Non-strict languages — programming and implementation

S. C. Wray[1] & J. Fairbairn[2]

February 25, 1994

Abstract

Non-strict evaluation improves the expressive power of functional languages at the expense of an apparent loss of efficiency. In this paper we give examples of this expressive power, taking as an example an interactive functional program and describing the programming techniques depending on non-strict evaluation which improved its design. Implementation methods for non-strict languages have delivered poor performance precisely when such programming techniques have been used. This need not be the case, however, and in the second part of the paper we describe Tim, a method of implementing non-strict languages for which the penalty for using lazy evaluation is very small.

1 Introduction

Effort in the functional programming community is today divided into two main activities: making efficient implementations of functional languages and exploiting the expressive power of these languages by writing elegant programs. To a large extent these activities are carried out by separate groups of people, and there is a danger that the claim "we can implement non-strict languages efficiently" from the former group applies chiefly to the kind of programs that the latter group don't write.

In the first part of the paper we discuss programming techniques which make essential use of non-strictness, taking as an example *Nas*, an interactive spreadsheet

[1]Olivetti Research Limited, 24a Trumpington Street, Cambridge

[2]University of Cambridge Computer Laboratory, Pembroke Street, Cambridge

that was written in Ponder [Fairbairn 83, Fairbairn 85, Tillotson 85]. *Nas* makes significant use of the non-strict semantics of Ponder, so it serves as a good illustration of functional programming techniques. Most descriptions of functional programming techniques concentrate on small examples. We shall describe *Nas* in some detail, because at around 2 000 lines it exploits techniques that help control the structure of large interactive programs.

The second part of the paper is concerned with Tim, a non-interpretive implementation technique for non-strict functional languages. The authors' previous experience of implementing functional languages was with graph reducers, which interpret a graph representation of the program. Simple combinator graph reducers [Turner 79] spend a considerable time interpreting the graph, and although a supercombinator version can be ten times faster [Hughes 82, Augustsson 84, Fairbairn&Wray 86], the main cost is still interpretation.

Implementations of supercombinator graph reduction can be quite successful in producing code for strict functions on basic values (integer arithmetic for example). For functions over basic values that are detectably strict in all arguments the stack machine code produced by our previous compiler is optimal. However, a relatively small proportion of the time taken by typical data-processing programs is spent on arithmetic, so we wanted to find a way of reducing the cost of the other kinds of processing (list and function manipulation in particular).

In producing the Tim design we aimed to remedy the most obvious flaw in these earlier implementation techniques: it was rather expensive to use functions, because it was expensive to suspend evaluation. In a language which encourages and relies heavily on the use of higher order functions this is not satisfactory.

In non-strict functional languages it is important not to evaluate expressions that are not needed. At best this is a waste of effort. At worst it can cause the program to go into an infinite loop or to crash in some other way. When an expression is passed as an argument to a function it must thus be passed as a suspension. The function can then use this to create the value of the expression if it is needed, pass it to another function or simply discard it. If we knew in advance how an argument was going to be used we could generate perfect code without suspensions. If its value will certainly be used we could avoid the expense of building a suspension by calculating the value immediately. Otherwise the argument will not be required at all, in which case there is again no need for the suspension. It is in general impossible to tell in advance which of these alternatives should be chosen, although the techniques of strictness analysis give useful information [Burn,Hankin&Abramsky 85, Wadler&Hughes 87].

The approach in Tim is different: rather than attempting to eliminate suspensions, it makes them very cheap to create and use. Tim avoids much of the book-keeping involved in building the suspensions in graph reducers by postponing it until the value of the suspension is required. By this time it is often unnecessary to do this book-keeping at all. We believe that this makes Tim fundamentally faster than graph reducers. Two earlier abstract machines with a similar style to Tim are the Functional Abstract Machine [Cardelli 84], and the Categorical Abstract Machine [Cousineau et al 87], both designed primarily to support applicative order evaluation. The Spineless Tagless G-Machine [Peyton Jones 88] incorporates some of the ideas behind Tim into a G-Machine-like framework.

2 Programs

In this section we describe the interactive functional program *Nas*. Before *Nas* was written we had produced a library of graphical interface functions and small interactive programs to experiment with graphical and mouse driven interfaces. *Nas* was written as a larger experiment using these interfaces, not as an exercise in user friendly programming!

Nas is based on ideas from spreadsheets; the display shows cells containing values which may have been derived from the values of other cells. The cells in *Nas* do not lie on a rectangular grid, as in conventional spreadsheets, but can be positioned anywhere on the screen, even overlapping each other like pieces of paper. A typical screen is shown in figure 1.

At the top of the display are 'screen buttons' which are used to give commands

Figure 1: A typical *Nas* screen

to *Nas*. The screen buttons behave as simulated push-buttons. To push one, the mouse cursor is placed over the screen button, and the button on the mouse is clicked. When pushed, each screen button causes one of the following actions to be performed, some of which require further selections of cells using the mouse. Screen buttons light up until their associated action is complete.

