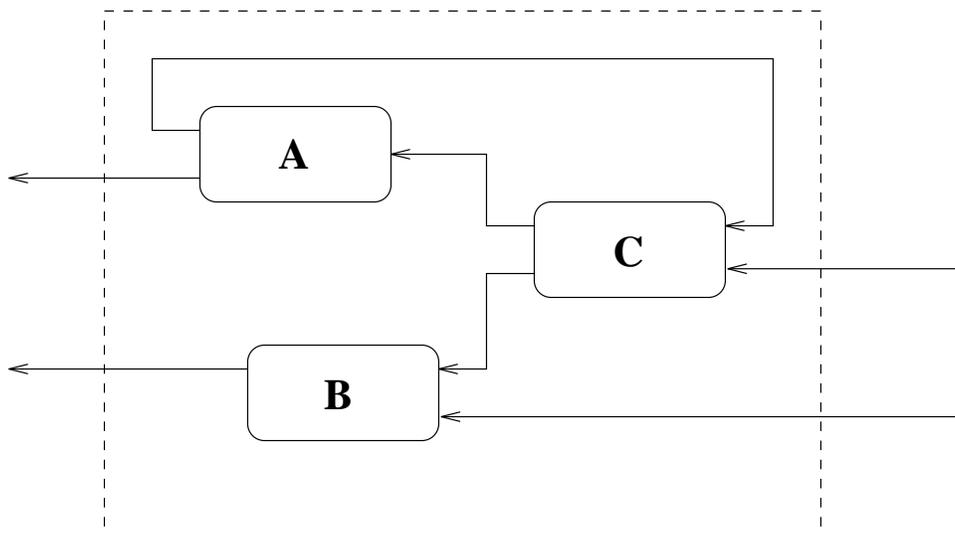


Figure 5.6: Network of Processes as a single Process

This is a very powerful abstraction mechanism, much in keeping with the design of programs using pure functions. This equivalence is not completely true in the reverse direction: a program connected by lazy streams cannot necessarily be split up into separate processes. This is because lazy streams are more general than hyperstrict communications channels. We can however enforce hyperstrictness on lazy streams using head strict cons (`!`, [33]) to guarantee equivalence. Alternatively, as we suggested in section 5.4.5, communications channels can be implemented that allow concurrent access.

We have suggested that data driven networks favour heavyweight processes containing single threads of execution. In contrast, demand driven functional systems work better inside a single heap where sharing of lazy values more complex than hyperstrict streams is simpler and less costly. The only external connections that we need are to device drivers and remote computers. This is the model adopted by our prototype system.

We have suggested that a process with  $n$  output streams in general requires  $n$  threads of execution to implement in a hyperstrict environment. An interesting consequence of

the “process as network” hypothesis is that the complex network given in figure 5.6 only requires two **active** threads of execution to implement, even if the network is distributed.

## 5.6 Network Configuration

We have presented our model in terms of multiple output channels, where each channel is directed towards a separate external resource. An important question is: how can we interface to additional resources while a process is running? In this section we present three different mechanisms for adding extra output channels to a process. The first two of these mechanisms are ways of linking in an external data driven server that acts as a source of concurrency to our system. The third is a way of setting up demand driven communications channels between lazy functional heaps.

The network manager dialogue used by the first two mechanisms is another source of demand in our system, greedily evaluating requests to configure the network. This thread is the initial source of demand in our system. However, the program does not simply shut down when this ‘master’ thread terminates: we must wait for all other threads of execution to run their course.

Note that we only need mechanisms to ADD channels and processes. The garbage collector can transparently discard resources when their useful lifespan has finished. These models provide a stronger handle on network configuration than with the sorting office. There, special procedural network management processes invoke and plumb in whole processes with little control provided by functional code.

### 5.6.1 WriteChan Request

The simplest and most familiar way of adding concurrent output streams is to extend the Haskell I/O Dialogue by a single Request. The existing channel operations for sequential use are as follows:

```
data Request = ReadChan Name | AppendChan Name String | ...

data Response = Str String | Success | ...
```

`ReadChan` returns a lazy stream that will be evaluated as required to get the correct text interleaving in interactive applications. Multiple instances of `ReadChan` create multiple

lazy input channels that can be accessed in any order.

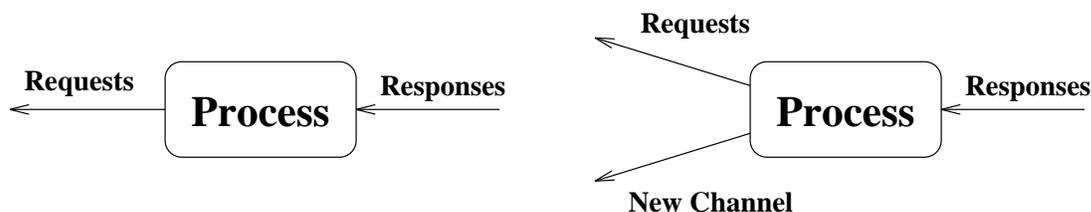
In contrast, the use of a single output stream means that each `AppendChan` request must completely evaluate its string argument before sending to be processed. This is because a sequential language can only evaluate a single expression at a time. Consequently, if a program wishes to control a set of resources rather than a single channel, a number of distinct `AppendChan` requests must be issued to construct a single logical output stream. For example:

```
outputs = [AppendChan stdout "Line 1\n",
           AppendChan stderr "Error Message",
           AppendChan stdout "Line 2\n"]
```

The extension proposed here is to add a single new request, `WriteChan`, that opens an independent output channel:

```
data Request = ... | WriteChan Name Stream
```

A request `WriteChan chan cs` creates a separate thread of execution within the process, that will evaluate the list `cs` as required by the external entity `chan` (which may be either data or demand driven, as previously discussed). A separate `ReadChan` request can be issued to complete the data flow loop. In this way we can set up a number of independent Dialogue interfaces, where each Dialogue contains a thread of execution to drive its request stream. This is a very intuitive presentation of concurrency for a functional programmer.



Multiple `WriteChan` requests to the same destination can be either be defined as an error, or queued so that they are evaluated in strict order.

While this style presents a simple and familiar model, the use of an explicit and arbitrary name space of output channels causes housekeeping problems. The `AppendChan` request works well because it communicates with only two channels (`stdout` and `stderr`) whose effects are well defined and configured before the program starts running. In contrast

the `WriteChan` extension (and its suitable augmented partner `ReadChan`) are intended to deal with a dynamically configurable network of services. In short, we have no guarantee that a given endpoint actually exists, unless we ourselves create it. Two possible ways of managing this situation are as follows:

1. We presume a (possibility infinite) set of predefined endpoints split into appropriate groups (e.g. “window/1”, “window/2”, “disk/1”, “disk/2”). Any attempt to `WriteChan` or `ReadChan` to one of these names automatically invokes a handler service that will generate demand as required.
2. We add a separate output request `CreateChan ChannelType` to the `Dialogue` type. This creates a service of the required type and returns a name bound to it through the response arm of the `Dialogue`. Later `ReadChan` and `WriteChan` requests can bind directly to this name.

Neither of these solutions is particularly elegant or pleasant to implement.

## 5.6.2 MakeChannel

The problem with the `WriteChan` primitive is the dependence on explicit names. When dealing with independent resources such as top level windows, most of the time we just want to create an interface using the “next available slot”. The only need for an explicit channel name at all is so that we can bind the input and output streams as two separate operations. Typically however there is no reason that we could not bind both streams as a single action (See figure 5.7).

```
data Request = ... | MakeChannel ChannelType Stream
```

```
data Response = ... | ChannelMade Stream
```

Here we presume that `ChannelType` is the name of a service to bind to, for example `Window`, and that the Meta-server is capable of fielding all such requests. An alternative scheme, used by our prototype system, is to name an external program or **server process** to execute. The output and input streams are bound to the standard input and output of this external server, using a demand to data driven converter as described previously. This makes the external server a new source of demand in the system. The server used

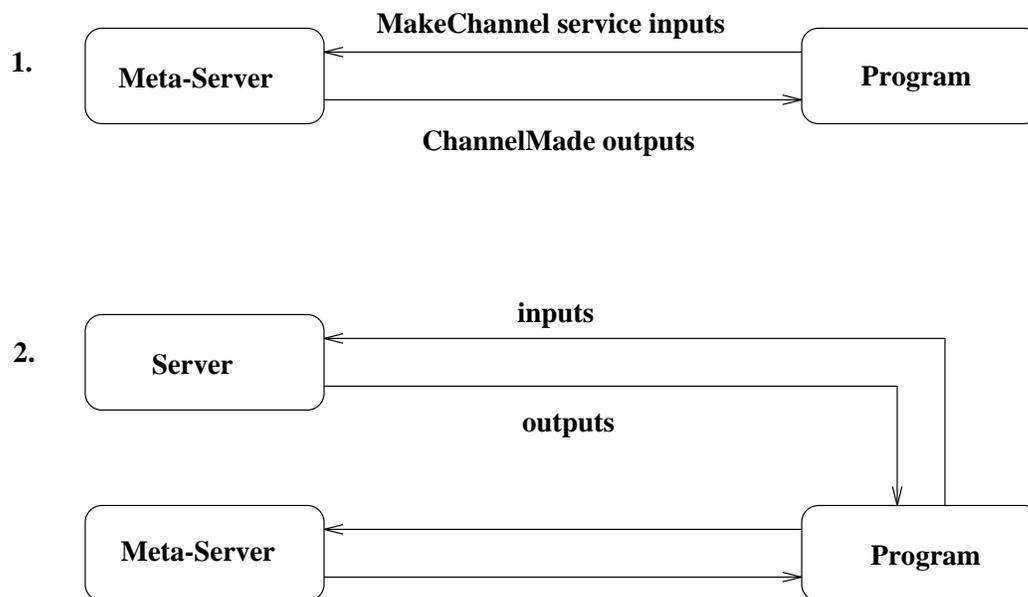


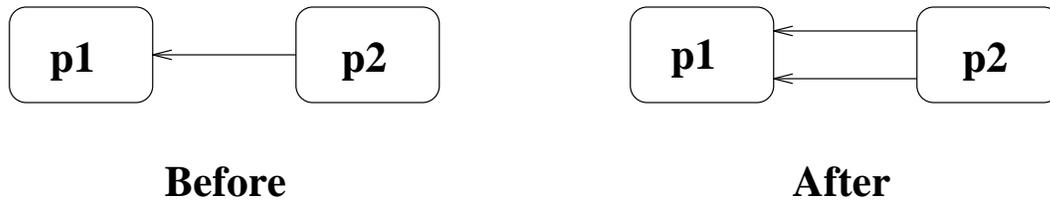
Figure 5.7: MakeChannel operation

by our prototype is “`xterm -e server`”, where `server` is just such a converter process for line based terminal applications. Obviously care must be taken to ensure that the external servers do not interfere with one another causing implicit nondeterministic effects. However, it works well as a prototyping mechanism.

### 5.6.3 Channel Passing

Both of the previous schemes are ways in which functional processes can communicate with procedural support code generating extra concurrency in the system. This does nothing to address the problem of setting up connections between a set of lazy functional processes, possibly executing in separate heavyweight processes.

When a functional process is started, a number of input and output channels can be created, as with a standard function call. If we wish to plumb in extra communication channels as some later point, the only access points that we have are those channels. Therefore we must provide some capability for passing a channel over an existing channel from one process to another:

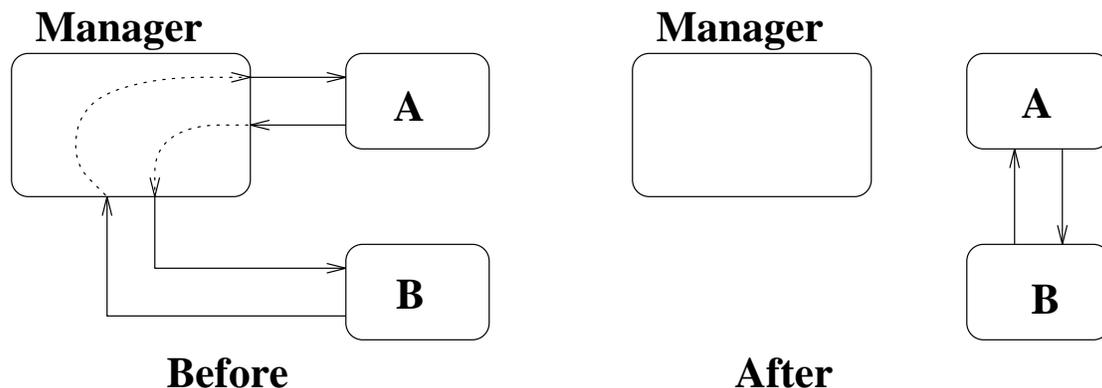


This is obviously a trivial operation to implement with a single shared heap, you simply turn the stream type into an enumerated type, one element of which can be a channel itself:

```
type ChannelType a = [ElementType a]
data ElementType a = Value a | Channel (ChannelType a)
```

It is rather more difficult to drop such a mechanism on top of a hyperstrict communications channel. Passing a channel corresponds to negotiating a new network connection between two processes. This would be quite easy, but our lightweight threads system forces us to use asynchronous nonblocking communications. This requires a large amount of state to be associated to each socket connection so that partly completed interactions can be continued.

In a network of functional processes, communications channels are frequently received by coordinating functions and immediately forwarded to their correct destination without processing. This creates a large amount of internal plumbing. Traditionally we rely on garbage collection to short circuit out unnecessary links. This is not directly possible when communications channels link across heaps. We must introduce some form of optimisation that work with the garbage collection processes in individual heaps to replace lengthy multi-hop connections with an optimised route. This optimisation is termed replugging.



### 5.6.4 Conclusions

In the next chapter we describe a simple prototype implementation based around a single heap concurrent interpreter. The concurrency primitive used is a simple variation on the `MakeChannel` request. This connects to external data driven server processes that act as sources of concurrent demand as previously described.

# Chapter 6

## Deterministic Design

Chapter 3 argued that nondeterminism has unpleasant effects on the purity of a functional language. It is, however, **required** in the design of certain operating system software. We concur with this viewpoint. Certain characteristics, such as hardware interrupts controlling device drivers, create nondeterminism.

The unwritten implication is that if we are to write operating system software in a functional language, we **require** some way of expressing nondeterminism in that language. We believe that this is only partly true. Strictly speaking, the term “functional operating system” is something of a misnomer. Most previous projects have focussed on ways of replacing the systems software (shells, windowing systems etc) using the existing procedural kernel and its support libraries. A great deal of nondeterminism can be confined to this procedural core code. The result is a functional “user environment”. Most attempt to emulate the services provided by existing procedural systems.

Let us consider the situation from a different standpoint. Do we need to supply all the facilities of a traditional operating system in order to produce a useful working environment? Most computer users find the unpredictable nature of nondeterministic systems disconcerting, and frequently frustrating. And while nondeterminism appears to offer a short cut to many program design problems, it is often also the cause of subtle bugs that are very difficult to track down.

We suggest that a potentially quite powerful, single user, multitasking, user environment can be constructed with no nondeterministic feature available to the functional programmer at all. This requires a certain amount of careful redesign to remove both explicit and implicit forms of nondeterminism. A simple, but interesting, demonstration system

based around an extended Gofer interpreter is presented and its features are discussed.

The major restriction on the type of environments that can be constructed using this model is that polling for input events is not allowed: this automatically causes nondeterminism. In short, we can only build systems that are strictly input driven: an event is captured and a well defined sequence of actions occurs before another event is required. Self perpetuating systems, for example most interactive computer games, cannot be built in this restricted universe.

Complete functional source of this prototype is presented in appendix B.

## 6.1 Nondeterminism and resources

Merge operations are used to select the first available event from a set of input ports. This very low level description explains how nondeterminism is introduced into a system, but not why it is required in the first place. In order to understand this, we must take a step back and view the network of processes as a whole.

The simple answer is that unrestricted concurrent access to a shared resource requires nondeterminism. Consider a file system process that is serving two concurrent processes A and B. Each of the clients can issue a `ReadFile` or `WriteFile` request an any time. If the entire filesystem is shared, access control must be centralised. The file system must be able to respond to two separate request streams, and this requires some form of merge operation (see figure 6.1).

In traditional procedural operating systems, concurrent access is even less restricted. For example, every single user process can attempt to open any file at any time on Unix systems. Most programs pay little attention to the simple advisory locking that exists in many implementations. The practical upshot of this is that each process is unaware of any others that may be editing the same set of files. The cumulative effect of overlapped edits is unpredictable to say the least.

