Optimising Partial Applications in TIM

David Wakeling and Alan Dix
University of York[*]

Technical Report Number 215 (November 1993)

1 Introduction

In [3] Fairbairn and Wray introduced TIM, a simple abstract machine for executing supercombinators. Their abstract machine is an elegant design optimised for normal order evaluation. However, the addition of a mechanism to implement lazy evaluation considerably increases its complexity and imposes a significant overhead.

We originally made this observation in March 1989, when a draft of this paper was circulated privately. Since then, there have been a number of publications about TIM (see, for example, [1, 10, 8]). Inevitably, some of the ideas presented here have been invented independently, or have been improved upon in these subsequent publications. However, we feel our central idea for optimising partial applications in TIM has still to gain the attention that it deserves.

2 The TIM Architecture

A frustrating aspect of the TIM literature is that every author seems to adopt slightly different terminology and notation to describe the same abstract machine. For the sake of definiteness, we take Fairbairn and Wray's 1987 paper [3] as the source of our terminology and notation, referring to the machine described there as the *standard TIM*. However, in addition to that paper, the reader may also wish to consult the book by Peyton Jones and Lester [8] for an excellent tutorial.

TIM represents all values as *closures* of the form $< c, f >$, where c is the *code pointer* and f is the *frame pointer*. The frame pointer points to a *frame* containing the arguments used by the code at c. The are just three kinds of instructions for manipulating closures. The instruction

    Enter $< c, f >$

[*]Authors' address: Department of Computer Science, University of York, Heslington, York Y01 5DD, United Kingdom. Electronic mail: `dw@minster.york.ac.uk`, `alan@minster.york.ac.uk`

computes the value of a closure by executing the code at c, and

`Push` $< c, f >$

pushes the closure $< c, f >$ onto an argument stack. The instruction

`Take` n

takes n closures off the argument stack into a new frame in the heap. During this operation, special "markers" may be encountered amidst the closures on the stack, indicating that an update must be performed in order to preserve laziness. A marker specifies the frame argument that must be overwritten with its normal form. A `Take` instruction only occurs once at the beginning of the code for each combinator.

The state of the TIM itself is represented as a tuple <PC, CF, Stack, Heap>. The two registers, PC and CF are the *current code pointer* and the *current frame pointer* respectively. The stack holds the arguments to supercombinators as they are being built up, and the heap holds the frames of combinators "in the midst" of reduction. The operation of the machine is described using state transitions of the form:

$$\text{<PC, CF, Stack, Heap>} \Rightarrow \text{<PC', CF', Stack', Heap'>}$$

3 A TIM Implementation

We wrote a compiler to translate programs written in the Functional Language Intermediate Code, FLIC [9] into TIM code. This compiler follows the scheme outlined by Fairbairn and Wray [3] except in the compilation of structured types.

The treatment of structured types in FLIC is fundamentally different from their treatment in TIM. In FLIC they are represented as tagged-tuples and case-analysis and untupling are performed by special instructions. In the original TIM design they are represented as partial applications of higher-order "distributing functions", and case-analysis and untupling are performed by the constructors of the type. Distributing functions are elegant but rather inefficient, and so our TIM uses a tagged-tuple representation instead. A structured type is represented by a closure $< \texttt{Struct}, f >$ whose frame contains the structure's components together with a numerical *tag* which identifies the constructor that was used to build the structured type. To create this representation we have the instruction `Construct` n t with the rule:

$$<[\texttt{Construct}\ n\ t; I],\ f_1,\ (a_1, \ldots, a_n, A),\ F>$$
$$\Rightarrow\ <I,\ f_1,\ (< \texttt{Struct}, f >, A),\ F\,[f \mapsto (a_1,\ \ldots,\ a_n, t)]>$$

Two further instructions are required to implement pattern-matching against structured types. The first, `Deconstruct`, takes a tagged tuple apart on the stack

$$<[\texttt{Deconstruct}\ n;\ I],\ f_1,\ (<\texttt{Struct}, f>, A),\ F\,[f \mapsto (a_1, \ldots, a_n, t)]> \\ \Rightarrow\ <I,\ f_1,\ (a_1, \ldots, a_n, A),\ F>$$

while the second, `Switch`, is for performing case analysis on the value of a structure tag

$$<[\texttt{Switch}\ (l_1, l_2, \ldots, l_m)],\ f_1,\ (<\texttt{Struct}, f>, A),\ F\,[f \mapsto (a_1, \ldots, a_n, t)]> \\ \Rightarrow\ <l_t,\ f_1,\ A,\ F>$$

This representation of structured types has similarities with the one described in [8].

