Technical Report

UCAM-CL-TR-92
ISSN 1476-2986

Number 92

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Implementation and programming techniques for functional languages

Stuart Charles Wray

June 1986

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 1986 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Christ's College.

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

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

In this thesis I describe a new method of strictness analysis for lazily evaluated functional languages, and a method of code generation making use of the information provided by this analysis. I also describe techniques for practical programming in lazily evaluated functional languages, based on my experience of writing substantial functional programs.

My new strictness analyser is both faster and more powerful than that of Mycroft. It can be used on any program expressed as super-combinator definitions and it uses the additional classifications *absent* and *dangerous* as well as strict and lazy. This analyser assumes that functional arguments to higher order functions are completely lazy.

I describe an extension of my analyser which discovers more strictness in the presence of higher order functions, and I compare this with higher order analysers based on Mycroft's work. I also describe an extension of my analyser to lazy pairs and discuss strictness analysers for lazy lists.

Strictness analysis brings useful performance improvements for programs running on conventional machines. I have implemented my analyser in a compiler for Ponder, a lazily evaluated functional language with polymorphic typing. Results are given, including the surprising result that higher order strictness analysis is no better than first order strictness analysis for speeding up real programs on conventional machines.

I have written substantial programs in Ponder, and I describe in some detail the largest of these, which is about 2500 lines long. This program is an interactive spreadsheet using a mouse and bit-mapped display. I discuss programming techniques and practical problems facing functional languages with illustrative examples from programs I have written.

Acknowledgements

I am grateful to my supervisor, Mike Gordon, for his guidance and encouragement when I needed it, and for leaving me to my own devices when I had the bit between my teeth.

I would like to thank John Hughes, Alan Mycroft and Simon Peyton Jones for helpful discussions about strictness analysis, and Henk Barendregt for showing an interest in this work when it was rather half baked.

I would also like to thank to Mike Gordon, Inder Dhingra, Will Stoye, Jon Fairbairn, Sarah Woodall and Thomas Clarke for reading drafts of this thesis.

The SERC and Christ's College provided money, for which I am grateful.

The Ponder 'back end' was written in collaboration with Jon Fairbairn, except for the strictness analyser and code generator described in this thesis which are my own work.

Except where otherwise stated in the text, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree, diploma or any other qualification at any other university. No part of this dissertation has already been or is being currently submitted for any such degree, diploma or any other qualification.

Contents

Chapter 1

Background

1.1 Early Computers

The first computers were thought of as fast calculators, suitable for making mathematical tables and performing the tedious computations of numerical analysis. ENIAC, the first program controlled electronic calculator, was built during the second world war to produce firing tables for field guns, for which there was a desperate need. Although pathetically slow by modern standards it was a thousand-fold improvement on the mechanical desk calculators then in use. Calculations that had previously taken three days could now be performed in 20 seconds.

It was not clear that such blinding electronic speed would be useful outside a few specialist areas — after all, there was only so much calculation that needed doing. Problems had always been considered intractable if they could not be solved by hand, because there had been no alternative. The pioneers of computing had to emphasise that the invention of the electronic computer had changed the economics of problem solving. Problems that were too long winded for solution by hand might now yield to brute-force computer solutions.

While ENIAC was still being built, the designers of another machine, EDVAC, conceived the idea of a 'stored program electronic computer'. Such a machine could be re-programmed by changing the contents of its control store, rather than by the time consuming chore of re-wiring a plug-board. ENIAC was modified to have a read-only control store, and BINAC and EDSAC were the first computers to run with a writable control store, which could be modified by a program [Metropolis & Worlton 80]. Now not only numbers, but programs themselves could be manipulated automatically.

It had long been realised that calculating machines could manipulate symbols as well as numbers [Lovelace 43], and what are programs if not strings of symbols?

Machines now had the capability to convert an abstract description of an algorithm into a program which their hardware could execute directly. In this way they could appear to execute the abstract description.

The first programming languages were just a formalisation of the instructions one might give to a somewhat stupid human calculator. As such they were only a small part of the problem solving process, which in numerical analysis concentrated mainly on transforming the problem into a tractable form. Nevertheless there was considerable interest in how programs should be written and what the best notation was.

At this time many novel programming languages were invented, some having a two-dimensional layout that even now would be regarded as avant-garde [Knuth 77]. They were often not meant to be implemented on the computers of the day, but to be used as mental aids to programmers, who would produce their code by hand after writing out an algorithm in a high level language. It should be noted that such languages, to achieve their aims, had to mirror the architecture of the machines for which the code would be written, otherwise it would have been more difficult to write the code from the high-level program than it would without it.

In 1950 Alick E. Glennie wrote the first real compiler, for a language called AUTOCODE. It was implemented and used; it took algebraic statements and translated them into machine code. It was not a success. As Glennie related in the 1960s:

> The climate of thought was not right. Machines were too slow and too small. It was a programmer's delight to squeeze problems into the smallest space. [Knuth 77]

Knuth commented that

> An improvement in the coding process was not regarded then as a breakthrough of any importance, since coding was often the simplest part of a programmer's task. When one had to wrestle with problems of numerical analysis, scaling and two-level storage, meanwhile adapting one's program to the machine's current state of malfunction, coding itself was quite insignificant. [Knuth 77]

Of particular interest is an early (perhaps the earliest) declarative language, ADES II, implemented by E. K. Blum in the mid fifties.

> The ADES approach to automatic programming is believed to be entirely new. Mathematically, it has its foundations in the bedrock of the theory of recursive functions. [Blum 56]

Blum goes on to say why he considers this approach to be worthwhile:

> The b-equations constitute the main part of an ADES formulation. They define the quantities to be computed, and closely resemble the conventional equations which a mathematician writes to describe a computational problem. But, in automatic digital computation, mathematical formulas are not sufficient. One must also specify the order in which quantities are to be computed, that is a flow chart must be drawn up. This is frequently more difficult than writing the formulas. In ADES, a large part of the flow chart is determined implicitly in a natural way by the very structure of the b-equations. [Blum 56]

These comments are very perceptive, considering that the work on ADES was done thirty years ago. Unfortunately, it was before its time. ADES was implemented on an IBM 650, which only had a small amount of storage. Consequently the compiler had to be fed through the computer on punched cards once for each equation. ADES is rather baroque in appearance, especially when encoded in the limited alphabet available, and rather feeble in its capabilities. It did not have general recursion as used in Algol and LISP. A fear of recursive definition seems to pervade all early languages.

By the late sixties hardware had improved, but the techniques for writing programs had not changed very much compared with the changes in the computers themselves. Programming methods had at all times been barely adequate for the newest hardware, and in 1968 at the first "Software Engineering" conference [Naur & Randell 68] the truth was finally admitted: the methods used to program the high-speed calculators of the early fifties simply were not adequate for the more advanced computers of the late sixties. The term "software crisis" was coined to describe this state of affairs, and people became willing to seek solutions to their newly perceived problem. Out of this has come the idea that the fundamental requirement of programs is that they should work. It does not matter how fast or how pretty they are — if they do not do what they are supposed to they are not worth having.

The structured programming approach [Dijkstra 68] made it more likely that programs would work, because they are constrained to have a simple and regular

structure. The intuitive justification behind functional languages is that they are so simple and so regular that programs written in them are even more likely to work.

Theoretical work had shown that it was not necessary for a programming language to have assignment, but it was not clear that it would be possible to write programs easily in such a language. By the mid seventies, spurred on by the software crisis, work on functional languages had shown that at least for small examples it *was* possible to program effectively in a language without assignment [Burge 75]. Even Backus, the high priest of FORTRAN, came down on the side of functional programming with his language FP [Backus 78].

The notion of lazy evaluation began to catch on about this time as well [Vuillemin 74, Friedman & Wise 76, Henderson & Morris 76], largely on the grounds that it provided an implementation of recursion that was mathematically elegant. The practical techniques for implementing laziness depended on there being no side effects, so this drove together the two concepts of lazy evaluation and functional languages, since each was impractical without the other.

Functional languages are based directly on the lambda calculus and can easily be transformed into lambda expressions. Most functional languages are written in a "sugared" version of lambda notation, designed to be easier to read and perhaps containing information about the data types being used. Some functional languages are written as "recursion equations", but since these are equivalent to lambda calculus definitions I will only consider functional languages based more closely on the lambda calculus.

1.2 The Lambda Calculus

The lambda calculus was invented in the 1930s as an aid to clearer thinking about mathematical functions. One view of functions is that they can be thought of as tables, so that when given an argument the corresponding result can be found in the table. This idea is all very well, but it does not describe how in practice to find the result for a given function applied to a given argument—the tables needed do not actually exist, after all. The lambda calculus is a notation for writing functions as "recipes" or procedures to be followed, so that for any arguments the answer can be found by following simple rules.

As an example, define F by

$$F = \lambda a.\, \lambda b.\, a$$

F is a function that takes two arguments, and returns the first of them as its result. A name after a lambda (λ) is a *bound variable*, which will be systematically replaced by the corresponding argument when the function is applied. Suppose we apply F to two arguments:

$$F\ 2\ 3$$

The argument 2 is associated with the bound variable a, and 3 is associated with b. Wherever a occurs in the body of F it is replaced by 2, and b is replaced by 3. The result of F 2 3 is thus obtained by replacing the body of F (the single bound variable a) by 2. The answer is 2.

In more complex cases it would be possible to confuse one variable with another when doing substitutions. The rules for such cases are explained below, after the formal syntax of lambda expressions has been described.

The notation for writing a function F applied to an argument A is '$F\ A$' rather than '$F\ (A)$' as in most programming languages. Brackets may be used to group expressions together and there is an implicit bracketing to the left. Thus '$F\ A\ B$' is equivalent to '$((F\ A)\ B)$', meaning "apply F to A and then apply the resulting function to B". It is quite different from '$F\ (A\ B)$', which means "F applied to the result of applying A to B".

F as it is written here is a *curried* function, that is to say it takes one argument and returns another function. This second function does the rest of the work when it is presented with further arguments. For instance if F had only been applied to one argument, the '2' in the above case, the result would have been the function $\lambda b.\ 2$. This is a function that takes a single argument, throws it away and returns '2'. Any function of many arguments has a curried form which takes one argument and returns a function to do the rest of the work, so only being able to write functions of one argument is no disadvantage. Currying is named after Haskel B. Curry who discovered this property (it was discovered earlier and independently by Frege and Schönfinkel [Rosser 82], but their names are harder to pronounce).

To define the lambda calculus more formally it is first necessary to specify the syntax of permissible lambda expressions:

$$expression \;=\; \begin{cases} variable & \text{A variable} \\ expression\ expression & \text{An application} \\ \lambda variable.\ expression & \text{An abstraction} \end{cases}$$

A *variable* in this grammar is chosen from an infinite number of distinct names. The grammar defines the abstract syntax of lambda expressions. To disambiguate expressions like $e_1\ e_2\ e_3$ that could be parsed in two ways, the concrete syntax

allows brackets to be used. In the absence of brackets this expression would by default be parsed as $(e_1\ e_2)\ e_3$.

If a function will be referred to repeatedly it is customary to name it using the notation

$$F \ = \ expression$$

It is important to note that this naming mechanism does not allow recursive definition. The value of F cannot be referred to directly from within *expression*. The meaning of

$$F \ = \ expression_1$$
$$expression_2$$

is

$$(\lambda F.\ expression_2)\ expression_1$$

In other words, wherever F occurs in $expression_2$ it is replaced by $expression_1$, but since F is not bound in $expression_1$ it cannot refer to itself. For recursive definitions a fixed point function, conventionally Y, is used. This will be described later.

There are three rules that define all the ways of interpreting lambda expressions. These are called α-, β- and η-conversion. Before describing these, we must define precisely what the terms "bound variable" and "free variable" mean, and how substitution for a variable is performed. The letters x, y and z will be used to stand for distinct variables, and the letters X, Y, Z and M will be used to stand for whole expressions (each of which could consist of just single variables).

Free variables

1. A variable
   (a) x is free only in x, and not in any other variable
2. An application
   (a) x is free in $X\ Y$ if it is free in X or Y or both
3. An abstraction
   (a) x is not free in $\lambda x.\ X$
   (b) x is free in $\lambda y.X$ if it is free in X, where y is distinct from x

Bound variables

1. A variable
   (a) x is not bound in any expression consisting of a single variable
2. An application
   (a) x is bound in $X\ Y$ if it is bound in X or Y or both
3. An abstraction
   (a) x is bound in $\lambda x.\ X$
   (b) x is bound in $\lambda y.X$ if it is bound in X, where y is distinct from x

Substitution

The notation $X[M/x]$ is used to mean "the expression X with the expression M substituted for x wherever it occurs in X". The result of this substitution is defined as follows:

1. X is a variable
   (a) If X is x the result is M
   (b) If X is not x the result is X
2. X is an application, $Y\ Z$
   (a) The result is $(Y[M/x])\ (Z[M/x])$
3. X is an abstraction
   (a) If X is $\lambda x.\ Y$ the result is $\lambda x.\ Y$
   (b) If X is $\lambda y.\ Y$, where y is distinct from x, then
      i. If x is not free in Y the result is $\lambda y.\ Y$
      ii. If y is not free in M the result is $\lambda y.\ Y[M/x]$
      iii. Otherwise the result is $\lambda z.\ (Y[z/y])[M/x]$, where z is chosen to be distinct from any of the variables free in M or Y

The rules for variables and applications are fairly straightforward. The rules for abstractions are complicated by the possibility of name clashes, where a variable which was free in M could become bound in $X[M/x]$ if the substitution was done carelessly. For instance consider

$$(\lambda y.\ x)[(y\ 3)/x]$$

One might initially think this should be

$$(\lambda y.\ (y\ 3))$$

but now the y of $(y\ 3)$ is bound and it was previously free. Following through the rules above, we see that the variables must first be renamed:

$$\begin{array}{l}\lambda z.\ (x[z/y])[(y\ 3)/x] \\ \lambda z.\ x[(y\ 3)/x] \\ \lambda z.\ (y\ 3)\end{array}$$

In this expression y is still free, which is correct.

We are now in a position to define the three conversion rules previously mentioned. The notation $A \overset{\varphi}{\longleftrightarrow} B$ means "A can be replaced by B or B by A using the φ-conversion rule".

α-conversion

If y is not free in X, $\lambda x.\ X \overset{\alpha}{\longleftrightarrow} \lambda y.\ X[y/x]$

β-conversion

$(\lambda x.\ M)\ N \overset{\beta}{\longleftrightarrow} M[N/x]$

η-conversion

If y is not free in X, $\lambda y.\ (X\ y) \overset{\eta}{\longleftrightarrow} X$

These rules are used to evaluate expressions and they may be applied to any subexpression of an expression. Expressions of the form $(\lambda x.\ M)\ N$ are known as *β-redexes* and expressions of the form $\lambda y.\ (X\ y)$, where y is not free in X, are known as *η-redexes*. An expression containing no redexes is said to be in *normal form*. When the above rules are used in their left to right forms to eliminate redexes they are called *reductions* since they usually make expressions smaller. The notation $A \overset{\varphi}{\longrightarrow} B$ means "A can be reduced to B using the φ-conversion rule

in its left to right form". For instance the example given earlier of $F\ 2\ 3$ would be reduced by the following steps:

$$\begin{aligned} F\ 2\ 3 &\equiv (\lambda a.\ (\lambda b.\ a))\ 2\ 3 \\ &\xrightarrow{\beta} (\lambda b.\ 2)\ 3 \\ &\xrightarrow{\beta} 2 \end{aligned}$$

In addition to α-, β- and η-reduction, other reductions called δ-reductions are sometimes introduced. For instance most implementations of functional languages use integers provided by the hardware to do arithmetic, and these built-in functions can be "retro-fitted" into the lambda calculus by introducing them as δ-reductions.

Although the rules for reducing lambda expressions have been given, I have not described what to do if more than one reduction is possible in an expression. Which one should be done first? It is not immediately obvious that this is a problem, but consider the following definitions:

$$\begin{aligned} F &= \lambda a.\ \lambda b.\ a \\ X &= \lambda x.\ x\ x \end{aligned}$$

What is the value of $F\ 3\ (X\ X)$? First consider $(X\ X)$:

$$\begin{aligned} (X\ X) &\equiv ((\lambda x.\ x\ x)\ X) \\ &\xrightarrow{\beta} (X\ X) \end{aligned}$$

Since $(X\ X)$ reduces to $(X\ X)$, a reduction order that does this first will never produce an answer: it will continue trying to reduce $(X\ X)$ forever. Fortunately there is a reduction order which is guaranteed to terminate if any order would terminate for a particular expression, and that is *normal order reduction*. In this scheme the function at the head of the expression (in this case F) is always reduced first. Thus

$$\begin{aligned} F\ 3\ (X\ X) &\equiv (\lambda a.\ \lambda b.\ a)\ 3\ (X\ X) \\ &\xrightarrow{\beta} (\lambda b.\ 3)\ (X\ X) \\ &\xrightarrow{\beta} 3 \end{aligned}$$

Another reduction order, *applicative order*, reduces the argument to a function first. In this example it would thus try to reduce $(X\ X)$ first, and never get an answer. Applicative order reduction corresponds to *eager evaluation* or *call-by-value*, used in most conventional programming languages to pass arguments to functions. In call-by-value arguments are always evaluated before a function is invoked. Normal order reduction corresponds to *lazy evaluation* or *call-by-need*, used in lazily evaluated functional languages. In these methods of argument passing arguments are evaluated when required, after a function has been invoked.

An argument will not be evaluated at all if it is not required for the function to return a result.

There is often confusion about the terms 'normal order' and 'lazy'. Lazily evaluated functional languages are implementations of normal order reducers. 'Lazy evaluation' refers to a particular implementation technique, 'normal order' to the principle behind it. Call-by-name also implements normal order reduction, the difference being that call-by-name may evaluate an argument every time it is used in a function, but lazy evaluation will evaluate an argument at most once. If an argument is used more than once, a lazy evaluator supplies the value it calculated earlier, thus potentially saving itself a lot of work.

Different reduction schemes can yield an answer or fail to terminate when given the same expression to reduce. If an expression is reduced in two different ways, and they both terminate, will the answers always be the same? The Church-Rosser Theorem says that they may not be exactly the same, but they will be equivalent. More precisely:

The Church-Rosser Theorem

> Define $A \overset{*}{\longleftrightarrow} B$ to mean A can be converted to B by some number of α-, β- and η-conversions, and $A \overset{*}{\longrightarrow} B$ to mean A reduces to B by some number of α-, β- and η-reductions.
>
> If $X \overset{*}{\longleftrightarrow} Y$
>
> then there exists an expression Z
> such that $X \overset{*}{\longrightarrow} Z$
> and $Y \overset{*}{\longrightarrow} Z$

Using the Church-Rosser theorem we can prove that no expression can be reduced to two distinct normal forms. The proof is as follows:

Suppose that the original expression could be reduced to two distinct normal forms. The first normal form could be converted back to the original expression and hence to the second. But by the Church-Rosser theorem there would be a further expression to which they were both reducible. This is a contradiction, since normal forms have no redexes, so they cannot be reduced at all. Thus no two orders of reduction can give different normal forms.

At first it might seem that the lambda calculus is not very expressive since it is so simple. This is far from the case: it is even possible to represent the natural numbers in the lambda calculus. The following definitions give one of the many ways of doing this.

$$\begin{aligned}
\overline{0} &= \lambda x.\, \lambda y.\, y \\
\overline{1} &= \lambda x.\, \lambda y.\, x\ y \\
\overline{2} &= \lambda x.\, \lambda y.\, x\ (x\ y) \\
&\cdots \\
\overline{n} &= \lambda x.\, \lambda y.\, \overbrace{x\ (x\ \ldots\ (x}^{n}\ y)\ \ldots)
\end{aligned}$$

In other words, $\overline{n}$ applies its first argument n times to its second. Truth values can be represented in similar fashion:

$$\begin{aligned}
\mathit{True} &= \lambda x.\, \lambda y.\, x \\
\mathit{False} &= \lambda x.\, \lambda y.\, y
\end{aligned}$$

So if we define

$$\mathit{IsZero} = \lambda p.\, p\ (\lambda x.\, \mathit{False})\ (\mathit{True})$$

then

$$\begin{aligned}
\mathit{IsZero}\ \overline{n} &\xrightarrow{*} \mathit{True} \quad \text{If } \overline{n} \text{ is } \overline{0} \\
&\xrightarrow{*} \mathit{False} \quad \text{Otherwise}
\end{aligned}$$

Although the lambda calculus has no conditional construct built in this is not a problem, because it can easily be defined.

$$\mathit{If} = \lambda a.\lambda b.\lambda c.\, a\ b\ c$$

So that

$$\mathit{If}\ \mathit{True}\ x\ y \xrightarrow{*} x$$

and

$$\mathit{If}\ \mathit{False}\ x\ y \xrightarrow{*} y$$

I mentioned earlier that a fixed point function is necessary for recursive definitions, since the name of a function cannot be referred to directly from within its own body. The standard fixed point function is Y which may be defined by:

$$Y = \lambda f.\, (\lambda x.\, f\ x\ x)\ (\lambda x.\, f\ x\ x)$$

So that

$$Y\ f \xrightarrow{*} f\ (Y\ f) \xrightarrow{*} f\ (f\ (Y\ f)) \xrightarrow{*} \cdots$$

Recursive functions may now be written as

$$F = Y\ (\lambda f.\, \mathit{expression})$$

Where *expression* is the 'real' body of F, and within this body the variable f may be used to refer recursively to the function.

The discovery of the power of the lambda calculus led to Church's Thesis, a surprising hypothesis which cannot be proven because it is couched in informal terms. Church's Thesis states: "Effectively calculable functions from positive integers to positive integers are just those definable in the lambda calculus." The intuitive notion of "effectively calculable" is taken to mean those functions that you or I or some mechanism could in principle calculate. If "effectively calculable" is taken to mean "What a Turing machine can calculate" then Church's Thesis is a correct theorem — a Turing machine can calculate exactly those functions expressible in the lambda calculus, no more, no less. This is particularly interesting since digital computers can be regarded as finite approximations to universal Turing machines. It is the same with all other models of computation: general recursive functions, register machines and so on. They have exactly the same power as the lambda calculus. No one now questions that Church's Thesis is correct, even though it cannot be proved.

1.3 Functional Languages

Pure LISP [McCarthy 60] is often regarded as a functional language, but there are substantial differences between it and a language based directly on the lambda calculus. Although LISP has an equivalent of β-conversion, it uses a simpler substitution rule and its functions are not curried. Some parts of LISP look superficially similar to lambda expressions and in many cases behave like lambda expressions, but in the case of higher order functions (functions which take functions as arguments) the behaviour can be totally different.

The reason for this is that while the inventors of LISP knew of the lambda calculus they treated it as just a notational hint [Park 85]. It was not for some time that they realised that the LABEL construct of LISP (which allowed recursion) was unnecessary if the lambda calculus reduction rules were implemented properly. By then it was too late, as the excitement of having such a powerful language as LISP, even with its flaws, carried it forward. People wanted to use LISP because it was so expressive, and were unconcerned with what they saw as theoretical niceties.

However, a language using normal order reduction would have been impractical in the early sixties for reasons of efficiency. Even now the problems of implementing normal order reduction efficiently are far from solved and in the early sixties efficiency was a major concern for compiled languages. The idea of *pure* func-

tional languages was considered ridiculous. Functional languages were assumed to be embeded within an imperative language [Landin 66], perhaps because without normal order reduction they are rather feeble and really do need imperative support. For instance Naur commented:

> I can write every program you can conceive of in the world in one statement. The statement reads as follows:
>
> $$Output \;=\; Program\ (Input)$$
>
> This covers every program in the world provided you have been sure to have established the rules for all programs conceivable. Now normally this is a very inconvenient way, a very uneconomical way, and it would be terrible to use our machines in this way. [Raphael 66]

This is exactly what programs written in a pure functional language look like, and exactly the way they work. Sometimes it can be inconvenient, but it is not as inefficient as Naur feared. Lazy evaluation can be implemented with reasonable efficiency for a pure functional language, but for an imperative language the problems of side effects make lazy evaluation hopelessly inefficient.

The early implementations of functional languages either reduced lambda expressions directly, or translated them into a program for an SECD machine (a very simple kind of abstract machine [Landin 64]). Both of these approaches were rather slow as implementations of normal order reduction because they had to do a lot of copying.

Turner produced an implementation that was superior to these reducers by translating lambda expressions into expressions containing only *combinators* and then interpreting these combinatory expressions [Turner 79]. Combinators are functions with no free variables. Any lambda expression can be converted into a corresponding combinatory expression which has no abstractions and no bound variables and only uses a small number of different combinators. Turner used S and K, originally proposed by Schönfinkel, with a few others chosen to make the combinatory expressions more compact. S and K can be written in the lambda calculus as

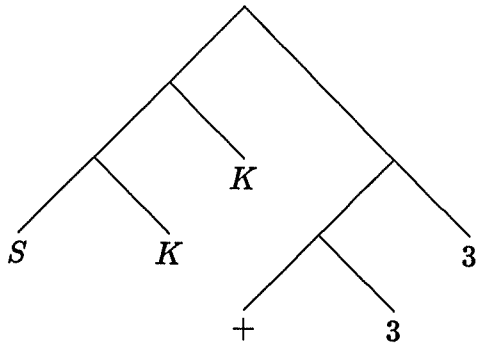
$$\begin{aligned} S &= \lambda a.\ \lambda b.\ \lambda c.\ a\ c\ (b\ c) \\ K &= \lambda a.\ \lambda b.\ a \end{aligned}$$

S and K are sufficient on their own to express any function that can be written in the lambda calculus, but the combinatory expressions would then be rather large.

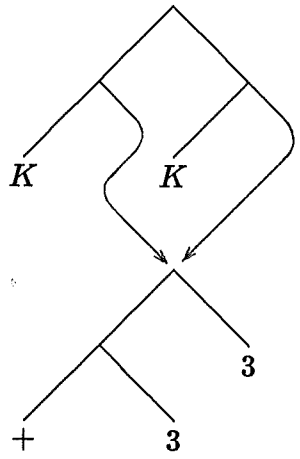
Turner's interpreter used the idea of *graph reduction* [Wadsworth 71] which has proved extremely popular with other implementors of lazy functional languages, as it presents a simple way to perform lazy evaluation. The basic idea is that an expression is stored as an *application tree*, so that the expression

$$S\ K\ K\ (+\ 3\ 3)$$

would be represented by the tree



S has three arguments, so it can be reduced, since $S = \lambda a.\ \lambda b.\ \lambda c.\ a\ c\ (b\ c)$. There would be no point in reducing S if it had been given fewer than three arguments, since it could have done no useful work. Note that the third argument is used in two places. Because of the graphical representation this argument need not be duplicated. Instead a pointer to the expression (+ 3 3) is used in both places. Now if the expression is reduced in one place, all other occurrences of it will have been reduced too and at no extra cost. The previous graph is rewritten as



An algorithm for conversion from lambda terms to combinators similar to Turner's is described in [Abdali 76]. This would produce more compact code

which would run faster than Turner's implementation, but it appears never to have been implemented, probably because it has been superseded by the newer implementation method using "super-combinators".