The only way to overcome these effects is to enforce a very strong system of ownership, where only a single process may update a resource at a time. This draconian approach can be made rather more flexible by allowing temporary delegation of ownership, especially where a resource can be split into two or more independent parts that can be processed concurrently without fear of overlap.

We now consider possible deterministic designs for two common types of resource: a

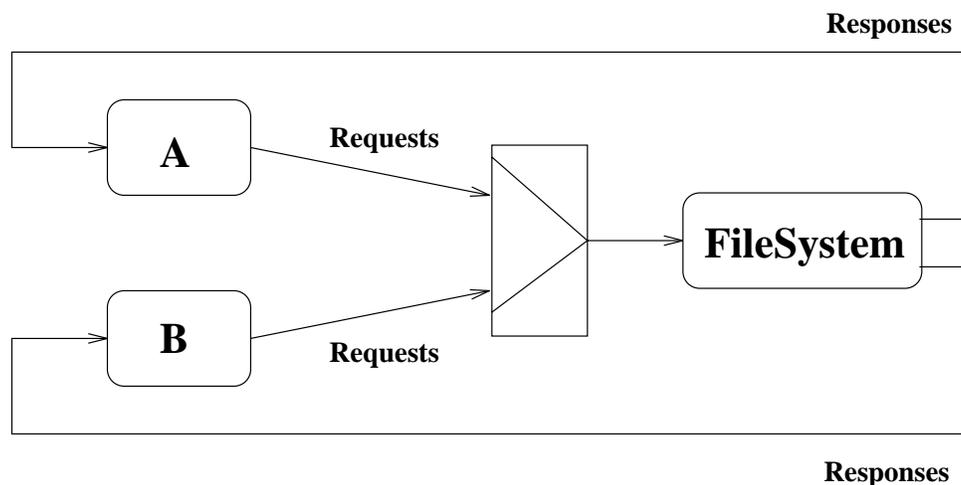


Figure 6.1: File System Nondeterminism

bitmapped display and a simple filesystem. While many other examples would be possible, these two items are the most important elements in a single user working environment. We suggest that other resources can be divided into the classifications of “stateful object” and “I/O device” that match the characteristics of these two examples.

### 6.1.1 Bitmap Display Example

In a procedural system, it is easy to see that a display can be managed by directly updating the grid of characters or pixels that represent that display. A functional program could control a display by feeding a stream of commands to such a process. This is very much the model that the X windowing system adopts for socket based connections.

Unrestricted concurrent access to the bitmap would cause nondeterministic and unpredictable effects. Using simple text based terminals without cursor motion control, the interleaving caused by a merge might create a usable, if confusing display. This is obviously not true in the general bitmap case.

In traditional windowing system models this problem is addressed by providing a set of top level windows, that can be manipulated independently without fear of interference. In the simplest presentation, these windows are **tiled** onto the display as separate, non-overlapping regions of the bitmap. This is a very obvious way of splitting the display resource into sections that can be manipulated concurrently in safety.

A common refinement of the tiling model is to use a window manager. This is a privileged application that can control all the windows on the display. It allows the user to position and resize windows with arbitrary overlapping. The display as a whole is a purely deterministic mapping of the contents of the windows and the state of the window manager<sup>1</sup>. Here, we presume that each window has its own personal bitmap that is updated so that the window manager can move and reorder windows without any cooperation required on the part of individual applications. This can be very memory intensive. A common optimisation is to force the window manager to add special redisplay events into each window's event stream. Each time that the controlling application receives such an **Expose** event, it rebuilds part or all of its display as requested. In this model, each graphics operation is clipped against the boundaries of overlapping windows before being rendered onto the physical display.

Generally, windows have an associated keyboard and mouse that create a stream of events. We can therefore consider a simple window as having a Dialogue style interface type `[Command] -> [Event]`. Note that the input and output streams are unconnected and independent: graphical output commands change the visible state of a workstation, but do not influence the event input stream. We can regard the window's controlling Dialogue as being a stateless abstract data type. In effect the user is the algorithm of that abstract data type: she views the current, visible, state of the window, and decides which events to generate accordingly. Consequently, it is important that the display is an accurate and up-to-date portrayal of the program's state. In the context of lazy evaluation, there are two ways of doing this:

1. Enforce a strong pairing between the input and output sides of the Dialogue so that we know when an input event is expected. As a process can send an arbitrary number of output commands before it suspends to wait for an input event, we must change the Dialogue type to `[[Command]] -> [Event]`. This is analogous to the Fudget law discussed in [7].
2. Make the window Dialogue a source of perpetual demand. If we regard each window as a data driven object (and this will invariably be how it is implemented at the low level), then we can completely detach the input and output sides of the Dialogue. A

---

<sup>1</sup>However, as the windows are under individual and concurrent control it is impossible to write such a function except as a snapshot mechanism.

thread of evaluation evaluates the `Command` stream as far as possible at any point, while `Events` are queued on an output port as they are generated.

With this model windows become the source of concurrency in a system, as more than one window implies more than one source of concurrent demand.

We choose the second of these options, as it is a useful way of defining the source of concurrency in our world.

A set of top level windows are completely equivalent and unstructured: it is down to the window manager to organise them into a hierarchy for presentation. The windows are also anonymous and have no persistent state: each fresh window starts from a blank, empty state, rather than reusing some previous window's bitmap. We shall return later to see how these windows are created and discarded.

### 6.1.2 FileSystem Resource

Files behave differently from windows, in that they are stateful and persistent entities. In general a process will check out a file to be edited and later update the copy held in the central filestore.

How might we update this model to allow for concurrent updating of files? We have already suggested that a single filesystem process that services arbitrary `ReadFile` and `WriteFile` requests from other processes requires nondeterminism to implement. While we could partition a filesystem and then process individual components concurrently, we would face the same situation when we came to recombine the filesystem into a whole.

The core of the problem appears to be in the client-server model, which encourages nondeterministic access. This leads to the question: how else might we go about organising concurrent access to a filestore resource?

Our salvation comes in the form of lazy evaluation. We can implement a lazy filestore using the functional heap (with possible transparent extensions to disk based store). It is a small leap of the imagination to deliberately embed unevaluated expressions into this heap. These expressions can represent some form of asynchronous computation under the control of separate threads of execution. We then place the filesystem under the central control of a single shell process. This thread can separate out a file for asynchronous editing by another thread, and shared state between the two threads will be used to generate the eventual value in the filestore.

In this scheme, the filesystem processes a stream of `EditFile` requests sent by the single controlling thread. Each request contains the name of a file to be edited, and an unevaluated expression that will become the new value of that file. The filesystem checks out the old copy and replaces its value with the unevaluated expression (this design is heavily dependent on lazy evaluation). The major headache caused by explicit reintegration of files when the client finishes its computation disappears. (See figure 6.2).

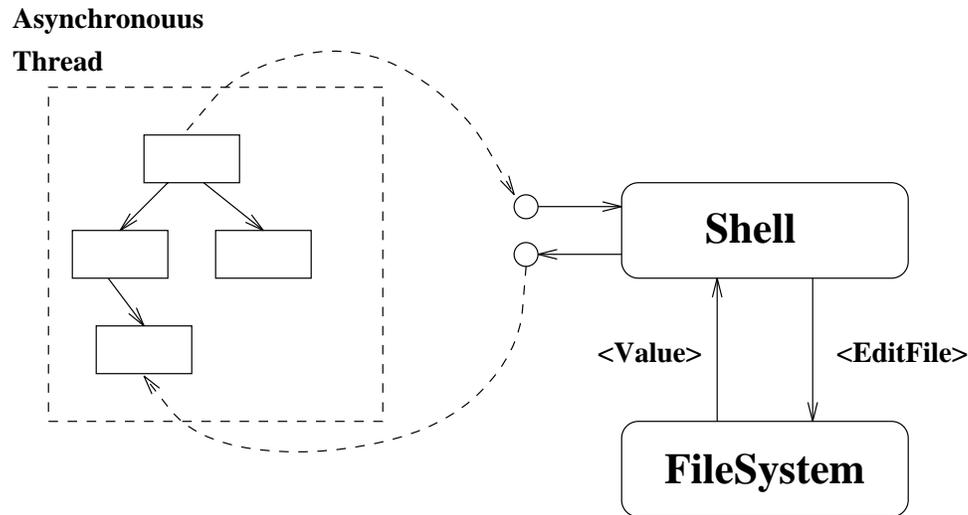


Figure 6.2: The EditFile Request

It is very important that the thread controlling the filesystem does not itself attempt to evaluate the place holder expression. If it attempted to do so, it would suspend until the asynchronous computation completed. As no further asynchronous requests could be processed until this evaluation terminated, the system as a whole would be slaved to a single subprocess<sup>2</sup>.

This favours a design with a file manager process that has direct, sequential, control over the filesystem. The file manager handles all structural changes to the filestore, such as creating directories and renaming files. The file **contents** should only be accessed by asynchronous editing processes. This creates some very interesting and useful behaviour that will be discussed in a later section. For the moment we note only that the file manager has replaced the function of the shell in a conventional text based user interface such as

---

<sup>2</sup>Note: The system would remain determinate and well behaved, only concurrency would be destroyed

Unix. It is responsible for starting new threads of execution and passing in file resources.

The individual user processes no longer have direct access to the filestore. Instead files must be presented by the file manager in a way that does not require a merge of request streams. This requires some form of consistent interface that can be enforced on all user processes. The simplest such interface is to pass a list of files into each user process, and expect a corresponding list of edited files that can be plugged back into the filesystem.

## 6.2 Handling Interruptions

How is an unwanted process interrupted? In traditional systems, this is a particularly dirty business. Typing some special character causes a signal to be sent to the process. If the process is allowed to trap the signal, then faulty processes which ignore it cannot be interrupted; if the process is not allowed to trap it, then correct processes can be stopped in the middle of important updates, leaving an inconsistent filestore or environment behind.

In our setting, it is not possible to regard an interrupt as a signal to the process, because this would have to be merged in a time-dependent way with its other inputs. As an alternative, we provide a complete history of each file and an “undo” operation used to restore a file to a previous state. While this can obviously prove rather expensive in storage, it has the advantage of being completely deterministic: Undo has a well defined effect regardless of whether or not an asynchronous thread is currently editing a file. (See figure 6.3.)

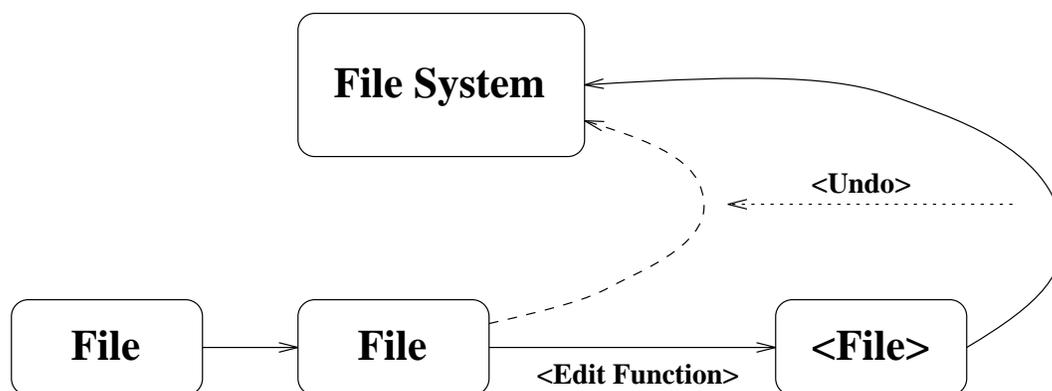


Figure 6.3: The Undo Operation

If the result values of an asynchronous thread are discarded using `undo`, the thread itself and any associated external resources (such as windows and remote connections) can be shut down by the garbage collector.

A `commit` command is used to discard all old versions of files in the filestore, to prevent a huge space leak. This command should be used with discretion, as any subprocesses running at the time of its use cannot be “interrupted” at a later date.

### 6.3 A Design for a User Environment

We now have sufficient background to present the design for a simple user environment based around the concepts introduced in the previous sections, and the concurrency primitives of the last chapter.

The aim is to create a single user working environment that contains a simple filestore and allows asynchronous editing of files. The following design points are raised:

1. The filesystem is controlled by a single thread of execution called the File Manager. This thread is responsible for all structural changes to the filesystem, such as adding directories and renaming files.
2. The manager can run user processes. These are allocated their own independent top level windows. The file manager has no direct access to these windows. Conversely the individual user process cannot access the manager’s window. Consequently we have no output contention.
3. User processes are functions that map file contents and window inputs to file contents and window outputs. They have no dynamic access to file resources.
4. The file manager maintains revision control over files and provides built in commands to cancel the effects of update, even while a process is running. This provides both a useful “restore” facility to the user, and an alternative to process interrupts.
5. The file manager has an associated window. The user uses this to enter commands that either manipulate the filestore or start up asynchronous user processes as described above.

## Simplifications to the model

To keep things simple and clear in the presentation of a prototype system, we can make the following simplifications without loss of generality.

We adopt a flat filestore, i.e. a simple association list mapping between filename and contents. A hierarchical filestore would be very easy to implement, but it would complicate the source code. We further restrict our prototype system to user processes that edit a single input file and own a single window.

Rather than interfacing to displays containing complex bitmaps, we restrict ourselves to simple character terminal-like displays. We use the `xterm` program and a traditional X window manager such as `olwm` to approximate the behaviour of independent top level windows as described in a previous section. Further we restrict the interface of the thread that controls each `xterm` to `[String] -> [String]`, in order to reduce the external extra-functional plumbing to the minimum possible. It is not difficult to create Xlib interfaces using Dialogue structures. Indeed the initial prototype system for this project contains a very simple graphics interface with support for text and lines only. The main complication is efficient processing of bulk data structures [51]. This is not directly relevant to our thesis, so we deliberately sidestep the issue.

While it is difficult to build general purpose, interesting and varied applications with such a simplified model, it should be evident that extensions can be made to both the functional code and support infrastructure presented here in order to support all of the features outlined above.

## 6.4 Overview of Prototype Implementation

In this section we give a brief overview of our demonstration system. The structure is largely defined by the use of multiple concurrent dialogues and the use of an `MakeChannel` style primitive.

The major components in the system are shown in figure 6.4. They are:

1. **FileSystem:** An abstract data type that acts as a database for file objects. The most fundamental activity is the ability to insert a file into the database and later retrieve it. We can build on this base to offer a number of other services convenient to clients.

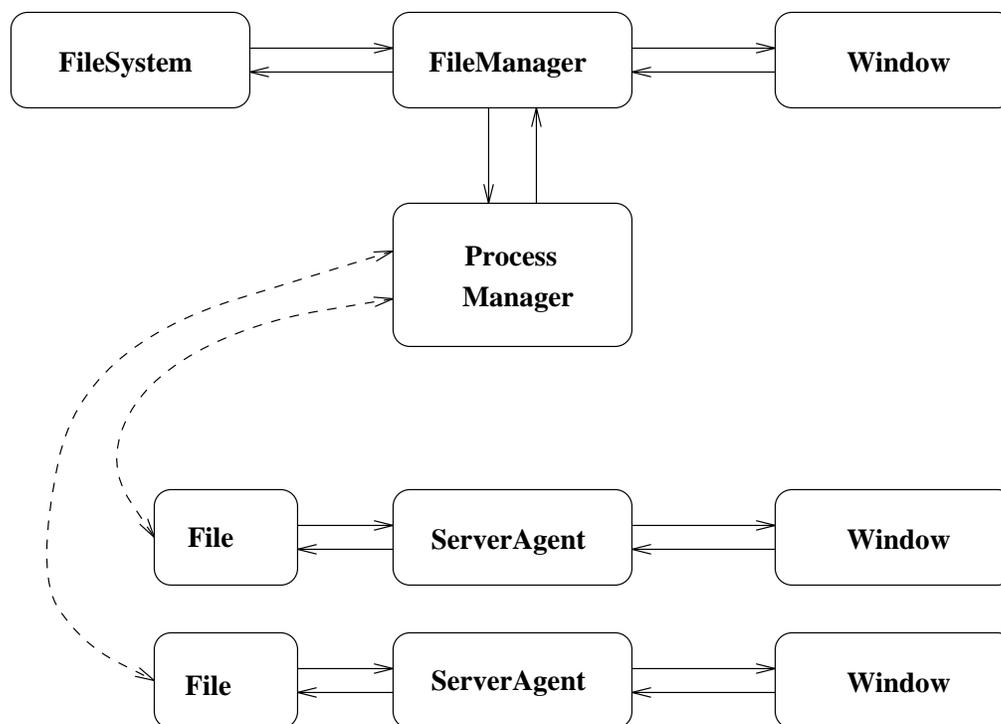


Figure 6.4: Components in prototype system

2. **Files:** Individual files can be detached from the filestore and passed into asynchronous computations.
3. **Windows:** Dialogue style interfaces that maps output text (`[String]`) to input text (`[String]`) for their client.
4. **Process Manager:** Dialogue interface that starts up other asynchronous threads of execution, using the `MakeChannel` operation discussed in section 5.6.2.
5. **File Manager:** The file manager has direct control of the filesystem, and uses interfaces to its window and the process manager to request asynchronous editing for files.
6. **Server Agents:** A set of asynchronous computations that map an input file to an output file depending on the commands entered into that agents window.