- Finish exits from *Nas*.
- Delete removes a cell.
- Create makes a new cell.
- Move moves a cell from one position on the screen to another.
- Re-draw clears the screen and draws all the screen buttons and cells again.
- Name changes the name of a cell.
- Value changes the formula of a cell, from which its value is calculated.

- Inspect displays the formula of a cell.

- Tick recalculates all the displayed values of cells as described below.

If the value of a cell is changed by the user, the values of other cells that depend on it change immediately. In the recalculation performed by Tick, cells may again depend on the displayed values of cells, but also on their 'last' values, which is to say their values before the last Tick. For example, if the 'last' value of cell Count was 41 and its formula was Last Count + 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.

When a cell is created its value is not a number, but the special value Undefined. There is an operator to give a default value to a formula if it depends on the values of cells that are undefined.

The most complex spreadsheets that have been constructed using *Nas* are a simulation of a flip-flop constructed from NOR-gates and a Newton-Raphson square root finder using the 'history' mechanism provided by the Last operator.

How it works

Figure 2 is a simplified picture of the main components of *Nas*.

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

Figure 2: The internal structure of *Nas*

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. The filter strips out button releases 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. The parser converts character strings into formulæ which will become the contents of cells. The selector decides whether a screen 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 the cells, which the main program alters and passes to recursive calls of itself. The 'kernel' is the part of the 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 display so that the cell's previous value is erased and its new value written.

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 it is shown like that in figure 2 for clarity. In fact the output from the program is a list of 'file actions'. In the same way that the output list of graphics contains high level commands to drive a display, 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 contain another string — for instance the 'append' file action also contains the text to be appended to a file. The back-up file is written using this mechanism, and the list of bytes sent to the terminal is sent in an 'append' to the standard output.

3 Programming Techniques

Like someone learning their first programming language, the novice functional programmer has an initial feeling of paralysis when writing a substantial program for the first time. Irrespective of how competent the programmer is in an imperative language this is likely to be the case, because most of the programming techniques we have been taught 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 novice functional programmer is made doubly hard. Not only must he master the unfamiliar programming style of a new language, he must also reformulate familiar algorithms in order to express them in the new language.

It took about a month to write and type-check *Nas*, much 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 the first time largely because it was written in the typed language Ponder. Peyton Jones has written substantial programs in SASL, an untyped functional language, and his experience supports this view [Peyton Jones 85].

Part of the convenience of using a functional programming language was that large parts of the earlier experimental programs could be reused when writing *Nas*. Although some reuse occurs when writing imperative programs, it is almost always by taking the old code and hacking it. In a functional language it is much more common to reuse exactly the same function as before.

In the remainder of this section we describe two programming techniques, 'stream processing functions' and 'almost circular definition', which make essential use of non-strict evaluation. These techniques have been invented independently in a number of places. They were invaluable when writing *Nas*.

Stream processing functions

The first technique is the use of stream processing functions (which will be abbreviated to SPFs [Jones 84, Wray 86]). A stream is a non-strict list (in early functional languages list construction did not have to be strict even when all other functions were [Friedman&Wise 76]).

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

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

This type describes a function that takes a list of some objects of type Input, and returns a list of objects with type Output. The lists are implicitly non-strict. An example of a common SPF is a function that upper-cases the characters in its input list and returns this as its output list:

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

Where *Upper* is the function to upper-case a single character, and *map* is the usual pointwise application over lists. *UpperCasingSPF* has the type

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

Notice that this is exactly the definition one would use for the upper-casing function on lists in general. This is one of the strengths of functional programming — the same techniques can readily be reused in different circumstances.

The stately SPFs 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 non-strict functional programs. From the List [Input] and the State it will construct a List [Output] using the first few elements of the input. It returns this output list, 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 along with the new state, and in this way the entire input list can be consumed incrementally. The best way to write such functions is to use recursion as little as possible, using higher order functions to encapsulate the recursion when it is necessary.

Here is a higher order function to pipe the output of one SPF into the input of another:

$$
\begin{aligned}
\mathsf{Let}\ \ \mathit{Pipe}\ =\ \ & \mathit{SPF}_1 \mapsto \mathit{SPF}_2 \mapsto \mathit{state} \mapsto \mathit{in} \mapsto \\
& \mathsf{Begin} \\
& \quad \mathsf{Let}\ \mathit{out}_1,\ \mathit{stub}_1,\ \mathit{state}_1 \ = \ \mathit{SPF}_1\ \mathit{state}\ \mathit{in}; \\
& \quad \mathsf{Let}\ \mathit{out}_2,\ \mathit{stub}_2,\ \mathit{state}_2 \ = \ \mathit{SPF}_2\ \mathit{state}_1\ \mathit{stub}_1; \\
& \quad (\mathit{out}_1 \mathbin{+\!\!+} \mathit{out}_2),\ \mathit{stub}_2,\ \mathit{state}_2 \\
& \mathsf{End}
\end{aligned}
$$

The notation $\mapsto$ is used to separate a bound variable from the body of a function and $+\!\!+$ is used for infix append.