This more recent technique generates combinatory code for a program using 'tailor made' combinators, designed to be the best possible for that particular program. Using a small set of combinators as in Turner's system means that a program will be executed in very small steps, since each combinator does only a very small amount of work, and 'book-keeping' overheads will predominate. The "super-combinators" of Hughes [Hughes 82] are tailor made so as to be as large as possible without losing *full laziness*. An implementation is said to be fully lazy if no expression is evaluated more than once, regardless of whether it forms the whole of an argument to a function. Similar work has been done by Johnsson [Johnsson 83] — his large combinators do not preserve full laziness, but are otherwise similar to those of Hughes. Super-combinators have formed the basis for high performance implementations of functional languages, for instance Ponder [Tillotson 85] and LML [Augustsson 84].

There are now many functional languages — SASL, Hope, KRC, Miranda, Ponder, LISPKIT, LML and so on. The implementations of these languages are efficient enough for people to attempt to write real programs, and to compare them with conventional languages. Functional languages have had a long and faltering history. Since before the invention of LISP, people have tried to make a mathematically respectable programming language, but not quite got it right. With the invention of lazily evaluated functional languages a decade ago it seemed that this had at last been done. However, progress in demonstrating that functional languages are convenient to use has been painfully slow since then. The largest purely functional programs are still rather small by conventional standards. It remains to be seen whether, for all their theoretical purity, functional languages are really any better than imperative languages for practical programming.

1.4 Appraisal

Programmers brought up on conventional languages usually find functional languages confusing when they first meet them. How, they ask, is it possible to write programs without assignment? How are input and output to be dealt with? It is very difficult to believe that any but the most trivial programs can be written. The analogy is sometimes drawn between structured programming, in which GOTOs are

banned, and functional programming, in which assignments are banned. However, for the programmer who is used to both GOTOs and assignment, it is a relatively small mental step to structured programming, since structured programs can easily be translated back into programs using GOTOs; the two are equivalent [Dijkstra 68].

Structured programming is merely a stylistic device that forces a programmer to impose *some* structure on his thoughts, where GOTOs impose no structure at all. The situation with functional programming is rather different, since most conventional languages would be useless if assignment were forbidden. Herein lies the novice's problem. He imagines that functional languages are merely conventional languages without assignment, whereas the crucial difference is really that functions are first class objects in functional languages, treated no differently from other values. It can be extremely difficult to translate a program from a language with higher order functions but no assignment into a language with assignment but no higher order functions.

Once a programmer has recovered from his initial mental paralysis there is a tendency for him to go overboard in his enthusiasm for functional programming. It is easy to imagine that functional languages are the panacea for all programming problems. This judgement is usually made on the basis of writing 100 line programs that use "neat coding tricks". Rather than writing larger programs it is very tempting to 'improve' a functional language so as to express these toy programs even more succinctly.

After a while the realisation dawns that the whole thing is much more complex and fraught with difficulties than it first appears. The question which must be answered is "Why use functional languages at all?" If there is no justification for them, no benefit to be gained from them, then we may as well give up now. Functional languages are usually presented as the solution to some problem, but it is rarely stated precisely what the problem is, and why less drastic remedies would not work just as well. The fact that it is possible to do in a functional language what can be done in an imperative language is really no incentive to change to functional languages. So what *is* the problem? One problem is that of increasing complexity.

In all branches of computer science there is the problem of how to control complexity. For example in VLSI design the analogy has been drawn between the complexity of a road layout in a town and the complexity of a VLSI layout. In the mid sixties the complexity of a typical chip was about the same as that of a small

town. A designer could find his way around the whole thing fairly easily. A late seventies microprocessor was of the same complexity as a large city. By the 1990s we can expect chips with a complexity greater than that of all the road layouts in the whole world — if they can be designed at all, and made to work.

In software the outlook is the same. Although software faced its "crisis" twenty years ago, the remedies suggested then solved the problems of the sixties, not the problems of the nineties. Large application programs such as telephone exchanges and flight control systems are millions of lines long. It has been estimated that the software for the United States' 'Strategic Defence Initiative', if it is ever written, will be between 10 and 100 million lines long.

Functional languages have the potential to control this complexity because their fundamental principle is abstraction — the hiding of detail. Part of a functional program can be examined and understood in isolation from the other parts, then packaged up for use by others who need not know its internal workings at all. The effects of changing part of a program are easier to contain, since any information used by a function must be explicitly passed to it. Without side effects all dependencies are much more obvious.

Another problem that functional languages might solve is that of programming parallel computers. In conventional languages there are so many implicit dependencies between different parts of a program that it is difficult to generate efficient code for a parallel machine. Conventional languages are designed to run on conventional machines, where such dependencies do not matter. Since expressions in functional languages have no side effects they can be evaluated in any convenient order — even in parallel. However, in languages with normal order reduction there will be few arguments that need reduction simultaneously. Only the arguments to built-in functions like addition, which must definitely evaluate both arguments, could be evaluated in parallel. Fortunately there is a technique known as *strictness analysis* which can be used to find out if the arguments to user defined functions can be evaluated eagerly without violating the normal order semantics. If they can, then they can be evaluated in parallel too.

The rest of this thesis is set out as follows:

- Chapter two is concerned with strictness analysis. It describes an entirely new method of strictness analysis and compares it with other work.
- Chapter three describes the extension of this strictness analyser to higher order functions. I also show how it can be extended to lazy lists and indicate some fundamental limitations of strictness analysis.

- In Chapter four I present the results of implementing this strictness analyser in the Ponder system for a conventional machine. I also discuss the techniques necessary to implement functional languages successfully on parallel machines.

- Chapter five is concerned with practical programming in functional languages. Among other things, I describe some programming techniques that can only be used in lazy functional languages.

Chapter 2

Strictness Analysis

2.1 Introduction

In lazy functional languages it is often the case that an argument to a function will necessarily be evaluated by that function. It could thus have been evaluated before being passed to the function (*eagerly*) rather than after (*lazily*). The aim of strictness analysis is to find out where eager evaluation would have the same effect as lazy evaluation, and so allow a compiler to produce more efficient code. There is generally a greater overhead associated with passing arguments to functions lazily than there is with passing them eagerly. The reason for this is that when an expression is passed eagerly, all that has to be done is to calculate its value and pass that value to the function. When an expression is passed lazily, a 'prescription' describing how to calculate its value must be passed to the function. If its value is subsequently required, it must be created by interpreting the prescription. Usually this interpretation mechanism is slower than evaluating the original expression eagerly, but in any case there is the overhead of creating and disposing of the prescription.

Strictness analysis could also allow lazy functional languages to make effective use of parallel evaluation. On a parallel machine all the strict arguments to a function can be evaluated at the same time, since they are certain to be required. Even if passing arguments lazily were as cheap as passing them eagerly, strictness analysis would be very useful for starting parallel reductions. In practice there are snags with this idea, not the least of which is that no one knows what kind of parallel machine would be suitable. I discuss this further in chapter four.

The strictness analysis method presented in this chapter is an improvement on that of Mycroft [Mycroft 81] in several ways:

- Mycroft's analyser only works on first order functions, so functions that

take functions as arguments or return functions as results (that is curried functions) are forbidden. My new analyser allows both higher order functions and curried functions. However, some strictness information is discarded when analysing higher order functions. A functional argument is assumed to treat *its* arguments lazily. No strictness information is lost when returning a function as the result of a function. In the next chapter I describe an extension which loses less strictness information about functional arguments, provided that these arguments are not themselves higher order functions. I call the first method, described in this chapter "first order" and the extended version "second order".

- Strictness analysers based on Mycroft's work classify arguments as being either strict or lazy. My new analyser uses the classifications *absent* (originally suggested in [Hughes 85a]) and *dangerous* as well as strict and lazy. If my analyser finds that a function will definitely evaluate an argument, that argument is said to be strict. If the analyser finds that a function will definitely *not* evaluate an argument, that argument is said to be absent. The classification lazy is not the opposite of strict (as it is in Mycroft's analyser) but instead represents a position of uncertainty between strict and absent. A lazy argument might be evaluated, but the strictness analyser could not predict that it certainly would be. If my strictness analyser finds that a function needs to know the value of an argument, but the function is found not to terminate, that argument is said to be dangerous. An argument that is absent or dangerous need not be evaluated, since it can never be used. In the case of a dangerous argument an implementation should print an error message such as "Program did not terminate" when that argument is needed, and stop. These additional classifications allow better machine code to be produced.

2.2 An overview of the analyser

This section is intended to give an overview of the strictness analyser. Detailed explanations are given in subsequent sections.

The program

The strictness analyser determines the strictness properties of a program written as combinator definitions. The transformation from an ordinary functional language

or the lambda calculus to combinator definitions is quite straightforward using Hughes' algorithm [Hughes 82]. The program to be analysed has the form

$$\begin{aligned} F_1 \quad & = \quad expression_1 \\ & \cdots \\ F_N \quad & = \quad expression_N \\ expression_r & \end{aligned}$$

There are some combinator definitions followed by a main expression, $expression_r$, which makes use of these definitions. The value of the whole program is given by $expression_r$. The expressions are given in the lambda calculus with the addition of an If construct.

Since they are the bodies of combinators, none of $expression_1 \ldots expression_N$ has free variables other than $F_1 \ldots F_N$. Hughes' algorithm produces combinators with abstractions only at the top level. My analyser will, however, accept abstractions anywhere within the body of a combinator.

It should be noted that the naming of functions using the '=' sign is not as described in connection with the lambda calculus in the previous chapter. The function names $F_1 \ldots F_N$ are available for use anywhere in the whole program. In particular, F_n can be used in $expression_n$, giving recursive definition directly without the use of Y.

A further difference between the language accepted by the strictness analyser and the pure lambda calculus is the introduction of the If construct.

If $expression_1$
Then $expression_2$
Else $expression_3$
Fi

is equivalent to

$$If\ expression_1\ expression_2\ expression_3$$

as defined in the last chapter.

The *If* function is a very common higher order function. It is so important to implement this well that all strictness analysers must treat this particular higher order function specially. *If* has two significant properties:

1. $expression_1$ will certainly be evaluated.

2. Neither $expression_2$ nor $expression_3$ will certainly be evaluated, but the result of the whole function must be either one or the other.

My analyser takes these facts into account when processing an If construct.

The method

The strictness analyser assigns a *mode* to every combinator in a program, describing the manner in which that combinator uses its arguments (lazily, strictly and so on). There are four different ways in which a function can use an argument, and for each argument that a function takes there is a *usage* in the mode of the function which specifies one of these ways. The four possible usages are S for strict, L for lazy, A for absent and D for dangerous. The syntax of a mode is:

$$mode \;=\; \begin{cases} usage \rightarrow mode \\ \mathsf{m_L} \\ \mathsf{m_D} \end{cases}$$

The syntax of a usage is

$$usage \;=\; \begin{cases} \mathsf{S} \\ \mathsf{L} \\ \mathsf{D} \\ \mathsf{A} \end{cases}$$

If the mode of a function ends in $\mathsf{m_D}$ or contains the usage D this means that it will not return a value — its execution will not terminate.

Built-in functions have predefined modes that are not changed by the strictness analyser. For example the built-in function to add two integers together has the mode

$$\mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L}$$

The two Ss indicate that the function is strict on both its arguments. The $\mathsf{m_L}$ after the second arrow indicates the the result does not have mode $\mathsf{m_D}$, and thus may terminate. Other than this $\mathsf{m_L}$ gives no information about an object.

The analyser determines the mode of a combinator by recursively examining the sub-expressions of the combinator's body. For each sub-expression the analyser derives a mode and a *usage set*. The usage set of an expression is a set of associations between bound variables and their usages in that expression. For example, if a and b are bound variables and F is a function with mode $\mathsf{S} \rightarrow \mathsf{L} \rightarrow \mathsf{m_L}$, then in the expression $(F\ a\ b)$, a has usage S and b has usage L. This is because a is the first argument to F, so its usage is the same as the first usage in the mode of F. Similarly b's usage is the same as the second usage in the mode of F. The usage set of $(F\ a\ b)$ is $\{a{:}\,\mathsf{S},\ b{:}\,\mathsf{L}\}$.

To find the mode and usage set of an expression the analyser combines the modes and usage sets of its sub-expressions using the operators Best, Worst and Choice. These are defined in section 2.4.

To determine the mode of a combinator it is necessary to know the modes of all of the combinators mentioned in its body. For recursive definitions an approximation to the combinator's own mode must be used when analysing its body. The mode m_D is used for this purpose. By iterating to obtain better approximations the analysis converges to a suitable mode for each recursive combinator. Examples of the analysis of recursive definitions are given in section 2.6.

2.3 The syntax of a program

The strictness analyser takes a program expressed as definitions of combinators in a notation based on the lambda calculus and determines the strictness properties of these combinators. The combinators cannot have any free variables, but they may use any other combinators and built-in functions. A program has the following form:

$$\begin{array}{lcl} F_1 & = & \mathcal{E}_1^{(0)} \\ & \cdots & \\ F_N & = & \mathcal{E}_N^{(0)} \\ \mathcal{E}^{(0)} & & \end{array}$$

A program consists of the definitions of the combinators $F_1 \ldots F_N$ followed by an expression using these definitions which is the value of the whole program. The superscript (0) in $\mathcal{E}^{(0)}$ is an attribute indicating that the expression $\mathcal{E}^{(0)}$ has no free variables. The notation $\mathcal{E}^{(n)}$ has the productions:

$$\mathcal{E}^{(n)} = \begin{cases} K & \text{constant or built-in function} \\ b_i & \text{bound variable, } 1 \le i \le n \\ F_i & \text{combinator, } 1 \le i \le N \\ \lambda b_{n+1}.\ \mathcal{E}^{(n+1)} & \text{abstraction} \\ \mathcal{E}^{(n)}\ \mathcal{E}^{(n)} & \text{application} \\ \mathsf{If}\ \mathcal{E}^{(n)} & \\ \mathsf{Then}\ \mathcal{E}^{(n)} & \\ \mathsf{Else}\ \mathcal{E}^{(n)} & \text{conditional} \\ \mathsf{Fi} & \end{cases}$$

This is similar to the lambda calculus, with some naming restrictions and the addition of If-Then-Else-Fi. This has defined the abstract syntax of a program. In the concrete syntax brackets have the same effect as in the lambda calculus. They do not alter the meaning of the expression they enclose, but simply ensure that function applications are parsed correctly.

The names bound in abstractions are restricted so that the first binding in an expression is always λb_1, the second is λb_2, and so on, numbered from the outside

of an expression inwards to its subexpressions. The superscript (n) in $\mathcal{E}^{(n)}$ governs what the next bound variable will be called: $\mathcal{E}^{(0)}$ can produce $\lambda b_1.\ \mathcal{E}^{(1)}$, $\mathcal{E}^{(1)}$ can produce $\lambda b_2.\ \mathcal{E}^{(2)}$, and so on.

A typical program written in the above language might be

$$
\begin{array}{lll}
F_1 & = & \lambda b_1.\ \lambda b_2.\ \ \mathsf{If}\ (\mathit{Equal}\ 10\ b_1) \\
 & & \quad\ \mathsf{Then}\ 666 \\
 & & \quad\ \mathsf{Else}\ F_1\ (\mathit{Plus}\ 1\ b_1)\ b_2 \\
 & & \quad\ \mathsf{Fi;} \\
F_1\ 1\ 42 & &
\end{array}
$$

'*Equal*', '*Plus*', '10', '666', and so on are all terminals of the production K, which gives constants and built-in functions. Note the use of brackets and of the semicolon to terminate a definition in the concrete syntax. It is not absolutely necessary to use this restricted naming scheme for combinators and bound variables, but it makes the description of the strictness analyser slightly simpler.

2.4 Modes and usages

There are four usages, forming a complete lattice:

$$
\mathit{usage} \ = \ \left\{ \begin{array}{ccc} & \mathsf{L} & \\ \mathsf{S} & & \mathsf{A} \\ & \mathsf{D} & \end{array} \right\}
$$

L stands for lazy, S for strict, A for absent and D for dangerous. A bound variable will have overall usage S in an expression if the strictness analyser finds that its value will certainly be required. It will have usage A if it is found that its value will certainly *not* be required. If it cannot be determined whether or not it is required, the usage will be L. If the strictness analyser finds that its value depends directly on itself, then its usage will be D, because its evaluation would not terminate.

The domain *mode* is defined by the following recursive domain equation:

$$
\mathit{mode} \ = \ \left\{ \begin{array}{c} \mathsf{m_L} \\ | \\ \mathsf{m_D} \end{array} \right\} \oplus (\mathit{usage} \otimes \mathit{mode})
$$

where $\oplus$ is the doubly coalesced sum and $\otimes$ is the coalesced product. $\oplus$ and $\otimes$ are defined as follows:

Let A, B be complete lattices with $\top_A \ = \ \bigsqcup A$ and so on. '$-$' denotes set difference.

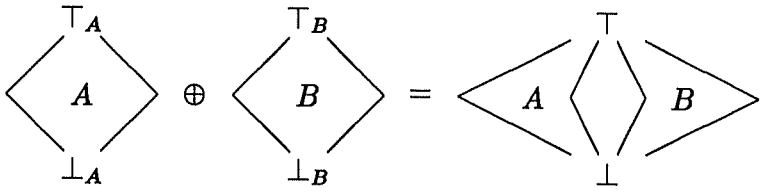
Then

$$\begin{aligned} A \oplus B \;=\; & \{(1,a) : a \in A - \{\bot_A, \top_A\}\} \cup \\ & \{(2,b) : b \in B - \{\bot_B, \top_B\}\} \cup \\ & \{(0,\top),(0,\bot)\} \end{aligned}$$

Ordered by

$$\begin{aligned} &(0,\bot) \sqsubseteq (1,a) \sqsubseteq (0,\top) \\ &(0,\bot) \sqsubseteq (2,b) \sqsubseteq (0,\top) \\ &(1,a_1) \sqsubseteq (1,a_2) \text{ Iff } a_1 \sqsubseteq a_2 \\ &(2,b_1) \sqsubseteq (2,b_2) \text{ Iff } b_1 \sqsubseteq b_2 \end{aligned}$$

So both the tops and the bottoms of A and B are coalesced:



And

$$A \otimes B \;=\; \{(a,b) : a \in A - \{\bot_A\}, b \in B - \{\bot_B\}\} \cup \{\bot\}$$

Ordered by

$$\begin{aligned} &\bot \sqsubseteq (a,b) \\ &(a_1,b_1) \sqsubseteq (a_2,b_2) \text{ Iff } a_1 \sqsubseteq a_2 \text{ and } b_1 \sqsubseteq b_2 \end{aligned}$$

Note that $\otimes$ only coalesces bottoms, not tops.

A mode of the form (*usage*, *mode*) is written as *usage* $\rightarrow$ *mode*. This is just syntactic sugar — such a mode is not meant to be a function, it just represents the strictness properties of a function. If an object with this mode is applied to an argument, then that argument will be used in the way indicated by *usage*.

The analyser needs to select usages and modes from the mode of a function so as to find out how it uses its arguments. To do this the notation $M_{[n]}$ and $M_{[n\ldots]}$ is used, where M is a mode. For example if a function F has mode $\mathsf{S} \rightarrow \mathsf{L} \rightarrow \mathsf{m_L}$, then its first argument has usage $(\mathsf{S} \rightarrow \mathsf{L} \rightarrow \mathsf{m_L})_{[1]}$, which simplifies to S. Similarly, its second argument has usage $(\mathsf{S} \rightarrow \mathsf{L} \rightarrow \mathsf{m_L})_{[2]}$ which is L. All functions are curried, so if F was applied to only one argument, the result would be a function taking one argument, and its mode would be $(\mathsf{S} \rightarrow \mathsf{L} \rightarrow \mathsf{m_L})_{[2\ldots]}$ which simplifies to $\mathsf{L} \rightarrow \mathsf{m_L}$.

Selection is defined as follows:

$$\begin{aligned} (\mathsf{m_L})_{[n]} &= \mathsf{L} \\ (\mathsf{m_D})_{[n]} &= \mathsf{D} \\ (U \rightarrow M)_{[1]} &= U \\ (U \rightarrow M)_{[n]} &= M_{[n-1]} \end{aligned}$$

So $M_{[n]}$ is the n^{th} usage in a mode.

$$\begin{array}{lcl} (\mathsf{m_L})_{[n\ldots]} & = & \mathsf{m_L} \\ (\mathsf{m_D})_{[n\ldots]} & = & \mathsf{m_D} \\ M_{[1\ldots]} & = & M \\ (U \rightarrow M)_{[n\ldots]} & = & M_{[n-1\ldots]} \end{array}$$

So $M_{[n\ldots]}$ is the part of M starting with its n^{th} usage.

$\mathsf{m_D}$ is the mode of a 'dangerous' object — its evaluation will definitely not terminate. If a function had mode $\mathsf{m_D}$ then its first argument would have usage $\mathsf{m_{D[1]}} = \mathsf{D}$, and so on for any other arguments. Note that because both sum and product coalesce the bottoms of the lattices any mode containing the usage D or ending with mode $\mathsf{m_D}$ is indistinguishable from $\mathsf{m_D}$ itself.

$\mathsf{m_L}$ is the mode of a 'safe' object — its evaluation may terminate, but it is not definitely known to terminate. If it is applied as a function the object is assumed to treat all its arguments lazily. For example, if an object had mode $\mathsf{m_L}$ and it was used as a function its first argument would have usage $\mathsf{m_{L[1]}} = \mathsf{L}$, and so on for any other arguments. Note that because the doubly coalesced sum identifies the tops of the lattices, $\mathsf{m_L}$, $\mathsf{L} \rightarrow \mathsf{m_L}$, $\mathsf{L} \rightarrow \mathsf{L} \rightarrow \mathsf{m_L}$ and so on are all indistinguishable, and equal to top.

Usage sets

The usage set of an expression is a set of associations between bound variables and usages. For each bound variable b_i the associated U_i indicates the overall usage of b_i in the expression.

An expression $\mathcal{E}^{(n)}$ as defined by the grammar in the previous section will have had n variables bound in the enclosing expression of which it is a subexpression. The only way that the superscript (n) attribute is incremented is when a lambda binding is crossed. Thus the expression $\mathcal{E}^{(n)}$ will have n bound variables, $b_1 \ldots b_n$, in its usage set, which will be written as:

$$\{b_1{:}U_1, \ \ldots \ , \ b_n{:}U_n\}$$

Best, Worst and Choice

These three operators on usages and modes are used in the formal description of the strictness analyser. Choice is defined on usages to be the least upper bound of its arguments:

$$U_1 \ \mathsf{Choice} \ U_2 \ = \ U_1 \sqcup U_2$$

In tabular form this is

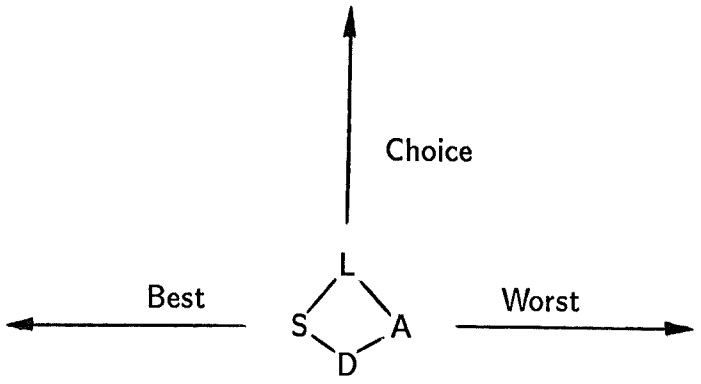
Choice	S	L	D	A
S	S	L	S	L
L	L	L	L	L
D	S	L	D	A
A	L	L	A	A

The idea of Best is that it gives the strictest of its arguments. Worst is exactly the opposite, and gives the most absent of its arguments. If one argument is L and the other D, then Best will give S, but Worst will give A. Best and Worst are defined by the tables

Best	S	L	D	A
S	S	S	S	S
L	S	L	S	L
D	S	S	D	D
A	S	L	D	A

Worst	S	L	D	A
S	S	L	D	A
L	L	L	A	A
D	D	A	D	A
A	A	A	A	A

A less ad-hoc way of looking at this is to imagine that Best takes the least upper bound of its arguments in the lattice of contexts turned on its side, so that S is at the top and A at the bottom. Worst is similar, but the lattice would be the other way up:



Choice is defined on modes to be the least upper bound of its arguments:

$$M_1 \text{ Choice } M_2 = M_1 \sqcup M_2$$

Best and Worst are not defined on modes.

The definitions of the operators on usages are extended to usage sets in a straightforward way. If *Op* is one of Best, Worst or Choice, U is a usage and S is the usage set $\{b_1{:}U_1,\ b_2{:}U_2,\ \ldots,\ b_n{:}U_n\}$ then

$$U\ Op\ S\ =\ \{b_1{:}U\ Op\ U_1,\ b_2{:}U\ Op\ U_2,\ \ldots,\ b_n{:}U\ Op\ U_n\}$$

If S' is the usage set $\{b_1{:}U'_1,\ b_2{:}U'_2,\ \ldots,\ b_n{:}U'_n\}$ then

$$S\ Op\ S' \ = \ \{b_1{:}U_1\ Op\ U'_1,\ b_2{:}U_2\ Op\ U'_2,\ \ldots,\ b_n{:}U_n\ Op\ U'_n\}$$

2.5 Strictness analysis

First I will define a function *Analyse*, which for any expression will return a pair containing the mode of that expression and its usage set. Note that the number of bound variables in the usage set and the names of the bound variables are always known, because of the restrictions on the syntax of expressions. As well as the expression to be analysed it is also necessary to supply *Analyse* with the modes of the user defined functions, $F_1 \ldots F_N$. Let M_i be the mode of F_i and M be the tuple of all these modes:

$$\mathrm{M} \ = \ (M_1,\ \ldots,\ M_N)$$

The value of *Analyse* $\mathrm{M}\ \mathcal{E}^{(n)}$ is defined as follows:

1. If $\mathcal{E}^{(n)}$ is a constant or built-in function, K, the result is

   $$(M_K,\ \{b_1{:}\mathsf{A},\ \ldots,\ b_n{:}\mathsf{A}\})$$

   The usage set contains only As because to find the value of K it is not necessary to evaluate any of the bound variables. M_K is a predefined mode associated with K. For example, an integer would have mode $\mathsf{m_L}$ and the integer addition function would have mode $\mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L}$.

