Tim: A Simple, Lazy Abstract Machine to Execute Supercombinators

Jon Fairbairn

University of Cambridge Computer Laboratory,
Corn Exchange Street,
Cambridge CB2 3QG,
United Kingdom.
Telephone +44 223 334649
Mail address: jf@UK.AC.Cam.CL.

Stuart Wray

Olivetti Research,
4a Market Hill,
Cambridge CB2 3NJ,
United Kingdom.
Telephone +44 223 323686.
Mail address: scw@UK.CO.Cam.cam-orl

Abstract

This paper is a description of the three instruction machine Tim, an abstract machine for the execution of supercombinators. Tim usually executes programmes faster than the G-machine style of abstract machine while being at least as easy to implement as an S–K combinator reducer. It has a lower overhead for passing unevaluated arguments than the G-machine, resulting in good performance even without strictness analysis, and is probably easier to implement in hardware.

The description begins with a presentation of the instruction set of the machine, followed by the operational semantics of the normal order version and the algorithm to convert combinators to instructions. It then develops the machine to allow lazy evaluation and the use of sharing and strictness analysis. The final sections of the paper give some performance figures and comment upon the suitability of the machine for hardware implementation.

Introduction

The current release of the Ponder compiler [Fairbairn 83, Fairbairn 85, Tillotson 85] uses an abstract machine [Fairbairn & Wray 86] for graph reduction that is similar to the G-machine [Johnsson 83]. Although that implementation performs moderately well, the architecture of the abstract machine seems to be an obstacle to further improvements in performance. Such machines spend most of their time in an essentially interpretive mode, reducing the graphs of expressions passed as arguments to functions. This interpretation is invariably slower than the direct execution of machine code.

The compile time techniques of strictness and sharing analysis help to reduce the amount of interpretive code, but neither of these can produce perfect code in all cases. The information derived by strictness analysis definitely helps improve performance, but higher order strictness information gives little or no improvement [Fairbairn & Wray 86]. While we expect that the combined use of higher order and non-flat domain strictness analysis [Hughes 85b, Kieburtz & Napierala 85, Wadler 85] would detect many more strict contexts, it is difficult to take advantage of this extra information at run-time on the old architecture. Similarly it is almost impossible to use sharing analysis to any advantage at run time.

The new machine arises out of an attempt to restructure things so that these techniques of analysis can be used more effectively. The result shares properties with the SECD machine [Landin 64], the architecture used at Yale [Hudak & Goldberg 85, Hudak & Kranz 84] and frame based reduction engines. Although the new machine often runs programmes at twice the speed of the old, it is unfortunate that no effective way has been found to exploit strictness information for other than functions over machine values. What is perhaps more important is the improvement for programmes where strictness is difficult to detect.

Basic idea

Efficient implementation of functional languages on conventional machines is difficult because one cannot always know whether an expression passed as an argument to a function will be evaluated. In such cases it the evaluation must be deferred until later, which necessitates the passing of some representation of the unevaluated expression. One of the conventional solutions is to use a combinator graph to represent the expression. This graph is either evaluated via graph reduction later, or never accessed again.

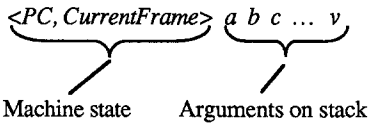
We observe that, when an expression is built as a graph, either it could have been evaluated directly, or we could have saved the expense of even building the graph. The idea behind our new machine is to replace graphs with pointers to code that will compute the desired result. Evidently the code needs to access local variables, so a frame pointer is needed as well. This means that objects are now represented by closures, but we require programmes to have been converted into supercombinators, and these only access variables in the frame immediately addressed by the frame pointer. This makes things simpler than in older frame based implementations where chains of environment pointers are needed.

The Architecture for Normal Order Evaluation

This section describes the simplest form of the machine, which makes no attempt to preserve the values of shared computations and hence performs normal order evaluation rather than lazy evaluation (it should be noted that without lazy evaluation the machine is hopelessly inefficient).