Thus if A and B are stately SPFs, *Pipe* A B will be a new stately SPF which is the 'composition' of these two functions. As an example of a higher order function encapsulating recursion, here is *Repeat*:

$$
\begin{aligned}
\mathsf{Letrec}\ \mathit{Repeat}\ =\ \ & \mathit{SPF} \mapsto \mathit{state} \mapsto \mathit{in} \mapsto \\
& \mathsf{Begin} \\
& \quad \mathsf{Let}\ \mathit{out}_1,\ \mathit{stub}_1,\ \mathit{state}_1 \ = \ \mathit{SPF}\ \mathit{state}\ \mathit{in}; \\
& \quad \mathsf{If}\ \mathit{null}\ \mathit{out}_1 \\
& \quad \mathsf{Then}\ \mathit{out}_1,\ \mathit{stub}_1,\ \mathit{state}_1 \\
& \quad \mathsf{Else}\ \ \mathsf{Let}\ \mathit{out}_2,\ \mathit{stub}_2,\ \mathit{state}_2 \ = \ \mathit{Repeat}\ \mathit{SPF}\ \mathit{state}_1\ \mathit{stub}_1; \\
& \quad \qquad (\mathit{out}_1 \mathbin{+\!\!+} \mathit{out}_2),\ \mathit{stub}_2,\ \mathit{state}_2 \\
& \quad \mathsf{Fi} \\
& \mathsf{End}
\end{aligned}
$$

Repeat A will be a stately SPF that returns all the output that A would return when applied successively to the results of its previous invocation, until A returns a null list. It is interesting to note, as in [Schmidt 82], that functions for combining SPFs are very much like the functions used in denotational semantics when describing the meaning of imperative programs. Schmidt remarks in his paper that by using denotational semantics rather than an imperative language one can define combining forms (such as *Pipe* and *Repeat*) suitable for particular problems, rather then having to accept the built-in control constructs of a particular imperative language.

Although these higher order functions require some thought, they remove the need to go through the mental contortions of writing recursive stately 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, it is likely that the same kind of construct will be needed again, in which case the higher order function can be re-used.

Almost circular definition

The second programming technique, 'almost circular definition,' is somewhat more bizarre. The technique is a form of recursive definition — but recursive definition of data structures rather than of functions. This works in non-strict 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 natural numbers.

$$\begin{aligned} \mathsf{Let}\ AddOne &= map\,(Add\,1)\ ; \\ \mathsf{Letrec}\ naturals &= cons\,1\ (AddOne\ naturals) \end{aligned}$$

The first element of *naturals* can be created immediately — it is 1. To calculate the second element, only the first element need 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. Actually, 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.

The core of *Nas* is the use of *transition_function* to produce a new state from the old state. It is an 'almost circular definition' because the current value of a cell may depend on the current values of other cells as well as their old values. It is surprisingly short:

$$\mathsf{Letrec}\ new\_state \ = \ transition\_function \quad old\_state \quad new\_state$$

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 non-strict evaluation.

Figure 3: Flip-flop

A further example of this kind of definition is found in another program, a simple gate-level logic simulator. By representing signals as streams of logic levels and providing a function *nand* that takes two signals, delays them and returns their logical nand, a flip-flop (figure 3) can be defined as follows

$$\begin{array}{lll}\mathsf{Let}\ flip\_flop & = & in_1 \mapsto in_2 \mapsto \\ & & \mathsf{Begin} \\ & & \quad \mathsf{Letrec}\ out_1 \ = \ nand\ in_1\ out_2 \\ & & \quad \mathsf{And}\ out_2 \ = \ nand\ in_2\ out_1; \\ & & \quad out_1, out_2 \\ & & \mathsf{End}\end{array}$$

This technique of using the answer before it is all there can only be used in a non-strict 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. Further examples of almost circular definition may be found in [Bird 84] and [Sijtsma 88].

Bird makes the point, which is true of both the techniques we have described, that care must be taken to ensure that the program is safe, i.e. that it will terminate.

4 Implementation techniques

The programming techniques described above depend in an essential way on the non-strict semantics of the language. Unfortunately, earlier implementations of non-strict functional languages rely on conversion to a strict reduction order for their efficiency. Usually this is done by using a static analysis of the program to determine where this can be done safely. However, such an analysis cannot eliminate all the non-strict applications from programs that use the above techniques.

This mismatch of technology led the authors to search for an evaluation mechanism that performed well without the use of static analysis. The major expense in conventional graph reducers is in manipulating the graph. If an expression is passed to a non-strict function, it is built as a graph, incurring a cost proportional to the size of the expression even if it is never used. A further overhead is incurred if the expression is needed, because the graph must then be interpreted.