2. If $\mathcal{E}^{(n)}$ is a bound variable, b_i, the result is

   $$(\mathsf{m_L},\ \{\ldots,\ \mathsf{A},\ \ldots,\ b_i{:}\mathsf{S},\ \ldots,\ \mathsf{A},\ \ldots\})$$

   All the usages in the usage set are A except for the one corresponding to b_i because none of the other bound variables is needed to find the value of b_i. The usage corresponding to b_i is S because b_i must be evaluated to find the value of b_i. The mode is $\mathsf{m_L}$ so that if b_i is used as a function it will be lazy on all its arguments, just as it would be if strictness analysis was not done and lazy evaluation was used throughout the program.

3. If $\mathcal{E}^{(n)}$ is a combinator, F_i, the result is

   $$(M_i,\ \{b_1{:}\mathsf{A},\ \ldots,\ b_n{:}\mathsf{A}\})$$

Where M_i is the i^{th} mode in the tuple M given as the first parameter to *Analyse*. This corresponds to the mode of F_i. Again the usage set is full of As because it is not necessary to evaluate any bound variables in order to find the value of F_i.

4. If $\mathcal{E}^{(n)}$ is an abstraction, $\lambda b_{n+1}.\ \mathcal{E}^{(n+1)}$, and

$$(M,\ S) \quad = \quad \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}^{(n+1)}$$

where

$$S = \{b_1{:}U_1,\ \ldots b_n{:}U_n,\ b_{n+1}{:}U_{n+1}\}$$

then the result is

$$(U_{n+1} \rightarrow M,\ \{b_1{:}U_1,\ \ldots b_n{:}U_n\})$$

In other words, a new mode is created for this abstraction, with U_{n+1} as its first usage. This usage is the overall usage of b_{n+1} in the body of the abstraction.

5. If $\mathcal{E}^{(n)}$ is a function application, $\mathcal{E}_1^{(n)}\ \mathcal{E}_2^{(n)}$, let

$$(M_1,\ S_1) \quad = \quad \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_1^{(n)}$$

and

$$(M_2,\ S_2) \quad = \quad \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_2^{(n)}$$

The result is then

$$(M_{1\,[2..]},\ S_1\ \mathsf{Best}\ (M_{1\,[1]}\ \mathsf{Worst}\ S_2))$$

When forming the new usage set, the usage set of $\mathcal{E}_2$ is adjusted using Worst to take into account the first usage in M_1. For example if $M_{1\,[1]} = \mathsf{A}$ then it does not matter if a bound variable is strict in S_2 — $\mathcal{E}_2$ will never be evaluated, so usages from this expression should not affect the overall usage set of the whole application. The new usage set is formed by combining the usage sets of the function and argument using Best, so that a bound variable will be strict in the new usage set if it is strict in either S_1 or in $M_{1\,[1]}$ Worst S_2, but only absent if it is absent in both.

Note that M_2 is not used. In this analyser the mode of a function's argument cannot affect the mode of its result.

6. If $\mathcal{E}^{(n)}$ is a conditional expression,

$$\begin{array}{l}\mathsf{If}\ \mathcal{E}_1^{(n)}\\ \mathsf{Then}\ \mathcal{E}_2^{(n)}\\ \mathsf{Else}\ \mathcal{E}_3^{(n)}\\ \mathsf{Fi}\end{array}$$

and

$$\begin{array}{rcl}(M_1,\ S_1) & = & \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_1^{(n)}\\ (M_2,\ S_2) & = & \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_2^{(n)}\\ (M_3,\ S_3) & = & \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_3^{(n)}\end{array}$$

the result is

$$(M_2\ \mathsf{Choice}\ M_3,\ S_1\ \mathsf{Best}\ (S_2\ \mathsf{Choice}\ S_3))$$

Note that M_1 is not used. This is because $\mathcal{E}_1^{(n)}$ cannot form part of the result, but instead determines which of $\mathcal{E}_2^{(n)}$ or $\mathcal{E}_3^{(n)}$ will be returned.

That concludes the analysis of expressions. It now remains to show how to use *Analyse* to analyse whole programs.

If the program under consideration is

$$\begin{array}{rcl}F_1 & = & \mathcal{E}_1^{(0)};\\ & \cdots & \\ F_N & = & \mathcal{E}_N^{(0)};\\ \mathcal{E}^{(0)} & & \end{array}$$

and M is a tuple of modes $(M_1, \ldots, M_N)$, let

$$\mathrm{Analyse\,M} \ = \ (\mathsf{left}\ (\mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_1^{(0)}),\ \ldots,\ \mathsf{left}\ (\mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_N^{(0)}))$$

where left is a function for extracting the first component of a pair.

This function takes the tuple of modes for the functions $F_1 \ldots F_N$ and returns a new tuple of modes. The body of each F_i is analysed, and the mode of the body placed in the new tuple at position i. The mode is in the left side of the pair returned by *Analyse* M, so left is used to retrieve this.

To find the modes of $F_1 \ldots F_N$ we start with

$$\mathrm{M}_0 \ = \ (\mathsf{m_D},\ \ldots,\ \mathsf{m_D})$$

so that each F_i is given the tentative mode $\mathsf{m_D}$ to start with, and then we iterate:

$$\mathrm{M}_{n+1} \ = \ \mathrm{Analyse\,M}_n$$

until $\mathrm{M}_{n+1} \ = \ \mathrm{M}_n$. This will terminate with M_n equal to the the least fixed point of the modes of $F_1, \ldots, F_N$ since

1. $M_0 = (m_D, \ldots, m_D)$ so Analyse starts at the bottom of the lattice of modes.

2. Analyse is monotonic because it is built from other monotonic functions: Choice, Best and Worst. It is clear that Choice is monotonic, since it takes the least upper bound of its arguments. It is somewhat surprising that Best and Worst are monotonic, but inspection of their definitions will show this to be the case. This means that if the analyser terminates it will find the least fixed point of the modes.

3. There are no infinite ascending chains in the lattice of modes, so the analyser will terminate. To show this define

$$mode_0 = \left\{ \begin{array}{c} m_L \\ | \\ m_D \end{array} \right\}$$

$$mode_{n+1} = \left\{ \begin{array}{c} m_L \\ | \\ m_D \end{array} \right\} \oplus \left(\left\{ \begin{array}{ccc} & L & \\ S & & A \\ & D & \end{array} \right\} \otimes mode_n \right)$$

So as n increases $mode_n$ approximates the lattice of modes as previously defined.

A chain is a set X such that for all $x_a, x_b \in X$, either $x_a \sqsubseteq x_b$ or $x_b \sqsubseteq x_a$. Now define $height\ (X) = m$ where $X = \{x_1, \ldots, x_{m+1}\}$. A complete lattice L must contain at least one chain. Let $height\ (L)$ be the length of the longest chain in L. Thus

$$height \left(\left\{ \begin{array}{c} m_L \\ | \\ m_D \end{array} \right\} \right) = 1$$

$$height \left(\left\{ \begin{array}{ccc} & L & \\ S & & A \\ & D & \end{array} \right\} \right) = 2$$

Now if A, B are complete lattices

$$\begin{array}{lcl} height\ (A \oplus B) & = & max(height\ (A), height\ (B)) \\ height\ (A \otimes B) & = & height\ (A) + height\ (B) - 1 \end{array}$$

So

$$\begin{array}{lcl} height\ (mode_0) & = & 1 \\ height\ (mode_{n+1}) & = & max(1, height\ (mode_n) + 1) \\ & = & height\ (mode_n) + 1 \end{array}$$

From this we can deduce

$$height\ (mode_n) \quad = \quad n+1$$

If a mode starts at $\mathsf{m_D}$ and increases monotonically, it must first increase to a mode M such that for some n, M is in $mode_n$. Since $height(mode_n) = n+1$ it can then increase at most n times more, so the analyser must terminate.

2.6 Examples

In this section I present some worked examples to illustrate the analyser. I will only deal with single functions, not whole programs, for simplicity. In what follows $(Equal\ b_1\ 0)$ means the same as $b_1 = 0$, but expressions are written in the first form so that the analysis is easier to follow. Both *Equal* and *Minus* are built-in functions of mode $\mathsf{S} \to \mathsf{S} \to \mathsf{m_L}$. As the first example, consider

$$\begin{array}{lll} F_n & = & \lambda b_1.\lambda b_2.\lambda b_3.\ \mathsf{If}\ (Equal\ b_1\ 0) \\ & & \quad \mathsf{Then}\ F_n\ (Minus\ b_1\ 1)\ b_2\ b_3 \\ & & \quad \mathsf{Else}\ b_3 \\ & & \quad \mathsf{Fi} \end{array}$$

Since $\mathrm{M_0} \quad = \quad (\mathsf{m_D},\ \ldots,\ \mathsf{m_D})$, the mode of F_n is initially set up as $\mathsf{m_D}$ in this recursive definition. Taking the If part first:

$$\begin{array}{lll} Analyse\ \mathrm{M_0}\ Equal & = & (\mathsf{S} \to \mathsf{S} \to \mathsf{m_L},\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}) \\ Analyse\ \mathrm{M_0}\ b_1 & = & (\mathsf{m_L},\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}) \end{array}$$

Using the rule for applications:

$$\begin{array}{l} Analyse\ \mathrm{M_0}\ (Equal\ b_1) \\ \quad = \quad ((\mathsf{S} \to \mathsf{S} \to \mathsf{m_L})_{[2\ldots]}, \\ \qquad \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}\ \mathsf{Best} \\ \qquad ((\mathsf{S} \to \mathsf{S} \to \mathsf{m_L})_{[1]}\ \mathsf{Worst}\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\})) \\ \quad = \quad (\mathsf{S} \to \mathsf{m_L},\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}) \end{array}$$

Now

$$Analyse\ \mathrm{M_0}\ 0 \quad = \quad (\mathsf{m_L},\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\})$$

So using the rule for applications again

$$\begin{array}{l} Analyse\ \mathrm{M_0}\ (Equal\ b_1\ 0) \\ \quad = \quad ((\mathsf{S} \to \mathsf{m_L})_{[2\ldots]}, \\ \qquad \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}\ \mathsf{Best} \\ \qquad ((\mathsf{S} \to \mathsf{m_L})_{[1]}\ \mathsf{Worst}\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\})) \\ \quad = \quad (\mathsf{m_L},\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}) \end{array}$$

Now the Then part:

$$\mathit{Analyse}\ \mathrm{M}_0\ F_n \ = \ (\mathsf{m_D},\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\})$$

(*Minus* b_1 1) is exactly the same as (*Equal* b_1 0) as far as strictness is concerned, because *Minus* is also strict on both its arguments. Thus

$$\mathit{Analyse}\ \mathrm{M}_0\ (\mathit{Minus}\ b_1\ 1) \ = \ (\mathsf{m_L},\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\})$$

Using the rule for applications

$$\begin{aligned}&\mathit{Analyse}\ \mathrm{M}_0\ F_n\ (\mathit{Minus}\ b_1\ 1)\\ &\quad = \ (\mathsf{m_{D[2...]}},\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}\ \mathsf{Best}\ (\mathsf{m_{D[1]}}\ \mathsf{Worst}\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}))\\ &\quad = \ (\mathsf{m_D},\ \{b_1{:}\mathsf{D},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\})\end{aligned}$$

Now

$$\mathit{Analyse}\ \mathrm{M}_0\ b_2 \ = \ (\mathsf{m_L},\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{S},\ b_3{:}\mathsf{A}\})$$

So in similar fashion

$$\begin{aligned}&\mathit{Analyse}\ \mathrm{M}_0\ F_n\ (\mathit{Minus}\ b_1\ 1)\ b_2\\ &\quad = \ (\mathsf{m_{D[2...]}},\ \{b_1{:}\mathsf{D},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}\ \mathsf{Best}\ (\mathsf{m_{D[1]}}\ \mathsf{Worst}\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{S},\ b_3{:}\mathsf{A}\}))\\ &\quad = \ (\mathsf{m_D},\ \{b_1{:}\mathsf{D},\ b_2{:}\mathsf{D},\ b_3{:}\mathsf{A}\})\end{aligned}$$

And again

$$\mathit{Analyse}\ \mathrm{M}_0\ b_3 \ = \ (\mathsf{m_L},\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{S}\})$$

So

$$\begin{aligned}&\mathit{Analyse}\ \mathrm{M}_0\ F_n\ (\mathit{Minus}\ b_1\ 1)\ b_2\ b_3\\ &\quad = \ (\mathsf{m_{D[2...]}},\ \{b_1{:}\mathsf{D},\ b_2{:}\mathsf{D},\ b_3{:}\mathsf{A}\}\ \mathsf{Best}\ (\mathsf{m_{D[1]}}\ \mathsf{Worst}\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{S}\}))\\ &\quad = \ (\mathsf{m_D},\ \{b_1{:}\mathsf{D},\ b_2{:}\mathsf{D},\ b_3{:}\mathsf{D}\})\end{aligned}$$

The Else part is just b_3, and

$$\mathit{Analyse}\ \mathrm{M}_0\ b_3 \ = \ (\mathsf{m_L},\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{S}\})$$

as before.

We now have three modes and three usage sets, one for each part of the conditional. Combining these, we find:

$$\begin{aligned}&\mathit{Analyse}\ \mathrm{M}_0\ \mathsf{If}\ldots\mathsf{Then}\ldots\mathsf{Else}\ldots\mathsf{Fi}\\ &\quad = \ (\mathsf{m_D}\ \mathsf{Choice}\ \mathsf{m_L},\\ &\qquad \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{A}\}\ \mathsf{Best}\\ &\qquad\ (\{b_1{:}\mathsf{D},\ b_2{:}\mathsf{D},\ b_3{:}\mathsf{D}\}\ \mathsf{Choice}\ \{b_1{:}\mathsf{A},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{S}\}))\\ &\quad = \ (\mathsf{m_L},\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A},\ b_3{:}\mathsf{S}\})\end{aligned}$$

Now we reach the lambda abstractions, and unpick the usage set so as to construct a new mode:

$$\begin{aligned}
\mathit{Analyse}\ \mathrm{M}_0\ \lambda b_3.\ \mathsf{If}\ldots\mathsf{Fi} &= (\underbrace{\mathsf{S}}_{b_3} \rightarrow \mathsf{m_L},\ \{b_1{:}\mathsf{S},\ b_2{:}\mathsf{A}\}) \\
\mathit{Analyse}\ \mathrm{M}_0\ \lambda b_2.\ \lambda b_3.\ \mathsf{If}\ldots\mathsf{Fi} &= (\underbrace{\mathsf{A}}_{b_2} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L},\ \{b_1{:}\mathsf{S}\}) \\
\mathit{Analyse}\ \mathrm{M}_0\ \lambda b_1.\ \lambda b_2.\ \lambda b_3.\ \mathsf{If}\ldots\mathsf{Fi} &= (\underbrace{\mathsf{S}}_{b_1} \rightarrow \mathsf{A} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L},\ \{\,\})
\end{aligned}$$

Thus the new mode of F_n is $\mathsf{S} \rightarrow \mathsf{A} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L}$. I will not go through the working, but if this mode is used instead of $\mathsf{m_D}$ in the second iteration of the analyser, the mode of F_n will still be $\mathsf{S} \rightarrow \mathsf{A} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L}$ after that iteration. The analyser has thus found the fixed point of the mode of F_n. Inspection of the function will show that this mode expresses the correct strictness properties.

As another example, consider

$$F_m \;=\; \lambda b_1.\ F_m\ b_1$$

F_m is defined to be a function that takes a single argument and applies itself recursively to that argument. As before, $\mathrm{M}_0 \;=\; (\mathsf{m_D},\ \ldots,\ \mathsf{m_D})$ so the mode of F_m is initially $\mathsf{m_D}$.

$$\begin{aligned}
\mathit{Analyse}\ \mathrm{M}_0\ F_m &= (\mathsf{m_D},\ \{b_1{:}\mathsf{A}\}) \\
\mathit{Analyse}\ \mathrm{M}_0\ b_1 &= (\mathsf{m_L},\ \{b_1{:}\mathsf{S}\})
\end{aligned}$$

F_m is applied as a function, so

$$\begin{aligned}
\mathit{Analyse}\ \mathrm{M}_0\ F_m\ b_1 &= (\mathsf{m_{D[2...]}},\ \{b_1{:}\mathsf{A}\}\ \mathsf{Best}\ (\mathsf{m_{D[1]}}\ \mathsf{Worst}\ \{b_1{:}\mathsf{S}\})) \\
&= (\mathsf{m_D},\ \{b_1{:}\mathsf{D}\})
\end{aligned}$$

Thus

$$\mathit{Analyse}\ \mathrm{M}_0\ \lambda b_1.\ F_m\ b_1 \;=\; (\underbrace{\mathsf{D}}_{b_1} \rightarrow \mathsf{m_D},\ \{\,\})$$

However, $\mathsf{D} \rightarrow \mathsf{m_D}$ is equivalent to $\mathsf{m_D}$, because of the way the lattice of modes is defined. The new mode of F_m is thus $\mathsf{m_D}$. On subsequent iterations this would not change, so the fixed point of F_m's mode is $\mathsf{m_D}$. This is entirely appropriate for a function that does not terminate.

2.7 Mycroft's strictness analyser

Mycroft's analyser uses abstract interpretation to determine the strictness properties of a program. The analyser described in [Mycroft 81] is only valid for first

order functions over flat domains, but it has recently been extended to higher order functions [Abramsky et al 85, Hudak & Young 85]. There are still some restrictions on the kinds of function these new analysers are suitable for. I will describe these extensions in the next chapter, along with the extension of my own algorithm to second order functions.

The idea of abstract interpretation can be illustrated by the now traditional example of the rule of signs. This is a simple rule of arithmetic taught to every schoolchild. When two numbers are multiplied together, to find the sign of the result it is sufficient to consider only the signs of the numbers being multiplied. Thus:

$$(+) \times^{\#} (-) = (-) \qquad (+) \times^{\#} (+) = (+)$$
$$(-) \times^{\#} (-) = (+) \qquad (-) \times^{\#} (-) = (-)$$

The '#'s above the multiplication signs indicate that these operations are defined not in the normal domain of integers, but in the *abstract domain* $\{(+), (-)\}$, containing only two points. The negative numbers are represented by $(-)$ and the positive numbers by $(+)$. I ignore zero here. The function ABS takes integers to this abstract domain and $\times^{\#}$ is a version of the multiplication function for values in the abstract domain. Now

$$ABS\,(a \times b) = (ABS\,a) \times^{\#} (ABS\,b)$$

So, from the value returned by the abstract multiplication function we can infer something (namely the sign) about the value returned by the real multiplication function. It is precisely this ability to infer properties about the real domain by studying the behaviour of objects in the abstract domain that makes abstract interpretation suitable for strictness analysis. This treatment has glossed over many of the finer points of abstract interpretation, but it is my aim to give the intuitions behind it rather than rigorous definitions.

The idea behind Mycroft's strictness analyser is that a function is considered strict in an argument if it fails to terminate when that argument fails to terminate. The appropriate abstract interpretation is defined for the two point abstract domain $\{0, 1\}$ ordered by $0 \sqsubseteq 1$. If x is a value in the real domain, define:

$$ABS\,x = \begin{cases} 0 & \text{If } x \text{ necessarily fails to terminate} \\ 1 & \text{If } x \text{ may terminate} \end{cases}$$

Now, from the definition of a function f in the program being analysed we can find the function $f^{\#} = ABS\,f$, the function in the abstract domain corresponding to f. Here ABS is a function taking expressions in the real domain to expressions in

the abstract domain. If $f^{\#}\ 0\ =\ 0$ we can deduce that $f\ \perp\ =\ \perp$, so we can regard the function as strict on its argument ($\perp$ is the non-terminating computation).

The abstraction function for non-recursive definitions is not complex:

- ABS constant $=$ 1 because constants always terminate.
- $ABS\ x\ =\ x$ when x is a variable.
- $ABS\ (Op\ a\ b)\ =\ (ABS\ a)$ AND $(ABS\ b)$, when Op is a built-in operator such as multiplication, which is strict on its arguments.
- $ABS\ f\ =\ f^{\#}$ when f is a function.
- $ABS\ \lambda x.\ e\ =\lambda x.\ ABS\ e$
- $ABS\ (a\ b)\ =(ABS\ a)\ (ABS\ b)$
- ABS If a Then b Else c Fi $=\ (ABS\ a)$ AND $((ABS\ b)$ OR $(ABS\ c))$

The operators AND and OR are the usual logical operators:

$$x \text{ AND } y\ =\ \begin{cases} 1 & \text{If } x = y = 1 \\ 0 & \text{Otherwise} \end{cases}$$

$$x \text{ OR } y\ =\ \begin{cases} 0 & \text{If } x = y = 0 \\ 1 & \text{Otherwise} \end{cases}$$

For example the function

$$f\ =\ \lambda x.\ \lambda y.\ x + y$$

has the corresponding function in the abstract domain

$$f^{\#}\ =\ \lambda x.\ \lambda y.\ x \text{ AND } y$$

When we come to recursive functions we again use an iterative method to find the least fixed point of the function in the abstract domain. For example

$$\begin{aligned} f\ &=\ \lambda x.\ \lambda y.\ \text{If } x \text{ Then } y \text{ Else } f\ (x-1)\ y \text{ Fi} \\ f^{\#}\ &=\ \lambda x.\ \lambda y.\ x \text{ AND } (y \text{ OR } (f^{\#}\ x\ y)) \end{aligned}$$

To find the least fixed point we should use as first approximation to $f^{\#}$ the least function of two arguments in the abstract domain. Since our ordering is $0 \sqsubseteq 1$, the least function is $\lambda x.\ \lambda y.\ 0$. This function can be thought of as the "most optimistic", since it is strict on both its arguments. Using this approximation we obtain:

$$f^{\#} \quad = \quad \lambda x.\ \lambda y.\ x \text{ AND } y$$

as our second approximation. All subsequent approximations are the same. Detecting that the fixed point has been found can be laborious, since it involves comparing functions. There are several plausible looking short-cuts for doing this that fail in some circumstances. I am not going to discuss this further, but a good treatment can be found in [Clack & Peyton Jones 85b].

The whole point of finding these functions in the abstract domain is to feed them values in the abstract domain and observe the results. By doing this it is possible to infer the strictness properties of the real functions and discover what I would call their modes. One way to compare my algorithm to Mycroft's would be to say that each of Mycroft's functions in the abstract domain corresponds to a number of my modes. Each characterises the real function, but my algorithm draws finer distinctions since a lazy argument in Mycroft's algorithm could be L or A in mine, and a strict one could be S or D. My algorithm finds the least fixed point of the modes, while Mycroft's finds the least fixed point of the functions in the abstract domain, but these things are very closely related.

However, in pathological cases involving non-terminating computations Mycroft's analyser sometimes finds functions to be strict on arguments when my analyser finds them to be absent or lazy. For instance, if we define

$$\begin{array}{lcll} \mathit{Loop} & = & \lambda x.\ \mathit{Loop}; & \\ F_1 & = & \lambda x.\ \lambda y. & \mathsf{If}\ x = 0 \\ & & & \mathsf{Then}\ y \\ & & & \mathsf{Else}\ \mathit{Loop}\ y \\ & & & \mathsf{Fi}; \end{array}$$

$$\begin{array}{lcll} F_2 & = & \lambda x.\ \lambda y. & \mathsf{If}\ x = 0 \\ & & & \mathsf{Then}\ y \\ & & & \mathsf{Else}\ \mathit{Loop}\ x \\ & & & \mathsf{Fi}; \end{array}$$

Then these functions have the modes:

Loop : $\mathsf{m_D}$

F_1 : $\mathsf{S} \to \mathsf{S} \to \mathsf{m_L}$

F_2 : $\mathsf{S} \rightarrow \mathsf{L} \rightarrow \mathsf{m_L}$

Now F_1 and F_2 behave in the same way, in that neither *Loop x* nor *Loop y* will terminate, and yet they are given different modes by my analyser. Mycroft's analyser will find both F_1 and F_2 to be strict on their second arguments, but my analyser finds F_2 lazy on its second argument.

Although it is perfectly acceptable to say that the second argument to the function may be evaluated eagerly in both cases, it should be noted that it is certainly not the case that y is strict (that is to say *necessarily* evaluated) in the expression *Loop x*. The deduction that it is acceptable to evaluate the second argument of F_2 eagerly must come from reasoning about the termination properties of F_2, rather than directly from the strictness properties. It is thus outside the scope of my strictness analyser to deduce that F_2 may evaluate its second argument eagerly.

It is more efficient to evaluate both arguments eagerly, and of course the Else part will only be executed in exceptional circumstances, when something has gone wrong with the program anyway. However, evaluating both arguments does have a disadvantage. Having detected that *Loop* has the mode $\mathsf{m_D}$, it would be perverse of a complier to generate code that actually did loop forever. It would be much more helpful for the program to return an error value when an attempt was made to evaluate an argument with usage D or to return a value of mode $\mathsf{m_D}$. The Ponder system prints an error message and then stops in these circumstances.

If eager evaluation is used in cases like those above, and a program does crash it may be because of evaluating an expression that would not have been evaluated using normal order reduction. This might be confusing if the programmer believes that he can deduce useful debugging information from the error messages that are printed. For instance, from the message printed by the run-time system when a dangerous object is evaluated it may be possible to determine whether the particular use of *Loop* in F_1 was evaluated, and hence whether $y \neq 0$. If F_1 evaluates its second argument eagerly, it is possible that this argument may itself crash, thus making the cause of the program's failure rather obscure.

In all normal cases my analyser is a significant improvement on that of Mycroft.

- My analyser accepts all programs, not just first order programs.
- The extra classifications *absent* and *dangerous* allow better machine code to be generated.
- My analyser finds the fixed points of the modes of functions quickly and

easily, in contrast to Mycroft's analyser [Clack & Peyton Jones 85b].

Chapter 3

Higher Order Strictness Analysis

3.1 Introduction

The analyser presented in the previous chapter can be used on programs containing higher order functions, but functional arguments are always treated completely lazily. For instance define

$$F \;=\; \lambda a.\, \lambda b.\, \lambda f.\, f\ a\ b$$

The mode of f will be $\mathsf{m_L}$, because all bound variables are given the mode $\mathsf{m_L}$. The a and b in the body of F will find themselves used lazily and F will have the mode $\mathsf{L} \to \mathsf{L} \to \mathsf{S} \to \mathsf{m_L}$. Now consider the application of F in the body of another combinator:

$$G \;=\; \lambda x.\, F\ x\ 3\ \mathit{Plus}$$

If we use the above mode for F then in the body of G the built-in function *Plus* will be used strictly, but x and 3 will be used lazily. Thus the mode of G will be $\mathsf{L} \to \mathsf{m_L}$. We can inspect the definition of F and see that the body of G could be reduced:

$$(F\ x\ 3\ \mathit{Plus}) \xrightarrow{\beta} (\mathit{Plus}\ 3\ x)$$

We would now say that G should be strict on its argument, since in its body x is definitely evaluated.