Most of this code is simple functional code. The `Process Manager` and each `Window` in the system are procedural support code controlled directly by functional code using

Dialogue style I/O. The essential difference from a conventional functional program, is that we have several independent active Dialogues. Each can be considered to be a source of demand forcing evaluation in the system.

### The Process Manager

For historical reasons, our lightweight processes are termed “agents”. The process manager is a “manager” agent responsible for starting all other “server” agents in the system. The manager agent is directly equivalent to the `MakeChannel` primitive presented in section 5.6.2.

```
type ManagerAgent = [Msg] -> [(String, [Msg])]
```

The process manager contains its own thread of execution that forces evaluation. This generates a list of `(program, outputs)` tuples. `Program` defines an external server process to run and `outputs` is a list of messages to send to that server. This will be run as an asynchronous thread, introducing concurrency into our system. A lazy list of input messages associated with the external server is guaranteed to appear on the dialogue’s response stream. This allows the program manager’s client (i.e. the file manager) to define the server process’s output stream in terms of its input stream and some initial state.

### Server Agents

It is quite possible to consider the `ServerAgents` in isolation. They are simple functional processes attached to terminal I/O Dialogues (powered by demand from the associated `Window`) in a very conventional way. The only complication is that they take an input state (a file) from the controlling process, and are expected to return an expression representing the new state:

```
type ServerAgent = State -> [Msg] -> (State, [Msg])
```

### The File Manager

The `File Manager` process is rather more complicated, as it must interact with two concurrent Dialogues simultaneously (See figure 6.5).

In essence, the interactive window dialogue controls the file manager, sending a series of commands from the user to be obeyed. Certain of these commands will cause activity

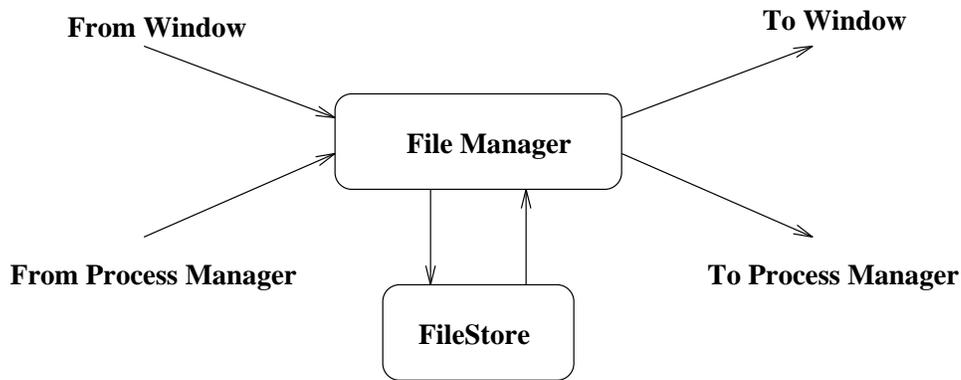


Figure 6.5: File Manager Structure

on the process manager dialogue as additional threads are spawned. Our current shell uses server agents to interface to the external services provided by the process manager. Each of these maps input stream and initial state to output stream and processed state, as follows:

```
(newState, outputs) = "serverAgent" (oldState, inputs)
```

Figure 6.6 shows how this functional code propagates through the process manager to control the external process provided by the process manager.

## 6.5 Structure of the Main Shell Loop

In this section we consider the file manager `shell` function, and discuss some points about its structure. Full functional source code for the prototype system is given in appendix B.

`shellWrapper` is the initial entry point into the file manager and the root of the process manager dialogue. This simply splits off a server dialogue to act as the controlling window for the file manager, and enters the function `shell`. `shell` prints a greeting message to its window, and then enters the main shell loop with an initial, empty, filestore.

The shell “process” then contains two interleaved threads of execution: an interactive section that determines what to do next and the process manager interface that starts asynchronous evaluation. The interaction between these two threads defines the working of the file manager.

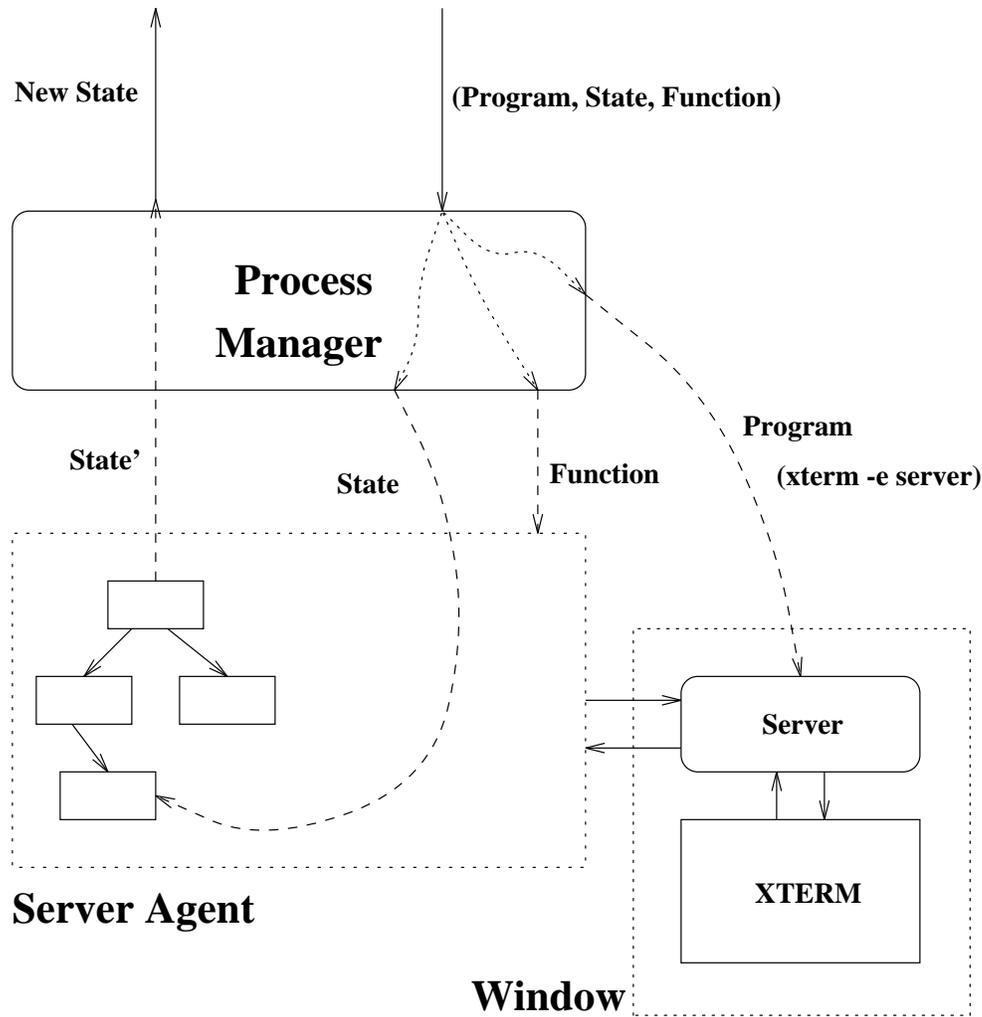


Figure 6.6: Agent Interaction

The main shell loop is written as a set of small cascading functions in an attempt to reduce complexity and simplify presentation:

1. `shellLoop` itself ensures that the command prompt is interleaved correctly.
2. `shellCheck` filters out unknown commands producing a relevant error message and returning to the main loop.
3. `shellValid` determines whether a command is a `FileOp` or an `Agent` and calls `runFileOp` or `runAgent` respectively.

4. `runFileOp` executes the relevant file system operation on the current filestore. It then continues at `shellLoop` with the new file store.
5. `runAgent` executes an agent function that maps an input stream and file to an output stream and file. The input and output streams are plumbed into an external process using the process manager as described in the previous section.

The main complications are plumbing together the result values from two separate I/O Dialogues and dealing with error conditions. An additional problem is that the process manager Dialogue is manipulated at the very core of this code (`runAgent`). Consequently the input and output streams have to be passed correctly through several cascading functions in order to reach their point of application. These are well known problems with the Dialogue style, although it is difficult to see how multiple threads could be incorporated into a continuation based system.

## 6.6 Behaviour of the prototype system

Let us now consider how this prototype system actually behaves compared to a traditional Unix-like shell. The most obvious effect is the clean partitioning of resources. Unix shells were originally designed to work on a single text based terminal shared by all processes including the shell. Typically, the user runs a single “foreground” application that takes control of the terminal for the duration of its execution, while the shell waits for it to terminate before requesting another command. The user can also run background processes that run concurrently to the shell. While Unix shells provide a powerful set of operators for redirecting the processes input and output streams to other files and devices than the terminal, the philosophy is not **enforced**. Consequently there is nothing to stop a user from running several concurrent processes that access the same terminal with interleaved and confusing output:

```
danno[carter]> echo I should be first & echo I should be second
[1] 855
I should be second
I should be first
```

In contrast our prototype file manager only supports “background” processes. No mechanism is provided to force the file manager to wait for a subprocess to terminate, and

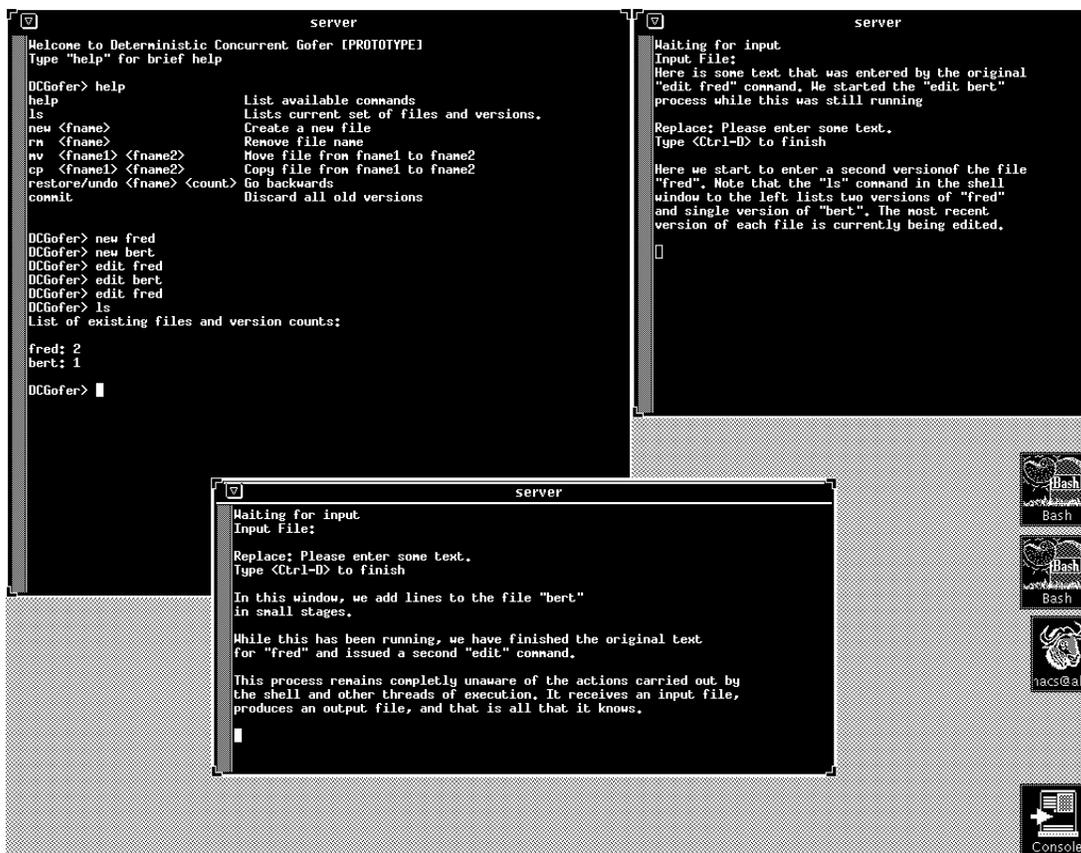


Figure 6.7: Concurrent Applications

in general the file manager is unaware of the current state of processes that it has spawned. Each process runs in its own independent windows so no interleaving effect is possible. The windows are sources of concurrency (See figure 6.7).

While the Unix shell can redirect input and output streams, it has little control over the process it spawns. In particular the individual subprocesses can open files for reading and writing as they choose. In contrast each user process in our prototype system takes a single input file and returns an unevaluated expression that its own editing will determine. Any thread that attempts to access the returned file will be forced to wait until the editing process completes. This enforces a very powerful sequencing effect on editing processes that prevents two processes from trying to manipulate the same version of the file.

Each process puts up a “Waiting for file” message before attempting to access its input file (See figure 6.8). Otherwise it would look very much as if the waiting subprocess had

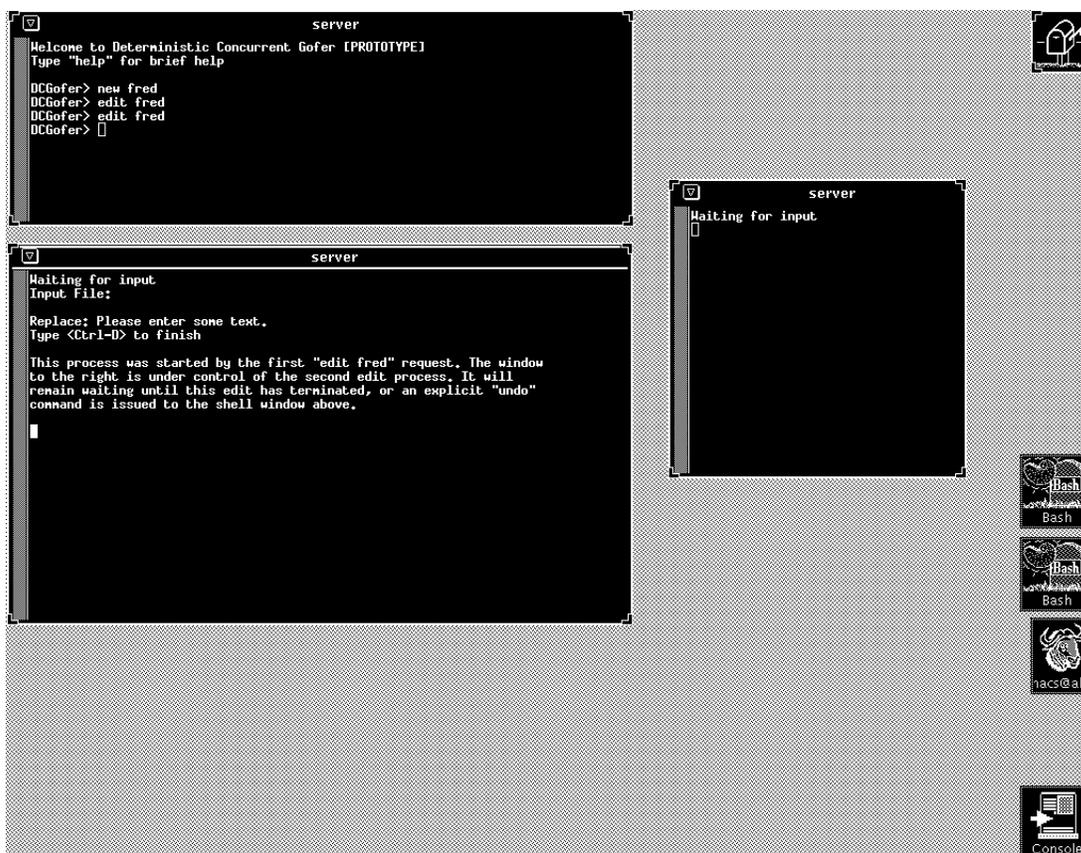


Figure 6.8: Functional Dependency sequences edits

just crashed. Note that the use of a lazy filesystem means that partial results can be passed from one thread of execution to another. This is a very obvious example of concurrency in action. (See figure 6.9).