Tim has smaller overheads because it uses closures, not graphs. The cost of passing an unevaluated argument is thus small compared to graph reducers. In less sophisticated closure reducers the cost of looking up variables in the environment is large compared with graph reducers. This accounts for the apparent superiority of simple graph reducers over simple closure reducers [Turner 79]. Tim depends on a compile time environment analysis technique (λ-lifting) to make environment handling cheap. The technique of λ-lifting was originally developed to improve graph reducers by producing a transformed source program consisting of combinator definitions [Johnsson 85, Hughes 82]. The term 'combinator reducer' is often (confusingly) used to mean 'graph reducer.' Combinators are simply functions that refer only to their arguments or to constants, so environment handling for them is very easy. It must be remembered that Tim is a closure reducer, not a graph reducer, even though it too uses a combinator program as its starting point.

It is important that an implementation technique for non-strict languages is suitable for adding lazy evaluation (recording the value of an expression when it is computed, for future use). In a later section we describe how Tim can be enhanced in this way. First we describe the machine in terms of a series of simple syntactic transformations of the λ-representations of combinators, which show that Tim code is really just a flattened-out version of the original combinators.

It is assumed that the original program has been λ-lifted [Peyton Jones 87] into combinator definitions before the transformations described in this section are applied. We will introduce the transformations gradually so that at each stage the resulting program can easily be seen to be equivalent to its previous form.

The program is assumed to be in the form

$$\begin{array}{lcl} \chi_1 & \equiv & f_1 \\ & \vdots & \\ \chi_n & \equiv & f_n \\ e_{main} & & \end{array}$$

Where the expressions $f_1 \ldots f_n$ and e_{main} may refer to any of $\chi_1 \ \ldots \ \chi_n$. The value of the whole program is given by e_{main}. Expressions have the following syntax (we will omit parentheses according to the usual convention):

$$\begin{array}{lcl} f & = & \lambda a_1 \ \ldots \ \lambda a_n .\ e \\ e & = & (e_1 \ e_2) \\ & | & a_i \\ & | & \chi_i \end{array}$$

The first step of the transformation is to introduce tuples to hold the environment. In a combinator program the environment of an expression contains only the bound variables of its enclosing combinator. In what follows, this environment is a tuple of the values of these bound variables, denoted ρ, and χ stands for any combinator name. The syntactic function C translates combinator definitions and T translates expressions into functions over environments.

Definitions

$$
\begin{aligned}
C[\![\chi \equiv f]\!] &= \chi \equiv T[\![f]\!] \\
C[\![e_{main}]\!] &= T[\![e_{main}]\!]\,\emptyset
\end{aligned}
$$

Where $\emptyset$ denotes the empty environment

Expressions

$$
\begin{aligned}
T[\![\lambda a_1 \ldots \lambda a_n \,.\, e]\!] &= \lambda\rho.\, \lambda a_1. \ldots \lambda a_n. T[\![e]\!]\langle a_1, \ldots, a_n\rangle \\
T[\![e_1\; e_2]\!] &= \lambda\rho.\, (T[\![e_1]\!]\,\rho)\ (T[\![e_2]\!]\,\rho) \\
T[\![a_i]\!] &= \pi_i \\
& \quad \text{where } \pi_i\,\langle a_1, \ldots, a_i, \ldots, a_n\rangle \;=\; a_i \\
T[\![\chi]\!] &= \chi
\end{aligned}
$$

In the notation used above λ is part of the syntax being transformed, so that the translation $T[\![a_n\; a_m]\!]\,\rho$ is $(\lambda\rho'.\, \pi_n\,\rho'\,(\pi_m\,\rho'))\,\rho$, rather than $\pi_n\,\rho\,(\pi_m\,\rho)$. Observe that if the $\lambda\rho$ terms introduced by the translation are reduced out, we get back to the original program. These rules bear a striking resemblance to the translation of ordinary λ-expressions into combinator expressions using multiple abstraction [Abdali 76].

The next step is to simplify the choice of order of evaluation by introducing explicit continuations giving the next thing to do. First we need some definitions of functions that manipulate continuations. Each of these functions takes a continuation and some other arguments, and passes control to the continuation by applying it to some combination of the other arguments:

$$
\begin{array}{lll}
(\mathsf{Take}\ n) & c\ \rho\ a_1 \ldots a_n & \Rightarrow \quad c\ \langle a_1, \ldots, a_n\rangle \\
(\mathsf{Push}\ e) & c\ \rho & \Rightarrow \quad c\ \rho\ (e\ \rho) \\
(\mathsf{Enter}) & c\ \rho & \Rightarrow \quad c\ \rho
\end{array}
$$

where e is a compiled expression, c is a continuation (which will be another compiled expression), and ρ is an environment as before. We also introduce some functions to manipulate environments:

$$\begin{array}{lcl} \mathsf{Arg}\, i\ \rho & \Rightarrow & \pi_i\, \rho \\ \mathsf{Comb}\ \chi\ \rho & \Rightarrow & \chi\ \rho \\ \mathsf{Label}\ e\ \rho & \Rightarrow & e\ \rho \end{array}$$