In fact we don't actually need to look at the definition of F at all, provided we know a few crucial facts about how F uses its arguments. In particular we need to know that the usages of a and b depend on the mode of f. F will be strict on its first argument if the function actually given as its third argument is strict on *its* first argument. F will be strict on its second argument if the function supplied as its third argument is strict on *its* second argument.

To express these dependencies some new notation must be introduced. The notation $\mathcal{A}i$ is used to refer to the mode of the i^{th} argument to a function. The new mode of F would be

$$\mathcal{A}3_{[1]} \rightarrow \mathcal{A}3_{[2]} \rightarrow \mathsf{S} \rightarrow \mathcal{A}3_{[3...]}$$

When F is applied to some arguments $\mathcal{A}3$ must be replaced by the actual mode of F's third argument. Let us consider the body of of G, as previously defined:

$$F\ x\ 3\ \mathit{Plus}$$

The third argument of F is the function *Plus*. This has mode $\mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L}$, so we can expand out the mode of F for this particular application as follows:

$$\mathcal{A}3_{[1]} \rightarrow \mathcal{A}3_{[2]} \rightarrow \mathsf{S} \rightarrow \mathcal{A}3_{[3...]}$$

becomes

$$(\mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L})_{[1]} \rightarrow (\mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L})_{[2]} \rightarrow \mathsf{S} \rightarrow (\mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L})_{[3...]}$$

which simplifies to

$$\mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L}$$

Note that it is only for this particular application of F that the mode is changed in this way. F's mode must be filled in with the appropriate mode for $\mathcal{A}3$ separately for each different application in the program. In this case F's first argument ends up being treated strictly, so we would be able to determine that G uses its bound variable x strictly, and we would have found G to be a function strict on its first argument.

This extension to the earlier strictness analyser will often do no better than that analyser when presented with higher order functions whose functional arguments are themselves higher order functions. The reason for this is discussed in the next section, which describes the details of this new "second order" analyser. The following sections give detailed examples of the second order analyser in use and compare it with higher order analysers based on Mycroft's work. I conclude the chapter by briefly discussing strictness analysis of lazy lists.

3.2 Changes to the analyser

Modes and usages

Modes may now contain $\mathcal{A}i$ and $\mathcal{B}i$. $\mathcal{A}i$ has been described above — the modes of higher order functions contain $\mathcal{A}i$ in the places where the mode of a functional argument should be substituted. The modes of higher order functions can be regarded

as functions on modes, since from the modes of the arguments in a particular application they give a mode for the higher order function in that application.

$\mathcal{B}i$ is a "dummy variable" used by the strictness analyser to stand for the mode of the bound variable b_i. The analyser transforms all $\mathcal{B}i$s into $\mathcal{A}i$s as the mode of a combinator is constructed. The modes of combinators cannot contain $\mathcal{B}i$ since this is the mode of a variable, and combinators have no free variables.

The subscripts used to select usages and modes from a mode are extended to $\mathcal{A}i$ and $\mathcal{B}i$ in a straightforward way . For instance $\mathcal{A}i_{[n...]}$ cannot be simplified, but $\mathcal{A}i_{[n...]\,[m]}$ can be simplified:

$$\begin{array}{lcl} M_{[n...]\,[m]} & = & M_{[n+m-1]} \\ M_{[n...]\,[m...]} & = & M_{[n+m-1...]} \\ (M_1\ Op\ M_2)_{[n]} & = & (M_{1\,[n]}\ Op\ M_{2\,[n]}) \\ (M_1\ Op\ M_2)_{[n...]} & = & (M_{1\,[n...]}\ Op\ M_{2\,[n...]}) \end{array}$$

where M is $\mathcal{A}i$ or $\mathcal{B}i$ or any other mode that cannot be simplified, as described below, and Op is one of Best, Worst or Choice. Op will be used to stand for one of Best, Worst or Choice throughout the rest of this chapter.

Best, Worst and Choice

Some changes to Best, Worst and Choice are necessary to cope with the new modes. For instance $\mathcal{A}2_{[1]}$ Best L, cannot be simplified — the result depends on what $\mathcal{A}2_{[1]}$ is replaced by in a particular application.

The expression would remain unsimplified and could end up in the mode of a function:

$$\cdots \rightarrow (\mathcal{A}2_{[1]}\ \mathsf{Best}\ \mathsf{L}) \rightarrow \cdots$$

However, many expressions involving $\mathcal{A}i$ and $\mathcal{B}i$ can be simplified. For the second order analyser to terminate it is necessary for the following transformations to be used, so that modes and usages can be transformed into a standard form and thus compared easily.

$U_1\ Op\ U_2 = U_2\ Op\ U_1$	Commutivity
$U_1\ Op\ (U_2\ Op\ U_3) = (U_1\ Op\ U_2)\ Op\ U_3$	Associativity
$U_1\ Op\ U_1 = U_1$	Idempotence
U_1 Best (U_1 Worst U_2) $= U_1$	Absorption
U_1 Worst (U_1 Best U_2) $= U_1$	
S Best U_1 = S	'Top'
A Worst U_1 = A	
L Choice U_1 = L	
A Best U_1 $= U_1$	'Bottom'
S Worst U_1 $= U_1$	
D Choice U_1 $= U_1$	

Commutivity and associativity allow expressions to be transformed, but leave them the same size. The other transformations are used to make them smaller. In the Ponder compiler, usage expressions are simplified to the form

$$U_1\ Op\ (U_2\ Op\ \ldots\ (U_{n-1}\ Op\ U_n)\ \ldots)$$

by using Commutivity to re-order $U_1 \ldots U_n$, Associativity to flatten out subexpressions of the form $(U_a\ Op\ U_b)$, and Idempotence to remove duplicates. This last process is most easily done by using some arbitrary ordering on expressions, and sorting the $U_1 \ldots U_n$ using this order. $U_1 \ldots U_n$ can of course be composite expressions (which must also be simplified to the standard form) but they will not have *Op* at the top level, because of the Associativity transformation. If more than one of $U_1 \ldots U_n$ is an ordinary usage (L, S, D or A) then these should be combined using the ordinary definition of *Op*. The expression should then be further simplified by the application of the Absorption, 'Top' and 'Bottom' transformations.

Modes are treated similarly, but are somewhat simpler because the only operator is Choice. The transformations needed for modes are

M_1 Choice M_2 = M_2 Choice M_1	Commutivity
M_1 Choice (M_2 Choice M_3) = (M_1 Choice M_2) Choice M_3	Associativity
M_1 Choice M_1 = M_1	Idempotence
m_L Choice M_1 = m_L	'Top'
m_D Choice M_1 = M_1	'Bottom'

Analyse

Analyse takes as an extra parameter a list of the modes of the arguments to which the expression being analysed is applied. For example

$$\mathit{Analyse}\ \mathrm{M}\ \mathcal{E}^{(n)}\ [m_1;\ m_2;\ \ldots;\ m_j]$$

where the expression $\mathcal{E}^{(n)}$ will be applied to j arguments with modes $m_1 \ldots m_j$. The arguments to a function must be analysed before the function, so that this list can be built up. If the expression is a combinator its mode can be *instantiated* from this list, using m_i to substitute for $\mathcal{A}i$ in the mode of the combinator.

Analyse is nearly the same as before:

$$\text{Analyse}\ \mathrm{M}\ =\ (\mathsf{left}\ (\mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_1^{(0)}\ [\,]),\ \ldots,\ \mathsf{left}\ (\mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_N^{(0)}\ [\,]))$$

where $\mathcal{E}_1^{(0)} \ldots \mathcal{E}_N^{(0)}$ are as before and [ ] is the empty list of modes.
The value of $\mathit{Analyse}\ \mathrm{M}\ \mathcal{E}^{(n)}\ [m_1;\ m_2;\ \ldots;\ m_j]$ is defined as follows:

1. If $\mathcal{E}^{(n)}$ is a constant or built-in function, K, the result is

   $$(M_K,\ \{b_1{:}\mathsf{A},\ \ldots,\ b_n{:}\mathsf{A}\})$$

   where M_K is the predefined mode associated with K as in the previous chapter.

2. If $\mathcal{E}^{(n)}$ is a bound variable, b_i, the result is

   $$(\mathcal{B}i_{[1\ldots]},\ \{\ldots,\ \mathsf{A},\ \ldots,\ b_i{:}\mathsf{S},\ \ldots,\ \mathsf{A},\ \ldots\})$$

   $\mathcal{B}i$ is a "dummy variable" standing for the mode that b_i will be given.

3. If $\mathcal{E}^{(n)}$ is an abstraction, $\lambda b_{n+1}.\ \mathcal{E}^{(n+1)}$, and

   $$(M,\ S)\ =\ \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}^{(n+1)}\ [m_2;\ \ldots;\ m_j]$$

   Where

   $$S\ =\ \{b_1{:}U_1,\ \ldots b_n{:}U_n,\ b_{n+1}{:}U_{n+1}\}$$

And *shift* and *shift'* are defined by

$$
\begin{aligned}
&\mathit{shift}\ \mathsf{m_L} &&= \mathsf{m_L}\\
&\mathit{shift}\ \mathsf{m_D} &&= \mathsf{m_D}\\
&\mathit{shift}\ \mathcal{B}(n+1)_{[p\ldots]} &&= \mathcal{A}1_{[p\ldots]}\\
&\mathit{shift}\ \mathcal{B}x_{[p\ldots]} &&= \mathcal{B}x_{[p\ldots]} \quad \text{when } x = 1\ldots n\\
&\mathit{shift}\ \mathcal{A}m_{[p\ldots]} &&= \mathcal{A}(m+1)_{[p\ldots]}\\
&\mathit{shift}\ (M_1\ \mathit{Op}\ M_2) &&= (\mathit{shift}\ M_1)\ \mathit{Op}\ (\mathit{shift}\ M_2)\\
&\mathit{shift}\ U \to M &&= (\mathit{shift'}\ U) \to (\mathit{shift}\ M)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{shift'}\ U &&= U \quad \text{when } U = \mathsf{L}, \mathsf{S}, \mathsf{A} \text{ or } \mathsf{D}\\
&\mathit{shift'}\ \mathcal{B}(n+1)_{[p]} &&= \mathcal{A}1_{[p]}\\
&\mathit{shift'}\ \mathcal{B}x_{[p]} &&= \mathcal{B}x_{[p]} \quad \text{when } x = 1\ldots n\\
&\mathit{shift'}\ \mathcal{A}m_{[p]} &&= \mathcal{A}(m+1)_{[p]}\\
&\mathit{shift'}\ (M_1\ \mathit{Op}\ M_2) &&= (\mathit{shift'}\ M_1)\ \mathit{Op}\ (\mathit{shift'}\ M_2)
\end{aligned}
$$

The result is then

$$((\mathit{shift'}\ U_{n+1}) \to (\mathit{shift}\ M),\ \{b_1\colon(\mathit{shift'}\ U_1),\ \ldots b_n\colon(\mathit{shift'}\ U_n)\})$$

The effect of this change in modes is that references to the variable just bound, $\mathcal{B}(n+1)$, will now refer to the first argument of the function, $\mathcal{A}1$. All other references to the function's arguments have been incremented from $\mathcal{A}m$ to $\mathcal{A}(m+1)$. This is because the function now takes another argument before all the ones its body took.

Note that when obtaining $(M,\ S)$ from the call to *Analyse*, the list of modes $[m_1;\ m_2;\ \ldots;\ m_j]$ was reduced in length by removing its first member, m_1. If the original list was empty, then it will not be possible to remove the first member, and in this case the empty list should be passed to the recursive call of *Analyse*.

4. If $\mathcal{E}^{(n)}$ is a function application, $\mathcal{E}_1^{(n)}\ \mathcal{E}_2^{(n)}$, let

$$(M_2,\ S_2) \quad = \quad \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_2^{(n)}\ [\,]$$

and

$$(M_1,\ S_1) \quad = \quad \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_1^{(n)}\ [M_2,\ m_1;\ \ldots;\ m_j]$$

The result is then

$$(M_{1\,[2\ldots]},\ S_1\ \mathsf{Best}\ (M_{1\,[1]}\ \mathsf{Worst}\ S_2))$$

Note that the list of modes in the first call of *Analyse* is empty, and that $\mathcal{E}_2^{(n)}$ is dealt with first because its mode, M_2, must be placed at the front of the list of modes given to the second call of *Analyse*.

5. If $\mathcal{E}^{(n)}$ is a combinator, F_i, the result is

$$(\mathit{instantiate}\ M_i\ [m_1;\ \ldots;\ m_j],\ \{b_1{:}\mathsf{A},\ \ldots,\ b_n{:}\mathsf{A}\})$$

Where M_i is the i^{th} mode in the tuple M associated with *Analyse* M. This corresponds to the mode of F_i.

The subsidiary function *instantiate* replaces all occurrences of $\mathcal{A}p$ in M_i by m_p from the list of modes it is given. This is described below.

The modes of all the arguments given to F_i must be considered at the same time rather than being dealt with individually, because the mode of a later argument can determine the usage of an earlier one and vice versa.

6. If $\mathcal{E}^{(n)}$ is a conditional expression,

$$\begin{array}{l}\mathsf{If}\ \mathcal{E}_1^{(n)}\\ \mathsf{Then}\ \mathcal{E}_2^{(n)}\\ \mathsf{Else}\ \mathcal{E}_3^{(n)}\\ \mathsf{Fi}\end{array}$$

and

$$\begin{array}{rcl}(M_1,\ S_1) & = & \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_1^{(n)}\ [\,]\\ (M_2,\ S_2) & = & \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_2^{(n)}[m_1;\ \ldots;\ m_j]\\ (M_3,\ S_3) & = & \mathit{Analyse}\ \mathrm{M}\ \mathcal{E}_3^{(n)}[m_1;\ \ldots;\ m_j]\end{array}$$

the result is

$$(M_2\ \mathsf{Choice}\ M_3,\ S_1\ \mathsf{Best}\ (S_2\ \mathsf{Choice}\ S_3))$$

As before, M_1 is discarded because $\mathcal{E}_1^{(n)}$ determines which of $\mathcal{E}_2^{(n)}$ and $\mathcal{E}_3^{(n)}$ will be returned as the value of the conditional, but it does not form part of this value itself.

Instantiate

The subsidiary function *instantiate* is used to substitute for $\mathcal{A}i$ in the mode of a function, using m_i from the list of modes $[m_1;\ \ldots;\ m_j]$ that it is given. Note that there are j modes in this list.

$$\mathit{instantiate}\ M\ [m_1;\ \ldots;\ m_j]\ =\ \mathit{front}\ j\ M$$

Where *front* is defined as follows:

$$
\begin{array}{lcl}
\mathit{front}\ 0\ M & = & \mathit{back}\ M \quad \text{and } \mathit{back} \text{ is defined below} \\
\mathit{front}\ n\ \mathsf{m_L} & = & \mathsf{m_L} \\
\mathit{front}\ n\ \mathsf{m_D} & = & \mathsf{m_D} \\
\mathit{front}\ n\ M & = & (\mathit{front}'\ M_{[1]}) \rightarrow (\mathit{front}\ (n-1)\ M_{[2\ldots]})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{front}'\ U & = & U \quad \text{when } U = \mathsf{L}, \mathsf{S}, \mathsf{A}, \mathsf{D} \text{ or } \mathcal{B}i_{[p]} \\
\mathit{front}'\ \mathcal{A}i_{[p]} & = & \begin{cases} \mathsf{L} & \text{If } i > k \text{ or } m_{i\,[p]} = \mathcal{A}x_{[y]} \text{ for some } x,\ y \\ m_{i\,[p]} & \text{Otherwise} \end{cases} \\
\mathit{front}'\ (U_1\ \mathit{Op}\ U_2) & = & (\mathit{front}'\ U_1)\ \mathit{Op}\ (\mathit{front}'\ U_2)
\end{array}
$$

The purpose of *front* is to ensure that for each of the j arguments there will be a usage in the instantiated mode which is either L, S, A, D or $\mathcal{B}i_{[p]}$ The usage $\mathcal{A}i_{[p]}$ is not acceptable, since it is only meaningful as an instruction to perform a substitution in the mode M.

If there is an $\mathcal{A}i_{[p]}$ in M where $i > j$ then there will be no corresponding mode in the list $[m_1;\ \ldots;\ m_j]$, so the only suitable replacement is L. For example if we define

$$F \quad = \quad \lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b$$

and apply F to only two arguments, there will be no mode to substitute for $\mathcal{A}3$ in the mode of F, since there is no third argument. $\mathcal{A}3_{[1]}$ and $\mathcal{A}3_{[2]}$ will be replaced by L in this case, so the two arguments to F will be used lazily.

If there is an $\mathcal{A}i_{[p]}$ in M where $m_{i\,[p]}$ is $\mathcal{A}x_{[y]}$ for some x and y then this is not a suitable replacement for $\mathcal{A}i_{[p]}$ since it will be meaningless in the context of the instantiated mode. The $\mathcal{A}i_{[p]}$ in M is replaced by L instead. This situation may arise when the argument to a function is itself a higher order function, whose mode contains $\mathcal{A}x_{[y]}$. This strictness analyser is only "second order" because these $\mathcal{A}x_{[y]}$s are replaced by L. Extending this method of strictness analysis to higher orders does not appear to be feasible without a huge increase in complexity.

The definition of *back* is

$$
\begin{aligned}
&back\ \mathsf{m_L} &&= \mathsf{m_L}\\
&back\ \mathsf{m_D} &&= \mathsf{m_D}\\
&back\ \mathcal{B}i_{[p...]} &&= \mathcal{B}i_{[p...]}\\
&back\ \mathcal{A}i_{[p...]} &&= \begin{cases} \mathcal{A}(i-j)_{[p...]} & \text{If } i > k\\ M' & \text{Otherwise, where}\\ & M' = m_{i\,[p...]} \text{ with every } \mathcal{A}x \text{ replaced by } \mathsf{m_L} \end{cases}\\
&back\ U \rightarrow M &&= (back'\ U) \rightarrow (back\ M)\\
&back\ (M_1\ Op\ M_2) &&= (back\ M_1)\ Op\ (back\ M_2)
\end{aligned}
$$

$$
\begin{aligned}
&back'\ U &&= U \quad \text{when } U = \mathsf{L}, \mathsf{S}, \mathsf{A}, \mathsf{D} \text{ or } \mathcal{B}i_{[p]}\\
&back'\ \mathcal{A}i_{[p]} &&= \begin{cases} \mathcal{A}(i-j)_{[p]} & \text{If } i > k\\ \mathsf{L} & \text{If } m_{i\,[p]} = \mathcal{A}x_{[y]} \text{ for some } x, y\\ m_{i\,[p]} & \text{Otherwise} \end{cases}\\
&back'\ (U_1\ Op\ U_2) &&= (back'\ U_1)\ Op\ (back'\ U_2)
\end{aligned}
$$

The purpose of *back* is to instantiate the remainder of the mode when all the usages corresponding to the j arguments have been taken care of. Since none of the usages in the remaining mode corresponds to an argument, there is no need to replace $\mathcal{A}i_{[p]}$ by L if $i > j$. It can instead be replaced by $\mathcal{A}(i-j)_{[p]}$. This adjustment must be made, rather than just leaving $\mathcal{A}i_{[p]}$ in the mode of the result because otherwise it would refer to the wrong argument.

Termination

I have no proof that this second order algorithm terminates. In order to terminate it must be possible to compare modes for equality, even when they contain $\mathcal{A}i$ or $\mathcal{B}i$. The Ponder compiler uses the transformations described above for Best, Worst and Choice to simplify modes to a standard form, so they can be compared easily.

3.3 Examples

F was defined earlier as

$$F = \lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b$$

As the first example I will show how the mode of F is derived. Let us start by considering the expression $f\ a\ b$. In this new analyser we cannot simply analyse function and argument separately as we did before, because the mode of a function can be affected by the mode of an argument it is applied to. So to find the result of

$$Analyse\ \mathrm{M}\ (f\ a\ b)\ [\,]$$

we must first analyse b, the argument of $f\ a$. Note that we have given an empty list of modes to *Analyse* M, because $f\ a\ b$ is not applied to any arguments in the body of F.

$$\mathit{Analyse}\ \mathrm{M}\ \ b\ [\,]\ =\ (\mathcal{B}2_{[1\ldots]},\ \{a{:}\mathsf{A},\ b{:}\mathsf{S},\ f{:}\mathsf{A}\})$$

The mode returned is $\mathcal{B}2_{[1\ldots]}$ because b is the second bound variable of F. Note that again *Analyse* M is given an empty list of modes. Having obtained the mode of b we can proceed to analyse $f\ a$:

$$\mathit{Analyse}\ \mathrm{M}\ \ (f\ a)\ \ [\mathcal{B}2_{[1\ldots]}]$$

Again we cannot find the mode of f without knowing the mode of a, so

$$\mathit{Analyse}\ \mathrm{M}\ \ a\ [\,]\ =\ (\mathcal{B}1_{[1\ldots]},\ \{a{:}\mathsf{S},\ b{:}\mathsf{A},\ f{:}\mathsf{A}\})$$

Now finally we can look at f:

$$\mathit{Analyse}\ \mathrm{M}\ \ f\ \ [\mathcal{B}1_{[1\ldots]};\ \mathcal{B}2_{[1\ldots]}]\ =\ (\mathcal{B}3_{[1\ldots]},\ \{a{:}\mathsf{A},\ b{:}\mathsf{A},\ f{:}\mathsf{S}\})$$

In fact when we look at f it does not need to know the modes of its arguments at all, because it is a bound variable, but it might have done if it had been a combinator. So

$$\begin{aligned}&\mathit{Analyse}\ \mathrm{M}\ \ f\ a\ \ [\mathcal{B}2_{[1\ldots]}]\\ &\quad=\ (\mathcal{B}3_{[1\ldots]\,[2\ldots]},\ \{a{:}\mathsf{A},\ b{:}\mathsf{A},\ f{:}\mathsf{S}\}\ \mathsf{Best}\ (\mathcal{B}3_{[1\ldots]\,[1]}\ \mathsf{Worst}\ \{a{:}\mathsf{S},\ b{:}\mathsf{A},\ f{:}\mathsf{A}\}))\\ &\quad=\ (\mathcal{B}3_{[2\ldots]},\ \{a{:}\mathcal{B}3_{[1]},\ b{:}\mathsf{A},\ f{:}\mathsf{S}\})\end{aligned}$$

And

$$\begin{aligned}&\mathit{Analyse}\ \mathrm{M}\ \ f\ a\ b\ [\,]\\ &\quad=\ (\mathcal{B}3_{[2\ldots]\,[2\ldots]},\ \{a{:}\mathcal{B}3_{[1]},\ b{:}\mathsf{A},\ f{:}\mathsf{S}\}\ \mathsf{Best}\ (\mathcal{B}3_{[2\ldots]\,[1]}\ \mathsf{Worst}\ \{a{:}\mathsf{A},\ b{:}\mathsf{S},\ f{:}\mathsf{A}\}))\\ &\quad=\ (\mathcal{B}3_{[3\ldots]},\ \{a{:}\mathcal{B}3_{[1]},\ b{:}\mathcal{B}3_{[2]},\ f{:}\mathsf{S}\})\end{aligned}$$

Now we start to build the mode of the whole function. Since f is the third argument, we must replace $\mathcal{B}3$ by $\mathcal{A}1$. Any other $\mathcal{A}i$ would be replaced by $\mathcal{A}(i+1)$.

$$\mathit{Analyse}\ \mathrm{M}\ \ \lambda f.\ f\ a\ b\ [\,]\ =\ (\mathsf{S}\to\mathcal{A}1_{[3\ldots]},\ \{a{:}\mathcal{A}1_{[1]},\ b{:}\mathcal{A}1_{[2]}\})$$

For the next lambda abstraction we replace $\mathcal{A}i$ by $\mathcal{A}(i+1)$:

$$\mathit{Analyse}\ \mathrm{M}\ \ \lambda b.\ \lambda f.\ f\ a\ b\ [\,]\ =\ (\mathcal{A}2_{[2]}\to\mathsf{S}\to\mathcal{A}2_{[3\ldots]},\ \{a{:}\mathcal{A}2_{[1]}\})$$

And similarly for the outermost lambda abstraction:

$$\mathit{Analyse}\ \mathrm{M}\ \ \lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b\ [\,]\ =\ (\mathcal{A}3_{[1]}\to\mathcal{A}3_{[2]}\to\mathsf{S}\to\mathcal{A}3_{[3\ldots]},\ \{\,\})$$

Thus the mode of F is

$$\mathcal{A}3_{[1]} \to \mathcal{A}3_{[2]} \to \mathsf{S} \to \mathcal{A}3_{[3...]}$$

and this will not change on subsequent iterations because the definition is not recursive and it does not depend on any other combinators.

As another example, for which I will not give any working, consider this definition of Y:

$$\begin{aligned} \Theta &= \lambda\theta.\, \lambda f.\, f\ (\theta\ \theta\ f) \\ Y &= \Theta\ \Theta \end{aligned}$$

Θ is given the mode

$$\mathcal{A}2_{[1]} \to \mathsf{S} \to \mathcal{A}2_{[2...]}$$

And Y is given the mode

$$\mathsf{S} \to \mathcal{A}1_{[2...]}$$

This is quite a reasonable mode, though of course it does not capture the full properties of Y. It is interesting to note that the other higher order strictness analysers, all based on Mycroft's analyser, cannot cope with the definition of Y at all.

3.4 Other higher order analysers

How does my analyser compare with other higher order strictness analysers based on Mycroft's work? Mycroft's first order analyser passed 0s and 1s to the functions in the abstract domain, and observed the results to determine the strictness properties of the real functions. In a higher order analyser, functions such as $\lambda x.\ 1$, $\lambda x.\ 0$ and so on are passed to a higher order abstract function when the real function expects a function taking one argument. In [Abramsky et al 85] the following example is given: consider the function

$$apply = \lambda f^{A \to A}.\, \lambda x^{A}.\, f\ x$$

The superscripts on the variable names are the types of the variables. The real domain is called A and the abstract domain $\{0, 1\}$ is called 2.

$$apply^{\#} = \lambda f^{2 \to 2}.\, \lambda x^{2}.\, f\ x$$

To see if *apply* is strict on its first argument we need to pass $apply^{\#}$ the bottom element of the domain $2 \to 2$. This is $\lambda x.\ 0$.

$$apply^{\#}\ (\lambda x.\ 0)\ 1 = 0$$

So *apply* does need its first argument. Does it need its second? At first it seems that we must test $apply^{\#}\ g\ 0$ for all values of g. There are actually only three, namely $\lambda x.\ 0$, $\lambda x.\ x$ and $\lambda x.\ 1$, but in higher function domains there would be many more. However, it turns out that it is only necessary to test with the most pessimistic value for g, the top element of $2 \rightarrow 2$, since if the answer is 0 for this it will be 0 for all the others. The top element of $2 \rightarrow 2$ is $\lambda x.\ 1$.

$$apply^{\#}\ (\lambda x.\ 1)\ 0 = 1$$

So it is not strict on its second argument.