The machine comprises an argument stack, a frame heap and a code stream. Each object is represented as a pair consisting of a frame pointer and a code pointer. The argument stack holds arguments to functions as they are being built up, and the frame heap holds frames of arguments for combinators.

There are registers PC to point to the current code, CurrentFrame to address the current frame and ArgP to hold the stack. You can think of the state of the machine as representing a function applied to some arguments; the function is represented by the code addressed by PC in the environment of CurrentFrame and the arguments are on the stack in order.



There are just three instructions:

`Take` *n*

Takes *n* items off the argument stack and puts them in a new frame on the heap, adjusting CurrentFrame to point to this new frame. Note that these frames are not explicitly deallocated and must therefore be garbage collected away. `Take` represents the beginning of a combinator.

`Push` <item>

Pushes an <item> onto the argument stack. <item> can be `arg` n — the n^{th} argument in the current frame, `combinator` C — the code sequence for the combinator C with an empty frame pointer, or `label` L — a label within the current combinator with the current frame pointer.

`Enter` <item>

Item is as in `Push` (except that it will never be a `label`), and the effect is to load CurrentFrame and PC with the item.

Formally the machine can be represented as a tuple <PC, CurrentFrame, ArgP, Frames>. Frames is an indexed structure; we will write F for frame heaps, f for indexes of frames in heaps and $F[f \mapsto (a_1, \ldots, a_n)]$ for a heap extending F such that f indexes $(a_1, \ldots, a_n)$. Each a_i is of course of the form $<c, f>$. Labels are represented by the code to which they point.

The operation of the machine is described by the following rewrite rules:

$<$[`Take` n; I], f_0, $(a_1, \ldots, a_n, A), F>$	$\Rightarrow <I, f, A, F\,[f \mapsto (a_1, \ldots, a_n)]>$, where f selects an unused frame
$<$[`Push arg` n; I], f, A, $F[f \mapsto (\ldots, a_n, \ldots)]>$	$\Rightarrow <I, f, (a_n, A), F\,[f \mapsto (\ldots, a_n, \ldots)]>$
$<$[`Push label` l; I], f, A, $F>$	$\Rightarrow <I, f, ((l, f), A), F>$
$<$[`Push combinator` c; I], f, A, $F>$	$\Rightarrow <I, f, ((c, 0), A), F>$
$<$[`Enter arg` n], f, A, $F[f \mapsto (\ldots, <c_n, f_n>, \ldots)]>$	$\Rightarrow <c_n, f_n, A, F[f \mapsto (\ldots, <c_n, f_n>, \ldots)]>$
$<$[`Enter combinator` c], f, A, $F>$	$\Rightarrow <c, 0, A, F>$

Examples

The behaviour of the machine will be illustrated by tracing the reductions of two combinators. First consider **K**. The λ-expression for **K** is $\lambda a.\ \lambda b.\ a$. This compiles into the code

```
[Take 2; Enter arg 1]
```

(see below for the compilation algorithm). An application of **K** to two arguments $\alpha_1\ \alpha_2$ would be represented by the machine state

$<$[`Take 2; Enter arg 1`], 0> $\alpha_1\ \alpha_2\ \ldots$

For clarity the frame heap will be represented separately. Reduction proceeds as follows:

$\Rightarrow <$[`Enter arg 1`], $f> \ldots$	$f \mapsto (\alpha_1, \alpha_2)$
$\Rightarrow <\alpha_1> \ldots$	$f \mapsto (\alpha_1, \alpha_2)$

leaving the machine executing α_1 as one would expect.

The code for $\mathbf{Z} = \lambda x\ \lambda y.\ y\ x$ is

```
[Take 2; Push arg 1; Enter arg 2]
```

The reduction of $\mathbf{Z}\ \alpha_1\ \mathbf{K}\ \alpha_2$ is represented by the following sequence:

<[Take 2; Push arg 1; Enter arg 2], 0> α_1 **K** α_2 ...