Here are the transformations that introduce the continuation functions:

$$\begin{array}{lcl} T[\![\lambda a_1\ \ldots\ a_n.\ e]\!] & = & \mathsf{Take}\, n\ (T[\![e]\!]) \\ T[\![e_1\ e_2]\!] & = & \mathsf{Push}\ (E[\![e_2]\!])\ (T[\![e_1]\!]) \\ T[\![a_i]\!] & = & \mathsf{Enter}\ (E[\![a_i]\!]) \\ T[\![\chi]\!] & = & \mathsf{Enter}\ (E[\![\chi]\!]) \end{array}$$

$$\begin{array}{lcl} E[\![e_1\ e_2]\!] & = & \mathsf{Label}\ (T[\![e_1\ e_2]\!]) \\ E[\![a_i]\!] & = & \mathsf{Arg}\, i \\ E[\![\chi]\!] & = & \mathsf{Comb}\ \chi \end{array}$$

For example, the combinator $I \equiv \lambda x.\ x$ is transformed like this:

$$\begin{array}{lrcl} & C[\![I & \equiv & \lambda x.\ x]\!] \\ = & I & \equiv & T[\![\lambda x.\ x]\!] \\ & & \equiv & \mathsf{Take}\, 1\ (T[\![x]\!]) \\ & & \equiv & \mathsf{Take}\, 1\ (\mathsf{Enter}\ (E[\![x]\!])) \\ & & \equiv & \mathsf{Take}\, 1\ (\mathsf{Enter}\ (\mathsf{Arg}\, 1)) \end{array}$$

Similarly the combinator $S\ \equiv\ \lambda a\ \lambda b\ \lambda c.\ a\ c\ (b\ c)$ transforms into

$$\begin{array}{rcll} S & \equiv & \mathsf{Take}\, 3 & \\ & & (\mathsf{Push} & (\mathsf{Label}\ l_1) \\ & & (\mathsf{Push} & (\mathsf{Arg}\, 3) \\ & & (\mathsf{Enter} & (\mathsf{Arg}\, 1)))) \end{array}$$

$$\begin{array}{rcll} \text{where } l_1 & = & \mathsf{Push} & (\mathsf{Arg}\, 3) \\ & & (\mathsf{Enter} & (\mathsf{Arg}\, 2)) \end{array}$$

Notice that all combinators immediately discard their environment argument, since they must start with a Take. This allows us to replace the definition of Comb with

$$\mathsf{Comb}\ \chi\ \rho \Rightarrow \chi\ \emptyset$$

Finally we introduce closures consisting of $\langle \text{code}, \text{environment} \rangle$ pairs. This is done by giving a new definition of Enter and new versions of the environment manipulating functions:

$$\begin{array}{lcl}
\mathsf{Take}\, n\ c\ \rho\ a_1\ \ldots\ a_n & \Rightarrow & c\ \langle a_1,\ \ldots,\ a_n \rangle \\
\mathsf{Push}\ e\ c\ \rho & \Rightarrow & c\ \rho\ \langle e', \rho' \rangle \\
 & & \text{when } e\ \rho \Rightarrow \langle e', \rho' \rangle \\
\mathsf{Enter}\ c\ \rho & \Rightarrow & c'\ \rho' \\
 & & \text{when } c\ \rho \Rightarrow \langle c', \rho' \rangle
\end{array}$$

$$\begin{array}{lcl}
\mathsf{Arg}\, i\ \rho & \Rightarrow & \pi_i\, \rho \\
\mathsf{Comb}\ \chi\ \rho & \Rightarrow & \langle \chi, \emptyset \rangle \\
\mathsf{Label}\ e\ \rho & \Rightarrow & \langle e, \rho \rangle
\end{array}$$

Evaluation now proceeds by applying the definitions of Take, Push and Enter as rewrite rules, making reduction take place in three phases:

- Take puts the arguments to the function into an environment for use later.
- Push puts continuations where they can be accessed by subsequently called functions.
- Enter transfers control to another continuation which specifies its own environment.

The name of this abstract machine comes from these three instructions: *Tim* = *T*hree *I*nstruction *M*achine.

5 Practical implementation and laziness

This section describes how to implement the normal order reduction machine described above. While it would be possible to implement the Tim instructions as rewrite rules, they are sufficiently simple that they can each be represented by a small number of machine instructions. In describing the implementation, we will present the basic representation on conventional hardware, and then describe some optimisations, including laziness. An important point is that the basic architecture is simple; the optimisations can each be considered separately as optimisations, not as alterations to the machine, in much the same way as one would consider using peephole optimisation to generate real code from any other abstract machine. The only essential optimisation is the introduction of laziness, but even this can be dispensed with when it makes no improvement.

The state of Tim can be divided into a head expression consisting of the current function and current environment, and the arguments to which it is applied. The current function will consist of a nested application of Tim instructions, and can easily be represented as a conventional instruction stream. The current environment is always a tuple of $\langle \text{code}, \text{environment} \rangle$ pairs, and can be represented as a frame of closures. The arguments are also closures, but the number of them changes as reduction proceeds, so they must be held on a stack. Finally there must be space in which to store frames as they are created, which will need a heap. This gives the abstract architecture shown in figure 4.