They go on to describe a technique similar to instantiating the modes of higher order functions in my analyser. This takes into account the actual arguments in a particular application of the function, so as to obtain a more strict answer. So, for instance they would find that *apply* $(\lambda x.\ x)\ q$ was strict on q, even though the *apply* function is not in general strict on its second argument.

The Abramsky-Burn-Hankin [Abramsky et al 85] and Hudak-Young [Hudak & Young 85] analysers are very similar. They are both true higher order analysers, in that they are not restricted to second order functions as mine is. There are still some restrictions on the kind of functions that they can cope with. A higher order strictness analyser built upon Mycroft's work cannot cope with certain forms of self application. This is because function application in the real domain is represented by function application in the abstract domain. To analyse the strictness properties of a function which uses other functions it is necessary to reduce the abstract versions of the other functions to find out what the result will be. If one of these functions is applied to itself, this reduction may never terminate. The Abramsky-Burn-Hankin analyser gets around this problem by using a monomorphic type system to forbid such functions.

This restriction may seem academic, but Ponder [Fairbairn 85] has a sufficiently powerful polymorphic type system that it is possible to define operators that must be built in to other languages. For instance it is possible to define Y, the fixed point operator, exactly as I defined it earlier in the lambda calculus. A program written in a language with an ML-style polymorphic type system can be automatically translated to a language with a monomorphic type system, so the restriction on the Abramsky-Burn-Hankin technique is not as severe as it seems at first. However, Ponder types are more powerful than ML types, and cannot be converted to monomorphic types. So if higher order strictness analysis is to be performed at all on a Ponder program it cannot be done safely by a method based on Mycroft's analyser.

How does my extended analyser relate to the other higher order analysers? Just as modes in my original analyser can be seen as roughly equivalent to the abstract functions in Mycroft's original analyser, so the new modes can be seen as roughly equivalent to *sets* of abstract functions in the Abramsky-Burn-Hankin and Hudak-Young analysers. For example define the integers as in the first chapter:

$$\begin{aligned}
\overline{0} &= \lambda x.\, \lambda y.\, y \\
\overline{1} &= \lambda x.\, \lambda y.\, x\ y \\
\overline{2} &= \lambda x.\, \lambda y.\, x\ (x\ y) \\
&\cdots \\
\overline{n} &= \lambda x.\, \lambda y.\, \overbrace{x\ (x\ \ldots\ (x}^{n}\ y)\ \ldots)
\end{aligned}$$

In my second order analyser $\overline{0}$ has mode

$$\mathsf{A} \to \mathsf{S} \to \mathcal{A}2_{[1\ldots]}$$

and $\overline{1} \ldots \overline{n}$ all have mode

$$\mathsf{S} \to \mathcal{A}1_{[1]} \to \mathcal{A}1_{[2\ldots]}$$

In the other analysers based on Mycroft's work, all of $\overline{0}, \overline{1} \ldots \overline{n}$ have distinct abstract functions, so they have distinct strictness properties.

Details of the performance of my second order analyser are given in chapter four. Neither the Abramsky-Burn-Hankin nor the Hudak-Young analyser has been implemented, so I cannot compare their performance with that of my analyser.

3.5 Lazy pairs

In this section I present a very brief description of how my original analyser can be extended to lazy pairs, and discuss some of the limitations of strictness analysis.

The basic idea is to extend modes and usages in a different way, so that the usages in the mode of a function that returns a pair may depend upon the usage of an application of this function. For instance if a function puts its first argument directly in the left of a pair it returns, it will be strict on that argument only if the resulting pair is used in a context that is strict on the left of that pair. To express this kind of dependency the composite usage $\mathsf{pair}(U_1, U_2)$ is introduced, and the composite mode $\mathsf{pair}(M_1, M_2)$. To take these apart, define $\mathsf{left\ pair}(x, y) = x$ and $\mathsf{right\ pair}(x, y) = y$. The usage of the whole application returning a pair is denoted by C. Using this notation the *pair* function would have mode

$$\mathsf{left}\ C \to \mathsf{right}\ C \to \mathsf{pair}(\mathcal{A}1_{[1\ldots]}, \mathcal{A}2_{[1\ldots]})$$

If this were applied to two integers (with mode m_L) in context pair(A, S) the substitutions would give

$$\text{left pair}(A, S) \rightarrow \text{right pair}(A, S) \rightarrow \text{pair}(m_{L[1...]}, m_{L[1...]})$$

which would reduce to

$$A \rightarrow S \rightarrow \text{pair}(m_L, m_L)$$

In other words this particular pair is strict on its second argument and can discard its first argument. John Hughes describes a strictness analyser for non-flat domains [Hughes 85b] which has many similarities to this scheme. Another list strictness analyser, described in [Kieburtz & Napierala 85] extends Mycroft's analyser so that there are pairs in the abstract domain as well as the real domain.

Unfortunately, there are problems with such analysers. Firstly, there are problems when integrating pairs into my second order analyser, which needs to know the modes of the arguments to a function before it can determine what their usages will be. To cope with pairs it is often necessary to know what an argument's usage is *before* its mode can be determined.

Another problem for all list strictness analysers is that the modes may be infinite. This is because recursive functions on structured data can have infinite modes which do not have infinitely many arrows at the top level, but have complex recursive usages instead.

Using a more expressive notation it would be possible to represent some of these infinite modes in finite space, because of their recursive structure. Modes that are representable in finite space using a notation of the same expressiveness as a context free grammar have been called 'rational' by Hughes. Such modes can still be compared for equality.

Unfortunately, there is no guarantee that the least fixed point will be rational. For instance, a function that takes a list of lists and returns the sum of the diagonal of this 'array' does not have a rational mode. This is because it is impossible to write a context free grammar which produces all the arrays with 'strict' on the diagonal and 'lazy' everywhere else. Since functions like this are quite common in functional programs an analyser of lazy lists which used these principles would probably not be much use. The best that could be achieved would be to find a rational solution slightly lazier than the true least fixed point. In practice, the solution actually found would probably be much lazier than this — perhaps too lazy to be useful.

If a still more expressive notation was used, of the same expressiveness as the lambda calculus, then more modes could be represented in finite space. However,

with a notation as expressive as this it would no longer be possible to compare the modes automatically, so the analyser could not be guaranteed to terminate without outside help.

A far *less* expressive notation has been suggested in [Wadler 85]. This can only distinguish completely lazy lists, lists where all the tails are strict, lists where the head is strict and lists where everything is strict. By throwing away a lot of information it appears that a good analysis can be done for these remaining cases.

As well as the problems of performing strictness analysis on lazy lists, there is also the problem of how to use this strictness information to increase the speed of a program significantly. The schemes of code generation used to take advantage of first order strictness information do not make very good use of list strictness information. I discuss this further in the next chapter.

Chapter 4

Implementation

4.1 Introduction

In this I chapter describe the performance improvements arising from the use of my strictness analyser in the Ponder compiler, which produces code for conventional machines. For comparison, the results of a completely lazy implementation are given, and also the results of an intermediate implementation which stops short of full strictness analysis but incorporates several straightforward improvements. These improvements are necessary anyway, in order to take advantage of the strictness information, and it is instructive to see how much difference they make on their own.

The second order analyser described in Chapter 3 makes no difference to the performance of real programs, though it finds the anticipated strictness properties for small test functions. This is rather surprising, and indicates that higher order strictness analysis may be less useful than previously imagined. These results will be given in detail in section 4.3. The last section of the chapter goes on to discuss the use of strictness analysis in parallel implementations, and the problems still to be resolved in this area. First I will describe how the Ponder compiler generates code for conventional machines.

4.2 Code generation

The overall structure of the Ponder compiler is shown in Figure 4.1. The 'front-end' translates the type-checked program into annotated lambda expressions. A variety of 'back-ends' have been written, to generate code for a 68000, a VAX, an IBM 3081 and SKIM2 [Elworthy 85].

A back-end reads the annotated lambda expressions produced by the front-end

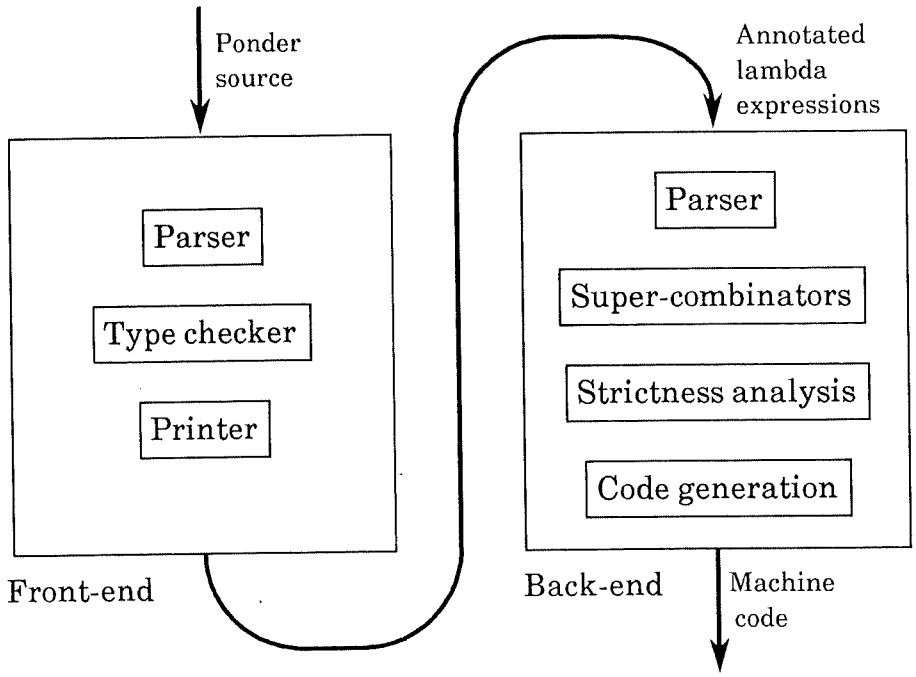


Figure 4.1: The structure of the Ponder compiler

and converts them into super-combinators using an improved version of Hughes' algorithm [Fairbairn 85]. The super-combinators produced by the original algorithm are rather inefficient and the improvements give a considerable speed improvement, doubling the speed for some programs. As Figure 4.1 shows, the strictness analysis is performed in the back end, and it is the super-combinator program that is analysed.

The Ponder compiler is written in Algol68c because if it were written in Ponder it would be too slow to be practical. Running on an 8MHz 68000, the compiler takes about half a minute to generate code for small programs, and about ten minutes for programs of several thousand lines. The first order analyser described in chapter 2 is very quick — it does not take any noticeable time for small programs, and even on larger programs (more than 2000 lines) only takes a few seconds. The second order analyser is about ten times as slow. The back end spends most of its time generating the machine code for the super-combinators.

To exploit the information gained by strictness analysis requires good code generation techniques. Appendix A describes in detail how to generate code for a simple stack machine. This appendix will be of interest mainly to those who intend to write a strictness analyser and code-generator themselves. It is not necessary to read the appendix to understand the rest of this chapter.

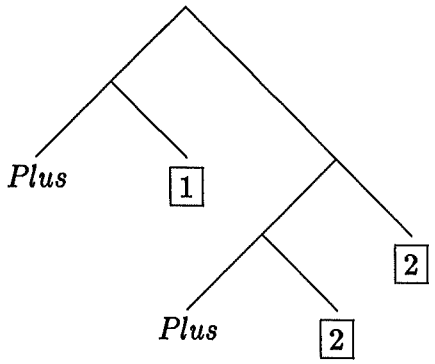
Tree templates

There are two ways of implementing super-combinators. The first way is to encode them as trees of applications, which are then used as templates. Places are marked in these trees where arguments should be substituted. When a super-combinator is applied to some arguments, a copy of this 'template tree' is made, with the arguments substituted into these places. This new tree is then reduced further by the evaluator.

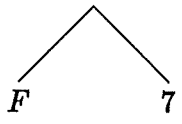
This first approach would convert the combinator

$$F \;=\; \lambda a.\ \lambda b.\ (\mathit{Plus}\ a\ (\mathit{Plus}\ b\ b))$$

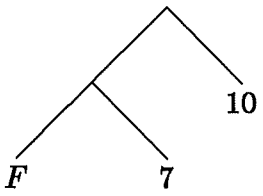
into the template



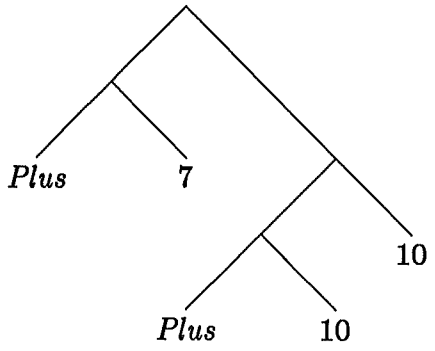
The value returned by the combinator is a copy of this tree with $\boxed{1}$ and $\boxed{2}$ replaced by the first and second arguments given to the combinator. If F was applied to one argument, say the integer 7, this would not be reduced, but would remain as



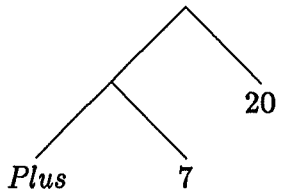
If F was applied to two arguments, say 7 and 10, this would be represented by



which would then be reduced to



This tree is a copy of the template, with 7 substituted for $\boxed{1}$ and 10 for $\boxed{2}$. This new tree can now be reduced further, first to



and then to just the integer 27.

Pure code

A better way to implement super-combinators is to generate code for a stack machine which produces the result of the super-combinator directly, rather than have the evaluator interpret a tree representation. This is much quicker, because it does not have the overhead of building trees and traversing them in the evaluator. Using this second approach the code for F would look something like

```
Push arg 2
Push arg 2
Call Plus
Push arg 1
Call Plus
Return
```

Techniques for generating code of this form for conventional machines are well understood, so good code can be produced. Some combinators will still return application trees constructed so as to incorporate unevaluated arguments, and these trees will need further evaluation, but there is a worthwhile performance improvement over the tree template approach.

result. Since this simulation did not take into account the problems of contention or communications costs, it shows that real programs would not run much faster on parallel machines than on uniprocessors if only first order strictness analysis were used.

However, if list strictness detection is used, the situation will be very different. Now there will be much more parallelism, and some programs will have an excess of parallelism. These programs will have more parallelism than any realistic machine could effectively provide, and the performance will drop because of this.

The reason for this is that on most parallel computers it takes longer to communicate between some processors than others, because every processor is not linked to every other processor. Only some links are present. Thus if the machine is to run as fast as possible, most communications should be to near neighbours. However, in functional languages, sharing is a necessity. If evaluated expressions are not shared then there may be an explosion in the size of an executing program. To share data in a parallel computer means communicating between the place it is stored and the places it is needed every time it is used. The more parallel processes that use a shared value, the further on average that value must be transmitted and the longer it will take to reach its destination.

Another reason why a parallel machine would run slower if a program were excessively parallel than if it were more restrained is that much of the parallelism would be trivial. For example, in divide-and-conquer algorithms the computation would spread out as a tree, with the work being done at the leaves and results being combined at the branches. It will usually be more costly to transmit program and data to several new processors than it would be to perform several leaf computations together at a higher branch. Only in exceptional cases will the execution time of a leaf process be longer than the time needed to set it up.

There is thus a need to restrict the parallelism of a program so that it does not "run away with the machine". A program with fine-grain parallelism must be changed into one that has a coarser grain of parallelism, suitable for the machine it is to run on. Some work has been done on restricting the parallelism in functional languages to those places where it is useful [Hudak & Goldberg 85] but much more work is needed. In the long run it would be desirable to be able to analyse the properties of machines so as to restrict parallelism appropriately, or even to choose which of a number of machines was the best to run a particular program. Reducing parallelism by reducing strictness should be no more difficult than increasing parallelism by increasing strictness, so I am confident that these techniques can

be perfected. When they are, functional languages will be an effective means to program parallel computers.

It should be noted that when using either approach there is no need to expand the tree template or to execute the code of a super-combinator until all its arguments have been given. For the pure code approach this means that conventional code-generation techniques can be used, since partial applications of functions are not reduced. Indeed, code-generation of the bodies of super-combinators is almost exactly the same as code-generation of routines in a language that has no dynamic free variables, for example BCPL [Richards & Whitby-Strevens 80].

4.3 Results

Four benchmarks were used to assess the increase in performance due to strictness analysis. The Ponder source code for all four programs is given in appendix C. Nfib is a standard performance test for functional languages, based on the Fibonacci series. A call of Nfib 20 executes more than 20000 function calls. Insertion sort and quicksort are the corresponding sorting algorithms used on a list of 100 random integers. Dragon 10 produces a picture of a fractal 'Dragon curve' [Mandelbrot 83] of depth 10 on a graphics terminal. This has 1024 line segments and involves a considerable amount of list processing, 'bit-shuffling' and arithmetic. Performance improvements for this program should be similar to those for 'realistic' programs. The other three are in many ways atypical.

Results are given for three different implementations: 'lazy', 'sensible' and 'strict'.

Lazy code

Lazy code is generated in the second form as described above, but the code always constructs trees, even when it could evaluate expressions immediately. It does not use strictness detection. The code is thus very similar in performance to the tree template approach and to other re-write style graph reducers [Turner 79]. The times taken by four example programs generated as lazy code are given below.

Program	Lazy
Nfib 20	29.66s
Insertion sort	6.96s
Quicksort	4.86s
Dragon 10	116.0s

Sensible code

The next results are for the 'sensible' code generator, which does everything except strictness analysis. Wherever possible, code is generated for function bodies to calculate their results directly, rather than building a tree which must then be interpreted. The arguments to built-in functions are evaluated eagerly and conditionals are implemented using jumping code rather than by building trees. The tree building approach to conditionals is to make two trees, one for the Then part and one for the Else part, and depending on what the condition evaluates to (true or false) to throw one of them away. This is a waste of time — not only is it futile building a tree and throwing it away, it is not even necessary to make a tree for the part which is needed. Code can be generated to produce the values returned by the Then and Else parts, just as it is for a function body, by direct execution of compiled code rather than interpretation of trees. The appropriate code fragment can be jumped to after testing the condition of the If. This produces a considerable speed improvement.

The table below summarises the results for sensible code and compares them with the results for lazy code. The column on the far right gives the percentage improvement when going from lazy code to sensible code.

Program	Lazy	Sensible	Gain
Nfib 20	29.66s	8.02s	270%
Insertion sort	6.96s	2.62s	166%
Quicksort	4.86s	3.20s	52%
Dragon 10	116.0s	90.0s	29%

It is interesting to note how the performance of the two sorts has changed. In lazy code, Quicksort was faster, but in sensible code insertion sort is faster. I did not expect this reversal. It would be nice if it were easy to tell whether one function would be faster than another by looking at them, but this is often impossible when many transformations are performed on a program.

The performance characteristics of lazy code are similar to those of an SKI combinator reducer, so on a hardware combinator reduction machine like SKIM, Quicksort would be faster. On a conventional machine using compiled code rather than interpreted trees, insertion sort would be faster. These figures are of course only valid for these particular versions of the algorithms as compiled by the Ponder system, and there is no reason to believe that this is a fundamental result about the efficiencies of sorting algorithms in functional languages.

Strict code

When generating 'strict' code the compiler ensures that all strict arguments to super-combinators are evaluated directly in machine code rather than by using trees. Absent arguments are not generated at all. Because of optimisations performed on super-combinators in the Ponder compiler there are very few of these anyway. Dangerous arguments are trapped, so that a program does not actually crash if it evaluates one; instead the run-time system returns an error message and stops. The tables below summarise the results for strict code.

Program	Lazy	Sensible	Strict
Nfib 20	29.66s	8.02s	2.99s
Insertion sort	6.96s	2.62s	2.54s
Quicksort	4.86s	3.20s	2.97s
Dragon 10	116.0s	90.0s	74.1s

Program	Gain Lazy to Sensible	Gain Sensible to Strict	Total Gain Lazy to Strict
Nfib 20	270%	168%	892%
Insertion sort	166%	3%	174%
Quicksort	52%	8%	64%
Dragon 10	29%	21%	56%

Insertion sort and Quicksort involve much list manipulation, so their improvement over the sensible code is almost negligible. Nfib shows a considerable improvement over sensible code — it runs at more than double the speed. This is the best improvement that can be hoped for, since the intermediate code for the strict version of Nfib is 'perfect' — as good as handwritten code for the recursive function. Dragon is more representative of an ordinary program, since it performs a lot of character manipulation and arithmetic, but also uses lists extensively. The improvement from sensible to strict is worth having, but not as earth-shattering as might have been hoped for. It is quite good 'value for money', since in the Ponder compiler the code to perform first order strictness analysis is about 300 lines, out of a total of 5000 for the whole back-end and takes an insignificant fraction of the compilation time.

I have also written Nfib and Dragon in the imperative language BCPL, transcribing the programs as closely as possible from Ponder. The timings are shown in the table below, with the percentage improvement given by using BCPL rather than strict Ponder.

Program	Strict	BCPL	Gain
Nfib 20	2.99s	1.52s	97%
Dragon 10	74.1s	15.66s	373%

Nfib is about twice as fast in BCPL. I mentioned above that the intermediate code for the Ponder Nfib was 'perfect'. However, the translation from this into 68000 machine code is far from perfect. Techniques such as stack slaving and peephole optimisation are not used in the Ponder compiler because these are quite well understood, and it would not be new research to implement them. The factor of two difference in speed indicates that all Ponder programs could be made to run perhaps twice as fast. If we discount this factor of two and imagine that these techniques had been used when compiling Dragon in Ponder, the BCPL version would still be more than twice as fast. Presumably this difference is due to the use of lazy lists in Ponder.

Higher order strictness

Now we come to the performance of the code produced by the second order strictness analyser. I will not give any tables comparing it with the other optimisations since this would be pointless — the performance is exactly the same as that of the ordinary strict code, as detailed above.

I found this rather surprising, so I investigated more closely. I took several Ponder programs, about 3000 lines in total, and compared the abstract machine code produced by the first order analyser with that produced by the second order analyser. In about 10,000 lines of abstract machine code there were only ten differences, and only five of these would have had any marked effect on performance. One example program, a parser, used higher order functions extensively. There were no differences between the code produced by the two analysers in over 3000 lines of stack machine code for this program. It was hardly surprising that the performance of the two sets of code was identical.

My analyser is only "second order", so it might be the case that a full higher order analyser could find significantly more strictness. This does not seem likely, since the reason for the poor performance appears to be that higher order functions have a very bad effect on strictness analysis. Consider the application

map function list

where *map* is the standard map function, *function* is strict on its argument, and *list* is some list. *map* will apply *function* to each element of *list* and it will return

a list of the results. A first order analyser will find that *map* is lazy on *function* and strict on *list* (although it will be lazy on the elements of *list*). With second or higher order analysis there will be no change.

One way to produce an improvement would be to alter the implementation so that there are two versions of *map*, one expecting a strict function, the other expecting a lazy function. As it stands, *map* must be pessimistic and assume that its functional argument is a lazy function, because it sometimes will be. If there were two versions of *map*, the appropriate one could be selected after determining the strictness properties of *function*. Care would have to be taken that only a small number of alternative versions of higher order functions were made, otherwise there would be an explosion in the size of the program.

For this scheme to yield significant results it would be necessary to have list strictness as well. This is because in the body of *map* there is the expression

$$cons\ (function\ (head\ list))\ (map\ function\ (tail\ list))$$

and (*function* (*head list*)) would still have to be constructed as an application tree if the standard lazy *cons* was used. With list strictness analysis it would be possible to determine that (say) the whole output list of *map* would be required, so a strict version of *cons* could be used. Unfortunately, *cons* is not an argument of *map*, so to exploit the list strictness information there would have to be even more versions of *map*, each with a different kind of *cons*.

List strictness

I have not implemented a list strictness analyser, and I do not know of any performance figures for this technique. There are two potential disadvantages to list strictness — the first, mentioned above, is that to exploit list strictness information a large number of alternative versions of list handling functions will be necessary. Whether this will significantly increase the size of real programs remains to be seen.

The second disadvantage is that having the whole of a list evaluated immediately rather than as it is needed might be disastrous if the elements of the list are very large when evaluated. A program that produces these elements one at a time, uses them and discards them could handle arbitrarily long lists. A program that produced all the elements at once, before it needed them, would crash if it tried to use long lists. This behaviour cannot be described as an improvement in performance.

4.4 Strictness and Parallelism

Strictness analysis allows lazy functional programs to exploit parallel machines. Without strictness analysis only the arguments to strict built-in functions like integer addition can safely be evaluated in parallel. With strictness analysis, any argument found to be strict can be evaluated in parallel with other strict arguments. The more effective the strictness analyser, the more effectively the analysed program can exploit a parallel computer.

However, parallel machines are not magic, and they will only be cost effective if the intrinsic parallelism in a program is well matched to the capabilities of the machine it runs on. For programs with very little intrinsic parallelism, an ordinary single processor computer will be best. To increase the power of such computers the different parts may be used simultaneously by pipelining the instruction or data streams, but the nature of the program itself will limit the increase in performance to be gained by these means. If the program has a very regular kind of parallelism, such as is often found in numerical analysis, then a machine such as a vector computer or systolic array would be best. In these types of machine the separate processors all perform the same operation at the same time, but on different data. For a program with irregular parallelism the best solution will be a machine with independent processing elements, each acting alone, but cooperating with the others. Parallel execution of functional programs will often be best done on such a machine, since functional languages can be used to write programs with very irregular as well as very regular parallelism.

Strictness analysis has the effect of increasing the effective parallelism of a program, so that it can make use of more processors. At the moment there is no way, other than trial and error, of finding out exactly how many processors a program could make use of. Simulations by Clack and Peyton-Jones [Clack & Peyton-Jones 85a] have shown that if only first order strictness analysis is done this number may be very small for most programs.

Their simulations did not attempt to model the behaviour of any particular hardware, but were simply investigations of the amount of parallelism *possible* in a lazy functional program after first order strictness analysis. They assumed an arbitrary number of processors with communication between them at no cost. Contention for shared values was ignored. The results thus place an upper limit on the increase in speed due to parallel evaluation. The programs ran between 300 times faster for Nfib 16 and twice as fast for a small expert system. Nfib is an exceptionally strict function — the doubling of speed is a much more typical

Chapter 5

Practical Experience

5.1 Introduction

Most functional programmers contend that functional languages are superb programming tools, although few have written substantial programs. Most functional programs are only a few hundred lines long, partly because the languages are quite terse, but also because the implementations are relatively puny. When a large program takes all day to run there is little incentive to write it. Consequently there has been little development of the techniques of functional programming beyond the 'neat tricks' that can be envisaged without even writing a program.

Large programs by current functional language standards are not large by conventional standards. A large conventional program might be 100,000 to a million lines long. The largest functional program I know of is the LML compiler, written in LML [Augustsson 84]. It is about 5000 lines long.