$\Rightarrow$ <[Push arg 1; Enter arg 2], f_1> α_2 ... $\quad f_1 \mapsto (\alpha_1, K)$

$\Rightarrow$ <[Enter arg 2], f_1 > α_1 α_2 ... $\quad f_1 \mapsto (\alpha_1, K)$

$\Rightarrow$ <[Take 2; Enter arg 1], 0> α_1 α_2 ... $\quad f_1 \mapsto (\alpha_1, K)$

and then proceeds as in the example for **K**.

Converting combinators to code

This section presents an algorithm to convert a series of combinator definitions into Tim code.
A programme is taken to be a sequence of combinator definitions of the form

c_1 $=_{def}$ *combinator*$_1$;

...

c_n $=_{def}$ *combinator*$_n$;

main-expression

We assume that a combinator is of the form $\lambda a_1, \ldots a_n$. *expression*, and that an expression is either an *atom* (i.e. an a_i or a combinator name c_i) or an application of one expression to another.

G ⟦p⟧ ρ is the function used to generate code for the programme p. The environment ρ is just used to remember the definitions of the combinators and would initially be empty or contain the definitions of any built-in operations. Since the definitions of the combinators may be mutually recursive the whole environment must be built before any code is generated. Although the algorithm as presented would generate infinite code sequences for recursive combinators it is to be understood that in practice labels are used to rather than passing copies of code around.The subsidiary function C generates code for combinators, calling P to generate pushes and E to generate enters.

G ⟦c_i $=_{def}$ *expression*$_i$; p⟧ ρ $\Rightarrow$ G ⟦p⟧ (ρ [*expression*$_i$ / c_i])

G ⟦*expression*⟧ ρ $\Rightarrow$ C ⟦*expression*⟧ ρ

C ⟦$\lambda a_1, \ldots a_n$. *body*⟧ ρ $\Rightarrow$ [Take n; C ⟦*body*⟧ ρ]

C ⟦e_1 e_2⟧ ρ $\Rightarrow$ [P ⟦e_2 ⟧ ρ ; C ⟦e_1⟧ ρ]

C ⟦*atom*⟧ ρ $\Rightarrow$ E ⟦*atom*⟧ ρ

P ⟦a_n⟧ ρ $\Rightarrow$ [Push arg n]

P ⟦c_i⟧ ρ $\Rightarrow$ [Push combinator (C ⟦ρ (c_i)⟧)]

P ⟦e⟧ ρ $\Rightarrow$ [Push label (C ⟦e⟧)]

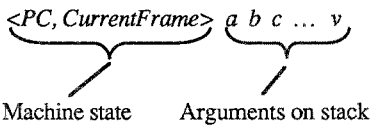
E ⟦a_n⟧ ρ $\Rightarrow$ [Enter arg n]

E ⟦c_i⟧ ρ $\Rightarrow$ [Enter combinator (C ⟦ρ (c_i)⟧)]

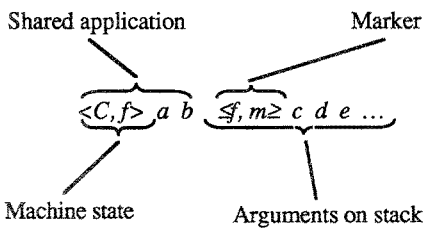
In the Tim implementation from which the data in this paper were gathered, the abstract instructions were simply macro expanded into Acorn Risc Machine assembler.

Making it Lazy

Laziness consists of remembering the value of shared expressions the first time they are reduced, so that subsequent accesses do not recompute the value. In a conventional graph reducer this is achieved by overwriting nodes in the graph. The new machine does not use a graph in this sense, so what can we overwrite? Fortunately, one of the properties of supercombinators is that only combinator arguments are ever shared. This means that, when the reduced value of an argument has been calculated the answer should be written into the frame that it came from. Notice also that a reduced value must always be a partially applied function (a combinator applied to too few arguments).
There is still a difficulty. As yet we have no way of holding up the reduction of an expression half way through in order to preserve a partially applied function: as soon as a combinator is entered it takes all its arguments off the stack and proceeds. The solution to this is to put a marker on the stack that indicates that a shared computation is in progress. Recall that the machine can be considered as a function applied to the arguments on the stack:



When an expression is shared, the picture is like this:



Here the marker $\leq f, m \geq$ indicates that the shared application came from the m^{th} argument of the frame f. Eventually the machine will enter a combinator C that takes more arguments than appear before the marker. Before entering C the argument indicated by the marker should be updated with a representation of C applied to the arguments above the marker.
Below is a modified description of the operation of the machine that reflects the changes needed to handle laziness:

$<[\texttt{Take}\ n; I], f_0, (a_1, \ldots, a_i, \leq f, m \geq, A), F\ [f \mapsto (\ldots, a_m, \ldots)]>$ for $0 \leq i < n$

$\Rightarrow <P, f_1, A, F\ [f \mapsto (\ldots, <P, f_1>, \ldots)]\ [f_1 \mapsto (a_1, \ldots, a_i)]>$

where $P = [\texttt{Push arg}\ i; \ldots; \texttt{Push arg 1}; \texttt{Take}\ n; I]$

$<[\texttt{Take}\ n; I], f_0, (a_1, \ldots, a_n, A), F> \Rightarrow <I, f, A, F\ [f \mapsto (a_1, \ldots, a_n)]>$

$<[\texttt{Push arg}\ n; I], f, A, F> \Rightarrow <I, f, (<[\texttt{Enter arg}\ n], f>, A), F>$

$<[\texttt{Push label}\ l; I], f, A, F> \Rightarrow <I, f, (<I, f>, A), F>$

$<[\texttt{Push combinator}\ c; I], f, A, F> \Rightarrow <I, f, (<c, 0>, A), F>$

$<[\texttt{Enter arg}\ n], f, A, F\ [f \mapsto (\ldots, <c_n f_n>, \ldots)]>$

$\Rightarrow <c_n, f_n, (\leq f, n \geq, A), F\ [f \mapsto (\ldots, <c_n f_n>, \ldots)]>$

$<[\texttt{Enter combinator}\ c], f, A, F> \Rightarrow <c, 0, A, F>$

There are two changes in addition to the one already described. The `Push arg` instruction can no longer copy the argument onto the stack — if it were to do so, the object on the stack would not be updated when the argument was. Instead, an `Enter arg` is pushed, ensuring that the argument is only accessed when its value is needed. Finally, `Enter arg` is responsible for putting the marker on the stack.
Although the formal description appears to create code 'on the fly,' in a practical implementation the number of arguments and the address of the `Take` instruction can be remembered in the newly created frame, so that the same code can always be used to push the arguments. A slight improvement to this would be to take advantage of the fact that the largest number of arguments to any combinator is known at compile-time, so that instead of using a loop one could index a table of entry points for code to perform n pushes.

Representations

To be useful, one would expect the machine to need built in representations of objects such as integers, characters and pairs. In fact in this machine it is unnecessary to have a special representation for pairs; their functional representation has the right operational behaviour. The existence of terminals and other peripherals which only understand a particular representation of characters makes it necessary to provide characters with this representation. Machine integers tend to be faster than their functional versions, so they should be provided as well. Fortunately this can be done without much disturbance to the structure of the machine.

Pairs

Before describing the representation of machine values, we shall consider the behaviour of the functional representation of pairs. The pair $\langle a, b \rangle$ will be represented initially as an application of the function $pair = \lambda a.\ \lambda b.\ \lambda u.\ u\ a\ b$ to the objects a and b. This has all the functional properties required of a pair, but what about the operational aspect? What we must consider is how a pair is represented after it is reduced. *Pair* is a combinator that takes three arguments, and *pair a b* supplies it with only two. If the application *pair a b* is shared, what will happen? If we follow the machine description, we find that the shared application will be entered with a marker on the stack, it will push a and b and then enter the combinator for *pair*. The take instruction at the beginning of *pair* will then update the argument indicated by the marker with code to push arguments and re-enter pair, with a frame that contains just a and b. This is exactly what we want! Shared copies of pairs are represented as two adjacent locations in store containing the left and right components of the pair.