4 Instruction Profiles

We used our compiler to compile seven benchmark programs written in FLIC. These benchmarks were:

nfib	— the nfib 20 benchmark computed using a doubly-recursive function;
tak	— the Takeushi function with arguments 18 12 6;
primes	— the list of prime numbers up to 500 computed using Eratosthene's sieve;
sort	— an insertion sort of 200 random numbers;
queens	— produces all of the solutions to the 8-queens problem;
folds	— accumulates the sum of the sums of all prefixes of a list of 50 numbers
quad	— maps the function *quad quad succ* over a list of 1000 numbers.

Although the *nfib* and *tak* programs are somewhat atypical, *primes*, *sort* and *queens* are all examples of more realistic programs. The *folds* and *quad* programs make extensive use of partial application, a style which is commonly advocated in textbooks on functional programming [2].

For each of these programs, we obtained two instruction profiles. These profiles are given in Appendix A and summarised in Figure 1 and Figure 2 below. The first profile, in Figure 1, is for the abstract machine which executes each TIM instruction in unit time, while the second profile, in Figure 2, is for a concrete machine which executes each TIM instruction in the time required to to execute the assembly language instructions that implement it.

Just three instructions: `Enter arg'`, `Self` and `Take` account for nearly 70% of the instructions executed by the concrete machine. The `Enter arg'` instruction is the indirection which is pushed onto the stack instead of an argument in order to avoid duplicating a redex, and `Self` is the pseudo-combinator which is executed when evaluating numbers and constructors in normal form.

The large number of `Enter arg'` and `Self` instructions can be reduced by performing sharing, strictness and evaluation analyses at compile-time. These analyses are well described in the original paper [3], and we shall not consider them further here. Instead, we shall concentrate on improving the efficiency of the most expensive instruction, `Take`.

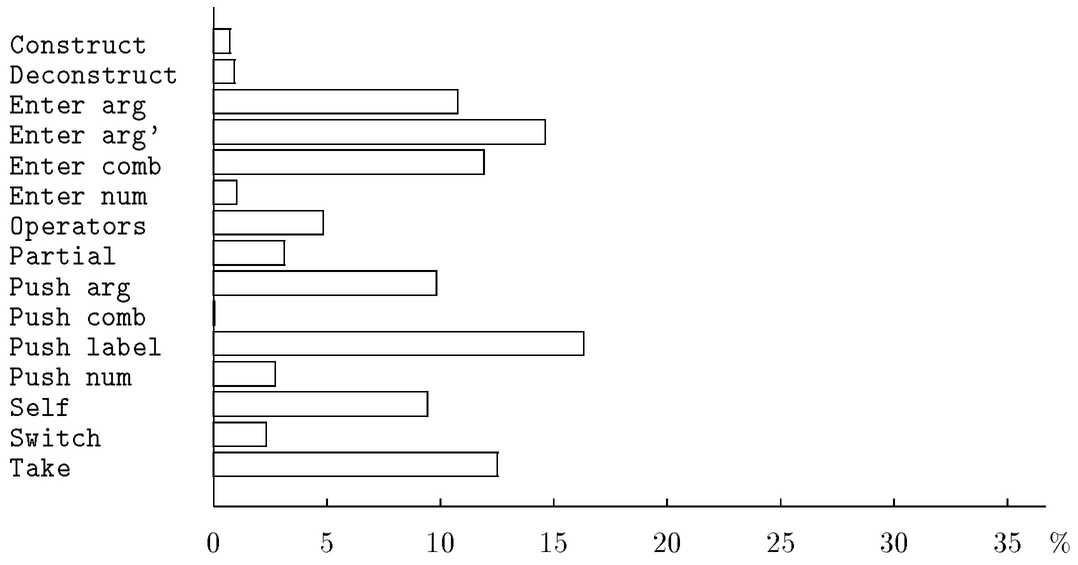

Figure 1: Instruction profile (abstract machine)