In this chapter I summarise the experience I have gained from writing substantial functional programs. Later in the chapter I will describe some of the problems of writing in functional languages. These problems are mainly uncertainties about the time and space behaviour of a program, caused by lazy evaluation. First I will describe *ANS*, an interactive spreadsheet, which at about 2500 lines is the largest functional program I have written. Since it illustrates some of the problems of large functional programs, especially interactive ones, I will describe *ANS* in some detail.

5.2 *ANS* — A Novel Spreadsheet

Conventional spreadsheets

A word processor can be thought of as a computerised typewriter, and a spreadsheet can similarly be thought of as a computerised version of the calculator and squared paper used by accountants. Typically, spreadsheets have cells laid out in a rectangular grid. Each cell is referred to by its coordinate in this array of cells — for instance rows are numbered and columns lettered, so "B-7" would be the second cell in the seventh row.

	A	B	C	D	E	F	
1		1003					
2		21					
3		322					
4		1000					
5		15					
6		---------					
7	Total =	2361					

Figure 5.1: A conventional spreadsheet

Cells contain values which can be text or numbers, or formulæ with references to other cells. For instance, in the spreadsheet in Figure 5.1, cells B-1 to B-5 have been filled with the numbers 1003, 21, 322, 1000 and 15. Cell B-6 has been filled with the text "---------" and A-7 with the text " Total =". Although B-7 appears to contain a number like any of the other cells, it actually contains a formula whose *value* is displayed as the contents of the cell. Commands would be provided to inspect and alter this hidden formula. If we inspected B-7 we would find that the formula was

$$\text{B-1} + \text{B-2} + \text{B-3} + \text{B-4} + \text{B-5}$$

The value displayed in B-7 is the total of the numbers in that column. If any of the numbers is changed, the total changes automatically to match.

ANS

ANS is novel in several ways. The cells in *ANS* do not lie on a rectangular grid, but can be positioned anywhere on the screen, even one on top of the other like pieces of paper. A typical screen is shown in Figure 5.2.

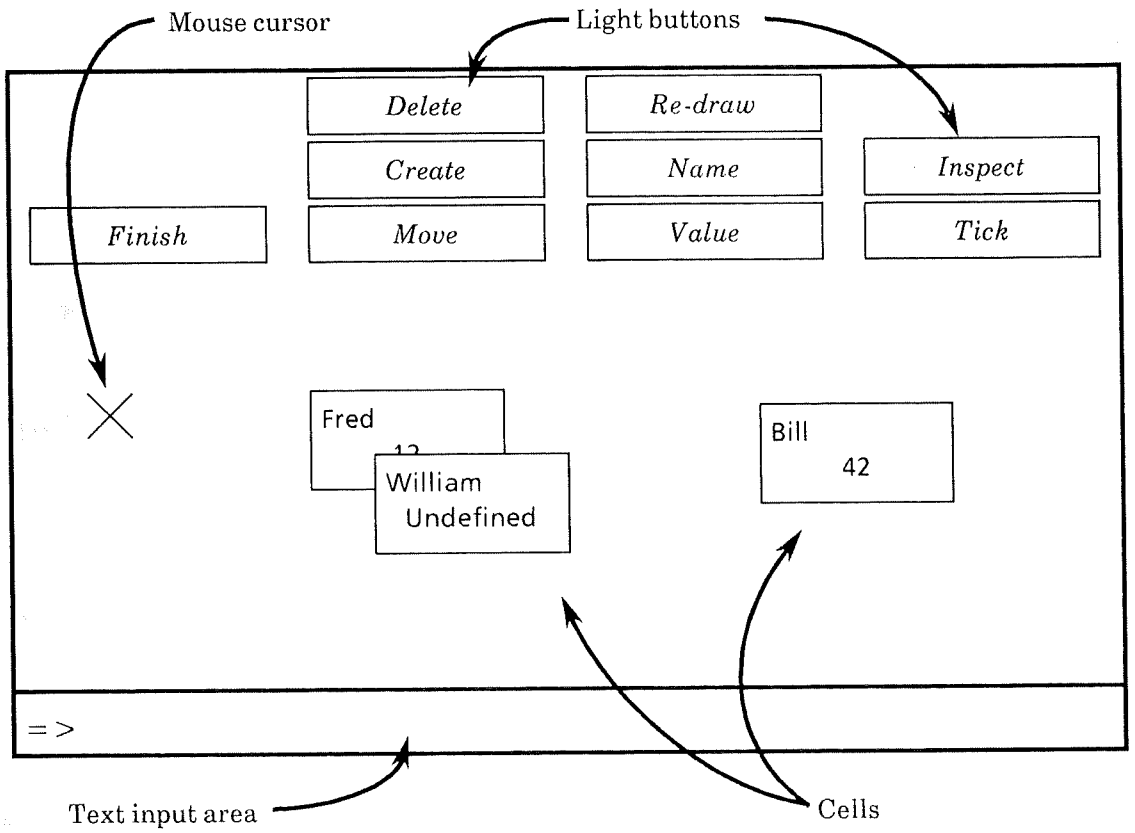


Figure 5.2: A typical *ANS* screen

At the top of the screen are 'light buttons' which are used to give commands to *ANS*. The idea of the light buttons is that they are simulated push-buttons. To push one, the mouse cursor should be placed over the light button, then the button on the mouse pressed and released. When pushed, the light buttons cause the following actions to be performed:

- *Finish* exits from *ANS*. The configuration of cells on the screen is saved in a back-up file and restored the next time *ANS* is used.
- *Delete* removes a cell. After pushing *Delete* the next cell selected with the mouse will be removed. The *Delete* light button lights up when it has been pushed, and remains lit up until a cell has been selected and removed from

the screen. All light buttons that require other actions to be performed remain lit up until everything is complete.

- *Create* places a new cell, with no name, at the location on the screen indicated by the next mouse selection.
- *Move* moves a cell from one position on the screen to another. The cell to be moved and its destination are indicated using the mouse.
- *Re-draw* clears the screen and draws all the light buttons and cells again.
- *Name* is used to change the name of the cell indicated by the next mouse selection. The new name is typed into the text input area at the bottom of the screen.
- *Value* is used to change the contents of the cell indicated by the next mouse selection. The new contents can be a number or a formula. Formulæ are described later in this section. The new contents are typed into the text input area.
- *Inspect* displays the contents of a cell in the text input area. If the cell contains a number, what is displayed will be the same as the value shown for that cell on the screen. If the cell contains a formula then the text of the formula is displayed.
- *Tick* changes the current values of cells into 'previous' values. This is described below, after formulæ.

As an example the sequence of mouse selections necessary to move a cell is shown in Figure 5.3. First the light button 'move' is selected, then the cell to be moved is selected, and finally the new position. At this point the cell disappears from its old position and reappears at the new one

The concrete syntax of formulæ is

$$formula = \begin{cases} number \\ name \\ \text{last } name \\ formula\ Op\ formula \\ (formula) \\ \text{Undefined} \\ formula\ \$\ formula \end{cases}$$

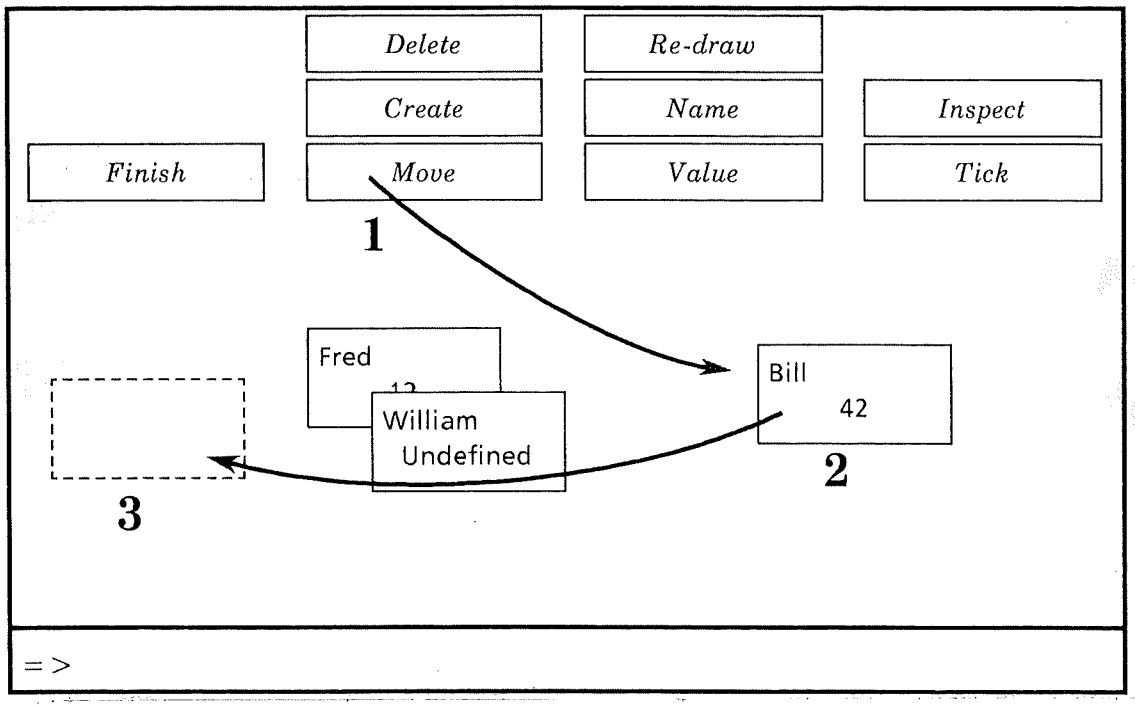


Figure 5.3: How to move a cell

A *number* is an integer of arbitrary length. A *name* gives the current value of the cell with that name. As well as the current value of a cell, it is possible to obtain the 'previous' value of a cell, using last *name*.

When the light button *Tick* is pushed, all current values are recalculated, with the old current values used as 'previous' values. For example, if the previous value of cell Bill was 41 and its formula was last Bill + 1, then its displayed value would be 42. If *Tick* was pushed this would change to 43, then 44 next time it was pushed, and so on.

In *formula Op formula* the two formulæ are combined using the operator *Op*. A variety of operators are available to combine formulæ, but I will not describe them here. All the usual arithmetic operators are provided.

When a cell is created its value is not a number, but the special value Undefined. The purpose of $, as in *formula* $ *formula*, is to give a default value for the first formula, should it be undefined. For instance, if the value of William is Undefined then the value of the formula William $ 13 is 13. However, if William is defined, then William $ 13 will have whatever value William has. The formula William $ 13 should be read as "William's value or 13 by default".

The most complex spreadsheets that have been constructed using ANS are a simulation of a flip-flop constructed from NOR-gates and a Newton-Raphson

square root finder. Despite their small size they demonstrate the power of this 'history' mechanism using last.

The program

Figure 5.4 is a simplified picture of the main components of *ANS*.

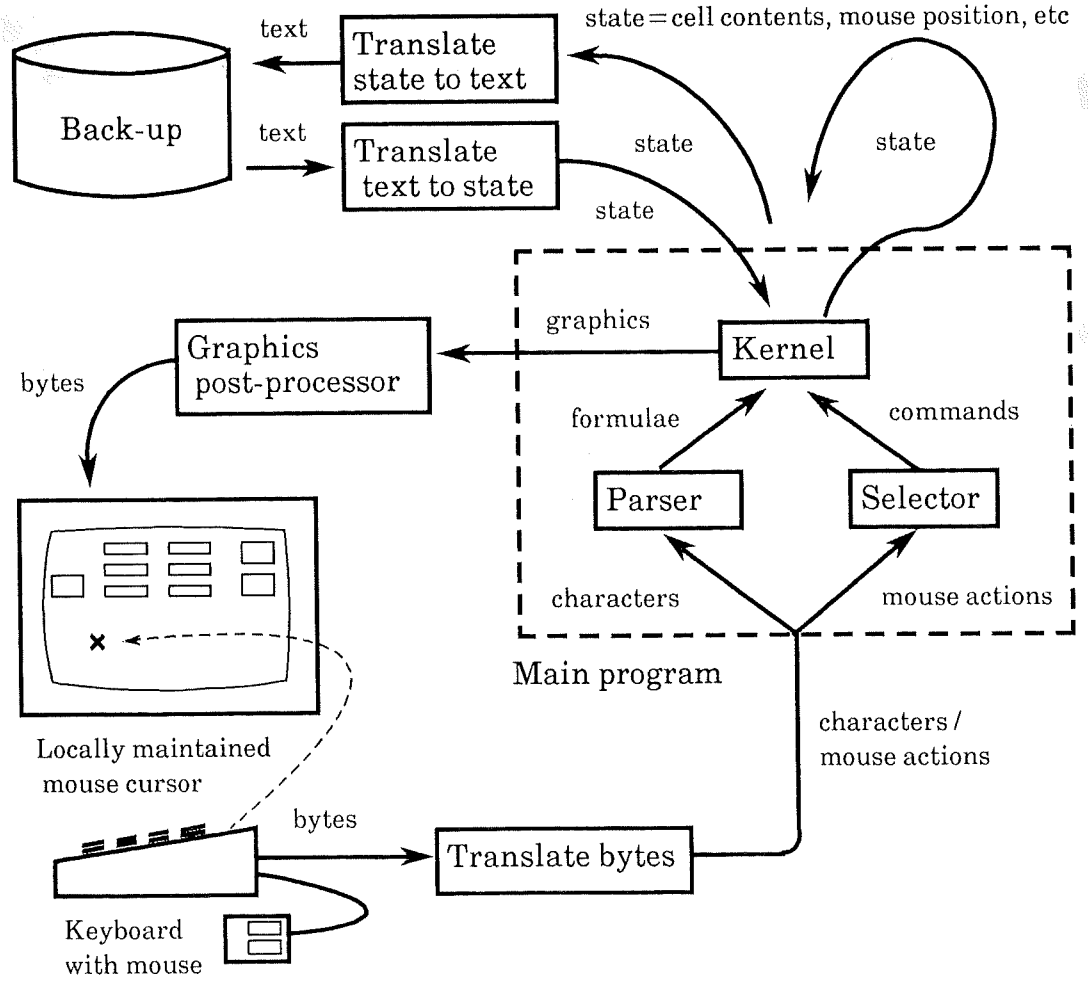


Figure 5.4: The internal structure of *ANS*

The program is a function that takes a list of bytes from the keyboard/mouse and sends a list of bytes to the screen. There is no other connection between the keyboard and screen except for a mouse cursor, which is maintained locally by the terminal. When *ANS* starts it looks for a back-up file containing a spreadsheet stored in text form. If it cannot find one it starts with no cells on the screen, just the light buttons. When the *Finish* light button is pushed, any cells on the screen are stored in textual form in the back-up file, so that they can be restored when *ANS* next runs.

The stream of bytes from the keyboard/mouse consists of intermingled mouse actions, with coordinates, and characters typed at the keyboard. The function 'translate bytes' converts the byte stream into an internal form which the rest of the program can use without further processing. Some filtering and simplification is done at this point, as the mouse has three buttons that all cause different codes to be transmitted when they are depressed and released. Button releases are ignored and all button depressions are made equivalent.

The main program decides whether to give the next element of its input list to the 'parser' or the 'selector', or to discard it if it is inappropriate. The parser converts character strings into formulæ which will become the contents of cells. The selector decides whether a light button or a cell has been picked by a mouse button push. The mouse position is maintained in the state, along with a list of cells, which the main program "updates" by recursing with altered arguments. The 'kernel' is the part of a program that determines the values of cells, and keeps the screen image up to date so that it is an accurate reflection of the current state.

When a cell's value changes it is re-written on the screen. There is no need to re-write the entire screen; all that happens is that the kernel sends some high level graphics commands to the graphics post-processor. These describe what is to be done and the graphics post-processor converts them into a list of bytes which are interpreted by the terminal so that the cell's previous value is erased and its new value written. There has in the past been some concern about how functional programs could drive bit-map displays [Morris 81], since they obviously could not use side effects to alter the contents of a screen memory. The answer is quite simple — output commands that the display will interpret to produce the desired effect. After all, few people want to use a display at the bit-by-bit level. They want to write characters at a particular place, draw lines, fill areas and so on. Many modern displays have BITBLT facilities, which would be very easy to drive from a functional language like Ponder, because arbitrary precision integers could be used as a convenient representation of bit arrays.

When driving the display for *ANS* there is the added complication that the local area network and terminal software ignore some byte values and translate others. Bytes thus have to be translated by a post-processor so that when they are re-translated by the intervening devices they will emerge as intended.

The output to the back-up file and the output to the terminal do not actually come out of the main program by different routes, though I have shown it like that in Figure 5.4 for clarity. In fact the output from the program is a lazy list of 'file

actions'.

In the same way that the output list of graphics contains high level commands to drive a display, so the list of file actions contains high level commands to drive the file system. Each file action contains a name specifying the file, and a code specifying the operation to be carried out on the file. Most file actions also have another string — for instance the 'append' file action also contains the text to be appended to a file. The back-up file is written by this means, and the list of bytes sent to the terminal is sent in an 'append' to the standard output.

5.3 Programming Techniques

Like anyone learning to program, the novice functional programmer has an initial feeling of paralysis when writing a substantial program for the first time. Irrespective of how competent a programmer is in an imperative language there will be this problem, and most of the programming techniques we have been taught simply are not suitable for functional languages. Practically all of the algorithms that computer scientists are familiar with are designed for imperative languages. Even when algorithms need not be presented in this form they usually are, so the work of the functional programmer is made doubly hard. Not only must he master the unfamiliar programming style of a new language, he must also translate the algorithms he is familiar with in order to express them in the new language at all.

When I wrote *ANS* I could not even see how to start for quite some time, despite having previously written interactive programs some hundreds of lines long. When I did start, it took about a month to write and type-check *ANS*, most of this time being spent in type-checking. After this the program was almost correct — it ran first time and only some minor adjustments had to be made to correct obvious errors.

The program worked so well first time largely because it was written in a typed language. Peyton Jones has written substantial programs in SASL, an untyped functional language, and his experience lends support to this view [Peyton Jones 85]. In a typed language, particularly a polymorphically typed language like Ponder, a large number of conceptual errors and inconsistencies are detected at compile time. In SASL such errors remain in the compiled program and must be found by debugging. This may not seem so bad, but in non-trivial cases debugging is much harder in functional languages, because traditional techniques simply cannot be used. For instance, it is impossible to "just put in a print statement"

— the corresponding modification of a functional program may alter its whole structure.

For the same reason it is difficult to write a parser, get it working, and then put in syntax error diagnosis and recovery afterwards, something which is relatively easy in an imperative language [Turner 85]. It seems that functional languages force clean programming habits. Unfortunately, a large part of current programming practice is revealed to be just 'hacking' — altering a program piecemeal until it does what was originally intended. This is of little use to the functional programmer who is forced to maintain the consistency of his program at all times.

Hacking is much more fun than trying to get a complete and consistent design before writing any part of the program. Functional programming in a typed language forces the latter, but I expect that if a programmer spent as much thought on an imperative program as he is now forced to spend on a functional program then the resulting imperative program would also work first time. It seems paradoxical that in order to make programming easier we must make it more difficult to write programs, but that certainly appears to be the case.

Low level software

Most functional programmers do not consider functional languages suitable for writing low level software, and very few people seem to have attempted this. Richard Sykes at Imperial College, London has written various 'operating system' programs, including software to drive a robot arm. I wrote the low level parts of *ANS*, to handle the mouse and graphics as more or less independent experiments (although they helped to clarify my ideas about how to write the spreadsheet itself). I found that for this low-level work, programming in a functional language was not much better than programming in a conventional language. I did find that higher order functions and partially parameterised curried functions were useful, and these would not have been available in a conventional language.

Most of the complexity of ANS comes from having to cope with the vagaries of hardware and people. Programs to present information to people in an intuitively appealing fashion are not very systematic (they have lots of special cases), presumably because people's expectations of what a program should sensibly do in a particular combination of circumstances are so bizarre and arbitrary. Programs that drive hardware directly or interface to communications protocols are often ad-hoc for similar reasons. To write such programs in a functional language involves "going against the grain", since functional languages are easiest to use

when expressing an elegant mathematical solution to a problem, rather than an ad-hoc heuristic one. Perhaps rule based systems are better for this kind of thing. Of course, there is no reason why the rule based systems themselves could not be written in a functional language, but I have not attempted to do this.

Stream processing functions

In the remainder of this section I present two programming techniques , 'stream processing functions' and 'almost circular definitions', which can only be used in lazy functional languages. These techniques have been invented independently in a number of places. I found them invaluable when writing *ANS*.

The first technique is the use of stream processing functions (which will be abbreviated to SPFs). I believe the name was coined by Simon Jones in [Jones 84], which is quite a thorough treatment of SPFs. However, Jones does not discuss higher order functions on SPFs, which I have found invaluable when writing interactive programs. A stream is a lazy list: the term was invented before lazy languages were thought practical. Early functional languages often had streams, providing the languages with some laziness, but were otherwise evaluated eagerly.

I will use Ponder notation to describe SPFs, but I will also give explanations in English, so it should be fairly obvious what is going on. A brief and informal description of Ponder is given in Appendix B. Those unfamiliar with the language may find it useful to read this.

There are two kinds of SPF, *simple* and *complex*. A functional program that takes input lazily from a keyboard and prints results lazily on a terminal is an example of a simple SPF. These functions just take a stream, process it somewhat and spit out another stream as their result. In Ponder a simple SPF would have a type like

$$\mathsf{List[Input]} \rightarrow \mathsf{List[Output]}$$

This type describes a function (hence the arrow) that takes a list of some objects of type Input, and returns a list of objects with type Output. The lists are implicitly lazy. An example of a simple SPF is a function which upper-cases the characters in its input list and returns this as its output list:

$$\mathsf{Let}\ \mathit{UpperCasingSPF} \ = \ \mathit{map}\ \mathit{Upper}$$

Where *Upper* is the function to uppercase a single character. *UpperCasingSPF*

has the type

$$\mathsf{List[Character]} \rightarrow \mathsf{List[Character]}$$

These simple SPFs can of course be recursive, although they are still relatively easy to understand. The complex SPFs really are more complex, because they have a 'state' as well. They have types like

$$\mathsf{State} \rightarrow \mathsf{List[Input]} \rightarrow (\mathsf{List[Output]} \times \mathsf{List[Input]} \times \mathsf{State})$$

This is the type of a function which takes an object of type State, then a list of Inputs and returns a tuple consisting of a list of Outputs, another list of Inputs and a new State. This kind of function is at the heart of all interactive functional programs. From the List [Input] and the State it will construct a List [Output] using only the first few elements of the input. It returns this, together with the tail of the input, which it has not used, and a new State. The tail, or 'stub', of the input stream that is handed back can then be passed to another SPF with the new state, so the entire input list can be consumed incrementally. One easy slip to make with complex SPFs is to hand back the *whole* input in the result rather than the 'stub' of the input. This will probably make the program go into an infinite loop.

The difficulty comes when writing recursive complex SPFs, since it is often hard to know where to start. This is really no different from writing any recursive definition, but it is easy to get confused. The best way to write such functions is not to use recursion at all, but to use higher order functions. For instance a higher order function can be defined to pipe the output of one SPF into the input of another:

```
Let Also =
    ∀Input. ∀Output. ∀State.
    (State → List[Input] → (List[Output] × List[Input] × State)) S1 →
    (State → List[Input] → (List[Output] × List[Input] × State)) S2 →
    State state →
    List[Input] in →
    (List[Output] × List[Input] × State):
 Begin
  Let out1, in1, state1  =  S1 state in;
  Let out2, in2, state2  =  S2 state1 in1;
  (append out1out2), in2, state2
 End
```

Thus if A and B are complex SPFs, *Also A B* will be a new complex SPF which is the 'composition' of these two functions. *Also* is not recursive, but here is a useful function that is, and saves a lot of thought.

```
Letrec Repeat =
    ∀Input. ∀Output. ∀State.
    (State → List[Input] → (List[Output] × List[Input] × State)) S1 →
    State state →
    List[Input] in →
    (List[Output] × List[Input] × State):
 Begin
  Let out1, in1, state1 = S1 state in;
  If null out1
  Then out1, in1, state1
  Else Let out2, in2, state2 = Repeat S1 state1 in1;
       (append out1 out2), in2, state2
  Fi
 End
```

Now *Repeat A* will be a complex SPF that returns all the output that A would return when repeatedly applied to the results of its previous invocation, until A returns a null list. It is interesting to note that functions for combining SPFs are very much like the functions used in denotational semantics when describing the meaning of imperative programs. The fact that denotational semantics could be used for programming as well as for describing programming languages was pointed out in [Schmidt 82]. Schmidt remarks in this paper that by using denotational semantics rather than an imperative language one can define combining forms (for instance *Also* and *Repeat*) suitable for particular problems, rather than having to accept the built-in operations of a particular imperative language.

Although these higher order functions require some thought, there is now no need to go through the mental contortions of writing recursive complex SPFs. This fits in well with the idea that explicit recursion in functional languages is a bad thing and that higher order functions ought to be used instead. Even if a higher order function has to be written specially for a particular piece of program, the chances are that the same kind of structure will be needed again. The higher order function will be there already so it will be less effort to write the new program.

Almost circular definition

The second programming technique, 'almost circular definition', is even more bizarre. The technique is a form of recursive definition — but recursive definition

of data structures rather than of functions. This works in lazily evaluated languages because the data structures do not have to be constructed in their entirety before being used, but can be constructed and used piecemeal. As an example, consider this definition of an infinite list containing all the integers.

$$\begin{array}{l}\mathsf{Let}\ \mathit{AddOne} \ = \ \mathit{map}\ (\mathit{Add}\ 1)\ ; \\ \mathsf{Letrec}\ \mathit{integers} \ = \ \mathit{cons}\ 1\ (\mathit{AddOne}\ \mathit{integers})\end{array}$$

The first element of *integers* can be created immediately — it is 1. To calculate the second element, the first element must be known. We add one to each element of the whole list of integers and take the head of the resulting list — 2. To find the third element we need to know the second, and so on. Provided we never need to know an element's own value to determine that value then this technique is safe. In practice we must also make the proviso that to find the value of an element it is only necessary to inspect a finite number of other elements.

