

Number 81



UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

The implementation of functional languages using custom hardware

William Robert Stoye

December 1985

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
<http://www.cl.cam.ac.uk/>

© 1985 William Robert Stoye

This technical report is based on a dissertation submitted May 1985 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Magdalene College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

<http://www.cl.cam.ac.uk/techreports/>

ISSN 1476-2986

Abstract

In recent years functional programmers have produced a great many good ideas but few results. While the use of functional languages has been enthusiastically advocated, few real application areas have been tackled and so the functional programmer's views and ideas are met with suspicion.

The prime cause of this state of affairs is the lack of widely available, solid implementations of functional languages. This in turn stems from two major causes:

Our understanding of implementation techniques was very poor only a few years ago, and so any implementation that is "mature" is also likely to be unuseably slow.

While functional languages are excellent for expressing algorithms, there is still considerable debate in the functional programming community over the way in which input and output operations should be represented to the programmer. Without clear guiding principles implementors have tended to produce ad hoc, inadequate solutions.

My research is concerned with strengthening the case for functional programming. To this end I constructed a specialised processor, called SKIM, which could evaluate functional programs quickly. This allowed experimentation with various implementation methods, and provided a high performance implementation with which to experiment with writing large functional programs.

This thesis describes the resulting work and includes the following new results:

Details of a practical Turner-style combinator reduction implementation featuring greatly improved storage use compared with previous methods.

An implementation of Kennaway's director string idea that further enhances performance and increases understanding of a variety of reduction strategies.

Comprehensive suggestions concerning the representation of input, output, and nondeterministic tasks using functional languages, and the writing of operating systems. Details of the implementation of these suggestions developed on SKIM.

A number of observations concerning functional programming in general, based on considerable practical experience.

Guide

Chapter One: Introduction **page 1.1**

This chapter explains the context in which this research was carried out, and the order in which the work described in the succeeding chapters was done.

Chapter Two: Combinator Reduction **page 2.1**

This chapter, and the next four, are concerned with implementation techniques for functional languages. This one gives details of the sort of functional language considered in this thesis, and describes Turner's combinator reduction scheme for implementing them.

Chapter Three: Microcode **page 3.1**

SKIM is a computer designed for the implementation of Turner's combinator reduction scheme. This chapter discusses many of the more unusual aspects of its low-level architecture. Building, programming and microprogramming SKIM acted as a focus for all of the research presented here.

Chapter Four: Optimisations **page 4.1**

An account of some advanced graph reduction techniques implemented on SKIM. Of particular interest are the reference counting scheme and a discussion of compilation and supercombinators.

Chapter Five: Director Strings **page 5.1**

A new approach to the representation of combinator graphs, resulting in considerable performance benefits, and some observations on the comparison of graph reduction schemes.

Chapter Six: Problems **page 6.1**

Current implementation techniques are still far from perfect, as this chapter shows. Several problem areas are demonstrated, mostly taking the form of innocent-looking programs with unexpected runtime behaviour. Of particular interest is the discussion of implementation techniques for arrays.

Chapter Seven: Functional Programming **page 7.1**

A collection of observations concerning functional programming, based on experience gained when working on SKIM. Of particular interest is a detailed exposition of how functional ideas can aid the construction of large programs, and of the approach to efficiency that should be used by the designers of functional languages.

Chapter Eight: Input and Output **page 8.1**

The interaction between functional programs and the outside world is not well understood. In this chapter input and output based on simple, sequential interaction is considered, and the idea of synchronised streams introduced to overcome some known problems in this area.

Chapter Nine: Operating Systems **page 9.1**

In this chapter the subject of input and output is extended to cover nondeterministic applications and interactions, while retaining the desirable linguistic properties of functional languages. The result is a new model of concurrent computation which has been implemented and used on SKIM. It is based on the passing of messages between a number of functional programs executing concurrently.

Chapter Ten: Conclusions **page 10.1**

A general review of some of the lessons learned in carrying out this research, and some suggestions for future work.

Appendix A: The Timeslicer Implementation **page A.1**

Details of the model used on SKIM to simulate parallel computation on a sequential combinator reduction machine. This model implements the message passing scheme described in chapter nine.

Appendix B: Miranda **page B.1**

Miranda is a functional language that is used for examples of simple programs. A brief summary of the language is given here.

Appendix C: A Functional Program **page C.1**

This appendix contains the lexical and syntax analysers of a small compiler that ran successfully on SKIM, and demonstrates the sort of programs that I worked on. A brief guide to the language Ponder is included, and various remarks about the development and performance of this compiler.

References **page R.1**

Introduction

Functional programming research at Cambridge began in 1979, when the student Microprocessor Group (with no official university backing) decided to build a microcoded processor to perform David Turner's recently published combinator reduction algorithm [Turner 79]. This suggested a way of implementing languages based on the normal order lambda calculus, by representing programs as graphs of S, K and I combinators. The result of their effort fitted on two circuit boards, and reduced combinators at about the same speed as a machine coded interpreter running on an IBM 370/165. This work was presented at the 1980 Lisp Conference, under the title "SKIM—the S, K, I Reduction Machine" [Clarke 80].

Unfortunately the members of the project then dispersed, before serious attempts could be made to use SKIM to tackle other functional programming issues. SKIM also suffered from signal noise problems, its mean time between inexplicable failures was about twenty minutes. The microcode memory was full, and further development hampered by a one-day turnaround in programming the microcode memory chips. Nevertheless, a demonstration had been made that very modest amounts of hardware (about five hundred pounds had been spent) provided a cost-effective way of implementing a functional language.

In the summer of 1982 Thomas Clarke, the designer of SKIM I's hardware, produced a design for SKIM II. Rather than attempting to go faster, his main priority was to construct a more stable basis for future development than its predecessor had provided. By removing some of the limitations that had plagued SKIM I issues concerning the use of functional languages in large scale programming projects, and the representation of input and output, could be investigated. Larger address spaces, more attention to reducing electrical noise, microcode in RAM and better support for hardware and microcode debugging were regarded as the most important changes. In addition, the experience gained in microcoding SKIM I had shown that most time was spent in one or two extremely small microcode "hot-spots": architectural improvements were made to ensure that the machine performed well in these areas.

In October 1982 I started as a Research Student at Cambridge University Computer Laboratory, and was given the task of building SKIM II. My reason for becoming involved in this was that reading [Backus 78] had convinced me there was something fundamentally wrong with the computer languages I was using. Although I did not understand the mathematics in Backus' Turing Award Lecture, its introduction sounded bold and exciting: the sort of work I wanted to get involved in!

"Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at their most basic level cause them to be both fat and weak: their word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer,

their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs" [Backus 78].

By the June of 1983 SKIM II (referred to as SKIM from now on) was complete. Debugging it was quite straightforward, owing largely to Thomas Clarke's skillful design work rather than my own ability, and to my intense relief the whole machine ran test programs for days without an error: no figure for mean time between failures was ever calculated as failures were not sufficiently frequent for this to be possible. I had also been writing microcode in parallel with the construction of the hardware, and shortly afterwards combinators were successfully reduced and garbage successfully collected.

In the second half of 1983 I started to write functional programs, and continued to make enhancements to the microcode. The details of these enhancements constitute chapter four, and later published in the 1984 ACM Lisp Conference [Stoye 84]. In addition to this the multitasking microcode described in appendix A was implemented, although a clear understanding of how to explain its operation (chapter nine) did not emerge until some time later.

In the first half of 1984 I continued to experiment with writing a variety of functional programs, and running them on SKIM. My main examples were the interactive shell of a simple operating system, a full screen text editor and a simple functional language compiler. In total about three thousand lines of source were produced in this effort, and although the three programs were never successfully merged into a single system the material in chapters six, seven and nine is based on lessons learned during this period.

In August 1984 I attended the ACM Lisp Conference in Austin, Texas. The most important benefit gained from this experience was the opportunity to visit the Burroughs Austin Research Center, where a machine very similar to SKIM was being built (see the end of chapter three). I had numerous fascinating discussions with the functional programming team at Burroughs ARC, and with David Turner who was also visiting them at the time, concerning the problems that I had encountered so far. Through these discussions my understanding of the operating system that I had written improved enormously. Two months later the chapter on operating systems (chapter nine) was completed, and appeared as a laboratory Technical Report [Stoye 84a]. A version of this has been accepted for publication by Science of Computer Programming.

In November of 1984 I was invited to participate in a Workshop on the Implementation of Functional Languages organised by the Chalmers University of Technology in Gothenburg, Sweden. This jogged me into doing some further work on SKIM's microcode, and at the workshop [Aspenäs 85] I presented an early version of the work on director strings (chapter five). The Aspenäs workshop demonstrated that a large number of people are now working on the problems involved in implementing functional languages.

This brief history explains the somewhat disjointed nature of this thesis. Attempting to use functional languages to write large programs has led to considerable advances in two other subjects: the implementation of functional languages, and the interaction of functional programs with the outside world. Although each of these three issues is really a distinct subject for research in its own right, they have to be tackled together as a lack of understanding of any one of them hampers progress in the other two. Chapters two to six are concerned with implementation issues, chapter seven attempts to tackle functional programming as the programmer sees it while chapters eight and nine are concerned with input and output.

There is little dependency between the various chapters, and (as a general rule) they can be read in any order. Chapter ten contains remarks on the progress of the project as a whole, and suggestions for future research. The contents page provides a summarising paragraph for each chapter. The most important new results in this thesis are the model for operating systems in chapter nine, and the uses for one bit reference counts described in chapters four and five. In a sense chapter seven (on functional programming in general) tackles the most important subject, but it is also the most difficult in which to make definite progress. I hope that the issues raised in chapters six and eight will have a strong influence on future implementations of functional languages.

Combinator Reduction

This chapter gives a general overview of the method that SKIM uses to implement functional languages. The method was first described in [Turner 79], and is based on a representation of the functional program as a combinator expression [Curry 68]. After brief descriptions of the lambda calculus, combinators and functional languages some explanation is given of the reasoning behind Turner's choice of combinators, and their behaviour at run time.

The Lambda Calculus

The lambda calculus provides a notation for functions, and some rules that allow the notation to be used as the basis for a model of computation. This section is only a brief summary of those aspects which are necessary for an understanding of functional languages: a fuller treatment can be found in [Stoy 77] or [Barendregt 81].

A lambda expression $\langle l \rangle$ conforms to the following syntax:

$$\begin{aligned} \langle l \rangle ::= & \langle \text{identifier} \rangle \\ & | (\langle l \rangle \langle l \rangle) \\ & | (\lambda \langle \text{identifier} \rangle. \langle l \rangle) \end{aligned}$$

The first two of these provide the familiar notation of function application. In this thesis the usual conventions concerning the omission of brackets are adopted, i.e. function application associates to the right and so "fn a b" denotes "((fn a) b)", and "(λ x. fn a)" denotes "(λ x. (fn a))" rather than "((λ x. fn) a)".

An occurrence of identifier x is said to be *bound* in expression E if it occurs inside some lambda clause of the form

$$(\lambda x. E')$$

within E . The notation

$$[v/x] X$$

is used to mean "the expression X with every unbound occurrence of the identifier x replaced by the value v ". The meaning intended for a lambda clause is best expressed by the equation

$$(\lambda x. E) v = [v/x] E.$$

This substitution may only occur if it does not cause some unbound identifier in v to become bound in the result. Two lambda expressions are equal if they denote the same function.

The notation is used as a method of computation by employing three steps known as *alpha* conversion, *beta* reduction and *eta* reduction.

$(\lambda x. E)$ can be converted to $(\lambda y. [y/x] E)$ by alpha conversion

$(\lambda x. E). v$ reduces to $[v / x] E$ by beta reduction

$(\lambda x. E x)$ reduces to E by eta reduction

For the third of these to be applicable there may not be any unbound occurrences of x in E . A *beta redex* lambda expression is one on which a beta reduction can be performed, the term *eta redex* is defined similarly. An expression is said to be in *normal form* if it contains no redexes.

A computation is performed on a lambda expression by reducing redexes in it until it is in normal form. The alpha conversion rule may be used freely to change the names of bound variables, in cases where reduction is prevented by a name clash. Using what has been described so far the lambda calculus can be used to express any computation [Church 41].

The reduction process is not guaranteed to terminate, for instance consider

$$\begin{aligned} & (\lambda g. g g) (\lambda g. g g) \\ = & (\lambda g. g g) (\lambda g. g g) \quad [\text{beta reduction}] \\ = & (\lambda g. g g) (\lambda g. g g) \quad [\text{beta reduction}] \\ & \dots \text{ and so on} \end{aligned}$$

This expression has no normal form: no sequence of reductions will terminate. For some expressions, termination depends on the order in which reductions are carried out. For instance suppose that, given a choice, the rightmost reduction is always performed first. This strategy is known as *applicative order* evaluation, and using it the reduction of the following expression does not terminate:

$$(\lambda x. y) ((\lambda f. f f) (\lambda g. g g))$$

This consists of the previous expression (the reduction of which does not terminate) supplied to a function that throws it away. Since normal order reduction implies reduction of the argument before the execution of the function the reduction of this expression will not terminate. Note, however, that two distinct normal forms cannot be obtained from the same lambda expression: this follows from the *Church-Rosser Property* of the lambda calculus.

The phrase *normal order* is used to describe the opposite strategy: that the leftmost reduction should always be done first if there is a choice. It is known that if an expression has a normal form then normal order reduction will terminate but may not do it in the fewest possible reductions. For instance, consider

$$(\lambda x. \text{plus } x \ x) ((\lambda y. y) \ z)$$

The normal form for this expression is "plus $z \ z$ ", and if the applicative order strategy is adopted, this will be obtained in two reductions. If the normal order strategy is used the expression will first reduce to

$$\text{plus } ((\lambda y. y) \ z) \ ((\lambda y. y) \ z)$$

and will finally achieve the same answer after three reductions.

Combinator Reduction

Functions can also be written down without the use of lambda. Any lambda expression can be rewritten into an equivalent form that does not include any lambdas. The best-known way of achieving this is by the introduction of two basic functions, known as S and K.

$$S = (\lambda a. (\lambda b. (\lambda c. a c (b c))))$$

$$K = (\lambda a. (\lambda b. a))$$

The rewriting is performed by using the following identities.

$$(\lambda a. a) = S K K \quad [1]$$

$$(\lambda a. m) = K m \quad [2]$$

$$(\lambda a. f g) = S (\lambda a. f) (\lambda a. g) \quad [3]$$

Any form in the target expression that resembles the left hand side of one of these should be replaced with the corresponding right hand side. [1] applies to the identity function. [2] applies when m is an expression containing no unbound occurrences of a. [3] applies in any other case: f and g may contain occurrences of the identifier a. These substitutions may be applied in any order (they all preserve the value of the expression being transformed), though if

$$(\lambda a. (\lambda b. f))$$

is encountered the transformation for b will have to be performed first.

In order to verify that each of these substitutions preserves the value of the expression being transformed, it is merely necessary to apply each to several arguments until they become identical. So:

$$\begin{aligned} [1]: \quad (\lambda a. a) x &= S K K x \\ x &= K x (K x) && [S \text{ reduction}] \\ x &= x && [K \text{ reduction}] \end{aligned}$$

S K K is the identity function, and is frequently written as I. SKIM derives its name from the phrase "S, K, I reduction Machine".

$$\begin{aligned} [2]: \quad (\lambda a. m) x &= K m x \\ m &= m && [K \text{ reduction}] \end{aligned}$$

(m contains no occurrences of a)

$$\begin{aligned} [3]: \quad (\lambda a. f g) x &= S (\lambda a. f) (\lambda a. g) x \\ [a / x] (f g) &= (\lambda a. f) x ((\lambda a. g) x) \\ ([a / x] f) ([a / x] g) &= ([a / x] f) ([a / x] g) \end{aligned}$$

Rather than writing down the value of such expressions it is frequently more convenient to write down an equation which they obey and which defines them.

So, S and K are defined by

$$\begin{aligned} S a b c &= a c (b c) \\ K a b &= a \end{aligned}$$

At this stage it is worthwhile for the reader unfamiliar with combinators to try transforming one or two lambda expressions into the equivalent forms employing no lambdas, merely occurrences of S and K. It will soon become apparent that the transformation process is immensely tedious, and the result unreadable, but

fortunately the process is easily mechanised.

Example:

$$\begin{aligned} & (\lambda y. \text{fn } x \text{ y } x) \\ &= S (\lambda y. \text{fn } x \text{ y}) (\lambda y. x) \\ &= S (S (\lambda y. \text{fn } x) (\lambda y. y)) (K x) \\ &= S (S (K \text{fn } x) I) (K x) \end{aligned}$$

Next it is worth trying to evaluate the resulting mess of combinators. This is also immensely tedious, the process for doing so is like that of the reduction of lambda calculus but simpler. Any part of the expression that takes the form

S a b c

can be replaced with

a c (b c)

and any

K a b

can be replaced with

a.

Example:

$$\begin{aligned} & S (S (K (\text{fn } x)) I) (K x) v \\ &= S (K (\text{fn } x)) I v (K x v) \\ &= K (\text{fn } x) v (I v) (K x v) \\ &= \text{fn } x (I v) (K x v) \\ &= \text{fn } x v x \end{aligned}$$

(as expected: compare with the abstraction example above).

Decisions about the order in which to perform reductions are exactly the same as in the lambda calculus.

Functional Languages

Many modern high level languages use as their main control and structuring facility a procedure mechanism that bears a passing resemblance to an applicative order version of the lambda calculus.

```
PROGRAM P ( OUTPUT );
FUNCTION FN ( X : INTEGER ) : INTEGER;
BEGIN
  FN := X * 3
END;
WRITELN ( FN ( 12 ) )
```

For instance, the Pascal program above has a strong intuitive resemblance to

$(\lambda \text{fn. fn } 12) (\lambda x. \text{multiply } x \text{ } 3)$

but more complex lambda expressions cannot be represented in such a direct form in Pascal, because functions are not thought of as first-class citizens in that

language. Instead Pascal has a collection of other features that the lambda calculus does not provide: assignment statements, conditional statements, loops, more precise requirements concerning order of evaluation and so on.

Functional languages (of the kind considered in this thesis) are based firmly on the lambda calculus. Full generality in the manipulation and construction of functions is held to be such a powerful and elegant idea that its inclusion as a programming language construct is viewed as being essential, and if normal order evaluation is used then no other control construct is necessary. Functional languages do however add the following ideas to the lambda calculus:

1. A family of elemental data values on which the program operates. The lambda calculus is sufficiently powerful that this is not necessary, but it is notationally convenient and provides a standard form for the display of results. Typically integers, characters and data pairs are built in to such a language.
2. Syntactic sugaring. Sticking to the minimal facilities that the lambda calculus provides would be nothing short of masochistic: designers of functional languages borrow freely from mathematical notations in order to make programs clearer and more communicative, while retaining function application and combination as the main semantic basis of the language.
3. A type system. This can help the programmer to spot careless errors by rejecting programs that are definitely wrong.

Despite these additions, all the *computation* occurs by the reduction of functions, and the correspondence between a program and a function is retained. The semantics of Pascal can be described in terms of mathematical functions, but the mapping is very contorted and is rarely of any help to practical Pascal programmers. Many programs can be expressed far more concisely and elegantly in a functional language, and this has led to the idea that larger scale experiments with functional languages should be tried. Unfortunately, these have been hampered by implementation technology.

Functional languages are quite tricky to implement: because the ideas behind them are based on idealised models of computation rather than on what could be made to generate efficient code, the results do not run well on conventional machines and have gained a reputation for inherent slowness. In contrast, the correspondence between Pascal programs and the equivalent machine code programs is fairly well understood, with the result that compiled Pascal could potentially run as fast as hand written machine code.

The first published implementation of a functional language with normal order semantics which was acceptably efficient appeared in 1976 [Henderson 76]. A lambda expression representing the program was represented as a tree in computer memory, very much like an interpreted Lisp program. The binding of variables to values was arranged by keeping a linear list of current bindings. The most important step forward in this evaluator was a strategy, known as *laziness*,

for achieving normal order semantics without causing a large increase in the amount of work to be done. A careful examination of the example of this extra work (in the section on lambda calculus) shows that extra reductions introduced by normal order reduction are really just repeated calculations: a lambda substitution can cause text to be duplicated, requiring any work in the duplicated text to be performed more than once. This problem can be solved by viewing the lambda expression as a graph and representing duplicated text by a shared subtree. When this subtree is reduced it can be overwritten with the reduced version, so that the reduction will not be performed again. This idea was first suggested in [Wadsworth 71].

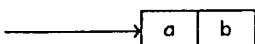
Henderson's evaluator was only "efficient" in the sense that the redundant recalculation that a naive implementation of normal order reduction implies was avoided. It still ran much slower than a compiled Pascal program running on a machine of equal power because it took the form of an interpreter acting on a graph in memory. Conventional machines are quite slow when operating in this way.

Practical Combinator Reduction

The use of combinator reduction as the basis for a practical implementation was first suggested by David Turner in 1979. His method is to rewrite the lambda expression using combinators, and then to reduce the resulting combinator expression. The expression to be reduced is represented as a binary directed graph, with function application denoted by a cell of the graph. Thus the expression

a b

is denoted by a pointer to a binary cell in computer memory containing a in its head and b in its tail:

a b is denoted by 

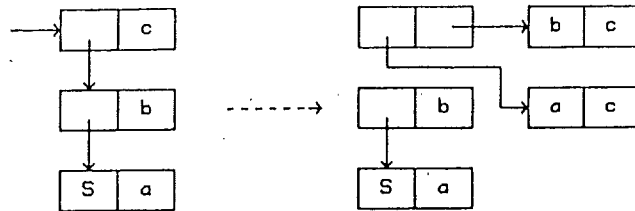
As a general rule lower case letters are used in these diagrams to represent arbitrary expressions (i.e. atomic values or pointers to arbitrary graphs) The graph contains no other symbols: in particular, nothing corresponding to a lambda expression. A number of the elemental values in the graph are recognisable as combinators. A combinator reduction occurs by transforming the graph in the appropriate way: for instance, consider the expression

S a b c

which should be replaced with

a c (b c).

This is represented by the following graph transformation:

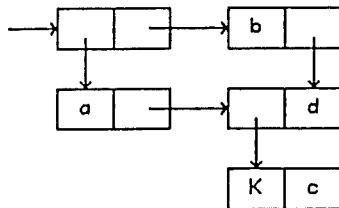


Note that the lower portion of the original graph must remain undisturbed, in case other pointers elsewhere in the machine point into the middle of this structure. Two new cells are provided by the store management system. It is worth bearing in mind at this point that a *cell* represents a *value*, and that rewriting never changes the value denoted by any cell. A reduction, either using combinators or in the lambda calculus, never changes the value being examined but simply renders it in a different form. Thus, in the left side of the diagram above the three cells represent the values $S a$, $S a b$ and $S a b c$. This fact remains true after the transformation. Once any portion of the graph is in normal form it will never change again (although it may be discarded).

In order to implement normal order reduction, the next reduction to be performed is found by following the chain of head pointers from the root of the expression being reduced. However, the order in which reductions are performed will not affect the final result. Laziness is achieved by the sharing of subgraphs: for instance, consider the reduction of the expression

$$\begin{aligned}
 & S a b (K c d) \\
 &= a (K c d) (b (K c d)) \quad [S \text{ reduction}] \\
 &= a c (b (K c d)) \quad [K \text{ reduction}] \\
 &= a c (b c) \quad [\text{redundant } K \text{ reduction}]
 \end{aligned}$$

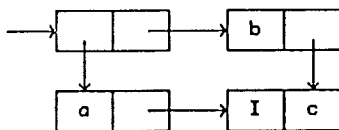
The repetition of the K reduction will be avoided because when it happens the first time its representation in the graph will be overwritten. After the S reduction the graph will become



This represents the value

$$a (K c d) (b (K c d))$$

but if either K in this expression is ever reduced then the graph will be overwritten as

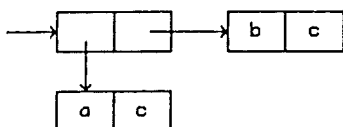


and so the other K reduction (when the expression's textual form is examined)

will have happened automatically. When a K is reduced the graph must be overwritten using an I combinator, so there is in fact some residual cost, but the reduction of an I is so simple that the diagram above can reasonably be viewed as denoting

a c (b c).

A node with I as its head is called an I-node and is really garbage: it acts merely as an indirection node, and the expression would be better off without it. On a practical reducing machine such nodes would be detected and removed by the storage management system: For instance, running the garbage collector on the graph above would cause it to be transformed into



This explains the basis of Turner's method, but various other points must be made, without which the whole scheme would still not be very practical.

S, K and I are the only functions that are strictly necessary, but a considerable performance improvement is obtained if a larger family of combinators is used. The new combinators that Turner recommends are called B, C, B', C' and S' and are defined by the following reduction rules:

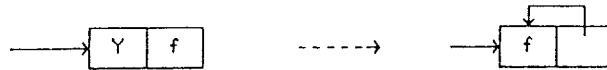
B a b c	= a (b c)
C a b c	= a c b
B' a b c d	= a b (c d)
C' a b c d	= a (b d) c
S' a b c d	= a (b d) (c d)

The next section attempts to explain the reasoning behind the choice of these combinators, which may otherwise seem rather mysterious, and give some idea of how they work together at run time. It is important to note that while a simple transformation from lambda expressions to combinator graphs is quite straightforward, a number of optimisations can be made: in general, the opportunities for generating better combinators are just as complex as the problem of generating good code from a high level language for any machine instruction set (e.g. see [Peyton Jones 82] which has suggestions concerning how to generate improved combinators from large parallel WHERE clauses in Miranda or SASL programs).

Values that are defined recursively in a functional program can be expressed using the existing combinators, but one extra function in particular seems appropriate for this task. It is called Y and is characterised by

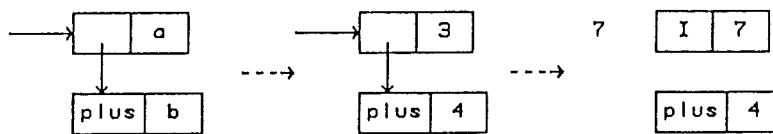
$$Y f = f (Y f)$$

The precise details of the way in which Y is used will not be given here, but as a general rule a simple recursive function can be represented using an occurrence of Y. As a graph operation, instead of performing the rewrite in the obvious manner that the equation implies, a cycle can be introduced into the graph:



This leads to a more efficient representation of recursive functions, with the disadvantage that the underlying store management system must be able to cope with cyclic graphs.

Various fundamental data types are provided, with operations on them. For example, integers can be represented by using higher order functions, but machine words are considerably more efficient. This means that a family of basic operations on integers must also be provided. Unlike S and K these functions must evaluate their arguments before yielding their results, for example consider the addition function:



The reduction of a strict combinator

A function is said to be *strict* on its argument if the argument must be evaluated before the result of the function can be produced. In the context of SKIM the word “combinator” is used to mean any primitive function, including plus and the like. The plus combinator works by calling the evaluator recursively on its two arguments in turn, and then verifying that each evaluates to an integer. The two values are then added, and the result overwritten (again using an I combinator) in the cell containing the second argument in order to prevent recalculation. This sort of extension to the lambda calculus is justifiable because integers can be defined in the lambda calculus, so their treatment as a special case is purely a performance issue. The value returned by the reduction of this expression is 7 rather than a pointer to a structure, which is why the final version of the diagram above has a 7 instead of a pointer at its top left hand corner.

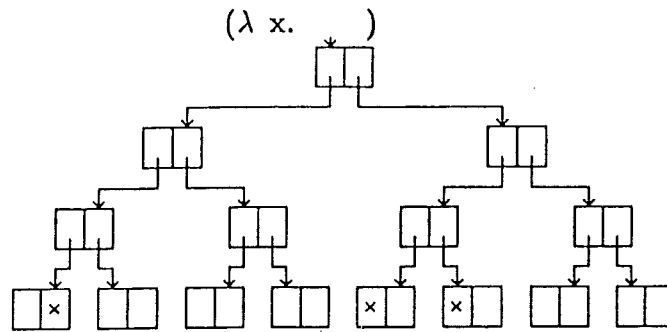
Understanding Turner’s combinators

What is the rationale behind Turner’s choice of combinators? In order to understand this, and to gain some insight into how combinator graphs behave at run time, it is necessary to visualise an expression as a parse tree. Consider the expression

$(\lambda x. \text{EXPR})$

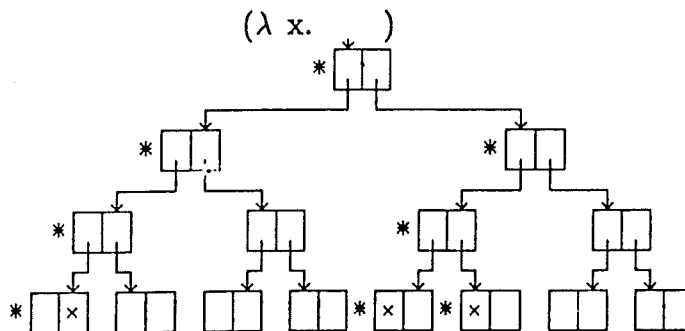
where EXPR is a large expression containing no other occurrences of lambda. EXPR consists of a binary tree, where a node represents function application and each leaf is either the bound variable x, or some other value (presumably defined

by some textually enclosing lambda clause).



In this diagram, as in others in this section, the intent is to portray "a tree" with no requirement for the tree to be balanced or of some particular size. The positioning of free variables such as x in this particular example is purely illustrative.

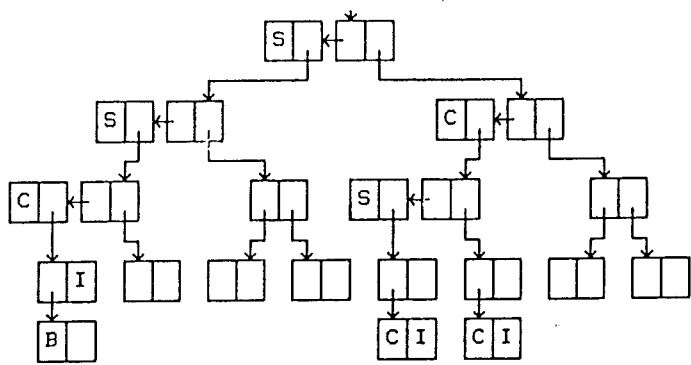
When this lambda expression is applied to some value n , the result is equivalent to a copy of this expression, with every occurrence of x replaced by n . In order to achieve this without copying the whole structure, it would be best to copy just those cells which contain x , or have descendant cells which contain x . To this end we mark with a $*$ every cell that needs to be copied in order to instantiate x , resulting in a branching *path* that must be followed by the value x .



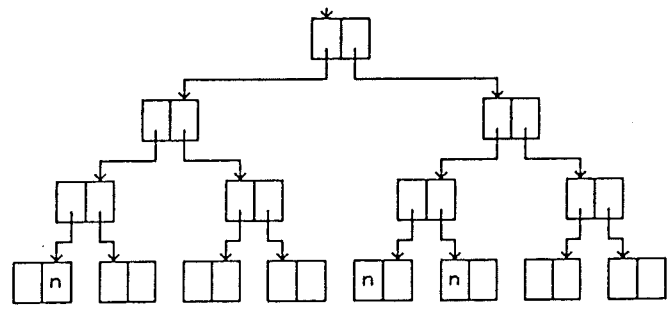
There are three varieties of node marked $*$: either x is needed by both descendants of such a node, or by the right descendant, or by the left descendant. The transformations required to perform these operations are exactly those provided by the three combinators S , B and C . This is easier to understand if the definitions of these three combinators are written down with some unnecessary brackets:

$$\begin{array}{ll}
 S \ l \ r \ x & = \ (l \ x) \ (r \ x) & [x \text{ required by } l \text{ and } r] \\
 B \ l \ r \ x & = \ l \ \ \ \ (r \ x) & [x \text{ required by } r \text{ only}] \\
 C \ l \ r \ x & = \ (l \ x) \ r & [x \text{ required by } l \text{ only}]
 \end{array}$$

The transformation from lambda expression to combinator graph occurs by replacing the nodes marked * in the diagram above by a double node containing B, C or S depending on the use of x by that node. In addition to this all occurrences of x should be replaced by an I combinator, yielding:

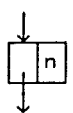


This graph is equivalent to the original one. A number of new nodes have been added to it, but the structure of the original lambda expression is still clearly visible. If this whole expression is applied to some value n, then all the combinators can be reduced and the resulting value will be



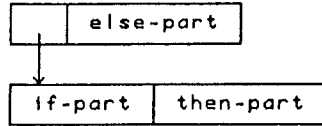
exactly as intended. Those portions of the result that are not dependent on n will be represented by subgraphs that are shared with the original graph.