Machine Values

We invent a pseudo-combinator `Self` that, when entered, simply pushes itself back onto the stack together with CurrentFrame and enters its first argument. A machine value n can then be represented as <`Self`, n>. Now we can have an <item> `constant` with the rule

<[`Push constant` k; I], f, A, F> ⇒ <I, f, (<`Self`, k>, A), F>

and Self performs the reduction

<`Self`, f, (<c, f_1>A), F> ⇒ <c, f_1, (<`Self`, f>, A), F>

Now machine values are functions like everything else, but can be called as subroutines simply by pushing a continuation on the stack. So the machine code for a strict built-in operators will begin by pushing continuations and entering each strict argument in turn. For example a unary operator such as negation will begin as follows:

```
   Take 1
   Push label L;
   Enter arg 1;
L: negate the the frame part of the object on the stack and enter it.
```

The Fixed point combinator

In conventional graph reducers cyclic structures are introduced by a built-in version of the fixed point combinator **Y**. Such a version seems indispensable [Kieburz 86] so what is the equivalent for Tim? What we want to do is to make sure that the structure representing $\mathbf{Y}f$ is built at most once. A way of achieving this is to make **Y** create a frame with two arguments in it. The first argument will be f, but the second will hold $\mathbf{Y}f$. Initially this will be a label that pushes its second argument ($\mathbf{Y}f$) and enters its first (f), but we want the second argument to be updated when $f(\mathbf{Y}f)$ reduces. Hence we must push a marker pointing at the second argument before we push it:

```
Y: Take 1 and extend it to two
   Push label yf into arg 2
yf:put a mark on the stack pointing to arg 2
   Push arg 2
   Enter arg 1
```

Optimisations

This section describes some ways of improving the performance of the machine, in particular how to take advantage of strictness and sharing analyses. Use of sharing information is described first since it is the simpler of the two.

Making use of Sharing Analysis

The alterations to the machine for laziness impose an overhead on the execution of the machine. The overhead appears in three places: Testing to see if there are enough arguments on the stack for a `Take` instruction (and behaving appropriately if there are not), pushing the marker on the stack in an `Enter arg` and the indirection involved in `Push arg`. The test in `Take` is only a small overhead; the main cost is creating the representation of the shared application if there are insufficient arguments. There will be a noticable improvement in speed if we can avoid putting unnecessary markers on the stack.

If an argument is only ever evaluated once there is no need to record the value of the reduced version. If we can detect this at compile time then we can avoid the overhead by changing the two instructions that access arguments. This means adding an extra (optional) parameter to the instructions and the rules:

<[`Push arg` n, `Unshared`; I], f, A, $F[f \mapsto (\ldots, a_n, \ldots)]$>

$\Rightarrow$ <$I, f, (a_n, A), F[f \mapsto (\ldots, a_n, \ldots)]$>

<[`Enter arg` n, `Unshared`], f, A, $F[f \mapsto (\ldots, \langle c_n f_n \rangle, \ldots)]$>

$\Rightarrow$ <$c_n, f_n, A, F[f \mapsto (\ldots, \langle c_n f_n \rangle, \ldots)]$>

These are just the rules from the normal order version of the machine. An unsophisticated analysis of the code that just counts the number of times that each argument appears in each obvious path through a combinator speeds the machine up by about 10%.

Making use of Strictness Analysis

The ponder code generator for the old abstract machine uses strictness annotations to decide whether expressions can be evaluated in applicative order, or must be built as graphs pending later evaluation. In that machine the advantage of evaluating function objects immediately rather than passing them as graphs is negligible, since the result of the evaluation would necessarily be a graph. In this machine functions

use up all their argumnents as soon as they are entered, so to evaluate functions in advance involves introducing new stack markers to halt the reduction. Our experiments have shown that this costs more than the resulting gain. An alternative is to evaluate only machine values strictly. In order to ensure return from the evaluation all that is necessary is to push a return label onto the stack before starting the strict computation, as mentioned above for primitive operations.