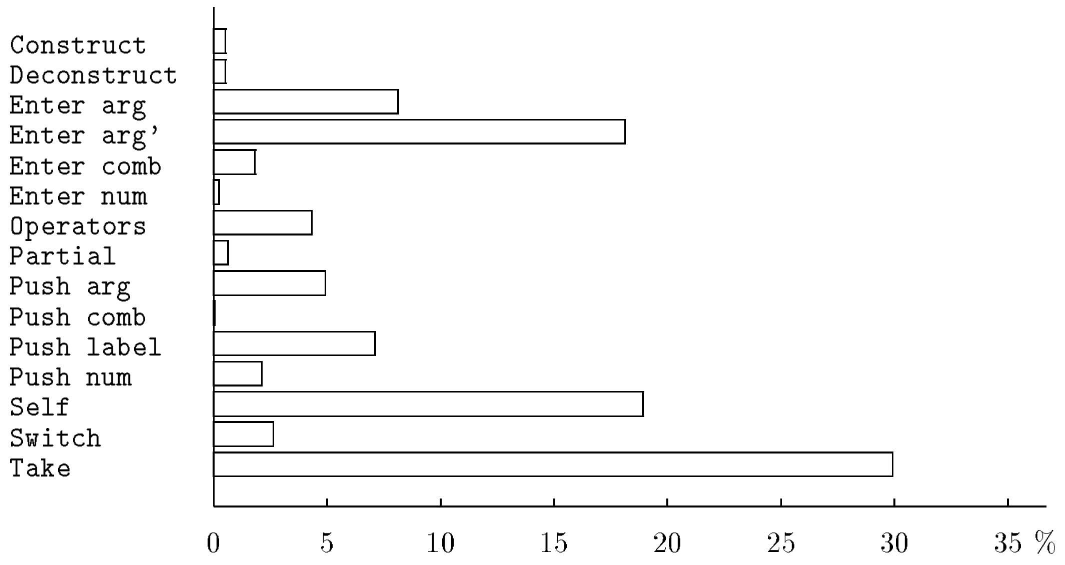

Figure 2: Instruction profile (concrete machine)

5 The Take Instruction

Recall that the `Take` n instruction takes n closures off the argument stack into a new frame in the heap, triggering an update mechanism when it encounters a "marker" instead of a normal closure. A marker is pushed onto the stack whenever a shared expression is to be reduced. It indicates the frame argument which should be updated when the shared expression reaches normal form. This normal form is always a partial application of a supercombinator.

In TIM, a partial application is represented as a closure whose code pointer is `Partial` and whose frame contains the address of a supercombinator together with some arguments. The code at `Partial` pushes indirections (to avoid duplicating a redex) to the arguments in the frame onto the stack before entering the supercombinator.

For illustration, consider the stages in the execution of the instruction `Take` 3 shown in Figure 3. When the `Take` instruction begins (Figure 3(a)) there are two arguments, a_1 and a_2 at the top of the argument stack followed by a marker and a third argument a_3. The first two arguments are transferred from the stack to the heap, (Figure 3(b) and Figure 3(c)) before the marker is encountered. The argument indicated by the marker is overwritten with its normal form, which is a partial application of the supercombinator, C, currently being reduced to the arguments that were on the stack above the marker (Figure 3(d)). After the marker has been updated, indirections to the partial application's frame are pushed onto the stack and the `Take` 3 instruction is restarted (Figure 3(e)).

The `Take` instruction is expensive for two reasons. The first reason is that it is essentially *interpretive*: its actions are controlled by the data it encounters on the stack. Such interpretive execution is, of course, much slower than the direct execution of machine code. The second reason concerns the representation of partial applications. The `Take` instruction must create an extra frame to represent each partial application of a supercombinator. This leads to more heap storage allocation and garbage collection.

The following sections present two techniques for making `Take` instructions less expensive by improving the representation of partial applications.