Figure 4: Tim on hardware

The environment manipulating functions Arg, Comb and Label can be regarded as addressing modes. Arg n refers to the n^{th} argument in the current frame, Comb χ is the address of the instruction stream for χ paired with an empty environment, and Label e produces a closure from the code for e and the current frame.

The instructions Push and Enter are now trivial, and can be implemented as one or two machine instructions. Push *object* transfers the *object* onto the top of the stack (figure 5) and proceeds with the next instruction, and Enter *object* copies the frame part of *object* into the current frame and proceeds with the code part of *object*.

Figure 5: Executing a Push instruction

The Take instruction is a little more complicated, in that Take n must transfer n objects from the argument stack into the heap, and set the current frame to point to them before proceeding with the next instruction (figure 6).

Figure 6: Executing a Take n instruction

5.1 Making it lazy

Laziness consists of remembering the reduced value of every shared expression the first time it is reduced, so that subsequent accesses do not recompute it. In a graph reducer this is achieved by overwriting nodes in the graph. Tim is designed to avoid the use of this kind of graph, so updating must be handled differently. We shall first take a closer look at the circumstances in which updating is necessary.

What must be updated? A value could only be recomputed if it were accessed more than once. In the supercombinator representation of a program the only way of accessing a value repeatedly is to use a variable (a lambda bound argument to a combinator). An expression can only become shared by being passed to a combinator in a variable that occurs more than once in the body of that combinator. The values of the variables of a particular invocation of a combinator are all held in a frame, so the places to be updated are those entries in the frame that correspond to shared variables.

When must the updates occur? $\mathsf{Enter}\ (\mathsf{Arg}\ n)$ initiates the reduction of a variable, but in the machine as it stands there is no way to tell when this reduction is complete. If we were to reduce this expression on its own, reduction would terminate when no more head reductions could be performed. It would then be of the form $\chi\, e_1\, \ldots\, e_i$ where χ is some combinator and i is less than the number of arguments of χ. There will, however, be sufficient arguments for χ on the stack when the expression is reduced via a variable. The sequence of reductions leading up to an update must go something like this:

$$\begin{array}{ll} \mathsf{Enter}\ (\mathsf{Arg}\ n)\, \rho\, e_j\, \ldots\, e_k & \\ \vdots & \text{various reductions} \\ \mathsf{Enter}\ (\mathsf{comb}\ \chi)\, \rho'\, e_1\, \ldots\, e_i\, e_j\, \ldots\, e_k & \end{array}$$

Where χ takes more than i arguments, so the original $\mathsf{Arg}\ n$ has reduced to $(\chi\, e_1\, \ldots\, e_i)$. In the normal order version of Tim, reduction will proceed without updating argument n of the original combinator, so we must step in at this point to do the update before continuing with the Take instruction in χ, which will gobble up some of the values $e_j \ldots e_k$ from the stack. This requires us to have preserved a certain amount of information in addition to that required by the normal order version of the machine:

- The address of the argument to be updated (that is to say, the address of location n in the frame of the combinator invocation that did the original Enter (Arg n)).
- The state of the stack before that argument was entered.
- The same information for any other updates that have not yet been performed.

One way to represent this is by keeping a list of pairs of pointers into the stack and their corresponding argument addresses. We will refer to these pairs as 'markers.' Whenever an argument is entered, a new pair is put in the list, consisting of the current value of the stack pointer and the address of the argument. The marker at the head of the list is inspected during Take instructions. If a Take finds that it requires values from the stack after this marker, it is interrupted while the appropriate argument is updated with a representation of its reduced form.

This mechanism makes sure that shared arguments in frames are updated with their normal forms as soon as they are available, but it is no use unless subsequent accesses to the value pick up the reduced form. Unfortunately it is not possible in general to determine the order in which accesses to arguments will occur. The code for a combinator may access a variable by pushing it in one place and by entering it in another, and the Enter will not necessarily occur before the Push. So the Push instruction cannot just copy the value of the argument onto the stack as in the normal order version. Instead Push (Arg n) becomes

Push (Label (Enter (Arg$n$)))

so that all references to shared arguments are now via the Enter instruction.

Two observations are worth making here. If static analysis of the program determines that an argument is never shared or that the argument will have been entered (and hence updated) before it is pushed, Push instructions that copy the argument may be used as before. Secondly, it is useful to avoid marking the stack for arguments that are already in normal form, because they cannot be reduced further and interruption of Take instructions to update these arguments is unnecessary. This can be achieved by use of 'deferred marking' as described later.

Interrupting the Take instruction.

In the lazy version of the machine, the Take instruction at the beginning of a combinator must check to see whether there are enough arguments on the stack before the first marker. This need only involve comparing two registers. If an update is necessary, the argument indicated by the mark will be updated with a representation of the combinator applied to the values on the stack before the mark (figure 7).