Another example that yields neatly to this programming technique is the mintree problem. Here the problem is to take an arbitrary tree with integers at the leaves and return a tree with identical structure, but with all the integers replaced by the minimum of the first tree's leaves. Here is a solution:

$$\begin{array}{l}\mathsf{Letrec}\ \mathit{MinTree} \ = \ \mathsf{Tree}\ t \rightarrow \mathsf{Int}\ \mathit{min} \rightarrow (\mathsf{Tree} \times \mathsf{Int})\ : \\ \quad \mathsf{If}\ \mathit{Leaf}\ t \\ \quad \mathsf{Then}\ \ \mathsf{Let}\ i \ = \ \mathit{LeafValue}\ t; \\ \qquad\quad (\mathit{MakeLeaf}\ \mathit{min},\ i) \\ \quad \mathsf{Else}\ \ \mathsf{Let}\ T_1,\ T_2 \ = \ \mathit{BranchValue}\ t; \\ \qquad\quad \mathsf{Let}\ \mathit{NewT}_1,\ \mathit{Min}_1 \ = \ \mathit{MinTree}\ T_1\ \mathit{min}; \\ \qquad\quad \mathsf{Let}\ \mathit{NewT}_2,\ \mathit{Min}_2 \ = \ \mathit{MinTree}\ T_2\ \mathit{min}; \\ \qquad\quad (\mathit{MakeBranch}\ (\mathit{NewT}_1,\ \mathit{NewT}_2), \\ \qquad\quad \mathsf{If}\ \mathit{Min}_1 < \mathit{Min}_2\ \mathsf{Then}\ \mathit{Min}_1\ \mathsf{Else}\ \mathit{Min}_2\ \mathsf{Fi}) \\ \quad \mathsf{Fi};\end{array}$$

$$\mathsf{Letrec}\ \mathit{MinimimTree},\ \mathit{min} \ = \ \mathit{MinTree}\ \mathit{InputTree}\ \mathit{min}$$

The function *MinTree* returns a pair containing the minimum tree (in which all the leaves are the minimum of the leaves in the input tree) and also the minimum of the input tree. It builds these two answers in parallel as it traverses the tree. *MinTree* can build the minimum tree in one pass because it is supplied with the minimum as its second argument, and it assumes that this is correct. The final line "joins the ends together" by passing the integer *min*, returned by *MinTree* back to *MinTree* as its second argument, so that it can be used to fill in the leaves of the minimum tree.

The "central loop" of *ANS*, by means of which the old state ('previous' values) is changed to produce a new state uses 'almost circular definition'. It is extraordinarily terse:

Letrec *new_state* = *transition_function* *old_state* *new_state*

The transition function takes the old state, and the new state, *which is being defined*, and produces that new state. All the control necessary, in the form of a dependency analysis of the components of the new state, is performed automatically through lazy evaluation.

This technique of using the answer before it is all there can only be used in a lazy language. Provided that there is no attempt to use the value of part of the new state before it is possible to determine it, this technique is completely safe. In [Bird 84] further examples of almost circular definition are given. Bird makes the point, which is true of both the techniques I have described, that great care must be taken to ensure that the program is safe — that it will terminate. It is easy to make a trivial slip, such as returning the wrong object from a function, which can cause the program to crash.

5.4 Problems with time

One of the unexpected problems that I ran into when writing *ANS* was the way that input-output dependencies caused strange program behaviour. As an example, consider the following program fragment:

```
If some condition
Then append "Hello" " there"
Else append "Hello" " world"
Fi
```

This appears to be equivalent to

```
append "Hello"  If some condition
                Then " there"
                Else " world"
                Fi
```

However, the two fragments will behave quite differently in some circumstances. Suppose that '*some condition*' depends directly on the input from a terminal, and cannot be evaluated until a key is pressed. Suppose also that the output from this fragment is sent directly to the screen of the terminal. Due to the demand-driven

nature of lazy evaluation, the first fragment will do nothing until a key is pressed, when it will output either "Hello there" or "Hello world". The second fragment on the other hand will output "Hello" immediately, and wait for a key to be pressed, after which it will output either " there" or " world".

This is an example of output coming out after it was expected, but it can be equally bad for output to come out before it was expected. The above example is a bit contrived and would only be written by a programmer in a moment of thoughtlessness, as it is obvious what will happen. In more complex programs similar things can happen that are very difficult to spot in advance.

For instance in *ANS* one of the bugs was in the "move" command, which moves a cell from one position to another on the screen. The intended sequence of actions was that the user would point to the "move" symbol on the screen with the mouse and select it, then point to the cell to be moved, select that, and finally point to the new position. The cell should then be erased from its old position and re-drawn in its new position. What actually happened was that as soon as the cell was selected it disappeared, only to re-appear wherever the mouse was pointing when the mouse-button was next clicked. This happened because in lazily evaluated programs output is driven by demand, and when the prerequisites for some output are available, that output is performed. To erase the cell from its old position, all that need be known was that old position. The erase did not depend on the new position having been given, so it happened as soon as it could. The trick I used to cure this was to send to the terminal an innocuous graphics command using the new position, ensuring that the erase could not be done until the new position had been specified.

It is always possible in a particular case to invent a "kludge" like this, but the code will often seem arbitrary and inelegant because of it. Without a warning comment in the program such a device could easily be misunderstood by another programmer and erroneously altered.

In programs that communicate with external devices using some protocol, the precise way in which input and output are interleaved is often very important. It is disturbing that interleaving errors are so easy to put into functional programs because of the counter intuitive nature of lazy evaluation. A possible solution to this problem has been suggested by Stoye [Stoye 85]. His solution is to make programs *ask* for their input by placing a special token on their output stream. This ensures synchronisation, because the program will not receive any input until it asks for it. This mechanism also allows environment enquiries, such as "Has the

break key been pressed?" — placing another special token on the output would cause a token giving the required information to be placed on the input.

5.5 Problems with space

Space Leaks

Lazy functional programs are inefficient in unpredictable ways, and this is often because they use storage inefficiently. Programs often retain large structures when it would be possible to discard them, causing "space leaks". Apart from the obvious point that if recursion is implemented naïvely programs will soon run out of stack space, there are more subtle points to do with garbage collection. For instance, suppose a function F has in its stackframe a pointer to a large object on the heap which will not in fact be used again and it calls another function G, performing a long computation before returning a result (see Figure 5.5). The

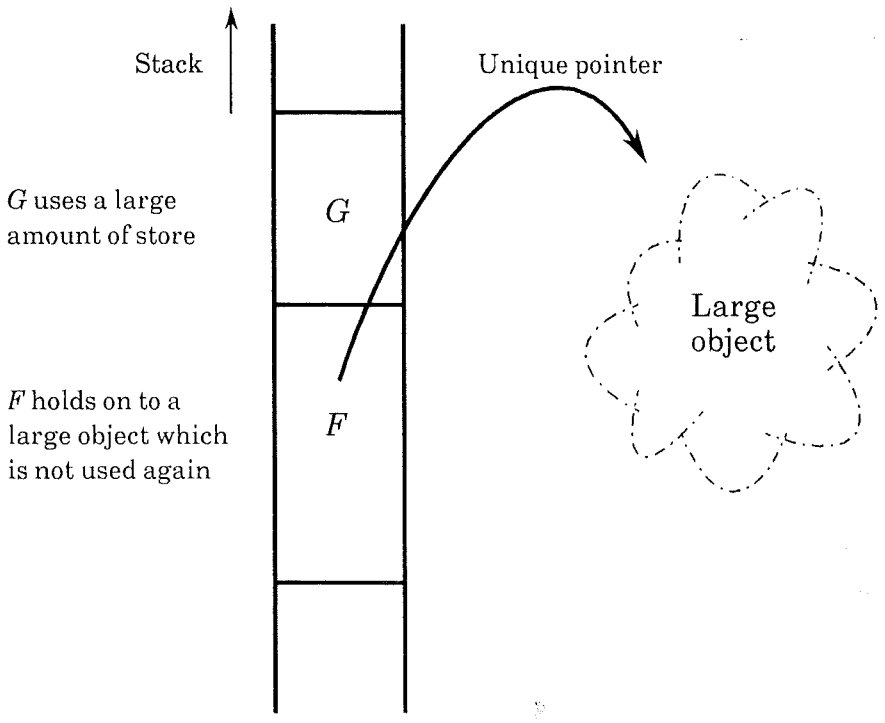


Figure 5.5: F retains a large object

time taken to do that computation will be greater than it would have been if the unnecessary data structure had been discarded before calling G since garbage collections will be done more often. It may even be the case that the computation will not be possible at all because the program runs out of store in G. Note that the data structure which is to blame will certainly be discarded and is in no

way needed, so a clever implementation could overwrite the pointer to it in the stackframe of F before calling G. The data structure would then be reclaimed by the garbage collector when it was first invoked during the evaluation of G, rather than when G had finished and the stackframes of both F and G were relinquished.

I have noticed space leaks in several programs. For instance after using *ANS* for about an hour it is noticeably slower than it is at the start, because of a space leak. The main function, within which the execution of the whole program is nested, holds on to the head of the input list. As characters are typed at the keyboard, this list grows longer and longer. The amount of free storage decreases and the program gets slower because garbage collections must be performed more frequently.

Another way to cause a space leak is to form a closure, containing a pointer to a large structure, which is not evaluated because of laziness. Thus the whole structure is retained, even though when the closure is finally evaluated only a small fragment of it is actually needed. If this is done repeatedly, or if the large structure grows, large amounts of store can be wasted.

For example, suppose the expression $left\ (a, b)$ is passed to a function that does not need its value immediately. This will retain not only a, which it might look at, but also b, which it cannot look at, but which is part of the pair so it cannot be discarded. The whole pair is retained because the evaluator cannot reduce $left\ (a, b)$ in a lazy context, but must instead form an application tree containing the function $left$ applied to the argument (a, b). Only when the function needs to know the value of its argument can $left\ (a, b)$ be evaluated and b be discarded. Note that b could be a very large structure.

In this case a clever garbage collector could perform the reduction

$$left\ (a, b) \xrightarrow{*} a$$

immediately. However in general there could be an arbitrary function applied to a pair, or even to a closure that would (eventually) evaluate to a pair. In this case the garbage collector could not always decide whether it was safe to attempt the reduction, so it would have to err on the side of caution. Thus there will always be some situations where space leaks could occur.

The "dragging problem" [Richards 84] is another kind of space leak, where a result is produced as two lists returned in a tuple. If there is a dependency between the head of the second list and the head of the first list, then even if the two lists could be made and discarded a piece at a time, in constant storage, the whole of the first list may be retained. This will happen when the whole of the first list is

used before the second list is inspected, for instance when producing output from a program.

Large lazy structures

Infinite data-structures are not a problem when they are held in procedural form, but arbitrarily large parts of them can be instantiated, with no easy way of going back. No implementations I am aware of allow data structures to be collapsed back into their procedural form to save space, although Turner described the idea some time ago [Turner 79]. The problem is knowing what to keep, since the implementation itself cannot make intelligent decisions about what to keep and what to collapse. Indeed it may be the case that an original expression and its final reduced form are rather large, but that an intermediate representation is much smaller. To keep track of reductions closely enough to spot such size changes would be a tremendous overhead.

Comparison of two programs, one using a lazy data structure and a similar one calculating values "on the fly" will show the difference that instantiating large parts of lazy structures can make. As an example I have compared the performance of two versions of the dragon program, used in the previous chapter as a benchmark. The first program, Slow Dragon, uses an infinite tree structure to store the dragon curve to an infinite depth. To draw the curve to a particular depth, the program descends the tree and takes data from that depth. The other program, Dragon, does not have such a data structure, but instead calculates the data it needs "on the fly" for each for each depth as required. The crucial portions of both programs are reproduced in appendix C.

Depth	Slow Dragon	Dragon	Gain
6	4.75s	4.15s	15%
7	9.39s	7.49s	25%
8	22.56s	18.64s	21%
9	43.93s	34.60s	26%
10	101.9s	74.1s	37%
11	232.8s	134.1s	74%
12	1342.7s	278.22s	383%

Slow Dragon is slightly slower, but as the instantiated data structure grows larger, so the amount by which Dragon is faster increases. Because Slow Dragon must instantiate larger portions of its data structure there is less free space for other work. Garbage collections become more frequent with increasing depth,

causing a drastic increase in the time needed to produce an answer.

A similar problem to the expansion of data-structures is that of the expansion of functions. Consider the following definition

```
Let chop = Int n →  If n = 1
                    Then tail
                    Else tail ∘ (chop (n − 1))
                    Fi
Let f = chop 100;
```

'∘' is an infix *compose* operator.

When f is first used, $\underbrace{tail \circ \cdots \circ tail}_{100}$ will be built. This is not very useful, particularly if it is only used again after a long period. It just uses up storage to no purpose — if it would cost less to recalculate it than it would in extra garbage collections, it should be kept in a compact procedural form. It is difficult to spot such things automatically, since that involves estimating how often the function will be used and how big the expanded form would be.

It seems unfortunate that the time needed to obtain a result and even whether or not a result will be obtained should depend so critically on the amount of cleverness in the implementation. A naïve programmer could not be expected to know that he should write his program one way rather than another to make it more efficient. This is also the case with many conventional languages, but one of the aims of functional programming is that programs should be clearer. As far as judging their efficiency is concerned they can be just as bad as the worst conventional languages.

Uncertainty about the time and space behaviour of functional programs is the worst blow to their credibility where guaranteed performance is needed. Although the problem of "lumpy" performance can be cured by using concurrent garbage collection, the space efficiency problems remain.

5.6 Conclusions

These last sections have painted a somewhat gloomy picture of functional programming. I have deliberately refrained from giving the standard arguments about the simplicity, semantic elegance and so on of functional languages. Although largely true, these arguments have been repeated many times before. To someone who has never written in a typed functional language it may seem puzzling that functional programmers are still so keen on their languages despite all the problems

of efficiency. The arguments about simplicity and elegance are only half valid — what really matters is the ease with which programs can be written. This topic is seldom addressed directly by functional programmers, but I think it holds the key to understanding the appeal of functional languages.

The key fact is that programs in typed functional languages often work first time. This may not seem very important, but computer programmers are a fallible lot, and well aware of their own fallibility. They have it rubbed in every time they debug their programs. The attraction of typed functional languages is that they largely prevent minor mistakes from entering programs. Major misconceptions are of course possible, but this is still an improvement. The thrill of having programs work first time, or even go wrong in ways that are easy to mend, is something that most programmers can only imagine. It never happens to them.

But functional programming has not taken the world by storm — the functional programming community still consists of people designing and implementing functional languages, rather than people using them to write new applications. The truth must be admitted that however much its proponents euglogise about functional programming it has not been an instant success. There has been very little practical programming in functional languages, largely because the implementations have been so slow. Now at last we are beginning to see functional languages that are sufficiently fast that we can write real programs — for instance LML and Ponder. Some unexpected problems have arisen, but programming in functional languages seems to be more or less as pleasant as its pioneers promised. What is needed now is to make the implementations even more efficient, so that they are comparable to traditional languages. I see no fundamental problems here, though it will certainly be several years before this state is reached.

Chapter 6

Conclusions

6.1 Strictness analysis

A new analyser

I have devised a new method of strictness analysis that is substantially different from Mycroft's original method [Mycroft 81]. Mycroft's analyser finds a function strict on an argument if the result of the function is undefined when the argument is undefined. My analyser finds a function strict on an argument if the argument must be evaluated to produce the result. Simple rules are used to determine the strictness properties of an expression from the properties of its subexpressions.

The new analyser accepts programs containing higher order functions, but functional arguments are treated as though they were completely lazy. No strictness information is lost when a function returns another function. I describe this analyser as "first order", although it is more powerful than Mycroft's first order analyser.

This analyser has been extended to "second order strictness". The extended analyser finds better strictness properties for higher order functions, unless their functional arguments are themselves higher order functions. In that case the analyser assumes that the functional arguments are lazy.

The "second order" analyser is not as powerful as other higher order analysers based on Mycroft's work [Abramsky et al 85, Hudak & Young 85], but it can analyse many more programs than they can. The other higher order analysers cannot cope with self application, since they will only work on monomorphically typed languages. My analyser works on polymorphically typed languages such as Ponder, and can even assign reasonable strictness properties to a defined version of the Y combinator.

Results

I have implemented both first and second order analysers in the Ponder compiler, and used them to compile substantial programs. The results are rather disappointing.

The first order analyser gave a worthwhile performance improvement on typical programs — between 50% and 10% speed-up. This is not as large as I had hoped for. It appears that strictness detection does not improve performance masssively: it is just another small optimisation.

The second order analyser produced exactly the same speed-up as the first order analyser. This was completely unexpected — I had imagined that it would give another small performance increase. Although my analyser is only second order it seems unlikely that higher order analysers would be any better, because of the way strictness information is used to produce better code.

My first order analyser is very fast, but the second order analyser is much slower and more complex. Until strictness analysers for lazy lists have been perfected it seems that first order analysers are the most practical.

Lazy lists

I have described informally how my first order analyser could be extended to lazy pairs, and other proposals for strictness analysis of lazy structures have been made in [Hughes 85b] and [Kieburtz & Napierala 85]. All these proposals share the defect that the description of the strictness properties of a function taking a lazy structure as argument may be infinite, because the function can be recursive. Some of these infinite descriptions can be represented in finite space using a context free grammar — such descriptions are called 'rational'. However, there is no guarantee that the least fixed point of the strictness description of a recursive function will be rational. Indeed, many functions do not have a rational strictness description — for instance the function which takes the sum of the diagonal of a list of lists does not.

The best that could be hoped for would be to find a rational solution somewhat lazier than the true least fixed point. In practice the analysers must either "overshoot" by finding a strictness description which is far too lazy to be useful, or else risk not terminating by attempting to find a rational description for a function which does not have one. These fundamental difficulties make strictness analysis of lazy lists much harder than higher order strictness analysis. It may be necessary for a programmer to offer guidence to a lazy list analyser, or to have heuristics to

guide it in producing useful strictness descriptions.

An alternative to this, suggested by Wadler [Wadler 85], involves throwing away a lot of information so as to guarantee that the analyser will get an answer for all programs. This may prove to be the best approach in the short term.

Parallelism

Strictness analysis allows lazy functional languages to run efficiently on parallel machines. Without it the only source of parallelism is in the arguments to strict built-in functions. However, two further developments are necessary:

- Strictness analysis must become much more powerful. Most of the power of functional languages comes from the use of higher order functions and lazy lists. Until strictness can be found in these there will not in my opinion be enough useful parallelism in programs to justify more than two or three processors.

- When enough parallelism has been found it must be restricted, so that small computations are performed locally rather than using the parallel evaluation mechanism. Some form of cost analysis is necessary, so that program fragments are evaluated in the way that takes least time. If a computation is sent to another processor for execution, a substantial amount of work must be done on that processor. If only a small amount of work is done, communications costs will predominate and the program may run slower on a multiprocessor than on a uniprocessor machine. The balance between too little and too much parallelism will have to be struck afresh for different parallel machines, since they will have different performance characteristics. It would be preferable to be able to assess the cost of a program fragment from a formal description of the machine it is to run on, rather than have to adjust a compiler by trial and error.

6.2 Practical experience

I have written substantial interactive programs in Ponder. On the whole this has not been difficult, although there have been some unexpected problems.

Low-level software

I have written graphics and mouse interfaces for Ponder, involving much arbitrary 'bit-shuffling' due to the ad-hoc nature of the hardware and protocols involved. I found that writing in a functional language was not much help. The programming task would not have been much harder in an untyped conventional language like BCPL.

Type checking

Polymorphic typing is extremely useful, as it allows programs to be written in a natural style, with easy re-use of functions. Many conceptual errors are caught at compile-time that would otherwise have to be tracked down at run-time by debugging. This is fortunate, because debugging lazy functional programs is much more difficult than debugging conventional programs.

Input-Output peculiarities

In conventional languages the sequencing of input and output is done explicitly by sequencing the input and output commands in the program. In a lazy functional language the sequencing of input and output is done by implicit dependencies between the objects in the input stream and objects in the output stream. An unforseen effect of lazy evaluation was that of output from the program arriving at unexpected times relative to the input that it was given. Stoye has proposed a solution to this problem in which the program asks for each piece of input in turn, by putting a special token on its output stream [Stoye 85], but I have not experimented with this scheme.

Efficiency

Functional languages are notoriously slow, and there is an unpredictability in their slowness. For instance the effects of optimisations on a program are not cumulative, and they can affect seemingly similar programs in different ways. In chapter four I gave the example of Quicksort being faster than insertion sort on a simple lazy implementation, but slower on a compiled implementation with strictness analysis.

The problem of 'space leaks' is another source of uncertainty. Space leaks occur if a large data structure is retained when it is no longer needed. This causes a general slowing down due to more frequent garbage collection. In extreme cases

this can cause a program to crash unnecessarily by running out of store. Code which causes space leaks is difficult to spot, and although many cases could be spotted automatically it would be impossible to find them all.

Programming Techniques

I found lazy evaluation to be a very useful mechanism, though it takes quite a lot of getting used to. With lazy evaluation and higher order functions, very powerful combining constructs can be built for particular applications, rather than having to take what is provided as in conventional languages. I have described two programming techniques for lazily evaluated languages, 'stream processing functions' and 'almost circular definition' I found these extremely useful when writing interactive programs. Doubtless further novel techniques will be invented, since practical functional programming is still in its infancy.

6.3 The future

Five years ago lazy functional languages could get by without type-checking. Now it is generally accepted that type-checking is a necessity. In the future I think that more and more supporting tools now considered optional will be accepted as necessary. Strictness analysers almost fall into that class now, and I think program transformers soon will as well.

There has been much interest in the past in transforming very high level programs into functional languages, so that they could actually be run on machines, but rather less in transforming functional programs so that they are more efficient. Most functional language compilers include ad-hoc transformations, but it would be more satisfactory to include a general mechanism for building transformations, so that a programmer or compiler writer could add to the rules in a systematic way.

A way of adding to the rules unsystematically would be to allow a programmer to simply assert that some property of a program was true. For instance he might assert that a particular list would never be null. This opens up the possiblity of disaster — eventually a programmer will make a mistake and a program will crash. Although this approach is simple to implement for the compiler writer, it should be recognised for what it is: an expedient bodge.

It is far better to incorporate in the programming system some mechanism for proving properties of programs and parts of programs, so that optimisations are

never applied unless they are safe, and the correctness of programs is never compromised. It is likely that a large number of strategies built with these mechanisms would be of general use, and would be incorporated into the standard compiler. The naïve programmer would then have the benefit of these heuristics, while the expert could optimise arbitrary programs using the full proof system.

Functional languages will certainly have an impact on the way people program, but it may be rather indirect. As James Morris commented:

> Functional languages as a minority doctrine in the field of programming languages bear a certain resemblance to socialism in its relation to conventional, capitalist economic doctrine. Their proponents are often brilliant intellectuals perceived to be radical and rather unrealistic by the mainstream, but little-by-little changes are made in conventional languages and economies to incorporate features of the radical proposals. Of course this never satisfies the radicals, but it represents progress of a sort. [Morris 81].

The most common programming language in thirty years time may well be a functional language, but it will still be called FORTRAN.

Appendix A

How to generate good code

This appendix describes how to produce stack based intermediate code, from a program expressed as super-combinator definitions. The code generator described here makes full use of strictness information from both my first and second order strictness analysers. The same syntax is used for the super-combinator program as was used in chapters 2 and 3 for the input to my strictness analyser, with the important restriction that lambda abstractions can only appear at the top level of a function, so

$$F \;=\; \lambda a.\ \lambda b.\ \lambda c.\ b\ c\ a$$

is fine, but

$$F \;=\; \lambda a.\ \lambda b.\ b\ (\lambda c.\ c)\ a$$

is forbidden. If Hughes' algorithm [Hughes 82] is used to produce super-combinators from the functional language source program these will be in the correct form.

As it stands Hughes' original algorithm is too pessimistic and produces super-combinators which are unnecessarily small. Several optimisations have been proposed, notably in [Fairbairn 85] and [Hudak & Goldberg 85]. These optimisations bring large performance increases, more than doubling the performance of typical programs.

To produce good code it is therefore vital to perform these optimisations to the super-combinators before attempting to produce stack based intermediate code.

Intermediate code

The intermediate code used here is a slightly simplified version of the Ponder intermediate code, which is also similar to that used in the LML compiler [Johnsson 83]. There are only six instructions: `Push`, `Fap`, `Call`, `Return`, `Eval` and `Switch`. These act on two stacks and a heap. The *return stack* holds return addresses and the

argument stack holds the arguments to combinators. The heap is used to store application trees.

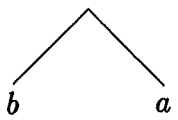
Taking the instructions in turn:

- `Push` pushes an object onto the argument stack. This can be used to push integers, combinator entry points, or any other objects. To push the n^{th} argument of the current combinator, `Push arg` n is used.

- `Fap` stands for "form application" and it is used to build application trees. It removes the top two items from the argument stack and places them in a new application node on the heap. A pointer to this new node is then pushed onto the argument stack. For example

  ```
  Push a
  Push b
  Fap
  ```

  Would leave a pointer on the argument stack to the newly created application node

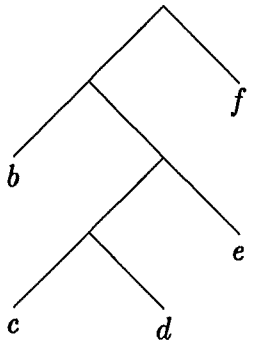
  To generate an application tree corresponding to the expression

  $$b\ (c\ d\ e)\ f$$

  the following code would be produced:

  ```
  Push f
  Push e
  Push d
  Push c
  Fap
  Fap
  Push b
  Fap
  Fap
  ```

  This would leave a pointer to the following application tree on the argument stack:



- `Call` pushes a return address onto the return stack, then jumps to the entry point of a combinator. Combinators provided by the run-time system (built-in functions) and combinators produced from the user's program are treated alike. The intermediate code has no operations for arithmetic, list processing and so on, because these are provided by combinators in the run-time system. To add two numbers, *a* and *b*, the following code would be produced:

  `Push` *a*
  `Push` *b*
  `Call` *Plus**

  Arguments to a called combinator are pushed on to the argument stack.

- `Return` returns from a called combinator. The top object on the argument stack is removed — this is the result of the combinator. The next objects on the stack are the arguments to the combinator. These are removed and discarded. The result is then pushed back on to the argument stack. Finally the return address is removed from the top of the return stack and execution resumes at that point. In the Ponder implementation all combinators are defined to return a fully evaluated result (in other words if the evaluator is used on the result of any combinator that result will be unchanged afterwards).

- `Eval` causes an object to be evaluated. It has two versions:

  - `Eval top` evaluates the object on the top of the argument stack. The top of the stack will contain the evaluated version afterwards.
  - `Eval arg` n evaluates the n^{th} argument of the current combinator. Subsequent use of `Push arg` n will access the evaluated version.

- `Switch` is a construct rather than a single operation. It is used to implement the conditional expression If-Then-Else-Fi.

`Switch`

code for Then *part*

`Or`

code of Else *part*

`EndSwitch`

Switch removes the top object from the argument stack and treats it as a boolean. If it is 'true' the first block of code is executed, and if it is 'false' the second block is executed. After either of these, execution resumes with the next instruction after the `EndSwitch`.

That concludes the summary of the intermediate code. It is easy to translate this code into machine code for conventional machines, so I will not describe how to do this. It is interesting to note that the intermediate code has no facilities for GOTOs or assignments.