A reader wishing to try this should find an extremely large piece of paper, or a blackboard, and copy the diagram before this one onto it. Then add one extra node



at the top, pointing to the main expression. It should now be possible to reduce all the combinators on the graph, yielding the value above whilst leaving the original graph intact.

In effect, the combinators control the copying of a minimal portion of the graph, in order to generate the new value. Also, a normal order reduction strategy will ensure that if a portion of the graph does not get used then no effort will be expended in copying it. This happens quite often in conditional statements, which take the form



The if-part of this expression will evaluate to either TRUE or FALSE, where

$$\begin{array}{ll} \text{TRUE } a \ b = a & \text{[the same as K, in fact]} \\ \text{FALSE } a \ b = b & \text{[equivalent to K I]} \end{array}$$

Either then-part or else-part will now be abandoned, and the other taken as the value of the whole expression. Normal order combinator reduction ensures that no work is expended on constructing the value of the abandoned portion.

In real implementations there are one or two simple optimisations that can be made at this point. One example of this is that

$$B \ a \ I = a$$

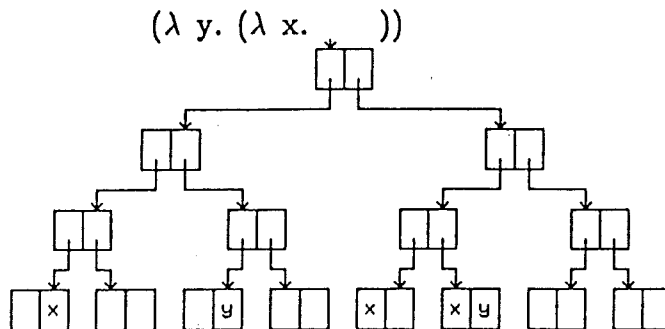
for any a. This can be checked by supplying an arbitrary argument to each, and verifying that they produce the same result:

$$\begin{array}{ll} B \ a \ I \ x = a \ x & \\ a \ (I \ x) = a \ x & \text{[B reduction]} \\ a \ x = a \ x & \text{[I reduction]} \end{array}$$

obviously true.

This pattern occurs in the bottom left hand corner of the main example, and tends to occur quite often near the leaves of a translated tree. The substitution reduces the size of the tree and speeds up reduction, so it is a useful local optimisation. This sort of optimisation will be omitted in this section, however, as it makes the resulting trees somewhat harder to understand.

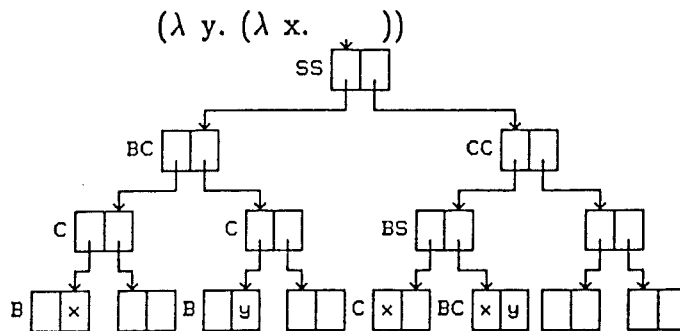
The need for S', B' and C' becomes clear when we try to abstract a second variable from some simple expression. For instance, consider the expression



The abstraction of the variable x must be performed first, and as the previous

example showed it generates eight new cells on the tree, and eight combinators. It would seem reasonable that the abstraction of y should generate seven extra cells, because the path for y includes seven cells. Unfortunately if S , B and C are used then the extra nodes generated by the abstraction of x must also be counted, and so each node of the original graph that is on the path for both x and y will generate more nodes than one might expect: in this example, the abstraction of y will generate nine extra cells. With more variables being abstracted from an expression this phenomenon gets worse and worse.

The use of S' , B' and C' means that only one extra node is generated for each cell on the path of each combinator. Rather than annotating the paths with $*$, sequences involving B , C and S seem more useful.

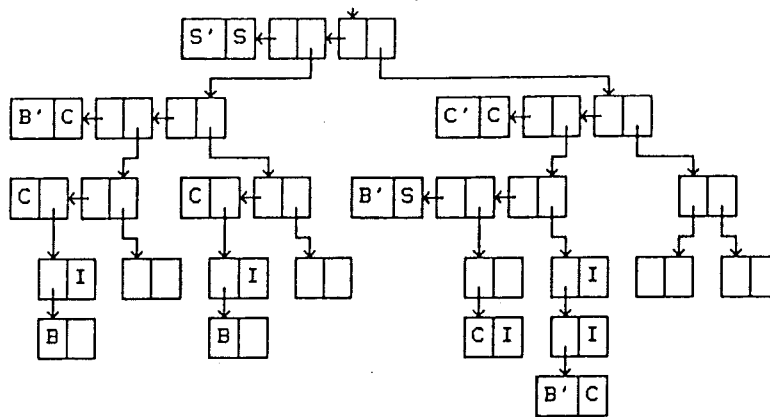


A node containing two annotations can be represented using a dashed combinator applied to a simple one. This can be verified by applying arguments to them, in the following manner:

$$\begin{array}{ll}
 B' B l r y x & [r \text{ wants both arguments}] \\
 = B l (r y) x & [B' \text{ reduction}] \\
 = l (r y x) & [B \text{ reduction, achieving the desired result}] \\
 \\
 S' C l r y x & [l \text{ wants both, } r \text{ wants only } y] \\
 = C (l y) (r y) x & [S' \text{ reduction}] \\
 = (l y x) (r y) & [C \text{ reduction, achieving the desired result}]
 \end{array}$$

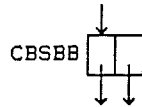
There are nine possible cases; the reader should be able to fill in the other seven.

Using this method on the example above yields the following tree:

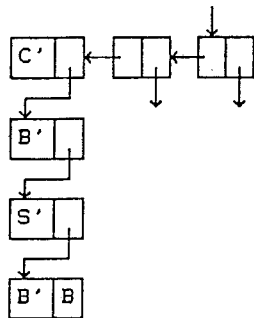


Once again, the original structure of the graph is still clearly visible.

This scheme will also cope with the instantiation of more arguments without any requirement for inventing S'' , S''' and so on. A node of the form



will be transformed into



In large programs such nodes are not unusual. This phenomenon was first pointed out by Kennaway and Sleep [Kennaway 82] and the term *director string* used to describe such a regular string of combinators. Chapter five describes a way of making use of this regularity in order to speed up reduction.

The abstraction algorithm as I have described it causes an extra node to be required for every node on the path of every instantiated variable. In the worst case this can result in a quadratic expansion in the number of nodes needed to represent an expression, but fortunately there is a simple global optimisation (known as *balancing*) which reduces this to a logarithmic growth. The introduction of B' , C' and S' also provides a number of further opportunities for local optimisation near the leaves of a combinator tree. Turner's paper gives details of these. The emphasis here has been on trying to give some idea of the reasoning behind the precise set of combinators that Turner suggests.

Microcode

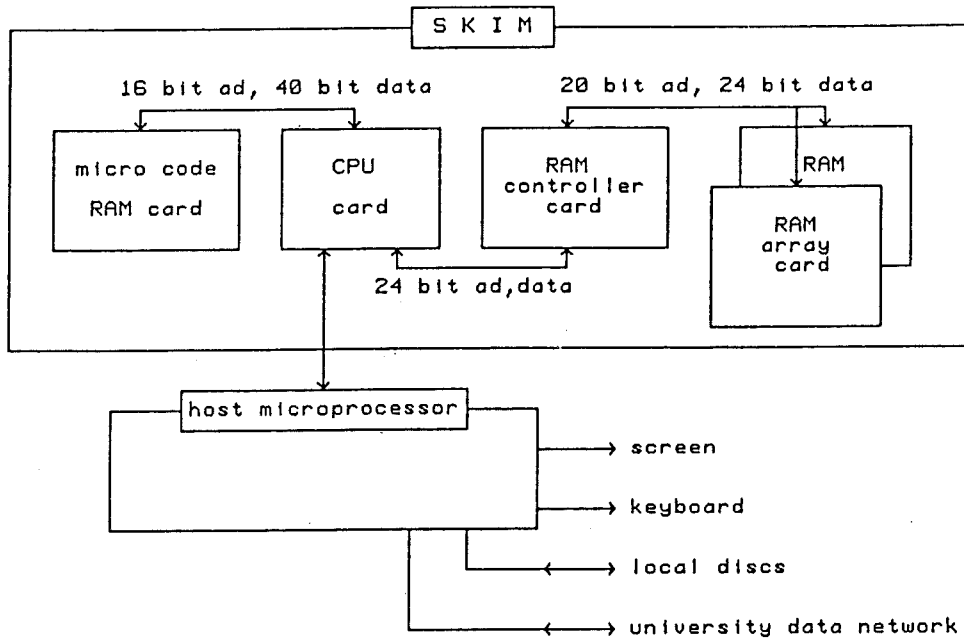
The combinator reduction scheme described in [Turner 79] and the preceding chapter can be implemented as a machine coded interpreter on a conventional computer and provides a way of implementing functional languages. Unfortunately the resulting performance is unsatisfactory, and acts as a serious impediment to the development of large functional programs.

A solution to this problem is to build combinator reduction hardware. Since the obvious implementation technique on a conventional machine involves an interpreter, it seems natural that improvements in performance should be sought by implementing this interpreter more directly.

SKIM is a very simple processor with a microarchitecture designed specifically for combinator reduction. Microcoding SKIM to reduce combinators has been highly rewarding, and has led to a far greater understanding of the combinator reduction process. Conversation with people who have machine coded combinator reduction on a conventional machine indicates that their attention was focussed on circumventing the drawbacks of their particular target machine rather than seeking better solutions to the problem itself. Using a specialised processor focussed attention on implementing functional languages, rather than on circumventing an inappropriate architecture. A combinator reducer coded on an 8-MHz 68000 goes at about one thirtieth of the speed of SKIM, and was considerably harder to write than SKIM's microcode [Wray 83].

This chapter gives a description of SKIM's microcode architecture, and shows how various aspects of its design relate to Turner's combinator graph reduction scheme. More detail, including the complete syntax of microcode assembler programs, can be found in [Stoye 83].

Physical Organisation



SKIM currently occupies five circuit boards:

- the main CPU card

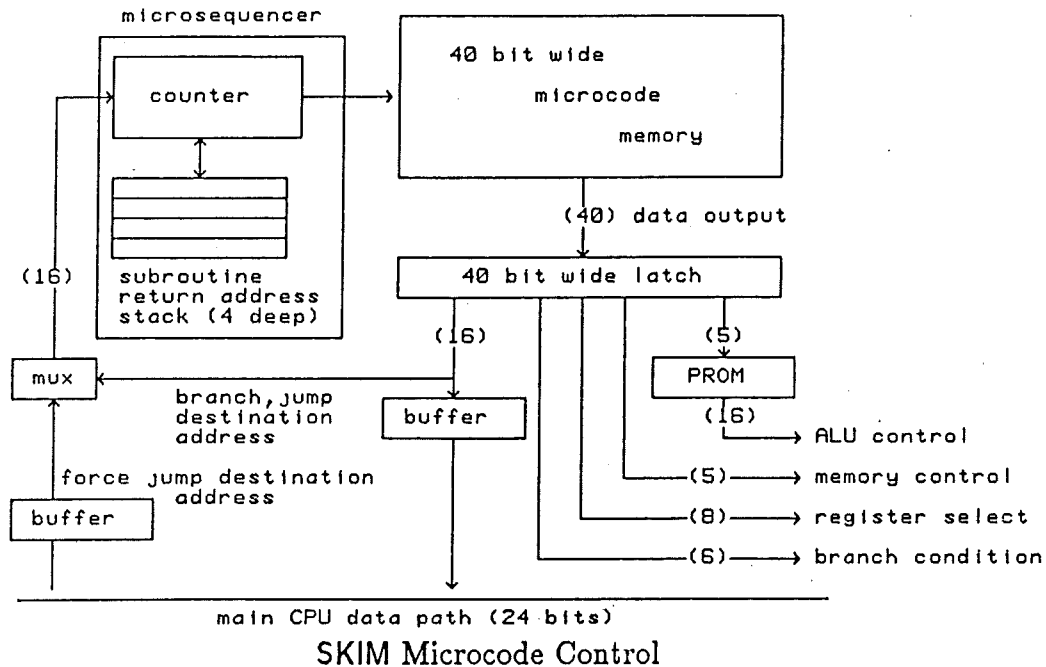
- a microcode memory card

- dynamic memory refresh and buffering card

- two main memory array cards, each containing 64K nodes

The circuitry is constructed from standard TTL components and MOS memory chips. All input and output is done via an 8-bit parallel port to a host microprocessor system, which also acts as a diagnostic processor. Up to two more memory array cards could be added if desired, and the 64K bit memory chips in them upgraded to 256K ones. So far, however, a shortage of memory has not been a problem.

Microcode



The microcode memory is constructed using RAM chips, and could contain up to 64K 40 bit words of microcode. SKIM I had space for only 2K words of microcode, and this limitation turned out to be a serious inconvenience for its programmers. At the time of writing, about 4K words of microcode program have been written for SKIM. The microcode memory is loaded by SKIM's diagnostic processor before SKIM starts computing, and remains unchanged from then on.

The microcode is controlled by a microsequencer circuit, based around four AMD2911 chips. This issues addresses to the microcode memory, and also maintains a stack of return addresses for a microcode subroutine mechanism. Every clock cycle (250 ns) the microsequencer increments its address and reads a word from the microcode memory into the 40 bit wide instruction pipeline. The values in this latch control what operations are to be performed in this cycle by the rest of the processor.

A single microcode instruction can cause several things to happen. As the diagram indicates, different fields of the microcode word are used to control different parts of the CPU. One field is used either for control transfer destinations or for constants to be used by the ALU: this causes slight constraints on the combinations of instructions that can occur in a single cycle, but is worth the saving in microcode memory. The words *horizontal* and *vertical* are often used to describe microcoded architectures: horizontal means a very wide microcode word and the ability to perform many operations in a single cycle, vertical means a narrow microcode word and more complex circuitry, in an

attempt to find the most frequently used microcode instructions and encode them using fewer bits. Like many practical designs SKIM represents a compromise between these two philosophies, though in critical parts of the microcode it achieves a good degree of parallelism and can thus claim to be more horizontal than vertical.

There are several different ways in which a microcode transfer of control can occur:

A *jump* causes an unconditional transfer of control to a new location in the microcode memory. Special circuitry detects jumps as soon as they appear, so that instruction prefetch occurs from the jump destination. This means that jumps do not cause any delay.

A *branch* is a conditional transfer of control. As the address of the next microcode instruction to be obeyed will depend on the result of some ALU operation, it is not possible for the pipelining to be successful all the time. The strategy adopted is the simplest: the next instruction is prefetched by the microsequencer. This means that if the branch is taken, a cycle will be wasted while the sequencer is loaded with the destination address, and the microcode memory accessed.

A *subroutine jump* pushes the current microsequencer address on the microsequencer return address stack, and performs a jump. No delay occurs.

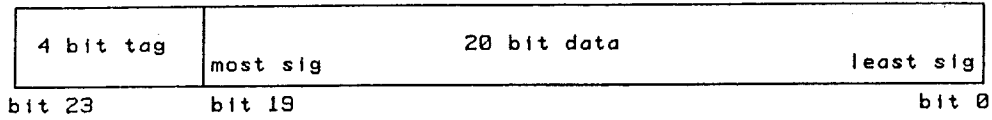
A *subroutine return* pops a return address from the microsequencer return address stack. No delay occurs.

A *force jump* examines the value on the data bus in this cycle, and if a tag test identifies it as a microcode address then a jump occurs to that location (see the section on ALU words concerning tags). A delay of one cycle occurs if this jump is taken. This operation is not found on most processors, but has two very important uses: it allows arbitrarily nested subroutines in the microcode, and it is used as a dispatch mechanism in certain critical sections of the graph reduction microprogram.

A transfer of control can occur in the same cycle as an arithmetic operation. A careful analysis of the critical parts of the combinator reducing code reveals that a great many instructions have tests associated with them. When writing such code every effort is made to try to guess which control path is followed most often, and to make this the path that takes fewest conditional branches.

Some microcoded machines employ a three stage pipeline: this means that an extra latch is inserted between the microsequencer and microcode memory, and that instruction execution, instruction fetch and microsequencer increment can all occur concurrently. Unfortunately it increases the delay in taking conditional branches, and so SKIM does not use such a mechanism. SKIM has a high branch bandwidth, i.e. it can make decisions and respond to them quickly.

ALU words



The main processor operates on 24 bit words, which are organised as 4 bits of *tag* and 20 bits of *data*. The tags are used to distinguish different types of values at runtime. As a general rule, a value with a tag of less than 8 is a pointer to main memory, and a value with a tag of 8 or more is an atomic object of some kind (such as 'integer' or 'microcode entrypoint'). Every aspect of SKIM's operation is affected by the fact that all values are tagged at runtime. Some examples of this have already appeared: for instance, the force jump form of control transfer recognises microcode entrypoints by examining the tag of the value on the CPU's main data path.

Branch Conditions

Earlier attempts to build or program combinator graph reducers had shown that critical portions of code tend to have a great many branch operations: they frequently take the form "get the next node on the graph, and depending on what it is (i.e. the tag) do one of the following:". For this reason, much of SKIM's most complex circuitry is concerned with providing a rich variety of branch operations. The ability to test tags without obeying shift and mask instructions first is of great importance.

Any microcode instruction can include a conditional branch. The branch usually tests the value on the main ALU data path during the instruction. SKIM has the usual arithmetic tests but in addition the following forms are available:

BTAGEQ	tag	label	if the tag of the value on the databus is equal to the tag specified here, take the branch
BTAGNE	tag	label	if the tag of the value on the databus is not equal to the tag specified here, take the branch
BTAGLT	tag	label	if the tag of the value on the databus is less than the tag specified here, take the branch
BTAGGE	tag	label	if the tag of the value on the databus is not less than the tag specified here, take the branch

The form on the left shows the syntax used by the microcode assembler. The presence of these operations means that a value can be moved and tested in a single microcycle.

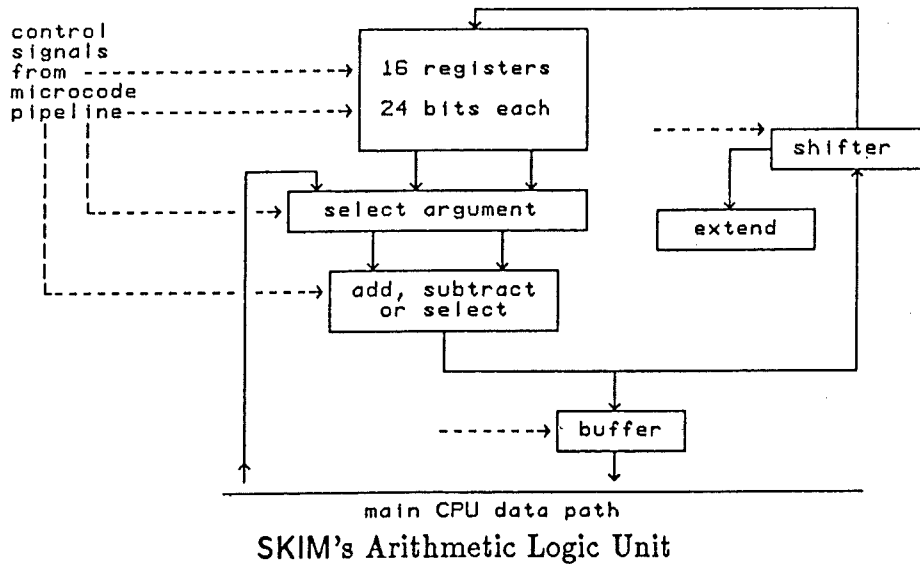
Even more complex cases can occur, when a force jump occurs in the same cycle as a conditional branch. This effectively allows a data move and two tests to occur in a single cycle. So, the instruction

```
r1 := r0    FJMP comb BTAGNE appl lab;
```

which is shown here in as a microcode assembler instruction, can be read as:

"Move register 0 to register 1. If the value on the bus (i.e. the value being transferred) has a tag equal to the constant 'comb' (i.e. it is a combinator, and its data field is a microcode address) then force the data field into the microsequencer. Otherwise, if the tag is not equal to 'appl' (i.e. the value is not an application pointer) branch to 'lab'. Otherwise continue with the next instruction".

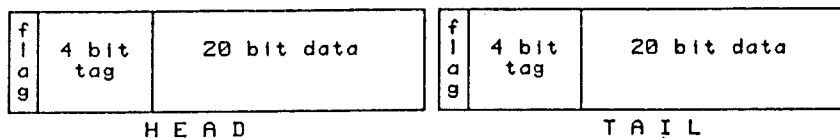
The ALU



SKIM's ALU is provided by six AMD2901 chips. Each of these provides a four-bit slice of sixteen registers, addition, subtraction and selection circuitry, and an extra extend register to aid rapid multiplication and division. In a single cycle two registers can be fetched, an arithmetic operation performed and the result saved in one of the registers. The current microcode instruction, latched in the microcode pipeline, provides signals that control every aspect of the ALU's operation. The addition and subtraction operations only operate on the data portion of a word.

Main Memory

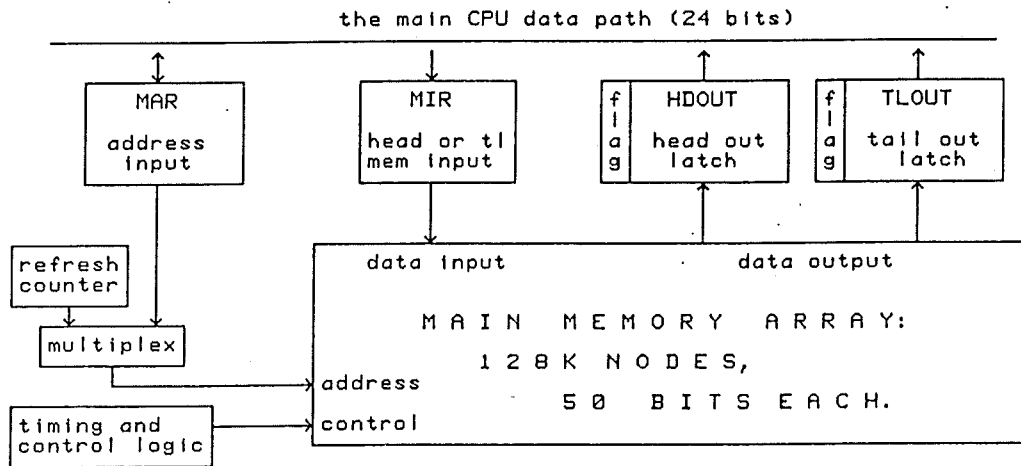
SKIM's main memory is organised into *nodes*. Each node (or *word*, or *cell*) consists of two ALU words, which are referred to as the head and tail halves of the node. In addition there are two extra bits which are specifically intended for use by a garbage collector. SKIM can address up to 1M nodes, since the data portion of a CPU word is twenty bits wide.



A Main Memory Node

SKIM's main memory is usually used to represent a binary directed graph structure. Combinator graph reduction, which is the algorithm that SKIM was designed to execute, involves loading such a graph into memory and then transforming it according to a few simple rules. This process requires a continuous stream of new cells and so garbage collection is necessary, and the flags were designed specifically for this use. In fact there are four flags associated with each node, in case some other use for them appeared: so far, however, the other two are unused. The precise details of the garbage collection method will not be described here (a simple "mark and sweep" algorithm is used).

There are many different kinds of memory cycle, because controls for writing or reading in the various portions of a memory node are independent of each other. All memory cycles read an entire node, in addition a write may also occur to one half or the other of the node. Thus, rather than read and write operations the memory supports read and exchange, which turns out to be extremely useful in certain pointer manipulation operations. Exchange is extremely easy to implement for a memory system based on modern dynamic memory chips, as it is implemented at the physical level by a single read-modify-write memory cycle.



SKIM Main Memory Access

Access to the memory is achieved from the CPU by means of four latches:

MAR	Memory Address Register
MIR	Memory Input Register
HDOUT	Head Output Register
TLOUT	Tail Output Register

These latches can be read and written from the CPU in single cycles, a special bit in the microcode word allows a write to occur to the MAR in any cycle. When performing an operation on memory the microcode program must write to the MAR and (if the operation is a write) to the MIR. Then the memory cycle is triggered, and the results will appear in the HDOUT and TLOUT registers on the following microcode cycle. A certain amount of clock-stretching occurs so that, if necessary, the microcode is slowed down until the main memory is ready.

Flag input data is part of the memory cycle type, rather than being computed data. Flag outputs are not routed through the main data bus but passed to the

branch circuitry where conditional branches can be performed based on their values. They are extremely useful when performing a marking garbage collection, where the processor has to crawl over tree structures, setting flags on all cells that are accessible. The setting of flags, the moving of addresses and the testing of the structure of the resulting graph can all occur concurrently, with the result that the speed of the mark operation is quite close to the theoretical maximum imposed by the bandwidth of main memory.

A Graph Traversal Example

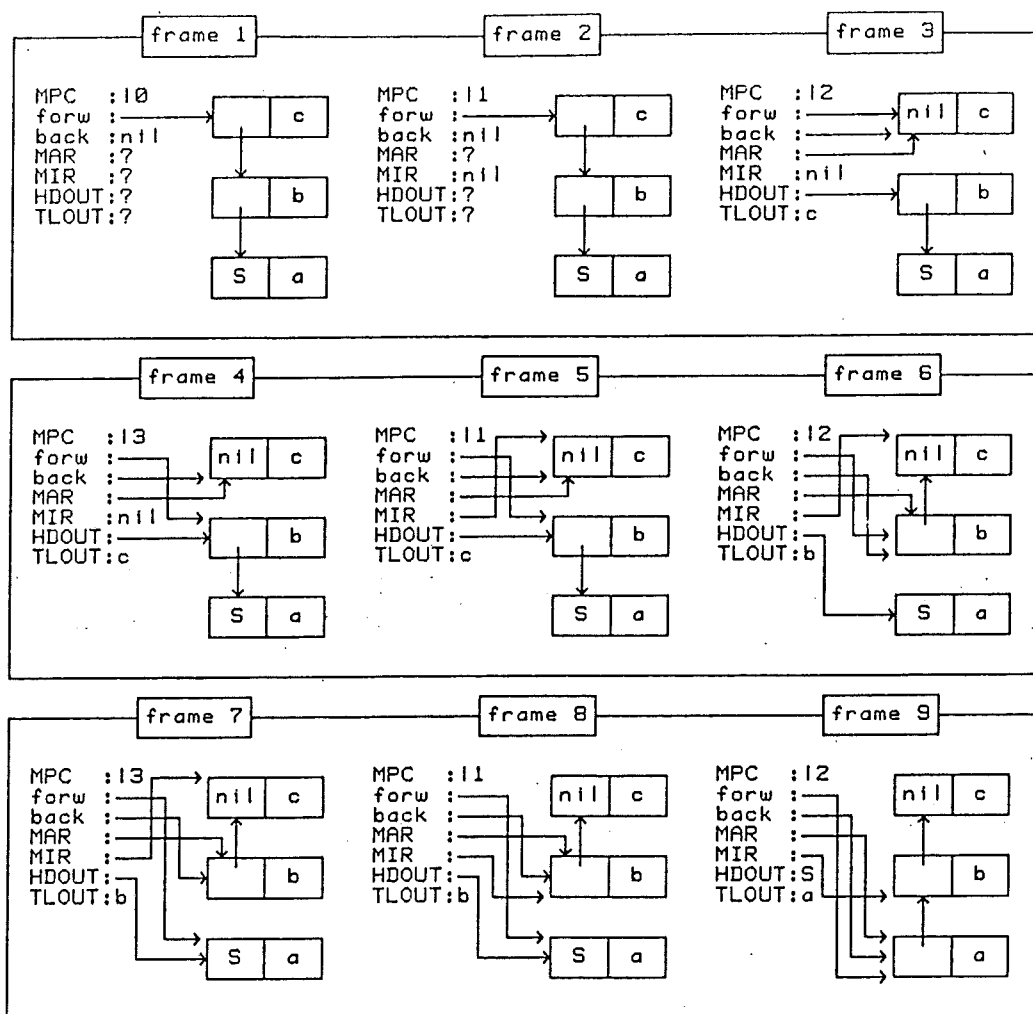
While not complete, the details given here are sufficient that a small example of some microcode can be shown. This will allow the reader to observe how the various parts of the CPU work together, and how various aspects of SKIM's design have been carefully tuned for combinator reduction. The example is a fragment of code which has the following task to perform:

“Follow a chain of pointers in the head spine of the graph provided until an object is reached at the far end. Reverse the pointers in the graph as you traverse them, so that the path can be retraced. If this object is a microcode entrypoint, branch to it. Otherwise, report an error.”

The head spine of a graph is those cells encountered by following pointers in the heads of cells, i.e. the top node, the node pointed at by the top node's head, the node pointed at by that node's head, and so on. Combinator graphs never have a cyclic head spine. This operation performs a typical part of the combinator reduction process: it occurs between combinator reductions, and constitutes the search for the next combinator to be reduced. It is vital that it execute quickly. In the notation of the microcode assembler the loop to perform this operation looks like this:

```
10:  MIR := back;
11:  MAR, back := forw  XHD;      ["exchange with head" memory cycle]
12:  forw := HDOUT
                                FJMP comb
                                BTAGNE appl error;
13:  MIR := back                  JMP 11;
```

This shows four microcode instructions, of which the last three form a loop. 'forw' and 'back' are the symbolic names of two ALU general-purpose registers, 'comb' and 'appl' are symbolic tag names. '10', '11', '12', '13' and 'error' are labels. The action of this program on a typical graph fragment is shown here, as a sort of cinefilm running at one frame per microcycle.



The execution of a simple fragment of microcode

The initial setup is shown in frame 1, with the graph to be traversed in register 'forw' and the values of the microsequencer (MPC), the backchain pointer 'back' and the memory access registers MAR, MIR, HDOUT and TLOUT. The instruction at 10 loads MIR with the value of back, after its execution the situation is as in frame 2. A precise blow-by-blow account of each cycle depicted here would be somewhat tedious, and can easily be traced though by the interested reader. However, it is worth pointing out that the process of pointer reversal in the graph can clearly be seen, and that the processor's data movement and control circuitry operate concurrently to perform the desired operation with very little time wasted.

This particular example has been chosen for careful study because it represents an operation that takes place before every combinator reduction. The S in the diagrams is intended to denote an S combinator, which is represented by a value with a tag of 'comb' and data that is the address of a passage of microcode that performs the S combinator reduction. The microcode at this address will reverse the last two links of the reversed chain and pick up S's three arguments. It will then rewrite this portion of the graph in accordance with S's rewrite rule, and return to this loop so that the next combinator can be located.

NORMA

It was suggested earlier that building a specialised machine allowed SKIM's design to concentrate on the problem in hand rather than on circumventing an unsuitable host architecture. This is only partly true: SKIM is built out of MSI TTL, and some of the building blocks available in the TTL logic family are not really ideal for the application. The use of custom chips would have allowed still greater design freedom. This approach has been taken at the Burroughs Corporation's Austin Research Center, in building NORMA.

NORMA (Normal Order Reduction Machine) [Richards 85] is another machine designed specifically to reduce combinators. It is slightly bigger than SKIM, and its prototype first worked in December 1984 with about twice the performance of SKIM. The AMD2901 ALU slices that SKIM is constructed around are replaced on NORMA with a special register file implemented using ULAs (semi-custom chips of a few thousand gates). The emphasis in this register file is on speed of data movement rather than on arithmetic operations: it contains a great many data paths, so that the contents of the registers can be permuted arbitrarily in a single cycle. This allows many optimisations over the methods that SKIM employs, in particular the arguments to combinators can be cached in registers before a combinator is reached, which eliminates the time that SKIM spends picking up the arguments for each combinator.

Another important optimisation is a stack cache, which caches all of the cells on the head spine. This cache can be accessed in a single microcycle, and substantially increases the total memory bandwidth available: a large fraction of all memory accesses on SKIM are to cells on the head spine. There is also support for performing a multi-way switch in a single microinstruction, based on testing a number of different conditions in a single cycle. This increases the branch bandwidth in line with the increase in data bandwidth brought about by the use of the register file and stack cache.