## Operations that the current File Manager supports

As we have previously described, the file manager supports two separate sets of operations. These are commands that immediately affect the structure of the filestore (`FileOps`) and commands that start up asynchronous computation on an existing file in the filestore (`ServerAgents`)

The complete set of file operations supported by the current prototype are listed in figure 6.10. Currently, only a very small set of `ServerAgents` are supported (see figure 6.11).

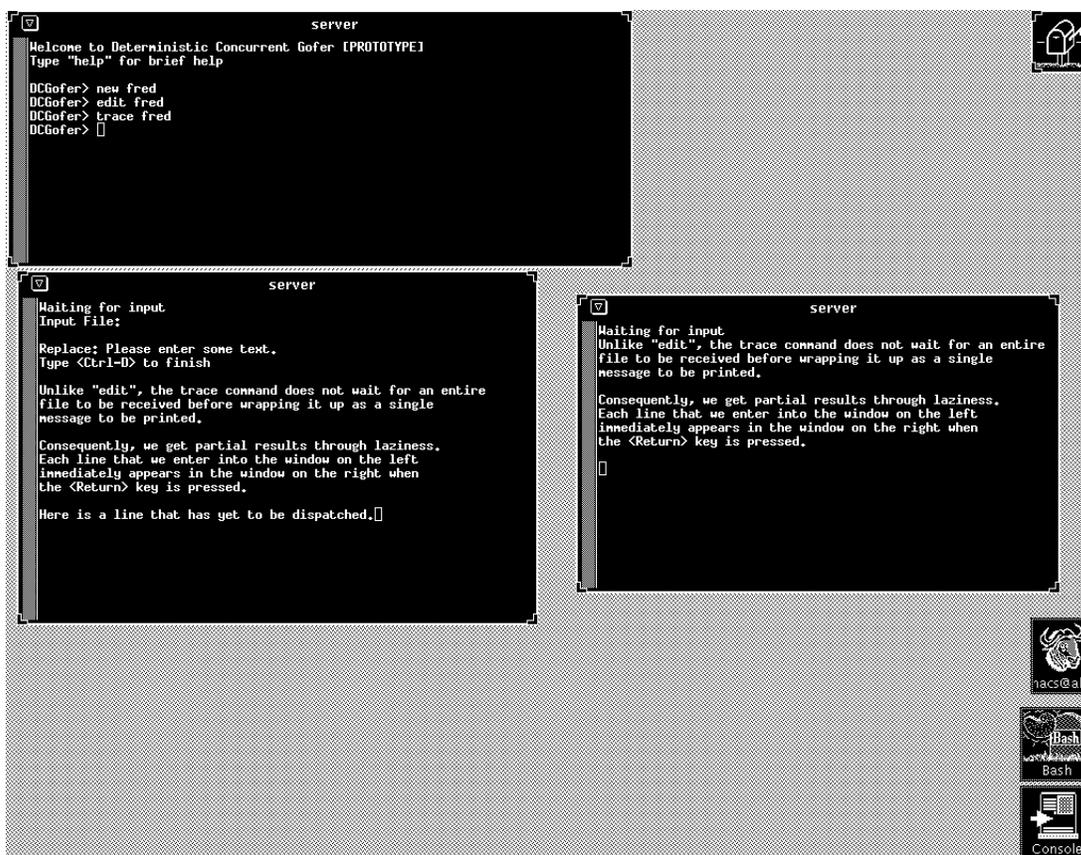


Figure 6.9: Partial Results

`Backwards` and `trace` are used to demonstrate the effects of partial results in the lazy file system.

## 6.7 FileSystem Operations

The only significant abstract data type in the functional code of the prototype system is the file system datatype (`fs.gs`). This implements a flat filestore using a simple association list mapping:

```
data File      = F String

type FileSystem = [(Name, [File])]
type Name      = String
```

Command	Description
help	List available commands
ls	Lists current set of files and versions
new <fname>	Create a new file
rm <fname>	Remove file name
mv <fname1> <fname2>	Move file from fname1 to fname2
cp <fname1> <fname2>	Copy file from fname1 to fname2
restore/undo <fname> <count>	Discard count versions of file
commit	Discard all old versions

Figure 6.10: File Operations supported by the shell

Command	Description
edit	Show and then discard file contents. User enters new file
backwards	As edit, reverses output lines
trace	Show reverse file contents

Figure 6.11: Server Agent Operations supported by the shell

```
emptyFS = [] :: FileSystem -- The initial FileSystem
```

The operations that this filestore supports are list in figure 6.12.

Apart from the operations with explicitly labelled types, all of the above operations return an object of type `Return FileSystem`. The `Return` and `MaybeError` types are systematic approaches to handling error messages that respectively augment and replace the normal result type.

```
data MaybeError a = Ok a | Error String
data Maybe      a = Yes a | No
type Return     a = (String, a)
```

Function	Description
fileList fs	Map fileSystem to list of (name, version count) pairs, of type [(Name, Int)]
fileLookup fs name	Return most recent version of file. Return type: MaybeError File.
fileExists fs name	Determine if file with given name exists. Returns type Bool
fileCreate fs name	Add empty file to filesystem. Error to create twice
fileUpdate fs name file	Add new version to file in filestore
fileDelete fs name	Delete file from filesystem
fileRename fs oldname newname	Rename file from oldname to newname. Existing overwritten
fileCopy fs oldname newname	Make copy of file includes entire version history
fileRewind fs name count	Discard given number of new versions of a file.
fileCommit fs	Discards all old versions of files

Figure 6.12: FileSystem operations

# Chapter 7

## Overview of Support System

In this chapter we consider the support infrastructure required to implement a concurrent graph reduction machine that communicates with external processes. Figure 7.1 shows the interactions between the existing Gofer run time system and the concurrency subsystem. The model consists of the following components, that will be discussed in later sections:

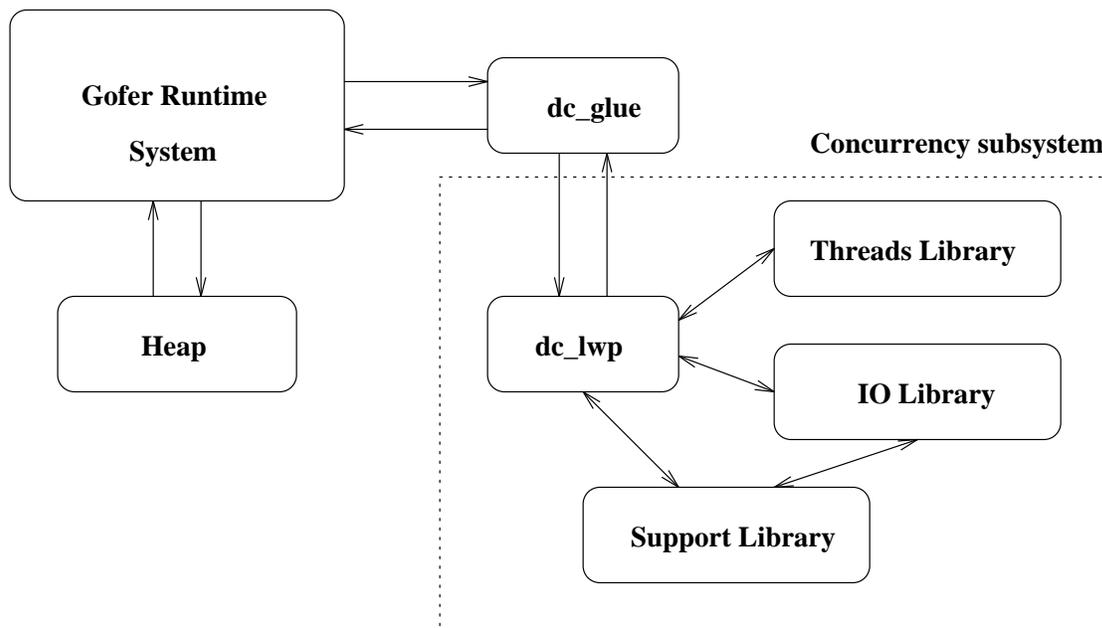


Figure 7.1: Structural overview of Implementation

1. The existing Gofer run time system, with minimal changes to support concurrent evaluation within its heap.
2. A concurrency subsystem that provides concurrency primitives targeted at supporting a functional interpreter. This consists of a single library `dc_lwp` formed by combining threads, asynchronous I/O and support libraries with a set of Dialogue support functions.
3. A glue module `dc_glue` that interfaces the two major components. The idea is to reduce the complexity of the glue module by presenting a set of concurrency primitives well matched the the needs of the existing interpreter.

Appendix C contains C source header files that define the functions provided by the `dc_lwp` library.

## 7.1 Implementing a concurrent interpreter

The evaluation engine in the conventional Gofer interpreter consists of a Gcode interpreter, special code for primitive operations, and support for heap data structures, all written in C. Fortunately for us, the highly recursive nature of functional evaluation means that the vast majority of the state associated with a particular evaluation is “local data” stored on the machine stack. While the process of compiling a Gofer script sets up a large number of read only data tables, the only significant global read/write structure is the functional heap.

This makes the migration from a single thread of execution representing sequential graph reduction, to multiple threads of execution, each representing an independent graph reduction process much easier. As we shall see, the heap is very well behaved for such a large data structure and with suitable care can be accessed and updated concurrently without the individual threads needing to be aware of one another. In effect, a concurrent Gofer interpreter is no more than a single C program containing multiple threads of execution, each attempting to be a simple sequential interpreter with a certain amount of shared state.

An important consideration in implementing this system is that our design states that no thread should ever block the execution of another active thread. This implies that we

either need some method of preemptive multitasking between threads or a reliable way of ensuring that threads yield voluntarily. A related consideration is that individual threads cannot performing blocking I/O operations as this might indefinitely stop other threads from evaluating.

More detailed discussion about the implementation of a parallel G-machine can be found in chapter 5 of [35].

### 7.1.1 Shared heap

As we have seen, sequential evaluation in a functional language is powered by replacing a single expression at a time using a substitution engine. Intuitively, concurrent evaluation involves making several substitutions simultaneously. Where no shared values are involved this is very easy: we simply use a number of threads of execution, each of which consists of a sequential graph reduction machine with its own private data structures.

This process is complicated where a single heap is shared between the threads of execution. If no control is enforced between the separate evaluation machines, the possibility arises that one thread will access a value while another is attempting to update it. As in the procedural case, this introduces the possibility of nondeterminism.

Fortunately, a heap containing data structures for a functional programming language is a very well behaved entity, that only supports the following operations:

1. Allocate cells for new graph from a common pool.
2. Update a cell with an indirection pointer to new graph that has been constructed by an outside entity.
3. Garbage collect the heap. In the Gofer implementation, this involves finding all unreferenced cells and linking them into a free list without touching “live” cells.

In order to construct a piece of graph, a thread of execution will acquire a number of cells for private use: other threads have no access to these cells. Once this graph has been constructed it is advertised back to the other threads using an update operation.

This acquire, construct and update cycle is only used to replace an expression with its evaluated form. Equational reasoning allows us to treat the evaluated and unevaluated forms as equivalent: even if the expression is evaluated independently several times, we

know that the same result will be forthcoming. Therefore, so long as the allocation and update procedures are atomic, the heap can be both concurrently accessed and updated in safety.

Of course this is not very efficient: if two threads try to access a single expression at the same time, evaluation will be repeated and the root cell will be updated twice with separate, but equivalent and interchangeable result graphs. A better solution is to serialise access using simple interlocks. Whenever a thread attempts to evaluate an updateable cell or CAF, it is locked in that thread's name. If any other threads attempt to access the same expression before its evaluation is complete (i.e. the update phase has yet to take place), their execution is suspended. The update operation is then augmented to wake up any threads that are suspended on each shared value.

The simplest way of implementing this scheme is to associate a linked list of threads to each cell in the heap. As each thread can only suspend on a single cell at a time, it is possible to allocate a single structure for each thread and then chain them together as required with very little space overhead (See figure 7.2)

The locking scheme has a small bonus of detecting black holes in the evaluation process: if a thread attempts to lock a cell that it has already locked once before, an infinite loop of evaluation is involved and the interpreter can take appropriate counter-measures.

### 7.1.2 Types of Dialogue

The unit of concurrency in our world is the thread. However this is a very low level concept. We have deliberately presented a model where the threads corresponded to first output streams and later Dialogue pairs, in an attempt to create a higher level representation of concurrency that is directly meaningful (indeed trivially so) to the functional programmer. A dialogue consists of a pair of input and output streams. We bind a thread of execution to a dialogue to power the evaluation of the output list. This may involve evaluating elements from the input lists of this and other dialogues.

The prototype system described in chapter 6 is based around two types of dialogue interface. A single `ManagerDialogue` is responsible for starting up other dialogues in the system. There is no reason that a single `ManagerDialogue` could not control a number of different types of slave dialogue. However, the only slave dialogue type supported by the prototype implementation is `ServerDialogue`. These control an external process through

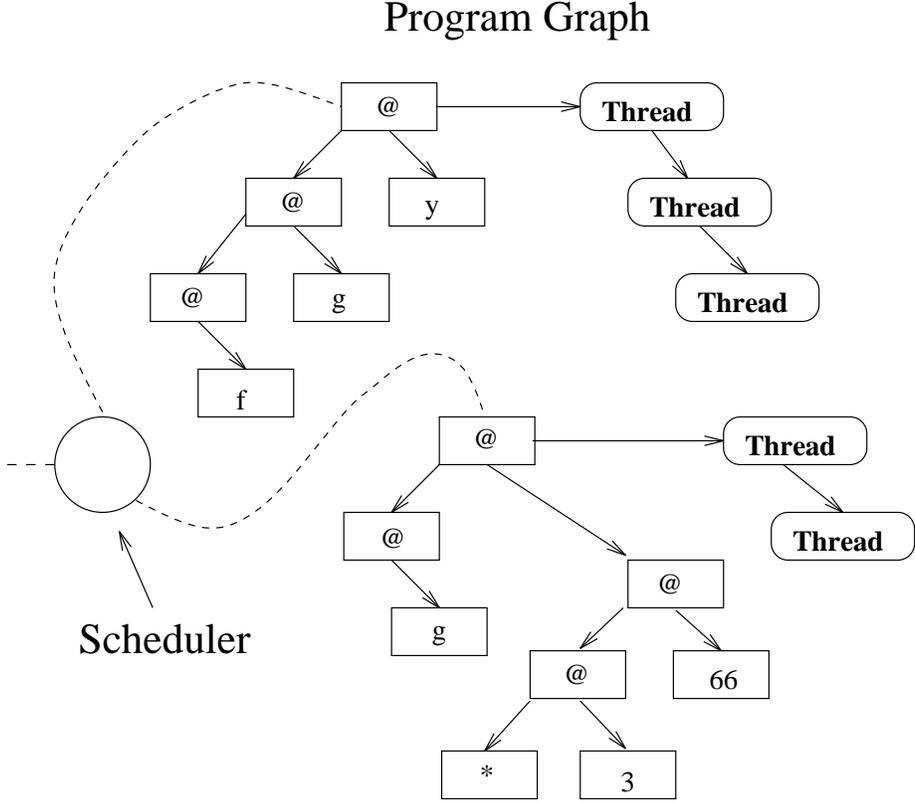


Figure 7.2: Cell locking

a two way communications channel that maps a list of input messages to list of output messages. `ServerDialogues` are consequently responsible for all the I/O behaviour in our program.

The thread controlling the manager carries out the following procedure for each element in the output list.

1. Evaluate a dialogue `Request`. This is a tuple `(program, outputs)` of type `(String, [Msg])`.
2. Start an asynchronous thread that is bound to an external process defined by `program`. `outputs` defines the output list for this dialogue.
3. The server thread will immediately return a handle to a list `inputs` of input messages for that dialogue

4. Bind this list into the response stream associated to the Dialogue. It can then be used to control the output list through functional code.

Because each output request generates exactly one response, the `ManagerDialogue` C function is almost an exact copy of the existing Gofer Dialogue code: instead of calling C functions to perform I/O actions, functions in the `dc_lwp` library are called. As we shall see, `ServerDialogues` are a little more involved because they interface to an asynchronous I/O system. However the viewpoint of the functional language programmer remains clear and simple.