Examples

Before giving an exact description of how to generate this intermediate code, I will give some examples of combinator definitions and the code that is produced for them. I will start with factorial.

$$
\begin{array}{lcl}
\mathit{factorial} & = & \lambda n.\ \mathsf{If}\ (\mathit{Equal}\ n\ 0) \\
 & & \quad \mathsf{Then}\ 1 \\
 & & \quad \mathsf{Else}\ (\mathit{Times}\ n\ (\mathit{factorial}\ (\mathit{Minus}\ n\ 1))) \\
 & & \quad \mathsf{Fi}
\end{array}
$$

When generating code it it necessary to take the modes of functions into account. The mode of *factorial* is $\mathsf{S} \rightarrow \mathsf{m_L}$. The intermediate code for this is

```
factorial:
    Eval arg 1
factorial*:
    Push arg 1
    Push 0
    Call Equal*
    Switch
     Push 1
    Or
     Push 1
     Push arg 1
     Call Minus*
     Call factorial*
     Push arg 1
     Call Times*
    EndSwitch
    Return
```

Every combinator (including the ones provided by the run-time system) has two entry points. The first one — *factorial*: in this example — is the entry point used if the combinator is called from the evaluator. The other entry point — *factorial**: — is used if the combinator is called directly from intermediate code. In that case all the strict arguments to the combinator will have been evaluated already, so the `Eval arg 1` which follows the first label can be skipped over. The recursive call is thus to the *factorial** label, since the argument on the stack is guaranteed to be fully evaluated since it has just been returned by the call to *Minus**. Where the argument is passed to a strict combinator, as in

```
Push arg 1
Push 0
Call Equal*
```

it must be fully evaluated first. An optimisation which saves a lot of work is to remember whether an argument has already been evaluated, rather than putting `Eval arg` n before every `Push arg` n.

As another example, consider the definitions

$$\begin{aligned} F &= \lambda x.\ (Plus\ x\ x) \\ G &= \lambda x.\ \lambda y.\ (Plus\ y\ (F\ 3)) \end{aligned}$$

F has mode $\mathsf{S} \rightarrow \mathsf{m_L}$ and G has mode $\mathsf{A} \rightarrow \mathsf{S} \rightarrow \mathsf{m_L}$. The intermediate code for these combinators is

```
F:
    Eval arg 1
F*:
    Push arg 1
    Push arg 1
    Call Add*
    Return

G:
    Eval arg 2
G*:
    Push 3
    Call F*
    Push arg 2
    Call Add*
    Return
```

In this case G only needs to ensure that its second argument is evaluated, because it does not use its first argument.

Code generation

The syntax of combinators that can be generated into intermediate code using the scheme described here is

$$F_i \;=\; \lambda b_1.\ \ldots \lambda b_n.\ \mathcal{E}$$

Where

$$\mathcal{E} \;=\; \begin{cases} K & \text{constant or built-in function} \\ b_i & \text{bound variable} \\ F_i & \text{combinator} \\ \mathcal{E}_1\ \mathcal{E}_2 & \text{application} \\ \text{If } \mathcal{E}_1 \\ \text{Then } \mathcal{E}_2 \\ \text{Else } \mathcal{E}_3 \\ \text{Fi} & \text{conditional} \end{cases}$$

If F_i has n bound variables and its mode is M then its intermediate code is

```
Fi:
    Eval arg i   If M[i] = S, for i = 1...n.
                 This ensures all strict arguments are evaluated.
Fi*:
    generate E 0 M[n+1...]
    Return
```

The function *generate* is used recursively to generate the body of the combinator. It takes three parameters: the expression to be generated, the number of objects that have been pushed on the argument stack and the mode of the result of the enclosing expression.

generate $\mathcal{E}$ n M is defined to produce the following intermediate code:

If $M = \mathsf{m_D}$, produce

```
Call *trap*
```

where `*trap*` halts the execution of the program with an error message.

Otherwise:

1. When $\mathcal{E}$ is a constant K, *but not a* built-in function, produce:

```
Push K
```

2. When $\mathcal{E}$ is a bound variable, b_i: If b_i has not already been evaluated and $n = 0$, produce:

```
Eval arg i
```

   Whether or not it was evaluated, then produce

```
Push arg i
```

   If $n > 0$, then produce n `Faps`:

```
Fap
...   } n
Fap
```

   If M is not a mode containing an arrow, then produce

```
Eval top
```

3. When $\mathcal{E}$ is a combinator F_i with m bound variables, or a built in function K taking m arguments, let $F = F_i$ or K as appropriate.

   - If $m > n$, produce code to push the entry point which will subsequently evaluate all the strict arguments of F:

Push F

Produce n Faps:

$$\left.\begin{array}{l}\texttt{Fap}\\ \cdots\\ \texttt{Fap}\end{array}\right\} n$$

- If $m \leq n$, produce code to call the 'evaluated arguments' entry-point of F:

Call $F*$

Any arguments remaining are dealt with using Fap:

$$\left.\begin{array}{l}\texttt{Fap}\\ \cdots\\ \texttt{Fap}\end{array}\right\} n-m$$

Unless $n = m$, check to see if M is a mode containing an arrow. If it is not then produce

Eval top

4. When $\mathcal{E}$ is a function application, $\mathcal{E}_1\ \mathcal{E}_2$, with $\mathcal{E}_1$ having mode M_1 and $\mathcal{E}_1$ having mode M_2.

   - If $M_{1\,[1]} = \mathsf{S}$, produce *generate* $\mathcal{E}_2\ 0\ M_2$
   - If $M_{1\,[1]} = \mathsf{D}$ or A, produce

Push *trap*

As before,*trap* will halt the execution of the program with an error message if it is subsequently evaluated.

   - Otherwise produce *genlazy* $\mathcal{E}_2$

After this, produce code for $\mathcal{E}_1$:

generate $\mathcal{E}_1\ (n+1)\ M$

The mode M is passed to *generate* $\mathcal{E}_1$ unaltered.

5. When $\mathcal{E}$ is a conditional expression,

$$\begin{array}{l}\mathsf{If}\ \mathcal{E}_1\\ \mathsf{Then}\ \mathcal{E}_2\\ \mathsf{Else}\ \mathcal{E}_3\\ \mathsf{Fi}\end{array}$$

and $\mathcal{E}_1$ has mode M_1 and so on, produce

$$\begin{array}{l}\textit{generate}\ \mathcal{E}_1\ 0\ M_1\\ \texttt{Switch}\\ \textit{generate}\ \mathcal{E}_2\ n\ M_2\\ \texttt{Or}\\ \textit{generate}\ \mathcal{E}_3\ n\ M_3\\ \texttt{EndSwitch}\end{array}$$

The function *genlazy* $\mathcal{E}$ is defined to produce code that forms application trees rather than evaluating $\mathcal{E}$.

genlazy $\mathcal{E}$ produces the following code:

1. When $\mathcal{E}$ is a constant or a built-in function K, produce:

$$\texttt{Push}\ K$$

2. When $\mathcal{E}$ is a bound variable, b_i, produce:

$$\texttt{Push arg}\ i$$

3. When $\mathcal{E}$ is a combinator F_i, produce

$$\texttt{Push}\ F$$

4. When $\mathcal{E}$ is a function application, $\mathcal{E}_1\ \mathcal{E}_2$, with $\mathcal{E}_1$ having mode M_1 and $\mathcal{E}_1$ having mode M_2:

   - If $M_{1\,[1]} = \mathsf{D}$ or A produce

$$\texttt{Push *trap*}$$

   - Otherwise produce *genlazy* $\mathcal{E}_2$

After this, produce code for $\mathcal{E}_1$, then a Fap:

genlazy $\mathcal{E}_1$
Fap

5. When $\mathcal{E}$ is a conditional expression,

If $\mathcal{E}_1$
Then $\mathcal{E}_2$
Else $\mathcal{E}_3$
Fi

and $\mathcal{E}_1$ has mode M_1 and so on, produce

genlazy $\mathcal{E}_3$
genlazy $\mathcal{E}_2$
genlazy $\mathcal{E}_1$
Fap
Fap

The purpose of *genlazy* is simply to leave on the stack an application tree for the expression it is given.

Super-combinators with no lambdas

Some super-combinators have no lambdas. These are known as *constant applicative forms*, or CAFs. They could be compiled into pure code, but *generate* would produce code that just built an application tree for the body and returned that. A better solution is to generate a single application tree at compile-time for each CAF, and refer to this from all the places that the CAF is used. This will preserve full laziness, since if the CAF is reduced from one place, all the other places will access the reduced version.

Care must be taken with garbage collection, lest CAFs are reclaimed when they are still referenced from the program. An equally serious mistake is to *fail* to reclaim CAFs when they are no longer referenced. CAFs can expand into arbitrarily large structures on the heap if they are evaluated. If all the structures pointed to by the original CAFs were retained this would cause a space leak, so CAFs *must* be discarded when they are no longer referenced from the program. Note that there may be references to CAFs from within the bodies of compiled combinators, so it is not sufficient to mark objects on the stack when garbage collecting. The bodies of all the combinators that can be referenced by the program must be scanned too, in order to mark the CAFs which they could use.

Optimisations

A very worthwhile optimisation which does not speed the program up much but makes it less likely to run out of store is to merge `Call` f followed by `Return` into one instruction: `Tailcall` f. This discards the current combinator's arguments from the stack exactly as return does (but preserves those destined for f), and then enters f before returning to the current combinator's caller. This frees stackframes that will never be used again, allowing tail recursive functions to run in constant space. `Eval top` followed by `Return` can be replaced by `Taileval` with a similar saving of space.

Appendix B

Introduction to Ponder

This appendix gives an informal introduction to Ponder by means of examples. For a complete description of Ponder the reader should consult [Fairbairn 85] or [Tillotson 85]. The treatment given here is sufficient to cover all the examples used in this thesis and to allow the reader to write simple programs. For an introduction to the lambda calculus see chapter 1.

Syntax

The first example is a program which defines the factorial function and uses it to find factorial 42.

```
Letrec factorial  =  Int n → Int :
                     If n = 0
                     Then 1
                     Else n * (factorial (n − 1))
                     Fi;
factorial 42
```

In the first line Letrec indicates that this is a recursive definition, so the name *factorial* is available within the definition of *factorial*, and refers to the function being defined.

After the '=' sign the type of *factorial* is specified. Int n is equivalent to λn in the lambda calculus, but says that n is an integer. The arrow '→' just says that this is a function. This syntactic sugar is redundant, but it serves to separate the first part of the type from the rest. Since this is a recursive function returning an integer the type ends with Int:. This just says that the result is an integer. It is mandatory to indicate the result type for recursive definitions, but optional for non-recursive definitions.

Infix operators are used in the body of *factorial* — '=', '*' and '−'. There are facilities in Ponder for defining infix, prefix and bracketing operators (but not postfix). I will not describe these here. Most of the constructs and operators usually built in to functional languages are definable in Ponder using these facilities. The standard prelude defines about 150 such operators and functions, using 30 basic functions provided by the run-time system.

It is important to note the difference between a type and the notation used at the head of a function to specify its type and introduce bound variables. For instance the notation at the head of *factorial* is

$$\mathsf{Int}\ n \rightarrow \mathsf{Int}\ :$$

This introduces the bound variable n. The type of *factorial* is

$$\mathsf{Int} \rightarrow \mathsf{Int}$$

This does not mention bound variables.

After the Fi of *factorial* there is a semi-colon. This marks the end of the body of *factorial* and the rest of the program follows. A Ponder program consists of a number of definitions followed by an expression. The number can be zero, but there must be an expression. This expression is the result of the program, which will in this case be factorial 42.

As another simple example, here is the *nfib* function:

$$\begin{array}{lll} \mathsf{Letrec}\ nfib & = & \mathsf{Int}\ n \rightarrow \mathsf{Int}\ : \\ & & \mathsf{If}\ n < 2 \\ & & \mathsf{Then}\ 1 \\ & & \mathsf{Else}\ nfib\ (n-1) + nfib\ (n-2) + 1 \\ & & \mathsf{Fi}; \end{array}$$

Types

The principal difference between Ponder and other functional languages is Ponder's polymorphic type system, which allows quantified types. For instance the identity function can be defined over all arguments as:

$$\mathsf{Let}\ identity\ =\ \forall \mathsf{T}.\mathsf{T}\ i \rightarrow \mathsf{T} : i$$

The ∀T. introduces a new type T, the place of which could be taken by any type when *identity* is used. The definition can be read as "let *identity* be the function

that for all types takes an argument of some type ('... T i...') and returns that argument unchanged ('... $\rightarrow$ T : i...').

The notion of quantified types is similar to that of ML, but the type variables (like T above) are quantified implicitly over the whole type of a function in ML. In Ponder types can be quantified over a smaller scope than this, because they are introduced explicitly by $\forall$.

The last part of a type that I will introduce here is the 'type generator'. An example of this is List [Int], which is the type of a list of integers. List is a type generator that is followed by the type of the members of the list in square brackets. Type generators and infix type generators can be defined in Ponder, but I will not describe how this is done. The infix type generator '$\times$' is defined in the standard prelude to be the same as Pair. The type of the pair (666, 42) could be written as Pair [Int, Int] or Int $\times$ Int — they are equivalent.

It is also possible to define recursive types in Ponder, subject to certain constraints which ensure that the types in a program can always be checked automatically without further aid from the programmer.

As an example of a function using both quantified types and type generators, consider

```
Letrec append  =  ∀Y. List [Y] a → List [Y] b → List [Y] :
                  If null a
                  Then b
                  Else cons (head a) (append (tail a) b)
                  Fi;
```

The '$\forall$Y.' introduces a type variable Y which can correspond to any type when *append* is used. The rest of the type specification says that *append* takes two bound variables, both List [Y], where Y could be any type. Note that Y must be the same type in both cases. The result is a list of the same type.

Appendix C

Example programs

This appendix gives the Ponder source code of the benchmarks used in chapters four and five.

Nfib

```
Letrec nfib  =  Int n → Int :
                If n < 2
                Then 1
                Else nfib (n − 1) + nfib (n − 2) + 1
                Fi;
```

This function was originally used as a benchmark for implementations of functional languages by Henderson. The integer returned by *nfib* gives the number of calls of *nfib* that were executed to obtain the result. The timings in chapter four are for *nfib* 20, which executes 21891 calls of *nfib*.

Insertion sort

```
Letrec InsertIn  =
 Int n → List[Int]list → List[Int] :
 If null list
 Then n  ::  nil
 Elif n > head list
 Then head list  ::  InsertIn n (tail list)
 Else n  ::  list
 Fi;
```

```
Letrec InSort =
 List[Int]sorted → List[Int]unsorted → List[Int] :
 If null unsorted
 Then sorted
 Else InSort (InsertIn (head unsorted) sorted) (tail unsorted)
 Fi;

Let InsertionSort = InSort nil
```

The operator '::' is an infix *cons*.

To obtain timings for *InsertionSort* it was applied to a list of 100 random numbers. The numbers in the resulting sorted list were then summed and the sum output by the program, to ensure that the list was fully evaluated.

Quicksort

```
Letrec Filter =
 ∀T.(T → Bool) predicate → List[T]list → List[Int] :
 If null list
 Then nil
 Elif predicate (head list)
 Then head list :: (Filter predicate (tail list))
 Else Filter predicate (tail list)
 Fi;

Let GetMedian =
 List[Int]list → Option[List[Int] × List[Int] × List[Int]]
 If list Is Int m, List[Int] rest
 → OptIn (Filter (< m) rest,
          consm(Filter (= m) rest), Filter (> m) rest)
 Else nil
 Fi;

Letrec Quicksort =
 List[Int]list → List[Int] :
 If GetMedian list Is List[Int] l, List[Int] m, List[Int] r
 → Quicksort l @ m @ Quicksort r
 Else nil
 Fi;
```

The operator '@' is an infix *append*.

This program uses 'options' to pass data from *GetMedian* to *Quicksort*. An option is something which is either a data structure or *nil*. *OptIn* is used to make an option containing a data structure, and the If... → ...Else...Fi conditional

construct is used to take options apart. Lists are not built in, but are defined recursively by

$$\mathsf{List[T]} \;=\; \mathsf{Option[T \times List[T]]}$$

GetMedian takes its list argument apart using the conditional construct for options.

QuickSort was applied to 100 random numbers, as with *InsertionSort*, to obtain timings.

Dragon

I will not give the whole Dragon program here, only the function for translating bytes sent to the terminal and the main function for generating the list of graphics commands.

```
Letrec Dragon =
 Int depth → Sex sex → Vector v1 → Vector v2 → Fractal :
 If depth = 0
 Then Draw (v1 + v2) :: nil
 Else Let base1 = (v1 + v2)/2;
      Let base2 = v1 − base1;
      Let next = depth − 1;
      If sex = male
      Then Dragon next male base2 base1 @
           Dragon next female (−base2) base1
      Else Dragon next male base1 base2 @
           Dragon next female base1 (−base2)
      Fi
 Fi
```

Dragon uses a vector arithmetic package, but I will not describe this.

To draw a dragon curve of depth n, which is *size* units high on the screen, *Dragon* is called like this:

$$\mathit{Dragon}\ n\ \mathit{male}\ (\mathit{MakeVector}\ 0\ \mathit{size})\ (\mathit{MakeVector}\ \mathit{size}\ 0)$$

The resulting List[Graphic] is translated into a List[Bytes] by a graphics package, and the individual bytes are then translated by *ByteToBBC* so that they will survive passing through the local area network and the terminal emulator of a BBC micro.

```
Letrec ByteToBBC =
 Int byte → List[Char] :
 If byte ≤ Hex0F
 Then IntToChar (byte + Hex60)
 Elif (byte ≥ Hex60) Or (byte ≤ Hex20)
 Then IntToChar ((byte ShiftRight 4) + Hex70) ::
      IntToChar ((byte & Hex0F) + Hex60) ::
      nil
 Else IntToChar byte :: nil
 Fi;
```

Slow Dragon

Slow Dragon is similar to Dragon, but instead of producing a List[Graphics] 'on the fly' it produces an infinite Fractal from which a dragon curve of any desired depth can be extracted.

```
Letrec Dragon =
 Sex sex → Vector v1 → Vector v2 → List[Graphic] :
 Begin
  Let base1 = (v1 + v2)/2;
  Let base2 = v1 − base1;
  (v1 + v2),
  If sex = male
  Then Dragon male base2 base1 ::
       Dragon female (−base2) base1
  Else Dragon male base1 base2 ::
       Dragon female base1 (−base2)
  Fi
 End
```

The type Fractal is defined recursively

$$\text{Fractal} \;=\; \text{Vector} \times \text{List[Fractal]}$$

To draw a dragon curve of depth n, which is *size* units high on the screen, first make an infinite Fractal dragon curve of the required size, then flatten the lists of Vectors selected from this at the required depth:

```
Let Fractal = Dragon male (MakeVector 0 size) (MakeVector size 0);
map Draw (flatten n Fractal)
```

The result of mapping *Draw* down this flattened list is exactly the same List[Graphic] as that returned by the previous version of *Dragon*.

References

[Abdali 76]
S. Kamal Abdali,
An Abstraction Algorithm for Combinatory Logic,
J. Symbolic Logic, 41(1), March 1976, 222–224.

[Abramsky et al 85]
S. Abramsky, G. L. Burn & C. L. Hankin,
The Theory and Practice of Strictness Analysis of Higher Order Functions,
Department of Computing,
Imperial College London,
Research Report Doc 85/6, April 1985.

[Augustsson 84]
Lennart Augstsson,
A compiler for lazy ML,
Institutionen för Informationsbehandling,
Chalmers Tekniska Högskola,
Göteborg,
1984

[Backus 78]
John Backus,
Can programming be liberated from the von Neumann style?
Comm ACM 21(8), August 1978.

[Bird 84]
R. S. Bird,
Using circular programs to eliminate multiple traversals of data,
Acta Informatica 21, 239–250 (1984).

[Blum 56]
E. K. Blum,
Automatic Digital Encoding System II (ADES II),
NAVORD Report 4209,
Aeroballistic Research Report 326,
U.S. Naval Ordinance Laboratory,
8 February 1956

[Burge 75]
W. H. Burge,
Recursive Programming Techniques,
Addison Wesley 1975.

[Clack & Peyton Jones 85a]
Chris Clack & Simon Peyton Jones,
Generating Parallelism from Strictness Analysis,
Department of Computer Science,
University College London,
Internal Note 1679, February 1985.

[Clack & Peyton Jones 85b]
Chris Clack & Simon Peyton Jones,
Strictness Analysis — a practical approach,
in [Jouannaud 85].

[Dijkstra 68]
E. W. Dijkstra,
The GO TO statement considered harmful,
Comm. ACM 11(3), March 1968.

[Elworthy 85]
David Elworthy,
Implementing a Ponder Cross-compiler for the SKIM processor,
Cambridge University Computer Laboratory,
Diploma Dissertation 1985.

[Fairbairn 85]
Jon Fairbairn,
Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus,
University of Cambridge Computer Laboratory,
Technical Report No. 75, May 1985.

[Friedman & Wise 76]
D. P. Friedman & D. S. Wise,
CONS should not evaluate its arguments,
in [Michaelson & Milner 76], 257–284.

[Henderson & Morris 76]
P. Henderson & J. H. Morris Jr.,
A lazy evaluator,
Conf. Record Third ACM Symp. on Principles of Programming Languages, January 1976.

[Hudak & Goldberg 85]
Paul Hudak & Benjamin Goldberg,
Distributed Execution of Functional Programs Using Serial Combinators,
Yale University,
Department of Computer Science,
1985

[Hudak & Young 85]
Paul Hudak & Jonathan Young,
A Set-Theoretic Characterization of Function Strictness in the Lambda Calculus,
Yale University,
Department of Computer Science,
Research Report YALEU/DCS/RR-391 (April 1985)

[Hughes 82]
John Hughes,
Graph Reduction with Super-Combinators,
Oxford University Programming Research Group,
Technical Monograph PRG-28 (1982)

[Hughes 85a]
John Hughes,
Short talk at Workshop on Functional Programming,
Aspenäs, Sweden,
February 1985.

[Hughes 85b]
John Hughes,
Strictness Detection in Non-Flat Domains,
Institutionen för Informationsbehandling,
Chalmers Tekniska Högskola,
Göteborg,
Draft, August 1985

[Johnsson 83]
Thomas Johnsson,
The G-Machine: An Abstract Machine for Graph Reduction,
Proceedings of SERC Declarative Programming Workshop at UCL,
April 1983

[Jones 84]
Simon B. Jones,
A range of operating systems written in a purely functional style,
University of Stirling, Department of Computing Science,
Technical Report TR. 16 (1984)

[Jouannaud 85]
Jean-Pierre Jouannaud, ed.,
Functional Programming Languages and Computer Architecture,
Lecture Notes in Compute Science 201, Springer-Verlag, 1985.

[Knuth 77]
Donald E. Knuth and Luis Trabb Pardo,
The early development of programming languages,
in [Metropolis et al 80].

[Kieburtz & Napierala 85]
Richard B. Kieburtz & Maria Napierala,
A studied laziness—strictness analysis with stuctured data types,
Oregon Graduate Centre,
Extended Abstract, July 1985.

[Landin 64]
P. J. Landin,
The mechanical evaluation of expressions,
Computer Journal, 6, 1964, 308–320

[Landin 66]
P. J. Landin,
The next 700 programming languages,
Comm. ACM 9(3), March 1966.

[Lovelace 43]
Ada Augusta, Countess of Lovelace,
Sketch of the Analytical Engine Invented by Charles Babbage, by L. F. Menabrea, of Turin, Officer of the Military Engineers, With notes upon the Memoir by the Translator,
1843.

[McCarthy 60]
John McCarthy,
Recursive functions of symbolic expressions and their computation by machine,
Comm. ACM 3(4), April 1960, 185–195.

[Mandelbrot 83]
Benoit B. Mandelbrot,
The fractal geometry of nature,
W. H. Freeman, New York 1983.

[Metropolis et al 80]
Metropolis, Howlett, Rota (Ed.),
A History of Computing in the Twentieth Century,
Academic press 1980.

[Metropolis & Worlton 80]
N. Metropolis & J. Worlton,
A Trilogy on Errors in the History of Computing,
Annals of the History of Computing, 2(1), January 1980, 49–59.

[Michaelson & Milner 76]
S. Michaelson & R. Milner (Ed.),
Automata, Languages and Programming,
Edinburgh, 1976.

[Morris 81]
James H. Morris,
Real programming in functional languages,
Xerox Palo Alto Research Center,
Report CSL-81-11, July 1981.

[Mycroft 81]
Alan Mycroft,
Abstract Interpretation and Optimising Transformations for Applicative Programs,
PhD Thesis, University of Edinburgh, December 1981.

[Naur & Randell 68]
Peter Naur & Brian Randell (Ed.),
Software Engineering,
Report on a conference sponsored by the NATO Science Committee,
Garmisch, Germany, 1968.

[Park 85]
David Park,
Private communication, 1985

[Peyton Jones 85]
Simon Peyton Jones,
Yacc in Sasl—an Exercise in Functional Programming,
Software Practice and Experience Vol 15(8), 807–820, August 1985.

[Raphael 66]
Bertram Raphael,
The structure of programming languages,
Comm. ACM 9(3), March 1966, 155–6.

[Richards 84]
Hamilton Richards Jr.,
A small but realistic experiment in SASL programming,
Burroughs Corp., Austin Research Center,
February 1984.

[Richards & Whitby-Strevens 80]
Martin Richards & Colin Whitby-Strevens,
BCPL, the language and its compiler,
Cambridge University Press, 1980.

[Rosser 82]
J. Barkley Rosser,
Highlights of the history of the lambda-calculus,
Conf. record 1982 Symp. Lisp & Functional Programming.

[Schmidt 82]
David A. Schmidt,
Denotational Semantics as a Programming Language,
University of Edinburgh Department of Computer Science,
Internal Report CSR-100-82, January 1982.

[Stoye 85]
William Stoye,
Implementation of Functional Languages using Custom Hardware,
University of Cambridge Computer Laboratory,
Technical Report No. 81, December 1985

[Tillotson 85]
Mark Tillotson,
Introduction to the Functional Programming Language "Ponder",
University of Cambridge Computer Laboratory,
Technical Report No. 65, May 1985.

[Turner 79]
D. A. Turner,
A New Implementation Technique for Applicative Languages,
Software Practice and Experience, Vol 9, 31–49 (1979).

[Turner 85]
David Turner,
Private communication, 1985.

[Vuillemin 74]
J. Vuillemin,
Correct and optimal implementation of recursion in a simple programming languages,
J. Comp. Sys. Sci. 9 (1974), 332–354.

[Wadler 85]
Phil Wadler,
Strictness Analysis on Non-Flat Domains,
Programming Research Group, Oxford University,
November 1985.

[Wadsworth 71]
C. P. Wadsworth,
Semantics and Pragmatics of the Lambda-Calculus,
PhD. Thesis, University of Oxford, 1971.