NORMA's garbage collection is partially overlapped with combinator reduction, in the following way. Like SKIM, NORMA uses a simple mark-and-sweep garbage collector, but unlike SKIM the flags attached to each memory node can also be accessed independently. This means that the sweep phase of garbage collection, which examines flags and produces addresses of free cells, can be executed in parallel with reduction.

Various Optimisations

This chapter first appeared in the proceedings of the 1984 ACM Lisp conference, under the title "Some Practical Methods for Rapid Combinator Reduction", by W. R. Stoye, T. J. W. Clarke and A. C. Norman. It represents the next level of implementation thinking beyond the previously understood method described in the previous two chapters: everything already described was known when SKIM's hardware was constructed, this chapter contains ideas that were produced later on. The original idea behind the reference counting scheme was suggested by Thomas Clarke. The idea for removing recursion from the combinator evaluation microcode is also used in NORMA.

The inclusion of the ideas presented in this chapter in SKIM's combinator reduction microcode has improved its performance by about fifty percent. The diversity of methods described suggests that there is still considerable room for improvement in the implementation of functional languages, because the lambda calculus is sufficiently far removed from a hardware-dominated model of computation that the implementor has considerable freedom in the choice of evaluation methods.

This chapter describes the following ideas:

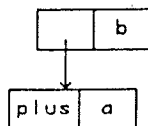
- Eliminating recursion in the combinator reduction microcode
- One bit reference counts on the heap
- Equating identical structures transparently
- Compilation of complex operations into microcode

The Elimination of Recursion

Turner's original scheme for combinator reduction is that expressions should be represented as graphs, with function application denoted by head linked lists of binary cells. For example, the expression

plus a b

is represented as



plus a b

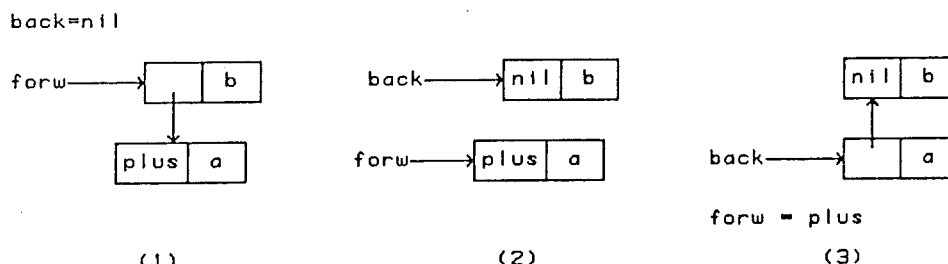
graph representation of an expression

Note that the 'application' pointers shown in this diagram are distinguishable from data pointers, so that a cell containing a and b represents 'b applied to a' rather than 'a pair with a on the left and b on the right'. Turner's algorithm for evaluating this object is to follow the chain of application pointers in the head sides of cells, pushing onto a stack the addresses of cells passed. When a combinator object (like plus) is reached at the extreme left hand end, a jump is performed in the reducer to code specific to that combinator.

The code for plus, to continue this example, checks on the stack that it has two arguments, calls the evaluator recursively on each of its arguments (they should both evaluate to be integer objects in this case), adds their values together and overwrites the top cell in the diagram with the new value. The overwriting occurs because there might be other application pointers to this fragment of the graph, and it is an important principle of lazy evaluation that no computation should be performed twice.

Note that plus is a strict combinator, i.e. it has to evaluate its arguments before producing a result. Other combinators (like S and K) do not have to do this, but in practice many of the combinators evaluated are strict ones.

There are two uses for a stack in this process, one to record the sequence of cells passed when traversing the left hand sides and another associated with the need to evaluate the arguments of some functions before performing the function itself. SKIM uses pointer reversal to cope with the first of these. Two registers are used when evaluating called **forw** and **back**, to hold the expression being evaluated and the list of reversed pointers. To evaluate an expression **forw** is initialised to point to the graph representing the expression and a null value is placed in **back**. Evaluation proceeds like this:

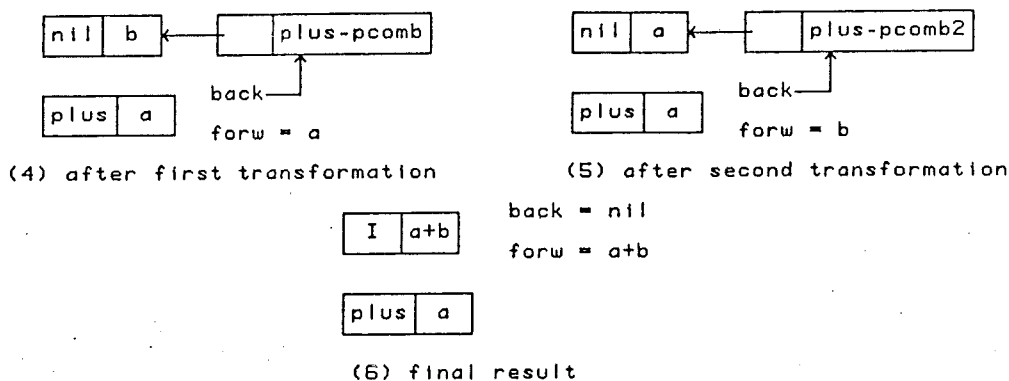


Pointer reversal during the evaluation of an expression

At this point evaluation continues at the code specific to the plus combinator. The combinator has to re-reverse the pointers in order to find its arguments, and so that the tree is left in a tidy state after the reduction. Pointer reversing performance is enhanced through the use of read-modify-write memory cycles so that fetching the forward pointer and writing the backward pointer occur in one memory cycle.

The original microcode used a stack for the implementation of strict combinators, when the evaluator was called recursively to evaluate the arguments. This stack was implemented as a list of cells on the heap, mainly because there was no hardware support for anything else.

The new microcode extends the idea of pointer reversal to deal with strict arguments. This is reminiscent of the pointer reversal in Deutch-Schorr-Waite garbage collection [Schorr 67], where both left and right subtrees of a cell must be marked without using any extra storage. The pointer reversal process is continued when evaluating the arguments a and b, leaving a special marker in the back chain saying where to continue computation of the suspended 'plus' operation.



Reduction of the expression plus a b using postfix combinators

The first part of the code for plus reforms the graph to look like diagram 4 above. The (plus a) cell has to be left unchanged, in case there are other pointers to it. A new cell has been generated, which sits on the back chain and contains a special marker which is designated plus-pcomb in the diagram. This marker is called a postfix-combinator or pcomb and it contains a microcode entrypoint, just like a normal combinator. Evaluation now continues, and the argument a becomes fully evaluated.

When the **forw** register contains an object that is fully evaluated, the reducer looks at the backward pointer. If it contains the null value, evaluation is complete. If not, the chain of pointers that it represents will be re-reversed until a cell is found with a postfix-combinator in its tail. A jump will occur in the reducer to the code represented by this pcomb.

In this way the first argument has been 'recursively' evaluated with a minimum of state-saving in the evaluator. There is now no data structure involved in evaluation except the expression graph itself, which fits in well with the reduction paradigm that SKIM is designed to support. The further evaluation of plus in fact involves a second pcomb, because the evaluator has to be 'reentered' again for the evaluation of b, but this is merely a repetition of the above process.

There have been two main benefits from the implementation of this algorithm, the first being a performance improvement of about ten percent overall due to reduced state-saving in evaluating strict combinators. The second is that the reducer has a very small amount of internal state, and is in fact simpler than the first version of the microcode, which used a stack. This proved a great advantage when adapting SKIM's microcode to provide a multi-tasking model of parallel reduction (see chapter nine). The combinator expression graph in SKIM's main memory is reduced at many places 'simultaneously', to simulate the operation of

a machine with many processors, and it is very important that this should not cause incorrect interactions when several processes try to reduce the same expression graph at the same time. Keeping all state information as part of a single graph structure has helped us concentrate on the interlocks that are required. The effect of having multiple physical processors involved in this operation has not yet been considered by us in detail.

It might be argued that better performance would be obtained from SKIM by the addition of a separate hardware stack, which could be used to remember the arguments of any suspended operations. Sadly this would add substantially to the complexity of SKIM's hardware, and would have made multi-tasking more complicated, while greater performance improvements could probably be gained by a simple associative cache on the main memory.

One Bit Reference Counts

One problem with combinator reduction is that the use of heap storage seems profligate. The process of reduction involves the continuous use of heap cells, and although SKIM's memory structure is carefully optimised to allow rapid marking during garbage collection it was found to be spending more than half of its time in the garbage collector when memory was only one third full. Most cells are only in use for a very short time, and then discarded.

Reference counting is often suggested as a replacement for or an addition to garbage collection, but the bookkeeping overheads concerned with doing it for very small scale objects are considerable. Every time a pointer to an object is copied the object has to be accessed and its reference count incremented and written back. As it is necessary to cope with increment overflow this can be an absurdly complex operation, so that although most storage may be reclaimed computation would proceed overall at a slower rate.

SKIM uses reference counts in a way that speeds up computation even for calculations that do not provoke garbage collection, as well as reducing garbage collection. This is done by maintaining one bit reference counts *in pointers* rather than in objects.

The method is that any application pointer is identifiable as being either a *multiple application pointer* or a *unique application pointer* (**mappl** and **uappl**). Use of a **mappl** is synonymous with the conventional meaning of an application pointer, but use of a **uappl** has the additional meaning that "there are no other pointers to this cell". Thus, if a **uappl** pointer is being discarded then the cell that it designates is known to be free and can be given back to the memory allocator (i.e. placed at the head of the freechain).

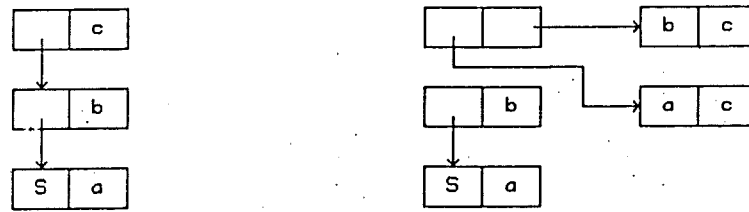
If an object is being copied, there is no need to access it under this scheme. If the copied object is a **uappl** then it must be changed to a **mappl**, but this is a

very simple operation and does not take very long. It is not useful to have more than two states, because the increment operation would have to change all other pointers to the object and that is clearly not possible.

This form of reference count works particularly well with combinator reduction because new cells are frequently needed just as old ones are freed, so that overheads associated with freechain maintenance are reduced. As an example, consider the S combinator which performs the following action:

$$S a b c = a c (b c)$$

In terms of graph structures this is represented by the following transformation:



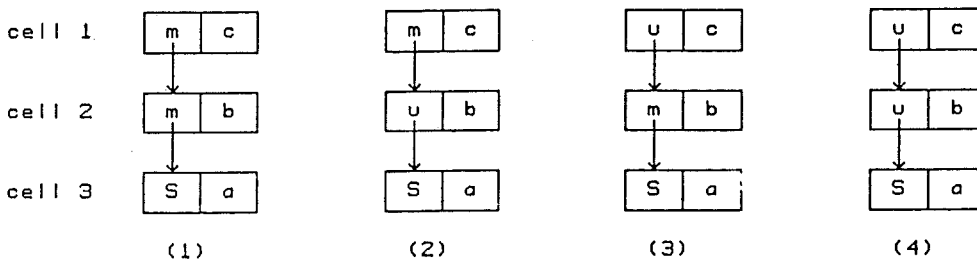
S a b c is transformed into a c (b c)
two new cells are generated

Conventional reduction of the S combinator
(pointer reversal has been omitted for clarity)

Note that the original cells have to be left unaltered, in case there are other pointers to the middle of this structure. Two new cells have been generated, and the value c copied without being evaluated.

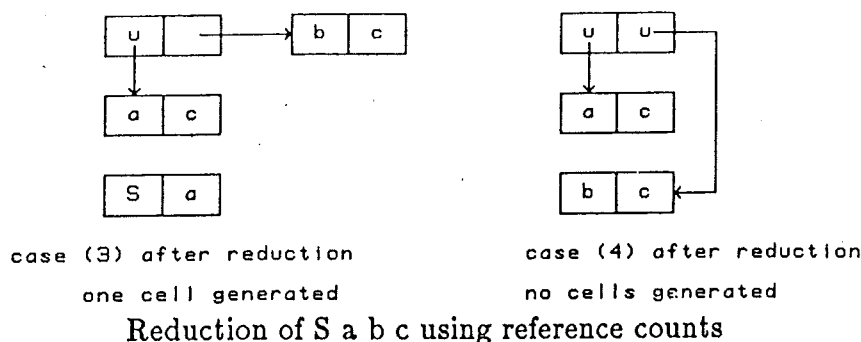
In terms of **uappls** and **mappls**, there are four forms in which this expression can appear, depending on the two pointers in it:

u → denotes a unique application (uappl) pointer
m → denotes a multiple application (mappl) pointer



Reference counts: the four different forms for S a b c

When cases 1 and 2 are encountered, the reduction method is unchanged. In case 3, cell 2 will not be accessible after the reduction and so can be reused. In case 4, cells 2 and 3 can both be reused, which is very convenient as we were just about to generate two new cells. By not giving the inaccessible cells back to the space allocator, but reusing them straight away, the reduction of this combinator will proceed faster. After the reduction, cases 3 and 4 will look like this:



Deciding which of these cases to use happens while picking up the arguments, and works very neatly. Although the amount of microcode for each combinator is increased because of the number of different cases to prepare for, in practice *most* reductions salvage some space in this way. The S combinator has an additional complication in that it duplicates its third argument, so if c is a **uappl** it must be changed to a **mappl**. Some other combinators (like plus) have free cells that they cannot immediately use, these are placed on the freechain.

The results of applying this technique have been spectacular—on average, about seventy percent of wasted cells are immediately reclaimed. Detailed timings were made for two small computations. The first is defined by

```
tak x y z = z, y >= x          ; If y >= x Then z Else ...
           = tak (tak (x - 1) y z)
             (tak (y - 1) z x)
             (tak (z - 1) x y)
```

This test is called Takeuchi's function and has been used on a number of Lisp systems as a crude benchmark. The second computation is the sorting of 2000 numbers, using a very simple sort program that divides its input into two lists of equal length, recursively sorts the two sublists and then does a binary merge. The results are as follows:

test name	free memory (K nodes)	microcode version 1		microcode version 2		microcode version 3	
		secs	GC's	secs	GC's	secs	GC's
tak	125	16	12	15	5	13	4
	105	16	13	15	6	14	6
	75	18	14	17	9	14	7
	45	29	32	20	15	19	12
	15	82	96	38	41	36	38
sort	125	23	18	21	6	17	5
	105	25	22	21	7	18	6
	75	32	32	23	9	19	8
	45	48	58	27	17	23	15

Speed improvements for reference counts and recursion removal

The table shows measurements for three different versions of microcode, running with varying amounts of free memory. The machine has a total of about 130,000 cells of heap. For each run the total time t in seconds and the number g of garbage collections was recorded. Version 1 microcode did no reference counting and used recursion to evaluate strict functions, version 2 microcode did reference counting, version 3 also used pointer reversal instead of recursion for strict functions. As the figures show, performance degradation with memory full is dramatically reduced when using reference counts.

At the moment this technique has only been implemented on application pairs, but there is no reason why it should not be used on data pairs too. Indeed, it could be useful in any heap storage system.

Equating Equal Structures

One frequently used combinator compares two data structures recursively for equality and returns a boolean value, like the Lisp function `EQUAL`. The implementation on `SKIM` has a secret side effect, that if the objects are equal their space is commoned up. This not only saves space, it speeds up future comparisons of the same objects.

This would not be possible in a Lisp system, because future use of `RPLACA` and `RPLACD` would cause chaos. There is also the more subtle possibility that future use of `EQ` would generate a misleading reply. Optimisations like this can be performed safely on a functional machine because there are no side effects to consider, and because there are no functions whose semantics are based on the conventional view of computer memory. It would be perfectly legitimate to common up equal objects or expressions wherever they are found, and this has a number of possibilities for further optimisation.

In idle time there is no reason why the machine should not scan over the heap and common up any equal structures that it finds. Although no measurements have been done, we estimate that combinator expressions could be reduced to about half their previous size by commoning up equal parts. This is because of the very small number of distinct combinators. It would be very interesting to implement Hashcons [Goto 74] on `SKIM`, as many of the conventional arguments against it for Lisp systems do not apply, but this has not yet been done. The saving of space would provide a considerable performance boost, and structure comparison would always be instantaneous.

Another operation that could be performed in any idle time is the reduction of any combinators that have enough arguments to be reduced, and whose reduction would not increase the size of the resulting expression. This limitation, or perhaps a less severe one, is needed to prevent store being filled with unwanted expansions of expressions.

Compilation into Microcode

The lack of fast implementations for functional languages is frequently bemoaned, as being a serious stumbling block in the examination of large software systems written in a functional style. We have chosen to deal with this problem by constructing special hardware to implement existing techniques of evaluation (i.e. combinator reduction), but many other groups are attempting to develop new techniques for the evaluation of functional languages.

In particular, a great deal of work is being done on the design of sophisticated compilers targeted towards conventional machines, and theoretical and practical knowledge in this field is increasing all the time. Work in Cambridge [Fairbairn 82] paralleling the SKIM development has produced a compiler from a pure functional language into code for a 68000 microprocessor [Motorola 82]. This compiler's code is considerably faster than interpreted combinator reduction on a 68000, and it would be highly beneficial if SKIM could make use of advances of this kind.

Microcode is usually viewed as a small control program sitting in fast, expensive RAM: it is part of the main processor, as an engineer's implementation mechanism for complex hardware. Even on machines with microcode in RAM such as the DEC VAX-11 and the Xerox Dorado, most microcode is hand written by experts as part of systems implementation.

SKIM is better viewed as a machine with separate data and program memory, with the structure of each optimised for its intended use. The program memory is only microcode in as much as it is viewed as being static. It is implemented using cheap 150 nanosecond static RAMs organised into 40 bit words, with a 250 nanosecond cycle time. There is a 16 bit microcode address bus, so that the processor could support up to 64K*40 bit words of microcode. Using 8K*8 static RAMs, which should be cheap and common within the next year, this occupies one Extended Double Eurocard.

Thus there is no reason why SKIM microcode should not be generated by compiler, from a functional language. As the same language is used to generate combinators, compilation into microcode would not require functional programs to be rewritten, merely submitted to a different version of the compiler. This would allow the programmer to boost the performance of important or frequently used functions while retaining the flexibility of combinators for less critical ones.

Some small functions have been 'compiled by hand' for SKIM, which are nevertheless more complex operations than what is generally viewed as being a 'combinator'. The microcode instruction set was found to be no more difficult to work with than the machine code of a conventional machine. The functions compiled so far have included the recursive structure comparison EQUAL, and reverse and append for lists.

The results showed a speedup by a considerable factor over combinator evaluation:

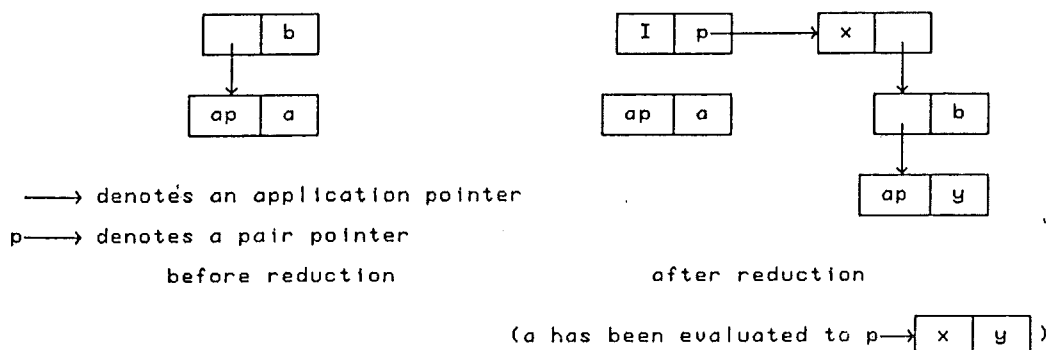
combinators		
length	7000 per second	
reverse	7000 per second	
append	7000 per second	
equal	1650 per second	
nfib benchmark	8000 per second	
hand compiled		
length	220000 per second	(*30 speedup)
reverse	120000 per second	(*18 speedup)
append	30000 per second	(*6 speedup)
equal	20000 per second	(*13 speedup)
tak	tak 18 12 6 in 1.5 seconds	
nfib	90000 per second	

Speed improvements for microcode compilation

The measurements are crudely expressed in terms of cells per second of action on a linear list. Reverse and equal are completely strict and compile into microcode loops, while append does not. Garbage collection time is included. A compiler's code would not be quite as fast as this, but should be close.

The best strategy for compilation appears to be to remove laziness wherever possible, as laziness means the generation and reduction of more graph structures. It is clear that some functions are more susceptible to this than others, and so while some functions (e.g. simple recursive numerical computations like factorial) would speed up far more than this, others (complex manipulation of higher order functions) will hardly speed up at all.

The best 'grain' at which to work appears to be that of supercombinators [Hughes 82]. For instance, the append combinator performs the following operation:



The append combinator

The argument a must be evaluated and examined. If it evaluates to a data pair then the head and tail halves are fetched and a new structure built as in (2).

Otherwise, the result is b. Three new cells are needed, although in practice the reference counting mechanism will collect up the two superfluous ones.

We do not yet have any practical experience of how large programs written in a functional language will perform when compiled, but (as an example) current compilation techniques [Mycroft 81, Wray 85] will not reduce the number of new cells used in the append example above. Even when compilers become common for functional languages, SKIM will still provide competitive performance.

Conclusions

Details have been given of various optimisations to Turner's combinator reduction method. Many of these methods are not at all obvious, and there may be many more similar ideas waiting to be discovered. They have resulted in considerable performance improvements for SKIM. Although such things are difficult to measure absolutely, SKIM appears to deliver a performance in solving common problems that is about one-quarter of that of a conventional compiled language on hardware of similar cost. The machine used for this comparison had a 68000 CPU running at 8-MHz, running BCPL and Algol68C as representative high level languages.

It is important to note that all of these improvements are transparent to the high level, functional programmer. They have been possible because the combinator model of computation is not tied to any one hardware scheme. By resisting the temptation to bend the high level programming model towards what is efficient to implement, as has happened to many Lisp and Prolog systems, the implementor keeps future options open as well as ensuring software compatibility. Although current implementations of functional languages may not yet rival conventional languages, future improvements in implementation expertise are likely to narrow this gap over the next few years.

Director Strings

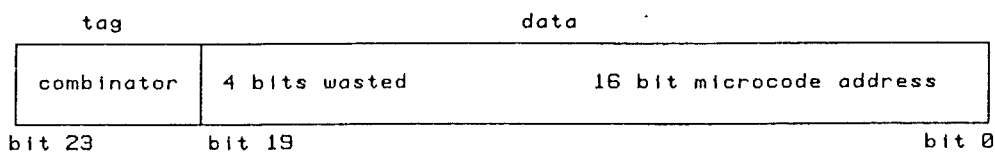
This chapter demonstrates more of the wide range of techniques that is available to the implementor of functional languages. Conventional languages have been around for sufficiently long that implementation techniques for them are fairly well understood, but the implementation of functional languages is a relatively recent pastime and it still contains surprises. The main example in this chapter is based on an unusual interaction between the reference count scheme described in the previous chapter, and the "director string" idea first presented in [Kennaway 82]. Based on these a number of further new ideas are suggested and connections observed between Turner's combinator scheme and a number of other implementation methods.

Combinator Graph Reduction

Turner's combinator reduction scheme has been in practical use for some time now, it appears to be the best (simple) way of implementing a normal order lambda reducer. Much of its popularity is due to the fact that it is easy to understand and easy to implement, but there are a number of reasons for suspecting that a more sophisticated implementation method will yield better performance.

The reasoning behind this thought is concerned with the use of main memory. The reduction process seems to require a large number of accesses to main memory, and the reducer is, on the whole, limited by main memory bandwidth. The combinator representation of programs is somewhat wasteful of space: in particular, using a whole word for the representation of a combinator wastes most of that word (on SKIM there are less than fifty combinators in all, including arithmetic primitives and so on). The reduction of a single combinator is a very small computational step, and it involves quite a large amount of data movement. The result is a large volume of data traffic in the CPU, and between the CPU and main memory, much of which is unnecessary.

This argument is easiest to understand in terms of machine words. A word on SKIM consists of twenty-four bits, and a word representing a combinator is of the form



The tag indicates what kind of value it is: in this case, a combinator. The other twenty bits hold the microcode address of the code for this combinator. On SKIM there are about fifty distinct combinators and so only about six bits of the

data are really needed. As every instruction obeyed by the machine is in this form, a large amount of space and of main memory bandwidth is clearly wasted.

The reference count scheme is helpful not only because it retrieves memory cells, but because it reduces the number of memory references needed by the reduction process and so makes better use of memory bandwidth. Under it pointers to application cells are of two types, a *multiple application pointer* and a *unique application pointer*. The first of these is equivalent to the usual idea of a pointer in a computer data structure, the second has the additional implied meaning that "there are no other pointers to this cell in the entire machine".

Director Strings

A visual examination of typical combinators produced by the compilation of functional programs reveals a regularity which is ill exploited by Turner's combinator graph reduction scheme. This concerns large regular strings of combinators, of the form

$$(X' (X' (X' (X' X)))) \quad [1]$$

and

$$(X' (X' (X' (X' fn)))) \quad [2]$$

where

X is one of B, C or S

X' is one of B', C' or S' (not necessarily all the same)

fn is an expression with no free variables.

The reason for the frequent occurrence of these patterns is quite easy to understand, if the program is visualised as a lambda expression. The first occurs from the application of a large function *EXPR1* to a large argument *EXPR2*, both of which have many free variables.

EXPR1 EXPR2

For each variable that is to be abstracted from this function a B, C or S will appear, depending on which of the two needs the argument:

S if they both want it

C if only *EXPR1* wants it

B if only *EXPR2* wants it

After the first such combinator the rest build up as X' because of the presence of the first. The way in which this happens was described in more detail in chapter two.

The second of these forms (labelled [2]) comes from a similar phenomenon: a simple dyadic function with two complex arguments:

fn *EXPR1 EXPR2*

The function is frequently something named at the "top level" of the whole program, and thus (on SKIM anyway) its value will be instantiated by a table lookup when the system is loaded, rather than by combinators at run time. Frequently occurring functions like plus, cons and append make this very common.

Both of these patterns have a strong intuitive feel as the flow of a number of variables past a node in the graph. Each variable being thus directed can be represented by two bits: one says whether the left part of the node refers to next variable, the other says whether the right arm does.

The name "director strings" [Kennaway 82] has been suggested to describe these sequences of combinators. A representation for director strings is now introduced which is not only more compact than the corresponding combinator graph, but which can also be reduced more quickly.

In order to describe this representation two new combinators are introduced called D and D'. Both of these take as their first argument a bit pattern, which describes the flow of arguments that they control. Their reduction rules are as follows:

D (pattern++00)	a b c	=	D pattern a b
D (pattern++01)	a b c	=	D pattern a (b c)
D (pattern++10)	a b c	=	D pattern (a c) b
D (pattern++11)	a b c	=	D pattern (a c) (b c)
D nullpattern	a b c	=	a b c

Each of the above clauses describes the behaviour of D for some particular bit pattern, the notation being adapted freely from Miranda. Together they form a complete definition of D. D' is similar:

D' (pattern++00)	fn a b c	=	D' pattern fn a b
D' (pattern++01)	fn a b c	=	D' pattern fn a (b c)
D' (pattern++10)	fn a b c	=	D' pattern fn (a c) b
D' (pattern++11)	fn a b c	=	D' pattern fn (a c) (b c)
D' nullpattern	fn a b c	=	fn a b c

A D combinator will appear when transforming a lambda expression to a form with no bound variables, at any expression of the form

LARGEEXPR1 LARGEEXPR2.

For each bound variable that is to be abstracted from these 01, 10 or 11 will be appended to the D combinator's bit pattern:

11 if they both want it
 01 if only EXPR2 wants it
 10 if only EXPR1 wants it

(compare this with the earlier statement about combinator abstraction).

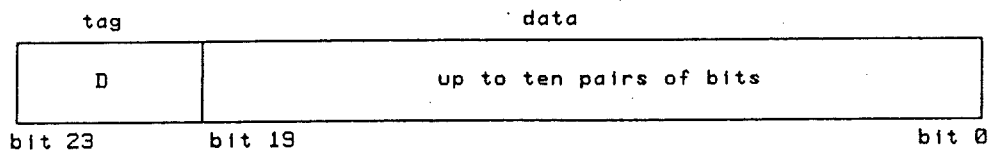
Thus it can be seen that

$$\begin{aligned}
 B &= D \ 01 \\
 C &= D \ 10 \\
 S &= D \ 11 \\
 B' B &= D \ 0101 \\
 S' C &= D \ 1110 \\
 B' (S' C) &= D \ 011110 \\
 X' (X' (X' (X' X))) &= D \ \text{xxxxxxxxxx}
 \end{aligned}$$

Similar rules apply to D' , which is related to the second common form (described above) in exactly the same way as D is related to the first.

$$\begin{aligned}
 B' \text{ fn} &= D' \ 01 \text{ fn} \\
 S' (C' \text{ fn}) &= D' \ 1110 \text{ fn} \\
 X' (X' (X' (X' \text{ fn}))) &= D' \ \text{xxxxxxxx} \text{ fn}
 \end{aligned}$$

These bit patterns are not values that the user can manipulate directly, and because of this an extremely compact representation of D and D' can be used.



The diagram shows how D combinators are represented on SKIM, replacing a chain of up to ten conventional combinators. D' combinators are represented in the same way, using a different tag value. The result of this is not just a saving of space, but a considerable boost to performance.

Example

The simplest way of showing why the D representation is preferable to a combinator string is to work through a simple example, which demonstrates how much less work the reduction of the D form requires. The amount of work saved is considerably more than the saving implied by the reduction in program size, because the D combinator has distinct advantages in the evaluator: especially when combined with the reference count scheme.

The following reduction will be used as an example:

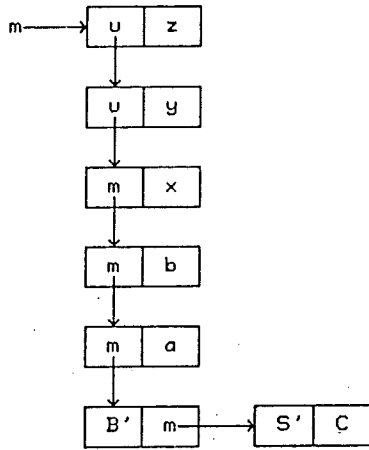
$$\begin{aligned}
 &B' (S' C) a b x y z \\
 &= S' C a (b x) y z \\
 &= C (a y) (b x y) z \\
 &= a y z (b x y)
 \end{aligned}$$

This should be visualised as deriving from the expression

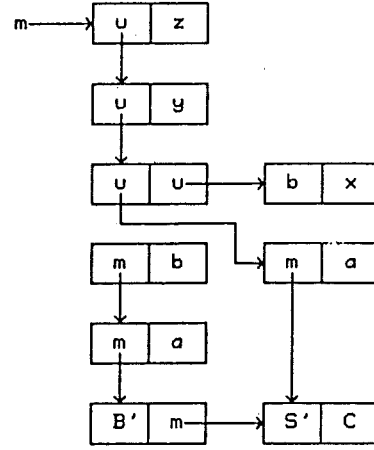
a b

with three free variables x , y and z : a contains references to y and z , b to x and y . The initial transformation of the lambda expression causes the combinator string to appear. The reduction listed above would appear in terms of graphs as follows:

$m \longrightarrow$ indicates a multiple application pointer
 $u \longrightarrow$ indicates a unique application pointer



(1) Initial setup



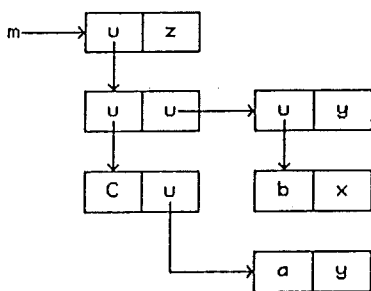
(2) after B' reduction

The diagram on the left shows the initial graph. The main body of the function is constructed from multiple application pointers, indicating that this is a long-lived memory construct. The arguments x , y and z have been supplied to this using unique application pointers, indicating that no intermediate results in the forthcoming calculation will be reused.