An advantage of the new machine over the old is that when an argument is in a strict position we can generate strict code, even if it is a subexpression of a lazy one. In the old machine strict arguments in lazy expressions had to be built as graphs. For example, consider the expression $f\,(-\,(3 \times x))$. If f is lazy on its first argument, the entire expression $(-\,(3 \times x))$ would have to be built as a graph in the old machine despite the fact that – is strict on its argument. For Tim, the code for $(3 \times x)$ can be strict even though a closure must be passed to f.

An advantage of evaluating strict arguments in advance is that arguments that are known to be evaluated may be treated as unshared, removing the overhead involved with updating arguments to their reduced form.

Other Optimisations

The machine presented above is the simplest that perfoms lazy evaluation. It is not yet as efficient as it can be when dealing with lazy expressions. The simplest case of this is that when a combinator is entered with a marker at the top of the stack, so that the machine state is $C \leq f, m \geq \ldots$; the machine will create an empty frame, put no arguments in it and update the place addressed by the marker with C and this frame. Although this has the right effect, in a real implementation it wastes time, and it can be avoided by treating markers at the top of stack as a special case, and updating them with the combinator.

A similar, but more severe inefficiency occurs when several markers occur on the stack in succession. The machine will pull the arguments that occur above the markers into a frame and update the first marker, push the arguments back onto the stack and re-do the `Take`, and then go through the whole procedure again for the remaining markers. The simple solution to this is to use the same frame for all the markers. This also avoids building chains of indirections to the arguments in the first created frame. This optimisation could be extended to share the frames created when a `Take` instruction is interrupted more than once, but initial experiments suggest that this may not be worthwhile.

A slight inefficiency occurs when the text of a programme contains $C\ a_1 \ldots a_n \ldots$, when C is a combinator that takes n arguments. The machine will push each of a_n to a_1 onto the argument stack and then enter C, which will immediately take them off the stack and build a frame. In such cases it is obviously better to create the frame before pushing a_n, push them all directly into the frame and enter C after the `Take` instruction.

Unused Frames

There is a class of combinators that compile into Tim code that contains no labels. If such a combinator has no shared arguments, making a frame is redundant. A typical example is **K**, which takes two arguments off the stack and enters the first. In such combinators the frame is not needed after leaving their bodies and can be deleted immediately. Indeed, for combinators as simple as **K** the needed arguments can probably be pulled off the stack into registers, avoiding the use of a frame entirely.

On the whole this sort of machine level optimisation only makes two or three percent improvement, but it is worth doing at least a few of them: it was possible to speed up our initial implementation by about 20% by improving the codings of the abstract machine instructions and primitive operations.

Garbage Collection

One disadvantage of this machine architecture is that the garbage collector must be able to handle variable length objects. On top of this, there is a potential problem with space leaks. 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 annotate each label with a bit pattern indicating which arguments it needs. The garbage collector can then use this pattern to decide which entries in the frame must be kept.
Simon Finn and Simon Peyton Jones pointed this out to the authors independently.

Observations

Relation to other architectures

An important aspect of the design of the Tim machine is that it is optimised for normal order evaluation. The separate addition of a lazy evaluation mechanism means that there is no overhead when it is not needed. This addition of laziness is facilitated by the fact that the design assumes prior compilation into supercombinators (or at least lambda-lifted combinators), an assumption not made in the design of earlier machines.
Related machines are the Functional Abstract Machine [Cardelli 84], which was designed primarily to support applicative order evaluation, and the Categorical Abstract Machine [Cousineau et al 87], again applicative order. The categorical machine differs more significantly in that it addresses a different level of abstraction, details of machine representation and updating for laziness being left to lower levels.

A Hardware Implementation

The simplicity of this machine suggests that it would not be difficult to implement as a chip. Evidently a few more instructions would be needed since the `Take` instruction is a little too complicated for a single instruction, but there are three ways in which a specially designed machine would have advantage over a standard processor.

i) All objects are represented as pairs of words with code and frame pointer. This takes two memory cycles per object access, whereas one could make the bus wide enough to transfer whole objects at a time.
ii) The stack marker for laziness could be put in a limit register, and the value of this register tested in parallel with the execution of `Take` instructions. This would reduce the overhead by making the updating part of `Take` an interrupt event.
iii) On a conventional machine the transfer of data from the argument stack to the frame heap must go through the registers of the processor. If the stack and heap were held in separate memory units, the `Take` instruction could transfer all the arguments from stack to heap without getting them into registers.