## 7.2 Lightweight Threads Library

An essential requirement for the implementation is the availability of a lightweight threads mechanism that can be interfaced to the C code of the Gofer interpreter. Sun provide their own threads library on the SunOS 4.1.3 systems used to implement our prototype system. However code written using this library is not portable to other brands of Unix. This library is also rather large and over-featured for our requirements.

As an alternative we use a freely available and very portable library provided by Stephen Crane of Imperial College's Department of Computing ([jsc@doc.ic.ac.uk](mailto:jsc@doc.ic.ac.uk)). The library was originally designed to provide concurrent features in the Rexx programming language, but has provided to be highly adaptable. It has also proved very easy to port this library to unsupported architectures.

In the following section we give a brief overview of the features provided by the threads library. We proceed to explain how problems associated with timer and I/O signals force us to build our own I/O subsystem and use only a limited set of the library features. We also explain how the library has been augmented to provide a simple state paging mechanism.

### 7.2.1 Features of the threads library

Unsurprisingly, the threads package allows a number of lightweight processes to coexist within a single heavyweight process. At any time a single thread will be executing, the others may either be ready to run or be suspended. Global variables are shared by all threads of execution. In addition, each thread has its own independent machine stack and a current "environment" structure whose purpose is defined by the client programmer. To

all intents and purposes each thread behaves as a separate C program, with the caveat that any global variables may be unexpectedly changed by another thread.

A simple scheduler is provided that allows transparent switching between threads. Switching is not generally preemptive: each thread must explicitly yield control of the processor. In doing so, a thread may request to be paged back in at some later point in round robin style. Otherwise it is suspended until some other thread wakes it up again. A timer mechanism allows a thread to be suspended for (at least) a certain amount of time before being added back onto the run queue.

This model is enhanced by grouping threads into 8 priority levels. High priority threads preempt and completely block the execution of low priority ones when activated. Combined with the timer mechanism this provides a very simple way of providing preemptive scheduling between threads on the same priority level: a high priority process that wakes up once in a while, but does nothing at all, will automatically schedule the threads at a lower level in round robin fashion. Providing more than two priority levels allows services to block this preemption.

The library also contains limited support for asynchronous I/O. The need for this, and some problems with the provided system are discussed in a later section.

### 7.2.2 Local state extension

The main fly in the ointment is that Gofer was not originally designed to be a concurrent evaluation engine. In particular it has no concept of the environment mechanism provided by the threads library, and uses global variables to store a number of important values that become per thread entities in the concurrent world:

Variable	Description
<code>void *cstackbase;</code>	Base of machine stack for this thread
<code>void *cstacktop;</code>	Top of machine stack for this thread
<code>Cell *cellStack;</code>	Gmachine stack array
<code>int cellSp;</code>	Gmachine stack pointer
<code>int threadCount;</code>	Count reductions between context switches

If the Gcode interpreter is to function properly, these variables must be transparently paged in and out as context switches occur between threads. The simplest way to do this is

to define `pagein()` and `pageOut()` functions for each thread that are invoked as switches take place. These functions receive a pointer to the thread's local environment structure, in which space is set aside for the thread's own values for the above variables. In this way a set of global C variables can be multiplexed between a set of threads.

### 7.2.3 Asynchronous I/O

A major problem associated with the use of lightweight threads is that input and output channels provided by the heavyweight process are shared by all the threads. If one thread makes a system call that blocks, all other threads will suspend. This destroys concurrency, and can easily lead to deadlock in demand driven process networks.

The simplest solution to this problem is to place file descriptors into a non-blocking mode. In this state, a system call that would block fails with an error value indicating that the attempt should be repeated at some future point. File descriptors must be polled at regular intervals, either individually or using a non-blocking `select()` call in order to determine where forward progress can be made.

Explicitly monitoring the state of communications ports using a busy wait is clearly inefficient. A better solution is to use asynchronous, or interrupt driven, I/O. "SIGIO" signals can be generated whenever a process's I/O status changes. A suitable handler routine can then examine the new status using non-blocking data transfer or select statements as before. A disadvantage of using signals, for either I/O or timer events is that any system call in the program can be interrupted, and must be prepared to recover and retry its operation.

The threads library provides simple interrupt driven support for input channels only. However this suffers a major flaw, in that interrupts are only enabled when all normal threads in the system are sleeping, causing a special low priority "null" thread to run. We believe that this was required in order to avoid race conditions in the threads support system. Unfortunately this means that we cannot use this code: our design requires us to be capable of transferring data when runnable threads are present, both to avoid deadlock situations and ensure fairness.

## 7.3 Handling Input and Output

A system that explicitly avoids the use of interrupts must have centralised control of I/O in order to efficiently use `select()` calls rather than blind polling. If the program contains multiple threads of execution, the only effective way of doing this is to concentrate all data transfer into a single thread. This favours a design with a ring of same priority processes, one of which is a dedicated communications thread. Whenever this thread runs, it performs a complete communications cycle, moving as much data as possible to and from internal data structures without blocking. It then schedules itself to run the next time around the round robin loop and yields control of the processor. See figure 7.3

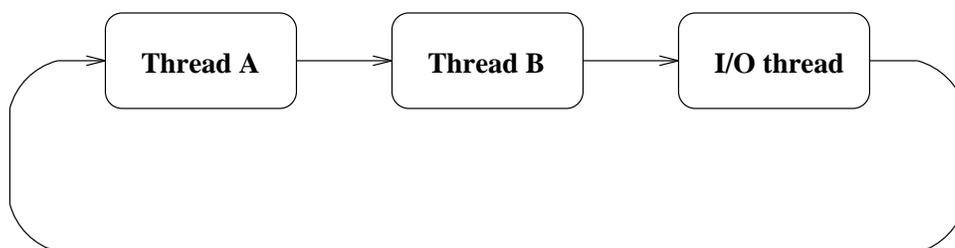


Figure 7.3: Round robin scheduling with I/O thread

Our design states that no thread should ever block the execution of another active thread. As the timer mechanism is considered unsafe, we cannot implement preemptive scheduling in the way outlined earlier. Instead each thread must guarantee that it yields control of the processor after a finite time. This is quite simple to achieve in the context of a concurrent graph reduction machine: each evaluation thread simply counts reduction steps and yields at regular intervals. A non-blocking I/O thread is known to be safe as it will only take a finite amount of time to check the status of its channels. To make it even more robust, our I/O subsystem is nonblocking in both input and output directions. This should avoid some spurious deadlock situations that might otherwise arise. Of course if the communications thread is the only active thread in the system we know that it will poll continuously, and it is safe to perform a blocking `select()` as an optimisation.

The need to poll for I/O behaviour makes our implementation less efficient than one based on interrupts. However, if the original author of the threads system felt that unrestricted use of interrupts was dangerous in his code, maybe this is a safer route.

Another strong motivation for using a centralised I/O system is that it is not generally safe to allow concurrent threads to have shared access to a communications channel. If two threads write to a single output channel, the output stream will depend on the ordering of scheduling. This is clearly nondeterministic.

A clear advantage of using demand driven Dialogue communications is that they remain deterministic in a multithreaded environment: each output channel is defined to be a single thread of execution, while input messages are placed in a shared list, where they can be accessed safely by several different threads.

### 7.3.1 Interaction between ServerDialogues and the I/O thread

As we have seen, `ServerDialogues` are used to interact with an external server process using two way communications streams. If each dialogue thread was responsible for its own I/O things would be very simple: each output message would be written directly to the output channel as it is generated by its controlling thread. This thread would read input directly from dialogue input ports, blocking if no events were available. The need to centralise I/O in a single thread complicates matters. We need to define the flow of data between the evaluation and I/O threads through some kind of shared data structure.

Each server dialogue structure provides event input and output queues to be shared with the I/O thread. As the dialogue evaluates elements, as they are added to the output queue. When the I/O thread runs it tries to send as many elements as possible from the output queue. At the same time it tries to read input events, adding them to the input queue. No restrictions are placed on the length of either queue. See figure 7.4. The only complication is when a dialogue thread attempts to read from an empty input queue. In this situation, the thread must suspend. It is the responsibility of the I/O thread to wake up any threads waiting on an empty input queue when messages finally arrive. Each dialogue thread sees only an infinite stream of input events with possibly erratic timing: this is a completely declarative view of I/O.

The low level plumbing provided by the companion thread that controls the flow of data to and from queues is completely transparent, indeed the dialogue is completely unaware that it is performing I/O operations by proxy. In this way we provide a high level dialogue style I/O system, hiding all the complications of buffering over non-blocking channels.

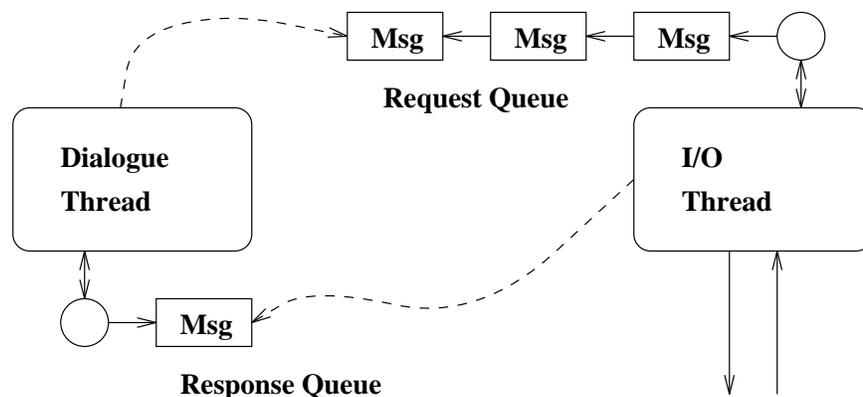


Figure 7.4: Symmetry of I/O queues

### 7.3.2 Providing Reliable Buffered Message Streams

Unix systems provide communications channels based on byte streams. We use Berkeley sockets that provide bidirectional transmission on each channel. There is an obvious correspondence between this and the character based dialogues of a language such as Miranda. Unfortunately the use of non-blocking I/O streams means that we cannot directly use the buffered I/O routines provided by the standard C library, they cannot deal with partial results caused by non-blocking I/O. One solution would be to send and receive each character individually using system calls, using a single character buffer at the sending end in case I/O fails. However this is horribly inefficient as each system call has a high overhead. We really need to provide a buffered I/O service over the non-blocking channel.

From the viewpoint of the functional programmer, I/O is controlled by the individual Server Dialogues mapping request events to response events. Potentially, different types can be used, so we use a wrapper system. Types that are instances of class `MsgAble` provide conversion functions to hyperstrictly convert their value to and from a `Msg` abstract data type that can be represented as a byte stream and transferred across the “wire”. The only `MsgAble` type currently provided is `[Char]`:

```
class MsgAble a where
  toMsg      :: a    -> Msg
  fromMsg    :: Msg -> a

instance MsgAble [Char] where
  toMsg      = primCharsToMsg
```

```

fromMsg = primCharsFromMsg

primitive primCharsToMsg    "primCharsToMsg"    :: [Char] -> Msg
primitive primCharsFromMsg  "primCharsFromMsg"  :: Msg -> [Char]

```

The Gofer keyword “primitive” declares that the operation is implemented using a C function of the same name. The `Msg` type can also represent an end of file value. This can obviously be used to convert an infinite input stream into a finite list.

```

isEOFMsg    :: Msg -> Bool
isEOFMsg    = primIsEOFMsg
primitive primIsEOFMsg    "primIsEOFMsg" :: Msg -> Bool

```

From the viewpoint of the I/O subsystem, each message is a `malloc()`ed byte array with the following header:

```

struct Msg {
    struct Msg *next; /* So that we can link in a chain */
    int    number;   /* Sequence number */
    int    flags;    /* Currently labels EOF Msg only */
    int    len;      /* Followed by message of length len */
#ifdef __GNUC__
    char   body[0]; /* A placeholder for simpler debugging */
#endif
};

```

As we have seen, each `ServerDialogue` has two associated queues of I/O events. Requests are queued as output events until the I/O threads can process them. Messages delivered from the I/O thread are stored on an input list until some evaluation thread (not necessarily the one linked to the `ServerDialogue`) processes them as requests. Both queues are potentially infinite.

Messages are therefore transferred by sending a header block followed by an arbitrary number of bytes. The major complication is that we can never guarantee that an I/O system call will actually transfer bytes in nonblocking mode. Consequently we have to store the status of partially transmitted messages (for both directions) in each `ServerDialogue` structure.

```

struct MsgHdr { /* Need this much before we can malloc & fill */

```

```

    u_long  number; /* Encoded sequence number */
    u_long  flags;  /* Encoded flags */
    u_long  len;    /* Encoded Length [body only] */
};

typedef enum {NONE,HDR,BODY,COMPLETE} PartialState;

struct PartialMsg {
    PartialState  state;
    int           hdrXfer; /* Amount of hdr read/written */
    struct MsgHdr hdr;    /* Size = HDRSIZE */
    int           msgXfer; /* Amount transferred */
    int           msgSize; /* Complete size of message */
    struct Msg    *msg;
};

```

Because the msg header and body are stored separately, two separate operations are required to transfer data to or from a buffer or I/O channel. Either of these operations may complete, partially succeed or fail completely. This leaves 4 possible states:

1. NONE. No data was transferred.
2. HDR. The header is partially transferred.
3. BODY. Header has been totally transferred, body partially.
4. COMPLETE. Message was completely transferred.

The receiving end may receive several messages in a single `read()` to a statically sized buffer. In general it will continue (but possibly not complete) reading a partially completed message, followed by zero or more complete input messages, followed by a partial input message. The important point is that it always ends up with at most a single outstanding partial input message. The situation at the sending end depends on the buffering policy. If we send each message individually, writing the header block and message body as two separate `write()`s we have at most a single outstanding message. If instead we pack a number of messages into a “bounce buffer” of static size, a partial `write()` will leave many messages in the buffer, as well as the outstanding `partialMsg` left over from filling the cache. The extra complexity has the advantage that we are guaranteed that the minimum

number of system calls will be used to transfer a list of messages from one process to another.

One minor point is that `Msg` structures should be copied rather than unlinked when moving a message to or from the heap from the `ServerDialogue` and I/O subsystem. This is because the garbage collector attempts to free any `Msg` structures that are not referenced by the functional subsystem. Sounds obvious, but it took me the best part of a week to track down a bug caused by ignoring this simple rule . . .

## 7.4 Garbage collection

The Gofer interpreter uses a traditional two phase mark-scan garbage collector<sup>1</sup>. The first phase involves marking all of the cells that are accessible, either directly or indirectly, from a collection of root cells. The garbage collector then scans through the heap looking for cells that are not reachable and that can consequently be linked directly into a list of free cells. The root cells used include expressions referenced by: the current value, all CAFs, and many tables used by the Gofer compilation process.

In the sequential case, the garbage collector must also mark all expressions available from the G machine cell stack and the hardware machine stack in order to reach intermediate values generated by the Gcode and C support environment. Marking the C stack is a particularly unpleasant and unportable concept. It works only because the run time system is designed to store only primitive data values and pointers on the C stack. The garbage collector does not have enough context to determine which of these values are genuine heap references. Therefore it must treat all values as potential references and mark them. We say that the garbage collector is “conservative”: it does not guarantee to find all of the free cells at a given point.

A concurrent G machine must mark the cell stack and C stack for each evaluation thread in the system, including the currently active thread that forced the garbage collection. The simplest way of doing this is to start up a special purpose garbage collection thread. This causes the currently active evaluation thread to page out, storing its local state in an environment structure, as described in the previous section. The garbage collection thread can then access all of these structures to mark the cell and C stacks of all evaluation

---

<sup>1</sup>The GofC compiler also provides a two space compacting collector although this is incompatible with the cell locking mechanism that the concurrency library employs

threads. This has a secondary advantage, in that the mark phase as implemented is highly recursive and requires a large machine stack. Using this scheme, we can allocate a large, temporary stack for the garbage collection thread and use much smaller stacks for the evaluation threads.

We rely on garbage collection to free up state and system resources associated with the discarded threads. In the current prototype, windows are represented using special magic numbers (integers) that point to Dialogue structures outside the heap. Therefore, the garbage collector does not have enough context to discard unreferenced windows in the same way that it can close unreferenced files. All that would be required to implement this concept is a special representation for window values in the heap akin to the existing use of file cells. The garbage collector would then be adapted:

1. Mark all cells reachable from the output expression of Manager Dialogue(s).
2. Any attempt to mark window cells causes the associated Server Dialogue to be labelled as “currently active”. The output expression for its dialogue is then marked.
3. After the mark phase has completed, any Server Dialogue that remains registered but unmarked is no longer referenced. The interpreter can close down the window and free up all resources associated with that thread.