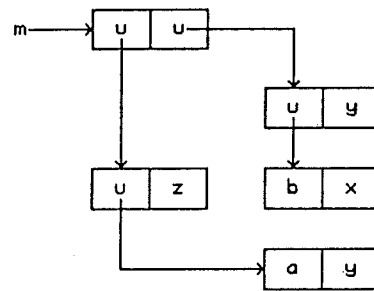
After the B' reduction the graph is as shown on the right. Two new heap cells have been generated, and a careful following of all the arrows will show that the value represented by a pointer to the top cell in the diagram is

$S' C a (b x) y z$

as expected.



(3) after S' reduction



(4) after C reduction

Diagram 3 shows the graph again after the reduction of the S' , except that the four cells at the bottom representing

$B' (S' C) a b$

are not shown as they play no further part in this calculation. Two cells have been reclaimed as they are no longer accessible (the reference count information reveals this) and immediately reused. In addition, a third new cell has been generated.

Diagram 4 shows the final result, after the reduction of the C combinator. Another two cells have been freed, of which one is immediately reused and the other given back to the storage allocator.

It is hard to show in these diagrams exactly how much work is being done to perform each reduction, and this is important as it is what the new scheme attempts to cut down. Timings for SKIM would not be very meaningful as they might be distorted by quirks of this particular architecture. A better approach is to count the main memory cycles needed by a highly optimised combinator reduction machine. Examination of the diagrams suggests the following (approximate) figures:

- 6 cells read to get to the B' combinator
- 3 cells written in the B' rewrite
- 1 cell read to get to the S' combinator
- 4 cells written in the S' rewrite
- 3 cells written in the C rewrite
- total: 17 memory cycles

This figure assumes that all nodes on the chain of pointers from the root to the combinator are cached in the processor, and that only one write can happen at a time. The reading or writing of a whole cell is assumed to take one cycle.

Here is the same operation using the D combinator.

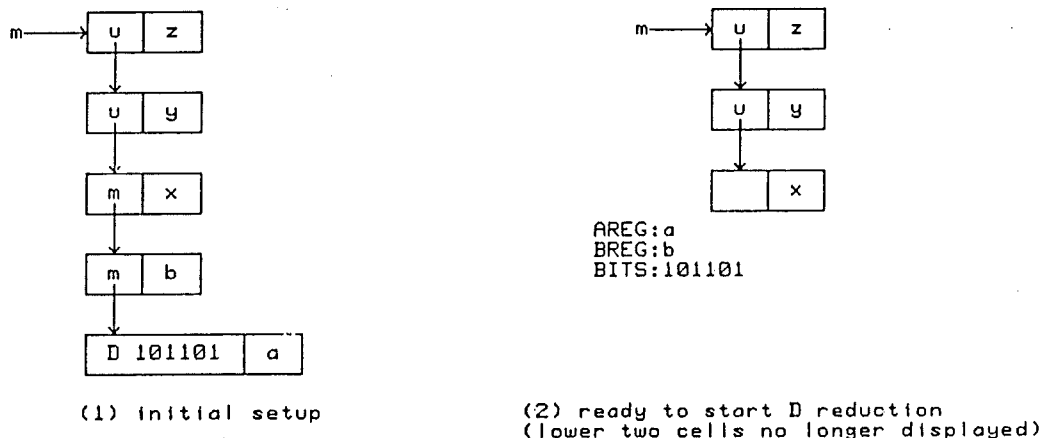
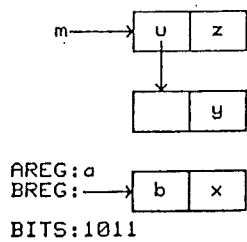
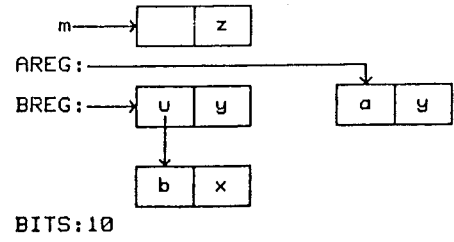


Diagram 1 shows the initial graph: the D combinator and its bit pattern fit into a single machine word. The rewrite for a D combinator takes place as a sequence of steps, and during this process three registers are necessary in the reducing machine to hold intermediate values. These are shown in diagram 2: the register BITS holds the current value of the bit pattern, while AREG and BREG hold the current values of a and b respectively. As the D reduction proceeds, AREG and BREG will be applied to arguments as dictated by the bit pattern.

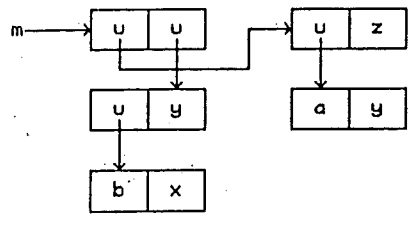


(3) After 01 reduction



(4) After 11 reduction

Diagram 3 shows the situation after one step. The x argument is needed by b. The cell that held it has been freed and reused. The y argument is needed by both a and b, so a new cell must be used in addition to reusing the one that contained y. Diagram 4 shows the graph after this has occurred.



(5) Final configuration - a y z (b x y)

The final result is the same as when using Turner's combinators. Counting the number of memory cycles reveals

- 5 reads to get to the D combinator
- 1 write to perform the 01 update
- 2 writes to perform the 11 update
- 2 writes to perform the 10 update, and write out the result
- total: 10 memory cycles

If multiple application pointers occur in the argument structure of a D combinator the procedure is somewhat more involved than this. In order to preserve full laziness, the half-used D combinator must be written back into graph memory. This complication has been ignored here. By far the most common form of D reduction is the straightforward one outlined above.

Slight irregularities

It is worth digressing at this point concerning a couple of slight irregularities in the method, which may confuse if left unexplained but which are in fact of little consequence.

The first occurs in Turner's combinator set, but is only revealed clearly by the bitwise encoding of director strings. It has been glossed over in the earlier description, and it concerns the lack of more K-like combinators. Viewing B, C and S as being described by two bits indicates the existence of a fourth combination, corresponding to the bit pattern 00.

This combination might be called M for "missing", and Turner's combinator set would seem more regular if it included M and M', defined by

$$\begin{aligned} M a b c &= a b \\ M' a b c d &= a b c. \end{aligned}$$

The justification for M's omission is that K combinators are in fact quite rare. They only occur if a function in the source language ignores an argument. In a language that includes argument pattern matching, however, I suspect that this occurrence is not as rare as it might at first appear. Also, when it does occur the lack of M and M' can be quite damaging to the code quality. A statistical examination of some large functional programs to determine this would be quite interesting, although sadly no such study exists. Despite these arguments I suspect that the inclusion or omission of M is not in fact all that important.

The second is that D's bit pattern has a fixed maximum size, determined by the width of the computer's datapaths. Fortunately this maximum is quite large, on SKIM a single D combinator can handle up to 10 variables and a node with more than that number of variables passing it is quite rare. When it does occur, Ds can easily be cascaded together to form a bigger node capable of handling any number of instantiated variables. The result is slightly less efficient, but does not occur very often.

Discussion

SKIM was originally microcoded to implement Turner's combinator reduction algorithm, and the D combinator is not viewed as a replacement for this but as an optimisation at the lowest implementation level. It should not be viewed as a new formal model of computation, but as a demonstration that estimating the usefulness of an abstract implementation scheme is a tricky business. Numerous authors [Kennaway 82, Abdali 76, Noshita 84, Noshita 84a, Cousineau 85] have suggested schemes for representing lambda or combinatory expressions, and proved a variety of theorems concerning translation speed and worst-case static code size. However, to the implementor all of these arguments are dwarfed in importance by the speed with which the various representations can be reduced. Static size is likely to be a pointer to this, but only a pointer. An abstract design for a computing machine, and some measure of how much 'work' gets done in the reduction process, is necessary in order to perform a detailed evaluation of how useful one particular scheme is to an implementor. Once the general principle had been established, the precise design of the D combinator implemented on SKIM was decided entirely by the existing microcode architecture.

This line of thought suggests that there might well be other forms of representation that would produce even greater benefits than D or D'. After all, these two were chosen simply by a visual examination of compiled combinators, and some tenuous reasoning. Two examples of related schemes will be produced

to illustrate this idea. Each represents an infinite family of combinators, of which only a finite number would be implemented in practice.

The first such example is a three-way director string called D3. D3 takes as its first argument a bit pattern that is divided into groups of three bits, and is defined by

```
D3 (bits++000) a b c x = D3 bits a b c
D3 (bits++001) a b c x = D3 bits a b (c x)
D3 (bits++010) a b c x = D3 bits a (b x) c
D3 (bits++011) a b c x = D3 bits a (b x) (c x)
D3 (bits++100) a b c x = D3 bits (a x) b c
D3 (bits++101) a b c x = D3 bits (a x) b (c x)
D3 (bits++110) a b c x = D3 bits (a x) (b x) c
D3 (bits++111) a b c x = D3 bits (a x) (b x) (c x)
D3 nobits a b c = a b c
```

The similarity to D should be obvious, as well as the generalisation to even bigger versions. The reason that D3 in particular might be useful is that conditional clauses occur very frequently in functional programs. When stripped of its syntax, a conditional clause looks like this:

```
IFEXPR THENEXPR ELSEEXPR
```

Each of the three parts may have several free variables, and IFEXPR evaluates to either TRUE or FALSE, where

```
TRUE a b = a
FALSE a b = b
```

Using D, this form would be viewed as

```
(IFEXPR THENEXPR) ELSEEXPR
```

and two D combinators would be generated, with bit patterns of equal length. These could be replaced by a single D3 string, with a corresponding improvement in performance.

The second example is called G for "general" and can mimic the operation of any combinator within certain size constraints. It would be extremely useful near the leaves of a combinator expression, in contrast to D whose main use is near the centre.

G takes as arguments a bit pattern, and four other arguments called a, b, c and d. The bit pattern is interpreted in triplets, with the following meanings assigned:

000 - cons
 100 - push argument a
 101 - push argument b
 110 - push argument c
 111 - push argument d

The bit pattern can be used to express, using reverse polish notation, the construction of any expression involving just a, b, c and d. The complexity of the result would be limited by the maximum allowable size for the bit pattern. For instance, suppose the the following expression were required:

$(\lambda a. (\lambda b. (\lambda c. (\lambda d. d c b a))))$

This function is typical of the sort of form that is found near the leaves of combinator trees. Its combinator form is quite embarassingly large:

$C' (C' C) (C' C (C I))$

Use of D still leaves this as two short director strings and a (C I). Using G, however, the whole thing can be represented as

G pusha++pushb++pushc++pushd

This represents a G combinator with a single bit pattern, containing 12 bits. G requires a number of cycles in which to execute, but these are cycles of some very small finite state machine within the CPU, and could be made to go very fast. Different versions of G could handle three and five variable combinators. A maximum bit pattern length of 24 bits is probably quite adequate.

The design of G is strongly remeniscent of something even more familiar than combinator reduction: the instruction set of a conventional computer! With little effort even more elaborate versions can be constructed, with such extra features as:

primitives (e.g. integer and list manipulation) built in
 common constants (integers, TRUE and FALSE)
 encoded in the bit pattern
 transfers of control within the bit pattern

Before long we'll start reinventing addressing modes, subroutines and the like, with every step being justified by a saving in main memory bandwidth. The bit pattern size limit could be breached by suggesting more elaborate store management. More and more elaborate transformation algorithms will be needed in order to make use of such facilites, but these can be viewed (if we so desire) as transformations to the representation of the lambda expression. The machine that evaluates the result will be concerned less with graph reduction, and more (as on a conventional computer) with instruction stream management.

This presents a whole spectrum of possibilities to the machine designer. At one extreme, a "pure" graph reducing machine implements a scheme like Turner's (i.e. based on a finite collection of basic combinators) without the use of any clever encoding tricks. The most important specialisations of such a machine would be an appropriately designed memory with tags, garbage collection bits and the like. SKIM with its initial microcode represents an early attempt at this. A higher performance version would concentrate on the use of clever pipelining and caching to maximise total bandwidth, relying on the inherent simplicity of the operations that it is implementing to allow an extremely fast combinator reduction rate. NORMA [Richards 85, chapter three] contains features that move in this direction, and [Clarke 84] gives a number of suggestions concerning more sophisticated optimisations that a pure graph reducing machine might employ in order to improve performance.

The opposite end of this spectrum is represented by conventional processors, which can be viewed as graph reducers in the following way. The graph is converted into a linear sequence of instructions that are adjacent in physical store, with arcs of the graph represented by branch and call instructions. Any portions of the graph that are not static must be represented using the same techniques as a dedicated reducing machine, but with somewhat lower performance than a machine dedicated to this mode of operation. The instruction stream machine has considerable advantages over the pure graph reducer when executing sequential instructions, but a disadvantage in graph operations. Because early computers were of this type, conventional computer languages have been carefully designed to ensure that programs are representable by a static graph. Unfortunately (even when using sophisticated compilation techniques) functional programs require substantial amounts of dynamic graph manipulation.

The material presented in this chapter suggests a way of combining these two extremes in order to construct an extremely high performance machine for functional language implementation. By using a graph to represent the structure of a program, but using a "ministream" form of instruction (based on G described above) instead of simple combinators as the leaves of the graph, a memory and bus structure designed for optimised graph reduction could be combined with many of the advantages of an instruction stream machine. The ministream would not contain any transfer of control instructions, and all instructions would probably be of a constant size, resulting in extremely simple decoding hardware.

Conclusions

Evaluating graph reduction schemes is tricky because the best representation is not always obvious: Turner's scheme, for instance, is so abstract that a great deal of flexibility is available to the implementor. When evaluating the performance of new schemes lower level details are essential: for instance, a useful comparison of lambda reduction with combinator reduction is really futile until more is known about the precise strategies being used.

D and D' were implemented on SKIM, and caused a slight performance improvement. In order to achieve their full potential they need specially designed hardware. Given that, I estimate that they would cause a ten percent speed increase. Implementing G would be slightly more complex, but could cause a further twenty percent improvement.

The one bit reference count has some surprising uses, in some ways saving store seems to be the least of its benefits! It provides the reducer with so much extra information for so little cost, that it seems an essential element of any future graph reduction system. GRIP, a parallel graph reduction machine being designed at University College London, seems likely to employ this scheme [Peyton Jones 85] and the feasibility of its use in NORMA [Richards 85] is currently being investigated.

Until now there have been two approaches to the implementation of functional languages: either custom hardware must be constructed to perform graph reduction effectively, or compilation techniques developed to avoid graph reduction as much as possible. This chapter indicates that a high performance functional language computer is likely to use a combination of these techniques, and that future research on pure graph reducing machines and on compilation will both contribute to such a machine. The use of simple bit patterns like D and G may turn out to be more effective for implementing functional languages than a general purpose instruction set, for they keep the emphasis of the design firmly on graph reduction.

Problems

This chapter describes some problems that were discovered when using SKIM for application work, and discusses their impact on implementation. The results are quite worrying, in that current solutions are ad hoc and incomplete: pathological cases still remain whereby a seemingly innocent program can consume unreasonable amounts of time and space to execute. Most of these problems only become important when computations are very long lived, or contain data structures that are not small compared to the size of the computer's memory.

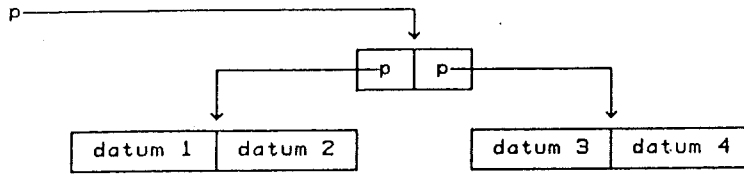
This chapter contains one or two very small functional programs written in the language Miranda. Precise knowledge of this language is not really necessary, but if required more information can be found in appendix B.

Large Structures

Programming in a functional language with normal order semantics changes the way that a programmer thinks about the values that he is manipulating. The tendency is to think less about the order in which things are done, and more about the values being produced. Calculations not necessary for the production of the answer will not be performed, and the order in which the calculations are done at run time is not important. In most cases this mode of thought simplifies programming tasks, and makes it easier to write correct programs. Unfortunately, it can let the programmer down when large structures are involved.

In conventional languages the programmer is directly responsible for every item of store on the heap. It is quite easy to estimate memory requirements, and to discover which parts of a program are using most store. In functional languages this is not the case: the construction of closures goes on without the programmer being aware of it, with the consequence that the amount of store consumed by the representation of a value at run time may be entirely unrelated to the value itself.

As an example of this, consider the implementation of a data structure consisting of a binary tree of nodes. The programmer thinks of the structure as looking like this:

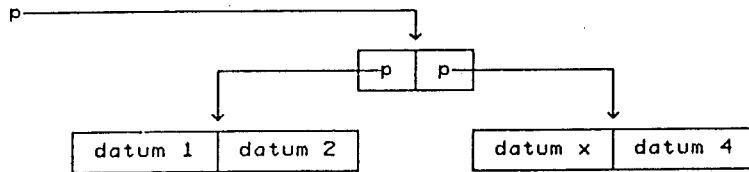


p → is a data pair pointer

A tree structured value

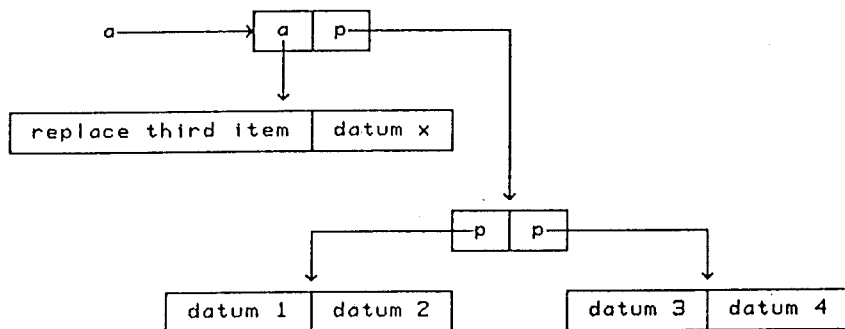
This example assumes that the machine's heap is divided into Lisp-like nodes or cells. There are two kinds of cells, data pairs and applications (i.e. closures). A data pair is similar to a Lisp cons cell, while an application pointer denotes a value that is a function or is not yet fully evaluated.

Now suppose that the programmer wishes to discard this value and use a new one, like this one except that the value in the third field (currently datum 3) is replaced by datum x. This sort of requirement occurs frequently, and corresponds very closely to the idea of updating a field of a record in a conventional program. The programmer's conceptual model of the value is now



The programmer's vision of the updated structure

but in fact a closure is formed until the structure is actually touched, so that the following structure exists in the store of the machine:

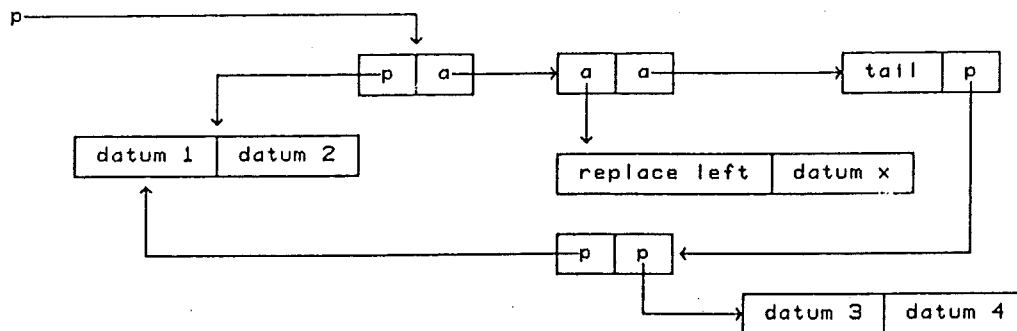


a → is an application pointer

The actual structure representing the value

The value `replace third item` is a function that takes two arguments: a replacement value (in this case `datum x`) and a structure on which the function is to act. This structure will only be evaluated when some part it is accessed, *and then only partially*: if the field that is next accessed of this structure is not the

newly updated one, some amount of computation will remain unperformed. Suppose that the value of datum 1 is required. Some access function will extract datum 1, causing the structure as a whole to become:



The actual representation of the structure after a read access to datum 1

The left part of the tree is fully evaluated, because datum 1 has been accessed recently. However, the right part is still represented by a closure. When the function that accessed datum 1 acted on the tree replace third item did the minimum necessary amount of work, and formed a closure for the rest in case the other work is never needed. This second closure is represented by the function replace left in the diagram.

Note that the programmer has forgotten some time ago about the old value datum 3 and has even accessed the structure, yet the store used is somewhat more than he imagines. There are two reasons for this. One is that datum 3 is still being referenced, and so the store for it cannot be reclaimed. Another is the space taken up by the closure itself. If the third place in this structure (currently occupied by datum x) is updated with further values but remains unaccessed, the total space occupied by the representation of this value grows without bound as the computation proceeds.

This effect is sometimes caused by what is known as redundant laziness. By sticking rigidly to normal order semantics the evaluating system sometimes avoids doing operations that in practice would free store. It isn't possible to detect automatically when laziness is unwanted; sometimes, as in the example described above, it requires changes to the definition of a function that allow it to be slightly more strict on its arguments than is actually necessary. Whether doing this would help in any particular instance is often a matter of careful judgement, so it seems improbable that a fully automatic system could ever be satisfactory in performing this operation.

Functional programmers sometimes combat effects such as these by writing code that deliberately evaluates some values early. For instance, consider the function

```
touch a b = b, a = a ; If a = a Then b
```

This function causes its first argument to be evaluated, and then returns its second. The appearance of any function or construction of this form is a clear

sign that the programmer has had to think carefully about the order of evaluation of the various parts of his program. A more detailed discussion of this function appears in chapter eight, where it reappears (under the name `strictify`) concerning input and output.

Another possible way of solving this problem is to increase the complexity of the run-time system. For instance, the graph reducer could be instructed to perform any reductions that reduced the size of structures in store. Unfortunately this is extremely difficult to implement because eventual shrinkage might require temporary growth: I know of no solution to this problem, or even a satisfactory approximation to a solution.

The phenomenon is particularly dangerous when it occurs with very large structured values such as parse trees, filing systems and databases. Because the programmer does not have direct control over the precise run-time representations of values like these, such structures can easily occupy considerably more store than an examination of the value itself would suggest. Consider the implementation of a program that manipulates a database, with a volume of raw data so large that it occupies most of the computer's memory. The considerations produced here suggest that such a database cannot safely be manipulated as a first-class value by a functional program, because the programmer does not have precise control over its representation.

In that case, how should such a program be written? My own view is that the solution of this problem requires the message passing mechanism described in chapter nine. If a data structure is so large that its accidental replication would be disastrous, then it should be represented not as a first class data object, but as a device in a message passing system.

Transient Store

In functional languages the use of loops is discarded in favour of recursion or higher order functions. This practice results in more powerful and elegant program control constructs, but can often result in transient store usage that is proportional to the number of iterations. Defeating this needs a combination of careful coding and clever compilation. The latter of these is not so damaging, but the former often results in a natural coding style being unreasonably inefficient. Also, quite trifling changes to the way in which a function is coded can have an unexpectedly large effect in this area.

For instance, here are four different ways of coding a function that sums the integers from 1 to n . Each represents a style of coding that is suitable for a different set of circumstances, and although this function itself is scarcely interesting all of the styles of iteration shown here can turn up in large functional programs.

style 1: simple recursion

```
sum 0 = 0
sum n = n + sum (n - 1)
```

This is probably the most natural style for simple loops. In any implementation, transient store usage is proportional to n . The value

```
(n +
 (n - 1) +
 ((n - 1) - 1) +
 (((n - 1) - 1) - 1) +
 ... +
 (...(((n - 1) - 1) - 1) ... )
```

will get constructed, and then evaluated. At first sight this looks like quadratic growth, but it's linear because of structure sharing, so that all occurrences of $(n - 1)$ are shared.

It is also interesting to note that transient *stack* usage is also proportional to n , so on a machine with stack caching this style of iteration will defeat the cache rather badly.

style 2: accumulator argument

```
sum n = sum1 0 n
sum1 a 0 = a
sum1 a n = sum1 (a + n) (n - 1)
```

This represents the use of an "accumulator" argument, which results in the program being *tail recursive*. This trick is common amongst Lisp programmers, who know that an applicative order evaluator can perform the computation in constant space. A good Lisp compiler should generate from this approximately the same code as a conventional language would generate from an equivalent loop.

On a graph reducer, however, the transient store used is still proportional to n , because the value a is constructed as

```
(((((0 + n) + (n - 1)) + ((n - 1) - 1))) + ... + (...(n - 1)...))
```

and then evaluated.

The maximum stack usage depends on the order in which the $+$ operator evaluates its arguments. If it evaluates the left one first then the maximum stack usage is small, if it evaluates the right one first then stack usage is proportional to n .

This can be changed by clever compilation. For instance, the analysis published in [Mycroft 81] will reveal to a compiler that the value of `a` in the body of `sum1` can be evaluated earlier than it would be by a simple normal order evaluator, without affecting the semantics of `sum1`. This sort of analysis is included in the Ponder compiler [Fairbairn 84] and recent versions of the ALFL compiler [Hudak 85a]. Both are languages with normal order semantics, yet these implementations can generate the same code as a Lisp compiler from this particular program.

style 3: exploiting parallelism

```
sum n = sum1 1 n
sum1 x x = x
sum1 from to = sum1 from split + sum1 (split + 1) to
              WHERE split = (from + to) / 2
```

This program attempts to find the maximum degree of parallelism in the problem, and might be expected to run faster than the others on a machine with several processors. On any sequential reducer it uses space that is proportional to \log of n . On a parallel reducer it would use space proportional to n .

In circumstances where sequential computers can have unexpectedly large transient store requirements the use of a parallel computer provides opportunities for spectacular disasters. For instance, consider a list of integers to which the operation `sum` is to be applied. The function

```
listsum x = map sum x
```

which achieves this on a sequential computer only uses transient store proportional to the length of the list `x` plus the value of its greatest element. By cleverly (!) attempting to exploit parallelism, however, this requirement can become proportional to the sum of the lengths of all the lists.

style 4: function combination

```
sum = fold plus 0 . range 1
```

This style represents the translation into Miranda of the sort of programming style recommended in [Backus 78], in that arguments are not named, recursion is not used, and results are achieved through combining higher order functions. It works by constructing the list `[1, 2, ..., n]` (this is what `range 1` achieves) and then adding up all the elements in the list. The infix dot denotes function composition. Backus suggests that this sort of program is easier to understand and reason about than a recursive one, and in some places I agree with him.

This sort of choice is made by individual programmers in particular situations, and such choices should be made on the grounds of clarity of expression rather

than performance. This particular example appears to use space proportional to n whatever compilation techniques are used: this surprises most people, who cheerfully assume that lazy evaluation will allow the expression to be evaluated in constant space. It defeats the optimisations proposed by Mycroft, but such techniques are currently a research topic in their own right [Hudak 85a, Wray 85]: compilation techniques could soon improve to the extent that this program would be turned into exactly the same code as the program using an accumulator argument.

Another source of unexpected transient store usage is in the use of lazy list filters that go wrong, for instance consider the following program that computes the average of a list of integers:

```
av l = sumof l / length l
```

Even assuming that `sumof` has been coded carefully to avoid the problems described above, this program will force the whole of `l` to exist at one time. There is no reason why the function should need to do this, and recoding can change the situation:

```
av l = av1 0 0 l
av1 sum length [] = sum / length
av1 sum length (x:y) = av1 (sum + x) (length + 1) y
```

Not only is the first program a lot clearer, but the problem may not occur to the programmer: intuitively, finding the average of a list of integers *shouldn't* require the whole list to exist. An interesting approach to this type of problem is demonstrated in [Hughes 84], which points out a number of examples where parallelism and synchronisation constructs can reduce transient store usage.

Any situation where functional programming causes a change in the complexity of an algorithm is a cause for concern. In the example I have chosen above this may not be very important, but some of these results are counter-intuitive, and when buried in a much larger and more complex problem they can result in a situation whereby it is not at all obvious how much space the program will use. In any program where some iteration is performed a great many times (i.e. not small compared to the number of cells in the machine's memory) the programmer will have to think carefully about this sort of thing.

One area of research that could well provide solutions to some of these problems is the program transformation approach suggested in [Darlington 81a]. Darlington suggests that program development should proceed by first producing a correct program, without regard for run-time behaviour, and then applying correctness-preserving transformations to it that improve the program's performance. He suggests that functional programming is a particularly appropriate environment for this approach and that writing a function as a set of recursion equations, as in Miranda, yields a natural source of transformation rules. It would be extremely useful, for instance, if the first program in this section could be automatically transformed into the second. The same is also

true of the averaging program.

Unfortunately it is still far from clear how the use of such techniques should be presented to the programmer. There are also problems concerning what transformations would be helpful. Ideally the computer itself should have some idea of what constitutes a beneficial transformation, but this seems quite hard to formulate.

Laziness

The lazy evaluation philosophy is that any structure that is evaluated will get overwritten with its evaluated form, so that recalculation is avoided. Laziness is purely a performance issue: it should not be confused with normal order semantics, which can be achieved without laziness. Unfortunately laziness can be undesirable if the evaluated form is bigger than the unevaluated one. A good example of this is the following function, which generates the *n*th prime number:

```
primes = ... ; infinite list of all prime numbers
nthprime n = element n primes
```

the problem with this is that the value `primes` will get evaluated through calls to `nthprime`, and will never disappear: the size of the structure representing them will be proportional to the largest argument ever given to `nthprime`. This may be what the programmer wants: it is in accordance with the philosophy of lazy evaluation, in that prime numbers are only calculated when they are needed, and nothing is ever recalculated. In a program which executes for a long time, however, this is not really desirable.

Another example is memo functions, which do not repeat computations if applied to the same argument twice. For instance, consider the function

```
memoise fn x = element x (map fn (1...))
```

When applied to some function that expects a natural number as an argument it yields an identical but possibly more efficient function: for instance, if `fib` is the fibonacci function coded in a naive recursive manner, then `memoise fib` is a more efficient implementation of the fibonacci function. The catch is that the space taken up by `memoise fib` is proportional to the largest argument that it is ever called with.

These examples may at first sight seem rather contrived, but the problem is that the programmer might not be aware of their existence. They are difficult to spot, and lessen the generality of some very useful programming tools, such as `memoise`.

The two examples shown above are both brought about by infinite values that do get used, and so an experienced programmer is unlikely to fall for them. The next example is somewhat more subtle, as it happens entirely within functions that look well behaved.

```
drop 0 1      = 1
drop n (x:y) = drop (n - 1) y
```

```
drop1000 = drop 1000
```

The function `drop` discards the first n elements of a list. `drop1000` drops the first 1000 elements of a list. This is known as the *parameterisation* (or *partial evaluation*) of a function, and is an extremely useful and common programming technique. Unfortunately, once it has been used the structure representing `drop1000` is likely to be of the form

```
B tl (B tl (B tl (... (B tl I)...))).
```

This means that the representation of the function will take up about 2000 storage cells—in effect the `drop` loop has been expanded out in memory. This is in accordance with the idea of laziness: after all, `drop1000` will now execute extremely quickly! (`B` is the `B` combinator and performs function composition, `tl` takes the tail of a list).

This particular example can depend on the precise details of the combinator abstraction algorithm that is being used. Unfortunately, it appears that more sophisticated algorithms generate code that is more susceptible to this sort of problem. The combinator philosophy is to regard laziness as paramount and to avoid doing repeated calculations whenever possible, without regard for the use of space in the result.

A preferable aim would be to act somewhat like a cache, in the following way. The size of every structure should be recorded at every stage during its lifetime, and if a structure has not been used for some time it could revert to some earlier, smaller form. This decision would also depend on how much calculation was needed to generate the large value from the small, so that a lengthy calculation would hold on to its result longer than a trivial one. Implementing this scheme precisely as stated here is probably impractical, but some approximation to it (perhaps controlled by explicit hints from the programmer) would be interesting. At least, the laziness philosophy should not be allowed to stand unchallenged: there are many cases where it does not represent the best course of action.

Finally, two example programs are produced here to demonstrate the unreliability of laziness. They are not concerned with space usage, but each could be written by an application programmer under the impression that laziness would save work, when in fact recalculation would occur. When explaining what laziness achieves a couple of examples and the assertion that “laziness means that senseless recalculation is avoided” is not enough guidance for a serious application programmer.