Figure 7: Interrupting a Take n instruction

One representation of $\chi\, e_1 \ldots e_i$ can be obtained by observing that when executing the code of the combinator, $e_1 \ldots e_i$ will be the arguments $a_1 \ldots a_i$ in the current frame, so $\chi\, e_1 \ldots e_i$ can be represented as $\chi\, a_1 \ldots a_i$. Now

$$\begin{aligned} T[\![\chi\, a_1 \ldots a_i]\!] = \;& \mathsf{Push}\ (\mathsf{Arg}\ \mathrm{i}) \\ & (\mathsf{Push}\ (\mathsf{Arg}\ \mathrm{i\text{-}1}) \\ & \vdots \\ & (\mathsf{Push}\ (\mathsf{Arg}\ 1) \\ & (\mathsf{Enter}\ (\mathsf{Comb}\ \chi))) \ldots) \end{aligned}$$

for which the environment will be the frame created by the interrupted Take. Since the representation uses the same frame, the Take can create it before testing for marks.

If a Take is interrupted several times (because the stack is marked in several places before it has enough arguments) all the updates must be performed before proceeding with the next instruction. The frame constructed by the Take is shared between all the suspensions.

An important optimisation can be performed here. If the above code is entered with a mark on top of the stack, the effect will be to put the arguments on the stack, enter the combinator, interrupt the Take and create an identical value for the new update. It is therefore sensible to add a test at the beginning of the code that updates any arguments pointed to by marks at the top of the stack with itself. Similarly it is worthwhile adding a special case so that if a mark is on the top of the stack at a Take the argument is overwritten directly with the combinator.

One might prefer to avoid having a different piece of code for every combinator, so an alternative is to put the combinator into the frame and use code like this:

< test for marks and do update if necessary >
Push (Arg i+1)
(Push (Arg i)
⋮
(Push (Arg 2)
(Enter (Arg 1))) ...)

Deferred marking

When a Take encounters a mark, a significant overhead is involved, so it is worthwhile reducing the number of marks created. To some extent this can be done by static analysis, but another way of doing it is to allow an object that is being entered to decide whether a mark is needed. In this scheme an Enter does not mark the stack itself, instead it puts the address of the argument in a register, so that the object it is entering can mark the stack if necessary. For example, if the object was in normal form it would be unnecessary for it to mark the stack. All labels in the program must now be capable of putting a mark on the stack if necessary. Rather than having each label test some condition, it is better to use two entry points for each label: one that might put a mark on the stack, and one that never does. An Enter for a shared argument uses the entry point that might mark the stack; an Enter for an unshared argument uses the other entry point. Labels that represent objects in normal form (such as a combinator or a fully reduced shared object) have both entry points the same, so that they never mark the stack.

5.2 Other Optimisations

The implementation presented above still involves many redundant data transfers (pushing things onto the stack and popping them off into the heap only to push them back onto the stack). Some optimisations that are particularly useful on this machine, such as the use of sharing analysis, are described in [Fairbairn&Wray 87]. In generating target code for a particular real machine one would also expect to get large improvements from traditional compiler techniques (appropriate register allocation, peephole optimisation, and so on).

Making use of Strictness Analysis

It is worth mentioning strictness analysis in the context of Tim, since its application is slightly different from other architectures. Because Tim is very good at handling suspended representations of functions there is little or no benefit in evaluating functional arguments before they are passed. The benefit of strictness analysis is, however, still there for objects of ground types such as numbers. In principle once an expression over ground types has been discovered to be strict, the whole thing can be code-generated in a more conventional style. Since we have been concentrating on the functional aspects, we have not investigated this in detail, but it is clear that generating strict expressions differently would increase the basic block size and hence improve performance.

5.3 Run-time support

All values in Tim are represented as functions (including numbers and pairs — see [Fairbairn&Wray 87]), and this uniformity allows one to play interesting tricks with interfaces to an imperative environment.

Interface with host operating system

An output stream can be represented as either $\lambda f.\, \lambda n.\, n$ for the empty stream or $\lambda f.\, \lambda n.\, f\ ch\ rest$ for the stream $(ch{::}rest)$. To print a stream it is applied to two procedures *Printhead* and *Finish*. *Finish* just returns control to the operating system. *Printhead* takes two arguments, prints the first and recursively prints the second.

An input stream can be implemented as a suspension of a procedure that when called reads a character from the input and replaces itself with a list containing the character and a further suspension to read the rest of the input.

Memory Management

A mechanism similar to that used for input streams can provide a virtual memory facility without special hardware. The garbage collector can copy aged objects into secondary store and replace them with functions that read them back in. The only way of accessing an object is to enter it, so this function will be invoked automatically if the object is ever needed.

Some care must be taken in implementing garbage collection for this machine; when a label is pushed, it is pushed with a pointer to the current frame. If the garbage collector were to treat frames as atomic, such pushed labels would result in the retention of the whole frame, and everything attached to it. It will often be the case that the code at the label refers to only a few of the arguments in the frame, so retaining everything in it could result in unexpected consumption of space. The solution is to store a bit pattern with each label, indicating which arguments it needs. The garbage collector can then use this pattern to decide which entries in the frame must be kept.