6 Child Frame Sharing

Definition. If the supercombinator C_2 is applied anywhere within the body of a supercombinator C_1, then we define C_1 to be the *parent* supercombinator and C_2 to be the *child* supercombinator.

In the standard TIM, the child combinator always begins with a `Take` instruction which creates a new frame and updates any markers that it may encounter by creating extra frames to represent partial applications of the child. An alternative, which avoids creating these extra frames, is for each partial application of the child to share the frame of its first full application. We can always be sure that there are enough arguments for the full application, there may just be some markers in between.

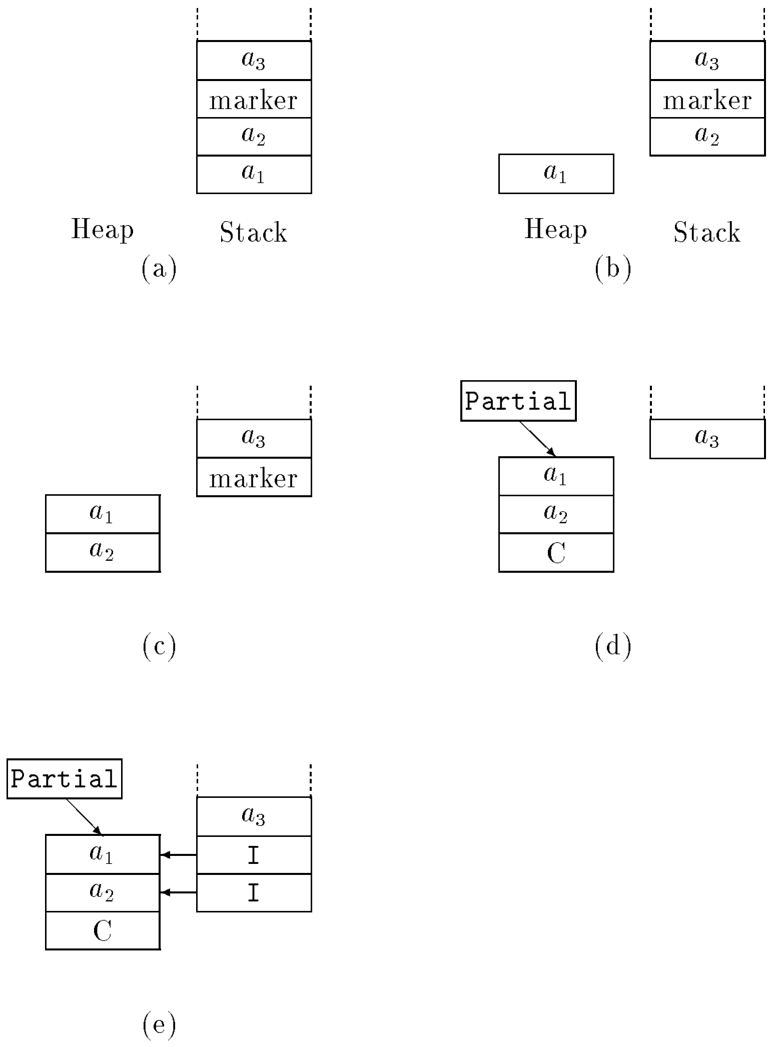

Figure 3: Take instruction example

When the `Take` instruction encounters a marker in, say, the *i*-th argument position, it updates the shared expression with an object whose code pointer is `Partial-`*i* and whose frame pointer is the new frame under construction. The `Take` instruction then continues. The code at `Partial-`*i* pushes indirections to the first *i* arguments in the current frame back onto the stack and then enters the address of the supercombinator as before.

6.1 Discussion

Child frame sharing is particularly advantageous when the markers are kept on a separate stack of their own (i.e. a "dump"), rather than being interleaved among the normal arguments. In this situation, the test for a marker and the transfer of arguments from the stack to the heap are completely independent operations. Thus, the marker test can be moved out of the inner-loop of `Take` instruction, allowing the argument transfer to be performed using whatever efficient block memory move instruction the computer provides.

However, when using this method, it is no longer possible to use the local sharing analysis described in the original paper. This is because the arguments in the child's frame are now shared by partial applications of the child.