This would be quite simple to implement, but there are more urgent priorities in the writing of this thesis. Garbage collection of windows is principally needed after an “undo” operation is used to discard computation. In order to close down the windows immediately after an `undo` operation as required, we would require a special strict `garbageCollect` primitive function of type `a -> b` that would force a garbage collection before proceeding.

## 7.5 Server Processes

The server process that we have defined acts as an interface to an `xterm` window. This can be a standard data driven procedural process as we have managed to contain the source of demand within the Gofer interpreter (See section 5.4.2). The only complication is that the process must be able to deal with input values from both the keyboard and controlling process at any time. Consequently it must use interrupt driven I/O rather than blocking

systems calls. This is quite safe as the server process does not rely on the threads library: it is a traditional procedural process that contains a single thread of execution.

There is no reason that other procedural services, such as databases and network connections could not be ‘wrapped up’ in the same way to provide well behaved functional components. The main problem is in ensuring that separate components remain independent and that we do not introduce implicit nondeterminism. We would have to rely on good design and intension, as this could prove quite difficult to enforce in the general case.

# Chapter 8

## Conclusions

Most previous “functional operating system” projects have attempted to emulate all the features traditionally available in an operating system environment. The main rationale appears to be a desire to show that functional programming languages are as versatile and adaptable as their procedural counterparts. This is perfectly true, however these goals can only be achieved by compromising the ideals of mathematical purity and clean design that functional programmers hold so dear.

In retrospect, it seems as important to find models for concurrency and network communications that preserve simple static semantics. One of the main initial motivations of this project was to demonstrate that a great deal of the nondeterminism used in traditional systems is unnecessary and can be eliminated with sufficient forethought in the design process. If nothing else, it is hoped that this thesis might challenge the automatic assumption that nondeterminism is required in system design.

While functional languages provide a very expressive programming notation, they generally offer poor support for interactive applications. The main reason for this is the guest status of a functional process in a procedural operating system. Declarative mechanisms for I/O control, such as dialogue or continuation style only allow for a single point of interface between a functional process and procedural services. This makes it very difficult to form meaningful high level abstractions for I/O behaviour. While functional languages promote modular design, interfaces to external resources are generally very messy, even when continuation passing or monadic style is adopted.

This problem is greatly exacerbated when a sequential functional process attempts to interface directly to a set of concurrent services. A large amount of support infrastruc-

ture is required to deal with tagging and untagging of messages required by the use of a nondeterministic merge operator on the input channels. Worse, we can no longer presume that request and response values are directly paired, which makes it very difficult to use a continuation or monadic style.

We have suggested that these problems are largely caused because we attempt to carry across a great deal of conventional operating system philosophy and wisdom. The most damaging of these philosophies is the use of input driven heavyweight processes that contain a single thread of execution. This immediately requires the use of a nondeterministic merge operation that destroys the pure mathematic properties of functional code. As an alternative, we have investigated a demand driven model of communications that naturally supports concurrency within functional processes without recourse to nondeterminism. This allows the use of separate Dialogue interfaces to communicate with logically separate and concurrent resources.

The extra expressive power provided by multiple Dialogue communications allow us to construct a completely deterministic concurrent user environment. While the demonstration system presented in chapter 6 is very much a minimalistic prototype, it does show that many common concepts can be redesigned to fit within a deterministic world.

### **Future Directions**

An issue that is not addressed by this thesis is the matter of programming style in the face of multiple concurrent dialogues. It is possible that continuation passing or monadic style might be applied to individual dialogues where a strong pairing between elements of the input and output streams was enforced. However, it is rather less obvious as to how independent threads share state in this situation.

It is rather unfortunate that so few tools for investigating concurrent functional user environments exist. This project has required a substantial investment in support infrastructure to build only a simple prototype system. While it might be an interesting exercise to build a more complete environment around the current prototype interpreter, we do not feel that it would add to our presentation. In order to construct a generally useful deterministic environment containing development tools such as a compiler we would need to add a concurrent support environment to one of the existing Haskell compilers. This would appear to be a very large commitment, especially given the current prototype nature of these compilers.

A very interesting long term goal would be to see just how far the concept of deterministic design could be extended. The emphasis would be on finding safe models of concurrent access to shared resources, particularly challenging in the case of multi-user systems.

# Appendix A

## The Gofer G machine instruction set

Instruction	Description
EVAL value addr	Evaluate the expression pointed to by the top stack item. The spine nodes of the WHNF expression are added to the top of the stack, so that further Gcode instructions can build expressions from the result. This scheme is slightly complicated by the fact that each spine group with the stack frame are pushed from right to left, and are therefore numbered “backwards”.
RETURN	Return to the unwind code for the current expression. This will perform repeated unwinds until the expression is in WHNF.
TEST value addr	Test the tag value of the last WHNF to be generated. If equal to “value”, then execute the following code. Otherwise jump to the address given.
GOTO addr	Branch to the given address.
FAIL	Enter error handling code. In addition a branch to the location 0000 counts as an error.
INTEQ value label	Test the most recent WHNF “value” that was an integer against the value given. If the value is the same, execute the following code. Otherwise branch to the code at label given.
INTGE value label	Test the most recent WHNF “value” that was an integer against the value given. If the WHNF value is greater than that that given, create an integer value onto the stack that is the difference, then execute the following code. Otherwise branch to the code at label given. Note that only INTEQ and INTGE are required - the G machine compiler can reverse function arguments when required to get the symmetric results.
INTDV value	Test if the most recent integer WHNF is a multiple of “value”. If so, execute the following code, otherwise branch to the label given.

Figure A.1: Control Flow Operations

Instruction	Description
ALLOC <i>i</i>	Allocate <i>i</i> heap cells and push their addresses onto the stack.
LOAD <i>i</i>	Push a copy of the $i^{th}$ entry on the current stack frame. Entries 0..n-1 are the function arguments, higher numbers correspond to values pushed on the stack by Gcode or recursive evaluation.
MKAP <i>n</i>	Create a spine chain from the top <i>n</i> +1 elements on the stack. These elements are replaced with a single pointer, to the root of the new application.
UPDATE <i>i</i>	Pop the top element on the stack, and replace the heap node pointed to by the $i^{th}$ stack entry (from the current root) with an indirection to this node.
UPDAP <i>i</i>	As update, but replace the node with an application built from the top two stack items.
SETSTK <i>i</i>	Discard all stack values in the current stack frame apart from the first <i>i</i> . This instruction is used to discard WHNF information generated by recursive evaluation.
ROOT <i>i</i>	Push a pointer to a partial application of the current expression. For example, ROOT 1 of “f x y z” produces a pointer “f x y”. This instruction is strictly speaking redundant, but it is useful for generating efficient code for recursive functions where the only argument to change is the last. See the example code given for map.
SLIDE <i>i</i>	Move the top stack ptr <i>i</i> elements up the stack, discarding all the elements that it passes. Used to discard temporary values on the stack.

Figure A.2: Stack and Heap Manipulation Instructions

Instruction	Description
CELL <i>i</i>	Push the global value <i>i</i> onto the stack.
CHAR <i>i</i>	Push an unboxed character representation onto the stack.
INT <i>i</i>	Either push a pointer to a boxed integer cell or a unboxed representation for small integers.
FLOAT <i>f</i>	Push a pointer to a boxed floating point number.
STRING <i>text</i>	Constant strings are stored in a shared pool area. This instruction creates a cell that points to a string in this pool. When the eval call encounters a such a cell, it transforms in in stages into a [Char] representation that is guaranteed to be in WHNF. For example, a cell created by STRING (pointer to) “fred”, would evaluate first to ‘a’: “red”.
DICT <i>i</i>	Replace the top cell on the stack with its $i^{th}$ dictionary value. This is used to implement type classes.

Figure A.3: Primitive Data Manipulation Operators

# Appendix B

## Source code

This appendix contains complete Gofer source code for the prototype system. This consists of the following modules:

1. **ui.gs**: User interface including file manager and server agents
2. **io.gs**: Interface to procedural I/O subsystem
3. **fs.gs**: File System abstract data type.
4. **support.gs**: Various helpful datatypes and support functions.

```

-----
Jul 31 16:13 1994 ui.gs Page 1

-- File: ui.g
-- Author: David Carter
-- Date: 23/03/94
--
-- User interface support code - sits as a layer between filesystem module
-- and gruesome Dialogue plumbing.
--
-- You start everything working by telling gofer "go 1". This doesn't am
-- awful lot of lot level messing around, and finally emerges running the
-- Manager Dialogue "shellWrapper"

go _ = runDCSystem shellWrapper

-- 'ManagerDialogue' and 'ServerDialogue' are the types of the external
-- interfaces. 'ManagerAgent' and 'ServerAgent' are the types of functions
-- that control these interfaces. Note that shared state between the two
-- sets of Dialogues imply that they are not necessarily the same.

type ManagerDialogue = [[Msg]] -> [(String, [Msg])]
type ServerDialogue = [Msg] -> [Msg]

type ManagerAgent = [[Msg]] -> [(String, [Msg])]
type ServerAgent = File -> [Msg] -> (File, [Msg])

serverName = "xterm -e server"

-- shellWrapper splits of a ServerDilaogue to use as input to the shell and
-- then calls 'shell'; with a sensible set of inputs. Isn't functional code
-- wonderful stuff?

shellWrapper :: ManagerAgent

shellWrapper ~(commands:serverIns)

= (serverName, map (toMsg) shellOuts) : serverOuts

where

(shellOuts, serverOuts) = shell (map (fromMsg) commands) serverIns
-----
-- 'shell':
--
-- Consider 'shell' to be the top level of the system. It takes in a command
-- stream and a list of input streams from async processes. Returns shell
-- output and a list of output streams to async processes.
--
-- Works by simply calling shellLoop with an initial state for the file store.

shell :: [String] -> [[Msg]] -> ( [String], [(String,[Msg])] )

shell commands serverIns

= (initialGreeting : shellOuts, serverOuts)

where

(shellOuts, serverOuts) = shellLoop initialFS commands serverIns

initialGreeting

= "Welcome to Deterministic Concurrent Gofer [PROTOTYPE]\n" ++

"Type \"help\" for brief help\n\n"

initialFS = emptyFS

-- 'shellLoop':
--
-- The biggie. Or rather the first bit of the biggie that is split
-- across 'shellLoop', 'shellCheck', and 'shellValid'. The real problem
-- is keeping both sets of input and output lists in step with error
-- conditions and the particular type of command require
--
-- 'shellCheck' discards all commands that cannot be identified as a valid
-- operation. 'shellValid' splits operations in two groups and runs them
-- using 'runFileOp' and 'runAgent'. Both these functions may update the
-- filestore, but 'runAgent' also splits out a new Dialogue to run.
--
-- All of these functions are of type "shellChain"

type ShellChain = FileSystem -> [String] -> [[Msg]]

-> ([String], [(String, [Msg])])

shellLoop :: ShellChain
shellCheck :: ShellChain

```

```

shellValid :: ShellChain

runFileOp  :: ShellChain

runAgent   :: ShellChain

shellLoop fs commands serverIns

= ("DCGofer> " : shellStream, serverOuts)

where (shellStream, serverOuts) = shellCheck fs commands serverIns

shellCheck fs []          serverIns = ([], []) -- End of input
shellCheck fs (command:rest) serverIns
= if (null (init command)) then
    shellLoop fs rest serverIns
else if (not . validCommand) command then
    let
        (outs', acts') = shellLoop fs rest serverIns
        error           = (init command) ++ "": invalid command\n"
    in
        (error : outs', acts')
else
    shellValid fs (command:rest) serverIns

shellValid fs commands@(x:rxs) serverIns
= if (isFileOp x) then
    runFileOp fs commands serverIns
else if (isAgent x) then
    runAgent fs commands serverIns
else
    error "Inconsistent Program"

runFileOp fs (x:rxs) serverIns
= ((unchop shellOut) : shellOuts, serverOuts)
where
    (shellOut , fs') = doFileOp fs x
    (shellOuts, serverOuts) = shellLoop fs' rxs serverIns

runAgent fs (x:rxs) serverIns
= case (parseAgent fs x) of
    Error errorMsg
    -> let
        (shellOuts,serverOuts) = shellLoop fs rxs serverIns
    in
        unchop ** = **
        unchop string = string ++ "\n"
    -----
    --
    -- Commands are either FileOps or RunOps
    --
    validCommand :: String -> Bool

    validCommand command
    = case (words command) of
        [] -> False
        (x:rxs) -> elem x commands
    where
        commands = agentCommands ++ fileOpCommands
    -----
    --
    -- Agent Command support.
    --
    agentAssoc :: [ (String, ServerAgent) ]

```

```

agentAssoc                                     "Replace: Please enter some text.\nType <Ctrl-D> to finish\n\n"

= [ ("edit",      edit),
    ("backwards", backwards),
    ("trace",     trace)
  ]

agentCommands = map (fst) agentAssoc

isAgent :: String -> Bool

isAgent command

= case (words command) of
  [] -> False
  (x:rxs) -> elem x agentCommands

parseAgent :: FileSystem -> String -> MaybeError (Name, File, ServerAgent)

parseAgent fs str

= case (words str) of
  [] -> error "Inconsistent program"
  [server,name]
    -> case fileLookup fs name of
      Error str -> Error str
      Ok file
        -> case (lookup server agentAssoc) of
          Yes agent -> Ok (name, file, agent)
          No -> Error "Can't find agent description"
  (x:rxs) -> Error ("Usage: " ++ x ++ " <file> to start async. process")

-----
--
-- Set of three possible agents. All are of type "ServerAgent".
--
--
-----

edit :: ServerAgent

edit file msgs

= ((toFile (unlines strings)),
   [toMsg "Waiting for input\n", toMsg greeting])

where

greeting

= "Input File:\n" ++ (fromFile file) ++
  "\n"

strings = map (init.fromMsg) msgs :: [String]

backwards :: ServerAgent

backwards file msgs

= ((toFile . unlines . reverse) strings),
  [toMsg "Waiting for input\n", toMsg greeting])

where

greeting

= "Input File:\n" ++ (fromFile file) ++
  "\n"

strings = map (init.fromMsg) msgs :: [String]

trace :: ServerAgent

trace file msgs

= (emptyFile,
   [toMsg "Waiting for input\n"] : map (toMsg . addNL) input)

where

addNL :: String -> String

addNL string = string ++ "\n"

input = lines (fromFile file)

-----
--
-- fileOps are functions that manipulate filesystems. These functions are
-- in general simple parsers that call the relevant function in the support
-- library 'fs.gs'
--
type FileOp = FileSystem -> [String] -> Return FileSystem

fileOpCommands = map (fst) fileOpAssoc

fileOpAssoc :: [ (String, FileOp) ]

fileOpAssoc

= [
  ("help" , myhelp),
  ("ls" , ls),
  ("new" , new),
  ("rm" , rm),

```

```

("cp"      , cp),

("mv"      , mv),

("restore", restore),

Jul 31 16:13 1994 ui.gs Page 3

("undo"    , restore),

("commit"  , commit)

]

isFileOp :: String -> Bool

isFileOp command

= case (words command) of

[]      -> False

(x:rxs) -> elem x fileOpCommands

doFileOp :: FileSystem -> String -> Return FileSystem

doFileOp fs str

= case (words str) of

(x:rxs)

-> case (lookup x fileOpAssoc) of

Yes fileOp -> fileOp fs rxs

No         -> error "Inconsistent program"

-- File System Operations that we what to support are as follows:

--

-- help                List available commands

-- ls                  Lists current set of files and version counts.

-- new <fname>         Create a new file

-- rm <fname>          Remove file name

-- mv <fname1> <fname2> Move file from fname1 to fname2

-- cp <fname1> <fname2> Copy file from fname1 to fname2

-- restore/undo <fname> <count> Go backwards

-- commit              Discard all old versions

myhelp :: FileSystem -> [String] -> Return FileSystem

myhelp fs _ = (unlines helpMsgs, fs)

helpMsgs

= [

"help                List available commands",

"ls                  Lists current set of files and versions.",

"new <fname>         Create a new file",

"rm <fname>          Remove file name",

"mv <fname1> <fname2> Move file from fname1 to fname2",

"cp <fname1> <fname2> Copy file from fname1 to fname2",

"restore/undo <fname> <count> Go backwards",

"commit              Discard all old versions",

""

]

--

-- File Operations follow. All are of type FileOp.

--

ls :: FileSystem -> [String] -> Return FileSystem

ls fs []

= (showFS fs, fs)

where

showFS fs

= "List of existing files and version counts:\n\n" ++

unlines (map (show1) fs)

where

show1 (name,flist) = name ++ ": " ++ show (length flist)

ls fs _ = ("ls does not take any arguments", fs)

new :: FileSystem -> [String] -> Return FileSystem

new fs [x] = fileCreate fs x

new fs _   = ("Usage: new <file> creates a new file", fs)

rm :: FileSystem -> [String] -> Return FileSystem

rm fs [x] = fileDelete fs x

rm fs _   = ("Usage: rm <file> deletes file from filesystem", fs)

mv :: FileSystem -> [String] -> Return FileSystem