The first problem is that work is only saved if arguments are supplied in the correct order. For instance, consider the two following uses of the function `f`:

```
f x y = 2 * factorial x + exponential y
```

```
map g [3, 4, 5, 12, 14, 8, 6]
WHERE g = f 5           ; factorial 5 only calculated once
```

```
map h [3, 4, 5, 12, 14, 8, 6]
WHERE h x = f x 8      ; exponential 8 calculated many times
```

The function `g` above represents conventional parameterisation, and the value of `factorial 5` will only be computed once (as the programmer intended). With `h`, however, the programmer imagines that `exponential 8` will be computed once: in fact, it will be recomputed for each useage of `h`. When writing a function, care should be taken to think out any possible parameterisation that could occur and to place arguments most likely to be parameterised at the front of the argument list. A lazy version of `f` that can be parameterised on `y` can only be achieved by rewriting and recompiling `f`.

This seems to indicate that the use of curried functions leaves something to be desired as a way of allowing parameterisation. It results in functions being "lopsided", in that they can only be parameterised in the order predicted by the author of the function, if laziness is to be retained. This is not a property of the semantics of curried functions, in that both `g` and `h` in the example above produce their intended results, but of performance. It may just be a deficiency in current implementation techniques, although I have not been able to devise any simple change that improves the situation.

The second program shows that quite trivial rewritings of a program can drastically affect the amount of work that it does.

```
f x 0 = 0
f x n = factorial x + f x (n - 1)
```

```
g x = p WHERE p 0 = 0
           p n = factorial x + p (n - 1)
```

The two functions `f` and `g` defined here seem very similar: in my opinion `f` is the clearer of the two, but this is not very important. What is not clear is that when they are executed, `factorial x` will get executed once by `g` and `n` times by `f`. A fairly careful study of the two is necessary in order to see how this happens. It should be possible for a program transformation system to substitute `g` for `f`, though I do not know of any such system in use today. What these two examples demonstrate is that not only is our ability to reason about the performance of functional programs extremely weak, but also our intuition can be very easily misled.

Arrays

In functional programs lazy lists are usually used in preference to arrays. In conventional languages there is a plethora of different ways of representing a *sequence* of values: arrays, linked lists, strings and input/output streams are all used in triflingly different circumstances to represent the same idea. The lazy list in a functional language replaces all of these, resulting in simpler, more powerful programming tools.

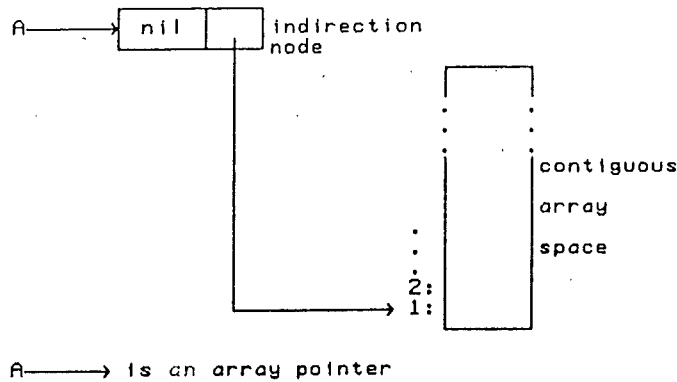
But elements of a lazy list cannot all be accessed quickly: efficient, randomly accessible arrays are needed for the coding of some algorithms. Examples include the construction of a histogram of some incoming data, or the use of symbol tables based on hashing. At first sight arrays seem difficult to implement on a functional machine. Arrays should be objects for which the following operations are available:

```
list_to_array l      = .. ; makes an array from this list
array_element i a   = .. ; extracts the ith element of the array
array_update i e a = .. ; an array the same as a, except that
                        ; element i is replaced by e
```

An array stores a number of values, none of which need be fully evaluated. Access to any element of the array by its index position can be thought of as an efficient operation, and is achieved through the use of `array_element`. A new array value can be generated by `array_update`, which is the same as some previous value except in one position: this too should be an efficient operation. Further details, such as determining the size of an array or multiple dimensions are not relevant to this discussion, as they can easily be constructed in terms of the ideas described above.

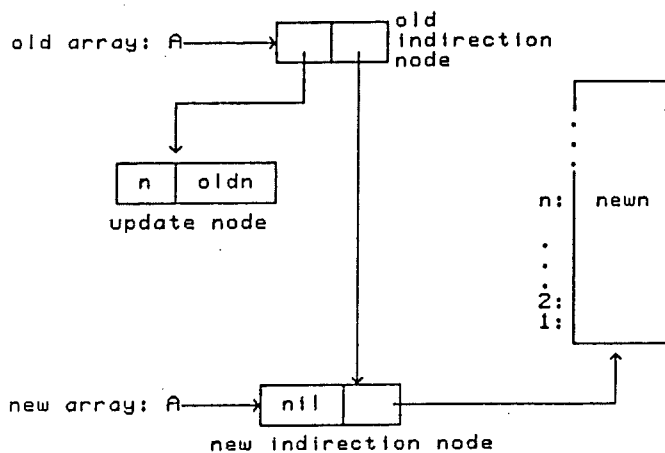
If the elements of the array are to be represented by consecutive locations in the computer's store, so that the read operation is fast, how is the update operation to be implemented without copying the whole array? Remember that the previous array value must be available, as the update operation generates a new array value rather than updating an array object.

A solution to this problem is outlined in [Holmström 81], based on the same principle as the UNDO facility available in some Lisp systems. It operates on the assumption that although the old array must be available, in fact it will rarely be used: the new array is much more likely to be used than the old one. Thus, an array is constructed using the following structure in store:



The representation of an array value

The array is represented as a specially tagged pointer, on which only array operations are legal. This points to an indirection node containing a null value in one side and a pointer to the array storage in the other. The elements of the array are stored in consecutive locations of store, and so reading any element of the array can occur quickly. When an update function is obeyed, the array is updated in place to form the following structure:



Generating a new array with element n different

In the diagram element n of the old array is the value oldn. The programmer wants a new array, the same as the old one except that element n is replaced by the value newn. The method relies on the idea that the old array is hardly ever accessed: so, the indirection node of the old array is updated to contain enough information to recreate the value oldn, and the array space itself is updated to be correct for the new array. Access to the new array will now be just as fast as ever. The old array can still be accessed, although it will be slower.

This mechanism seems to provide all that we expect from an array after working with conventional languages. Since conventional languages do not allow "backtracking" to old versions of the array, it seems that the use of familiar techniques when programming with these arrays will lead only to the efficient array accesses. This is known as the "single threadedness" argument. To test this hypothesis arrays like this were implemented on SKIM, but the result turned out to be less than satisfactory.

In order to see why this scheme can fail drastically it is necessary to study its behaviour when a great many updates occur to an array. Suppose that a series of updates occurs to array `a0` but that the original array is still required. `a0` will now be represented by a long chain of indirection nodes, one for each update that has occurred to the array. References to `a0` will now be *very* expensive, and in order to use it at all we should really make a new, fresh copy of `a0` in a new area of array space. This is reasonable: if use is being made of two arrays and good performance expected from both, two areas of array store are required.

Unfortunately, normal order evaluation causes a blurring of the order in which operations are actually performed on the structure in memory, with the result that this copying often occurs without really being necessary. For instance, consider the following function:

```
do_to_array_elem n fn ar = array_update n (fn (array_read n ar)) ar
```

The function applies the function `fn` to element `n` of the array `ar`. But what actually happens under a normal evaluation scheme is that the array is updated with a closure that refers to the previous array. If several such updates are made to an array before any of its elements are read, the result will include references to all the previous arrays: some via quite long chains of indirection nodes.

The application that actually prompted me to provide microcoded support for arrays was the provision of hash tables for a compiler, and for these `do_to_array_elem` was one of the first functions that was written in connection with arrays. The result was spectacularly awful: almost every access to the hash table caused it to be copied.

The problem was fixed in this instance by making `do_to_array_elem` a microcoded function rather than `array_update`. The latter can be constructed easily from the former, and does not lead to this problem. I am not proud of this solution: it works for the particular application which I required, but presumably would not work for some other program. Whatever scheme is employed, it is never very hard to suggest pathological cases whereby large numbers of copies of the array are made, causing serious performance problems.

A certain amount of work has been done on examining use of arrays at compile time in order to verify that accesses are single-threaded [Hudak 85]: unfortunately, Hudak's work only applied in a limited domain, where arrays hold fully evaluated objects and all evaluation is applicative order. Also, some quite clever ideas have been produced concerning what should be done if a reference to an array occurs through an indirection chain. [Hughes 85] suggests that the order of the indirection chain should simply be reversed, so that the most recently accessed array is represented in contiguous store. This works beautifully in some examples, for instance when backtracking in a depth-first tree search, but can also be defeated in other (quite simple) cases. This leads to the suggestion that the user should be provided with a variety of array implementation strategies, and asked to indicate in some way which should be used: however, this hardly presents functional programming as being "high level". Alternatively the

compiler could try to decide what the array's pattern of use is likely to be, but as Hudak's work shows this kind of analysis is extremely complex.

In the light of all these problems, the much simpler idea of using binary trees to represent arrays seems quite attractive. Binary trees have an obvious performance disadvantage, but with low level support could still be made attractive. In addition they are somewhat easier to use than contiguous memory arrays, as an array's size does not have to be declared when it is created, and sparse arrays are handled quite efficiently with no extra effort. They also reduce the complexity required of the underlying memory management system, which is no longer required to allocate contiguous chunks of memory. In a machine with multiple processors this sort of consideration may be quite important.

Compact Representations

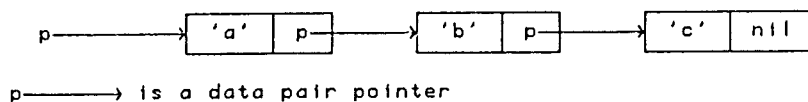
A scheme was devised and implemented on SKIM for reducing the amount of store taken up by strings. A string is a list of characters, but in the particular application envisaged for SKIM (a simple programming environment) it is quite possible that several tens of kilobytes of text would be kept in store. Using one node for each character results in significant amounts of SKIM's memory being taken up by such text.

For this reason the *packed string* was invented. This is an object that stores characters in contiguous memory, holding four characters in the same space as one node. The only operations available on packed strings are:

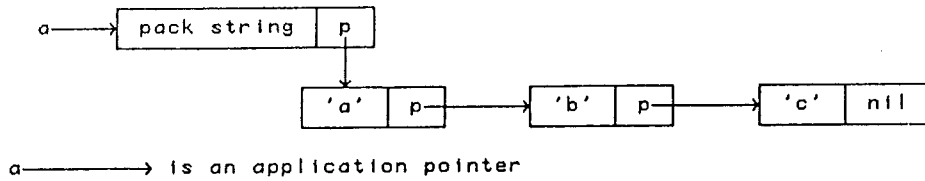
```
pack_string s = ... ; packed version of String s
unpack_string p = ... ; the string back again
```

This is a simple solution to a practical problem: it results in a dramatic improvement in store usage, and was quite easy to implement. Unfortunately using it is not as simple as it might at first appear.

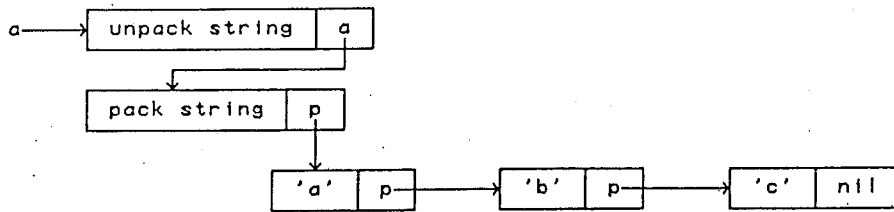
For instance, consider the string "abc", normally represented as



When the function `pack_string` is applied to it, a closure will be performed until the structure is actually accessed.



The function will only be evaluated when its value is required, and that will only be when `unpack_string` is applied to it. Thus, the following structure will be formed:



Both functions will now be evaluated in quick succession, and nothing will have been gained: the packed value spends most of its life as a closure including the unpacked string, and so no space has been saved. In order for this function to be useful we must touch the packed value deliberately, just to change its format in store.

`pack_string` should really be called

`I'm_probably_not_going_to_use_this_value_for_a_while.`

It should really be viewed as a hint rather than a function, that the value is not going to be used for some time and so the evaluation system may (if it wishes) make use of this information in any manner that it desires. On SKIM this is only useful for strings, and changes them to a compact format. On other machines it might apply to arbitrary structures, and result in them being moved to disc. This corresponds very closely to the idea of overlays in a conventional language. Care should be taken if this is used on large structured values, as some sharing might be lost. Also, program transformation systems must realise that it does something useful, even though it has no effect on the value of the program.

Solutions

All of the examples shown in this chapter represent things that can go wrong during the execution of a functional program, errors not of correctness but of performance. Many of them can be avoided if the programmer is aware of them, but I have called them implementation problems because they force the programmer to know much more about the implementation than he or she should

have to. Also, they can vary in an unexpected and unacceptable way depending on the implementation technique being used.

From these examples two lessons can be suggested.

The programmer should think carefully about operations on any single structure that consumes a significant fraction of the total machine store.

If the program is likely to run for some time the programmer should think carefully about any source of gradual space leak.

These lessons are extremely vague, and not likely to inspire the confidence of prospective application programmers. What advice can be given to a programmer whose program gets gradually slower, or which uses far more store than was expected? Sadly, not much: especially if the program is large and complicated. Such matters are likely to cause portability problems too, if triflingly different implementation techniques (or different amounts of memory) are used on different machines.

As a general rule, all of these problems become dramatically worse when a parallel evaluation machine is used. Actual transient store useage will vary spectacularly, depending on arbitrary choices made by the scheduler.

I have tried to suggest solutions to each problem, although most are unsatisfactory. In many cases half the problem is that the programmer may be unaware of what is going on at run time: the only symptom of any trouble is sadly reduced performance. To this end, it may be that the most important step towards solving these problems will be the provision of solid language implementations with better tracing, profiling and debugging facilities. Most current research in producing good functional language implementations is directed towards improving performance. The solution of some of the problems described in this chapter might be achieved more quickly by directing attention towards the design of better runtime diagnostics instead.

Functional Programming

One of the aims of the SKIM project was to investigate the use of functional programming for the construction of large programs. As a consequence of this a great deal of time was spent programming in functional languages and a number of ideas were formed by this experience. The remarks in this chapter refer only to the type of functional language that SKIM supports (i.e. using laziness and higher order functions, see page 2.5) and assume experience of programming in both conventional and functional languages. No attempt is made to produce an elementary guide to functional programming, as this ground is adequately covered elsewhere [Henderson 80, Darlington 82, Peyton Jones 84].

The central argument advanced in this chapter is that the design of many conventional languages is influenced far too heavily by performance considerations, rather than by attempting to make a language that aids the construction of useful software. In order to design better languages it is necessary to ignore the implementor's plight for a while, and search for programming constructs that aid the construction of well-structured programs with little regard for performance. Functional languages have benefited considerably by this approach.

The first three sections of this chapter cover three important features of functional language design, and try to show why they are preferable to the counterparts found in imperative languages. The first highlights the fact that performance and expressiveness can sometimes be in direct conflict, and suggests the principle that should guide the language designer in such circumstances. The second and third sections demonstrate that a small number of powerful language features is preferable to a plethora of less powerful ones, due to increased conceptual clarity and to the removal of many arbitrary decisions in the design of module interfaces.

The rest of the chapter tries to show the effect that these phenomena have on program construction, with first a small and then a large example. Rather than claiming that functional ideas "make programming easy", I suggest that they force a programmer to see much more clearly the true complexities of the problem being solved, and so lead to a better structured, and more reusable, solution.

Using a Heap

Most conventional languages provide facilities for the manipulation of complex data structures. In Pascal, for instance, records can be generated on a heap, and intricate structures produced and processed. The programmer is responsible for allocating and deallocating store, and is taught that generating new structures is a tricky and expensive operation, to be used with care. Many computations proceed by updating such structures in place, and it seems hard to understand how certain operations could be performed without this.

Functional languages also provide a heap on which records can be produced, but do not allow such structures to be updated once they have been created. If the programmer wants to change a structure he should create a new structure incorporating the changes, and discard the old one. Rather than a structure being transformed during computation, it is filtered to produce new structures.

This style of programming is considerably easier to use than updating, because there is no longer any need to think in terms of a heap in computer store: the programmer manipulates structured *values*, without having to think in terms of pointers and records.

It also removes some decision-making in the design of program interfaces. Consider a Pascal procedure which takes a structure as an argument: a vital part of the specification of this procedure is whether it overwrites any parts of this structure. If it does, and the the caller of the procedure wishes to use the structure for some other use, he will have to arrange to duplicate the structure in store. We are not complaining here about inefficiency, but trying to point out that the designer and maintainer of the program have more to think about than when using a functional language, and that their task is correspondingly harder.

Conventional languages are beginning to reflect this: in CLU [Liskov 81], for instance, all structures are sharply divided into those which are *mutable* and those which are not. Mutable objects may be overwritten in store. CLU programmers find it an extremely useful discipline to think in these terms.

The drawback that all this seems to ignore is the issue of performance: the style of programming advocated here for a functional language seems to consist of techniques that conventional programmers usually regard as inefficient. In answer to this we note that this style of programming is very important to the functional programmer, and that machines designed to execute functional languages efficiently are particularly good at allocating and reclaiming storage. This is part of the functional machine philosophy: if a desirable programming style is not efficient on our machine, *adopt the style and change the machine*. In the long term, this is the only strategy that will produce machine designs that efficiently support software that is written in the way that we *want* to write it.

Higher Order Functions

Many languages provide the ability to manipulate functions as first class data objects, but most of them fail to provide the support that is needed to allow this facility them to achieve its full potential. By limiting the extent of higher order functions (as in Algol 68) or by fudging the scope issue (as in Lisp) or by providing an unpleasant syntax and sweeping restrictions that discourage their use (as in Pascal) they ensure that higher order functions are thought of as an exotic language facility which is avoided unless its use is absolutely necessary.

In fact, higher order functions are such a powerful idea that most other control and data constructs can be expressed in terms of them. The importance of this fact is often misunderstood by conventional programmers, who think: "What does it matter that conditional statements can be defined in terms of higher order functions? It's much more convenient to have them built in to the language". The enthusiasm that many functional programmers have for boasting about the number of features that functional languages do *not* have is greeted with similar confusion.

For every feature in the design of a language, it is worth asking

Why is this built in to the language?

Could it be defined in terms of the other features of the language?

Could it be defined in terms of the expansion facilities that the language provides?

The objective of reducing the number of features built in to a language is to try to understand the power and expressiveness of its underlying concepts, in the hope of inventing better ones. Most programmers, and even language designers, are unaware of the liberating effect that higher order functions, properly supported, can have on programming style. They take the form of conventional languages, with their complex plethora of control and data constructs, more or less for granted. Functional languages provide fewer built-in constructs than their conventional counterparts, yet the result is more expressive and more flexible. Concerning efficiency, the remarks from the previous section apply: by building a machine that finds function application easy, we provide the programmer with a better language in which to work.

Arrays and Lists

Lists are used in functional programs to represent ordered sequences of objects. In the vast majority of cases they constitute a simpler and more useful tool than arrays. They are more powerful and flexible than the lists found in Lisp, because elements of a list in a functional language are not all constrained to exist at the same time: lazy evaluation allows lists to replace sequences of values usually found in loop variables and input and output statements.

The consequence of this is that lists become a "common currency", removing arbitrary choices from the design of module interfaces and increasing the generality of many programming tools. For instance, consider the following typical fragments of code from conventional programs:

```
FOR i := 1 TO asize DO
  BEGIN
    ... operations on a[i] ...
  END

WHILE flag DO
  BEGIN
    read(a);
    ... operations on a ...
  END

WHILE p <> NIL DO
  BEGIN
    ... operations on structure p^ ...
    p := p^.next
  END
```

All of these constructions are methods of iterating over a sequence of values. The programmer would choose one of these depending on the original form of his data and the required output form. A programming tool that assumes any of these of its input data is considerably restricted in its usefulness.

Unlike arrays, linked lists are not randomly accessible in unit time. Thus, it is sometimes necessary to use arrays in order to represent an algorithm efficiently, but far less often than it may seem to a programmer brought up using conventional languages. Many algorithms and structures (e.g. sorting, circular queues) may appear to need random access arrays, but in fact programs that perform them using linear lists are no less efficient, and considerably clearer and more compact. Some algorithms (e.g. hash tables) do need randomly accessible arrays if they are to be implemented efficiently. Functional programmers do not suggest that arrays should be done away with entirely, merely that they are very rarely the best tool to use.

Conventional computers implement arrays exceptionally well, because they correspond very closely to the conventional model of machine store. But they should be recognised as semantically complex objects, whose use is encouraged in conventional languages because they are easy to implement, rather than being the right conceptual model for a variety of tasks.

Abstraction

This section is expressed in terms of an example program that performs a simple textual operation: the splitting of text into words. This is an easy concept to grasp and is used quite frequently, so it seems reasonable that a library function should be provided which performs it. Despite its simplicity, this example illustrates numerous differences between functional and conventional programming.

In a conventional language, it is not at all clear what the precise specification of this library function should be.

What form should the input take?

Take characters from the current input stream?

A string?

An array of characters?

A sequence of calls to a provided function?

What form should the output take?

Send characters to the current output stream?

An array of strings? (but who should allocate the space?)

A sequence of calls to a provided function?

How should words be defined?

Tab, space and newline separates them?

A character array lookup table?

A predicate that says if a character is part of a word?

Should word definition be a parameter, or a global variable?

The implementor of the library function will make choices about matters like this, and the end result may or may not be easier to use than writing your own. An idea as general as "split text into words" is not easily amenable to treatment in this way.

In a functional language, this operation would be implemented as a function taking two parameters: a predicate that says if a character is part of a word or not, and a list of characters. The result would be a list of words. The predicate would be the first parameter, because its value is more likely to be stable than the value of the input characters, so a user may wish to parameterise the function to provide himself with one that is easier to use (this removes the awkwardness of having a global involved, while allowing the result to be used with just one argument).

The advantages of the functional approach in this particular example are as follows:

The result is easier to interface to, and more likely to be useful: the user's input will almost certainly be in the form of a list of characters.

The result has a specification that is easy to remember.

The result works on infinite streams or on data items: no assumptions have been made concerning this.

The result applies not just to lists of characters: e.g. finding paragraphs in a list of strings, by detecting sections bounded by blank lines.

The result is lazy, and can be used to solve just a fraction of the intended problem. It can be used efficiently to find the first word, or the first five words, if that is what the user requires.

This example demonstrates that when using a functional language, the abstraction of a small, nebulous idea is possible where previously the cost of doing so would outweigh the advantage of having a library function. Abstraction is an important idea in programming, and the procedure call is the most common mechanism by which passages of code in conventional languages can be abstracted. I suggest that functional languages provide *better* opportunities for abstraction in all aspects of program design, and encourage data, control and algorithmic abstraction in a way that conventional languages struggle to achieve. I do not suggest that this single, rather puny example proves anything, but that it illustrates a phenomenon that recurs constantly when using a functional language: abstraction becomes feasible (and effortless) in areas where it was not previously possible, and the result is program sections that are easier to reuse and programs that are easier to write, maintain and understand.

Abstraction in Large Programs

A change of scale changes many aspects of this discussion. Complex problems do not divide elegantly into a collection of simple filters, and many of the data objects being used are complex, abstract ones rather than simple characters. Input and output, profiling, debugging, program maintenance and evolution all become important issues. Writing a large program is usually achieved in practice by splitting the problem to be solved into manageable, smaller problems. In this process abstraction is a key idea, but one which is frequently misused when programming with conventional languages.

Consider the design of a compiler, which might be split (at the top level) into sections like this:

lexical analyser —> syntax analyser —> code generator —> translator

The lexical analyser is called as a subroutine of the syntax analyser. It reads characters from the input stream and places the result in a global variable or structure. The syntax analyser builds a tree from the resulting lexemes. The code generator then crawls over the tree, generating code for an abstract machine

which is sent to an output file. The code translator then converts this to the code for the target machine. This is written out to another file. Only the translator is target machine specific.

How much information does the diagram above really convey? A little contemplation reveals that it is purely illustrative, and does not really tell us anything about the interactions of the various sections. Does the lexical analyser initialise something that the translator needs? Does the code generator free space allocated by the syntax analyser? Exactly what information is communicated by the lexical analyser? Which sections are independent of the input/output system? In a conventional language there are many questions of this sort that are hard to answer with confidence, and documentation such as the block diagram above is of little help. (Modern modular languages make steps in the right direction, but it is still easy to make module dependencies unnecessarily complex).

But isn't hiding ugliness what "abstraction" is all about? The compiler example attempts to show that abstraction is frequently used in conventional languages in ways that not only obscure implementation details, but also obscure aspects of its specification. Functional languages help not only in performing abstraction, but also in highlighting those aspects of a program's definition which *shouldn't* be abstracted, as a false impression of the true complexities of the program will result and will lead to disaster later on.

Consider the design of a compiler written in a functional language. *All* flow of data between parts of the compiler must be planned at a high level and will show up clearly in the type definitions of lexemes, syntax trees and so on. Examination of the high level code of the compiler will give a true picture of the relationships of the various sections to each other. Such things as error messages and program statistics must be included as part of the high level design, or if they are added to the compiler at some later date then the code will clearly reflect this, rather than allowing hidden extra dependencies between low level sections of the program to creep in.

This sounds as though it could impede program development, by making programs less flexible in the face of changing specifications and developing ideas on the part of the programmer. It certainly feels like it when first embarking on a new project, but my experience is that the programmer is forced to think ideas out in greater depth before writing any code and that this soon pays back the extra time spent. In effect, the specification must be debugged before the program is coded. The result is more thinking before coding, but *much* less debugging time afterwards and a result that is a useful, coherent building block for larger programs.

Now consider the incorporation of either of these compilers into a syntax-directed text editor, several years after it was first written and by a different programmer. Which program would be the more flexible one? I suggest that it would be the functional one, for a variety of reasons:

Examination of the top-level code of the compiler is far more likely to lead to a clear understanding of the real complexities of the program.

Module dependencies are simpler and module communication methods clear and obvious. For instance, familiar tools can be used to manipulate streams of lexemes just like lists of anything else, even if such an idea had not occurred to the original programmer.

Such aspects of module specifications as debugging statements and error handling are more likely to be incorporated cleanly into the overall design of the module. For instance, a section that sends its error messages directly to the terminal is unlikely to be helpful from within a full-screen text editor.

If the new programmer wishes to replace one or more of the compiler sections, he is far more likely to be able to do so. The separate sections of the compiler will be less tightly intertwined, even if such an eventuality had not occurred to the original programmer.

All programs should be written as though they are later to be included as part of a bigger program. Their internal structure should be as clear and simple as possible, so that separate parts of the resulting program can be used as tools in their own right. These sound like laudable aims, but a functional style is quite likely to lead to them being achieved without any conscious extra effort on the part of the programmer.

Input and Output

There is considerable debate in the functional programming community concerning this subject, and a number of schemes have been suggested in order to represent it. The strategy presented here is based on treating lists as input and output streams. As we will see this strategy has a number of potential pitfalls, but the adoption of an approach called *synchronised streams* avoids many of these.

Why Worry?

Input and output operations, and the whole idea of controlling or interacting with the outside world, do not appear to fit gracefully with the functional programming view of computation. Unfortunately, if we wish to represent algorithms using functional languages then some method must be found for interfacing them to the outside world. The simplest is to limit the use of the functional language to an interactive “question and answer” session, such that the user types a functional expression and the system reduces it to normal form, prints the result, and asks for the next expression. This allows the user to perform any computation, but many real applications do not fit well within such a structure. In particular, many complex programs (especially portions of operating systems) require a subtle mixture of algorithmic computation and external control and interaction.

Another idea is to embed the functional language in an imperative one, which performs all input and output operations. The disadvantage of this approach is that in many cases it results in a substantial fraction of the total programming effort being expended on developing code in the imperative language. If functional languages are the best tool for writing programs, then this should be regarded as a defeat (if they're not, none of this is important). Thus, we must experiment with extending functional programming techniques to cover interaction and control operations as much as possible.

Why are input and output more of a problem than in conventional languages? In order to understand this it is necessary to review some basic properties of functional languages.

Referential Transparency

Functional languages bear a strong resemblance to the lambda calculus. Usually the only additions are a certain amount of syntactic sugaring, and some basic data items like integers and characters. For instance, consider the following program written in Miranda [Turner 85]:

```
double n = n + n
```

```
double (double 3)
```

Although it may not appear so to a reader unfamiliar with Miranda, this program is really the same as any of the following lambda expressions:

```
(λdouble . double (double 3)) (λn. n + n)
```

```
(λn. n + n) ((λn . n + n) 3)
```

```
(λn . n + n) (3 + 3)
```

```
(3 + 3) + (3 + 3)
```

```
12
```

This close relationship with the lambda calculus that functional languages enjoy leads to many desirable properties, to which I refer collectively by the term *referential transparency*. Programs written in a functional language can be transformed into equivalent programs by applying very simple rules. For instance, consider the following three function definitions, provided in Miranda and in a well-known imperative language:

```
fn1 x = myfn x + myfn x
fn2 x = 2 * myfn x
fn3 x = i + i WHERE i = myfn x
```

```
INTEGER FUNCTION FN1(X)
FN1 = MYFN(X) + MYFN(X)
RETURN
END
```

```
INTEGER FUNCTION FN2(X)
FN2 = 2 * MYFN(X)
RETURN
END
```

```
INTEGER FUNCTION FN3(X)
I = MYFN(X)
FN3 = I + I
RETURN
END
```

The functions FN1, FN2 and FN3 above are probably identical, but there is no way of being sure without knowing something about the code for MYFN, which could

contain updates or references to globals, input or output statements, or anything else for that matter. But the value returned by a call to a function in Miranda is determined only by its arguments, and has no other effect except to return that value. In the Miranda version of the example above we can be absolutely sure, without knowing anything about `myfn`, that `fn1`, `fn2` and `fn3` are identical and interchangeable. The importance of this property of functional languages is more fully described in [Turner 82] and [Darlington 82a].

It should be emphasised that this property is the hallmark of a functional language, and that under no circumstances should it be abandoned. If a program is being used to interact with external devices and agents then, in some sense, its evaluation will cause "side effects". But the ability to manipulate programs in the way that Darlington describes, and the conceptual clarity that this gives to the programming process, should be regarded as fundamental.

Simple Input and Output

Functional languages are extremely powerful and succinct tools for expressing algorithms, but the limitations described above on the properties of functions seem to pose problems for input and output operations. For instance, it is not possible for a function to return the next value from an input stream, because it would need to return a different value every time it was called. It would not be possible for a function to print its argument at the terminal and return some useless value, for then the following two functions would not be interchangeable, as they should be:

```
fn1 n = print n + print n
fn2 n = x + x WHERE x = print n
```

The usual way in which simple input and output are modelled is by viewing the input and output streams as lists of data objects. For instance, consider the following interactive program. The input from the keyboard is represented as a list of characters. The program evaluates to a list of characters, which are printed on the screen.

```
screen = map uppercase keyboard_input

uppercase x = ... ; takes character x,
              ; returns an uppercased version
map fn (a:rest) = fn a : map fn rest
map fn [] = []
```

(Details of Miranda can be found in appendix B if required). `uppercase` is used as a higher order function by `map`, which applies it in turn to every member of the list `keyboard_input`. The result is that, for every character pressed on the keyboard, the uppercase version of it appears on the screen. This interactive

program is a fairly trivial one, some more complex ones (built on the same lines) can be found in [Henderson 84]. Interaction between a human and a functional program can be achieved by making the program's output a pure function of its input.

A More Complex Case: Polling

Interaction involving a screen and a keyboard appears to fit naturally with the use of two lists of characters. How should more complex devices be represented? Although the use of characters may be simplistic, the use of lists is a perfectly adequate representation of sequential communication with any external device.

For instance, consider a keyboard with a somewhat different specification from the one above, in that the user can *poll* the keyboard to see if any characters have been pressed. The keyboard in the section above cannot support this action, if the program attempts to evaluate `keyboard_input` and no key has been pressed then the program will wait for a key to be pressed. For instance, suppose that a program is required which prints out prime numbers until a key is pressed, which causes it to stop.

The most obvious suggestion to attain this ability might be to provide a primitive that operates on the input stream:

```
is_fully_evaluated x = TRUE  if x is a fully evaluated object
is_fully_evaluated x = FALSE if is is not fully evaluated
```

This test may appear to solve the problem in hand, that of being able to poll the keyboard, but it sacrifices referential transparency and it must not be allowed. In its presence the following two programs are not equivalent, as they should be, and our ability to manipulate functional programs according to simple syntactic rules has been destroyed.