Table 1 below shows how our TIM with child frame sharing performs relative to our standard TIM. In all other respects, the two machines are identical. For each of the benchmark programs we measured both the decrease in execution time and the decrease in the amount of heap memory allocated.

	Time decrease %	Allocations decrease %
nfib	0	0
tak	0	0
primes	10	8
sort	0	0
queens	2	2
folds	0	0
quad	26	37

Table 1: Improvements due to Child Frame Sharing

For two of the programs which make explicit use of partial application, *primes* and *quad*, there is a decrease in execution time which can be attributed to the decrease in the amount of memory that they allocate. For the remaining programs there is no significant decrease in either execution time or memory allocated.

Child frame sharing eliminates the extra frame required to represent a partial application of the child combinator, which is why the performance of the programs which make extensive use of partial applications is significantly improved. It is an optimisation that has been suggested by several researchers, including Wray and Fairbairn themselves [10]. However, without the introduction of a separate stack for markers, child frame sharing does nothing to reduce the interpretive overhead of the `Take` instruction, which is why the performance of the remaining programs is unchanged.

An observation about supercombinators, however, tells us that this interpretive overhead can be reduced without a separate stack for markers.

7 Supercombinators

The TIM assumes that all programs have been converted either into *supercombinators* [5] or *lambda-lifted combinators* [6]. Let us assume conversion to supercombinators as described in [5]. Starting with the leftmost, innermost lambda abstraction of the program, every lambda abstraction λv.E is converted into an application of a new supercombinator C to the *maximal free expressions* $e_1 \ldots e_n$ of E. The new supercombinator C itself is then defined as

$$\mathrm{C}\ i_1 \ldots i_n\ v = \mathrm{E}[\ i_1/e_1,\ i_2/e_2 \ldots i_n/e_n\ ]$$

where the formal parameter names $i_1 \ldots i_n$ are not used in E, and the expression E$[x/y]$ denotes the result of substituting x for all occurrences of y in E. Any redundant supercombinator parameters are eliminated by *η-reduction* and then any redundant combinators are also eliminated.

Consider the conversion of the lambda expression

$$\lambda x\ \lambda y.\ +\ y\ (sqrt\ x)$$

into supercombinators. The only maximal free expression of the λy abstraction is $(sqrt\ x)$ and so we can replace the whole lambda expression with

$$\lambda x.\ C_2\ (sqrt\ x)$$

and then generate the new supercombinator C_2, defined as

$$C_2\ i_1\ y = +\ y\ i_1$$

Now the λx abstraction has no free expressions and so the whole lambda expression can be replaced by C_1. We then generate a second new combinator

$$C_1\ x = C_2\ (sqrt\ x)$$

Calling C_1 the parent and C_2 the child again, we can see that in this case the parent *always* supplies the child with one argument, $(sqrt\ x)$, and so the child need only test for a marker in the second argument position. More generally, if η-reduction has *not* been performed, the parent will always supply the child with all but its last "user-level" argument, and so the child need only test for the presence of a marker in the last argument position. Actually, this is an understatement of the case: sometimes the parent will supply the child with all of its arguments, in which case there is no need for the child to perform a test at all.

So, as the parent knows where a marker can occur, it would seem more reasonable to transfer the responsibility for updating markers from the child to the parent, leaving the child to simply pull the arguments off the stack into a new frame. This is the basis of the second technique, parent frame sharing.

8 Parent Frame Sharing

Whenever the code generator encounters a supercombinator applied to too few arguments, it inserts a new `Share label` instruction before the code which pushes the arguments onto the stack. For example, here is the code for an application of a supercombinator C of arity n to only $n-1$ arguments

```
L: Share label L
   Push $a_{n-1}$
      ⋮
   Push $a_1$
   Enter comb C

C: Take unshared $n$
   Code for body of C
```

The `Share label` instruction updates any markers at the very top of the stack with the label and the current frame:

$$<[\texttt{Share Label } l;\ I],\ f_1,\ (\leq f, m \geq, A),\ F\,[f \mapsto (\ldots,\ a_m,\ \ldots)]>$$
$$\Rightarrow\ <[\texttt{Share Label } l;\ I],\ f_1,\ A,\ F\,[f \mapsto (\ldots,\ <l, f_1>,\ \ldots)]>$$

$$<[\texttt{Share Label } l;\ I],\ f,\ A,\ F> \ \Rightarrow\ <I,\ f,\ A,\ F>$$

while The `Take unshared` instruction just pulls arguments off the stack into a new frame in the heap, with no checking for markers:

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

8.1 Discussion

Parent frame sharing separates the updating of shared expressions, which is done by the `Share Label` instruction, from the creation of argument frames, which is done by the `Take unshared` instruction. This allows each of these operations to be made more efficient.