6 Conclusions

People want to program in non-strict languages because of the expressive power that they bring. It is therefore important that implementation techniques for non-strict languages should deliver good performance for clean and elegant programs that exploit non-strictness. There was in the past a tendency for for implementations to be judged on their performance for unusually strict benchmarks. For such benchmarks it is not hard to produce very fast code, because they do not make essential use of non-strictness. Compile-time techniques can remove non-strictness from these simple benchmarks, so their speed depends solely on how well traditional compilation techniques (data-flow analysis analysis, register allocation etc) have been applied. However, where non-strictness is not removed by compile-time analysis, its cost will dominate the performance.

In this paper we have not addressed such unusually strict programs or implementation techniques suitable for them. While it is important to use traditional compilation techniques wherever they can be applied, it is also necessary to implement non-strictness efficiently. We have concentrated on the kind of program where it is not possible to remove non-strictness with current compile time analyses. Program transformation systems may eventually be able to remove dependence on non-strict evaluation but this cannot yet be done in a sufficiently automatic manner to be included in a general purpose compiler.

Tim implements non-strict evaluation more efficiently than graph reducers because suspensions in Tim are cheap to create and use. The combination of the evaluation style of closure reducers with the λ-lifting environment analysis developed for graph reducers produces an execution model that is simpler and more effective than either.

References

(Abdali 76) S Kamal Abdali,
An Abstraction Algorithm for Combinatory Logic,

Journal of Symbolic Logic Vol 41 No 1, March 1976.

(Augustsson 84) Lennart Augustsson,
A Compiler for Lazy ML
Proceedings 1984 ACM Conference on Lisp and Functional Programming.

(Bird 84) Richard S Bird
Using circular programmes to eliminate multiple traversals of data,
Acta Informatica 21, 239–250, 1984

(Burn,Hankin&Abramsky 85) GL Burn, CL Hankin and S Abramsky
The theory of strictness analysis for higher order functions
in Proceedings of the Workshop on Programs as data objects, Copenhagen, eds H Ganzinger and N Jones, Springer Verlag Lecture Notes in Computer Science Vol 217,1985.

(Cardelli 84) Luca Cardelli,
The Functional Abstract Machine,
Bell Laboratories Computing Science Technical Report No.107

(Cousineau 87) G Cousineau, P-L Curien, M Mauny,
The Categorical Abstract Machine,
Science of Computer Programming Vol 8,pp 173-202,1987.

(Fairbairn 83) Jon Fairbairn,
Ponder and its Type System,
University of Cambridge Computer Laboratory Technical Report No.31, 1983.

(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.

(Fairbairn&Wray 86) Jon Fairbairn and Stuart Wray,
Code generation techniques for functional languages,
1986 ACM Conference on Lisp and Functional Programming (proceedings) pp 95-104.

(Fairbairn&Wray 87) Jon Fairbairn and Stuart Wray,
Tim — A simple, lazy abstract machine to execute supercombinators,
Third Conference on Functional Programming Languages and Computer Architecture, Springer Lecture notes in Computer Science 274.

(Friedman&Wise 76) Daniel Friedman and David Wise,
Cons should not evaluate its arguments,
Automata, Languages and Programming, Edinburgh University Press 1976

(Hughes 82) John Hughes,
Graph Reduction with Supercombinators,
Technical monograph PRG-28, Oxford PRG.

(Johnsson 85) Thomas Johnsson,
Lambda-lifting: Transforming programs into recursion equations,
Proceedings of 1985 Functional Languages and Computer Architecture Conf., LNCS 201, Springer.

(Jones 84) Simon B Jones,
A range of operating systems written in a purely functional style,
University of Sterling Department of Computer Science, Technical report TR16

(Peyton Jones 85) Simon L Peyton Jones,
Yacc in Sasl—an Exercise in Functional Programming,
Software Practice and Experience Vol 15 (8), 807–820, August 1985

(Peyton Jones 87) Simon L Peyton Jones,
The Implementation of Functional Programming Languages,
Prentice-Hall international ISBN 0-13-453333-X

(Peyton Jones 88) Simon L Peyton Jones,
The Spineless Tagless G-Machine,
Proceedings 1988 Aspenäs workshop on implementation of functional languages (to appear).

(Schmidt 82) David A Schmidt,
Denotational Semantics as a Programming Language,
University of Edinburgh Department of Computer Science Internal Report CSR-100-82.

(Sijtsma 88) Berend A. Sijtsma,
Verification and Derivation of Infinite-List Programs,
PhD Thesis, Rijksuniversiteit Groningen, Netherlands, October 1988.

(Tillotson 85) Mark Tillotson,
Introduction to the Functional Programming Language "Ponder",
University of Cambridge Computer Laboratory Technical Report No.65, May 1985.

(Turner 79) David Turner,
A new implementation technique for applicative languages,
Software Practice and Experience, Vol 9 pp 31-49

(Wadler&Hughes 87) Philip Wadler and John Hughes

Projections for Strictness analysis

Third Conference on Functional Programming Languages and Computer Architecture, Springer Lecture notes in Computer Science 274.

(Wray 86) Stuart Wray,

Implementation and programming techniques for functional languages

University of Cambridge Technical Report 92