The first program has been written to use the function described above to display primes until a key is pressed. The function `start` is a simple loop that tests the keyboard input list and either stops, or prints a prime and repeats.

```
screen = start primes
  WHERE
  start pri = "Stop", is_fully_evaluated keyboard_input
             = int2string (hd pri) ++ start (tl pri)
```

`int2string :: Int -> String ; converts an integer to its character form`

Unfortunately it will not work, and demonstrates how a non-transparent construct in a functional language leads to confusion, even if it does not have a "side effect". In a lazy implementation the expression

```
is_fully_evaluated keyboard_input
```

will be evaluated once and then overwritten with the value that this yields. In order to make this program work the runtime system must be fooled into ignoring this, as follows:

```
screen = start primes keyboard_input
WHERE
  start pri in = "Stop", is_fully_evaluated in
               = int2string (hd pri) ++ start (tl pri) in
```

In this program the value `keyboard_input` is represented by the name `in` which is local to the `start` loop, and so the test will be executed every time round the loop. However, any implementation would feel justified (nay, proud) in transforming this program back into the first one again without telling the programmer, as surely this constitutes a useful optimisation. The fact that its properties depend so precisely on slight operational details clearly shows that `is_fully_evaluated` has no place in a functional language.

In order to represent this keyboard, I suggest that both an input and an output stream are required. Furthermore, the program as a whole should have only one output stream. Two different types of object will be produced on this output stream. The first is characters that are to be sent to the screen (without this, there is no point in running the program) and the second is poll request tokens indicating that the program wishes to poll the keyboard. Thus, we will represent the type of objects produced on this stream by the Miranda construct equivalent to a tagged union, as follows:

```
Out ::= ToScreen Char | PollKbd
```

`ToScreen` and `PollKbd` are *type constructors* for the type `Out`, and the program will be of type `[Out]` (i.e. list of `Out`). The evaluating system will consider each object in turn on this list, and if it is a `ToScreen` character then this will be sent to the screen. The input stream will be of type `[In]`, where

```
In ::= FromKbd Char | NoKey.
```

When a `PollKbd` token appears in the output stream the evaluating system will ensure that *something* is available on the input stream: either a character, if one has been pressed, or the token `NoKey`.

What this means is that in order to poll the keyboard, the program should first yield a `PollKbd` token on the output stream, and then examine the next object on the input stream.

The rules described above ensure that this will not cause a delay if no key has been pressed, and allows the following program to be written:

```
output = start primes keyboard_input
  WHERE
    start pri in = PollKbd ++ rest (hd in)
      WHERE
        rest (FromKbd c) = string2out1 "Stop"
        rest NoKey       = int2out1 (hd pri) ++
                          start (tl pri) (tl in)

string2out1 = map ToScreen ; simple conversion function
int2out1    = string2out1 . int2string
```

This program performs the desired operation while retaining the desirable syntactic properties of the functional language.

It is interesting to note that although this program is clearly a function, the *effect* that it produces is dependent on factors of time and synchronisation, and thus appears to go beyond what we can describe using the lambda calculus. Given a particular value for `keyboard_input` it will always produce exactly the same result, yet by allowing the partial result to affect the input value an interactive effect can be achieved.

It may seem that the use of the keyboard has now become more complex, and that a simpler scheme might suffice. Nevertheless, my impression is that the representation of a keyboard suggested above is the right one to use, because it fits well with the more powerful facilities described in the rest of this chapter.

Before proceeding, however, one final alternative strategy will be described, as follows. The output stream is composed merely of characters, but the input stream is of the type `[In]` defined above. An attempt to evaluate the input stream will always yield a result straight away, and that result will be either `NoKey` or the character that has been pressed. This is known as the *hiaton* approach, with the `NoKey` tokens known as *hiatons*. Using it the program above becomes:

```
screen = start primes keyboard_input
  WHERE
    start pri (NoKey : in)      = int2string (hd pri) ++
                                start (tl pri) in
    start pri (FromKbd ch : in) = "Stop"
```

Like the program that uses request tokens, this is a well behaved function that does not use any undesirable constructs. Nevertheless, the previous scheme is preferable because its usefulness is not limited to polling: it provides a way of performing arbitrary question-and-answer type interactions.

Multiple Devices

A number of important principles have been demonstrated by the previous section, the most important being that complex effects can be achieved without stepping out of the functional language framework. Another is that only a single input and a single output stream are used, in spite of the fact that a conversation is occurring with two distinct external agents: the screen and the keyboard. This requires objects of two separate types to appear on the output stream.

It may seem simpler to have two separate output streams, with the program itself being a pair of streams. Unfortunately, this means that the surrounding system must evaluate both output streams at once. A certain amount of experimentation suggests that if the program's input and output operations are inherently sequential then it is easier to produce a single output stream which gives an exact order for interactions with the outside world.

The same question can also be asked of input streams, for although the matter did not arise in the previous section it is clear that in a slightly more complex example, more than one external agent might wish to produce input values. My impression with this question too is that a single stream is usually best: in the same way as happened to the output stream above, the use of more complex types for the input stream can allow the "multiplexing" of communication with a number of different external agents over the same input stream. The following arguments suggest that this is a useful technique:

The number of agents in the outside world with which the program communicates can vary, especially when talking to more complex devices like filing systems and networks. This indicates a need for multiplexing, or a varying number of input streams. This latter idea is not impossible: an "open file" request to the filing system could result in a new input stream appearing as a single item in the old one. Conversation with the file occurs over this new stream, while conversation with the filing system continues on the old one. While this is a valid approach it has no particular advantages.

Program sections that wish to perform input and output operations will all be of type [In] -> [Out]. The user of such a program section does not have to be aware of what operations it performs.

One useful exception to this is a way of including simple input files. A directive of the form

```
get_file "myfile"
```

in the source program evaluates to the content of the named file, as a language object, in the program. There is no objection to the filename being computed during the initial execution of the program, but care must be taken that the value yielded by this expression is not affected by other file activity in the program. The best way to do this is to specify that `get_file` acts on the filing

system as it was at the moment that the program started executing, rather than the filing system as it is at the moment when the expression is evaluated. This is somewhat more tricky to implement, but a necessary way of limiting the generality of this device: as a rule, it should only be used in small programs. A reasonable limitation might be that the function fails (i.e. returns some error value that is taken as being semantically equivalent to nontermination) if used to access a file that has been updated since program execution began.

Synchronisation

A further problem with using streams for input and output is that of synchronisation. Consider the following program, which makes use of the simple interactive stream model. The program is of type `[Char] -> [Char]`. Characters struck on the keyboard appear on the input stream, and all characters in the result appear on the output stream. This plan seems simple and direct, but it has the drawback that synchronisation between input and output messages does not always go exactly as planned.

```
p in = "hit a key:" ++ reply
  WHERE
    reply = letterp (hd in) -> "that's a letter";
        "that's not a letter"
```

The program produces a prompt, waits for a key to be pressed and then says if the key was a letter. Prompts and answers interleave neatly in this case, but now consider the following:

```
p in = "hit a key:" ++ reply
  WHERE
    reply = "SURPRISE!"
```

The intention is that the surprise should appear as soon as a key is pressed, but in fact it will appear straight away, along with the prompt. The opposite problem can easily occur in the first program, where a rewrite of the program that looks completely harmless will in fact cause the prompt to wait for the input character before appearing. In order to make the surprise program work we need something like

```
strictify c x = x, c = c ; If c = c Then x Else x

start in = "hit a key:" ++ strictify (hd in) "SURPRISE!"
```

This is because the previous program achieved synchronisation through *data dependence*. A function that uses lists as input and output streams may wish to delay output until its input *exists*, even though the input value has no relevance to the output. This requirement never arises in normal computations.

This makes `strictify` a rather strange object, for it seems to have no other use besides stream synchronisation. Consider the function

```
nostrictify c x = x
```

which appears to be the same function as `strictify` and yet does less work in order to calculate its result! In fact the two functions are not the same, but nevertheless from an intuitive point of view `nostrictify` appears to be a more useful function than `strictify`, and the fact that we find `strictify` useful is a matter of some concern. For instance, consider the expression

```
If a Then b Else b.
```

It seems likely that value of this expression is `b`, but in the semantics of the normal order lambda calculus this is not necessarily so: it is only `b` if the computation `a` terminates, otherwise it is a nonterminating computation. This fact is rather disappointing: surely it would be helpful for a language evaluator to yield the value `b` in place of this expression, or to allow a program transformation system to perform such a substitution without having to prove that computation `a` terminates? Unfortunately this substitution corresponds directly to replacing `strictify` with `nostrictify` in a functional program, which would cause a synchronisation bug in the program above.

I consider that the appearance of `strictify` in a program is a sure sign that the programmer has had to think carefully about the order of evaluation, and that the functional language has to some extent failed. This is not quite as strong as the objection to the use of `is_fully_evaluated`. Also, `strictify` does in fact have another use: by changing the order of evaluation, it can affect a program's performance. Details of this are given in chapter six, where it appears under the name `touch`: however, in that context the use of another less offensive construct (`par` as described in [Hughes 84]) is usually adequate.

A fully polymorphic version of `strictify` (i.e. of type `* -> ** -> **`) cannot be defined in the lambda calculus [Barendregt 85], but for the purposes of stream synchronisation it is easy to produce a limited version. Thus it is not possible to "ban" the use of `strictify`, for in the case of the example above it can easily be simulated by inserting some artificial data dependency in the program. If input and output streams are represented in this way, data dependency becomes an important part of the program's specification.

In the section on polling I suggested that polling should be initiated by the functional program placing a "request token" on its output stream. In order to solve the synchronisation problem described above I extend this and suggest that *all* input to the functional program should be initiated in this way. When such a discipline is in use I suggest the name *synchronised streams* to describe the way in which the functional program interacts with the outside world.

The types In and Out are now extended as follows:

```
Out ::= ToScreen Char | PollKbd | ReadKbd
In  ::= FromKbd Char | NoKey
```

Not only must the program make an explicit request when polling the keyboard, it must also do so when reading the keyboard in the normal way. This means that the output list specifies completely the ordering of all input and output operations. The two programs used to demonstrate the synchronisation problem now become

```
start in = "hit a key:" ++ [ReadKbd] ++ reply
      WHERE
      reply = ...
```

and reply may depend on (hd in) to any extent that it wishes.

Suggestions can now be made concerning the facilities that should be available in a real language implementation. The type Out has a constructor for each possible interaction with the outside world, and In has one for each possible reply. Programs are functions of type [In] -> [Out].

```
Out ::= Screen String |           ; write string to screen
      KbdPoll |
      KbdChar |
      KbdLine |                   ; perform screen echo, rubout etc.
      Delay Int |                 ; do nothing for short time
      WriteFile String [String] | ; output to named file
      AppendFile String [String] | ; append to end of file
      ReadFile String             ; read content of named file

In  ::= KbdNull |                 ; dud reply to keyboard poll
      FromKbdChar Char |          ; char from keyboard read or poll
      FromKbd String             ; line from keyboard
      DelayOver |                 ; end of delay
      FileContent [String]       ; result of read file op
```

This allows the program to interact with the outside world in a sophisticated way, and to change the state of the computer and its filing system, without breaking the rules for functional languages. It can be extended to provide whatever model of file input/output is preferred by the host computer. Streams, random access files, networks, mice and menus all appear to be extremely "non-functional" ideas, yet using these facilities a functional program is able to control them.

It is unlikely that the collection above is a complete list. Each Out constructor corresponds to a system call in a more conventional language, in which case a serious implementation (on a machine with a large variety of peripheral devices) is likely to have a great many of them, organised into libraries.

Examples

Despite being a considerable advance on previous suggestions, the understanding of how to code complex interactive programs using these tools still isn't well developed. Programs frequently appear slightly cumbersome, and more work is needed on stylistic matters. Two examples serve to illustrate this.

The first is of a game program that keeps a scoreboard in a file. When a game is played the file is updated and a comment produced on the player's skill, and the scoreboard updated. The game is represented as a function that produces output and a score.

```
game :: [In] -> ([Out], [In], Score)

do in
  = [Screen "what is your name?", KbdLine] ++
    gameout ++
    [ReadFile "scores",
     Screen comment,
     WriteFile "scores" new_scores]
  WHERE
    (FromKbd name : in0) = in
    (gameout, in1, score) = game in0
    (FileContent scores : in2) = in1
    prev_score = lookup name scores
    new_scores = update name (max score prev_score) scores

    comment = "that's dreadful", score < (prev_score / 2)
            = "that's no good",   score < prev_score
            = "that's OK",       score < (prev_score + 10)
            = "that's great"
```

There is a problem revealed by this, of implementation rather than of style, which is closely related to the problems discussed in chapter six and which may have some worrying implications. A careful examination of this program reveals that the entire list `gameout` of output operations by the game function is constrained to exist at one time, and will not be released to the store allocator until the whole game is over. This is because it is a member of a tuple, of which other members are needed but only evaluated after all of `gameout` has been constructed (this means that closures will exist that access the tuple, and the value `gameout` will be referenced). This is unacceptable, as the game might proceed for a long time and might perform a great deal of input and output. It will result in the store of the machine being gradually consumed, and eventually exhausted, as the game proceeds. Some further remarks concerning this problem can be found at the end of the next chapter.

The second example represents an operation that appears in writing a full screen text editor which is to work on a conventional low bandwidth terminal. Under such circumstances the editor should minimise screen updating.

```
update :: Edstate -> (Out, Edstate) ; most of the editor is
edit   :: Edstate -> Char -> Edstate ; in these two functions

do state TRUE instream = KbdChar : do (edit state ch) FALSE rest
      WHERE (FromKbdChar ch : rest) = instream

do state FALSE instream
  = PollKbd : a instream
  WHERE
    (out, newstate) = update state
    a (NullKbd : rest) = out : do newstate TRUE rest
    a (FromKbdChar c : rest) = do (edit state c) FALSE rest
```

The editor has a state of type `Edstate` and is written as two functions: one (`update`) that indicates the state of the editor on the screen, and another (`edit`) that takes a keystroke and uses this to create a new editor state. While keys have been pressed on the keyboard `edit` is used to update the state of the editor, and if no keystrokes are outstanding then `update` is used to update the screen.

The fact that `update` is likely to be quite complex, yet is constrained by this high-level design to produce a single `Out` object, indicates that the following extra clause in the definition of `Out` might be useful:

```
Out ::= ... | OutPack [Out]
```

This would allow a list of `Out` objects to be packed into a single one, so that functions constrained to produce a single `Out` object could produce several and then package them up, if this is more convenient.

Consideration of other example programs reveals two further stylistic problems. The first of these is the *stub* problem: any part of the program that wishes to do input and output operations must be handed the input stream as one of its arguments, and must return the output stream and the stub of the input stream as parts of its output (or be handed a continuation as another argument). This can lead to a rather messy programming style, especially for large program sections.

The second is that there is a very strong relationship between request elements in the output stream and the corresponding reply in the input stream, but this relationship is not checked in any way, or reflected in the structure of the facilities provided. A possible programmer error is to get "out of step" on the input stream, and to use accidentally values that are shifted one along from their intended source. It may be that the solution to this problem is to present the input and output streams as a single object: recent work on "dialogue" objects [Redelmeier 84] may be a useful basis for future work here.

Discussion

This chapter has introduced facilities that provide the same range of input and output operations as is provided by most conventional language implementations. These suggestions go considerably further than any previously published scheme, while retaining the programming style of a pure functional language.

The introduction of un-functional constructs is not necessary for the representation of input and output operations, or for any other reason. It should be clearly recognised that any construct which destroys referential transparency is every bit as bad as any other, and in this respect `is_fully_evaluated` is in the same category as a print statement or an assignment statement.

Certain operations are still not possible using the facilities described so far. For instance, consider the language system described in this chapter as a program in its own right. Clearly it is quite a complex program—so, it should be written in a functional language. When attempting to do so a number of limitations become apparent. For instance, the provision of a STOP key that works even if the user does not poll it, and recovery from erroneous user programs, are not possible. More elaborate extensions such as background tasks might also be envisaged, and they too cannot yet be implemented.

It could be argued that such facilities are not really needed, and that the language system is powerful enough without them. However, the idea that we *can't* implement them reveals that this argument is irrelevant: the fact that a functional language is being used to implement a language system should not constrain the designers of that system. Furthermore, if functional languages are really the best software tool available for expressing complex algorithms, then whole operating systems for modern computers are best written in them. If this is to be possible then some means of controlling a number of separate devices and user tasks concurrently is required.

The next chapter is concerned with the development of an underlying computational model of sufficient power that complex multi-tasking operating systems, including and supporting the language system described in this chapter, could be implemented in terms of it.

Operating Systems

A scheme is described for writing nondeterministic programs in a functional language. It is based on message passing between a number of expressions being evaluated in parallel. I suggest that it represents a significant improvement over previous methods employing a nondeterministic merge primitive, and overcomes numerous drawbacks in that approach. A previous version of this chapter appeared as a laboratory Technical Report [Stoye 84a], and it has been accepted for publication by Science of Computer Programming.

Having built SKIM and achieved acceptable performance on it, the next task was to turn it into a useable machine that could support editors, compilers and the like. This meant that device handling, filing systems, error recovery and so on had to be tackled, and led to the need for a new understanding of inherently non-functional operations, and to the development of the scheme described here. The implementation of this model as part of SKIM's combinator reduction software is described in appendix A, this chapter presents it as a more abstract model.

Nondeterminism

One aspect of pure functional languages is their inability to behave nondeterministically. There are a number of reasons why nondeterministic behaviour might be desirable.

1. In order to write operating systems

The behaviour of an operating system seems to be nondeterministic in certain circumstances. These usually take the form of multiple simultaneous demands for attention, e.g. when communicating with multiple users, devices and background tasks.

2. In order to be sufficiently abstract

There are circumstances when one of a collection of results will do, and having to make an arbitrary choice seems undesirable.

3. In order to represent certain objects correctly

Some objects (e.g. sets) inherently have no ordering, and so a proper selection function on them must not impose an order. It is possible that the first two cases are covered by this one, when considered over sets of the correct domain.

4. For performance reasons when tackling certain problems.

Suppose that a problem has a number of possible solution strategies and a number of possible correct solutions, and that a machine with several processors is being used. Each processor should use a different strategy, and the first to yield a solution should cause this solution to be printed and the all processors to be stopped. This example seems easiest to model using nondeterminism.

In practice it seems very difficult to add nondeterminism to functional languages without destroying many of their desirable qualities. The proposals presented here are an attempt to preserve as much as possible the practical "feel" of functional languages, while providing extra tools that enable the programmer to express nondeterminism where it is needed.

Operating Systems

In [Henderson 82], Henderson uses the idea of lists as input and output streams to deal with quite complex examples, including programs that handle multiple files, users and devices. To do this he introduces the idea of *tagging* and *untagging* lists of data items. A list of tagged data items is a list of (tag, data item) pairs. A function `tag` turns a list of data items into a tagged list where all elements have the same tag. A function `untag` extracts from a list of tagged data items those items with a particular tag.

```
tag t (x:rest) = (t,x) : tag t rest
tag t []      = []
```

```
untag t ((t,x):rest) = x : untag t rest
untag t ((wrong,x):rest) = untag t rest
untag t []            = []
```

```
; thus, tag 5 [1, 2, 3] = [(5,1), (5,2), (5,3)]
; and untag 5 [(5,1), (6,2), (5,3)] = [1, 3]
```

The use of these functions allows streams of values from several inputs to be combined into a single list. The elements of the list can then be processed by a function, and passed on to one of several outputs depending on where they came from. His approach has, in my view, a number of drawbacks.

A further operation is necessary: the merging of two (tagged) streams into a single stream. Simply taking alternate elements from each of the two input lists will not work: this operation will frequently be used on "input streams" from the outside world, in which case we want to take the next element from either list, whichever is ready first. This choice is nondeterministic.

The second problem is one of style. When trying progressively larger examples the tagging, merging and untagging of multiple streams appears to form a web that is tangled and impenetrable, and the neat structure of the program rapidly disintegrates. The term "spaghetti programming" has been suggested to describe this. The operating systems of real machines involve a flow of data that is far more complex than that of the examples that Henderson tackles, and some experimentation indicates that this approach soon leads to confusion, with functional programming acting as a hindrance rather than as a help.

All of the tagging and untagging leads to considerable extra work, if it is actually implemented with data manipulations as Henderson seems to suggest. Sometimes values need several layers of tagging, in order to get from one function to another, to which it is not directly connected. In a large, complex program, this could lead to considerable inefficiency. (This is the least important objection, if the use of merge were the right approach on the grounds of style then this problem could certainly be overcome with some effort.)

Henderson's solution to the first problem is the introduction of a nondeterministic operator which I call `merge`.

```
almost_merge (a:rest) b = a : almost_merge rest b
almost_merge a (b:rest) = b : almost_merge a rest
almost_merge [] rest    = rest
almost_merge rest []    = rest
```

```
; This is almost a definition of merge:
; merge [1, 2] [3, 4] = [1, 2, 3, 4]
;                       or [1, 3, 2, 4]
;                       or [1, 3, 4, 2]
;                       or [3, 1, 2, 4]
;                       or [3, 1, 4, 2]
;                       or [3, 4, 1, 2]
```

`merge` behaves *almost* like the function defined above. If `almost_merge` were typed into a real Miranda system, it would merely append two lists together:

Miranda systems try each clause of a function definition in turn, whereas what we want is to apply whichever of the first two rules can be applied "first". `merge` takes two lists as arguments, and yields a single list containing all the elements of both. It does this (in current practical implementations) by creating a new process, and evaluating both of its arguments in parallel. Whichever produces a result first causes that result to be available to anyone requiring the value of this call to `merge`.

Now, this is not a function. Its evaluation does not cause any side effects, but a given call to `merge` may return different results on different occasions with the same arguments. In particular, the expression

$$(\lambda x. \text{fn } x \text{ } x) (\text{merge } a \text{ } b)$$

is not at all the same as

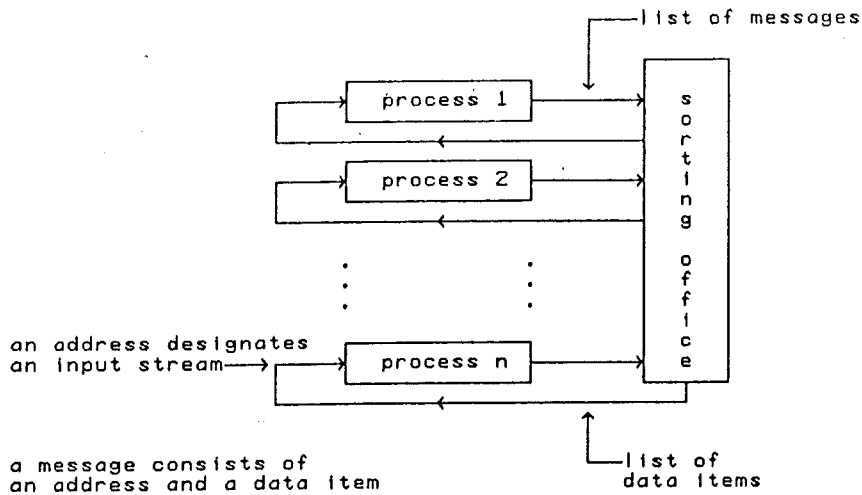
$$\text{fn } (\text{merge } a \text{ } b) (\text{merge } a \text{ } b)$$

and so referential transparency is lost, and our ability to reason about functional programs is undermined.

Message Passing

The following scheme represents a new approach to writing nondeterministic programs in a functional language. By avoiding the introduction of a primitive with a nondeterministic result (such as `merge`), the referential transparency of the language is retained. The idea of tagging streams of data items is retained, but large examples seem to avoid the "spaghetti" effect, and manage to avoid much of its inefficiency.

The scheme is based on evaluating a number of functional expressions in parallel. Each expression thus executed is called a *process*, and has a single input stream and a unique *process address*. Each process evaluates to a list of *messages*, which are pairs of the form (process address, data item). As each message is evaluated its data item is added to the input stream of the process to which it is addressed.



The New Message Passing Scheme in Action

The body of each process is a function which takes a list (the input of the process) as its argument, and yields another list (the output of the process) as its result. These output lists are all lists of messages. The system evaluates all the processes in parallel, and acts like a sorting office for all the messages: each message consists of an address and a data item, and each address designates the input stream of some process. the data item in each message is sent to the input stream of the process to which it is addressed.

Because the input streams are represented by lazy lists, there is in effect an arbitrary buffer in each input stream. The action of "reading" an incoming message is not well defined, and not synchronised in any way.

In Henderson's system merge may appear in any part of the program. This system constrains the use of nondeterminism so that it may only appear at the "bottom level" of a program. Programs written for it use no nondeterministic primitives or constructs, so the desirable mathematical properties of functional languages are retained.

A More Detailed Description

I now describe the operation of this system as a Miranda program containing an occurrence of merge. This program would be terribly inefficient to run; but it serves as a very useful definition of the system as a whole. In the next section it provides the basis for a number of small example programs.

A simple case is described. The system has three processes, whose addresses are 1, 2 and 3. Numbers are used as addresses because later extensions to the system will allow an arbitrary number of processes to run, and so the use of more mnemonic identifiers seems inappropriate.

```

screen      = untag 0 messages
input_to_1 = untag 1 messages
input_to_2 = untag 2 messages
input_to_3 = untag 3 messages
messages = merge3 (start_1 input_to_1)
                  (start_2 input_to_2)
                  (start_3 input_to_3)
merge3 a b c = merge a (merge b c)

```

A system of three processes

It is informative to study the various subexpressions of this program in some detail, as otherwise it is rather hard to understand.

The screen object

```

screen = untag 0 messages
screen :: [Char]           ; it's a list of characters

```

`screen` is the list of characters sent to the screen, and constitutes the “answer” produced by the program. All messages sent to address 0 should have a character data item. When such a message arrives at the screen handler the corresponding character will appear on the screen.

The messages object

```

messages = merge3 (start_1 input_to_1)
                  (start_2 input_to_2)
                  (start_3 input_to_3)
messages :: [Message]
Message == (Address, Data_item)
merge3 a b c = merge a (merge b c)

```

`messages` is a list of all messages ever sent in the system, in the order in which they are processed. It is formed by merging together the output from the three active processes into a single list of messages, and then sending the data item in each message in this list to its destination. The term “output” is used in an informal way here, in fact it is the *value* of each expression which is considered.

The start objects

```
start_1 :: [Data_item] -> [Message]
start_2 :: [Data_item] -> [Message]
start_3 :: [Data_item] -> [Message]
```

These three functions are the bodies of the three processes. Each is applied to a value representing the input stream of that process. The resulting values should be lists of messages.

Some Example Programs

1. A Simple Computation

```
start_1 in = tag 0 ("the answer is " ++ comp)
```

In this example we wish to perform a deterministic computation and display the result, as is possible with simpler systems. Only one process is necessary for this. The computation is called `comp` here and evaluates to a list of characters, which is converted to a list of messages by consing the destination onto each character. Thus `start_1`, when an argument has been applied to it, results in a list of messages. All of the messages are addressed to process 0, and as a consequence will appear in the input to address 0. Since address 0 designates the screen, the result of the computation will appear on the screen, preceded by the characters "the answer is ".

The style of programming takes a little while to get used to. Note that the parameter of `start_1` is not used, but only because this example is very simple. It is necessary in order to ensure that `start_1` is of the correct type to be made into a process. Simple input and output are slightly more cumbersome than in other functional programming systems, but an enormous degree of flexibility has been gained, and as we show later the ability to hide the complexity of device drivers and input/output operations has been provided. This function may seem ugly and artificial, but the author of `comp` (the application program) does not have to know anything about it: the vast majority of applications programmers will never use message passing, and need have no understanding of its operation. The examples shown here are only of relevance to the authors of operating systems.

2. A Nondeterministic Computation

```
start_1 in = [(0, hd in), (0, hd (tl in))]  
  
start_2 in = [(1, 'a')] ; i.e. a list with one element,  
                ; which is a pair  
start_3 in = [(1, 'b')]
```

`start_1` evaluates to a list of two messages, both of which are sent to the screen. `start_2` and `start_3` both send a single message, with a character data item, to `start_1`. The result of this program is to print either "ab" or "ba" on the screen.

The order in which operations occur when `start_1` is evaluated is very important. Process 1 evaluates to a list of two messages, whose data items are obtained by examining the process' input stream. Thus, the behaviour that is required of process 1 is as follows:

- wait for an incoming message
- send a message containing that incoming data item
- wait for a second incoming message
- send a message containing the second data item

Unfortunately, the program that I have specified so far could behave as follows:

- send a message consisting of a closure, referring to the first data item on the input stream
- send a message consisting of a closure, referring to the second data item on the input stream

This is a consequence of the normal order semantics; the value of the data items is not required when the message is sent, so it will produce output messages (with unevaluated data items) before any input has arrived. The first interpretation requires the evaluating system to fully evaluate the data item on a message before it is sent, and in practice it's the preferable interpretation. In this particular example it makes no difference to the final result, but in many other cases it can lead to a message being sent before it is really ready. For instance, suppose that `start_2` and `start_3` were substantial calculations and that there were other possible processes that could send messages to the screen. If the second interpretation of the system's semantics were used then the calculations inherent in `start_2` and `start_3` would not be performed until the messages had already been accepted by the screen. This would congest the screen until those calculations had been performed.

Thus, the first suggested interpretation seems the best in almost all cases. The current system evaluates data items at the top level (but not completely, in the case of a data item consisting of a function or structure) before allowing them to be sent. An analagous decision appears in defining `merge` (should `merge` force the evaluation of the elements of the lists that it is merging?).

3. Input from the Keyboard

A convention will have to be decided upon for the behaviour of the keyboard device. We will assume that, as each key is pressed, the corresponding character is sent to process address 1. A program to echo the characters pressed on the keyboard looks like this:

```
start_1 in = tag 0 in
```

Now, suppose that an application program wishes to take input from the keyboard a line at a time. Echoing and the delete key should be handled by "the system", the user program wants keyboard input to appear as a list of strings.

On functional systems that evaluate a single functional expression it is usually painfully obvious that this transformation (from a character based to a line based keyboard) is going on. It would typically be achieved by a function called something like `ch_to_line_kbd`, which takes a list of characters as input (the characters from the keyboard) and produces two outputs, the lines for the application program and the echoes to be sent to the screen. The use of functions of this sort rapidly leads to "spaghetti", and to data structures that persist and grow throughout the life of the program. This not only results in unclear programs, but if (for instance) it is necessary with file input and output, it could be extremely inefficient. In chapter eight this problem was solved by the language system providing both kinds of keyboard, and indeed for the applications programmer that is what should happen. However, for the author of the language system the ability to define the more complex one (i.e. the line keyboard) in terms of the simpler one, is very important.

This program performs the necessary transformation under the new scheme in a painless way:

```
userprog line_kbd = ... ; the user's program, of type [String] -> [Char]
```

```
start_2 in = tag 0 (userprog in) ; this task runs the user's program
```

```
start_1 in = kbd_loop [] in ; this task is the keyboard handler
```

```
kbd_loop [] (DEL : cs)      = kbd_loop [] cs
kbd_loop (deleted : charbuf)
  (DEL : cs)              = (0, BS) : (0, ' ') : (0, BS) :
                           kbd_loop charbuf cs
kbd_loop charbuf (RET : cs) = (0, CR) : (0, LF) :
                           (2, reverse charbuf) :
                           kbd_loop [] cs
```

The function `kbd_loop` in this program performs the echoing and the building up of a line of input in a buffer. Its first argument is the character buffer, the second is the input stream to process 1. It handles keyboard echo and the delete key, and when return is pressed it sends a message containing a string to process 2.

The program may seem unpleasant at first sight, but it is painless in the sense that the writer of the user program can be unaware of its existence. Note that the systems programmer can implement whatever model he chooses for what to do about typeahead, buffering user output and so on, and can render his choices in a functional language.

(In the program, the identifiers DEL and CR are the codes generated by the delete key and the carriage return key. CR, LF and BS are assumed to be the control characters for carriage return, line feed and backspace. Delete is echoed as backspace, space, backspace and return as carriage return and line feed.)

4. Polling the keyboard

How can a user program poll the keyboard, and react in different ways depending on the speed with which the user reacts? In chapter eight the system was augmented with a special "poll keyboard" primitive, but using parallel tasks no extra primitive is necessary. We will assume that the keyboard behaves as described in example 3. Process 1 provides a service for other processes. It accepts two kinds of input: keys from the keyboard, and "has a key been pressed yet?" requests from users.

```
start_1 in = kbd_loop [] in
```

```
kbd_loop keys (ch:in) = kbd_loop (keys++[ch]) in, character ch
kbd_loop (key : keys) (ad:in) = (ad, key) : kbd_loop keys in
kbd_loop [] (ad:in) = (ad, nullchar) : kbd_loop [] in
```

kbd_loop runs as process 1, while the user program is process 2. kbd_loop behaves in three different ways, depending on the values sent to it:

- If a character arrives, add this to the list of unclaimed characters.