The parent combinator only performs a test for a marker when there is a possibility that there will be one on the stack, and the child combinator can again simply transfer arguments from the stack to the heap using the computer's block memory move instruction because there are certain to be no markers interleaved among the arguments.

Table 2 shows how our TIM with parent frame sharing performs relative to our standard TIM. Once again, for each of the benchmark programs we measured both the decrease in execution time and the decrease in the amount of heap memory allocated. A negative number implies an increase.

	Time decrease %	Allocations decrease %
nfib	12	0
tak	-7	-20
primes	18	10
sort	12	-12
queens	-3	-10
folds	20	0
quad	33	37

Table 2: Improvements due to Parent Frame Sharing

These results show a clear decrease in the execution times of the programs which make use of partial application, *primes*, *folds* and *quad*. For the remaining programs the decrease is smaller and for two, *tak* and *queens*, there is even a small increase in execution time. This increase is due to the loss of η-reduction: to apply a function f to m arguments always requires $m+1$ supercombinator applications because we must (conservatively) assume that every partial application of f is shared, and so there could be a marker on the stack in each argument position. Each supercombinator application creates a new frame to represent a possibly shared partial application of f. Of course, if a partial application of f is unshared then the corresponding supercombinator is redundant. A sophisticated sharing analysis, such as [4], could be used to eliminate these redundant supercombinators.

As with child frame sharing, parent frame sharing eliminates the extra frame required to represent a partial application of the child combinator. In this case though, the partial application of the child shares the parent's frame rather than the child's frame. For programs which make extensive use of partial application, parent frame sharing produces an even larger performance increase than child frame sharing. For the remaining programs, parent frame sharing sometimes produces a small performance decrease, which could be overcome by using a sharing analysis. Parent frame sharing also enables a number of other optimisations to be performed, which we shall now describe.

9 Optimisations

9.1 Preserving Laziness

This optimisation should be regarded as essential, since without it laziness may be lost.

The proverbial 'observant reader' will have already noticed that parent frame sharing risks losing laziness whenever one of the arguments to the partial application of a supercombinator is itself an application. For example, consider the following partial application of a supercombinator C, of arity 4, in which the second argument is an application:

C 1 (*sqrt* 64) 3

Using parent frame sharing, the TIM code for this partial application is

```
L: Share label L
   Push num 3
   Push label sqrt64
   Push num 1
   Enter comb C
```

The loss of laziness occurs because every time the partial application is applied to a further argument by entering the code at L, the label 'sqrt64' is taken into a different frame, thus duplicating a redex. The standard TIM does not have this problem because a partial application always pushes indirections onto the stack to preserve laziness. To overcome this problem, we must introduce these indirections in another way.

Our solution is to introduce a local definition for each argument for which the TIM code generator would push a label. So the partial application of C given above is replaced by

let
    $x = sqrt\ 64$
in
    C 1 x 3

In our implementation, local definitions are implemented using a technique reminiscent of that used by the G-machine [7]. For each local definition we:

- allocate an extra argument (called a "hole") in the current frame;
- generate code for the right-hand-side, treating all references to the locally-bound name as references to the corresponding hole;
- push a label to the code for the right-hand-side into the hole.

The value of a local definition is accessed in the same way as a supercombinator argument. In particular, when it is pushed onto the stack, an indirection is used and so laziness is preserved.

9.2 Take Hoisting

We have already seen that transferring the responsibility for updating markers from the child combinator to the parent combinator provides substantial benefits. However, we can do even better if we transfer the responsibility for creating the frame from the child to the parent as well.