```

```

mv fs [x,y] = fileRename fs x y

mv fs _ = ("Usage: mv <src> <dest> renames a file", fs)

cp :: FileSystem -> [String] -> Return FileSystem
cp fs [x,y] = fileCopy fs x y
cp fs _ = ("Usage: cp <src> <dest> renames a file", fs)

undo :: FileSystem -> [String] -> Return FileSystem
restore = undo

undo fs [x,y]
  | all (isDigit) y
    = fileRewind fs x (atoi y)
  | otherwise
    = ("Usage: undo <file> <count> removes count copies of file", fs)
undo fs _ = ("Usage: undo <file> <count> removes count copies of file", fs)

commit :: FileSystem -> [String] -> Return FileSystem
commit fs [] = fileCommit fs
commit fs _ = ("Usage: commit [no parameters] removes all old files", fs)

-- File: io.g
-- Author: David Carter
-- Date: 31/03/94
--
-- Low level jiggery hackery. Do *NOT* change this stuff it is the
-- interface onto the low level C support code, which isn't guaranteed
-- to work in any other way that that given here!
--
{- ===== -}
--
-- Startup code.
--
-- [primStartDCSystem has type a -> String because typechecking code
-- barfs when you try to pass in a function. It seems to work, so don't
-- knock it. Presume either bug in Gofer or dodgy support code]
--
--
-- Type system barfs if next line enabled. Presume some problem with
-- inferring types from primitive...
--
-- runDCSystem :: (ManagerAgent) -> String

runDCSystem manager = primRunDCSystem manager

```

```

primitive primRunDCSystem "primRunDCSystem" :: a -> String

-- Input Channel stuff.
--
-- Don't mess with the "magic" values here - they are C ptrs!!!
--
-- This is just a quick hack 'till I can think of the proper way to do all
-- this. Probably involves constructing exactly the same graph in C, just
-- so you can't get your grubby paws on it.
--

serverInputStream          :: Int          -> [Msg]
primitive primGetMsg "primGetMsg" :: Int -> Int -> Msg

serverInputStream magic = msgList (map (primGetMsg magic) [1..])

-- msgList converts an infinite Msg stream into a simple list of messages
-- A convenience function that makes life easier for the grubby low level
-- plumbing

msgList :: [Msg] -> [Msg]
msgList stream = (takeWhile ((not) . (isEOFMsg)) stream)

--
-- Support for primitive Msg type. We don't support any Msgable types
-- but [Char], but who knows what the future might hold.
--

isEOFMsg :: Msg -> Bool
isEOFMsg = primIsEOFMsg

primitive primIsEOFMsg "primIsEOFMsg" :: Msg -> Bool

class Msgable a where
    toMsg      :: a -> Msg
    fromMsg    :: Msg -> a

instance Msgable [Char] where
    toMsg = primCharsToMsg
    fromMsg = primCharsFromMsg

primitive primCharsToMsg "primCharsToMsg" :: [Char] -> Msg
primitive primCharsFromMsg "primCharsFromMsg" :: Msg -> [Char]

```

```

emptyFile = F ""

toFile  str  = F str
fromFile (F str) = str

{- ===== -}

-- FileSystem DataType and operations:
--
-- Various types of filesystem are possible. The important thing is that we
-- support version control for the rewind operation.
--
-- A hierarchical file system is desirable, but doesn't add much to the
-- value of this demonstration module. We use a simple flat directory to
-- make our lives easier. This can be represented by an association list
-- from file names to lists. Each list contains all the versions of a
-- particular file, with the most recent at the front where they can be
-- accessed most easily.

Mar 30 16:49 1994  fs.gs Page 1

-- File:  fs.g
-- Author: David Carter
-- Date:  21/03/94
--
-- File System support code.

type FileSystem = [(Name,[File])]
type Name      = String

emptyFS = [] :: FileSystem -- The initial FileSystem

{- ===== -}

-- "File" type is simply string packaged as ADT:
--
-- Access via "toFile" and "fromFile" -- these could be made overloaded.
--
-- PRIVATE

data File      = F String

-- PUBLIC

toFile :: String -> File -- Allow for possibly overloaded "File" type
fromFile :: File -> String

-- File System operations that can be used by a client [i.e. PUBLIC]
--
-- fileList:  Map filesystem to list of (name, no. version) pairs.
-- fileLookup: Return handle to most recent version of file
-- fileExists: Determine if file with given name exists.
-- fileCreate: Add empty file to filesystem. Error to create twice
-- fileUpdate: Locate file in list, add new version. None => create
-- fileDelete: Delete file from filesystem
-- fileRename: Rename file from oldname to newname. Existing overwritten
-- fileCopy:   Make copy of file [including version history]
-- fileRewind: Discard given number of new versions of a file.
-- fileCommit: Discards all old versions of files

```

```

fileList :: FileSystem          -> [(Name,Int)]
fileLookup :: FileSystem -> Name      -> MaybeError File
fileExists :: FileSystem -> Name      -> Bool
fileCreate :: FileSystem -> Name      -> Return FileSystem
fileUpdate :: FileSystem -> Name -> File -> Return FileSystem
fileDelete :: FileSystem -> Name      -> Return FileSystem
fileRename :: FileSystem -> Name -> Name -> Return FileSystem
fileCopy    :: FileSystem -> Name -> Name -> Return FileSystem
fileCommit  :: FileSystem          -> Return FileSystem
fileRewind  :: FileSystem -> Name -> Int  -> Return FileSystem

fileList fs = map (\(name,list) -> (name, length list)) fs

fileLookup fs filename
= case files of
    No      -> Error ("Could not find file: " ++ filename)
    Yes []   -> Ok emptyFile -- Need some value out of nothing.
    Yes (x:rxs) -> Ok x
where
    (files,_) = fileExtract fs filename

fileExists fs name = any (\(n, fs) -> (name == n)) fs

fileCreate fs name
| (fileExists fs name) = ("File " ++ name ++ " already exists",fs)
| otherwise            = ("",((name, []) : fs))

fileUpdate fs name f
| (fileExists fs name) -- Update existing file
= case files of
    Yes flist -> ("", ((name,f:flist) : fs'))
    No        -> error "Inconsistent program"
| otherwise
= ("", ((name,[f]) : fs)) -- Add new file
where
    (files, fs') = fileExtract fs name

fileDelete fs name
| (fileExists fs name) = ("", (filter (\(n,fs) -> n /= name) fs))
| otherwise            = ("Attempt to delete non-existent file", fs)

fileRename fs oldname newname
| (oldname == newname)
= ("Attempt to rename to self", fs)
| (not (fileExists fs oldname))
= ("Attempt to rename non-existent file", fs)
| ((fileExists fs newname))
= case fs2' of
    (_, fs2) -> fileRename2 fs2 oldname newname
| otherwise
= fileRename2 fs oldname newname
where
    fs2' = fileDelete fs newname

fileRename2 fs oldname newname
= case files of
    Yes flist -> ("", ((newname, flist) : fs'))

fileCopy fs oldname newname
| (oldname == newname)
= ("Attempt to copy over self", fs)
| (not (fileExists fs oldname))
= ("Attempt to copy non-existent file", fs)
| ((fileExists fs newname))
= case fs2' of
    (_,fs2) -> fileCopy2 fs2 oldname newname
| otherwise
= fileCopy2 fs oldname newname
where
    fs2' = fileDelete fs newname

fileCopy2 fs oldname newname

```

Mar 30 16:49 1994 fs.gs Page 2

```

= case files of
  Yes flist -> ("", ((newname,flist) : (oldname,flist) : fs'))
  No        -> error "Inconsistent program"
where
(files, fs') = fileExtract fs oldname

fileRewind fs name cnt
= case files of
  No        -> ("Could not find file: " ++ name, fs)
  Yes flist -> let len = (length flist) in
    if (len < cnt) then
      (p len, fs)
    else
      ("", ((name, drop cnt flist) : fs'))
where
(files, fs') = fileExtract fs name

p 0 = "No previous versions available."
p 1 = "Only one previous version available."
p n = "Only " ++ (show n) ++ " previous versions available."

fileCommit fs
= ("", map (makeSingle) fs)
where
makeSingle (n,[]) = (n,[])
makeSingle (n,xs) = (n,[head xs])

{- ===== -}

-- fileExtract:
--
-- Many of the file system operations need to separate out a file from
-- the filesystem, perform some operation on that file, and then
-- insert it back into the filesystem. fileExtract does this. It
-- should not be exported/used outside the FileSystem ADT as it
-- does not guarantee fs integrity

fileExtract :: FileSystem -> Name -> (Maybe [File], FileSystem)

fileExtract fs name
= case fs2' of
  []          -> (No      , fs)
  ((name,fls):fs2) -> (Yes fls, fs1++fs2)

```

```

-- atoi
--
-- Simple implementation, can only handle positive numbers

atoi "" = 0
atoi x =
    atoi2 x 0
    where
        atoi2 [] count = count
        atoi2 (d:rds) count
            | (d >= '0' && d <= '9')
                = atoi2 rds (10*count + (ord d) - (ord '0'))
            | otherwise
                = error "atoi: not a number"

-- lookup:
--
-- Look into an association list and return the value that matches.

lookup :: String -> [(String, a)] -> Maybe a

lookup str list
    = case (filter (\ (name,fn) -> (name == str)) list) of
        [] -> No
        [(name, fn)] -> Yes fn
        _ -> error "lookup: list had multiple entries"

```

Jun 16 17:18 1994 support.gs Page 1

```

-- File: support.g
-- Author: David Carter
-- Date: 29/03/94
--
-- Various support routines and data types:
--

data MaybeError a = Ok a | Error String -- Various error handling support

data Maybe a = Yes a | No

type Return a = (String, a)

force :: Return a -> a -- Mostly to help debugging

force (_,x) = x

```



# Appendix C

## Functions provided by the Concurrency library

This appendix contains the text of the C header files “`dc_io.h`” and “`dc_lwp.h`”. These files define the services provided by the concurrency library.

```

Feb 24 17:47 1994 dc_io.h Page 1

/* File: dc_io.c
* Author: David Carter
* Date: 23/11/93 --> split out from old dialogue.c
*
* Shared I/O system prototypes. Note that LCD needs to know about
* "ServerDialogue" as primitive I/O data structure.
*
*/

#ifdef __DC_IO_H__
#define __DC_IO_H__

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/time.h>
#include <signal.h>
#include <errno.h>

/* ===== */

/* Common useful bits and bobs */

#ifdef __STDC__
#define PARAMS(paramlist) paramlist
#else
#define PARAMS(paramlist) ()
#endif

#ifdef __GNUC__
#define INLINE __inline__
#else
#define INLINE
#endif

#endif

#ifndef NULL
#define NULL ((void *) 0)
#endif

typedef int BOOLEAN;

#define FALSE 0
#define TRUE 1

#define MAX(x,y) (((x)>(y)) ? x : y)
#define MIN(x,y) (((x)<(y)) ? x : y)

/* ===== */

/* Dialogue is main interface structure between evaluation threads and the
* IO System. Each thread has an associated Dialogue struct through which
* all input and output is processed.
*
* Dialogues are arranged in a simple hierachical structure with Managers
* as the branches and Servers as the leaves
*/

#define IO_BUF_SIZE BUFSIZ /* Size for low level I/O */

typedef enum {NOTHREAD,RUNNING,SUSPENDED,BLOCKED} ThreadState;

#define MSG_EOF 1 /* Set in flags field => EOF Marker */

struct Msg {
    struct Msg *next; /* So that we can link in a chain */
    int number;
    int flags;
    int len;

    /* Followed by message of length len */

#ifdef __GNUC__
    char body[0]; /* A placeholder for simpler debugging */
#endif
};

#define MSG_STRUCT_SIZE (sizeof (struct Msg))

#define MsgContents(msg) ((char *) (((char *) (msg)) + MSG_STRUCT_SIZE))

```

```

#define MsgIsEOF(msg)\
    ((msg==NULL || (msg->flags & MSG_EOF) != 0) ? TRUE : FALSE)

struct Dialogue {
    ThreadState  threadState; /* Indicates why thread is suspended */
    void         *thread;     /* Thread with LWP, else null */
    struct Dialogue *nextLocked; /* (State == SUSPENDED)
                                   => lists of locked cells */
    struct Dialogue *blockedOn; /* (State == BLOCKED)
                                   => Dialogue that we are waiting for */
    void         *localState; /* Structure containing local state info */
    /*
     * Next two belong in ServerDialogue - not used by manager. Move over
     * when you have a week to spare for changing type sigs etc. Oh for an
     * OOP
     */
    struct Msg    *outputList; /* Output Queue */
    struct Msg    *inputList;  /* Input Queue */
};

/* PartialMsg is closest interface point to asynchronous input and output
 * channels. Contains more information than either half strictly needs, but
 * this way gives us a nice consistent interface. Down to low level routines
 * to give the relevant effect.
 *
 * This is MkII version
 */

struct MsgHdr {
    /* Need this much before we can malloc & fill */
    u_long  number; /* Encoded sequence number */
    u_long  flags;  /* Encoded flags */
    u_long  len;    /* Encoded Length [body only] */
};

#define HDRSIZE (sizeof (struct MsgHdr))

typedef enum {NONE,HDR,BODY,COMPLETE} PartialState;

struct PartialMsg {
    Feb 24 17:47 1994  dc_io.h Page 2
    PartialState  state;
    int           hdrXfer; /* Amount of hdr read/written */
    struct MsgHdr hdr;     /* Size = HDRSIZE */
    int           msgXfer; /* Amount transferred */
    int           msgSize; /* Complete size of message */
    struct Msg    *msg;
};

/* ServerDialogue is a Dialogue that supports asynchronous communication with
 * a subprocess spawned on its invocation
 */

struct ServerDialogue {
    struct Dialogue  common; /* Common hdr block */
    struct ServerDialogue *next; /* Servers are chained */
    char             *childName; /* Name of process to be run */
    int              childPID;   /* So we can kill it easily */
    int              inputChannel; /* Input FD [set async!] */
    int              outputChannel; /* Output FD [set to async!] */
    BOOLEAN          isInput;     /* read() will not block */
    BOOLEAN          canOutput;   /* write() will make progress */
    struct PartialMsg partialInputMsg; /* Used by IO thread */
    struct PartialMsg partialOutputMsg;
};

/* ===== */
/* ===== GLOBAL FUNCTION PROTOTYPES ===== */
/* ===== */

/* Miscellenous support fns */

struct Msg *MakeStrMsg PARAMS((char *));
struct Msg *AllocMsg  PARAMS((int));

```

```

struct Msg *CopyMsg    PARAMS((struct Msg *));

struct Msg *MakeEOFMsg PARAMS((void));

void      FreeMsg    PARAMS((struct Msg *));

/* Common Dialogue operations */

void InitDialogue
    PARAMS((struct Dialogue *,void *));

/*
 * Set up common fields.
 *
 */

void QueueOutputMsg
    PARAMS((struct Dialogue *, struct Msg *));

/*
 * Adds a message to the output queue, while be proceeded at the relevant
 * point in the I/O cycle.
 */

struct Msg *DeQueueInputMsg
    PARAMS((struct Dialogue *));

/*
 * Takes a message from the input queue. NULL => empty queue, and is
 * probably an illegal state if the I/O system is behaving itself
 */

/* ===== */
/* ===== FUNCTION PROTOTYPES ===== */
/* ===== */

/*
 * Internal bits and bobs that shouldn't be visible to the outside world
 * This is where a proper module system wouldn't half come in handy
 */

extern int errno;          /* Error value for system calls */

void myFatal PARAMS((char *,char *));
void syserr PARAMS((char *,char *));

/* Dialogue support fns */