- If a request arrives and we have characters saved up,
send the first character to the requester

- If a request arrives and we have no characters saved up,
send a null character value to the requester

These are reflected in the three clauses of the function definition.

Creating New Processes

The previous section demonstrated a system involving a fixed, finite number of processes communicating with each other via messages, and showed how such a system could be used to build a number of interesting programs. In the examples I have tried to show how it leads to better organised and more efficient programs than the use of merge operators. In this section a mechanism is presented for allowing such a network to change dynamically, so that new processes can be created as required.

The creation of new processes works as follows. Each process executing has a unique process address, to which messages may be sent. In addition to the addresses of the processes executing there may be other addresses to which messages can be sent, causing various special effects. One of these is the screen: we have declared that any message sent to this address must have a character data item, and that sending such a message will cause the character to appear on the computer's video screen.

In a similar way, there is another special address called the *process creator*. Any message sent to this address must have as data item a function of a suitable type (which is described more precisely below). Sending a message to this address causes a new process address to be created, and starts up a new process at that address using the data item of the message as its body.

The underlying message passing mechanism must now be made more complex. Because processes are created dynamically, so too are process addresses. When a new process is created its address must somehow be given to everyone who will want to send messages to it.

This operation is described with more detail and precision by the following program. Address 0 is the screen, and 1 the process creator. There is now only one initial user process, whose body is constructed from the function `start` and whose address is 2. More processes can be created as required.

```
screen = untag 0 messages
process_bodies = start : untag 1 messages
messages = MERGE (mapdyad fn process_bodies (2...))
fn body addr = body addr (untag addr messages)
mapdyad fn (a : as) (b : bs) = fn a b : mapdyad fn as bs
```

A system supporting an arbitrary, dynamic number of processes

The screen object

```
screen = untag 0 messages
screen :: [Char]
```

screen serves the same purpose as before; it is the "answer", or output, of the whole system. Note that in a more complex case there could be many other devices and special addresses visible at this level, depending on the hardware configuration involved.

The list of process bodies

```
process_bodies = start : untag 1 messages
process_bodies :: [Process_body]
Message == (Address, Data_item)
Address == Int (in these examples)
Data_item == any type
Process_body == Address -> [Data_item] -> [Message]
```

`process_bodies` is the list of all process bodies that ever exist, in the order in which they are created. This shows the precise type that is expected of an object sent to the process creator, and how the new address is made accessible to the program. The new process body has applied to it its own (newly created) process address, and its own (newly created) input stream. It should then evaluate to a list of messages. If this list is exhausted then the process is "dead" (messages sent to it will have no effect).

The method suggested here for introducing the new address into the system may at first seem a rather curious one, the most obvious idea being that the new address should be sent to the parent process as a message. However, it turns out to be extremely useful for every process to know its own address, for two reasons. The first is that many interactions between processes involve a request and a reply, and each request of this type must include an address for the reply. The second is that it provides a way for a process to poll its own input stream, by sending a message to itself.

The list of all messages

```
messages :: [Message]
messages = MERGE (mapdyad fn process_bodies (2...))

fn :: Process_body -> Address -> [Message]
fn body addr = body addr (untag addr messages)

mapdyad :: (* -> ** -> ***) -> [*] -> [**] -> [***]
mapdyad fn (a : as) (b : bs) = fn a b : mapdyad fn as bs
```

This is a list of all messages ever sent in the system, in the order in which they are processed. It is constructed by merging together all the members of a list of lists of messages. Each of these lists is the output of a process. Each process is constructed by applying to a process body (which has been sent to process 1) a new, unique address and an input stream. `2...` is an infinite list of integers (i.e. the value `[2, 3, 4, 5, 6, ...]`) and is used as the generator of unique addresses. Each body is applied to its own address (i.e. a value taken from `2...`), and the list of data items from all messages that are addressed to it (this is what the `untag addr` achieves).

MERGE (read as "big merge") takes a list of lists as its input, and nondeterministically merges all elements of these lists together into a single list.

Other Possible Extensions

One of the pleasing things about this model is that a number of necessary extensions appear to fit in fairly well. The features described in this section have not yet been implemented.

Control over Scheduling

The current scheme runs on a single processor, and will run any process that is eligible. It is not possible to implement a background task, whose scheduling is under more precise program control.

One possible method for implementing background tasks would be to designate one process as "background", and only run it if nothing else can. Alternatively, some more complex scheme could be implemented in the underlying system giving different priorities to the different tasks, and including a scheduling algorithm based on these priorities.

A more powerful idea is to use the "engine" concept [Haynes 84]. A "background" process is created in a special way that marks it as such. It may only compute when a special "clock tick" message is sent to it. Other processes comprising the operating system scheduler decide when some time can safely be spent evaluating this background task and send a "clock tick" message to it when this is the case. This is effectively permission to perform a small amount of computation on this process, after which the scheduler should be run again and may decide what to do. This scheme allows more precise control, for instance it might be appropriate if multiple background tasks with complex priority functions were required.

Another aspect of this is the use of the same system on a machine with several processors. Although processes are able to pass objects (or functions) of any type between them, there is no reason why they should all be running on the same processor, or even in the same address space. By making the process creation operation more complex it should be possible to write a system that runs on a machine with several processors. The allocation of processors to processes could be explicit, or performed by the system. One of the most pleasing aspects of the scheme at the moment is how little it assumes about the computer on which it is being run.

Error Recovery

This is very tricky, as there are a number of ways in which a user program could go wrong.

The simplest case of this is a value error detected at runtime, such as an attempt to divide by zero. Saying that the result is "error" is not really adequate: some method is needed of telling the operating system that something has gone wrong, and allowing the system to decide what to do about it.

The next example is an infinite loop, or a long computation that the user realises that he did not want. The user signals impatience by hitting break, or signalling in some other way that the attempt to evaluate this object should be abandoned. In this case the task should be killed: how is this to be achieved? There seems a need for a "kill-process" primitive. (This is also necessary to tackle case 4 in the introduction, where nondeterministic behaviour is being used to perform a breadth-first search). This would be implemented as a "death" message, a special data object which, when sent to a process, removes that process from the system.

Even worse than this is a program that uses up all of store: an accidental attempt to reverse an infinite list will do this. The machine must decide to throw something away before it can allow any more functional programs to run, as functional programs always need some spare heap in order to compute. In this case it seems necessary to mark in some way which processes are "untrusted". If a store jam occurs, the machine throws away all untrusted processes and sends a message to some agreed location to say that this has occurred.

What if this does not release any store? A user program creates a large structure, and passes this to a trusted process, who unwittingly holds on to it. This would prevent the garbage collector from reclaiming it, and the machine would die for lack of store. Preventing this seems to require some care. One method for presenting this is to keep a small reserve of store, which the user cannot use up. This is only released when the machine is diagnosed as being full. However, this will cause a reduction in processing power (because heap based machines go faster if more free memory is available). Another idea is to control objects passed from an untrusted program, and ensure that they are all fully evaluated, and of controlled size. The user program is a function rather than a process body, and so rules like this can be successfully enforced.

It is worth noting that recovery seems even more complex from certain errors that involve multiple processes. For instance, recovery from a program that went wrong by generating thousands of superfluous processes until the machine filled up might be very hard. Perhaps this could be solved simply by making the processes created by an untrusted process also untrusted. More thought is needed in this area.

Most user programs are deterministic, but some may not be. For instance, the ability to run experimental versions of the operating system under the old one would be extremely desirable. This may require a more complex version of the

"trust" mechanism, but would almost certainly be worth it.

Operating System Design

A certain amount of work has been done on SKIM concerning the use of this scheme in the construction of a real operating system. The result is not an explosion of new ideas on how operating systems should be designed, but a clear impression that it is possible to do things that conventional operating systems are able to do, with no significant loss of performance. I believe that, although there is no solid evidence for this yet, such an operating system can be constructed using clearer and simpler programs than their conventional counterparts in a conventional operating system. If this is the case it will be a considerable victory for functional programming.

Process Control

The kernel of an operating system is frequently thought of as that part which controls processes and scheduling: if this is the case, to what extent has SKIM cheated by implementing this sort of thing in microcode? In answer to this it seems worth pointing out that the model of a process implemented here is very much less complex than the processes implemented in a conventional operating system. The processes in this scheme provide a basic mechanism for controlling parallel execution, and are unlikely to correspond to what the user thinks of as tasks being performed by the machine. Such ideas as who the process belongs to, what the process is called and what resources it is allowed to access must be organised by other programs. Many fundamental decisions concerning the structure of the operating system are still to be made, and will be expressed in a functional language.

The Filing System

There is no reason why a filing system should not be built along much the same lines as in conventional operating systems. At the lowest level programs would communicate with the disc through messages, which read and write sectors on the disc. Further layers on top of this could control file structure, the garbage collection of disc blocks, archiving, directories, version control and so on.

Alternatively, the storage on which the filing system is based could be available over a local area network. Using similar communication protocols the functional program can deal with such a situation in exactly the same way that a conventional machine on a network does.

It would also be possible to use the disc to implement a virtual memory system, which retains the integrity of the heap when the system is powered down. This

merely with streams of characters.

Part of the (highly successful) UNIX philosophy is to think of programs as tools, allowing many common tasks to be performed simply by combining existing programs using a functional style. This benefit is limited to programs that communicate using streams of characters, and is distinct from the provision of subroutine libraries for programming languages. By encouraging the use of a functional style at every level (i.e. in the writing of applications and tools, and in the command language) programming tools become even more general. We approach a state where, if a programming task has been solved, the solution of the problem is available for re-use at the command level and at the language level.

Discussion

Semantics

Constructing an adequate mathematical characterisation of programs involving merge is itself a subject of research [Brock 81, Park 82] as various problems concerning the behaviour of merge make it difficult to relate to more well-founded models of parallelism [Kahn 74, Milner 80]. Message passing has been described in this chapter in terms of merge, but represents a useful simplification because merge is only used in a highly constrained way: Park indicates that programs of the form

$$\begin{aligned}x_1 &= M_1 x_1 x_2 \dots x_n \\x_2 &= M_2 x_1 x_2 \dots x_n \\&\dots \\x_n &= M_n x_1 x_2 \dots x_n\end{aligned}$$

(where the M_i include occurrences of merge) are considerably easier to model mathematically than programs of the form

$$\begin{aligned}f_1 x_1 x_2 \dots x_m &= M_1 f_1 f_2 \dots f_n x_1 x_2 \dots x_m \\f_2 x_1 x_2 \dots x_m &= M_2 f_1 f_2 \dots f_n x_1 x_2 \dots x_m \\&\dots \\f_n x_1 x_2 \dots x_m &= M_n f_1 f_2 \dots f_n x_1 x_2 \dots x_m\end{aligned}$$

The latter form is needed if unconstrained use of merge is to be allowed, but message passing (even with process creation and deletion) requires only the former, as demonstrated by the Miranda programs used in this chapter to define the message passing system.

would allow objects to be "filed" simply by holding on to them as data structures: a large virtual memory would ensure that they eventually migrated onto disc, and the result is, to a far greater extent, a "functional" program. My instinct, however, is to be wary of this approach for the following reasons:

1. It demands far more implementation effort "beneath" the level of functional programming. It is the low level parts of operating systems whose semantics are most in doubt and where hidden bugs are most harmful: this is where functional programming is needed most!
2. It seems more liable to disaster from some hidden bug in the operating system. Merely discarding a pointer might discard the filing system, requiring recovery in some manner "outside" the operating system.
3. It merely delays the issue of how one is to communicate with a persistent outside world. Sooner or later the question of interfacing to networks, remote filing systems and other computers must be faced.

How a filing system *should* be represented in a functional world, and the application of such an approach to a networked world (with the functional program being distributed over the network), are still subjects of debate and research in the functional programming community. However, it is pleasing to note that the use of this scheme provides a wide range of choices, and that one is not limited by problems of efficient implementation.

The User Interface

The design of the user interface is not constrained at all by this scheme. For instance, if desired a shell program could be written, along the lines of the UNIX shell. This could take input from the keyboard and access the filing system in order to perform tasks specified by the user. If required, it could maintain a history of previous commands, and any other fancy features that are required. Alternatively a menu scheme could be used.

It is amusing to note that the commands given to a shell effectively form an imperative command language, and there is no reason why programs should not be written by the user in this language. Hardly functional programming! Somehow, it should be possible to find something a bit more functional: I welcome suggestions. When sitting at a terminal giving individual commands to a computer, the imperative paradigm really does seem the most appropriate. It may be that this tendency can be minimised by providing a menu-driven interface, and a good collection of basic commands.

Some thought concerning commands available under the UNIX shell reveals that many of them are *almost* functions: the construction of pipe expressions, for instance, is very much like functional programming. In the command language of a functional machine this style of programming would be heartily encouraged, with the considerable bonus that such a style would not be constrained to work

Type checking

A problem concerning type checking has been ignored in the previous descriptions. If the items taken as input by a process form a list then all objects sent to one particular process should be of the same type. This may be inconvenient in some circumstances, but it is still perfectly possible. However, a process can send messages containing data items of different types, provided that each one is of an appropriate type for its receiver. So what exactly is the type Message? Conventional type checking technology seems unable to describe it. Under a more advanced scheme an Address would be a type generator rather than a type, so that Addressfor * is a process address to which objects of type * can be sent.

Message == (Addressfor *, *) for some *, rather than for all *

```
; e.g. if   intad :: Addressfor Int
;         and  chrad :: Addressfor Char
;         then [(intad , 3), (chrad , 'a')] is a legal list
;                                               of messages
```

Current languages seem unable to describe this type

The type system described in [McQueen 84] can describe this type, but no mechanical checker for such types exists. SKIM's operating system is in fact written in Ponder [Fairbairn 84], a language with a type system that is somewhat more powerful than that of ML and Miranda [Milner 78], but still unable to describe the type Message. However, there are numerous places in the operating system where the type checking is sidestepped. It seems a shame that this should be necessary. There are also places where a new process is created and an old one discarded, simply to change the type of the input stream: it seems undesirable that the use of such a powerful facility should be necessary in order to achieve such a simple effect.

It's in a functional language, but is it functional programming?

The programs presented here have all been written in a functional language, yet they display behaviour that is not normally associated with such programs. Using processes "side effects" can easily be manufactured if desired, based on a process having a "state" which other processes can interrogate or update. How will this and other aspects of the message passing scheme affect the design of large programs? It may be that programmers will be tempted to use methods just as bad as in any conventional program: this remains to be seen.

An example of the use of message passing for stylistic reasons, rather than any requirement for parallelism, this can be found in the game program described on page 8.12. the function game in that program returns a tuple of type ([Out], [In], Score), and this can cause the entire [Out] list to exist at one time, choking the machine's store if the game performs a great deal of input and

output. This problem is reflected in the structure of the function game: its result is a tuple, yet in practice the whole of the first element of the tuple will be produced before any of the others. This suggests that a more natural structure for it would be that of a process, sending the Out objects and then the final Score as messages. This also solves the "stub" problem described in that chapter.

Functional programming is difficult to define precisely, and it may be that, although concerned with functional languages, this scheme really steps outside the realm of functional programming as it is generally accepted. Programs are still functions, but their structure is quite complex. Expressions under this scheme are still deterministic, but the scheme does more than just take the value of an expression. On the other hand, something has to be done if functional languages are to be used to describe and implement time-dependent and nondeterministic systems. This scheme represents an attempt to achieve just that.

Conclusions

This thesis has presented a number of new results in the field of functional programming, including a number of new implementation techniques, comprehensive suggestions concerning input, output and nondeterminism in functional languages and remarks (based on extensive experience) concerning the use of functional languages for writing large programs.

Building SKIM was definitely worthwhile, and formed the basis for a successful programme of research. Scale was important in achieving this. SKIM was sufficiently small to be built by one person in less than one year, and sufficiently cheap that special funding was not needed. Very little time was spent applying for money, writing reports, showing SKIM to sponsors and making presentations. No promises were made concerning results, and so I was able to work on whatever seemed interesting, without undue pressure. No attempt was made to build more SKIMs beyond the prototype, or to make the software sufficiently robust that it could be distributed to other people: while such things might be highly desirable, they also constitute a considerable amount of work. There are a number of ways in which SKIM could have been made faster, at the cost of additional complexity, but these temptations were resisted. The ease with which hardware and microcode were developed was due to the simplicity of the underlying architecture, and added complexities would have severely impeded this.

The early decision that has been regretted since is the choice of host processor. A BBC Microcomputer was used, with floppy discs and slow connections to an adjacent mainframe, where the microcode assembler and functional language compiler resided. This meant that software compilation turnaround was annoyingly slow, and ultimately grew to limit the development of large functional programs more than any defect in SKIM itself. Unfortunately no larger host machine was available to us, and the idea that SKIM was not dependent for its operation on local networks and so on was highly attractive. One of the aims of the project was to write enough system software for SKIM to be a self-sufficient, useful computer: a slow compilation cycle was one of the main reasons why this was never accomplished.

A preferable arrangement would be to have a powerful standalone micro or mini, with its own operating system and hard disc, on which all support software would be developed. Unfortunately such a course has its own associated hazards. One of the reasons why I made progress in working out the basis of an operating system for SKIM was that it *needed* an operating system. If a large mini were connected to SKIM the need for a functional operating system would recede, why not do all editing and compilation on the mini and just use SKIM for actually running the functional program? Doing research and building something 'useful' are aims that can conflict.

What directions should future research take? In my opinion the most important step that must now be taken by functional programmers is to gain a wider range of users of functional languages, people not interested in functional programming for its own sake but who will use it as a tool for other applications. To achieve this end there is a desparate requirement for a widely available, well engineered implementation of a functional language. The construction of such an implementation is not research, and it involves a good deal of hard work. By "well-engineered" I mean:

Acceptable performance, using the most recently developed compilation techniques [Wray 85, Hudak 85a]. Good performance on conventional machines is still a research issue.

The availability of the implementation on computers that are common within research institutions. I do not know what the best choice would be, probably the DEC VAX-11. Ideally the implementation should be available on a wide range of machines.

Acceptable compilation speed and error messages: very boring to achieve, but important to users.

A good functional language, including type checking and a modular library system for separate compilation. Well-defined standard libraries implementing the facilities described in chapters eight and nine.

Good runtime diagnostics and error messages. There has been very little work on this so far, but as stated in chapter six they are very important for serious functional language use.

Good documentation, in the form of a language manual, a beginner's guide to the language, and an undergraduate course based on the practical use of the language.

A promise of some degree of maintenance and support.

Readers who have been involved in compiler development will recognise that these points are not research issues. The implementors of functional languages have so far limited themselves to achieving better performance, but if a wider circle of functional programmers is to be encouraged then an implementation must be produced that is *attractive* to use. This is the only way that a wider range of opinions will appear on how functional languages should be used, and their strengths and weaknesses better understood.

There is also considerable scope for further work on architectural support for functional languages. Programming languages represent a compromise between what is useful and convenient for expressing algorithms and what can be implemented efficiently. Our ability to implement different models of computation should be constantly explored and expanded if the language designer is to be given greater freedom. Are functional languages really better than

conventional ones? I still find it very difficult to make a convincing case for this, i.e. a case that convinces *me*. So much of this whole area seems to rest on the "aesthetics" of programming and to be highly subjective, and so choosing a method with which to research it is very hard. It is quite possible that functional languages in their current form are not the best way to program: the only way to find a better one is to keep trying!

Appendix A

Implementing the Timeslicer

The SKIM machine is a microcoded combinator reducer. In addition to a reducer, the microcode also includes a timeslicing mechanism which implements the message passing scheme described in chapter nine. This appendix describes how it works at the machine level, and may be of interest to other implementors. The scheme is described for a combinator reducer, but is equally applicable to any other form of lazy evaluator.

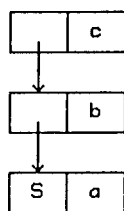
The Heap

SKIM's memory is organised as a heap of Lisp-like "nodes". Each node contains two "values", the "head" and the "tail" of the node. Values consist of a 20 bit "data" field and a 4 bit "tag" field. The tag can take one of the following values:

value is	abbreviation	data is
application	appl	heap pointer
pair	pair	heap pointer
process address	addr	heap pointer
integer	int	value
character	char	ASCII code
nil	nil	0
combinator	comb	microcode address

Types of value on the SKIM machine

For instance, the value S a b c is represented as:

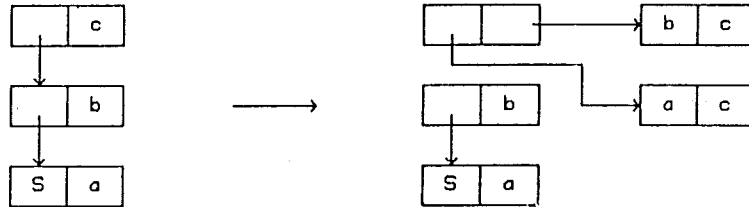


The representation of S a b c in SKIM's memory

The store is organised as a heap, and the garbage collector uses the tags on all values to determine which parts of memory are accessible and which are garbage. There are also other bits on each cell in memory which are used by the garbage collector to arrange this.

The Reducer

The bulk of SKIM's microcode is a normal order graph reducer. It uses pointer reversal to crawl over a graph of application pointers, which is continuously transformed until it will not reduce any further. For instance, if given the expression $S a b c$ the reducer would perform the following transformation:

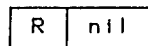


$S a b c$ is transformed into $a c (b c)$
 Conventional reduction of the S combinator

Thus the result of the reduction is $a c (b c)$, as we expect.

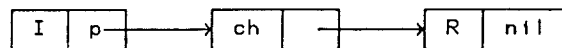
The Input Stream

In order to implement input streams, a special combinator called R (for "read") is used. The input list's value is initially represented as an application pointer to:



An input list's value is an application pointer to a cell like this

The keyboard system also keeps a pointer to this cell, which is known as the "readin" cell. When a key is pressed, this cell is overwritten like this:



→ is an application pointer

p → is a pair pointer

The readin value after a key has been pressed

The cell on the left originally contained an R

The I cell is necessary because the initial cell will have application pointers to it. The second cell is a pair cell, and the readin cell (where the next character will appear) has advanced to being the cell on the right.

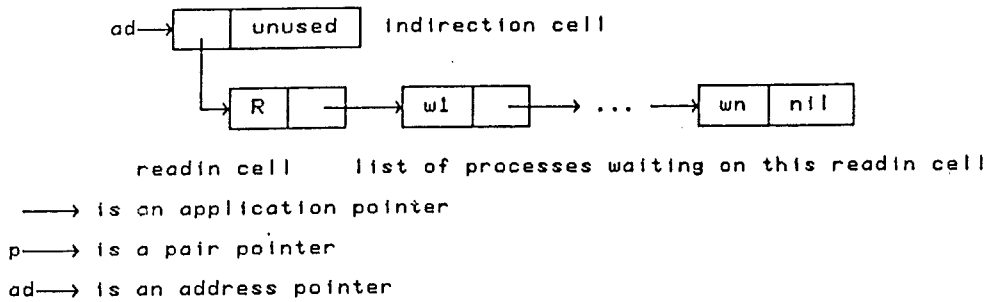
The Slicer Data Structures

The microcode that controls the scheduling of the processor is called the "slicer". It keeps a circular queue of "active" processes, i.e. those that may be evaluated. Each process is represented in this list merely by its value. Each process also has an input stream, at which messages sent to it appear. If an attempt is made to evaluate a readin cell then the running process will wait until something is sent

to that readin cell. The process is removed from the list of active processes and added to the list of processes waiting for something to arrive at this input cell.

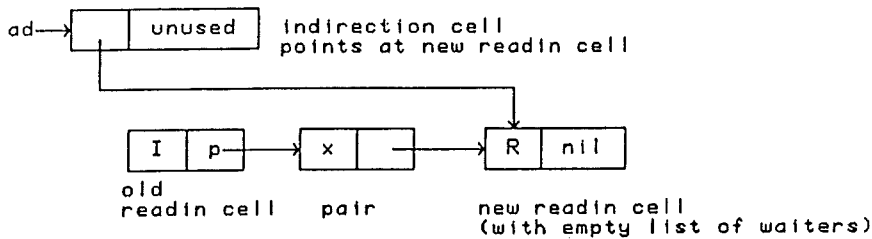
Note that the value of a process (i.e. the stream of messages output from the process) and its input stream are not bound together in any way. Because partially unevaluated messages can be sent, there is no reason why a process should not end up attempting to evaluate the input stream of another process, indeed this sometimes happens in real programs. An arbitrary number of processes can be suspended, waiting for input at one readin cell.

For this reason each input cell is represented by the following arrangement:



The full structure of a readin cell

The tail half of the read cell holds the list of those waiting for input at this cell. The process address is represented by pointers to an indirection cell. If a message with data item x is sent to this address, all the waiting processes will be added to the active queue and the readin structure will be turned into:



A readin cell after a message send

The Action of the Slicer

The slicer runs the reducer on the process on the head of the queue of active processes. The process should evaluate to a list of messages. It evaluates the first list cell, the cell representing the first message, the head of that cell (i.e. the destination address), and the tail of that cell (i.e. the message's data item). This action is performed until:

The process evaluates to nil:

The process is removed from the active queue: it is dead. The slicer continues by evaluating the next active process.

5000 reductions have been performed (i.e. an arbitrary time limit):

The slicer advances to the next member of the active queue. This is to prevent a process performing intensive computation from locking the others out.

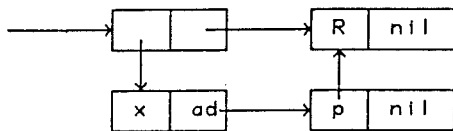
The evaluator attempts to evaluate an R combinator:

Further reduction of this value is dependant on a message arriving at this readin cell. This process is removed from the active queue and added to the list in the tail part of the R cell. The slicer continues by evaluating the next active process.

The process evaluates enough to produce a pair cell, with a well formed message in its head half:

This is equivalent to sending a message. The message will have two parts, a destination address (fully evaluated) and a data item (evaluated at top level). The process value in the active queue is overwritten with its own tail, so that futher reduction of this process will yield the next message in the list. Unless the destination address is a special address, the data item is added to the input stream indicated by the destination address, and all waiters are removed from the readin cell corresponding to that address and added to the active queue. The slicer proceeds by evaluating the next active process.

Special destination addresses include any devices, and the process creator. The action for devices is device dependant but not hard to decide upon. Note that the evaluator may have to poll input devices and stop so that they can send their input messages in some way. The action for the process creator, when sent a process body x , is to add the following structure, as a process to be evaluated, to the active queue.



The structure of a new process using process body x

This is the value x with two arguments applied to it: a newly created address, and a newly created input stream.

Discussion

The mechanism has a pleasing feel of being minimal in some sense: there are very few arbitrary choices in its design, and no special cases or features. It is reasonably fast, and has no gradual space leak. However, there are some points that need more thought:

1. The SKIM reducer reverses pointers while reducing an expression, and a process switch requires any existing chain of pointers to be re-reversed, so that the new running process does not fall foul of the reversed pointers of the other processes. Thus some task switches may take rather a long time. This happens when waiting for a message, and when the time limit comes into effect. Removing this overhead seems quite hard, and would probably need considerable changes to the reduction method.

One possibility is to mark in some way cells that have reversed pointers in them (at the moment it is not possible to detect them). If a process meets a reversed value then it and another process both need some common subvalue, but the other process has already started evaluating it. So add a "branch" to the reversed chain and suspend the current process, in such a way that it is restarted when the other process un-reversed this list. In a conventional system this is unlikely to be worth the effort, but arrangements of this sort might be quite useful in a multi-processor machine.

2. Ensuring fairness needs a certain degree of care and compromises response time. This is the requirement that no combination of processes should be able to grab all CPU time by sending messages to each other. When choosing which process to run next, it is best to choose one which has not been involved in the last transaction, in case two processes (by furiously sending messages to each other) manage to lock all the others out. This means that, when a message is sent, the activated process should not be the next one to run: even if it is probably the one which should run next if response time to devices is to be improved.

Appendix B

The Language Miranda

Miranda [1] [Turner 85] is a functional language. It is strongly related to SASL [Turner 76] and KRC [Turner 82]. This guide is intended not as a complete language reference manual, but as an aid to those seeking to understand the programming examples in this thesis: because of this, many of the more advanced features of the language have been omitted.

Programming on SKIM was done in Ponder [Fairbairn 85], a polymorphically typed language developed at Cambridge University. Miranda was chosen for written examples in this thesis because it is more succinct.

Functions

The main operation for combining values in Miranda is function application, which is denoted by juxtaposition. So, the expression

```
fn x
```

has a value that is obtained by applying the function `fn` to the value `x`. Function application associates to the left, and brackets may be used to indicate grouping. So,

```
plus x 3
```

is the same as

```
(plus x) 3
```

and denotes `x` added to 3 (assuming that is what the function `plus` does). All functions are fully curried, i.e. arguments may be applied to them one at a time. Functions may be used as first class data objects. The value that a function yields is uniquely defined by the arguments passed to it, and two textually identical expressions (in the same context) will always yield the same value. There are no "side effects" or assignment statements.

Primitive Data Types

Miranda closely resembles the lambda calculus in its semantics, but in order to make simple operations easier to understand some primitive data objects have been added. These are characters, integers, tuples and lists. Character constants

[1] 'Miranda' is a trademark of Research Software Ltd.

can be included in a program as the intended character in single quotes. Integer constants are a sequence of digits, with an optional leading minus sign. Tuples are written using infix commas within round brackets.

'A' 'B' '\$'	character constants
23 128 15	integer constants
('a', 23)	a tuple consisting of a character and an integer
(1, 2, 3)	a tuple consisting of three integers

A list may be written as a sequence of elements separated by commas and enclosed by square brackets, or using an infix colon. The null list is written as [].

[1, 2, 3, 4]

is equivalent to

1 : 2 : 3 : 4 : [].

In addition, string quotes may be used to denote a list of characters. So,

"hello"

is a shorthand for

'h' : 'e' : 'l' : 'l' : 'o' : []

There are a number of predefined primitive functions and infix operators in the language. For instance,

+ - * /	simple arithmetic operators
hd tl	operations to extract the left and right parts of a tuple
++	append lists

Function Definition Clauses

A function is defined by a sequence of one or more "clauses". For instance,

double x = x + x

is a clause which gives a value to the name double. A clause consists of the name to be defined, followed by zero or more arguments, followed by an equal sign, and then the corresponding expression. Thus, double can be thought of as the lambda expression

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

Arguments can be patterns instead of simple names, indicating that "this clause is only appropriate if the actual argument matches the pattern". This can lead to the need for more than one clause in the definition of a function. For instance,

```
append (a:x) y = a : append x y
append [] y = y
```

This function `append` appends two lists together. The first clause only applies if the first argument to `append` is not `nil`: if this is the case, the body of the clause is the result of the call to `append`, using `a` and `x` to denote the head and tail of the first argument. The second clause only applies if the first argument is `nil`. When evaluating a call to `append`, each clause is tried in turn until one matches: this is then taken to define the value returned by the call.

Guard clauses add to the generality of this device. A clause can be of the form:

```
fn <sequence of patterns> = <expr>, <guard>
```

where the guard is a boolean expression. The clause only applies if all the patterns match, and the guard clause is true. This is the usual form of the conditional in Miranda programs:

```
factorial x = 1, x < 2 ; If x < 2 Then 1 Else ...
             = x * factorial (x - 1)
```

Note also that repeated patterns can be omitted, and that recursion may be used freely. The compiler learns much of the structure of expressions like this from an *offside rule* used by the syntax analyser: an expression whose first lexeme appears in column `n` of the program source will terminate on the next line that is not blank in columns 1 to `n-1`. So, the compiler can successfully distinguish the following two functions, even though the lexemes in each are identical:

```
fn1 a b c = E1, a = b ; If a = b Then E1 (else error)

fn2 a b = E1, a ; If a Then E1
             = b ; Else b.
```

The `WHERE` construct takes the form

```
<expression> WHERE <sequence of clauses>
```

and allows a large expression to be broken up into a number of smaller ones. For example,

```
fn a b c = (x1 / quot, x2 / quot)
           WHERE x1 = - b + d
                 x2 = - b - d
                 quot = 2 * a
                 d = sqrt (b * b - 4 * a * c)
```

Types

Miranda uses a static type discipline very similar to that of ML. The compiler is able to deduce the types of the various objects used in the program, with no extra information being supplied by the programmer. The language also provides facilities for user defined types. The double colon symbol means "is of type", and can be used as compiler-checked documentation of the types of various objects.

```
plus :: Int -> Int -> Int

id x = x
id :: * -> *

map :: (* -> **) -> [*] -> [**]
map fn [] = []
map fn (a:x) = fn a : map fn x
```

The `->` symbol indicates a function type. Asterisks are used to indicate that any type is appropriate, so for example `id` is a function that maps an object of any type onto an object of the same type. The type of `map` is more complex but indicates precisely the range of values for which the function `map` defined above is applicable: it needs two arguments, a function and a list, and the argument to the function must be of the same type as the elements of the list. The result is another list, of which the elements are the same type as the results of calls to the argument function.