The optimisation is to simply "hoist" the `Take unshared` instruction from the beginning of the code for each combinator C to all of the places where C is entered. Thus, instead of the code

```
   Share if necessary
   push arguments to combinator C.
   Enter comb C;

C: Take unshared n;
   code for body of C.
```

we have

```
   Share if necessary
   push arguments to combinator C.
   Take unshared n;
   Enter comb C;

C: code for body of C.
```

Now the `Take unshared` instruction, which pulls arguments off the stack, comes immediately after the code which pushed them, and so it becomes easy for a peephole optimiser to replace many redundant stack accesses with code which pushes the arguments directly into the frame.

9.3 Local Sharing Analysis

A drawback of child frame sharing is that it is no longer possible to use the local sharing analysis described in the original paper. Parent frame sharing, however, does not suffer from this drawback.

When a child combinator of arity n is applied, its first $n{-}1$ arguments (at least) are always supplied by its parent combinator. When the parent pushed these arguments onto the stack it used indirections to its own frame in order to ensure sharing. These indirections can be freely manipulated by the child: there is no need for it to push extra markers onto the stack when entering them or additional indirections when pushing them. In the terminology of the original paper, the child combinator can treat its first $n-1$ arguments as `Unshared` (although this is now something of a misnomer).

10 Conclusions

We have presented two alternative ways of implementing partial applications in TIM, based on the new techniques of *child frame sharing* and *parent frame sharing*. Both techniques reduce the interpretive overhead of the `Take` instruction by separating the creation of argument frames from the updating of shared expressions. They also eliminate the extra frames required to represent partial applications by using existing frames to represent them.

For programs which make use of partial application, our machine with child frame sharing can produce a large performance gain over the standard machine. However, without the addition of a separate marker stack, it does not enable any further optimisations. Indeed, the local sharing analysis described in the original paper has actually been lost.

Parent frame sharing shows that a separate marker stack is unnecessary. For programs which make use of partial application, our machine with parent frame sharing produces larger performance gains over the standard machine than our machine with child frame sharing does. At the same time, it also enables several additional optimisations. The real advantage of parent frame sharing, however, is that it allows the results of a sharing analysis, such as [4], to be usefully applied in further improving performance.

Our preference, therefore, lies with parent frame sharing.

A Instruction Profiles

This appendix gives two instruction profiles for each of the benchmark programs described in section 4. The first profile, table 3, is for the abstract machine, which executes each TIM instruction in unit time, while the second profile, table 4, is for a concrete machine which executes each TIM instruction in the time required to to execute the assembly language instructions that implement it. All figures are percentages, and a "—" indicates that the compiler did not generate the given instruction.

B Compilation Rules

This appendix gives the rules for compiling a subset of FLIC into code for our TIM with parent frame sharing. The rules assume that the FLIC program has already been converted into supercombinators. The type of each object appearing in the compilation rules can be deduced from the letter representing it as follows:

	nfib	*tak*	*primes*	*sort*	*queens*	*folds*	*quad*	*average*
`Construct`	—	—	1.4	1.7	0.6	1.0	0.1	0.7
`Deconstruct`	—	—	1.3	1.6	1.4	1.8	0.0	0.9
`Enter arg`	8.0	11.7	9.8	10.0	10.3	7.9	14.9	10.4
`Enter arg'`	0.0	10.0	20.2	19.7	15.5	21.6	15.0	14.6
`Enter comb`	20.0	15.0	10.0	10.1	11.7	11.5	5.2	11.9
`Enter num`	2.0	—	—	—	0.0	—	—	1.0
`Operators`	12.0	6.1	2.9	1.9	4.7	0.9	5.0	4.8
`Partial`	0.0	1.7	4.1	1.7	2.1	2.6	9.8	3.1
`Push arg`	—	10.0	12.2	14.6	10.4	16.5	5.2	9.8
`Push comb`	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.0
`Push label`	22.0	22.8	14.1	16.7	17.4	10.7	10.1	16.3
`Push num`	10.0	1.7	1.4	0.2	0.5	0.1	5.0	2.7
`Self`	18.0	10.6	8.5	6.6	11.8	5.4	5.1	9.4
`Switch`	4.0	2.2	2.8	3.2	1.7	1.8	0.1	2.3
`Take`	4.0	8.3	11.1	10.0	11.3	18.2	24.7	12.5