Performance

Six programmes were used as benchmarks: 'Nfib' is the familiar nfib 20 benchmark and 'tak' is the Takeushi LISP benchmark. 'Parser' is a lambda expression parser, 'logic' is a logic (hardware) simulator, 'quicksort' is quicksort of 100 random numbers and 'turing' is a Turing machine emulator. The first table compares the best performance of the old machine with the best performance of Tim. All timings are for an Acorn RISC Machine.

	Old machine (seconds)	Tim (seconds)	Old/Tim (ratio)
nfib	1.17	1.21	0.97
tak	2.29	4.22	0.54
parser	3.58	1.31	2.73
logic	1.93	0.75	2.57
quicksort	1.77	0.64	2.77
turing	4.67	2.23	2.09

The Tim version of nfib actually uses fewer instructions than the old machine, but the cost of data transfers on the new machine is twice that of the old, since everything is represented by two word closures. Nfib and tak are compiled unusually well for the old machine because of their good strictness properties. Strictness analysis of the other, more realistic, programmes is much less effective and Tim is a clear win for these.
The second table shows the performance of Tim with and without two optimisations. The first column gives timings for Tim with neither optimisation. The second column shows the effect of using simple sharing analysis as described in the section above. The last column shows the effect of also treating strict built-in arithmetic functions specially.

	No optimisation (seconds)	Sharing analysis (seconds)	Semi-strict (seconds)
nfib	1.96	1.95	1.21
tak	5.30	5.30	4.22
parser	1.56	1.32	1.31
logic	0.90	0.75	0.75
quicksort	0.86	0.69	0.64
turing	2.74	2.38	2.23

Neither implementation takes any advantage of machine level optimisations such as stack slaving, but one would expect that the improvement would be about the same in both cases. It is worth remembering that the old implementation was the result of several years of research, whereas work on the new one amounts to a few months. We feel that as well as being a quick method of implementing lazy evaluation, Tim promises interesting developments in the future.

Acknowledgements

The research was begun while both authors were post-doctoral research fellows funded by the Science and Engineering Research Council of Great Britain. The first author worked on this paper while at Glasgow University. Special thanks to John Hughes and his research group there for helpful comments, advice and lively atmosphere.
The Acorn Risc Machine was generously lent by Acorn Computer Limited, Cambridge UK.

References

[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 & Stuart Wray,
Code generation techniques for functional languages,
1986 ACM Conference on Lisp and Functional Programming (proceedings) pp 95–104

[Hughes 85b]: John Hughes,
Strictness Detection in Non-Flat Domains,
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

[Hudak & Goldberg 85]: Paul Hudak & Benjamin Goldberg,
Serial Combinators: "Optimal" Grains of Parallelism,
Yale University Department of Computer Science 1985

[Hudak & Kranz 84]: Paul Hudak & David Kranz,
A combinator based compiler for a functional language,
11th ACM Symposium on Principles of Programming Languages ACM Jan 1984, pp 121–132

[Johnsson 83]: Thomas Johnsson,
The G-Machine: An Abstract Machine for Graph Reduction,
Proceedings of SERC Declarative Programming Workshop at UCL, April 1983

[Kieburtz & Napierala 85]: Richard B. Kieburtz & Maria Napierala,
A studied laziness — strictness analysis with stuctured data types,
Oregon Graduate Centre, Extended Abstract, July 1985.

[Kieburtz 86]: Richard B. Kieburtz,
When chasing your tail saves time,
Information Processing Letters, December 1986.

[Landin 64]: P. J. Landin,
The Mechanical Evaluation of Expressions,
Computer Journal Volume 6 Number 4 pp 308–320, 1964.

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

[Wadler 85]: Phil Wadler,
Strictness Analysis on Non-Flat Domains,
Programming Research Group, Oxford University, November 1985.