A convention that I adopt, but which is not part of Miranda, is that types are indicated by capitalised words (in Miranda `num` and `char` are reserved words, I prefer to use `Int` and `Char`). The double equal sign is used to introduce type synonyms. Commas and square brackets are used to indicate tuple and list types, in a manner analogous to the appearance of tuple and list values.

```
Int == num
Char == char
String == [Char]
```

Type *generators* can be defined in a manner analogous to the definition of functions.

```
Listof * == [*]
Matype * ** == (* -> **) -> Listof * -> Listof **
```

There are some minor discrepancies between the up-to-date version of Miranda described in [Turner 85] and the variant used in this thesis, these are:

- || rather than ; is used to introduce a comment.
- where is used instead of WHERE.
- an identifier may not start with a capital unless it is a constructor.

The ::= symbol is used to introduce tagged union types. For instance, the definition

```
Tree * ::= Node * (Tree *) (Tree *) | Leaf *
Colour ::= Red | Green | Gold
Person ::= Personrec String String Int Int
```

The first line defines a *Tree* type with two possible *constructors*, *Node* and *Leaf*. These can be used in expressions to build a *Tree* object, or in patterns to match one. *Colour* is defined using the same construct, and corresponds closely to a Pascal enumeration type. *Person* is a record with only one possible form, containing four fields.

There are some minor discrepancies between the up-to-date version of Miranda described in [Turner 85] and the variant used in this thesis, these are:

- || rather than ; is used to introduce a comment.
- where is used instead of WHERE.
- an identifier may not start with a capital unless it is a constructor.

Appendix C

A Functional Program

This appendix contains parts of a compiler for a KRC-like language, which was developed and run on SKIM. It was written in Ponder and demonstrates the sort of programs that I was attempting to work with. In order to keep the appendix to a manageable size only the lexical and syntax analysers are included, comprising slightly less than half of the entire compiler.

Ponder

Ponder [Fairbairn 84] is a simple functional language based on the normal order lambda calculus. Its structure is considerably simpler than Miranda's and its type system somewhat more powerful. [1]

```
34 'x 'y "hello"      -- double-underscore signifies a comment
''n''''''           -- integer, character and string literals
a                   -- newline, double and single quote character escapes
a b                 -- lower case identifiers denote expressions
a + b      a And b  -- juxtaposition denotes function application
                   -- there are various dyadic operators
                   -- capitalised identifiers are operators,
                   -- brackets, keywords or type generators.
- 3                 -- monadic operators are also allowed.
Begin a + b End     -- and bracket operators.
a * (b + c)         -- ( and ) denote grouping.
Let a == b; c       -- ( $\lambda a. c$ ) b
Letrec a == b; c    -- ( $\lambda a. c$ ) (Y ( $\lambda a. b$ ))
Let a, b == c; d    -- ( $\lambda a. (\lambda b. d)$  (right c)) (left c)
Int x -> expr       -- ( $\lambda x. expr$ ), x is an integer
Char: expr          -- an expression of type Char and value expr.
                   -- the Char: merely confirms this
Int Char String Bool -- common types
Int -> Int           -- the function application type constructor
Pair[Int, Char]     -- a common type constructor
3, 'x               -- an expression of type Pair[Int, Char]
Typeinfix >< == Pair; -- a type operator declaration,
                   -- 3, 'x is also of type Int >< Char
Type String==List[Char]; -- a type declaration
!X. List[X] -> List[X] -- the type of the list reverse function.
                   -- ! means "for all", replaces * in Miranda.
```

[1] A version of the compiler that runs under VAX UNIX can be obtained by writing to Fairbairn.

```

Capsule Type X == ...;  __ abstract type declaration,
...                    __ outside the capsule access to the type
Seal X;                __ can only occur through the functions defined
                        __ in the capsule.
Infix + == ...;       __ defining an operator, overloading is allowed.
Priority 5 And Associates Right;  __ priority and association of operators
                                __ can be specified.

```

In addition to the code reproduced here a number of library files must be used when compiling this program. These implement commonly used functions, types and operators.

```

-- *****
--
--      Syntax Tree Format and Pretty Printing.
--
-- *****

```

```

Capsule Rectype Expr ==
    String \./                __ bound variable name
    (String >< Expr >< Expr) \./  __ binary node (diop/apply)
    (String >< Expr) \./        __ unary node (monop/bracket)
    String \./                __ un-typechecked value
    String \./                __ error message
    Dot;

```

```

-- \./ and Dot are defined in a library,
-- and implement the union of a collection of types. The same package
-- defines Is_Type, in_1, is_1 and so on as used
-- in the following extraction and insertion functions:

```

```

Let make_expr_bvar == String          -> Expr: in_1;
Let make_expr_bnode== (String >< Expr >< Expr)  -> Expr: in_2;
Let make_expr_unode== (String >< Expr)         -> Expr: in_3;
Let make_expr_unkn == String          -> Expr: in_4;
Let make_expr_err  == String          -> Expr: in_5;

```

```

Let expr_is_bvar == Is_Type[String,          Expr]: is_1;
Let expr_is_bnode== Is_Type[(String >< Expr >< Expr), Expr]: is_2;
Let expr_is_unode== Is_Type[(String >< Expr),   Expr]: is_3;
Let expr_is_unkn == Is_Type[String,          Expr]: is_4;
Let expr_is_err  == Is_Type[String,          Expr]: is_5;

```

```

-- Expressions can only be built and accessed via these functions,
-- which are often used in conjunction with the
-- Case ... In ... *> ... [|| ... *> ...]* Out ... Esac
-- construction. The fact that a construction of this sort can be defined
-- in Ponder rather than having to be built in is a tribute to the expressive
-- power of this language.
-- (Case and Esac are brackets; In, || and Out are operators.

```

```

-- Suitable use of capsules ensures that these can only be used in the form
-- intended by the author of this package.)
-- The same is also true of the
-- If ... Then ... [Elif ... Then ...]* Else ... Fi
-- construct.

```

```

Seal Expr;

```

```

-- dyadic operators end up as bnode(op, arg1, arg2)
-- monadics and brackets are unode(mop, arg)
-- function application is bnode("", arg1, arg2)
-- lambda expression is bnode("->", name, body)
-- Everything is defined in these terms, and then the code generation pass
-- gives the special meanings to all operators like "," and "=".
-- This yields a tree in the same format, but containing only
-- symbols and function applications.
-- A final pass can convert this into the corresponding value.
-- Any errors are appended to the tree, and extra passes during the compilation
-- process check for any such.

```

```

-- Prettyprinting expressions.

```

```

-- @ is the list append operator.

```

```

Let indent == String s -> String:

```

```

  If head s = '+ Then s Else " " @ s Fi;

```

```

-- A function to remove leading space from a string.

```

```

-- dropwhile, space and monadic = are defined in a library.

```

```

Let dropspace == String -> String: dropwhile (= space);

```

```

Letrec do_to_last == !T. (T -> T) fn -> List[T] l -> List[T]:

```

```

  -- A simple utility function that applies fn to the last element of l.

```

```

  If null l Then nil

```

```

  Elif null (tail l) Then lis (fn (head l)) -- lis means cons with nil

```

```

  Else head l :: do_to_last fn (tail l) -- :: is list constructor

```

```

  Fi;

```

```

Letrec pexpr_open == Expr expr -> List[String]:

```

```

Begin

```

```

  Let pexpr_closed == Expr expr -> List[String]:

```

```

    Case expr

```

```

      In expr_is_bnode *> ((String >< Expr >< Expr) sab ->

```

```

        (Let p == pexpr_open expr;

```

```

          If length p = 1 Then lis ("(" @ dropspace (head p) @ ")") Else

```

```

            ("(" @ dropspace (head p)) :: do_to_last (c append ")") (tail p)

```

```

          Fi))

```

```

      Out pexpr_open expr

```

```

    Esac;

```

```

  Case expr

```

```

    In expr_is_bvar *> (String s -> lis (' :: s))

```

```

|| expr_is_unkn *-> (String s ->
  (Let x == type_change s;
    Lis ( __ to make it List[String]
      '  ::
      If repr_is_int x Then printdec x           __ int literal
      Elif repr_is_char x Then ''' :: (Char:x)   __ char literal
      Else "" @ (String: x @ "") __ give up, probably string
      Fi)))
|| expr_is_err *-> (String mes -> lis ("+++Error: " @ mes))
|| expr_is_bnode *-> ((String >< Expr >< Expr) sab ->
  (Let s, a, b == sab;
    Let s == If s = "" Then " " Else " " @ s @ " " Fi;
    Let left == pexpr_open a;
    Let right == pexpr_closed b;
    Let map_indent == If s = " ; " Then i Else map indent Fi;
    If length left = 1 And length (head left) < 50 Then
      Let left == dropspace (head left);
      If length right = 1 Then
        lis (left @ s @ dropspace (head right))
      Elif length (head right) < 20 Then
        (left @ s @ dropspace (head right)) :: map_indent (tail right)
      Else (left @ s) :: map_indent right
      Fi
    Else do_to_last (c append s) left @ map_indent right
    Fi))
|| expr_is_unode *-> ((String >< Expr) sa ->
  (Let s, a == sa;
    Let a == pexpr_open a;
    (s @ " " @ head a) :: map_indent (tail a)))
Out lis "???"
Esac
End;

Let pexpr == Expr e -> String:
  gather (append_with_char newline) nil (pexpr_open e);

Let apply == Expr a -> Expr b -> make_expr_bnode ("", a, b);
Priority 5 Applied_to Associates Left;
Infix Applied_to == apply;
Let error == String s -> Expr: make_expr_err s;

-- *****
--
--   Lexemes.
--
--   Lexical analysis turns an input stream of characters
--   into a stream of Lexemes.
--
-- *****

```



```

Capsule Type Lex == Expr \./           __ literal (int, char or string)
                  String \./           __ alphanumeric name or punctuation
                  String \./           __ error message
                  Dot;

Let make_lex_expr == Expr      -> Lex: in_1;
Let make_lex_item == String    -> Lex: in_2;
Let make_lex_err  == String    -> Lex: in_3;

Let lex_is_expr   == Is_Type[Expr, Lex]: is_1;
Let lex_is_item   == Is_Type[String, Lex]: is_2;
Let lex_is_err    == Is_Type[String, Lex]: is_3;
Seal Lex;

Let plex == Lex l -> String:
Case l
  In lex_is_expr *> pexpr
  || lex_is_item *> psimple "[":
  || lex_is_err  *> psimple "[err:"
  Out abort
Esac;

-----
--
--      Lexical Analysis.
--
-----
--
--      No KRC-style offside rule is used, the input is free-format.
--

Capsule Type Lex_List == List[Lex];

__ 0 is function composition
Let plex_list == gather append nil 0 map plex;
Let eof_lex == make_lex_err "unexpected end of input";

Let is_namechar == Char c -> is_letter c Or is_digit c Or c = '_';

Letrec lookup_front == List[String] table -> String srch -> String >< String:
  If null table Then nil, srch
  Elif string_eq_front (head table) srch Then
    split (length (head table) + 1) srch
  Else lookup_front (tail table) srch
Fi;

Let doubletons == "==" :: "->" :: ":@" ::
                 "<=" :: ">=" :: "~=";

Let singletons == "! &()=~-^\\|@'[{]}*+;/?.>,<";

```

```

Let literal == !T. T t -> Lex:
  Begin make_lex_expr (make_expr_unkn (String: type_change t)) End;

Let isn't_newline == Char c -> Not (c Is_in newline_string);

Let char_escape == String s -> Bool >< Char >< String:
  __ handling character escapes in strings.
  __ Ponder-style string escapes are implemented.
  If null s Then yes, ''x, ""
  Else Let ch1, s1 == head s, tail s;
        Let ch2, s2 == head s1, tail s1;
        If ch1 ~=''' Or null s1 Then no, ch1, s1 Else
          no, If ch2 = '\n' Then '\n'
              Elif ch2 = '\e' Then '\e'
              Else ch2
                Fi, s2
          Fi
        Fi;

Letrec get_string == String s -> String >< String:
  If null s Then nil, s
  Elif head s = ''' Then nil, tail s Else
    Let eof, ch, s1 == char_escape s;
    If eof Then "", s Else
      do_to_left (list ch) (get_string s1)
    Fi
  Fi;

Letrec lex == String in -> Lex_List:
  If null in Then nil
  Elif head in Is_in " \n" Then lex (tail in)
  Elif is_letter (head in) Then
    Let str, in1 == read_item is_namechar in;
    make_lex_item str :: lex in1
  Elif string_eq_front "_" in Then
    lex (dropuntil (= newline) in)
  Elif is_digit (head in) Then
    Let str, in1 == read_item is_digit in;
    literal (readdec str) :: lex in1
  Elif head in = ''' Then
    Let eof, ch, in1 == char_escape (tail in);
    If eof Then lis eof_lex Else literal ch :: lex in1 Fi
  Elif head in = ''' Then
    Let str, in1 == get_string (tail in);
    literal str :: lex in1
  Else
    Let str, in1 == lookup_front doubletons in;
    If null str Then __ not a doubleton

```

```

    If head in Is_in singletons Then
        make_lex_item (lis (head in)) :: lex (tail in)
    Else
        make_lex_err ("illegal input ..." @ (first 20 in)) :: lex (tail in)
    Fi
Else
    make_lex_item str :: lex in1
Fi
Fi;

-- we may later include other lexemes that show_lex will make use of,
-- but which will be discarded by lexhead and lextail.
Let show_lex == Lex_List l -> __ for error messages
    " at ..." @ first 30 (plex_list l);

Let lexhead == Lex_List l -> Lex: If null l Then eof_lex Else head l Fi;
Let lextail == Lex_List l -> Lex_List: If null l Then nil Else tail l Fi;
Let lexnull == Lex_List l -> Bool: null l;

Let lx == plex_list 0 lex;

Seal Lex_List;

-- *****
--
-- Syntax Analysis and Tree building.
--
-- *****

-- recursive descent needs a collection of mutually recursive functions.
-- expr ::= term OP term           for various ops
-- term ::= item item item         items applied to each other
--           MOP term              for various monadic operators
-- item ::= name                   variable
--           literal               int, string or char
--           BRA block KET         for various bracket systems
-- A tree is built up of type Expr.

Type Parse == Lex_List -> Expr >< Lex_List;

Let check_for == String s -> Lex_List in -> String >< Lex_List:
    __ produces complaint, restofin
    Begin Let complain == ("expecting " @ s @ show_lex in), in;
        If lexnull in Then complain Else
            Case lexhead in
                In lex_is_item *>
                    (String n -> If n = s Then nil, lextail in Else complain Fi)
            Out complain
        Esac

```

```

Fi
End;

-- There's a library package that defines
-- Type Table[X] == List[String >< X];
-- and various functions that operate on this (e.g. lookup, update).

Let function_operators == Table[Int >< Bool]: -- prio and left assoc
  -- special syntax for simple function calls
  ("O",      (5, yes)) :: -- function composition
  ("*",      (5, yes)) ::
  ("Div",    (5, yes)) :: -- yields quot.rem
  ("Quot",   (5, yes)) :: -- quotient
  ("Rem",    (5, yes)) :: -- remainder
  ("+",      (6, yes)) ::
  ("-",      (6, yes)) ::
  ("Shift",  (6, yes)) ::
  ("&",      (6, yes)) :: -- bitwise and
  ("|",      (6, yes)) :: -- or
  ("Xor",    (6, yes)) :: -- xor
  ("Eq",     (8, yes)) :: -- like = in comparisons (not very nice)
  ("And",    (9, yes)) ::
  ("Or",     (9, yes)) ::
  ("~=",    (8, yes)) ::
  (">",     (8, yes)) ::
  ("<",     (8, yes)) ::
  ("<=",    (8, yes)) ::
  (">=",    (8, yes)) ::
  ("@" ,    (8, no)) ::
  (":",     (8, no));

Let syntax_operators == Table[Int >< Bool]:
  ("=",     (11, no)) :: -- NOT an equality test
  ("Else",  (11, no)) ::
  ("",      (10, no)) ::
  (">" ,    (8, no)) ::
  (";" ,    (12, no));

Let operators == function_operators @ syntax_operators; -- order unimportant
Let minprio == 5;
Let maxprio == 13;

Let brackets == Table[String]:
  ("(", ")") ::
  ("{" , "}"); -- optional alternative brackets for pattern tests

Let monops == Table[Int]:
  lis ("Not", 1);

```

```

Let block_gen == (Int -> Parse) op_expr -> Lex_List in -> Expr >< Lex_List:
  op_expr maxprio in;

Letrec op_expr_gen == Parse term ->
Int priority -> Lex_List in -> Expr >< Lex_List:
  __ does operator precedence stuff.
  Begin
    Let op_expr == op_expr_gen term;
    Let leftpart, in1 ==
      If priority < minprio Then term Else op_expr (priority - 1) Fi in;
    Letrec op_expr_leftpart == Int priority -> Expr leftpart ->
      Lex_List in1 -> Expr >< Lex_List:

    Case lexhead in1
    In lex_is_item *> (String op ->
      (Let op_priority, left_assoc == (999, no) ? lookup op operators;
      __ lookup returns a value of type Option[T], the ?
      __ provides the default value.
      If left_assoc And op_priority <= priority Then
        Let rightpart, in2 == op_expr (op_priority - 1) (lextail in1);
        op_expr_leftpart op_priority
          (make_expr_bnode (op, leftpart, rightpart)) in2
      Elif op_priority <= priority Then
        Let rightpart, in2 == op_expr op_priority (lextail in1);
        make_expr_bnode (op, leftpart, rightpart), in2
      Else leftpart, in1
        Fi))
    Out leftpart, in1
  Esac;
  op_expr_leftpart priority leftpart in1
End;

Let bracket_item == Parse block -> Lex_List in -> Expr >< Lex_List:
  Begin
    Let blk, in1 == block (lextail in);
    Let open == Case lexhead in In lex_is_item *> i Out "< open >" Esac;
    Let close == "< close >" ? lookup open brackets;
    Let err, in2 == check_for close in1;
    If null err Then make_expr_unode (open, blk) Else error err Fi, in2
  End;

Letrec term_gen == Parse block ->
Lex_List in -> Expr >< Lex_List:
  __ juxtaposition denotes function application.
  Begin
    Let term == term_gen block;
    Let op_expr == op_expr_gen term;
    Let bracket_item == bracket_item block;

    Letrec get_items == Lex_List in ->

```

```

List[Expr] >< Lex_List:
-- function application is left associative.
-- We build a normal list, ie "a b c d" -> d :: c :: b :: a
-- This gets reversed and tail-folded afterwards.
Begin
  Let recurse ==
    ((Expr >< Lex_List) ein -> List[Expr] >< Lex_List:
      Begin Let lis, in == get_items (right ein);
        ((left ein) :: lis), in
      End);
  If lexnull in Then nil, in Else __ legitimate end of file
  Case lexhead in
    In lex_is_expr *> (Expr e -> recurse (e, lextail in)) __ literal
    || lex_is_item *> (String name ->
      __ could be an open bracket
      (Let br == "" ? lookup name brackets;
        If Not null br Then recurse (bracket_item in)
        Elif is_lowercase (head name) Then
          recurse (make_expr_bvar name, lextail in)
        Else nil, in
        Fi))
    || lex_is_err *> (String err -> (Lis error err, lextail in))
  Out Lis error ("bad item " @ show_lex in), lextail in
  Esac
  Fi
End;

Letrec head_link == List[ Expr ] l -> Expr:
  If null l Then error ("missing expression " @ show_lex in)
  Elif null (tail l) Then head l
  Else apply (head_link (droplast 1 l)) (head (reverse l))
  Fi;

__ handle monadic operators.
Let no_monop == (Let items, in1 == get_items in;
  head_link items, in1);
Case lexhead in
  In lex_is_item *> (String s ->
    Begin
      Let prio == lookup s monops;
      If null prio Then no_monop Else
        Let arg, in1 == op_expr ((Int:abort) ? prio) (lextail in);
        make_expr_unode (s, arg), in1
      Fi
    End)
  Out no_monop
  Esac
End;

```

```

-- *****
-- Now to bind all the mutually recursive stuff together...
Letrec block == Lex_List -> Expr >< Lex_List:
  block_gen (op_expr_gen (term_gen block));

Let syn == pexpr 0 left 0 block 0 lex;

```

Performance

With the compiler loaded SKIM has about 111K cells free (out of 128K). Some measurements were taken of the performance of the compiler when given the following small program as input:

```

map fn (a:b) = fn a : map fn b;
map fn nil   = nil;
plus a b = a + b;
map (plus 2) (3:4:nil)

```

The lexical analysis of this expression takes about 0.3 seconds, consuming 14,000 cells. The syntax analysis takes a further 1.2 seconds, consuming 35,000 cells. The rest of the compilation takes 1.5 seconds and consumes 61,000 cells. This compiles the program into combinators and executes it, yielding the result (5:6:nil). All of the conventional combinator compilation optimisations are implemented, so that for instance map is compiled into

```

Y (C (B' C (S' U')) (S' C (B' B' :))) (K I)).

```

References

- [Abdali 76]: S. Kamal Abdali,
An abstraction algorithm for combinatory logic,
pp. 222-224 of *The Journal of Symbolic Logic*, Volume 41, Number 1,
March 1976.
- [ACM 81]:
Proceedings, 1981 ACM Conference on Functional Programming Languages
and Computer Architecture, 1981.
- [ACM 82]:
Proceedings, 1982 ACM Symposium on Lisp and Functional Programming,
August 15-18, 1982.
- [ACM 84]:
Proceedings, 1984 ACM Symposium on Lisp and Functional Programming,
Austin, Texas, 3-6 August 1984.
- [Aspenäs 85]:
Proceedings,
Aspenäs Workshop on Functional Language Implementation,
Chalmers University of Technology,
S-412 Göteborg, Sweden, 4-6 February 1985.
- [Augustsson 84]: Lennart Augustsson,
A compiler for Lazy ML,
pp. 218-227 of [ACM 84].
- [Backus 78]: John Backus,
*Can programming be liberated from the von Neumann style?
A functional style and its algebra of programs.*
Communications of the ACM, Volume 21, Number 8, August 1978.
- [Barendregt 81]: H. P. Barendregt,
The lambda calculus: its syntax and semantics,
Elsevier, New York, 1981.
- [Barendregt 85]: H. P. Barendregt, personal communication.
- [Brock 81]: J. D. Brock, W. B. Ackerman,
Scenarios: a model of nondeterminate computation,
International Colloquium on Formalisation of Programming Concepts,
Lecture Notes in Computer Science 107, pp. 252-259, April 1981.

- [Church 41]: A. Church,
The calculi of lambda-conversion,
Annals of mathematical studies #6, Princeton University Press, 1941.
- [Clarke 80]: T. J. W. Clarke, P. J. S. Gladstone,
C. D. MacLean and A. C. Norman,
SKIM—the S, K, I reduction machine,
Proceedings of the 1980 ACM Lisp Conference, pp. 128-135, August 1980.
- [Clarke 84]: T. J. W. Clarke,
A preliminary study of fast hardware support for combinator reduction,
Computer Science Diploma,
Cambridge, July 1984.
- [Cousineau 85]: G. Cousineau, P-L. Curien, M. Mauny,
The Categorical Abstract Machine,
[Aspenäs 85].
- [Curry 68]: H. B. Curry, R. Feys,
Combinatory logic, volume 1,
North-Holland, 1968.
- [Darlington 81]: John Darlington and Mike Reeve,
*ALICE: A multi-processor reduction machine
for the parallel evaluation of applicative languages*,
pp. 65-74 of [ACM 81].
- [Darlington 82]: J. Darlington, P. Henderson and D. A. Turner (eds),
Functional programming and its applications,
Cambridge University Press, 1982.
- [Darlington 82a]: J. Darlington,
Program transformation,
pp. 193-215 of [Darlington 82].
- [Duce 84]: D. A. Duce(ed),
Distributed computing systems programme,
Peter Peregrinus Ltd.,
IEE Digital Electronics and Computing series 5, 1984.
- [Fairbairn 84]: J. Fairbairn,
A new type checker for a functional language,
Cambridge University Computer Laboratory Technical Report #53, 1982.
- [Fairbairn 85]: J. Fairbairn,
*The design and implementation of a simple typed language
based on the lambda calculus*,
Ph. D. Thesis, Cambridge, 1985.

- [Gordon 78]: M. Gordon, R. Milner, L. Morris, M. Newey, C. Wadsworth,
A metalanguage for interactive proof in LCF,
pp. 119-130 of [POPL 78].
- [Goto 74]: E. Goto,
Monocopy and associative algorithms in an extended lisp,
University of Tokyo, Japan, May 1974.
- [Haynes 84]: C. T. Haynes, Daniel P. Friedman,
Engines build process abstractions,
pp. 18-24 of [ACM 84].
- [Henderson 76]: P. Henderson, J. H. Morris,
A lazy evaluator,
pp. 95-193 of [POPL 76].
- [Henderson 80]: P. Henderson,
Functional programming: application and implementation,
Prentice-Hall International, 1980.
- [Henderson 82]: P. Henderson,
Purely functional operating systems,
pp. 177-189 of [Darlington 82].
- [Henderson 84]: P. Henderson, S. B. Jones,
Shells of functional operating systems,
pp. 290-298 of [Duce 84].
- [Holmström 81]: Sören Holmström,
*A simple and efficient way to handle large datastructures in
applicative languages*,
pp. 185-187 of [UCL 83].
- [Hudak 84]: Paul Hudak, David Kranz,
A combinator-based compiler for a functional language,
pp. 121-132 of [POPL 84].
- [Hudak 84a]: Paul Hudak and Benjamin Goldberg,
Experiments in diffused combinator reduction,
pp. 167-176 of [ACM 84].
- [Hudak 85]: Paul Hudak and Adrienne Bloss,
The aggregate update problem in functional programming systems,
[POPL 85].
- [Hudak 85a]: Paul Hudak and Jonathan Young,
A set-theoretic characterisation of function strictness,
draft form, Yale University, 1985.

- [Hughes 82]: R. J. M. Hughes,
Super combinators: a new implementation method for applicative languages,
[ACM 82].
- [Hughes 84]: R. J. M. Hughes,
Parallel functional languages use less space,
Programming Research Group, Oxford University, 1984.
- [Hughes 85]: R. J. M. Hughes,
Untitled presentation on the subject of arrays, [Aspenäs 85].
- [INMOS 84]:
IMS T424 Transputer preliminary data sheet,
INMOS Limited,
Whitefriars, Lewins Mead, Bristol BS1 2NP, 1984.
- [INMOS 84a]:
Occam programming manual,
Prentice-Hall International Series in Computer Science, 1984.
- [Kahn 74]: G. Kahn,
The semantics of a simple language for parallel programming,
Proceedings of the IFIP Congress, pp. 471-475, August 1974.
- [Kennaway 82]: J. R. Kennaway, M. R. Sleep,
Director strings as combinators,
Computing Studies Sector, University of East Anglia, Norwich NR4 7TU, 1982.
- [Kennaway 85]: J. R. Kennaway, M. R. Sleep,
Counting director strings,
[Aspenäs 85].
- [Liskov 81]: Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss,
J. Craig Schaffert, Robert Scheifler, Alan Snyder,
CLU reference manual,
Springer-Verlag, 1981.
- [MacQueen 84]: D. B. MacQueen, Ravi Sethi,
An ideal model for recursive polymorphic types,
[POPL 84].
- [Milner 78]: R. Milner,
A theory of type polymorphism in programming,
JCSS, 17(3), pp. 348-375.
- [Milner 80]: R. Milner,
A calculus of communicating systems,
Springer-Verlag, 1980.

- [Motorola 82]:
MC68000 16-bit microprocessor user's manual,
Prentice-Hall, 1982.
- [Mycroft 81]: A. Mycroft,
*Abstract interpretation and optimising transformations
for applicative programs*,
University of Edinburgh Dept. of Computer Science, 1981.
- [Noshita 84]: Kohei Noshita,
Translation of Turner combinators in $O(n \log n)$ space,
Department of Computer Science,
Denkitusin University, Chofu, Tokyo 182, 1984.
- [Noshita 84a]: Kohei Noshita, Teruo Hikita,
The BC-chain method for representing combinators in linear space,
RIMS Symposium on Software Science and Engineering,
Kyoto, September 1984.
- [Park 82]: D. Park,
The "fairness" problem in nondeterministic computing networks,
Proceedings of the Fourth Advanced Course on Theoretical Computer Science,
Mathematisch Centrum, Amsterdam, The Netherlands, 1982.
- [Peyton Jones 82]: Simon L. Peyton Jones,
The B epidemic and its cure,
technical report, University College London, 1981.
- [Peyton Jones 84]: Simon L. Peyton Jones,
Directions in functional programming research,
pp. 220-249 of [Duce 84].
- [Peyton Jones 85]: Simon L. Peyton Jones,
GRIP—a parallel graph reduction engine,
[Aspenäs 85].
- [POPL 76]:
Third Annual ACM Symposium on Principles of Programming Languages,
Atlanta, Georgia, 19-21 January, 1976.
- [POPL 78]:
Fifth Annual ACM Symposium on Principles of Programming Languages,
Tucson, Arizona, 23-25 January, 1978.
- [POPL 84]:
Eleventh Annual ACM Symposium on Principles of Programming Languages,
Salt Lake City, Utah, 15-18 January, 1984.

- [POPL 85]:
Twelfth Annual ACM Symposium on Principles of Programming Languages,
New Orleans, Louisiana, 14-16 January 1985.
- [Redelmeier 84]: D. Hugh Redelmeier,
Towards Practical Functional Programming,
Computer Systems Research Institute, University of Toronto, May 1984.
- [Richards 81]: Hamilton Richards Jr.,
Management of graph storage in a cellular S-K machine,
Burroughs Corporation Austin Research Center, 1981.
- [Richards 85]: Hamilton Richards Jr.,
An overview of the Burroughs NORMA,
[Aspenäs 85].
- [Schorr 67]: H. Schorr and W. Waite,
*An efficient machine-independent procedure for garbage collection in
various list structures*,
Comm ACM 10, 8, pp. 501-506, August 1967.
- [Stoy 77]: Joseph E. Stoy,
*Denotational semantics:
the Scott-Strachey approach to programming language theory*,
MIT Press, 1977.
- [Stoye 83]: W. R. Stoye,
The SKIM microprogrammer's guide,
Cambridge University Computer Laboratory Technical Report #40, 1983.
- [Stoye 84]: W. R. Stoye, T. J. W. Clarke and A. C. Norman,
Some practical methods for rapid combinator reduction,
pp. 159-166 of [ACM 84].
- [Stoye 84a]: W. R. Stoye,
A new scheme for writing functional operating systems,
Cambridge University Computer Laboratory Technical Report #56, 1984.
- [Turner 76]: D. A. Turner,
SASL language manual,
University of St. Andrews Computer Science Department, 1976.
- [Turner 79]: D. A. Turner,
A new implementation technique for applicative languages,
Software Practice and Experience, Volume 9, pp. 31-49, 1979.

- [Turner 82]: D. A. Turner,
Recursion equations as a programming language,
pp. 1-28 of [Darlington 82].
- [Turner 85]: D. A. Turner,
Miranda: a non-strict functional language with polymorphic types,
submitted to the ACM Conference on
Functional Programming Languages and Computer Architecture,
Nancy, 16-19 September 1985.
- [UCL 83]:
Proceedings of the Declarative Programming Workshop,
University College London, 11-13 April 1983.
- [Wadler 84]: P. Wadler,
*Listlessness is better than laziness:
lazy evaluation and garbage collection at compile time*,
pp. 45-52 of [ACM 84].
- [Wadsworth 71]: C. P. Wadsworth,
Semantics and pragmatics of the lambda calculus,
D. Phil. Thesis, University of Oxford, 1971.
- [Wray 83]: S. C. Wray, private communication.
- [Wray 85]: S. C. Wray,
A new strictness detection algorithm,
[Aspenäs 85].