```

```

BOOLEAN      InputPending    PARAMS((struct ServerDialogue *));
BOOLEAN      OutputPending   PARAMS((struct ServerDialogue *));

/* IO Support fns */

void SetNonBlocking    PARAMS((int));
void ProcessSelectServer PARAMS((struct ServerDialogue *,fd_set *,fd_set *));
int  SetupSelectServer PARAMS((struct ServerDialogue *,fd_set *,fd_set *));

void DoIOServer    PARAMS((struct ServerDialogue *));

#endif

```

```

/*
 * Reverse the effects of the above, so that system can be shutdown
 * and restarted without a spaceleak.
 */

void RunDialogueThread PARAMS((void (*)(), struct Dialogue *));

/*
 * Add fresh evaluation thread to the system.
 */

Dec 14 16:27 1993 dc_lwp.h Page 1

/* File: dc_lwp.h
 * Author: David Carter
 * Date: 23/11/93 --> Separated out from dialogue.h
 *
 * Interface to parrallel run time system and asynchronous I/O.
 *
 */

#ifndef __DC_LWP_H__
#define __DC_LWP_H__

#include "dc_io.h"

#define RunningThreads() (runningThreads)

extern BOOLEAN runningThreads;

/*
 * Function Prototypes for public access routines
 */

/* Thread support functions for Dialogues */

void InitThreadSystem PARAMS((void));

/*
 * Get the thread system up and running. Main thread remains outside the
 * dc_libraries control, although it is responsible for calling IOThread.
 */

void ShutDownThreadSystem PARAMS((void));

/*
 * Add service thread to the system.
 * (block = TRUE) => will run at a higher priority than other
 * threads in the system.
 */

void KillDialogueThread PARAMS((void));

/*
 * Destroy the current thread, does not return.
 * First reduces active and current dialogue count.
 */

void FreeDialogue PARAMS((struct Dialogue *));

/*
 * Deallocate dialogue structures
 */

void KillDialogue PARAMS((struct Dialogue *));

/*
 * Deallocate structures, unregister and destroy thread. Equivalent to
 * the above two functions combines
 */

void BlockDialogue
PARAMS((struct Dialogue *, struct Dialogue *));

/*
 * Suspend a Dialogue until input items are available on the given
 * Dialogues input stream [may be self]
 */

```

```

void UnblockDialogue
    PARAMS((struct Dialogue *));
/*
 * Called by I/O system when we have a message that can be processed
 */

void SwitchThread
    PARAMS((void));
/*
 * Switch out the current thread, while keeping it active.
 */

void DelayThread
    PARAMS((int));
/*
 * Suspend thread for at least this long - others may proceed.
 */

int DialogueCount PARAMS((void));
/*
 * Returns the number of evaluator threads in the system
 */

int ActiveDialogues PARAMS((void));
/*
 * Returns the number of evaluator threads that are not blocked
 * (on input or because of shared evaluation) in the system
 */

void SetDialoguePageFns PARAMS((void (*)(void)), void (*)(void));
/*
 * Set up the page in and page out functions for a thread. These functions
 * will be called with a ptr to the current environment [(Dialogue *) for
 * evaluation threads] whenever a thread context switch occurs.
 */

void *ThreadId PARAMS((void));
/*
 * Returns a handle (void *) to the current lowlevel thread structure.
 * Currently used by the WakeDialogue() function - initialisation of
 * Dialogue->threadp may have to be tightened up if need is generalised.
 */

struct Dialogue *CurDialogue PARAMS((void));
/*
 * Returns Dialogue structure for current Dialogue/thread. Note that this
 * may be different to Dialogue currently being processed.
 */

Dec 14 16:27 1993 dc_lwp.h Page 2

void AllocLockNodes PARAMS((int));
/*
 * Sets up the shared node locking system.
 */

void FreeLockNodes PARAMS((void));
/*
 * Shuts down the shared node locking system.
 */

BOOLEAN IsLocked PARAMS((int));
/*
 * Indicate lock status of cell.
 */

BOOLEAN LockNode PARAMS((int));
/*
 * Lock a node. Repeated lock attempts will suspend. Returns TRUE if the
 * thread was suspended.
 */

void UnLockNode PARAMS((int));
/*
 * UnLock a node - will wake up all suspended threads.
 */

```

```

*/
void SuspendDialogue
    PARAMS((struct Dialogue *));
/*
 * Called by thread to suspend on a shared node
 */

void WakeDialogue
    PARAMS((struct Dialogue *));
/*
 * Wake up Dialogue locked on a shared node
 */
/* Manager Dialogue is just a Dialogue with a list of slave servers.
 * Complexity is all in the support code
 */

struct ManagerDialogue {
    struct Dialogue    common;
    struct ServerDialogue *clientList;
};

/* Housekeeping Operations specifically for Manager Dialogues */

void AddManagerDialogue PARAMS((struct ManagerDialogue *, void (*)());

/*
 * Add a top level dialogue into IOSystems internal management system.
 * ManagerDialogues are just evaluations with well defined f/x.
 *
 */

struct ManagerDialogue *FindManagerDialogue
    PARAMS((struct ServerDialogue *));
/*
 * Add a top level dialogue into IOSystems internal management system.
 * ManagerDialogues are just evaluations with well defined f/x.
 *
 */

void ForEachDialogue PARAMS((void (*)(struct Dialogue *));

/*
 * Call a function for every Dialogue in the system
 */

void ShutDownManagerDialogue
    PARAMS((struct ManagerDialogue *));
/*
 * Remove Dialogue from IOSystem internal management system
 */

void IOCycle PARAMS((BOOLEAN));
/*
 * Do one complete I/O cycle, moving all data to and from temporary
 * streams as possible. Because its async I/O, we can do one root
 * Dialogue at a time [if more than one], or all in a row.
 *
 * *DANGER*:
 *
 * Deadlock can result if you call this routine in blocking mode while
 * there is unprocessed input for the client. Suggest that we add a
 * 'CanBlock()' fn that checks Dialogue thread state and input status to
 * determine if any useful work can be done outside the I/O scheduler.
 *
 * Come back and fill this in once you no what is happening with threads.
 */

/* ManagerDialogue specific operations */

void AddServerDialogue PARAMS((struct ManagerDialogue *,
    struct ServerDialogue *,
    char *,
    void (*)());
/*
 * Mallocs a ServerDialogue structure filling in childName and
 * outputStream fields. Starts up asynchronous process "name",
 * setting up input and output streams.
 *
 * Starts a thread whose local datastructure is the ServerDialogue.
 */

void ShutDownServerDialogue
    PARAMS((struct ManagerDialogue *, struct ServerDialogue *));
/*

```

```
* Locate Server, kill subprocess and connections. Free all memory
* allocated by server. Thread is killed seperately by a call to
* KillDialogueThread()
*/

#endif
```

# Bibliography

- [1] L. Augustsson and T. Johnsson. The chalmers lazy-ml compiler. *Computer Journal*, 32(2):127–141, 1989. Special Issue on Lazy Functional Programming.
- [2] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978. Turing Award Lecture.
- [3] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (Revised Edn.)*. Elsevier Science Publishers B.V. (ISBN 0–444–87508–4, 1984. Studies in Logic and the Foundations of Mathematics Vol 103.
- [4] R.S. Bird and P.L. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.
- [5] F. Warren Burton. Nondeterminism with referential transparency in functional programming languages. *Computer Journal*, 31(3):243–247, 1984. Also Dept. Elec. Eng. and Comp. Sci., Univ. Colorado at Denver, 1984.
- [6] F. Warren Burton. Encapsulating non-determinacy in an abstract data type with determinate semantics. *Journal of Functional Programming*, 1(1):3–30, 1991.
- [7] M. Carlsson and T. Hallgren. Fudgets - a graphical user interface in a lazy functional language. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*. Addison Wesley, Copenhagen, DK, 1993.
- [8] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [9] W. Clinger. Nondeterministic call by need is neither lazy nor by name. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 226–234. ACM, August 1982.

- [10] J. Cupitt. *The Design and Implementation of an Operating System in a Functional Language*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, 1990.
- [11] A. Dwelly. Functions and dynamic user interfaces. In *Functional Programming and Computer Architecture, Imperial College*, pages p71–381. Addison Wesley, 1989.
- [12] Jon Fairbairn and S. Wray. Tim: a simple, lazy abstract machine to execute supercombinators. *Lecture Notes in Computer Science*, 274:34–46, 1987. FPCA’87 Portland. Also University of Glasgow Dept. Comp. Sci. report CSC/87/R6, 1987.
- [13] A.J. Field and P.G. Harrison. *Functional Programming*. Addison Wesley, ISBN 0-201-19249-7, 1988.
- [14] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in haskell. *ESOP*, 1994. Available by Anonymous FTP from ftp.dcs.glasgow.ac.uk as /pub/glasgow-fp/papers/type-classes-in-haskell.ps.Z.
- [15] D. Harrison. Ruth: a functional language for real-time programming. *Lecture Notes in Computer Science*, 259, 1987. PARLE’87.
- [16] Dan Heller. *XView Programming Manual*. O’Reilly & Associates, Inc, 1990.
- [17] P. Henderson. *Functional Programming: applications and implementation*. Prentice-Hall, 1980.
- [18] Peter Henderson. Purely functional operating systems. In John Darlington, Peter Henderson, and David Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982. Original Landmark paper in subject.
- [19] Peter Henderson. Communicating functional programs. FP FPN–8, University of Stirling, Dept. Comp. Sci., 1984.
- [20] Peter Henderson. Process combinators. FP FPN–7, University of Stirling, Dept. Comp. Sci, 1984.
- [21] Peter Henderson and Simon B. Jones. Shells of functional operating systems. FP FPN–4, University of Stirling, Dept. Comp. Sci., 1984.

- [22] P. Hill and J. W. Lloyd. The gödel programming language. Technical Report CSTR-92-27, University of Bristol, October 1992.
- [23] Ian Holyer. *Functional Programming with Miranda*. Pitman Publishing, ISBN 0-273-03453-7, 128 Long Acre, London, WC2E 9AN, 1991.
- [24] P. Hudak and J.H. Fasel. A gentle introduction to haskell. *SIGPLAN*, 27(5):T1-T53, 1992. Also by Anonymous FTP from Glasgow (dcs.glasgow.ac.uk).
- [25] P. Hudak and R. Sundaresh. On the expressiveness of purely functional i/o systems. Technical Report YALEU/DCS RR-665, Dept. CS, Yale Univ., 1988.
- [26] P. Hudak, P. Wadler, et al. Report on the programming language haskell, a non-strict purely functional language, version 1.2. *SIGPLAN*, 27(5), 1992. Also FTP from Glasgow and Yale Universities.
- [27] Paul R. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3):359-411, 1989.
- [28] R.J.M. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98-107, April 1989. Also in *Research Topics in Functional Programming*, D.A. Turner Ed.
- [29] Mark P. Jones. *Gofer*. University of Oxford, Programming Research Group, 1992. Distributed with Gofer source. Available from ftp.dcs.glasgow.ac.uk: /pub/haskell/gofer.
- [30] Mark P. Jones. The implementation of the gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science P.O Box 209285, New Haven, CT 06520-8285, 1994.
- [31] Simon B. Jones. Abstract machine support for purely functional operating systems. Technical Report 15, Stirling, Dept. Comp Sci., 1983. Also Oxford University Computing Laboratory, Programming Research Group, Monograph PRG-34, August 1983.
- [32] Simon B. Jones. A range of operating systems written in a purely functional style. Technical Report 16, University of Stirling, Dept. Comp. Sci., 1984. Also Oxford University Computing Laboratory, Programming Research Group, Monograph PRG-42.

- [33] Simon B. Jones and Andrew F. Sinclair. Functional programming and operating systems. *Computer Journal*, 32(2):162–174, 1989.
- [34] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. Springer-Verlag, ISBN 3-540-54396-1 / 0-387-54396-1, 1991. My copy by Anonymous FTP from Glasgow.
- [35] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages – A Tutorial*. Prentice Hall, ISBN 0-13-721952-0, 1992.
- [36] Simon Peyton Jones. The spineless tagless g-machine. In *Functional Programming and Computer Architecture, Imperial College*, pages 184–201. Addison Wesley, 1989.
- [37] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [38] S.L. Peyton Jones. The spineless tagless g-machine: a second attempt. Will be in JFP eventually. The monster, 1991. Version 2.5 available by Anonymous FTP from Glasgow ([dcs.glasgow.ac.uk](ftp://dcs.glasgow.ac.uk)).
- [39] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Symposium on Principles of Programming Languages*, volume 20, Charleston, South Carolina, January 1993. Also available by Anonymous FTP from [ftp.dcs.glasgow.ac.uk](ftp://ftp.dcs.glasgow.ac.uk).
- [40] K. Karlsson. Nebula: A functional operating system. Technical Report Programming Methodology Group Memo LPM11, Chalmers University of Technology, Göteborg, 1981.
- [41] D. King and P.L. Wadler. *Combining Monads*. Springer-Verlag, New York, NY, 1992. Springer Verlag Workshops in Computing.
- [42] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.
- [43] M. Main and D.B. Benson. Functional behaviour of nondeterministic and concurrent programs. *Information and Control*, 62(2/3):144–188, 1984.

- [44] J. McCarthy. A basic mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North Holland, Amsterdam, 1963.
- [45] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.
- [46] C.S. McDonald. Fsh - a functional unix command interpreter. *Software*, 17(10):685–700, 1987.
- [47] E. Moggi. Computational lambda calculus and monads. In *IEEE symposium on Logic in Computer Science*, California, 1989. IEEE. A longer version is available as a technical report from Edinburgh University.
- [48] Adrian Nye and Tim O' Reilly. *X Toolkit Intrinsics Programming Manual*. O'Reilly & Associates, Inc, 1990.
- [49] John T. O'Donnell. Dialogues: a basis for constructing programming environments. *SIGPLAN*, 20(7):17–27, 1985. Proc ACM SIGPLAN 85 Symp on Language Issues in Programming Environments.
- [50] J. Olszewski. Communicating processes in a pure functional language. *Australian Computer Science Communications*, 14(1):615–629, 1992.
- [51] M.S. Parsons. *Applicative Languages and Graphical Data Structures*. PhD thesis, Computing Laboratory, University of Kent at Canterbury, 1987.
- [52] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial Colleg, UK, 1991.
- [53] N. Perry. Towards a concurrent object/process oriented functional language. *Australian Computer Science Communications*, 14(1), 1992. (= Proceedings of ACSC'15).
- [54] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. Technical Report 172, University of York, 1992.
- [55] D.A. Schmidt. *Denotational Semantics: a Methodology For Program Development*. Allyn and Bacon, 1986.

- [56] J. Shultis. A functional shell. In *ACM SIGPLAN: Symposium on Programming Language Issues in Software Systems*, 1983.
- [57] H. Söndergaard and P. Sestoft. Non-determinism in functional languages. *Computer Journal*, 1992.
- [58] W.R. Stoye. A new scheme for writing functional operating systems. Technical Report 56, Cambridge University Computer Lab., 1984.
- [59] S. Thompson. Writing interactive programs in miranda. Technical Report 40, University of Kent at Canterbury Computing Laboratory, 1986.
- [60] Phil Trinder. A functional database. Technical report, Department of Computer Science, University of Glasgow, 1990. Also DPhil Thesis, University of Oxford PRG.
- [61] D. A. Turner. An approach to functional operating systems. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 199–218. Addison Wesley, 1990. Also “Functional Programming and Communicating Processes”, Lecture Notes in Computer Science 259 (PARLE 87) pp54-74.
- [62] D.A. Turner. A new implementation technique for applicative languages. *Software*, 9:31–49, 1979.
- [63] D.A. Turner. Recursion equations as a programming language. In J. Darlington et al., editors, *Functional Programming and its Applications*, pages 1–28. Cambridge University Press, 1982.
- [64] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. *Lecture Notes in Computer Science*, 201:1–16, 1985. Functional Programming and Computer Architecture (Nancy, France).
- [65] D.A. Turner. On overview of miranda. *SIGPLAN*, 21(12):158–166, 1986.
- [66] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages pp61–78, 1990. Later version in Math. Struct. in Comp. Sci. 2, pp461-493. Also available by FTP from ftp.dcs.glasgow.ac.uk.

- [67] P. Wadler. The essence of functional programming. In *Symposium on Principles of Programming Languages*, volume 19, Santa Fe, New Mexico, January 1992. Also available by Anonymous FTP from <ftp.dcs.glasgow.ac.uk>.