Table 3: Instruction profile (abstract machine)

	nfib	*tak*	*primes*	*sort*	*queens*	*folds*	*quad*	*average*
`Construct`	—	—	0.9	1.1	0.4	0.8	0.0	0.5
`Deconstruct`	—	—	0.9	1.0	0.8	0.7	0.0	0.5
`Enter arg`	13.3	8.9	6.3	6.3	6.3	6.6	9.2	8.1
`Enter arg'`	0.0	14.0	23.9	22.9	17.3	31.8	17.0	18.1
`Enter comb`	5.5	1.9	1.1	1.1	1.2	1.5	0.5	1.8
`Enter num`	1.1	—	—	—	0.0	—	—	0.2
`Operators`	11.0	5.0	2.3	1.9	5.6	0.9	3.6	4.3
`Partial`	0.0	0.0	0.8	0.0	0.2	0.5	3.0	0.6
`Push arg`	—	1.4	6.6	7.0	5.4	11.0	2.7	4.9
`Push comb`	—	—	0.0	0.0	0.1	—	—	0.0
`Push label`	18.2	8.6	4.5	5.3	5.3	4.5	3.1	7.1
`Push num`	11.0	0.8	0.6	0.1	0.2	0.0	2.1	2.1
`Self`	19.9	23.4	17.4	27.0	24.6	10.6	9.5	18.9
`Switch`	7.7	2.0	2.3	2.6	1.4	2.0	0.0	2.6
`Take`	12.2	34.0	32.4	23.1	31.2	27.4	49.1	29.9

Table 4: Instruction profile (conventional machine)

c	is a combinator
e	is an arbitrary expression
l	is a unique new label
m	is ≥ 0
n	is number constant
x	is a variable

The "environment" r indicates where locally-bound names reside in the current frame.

The F Scheme (Function Definition)

The **F** scheme generates code for an entire combinator definition

$$\mathbf{F} \llbracket\ c\ x_1 \ldots x_m = e\ \rrbracket = \mathbf{C} \llbracket\ e\ \rrbracket\ [x_1 = 1, \ldots\ x_m = m]$$

The C Scheme (Compile Expression)

The $\mathbf{C} \llbracket\ e\ \rrbracket$ r scheme generates code for an expression e.

$\mathbf{C} \llbracket\ n\ \rrbracket\ \mathrm{r}$	$= \texttt{Push num } n\texttt{; Enter top}$
$\mathbf{C} \llbracket\ c\ e_1 \ldots e_m\ \rrbracket\ \mathrm{r}$	$= \texttt{Label } l_0\texttt{; Share label } l_0\texttt{; } \mathbf{C} \llbracket\ c\ e_1 \ldots e_m\ \rrbracket\ \mathrm{r}$ (if arity $c = m{+}1$)
$\mathbf{C} \llbracket\ c\ e_1 \ldots e_m\ \rrbracket\ \mathrm{r}$	$= \mathbf{L} \llbracket\ e_m\ \rrbracket\ \mathrm{r}; \ldots \mathbf{L} \llbracket\ e_1\ \rrbracket\ \mathrm{r};\ \mathbf{T} \llbracket\ \mathrm{c}\ \rrbracket\texttt{; Enter comb } c$
$\mathbf{C} \llbracket\ x\ e_1 \ldots e_m\ \rrbracket\ \mathrm{r}$	$= \mathbf{L} \llbracket\ e_m\ \rrbracket\ \mathrm{r}; \ldots \mathbf{L} \llbracket\ e_1\ \rrbracket\ \mathrm{r}\texttt{; Enter arg } \mathrm{r}(x)$
$\mathbf{C} \llbracket\ \mathsf{INT+}\ e_1\ e_2\ \rrbracket\ \mathrm{r}$	$= \mathbf{E} \llbracket\ e_2\ \rrbracket\ \mathrm{r};\ \mathbf{E} \llbracket\ e_1\ \rrbracket\ \mathrm{r}\texttt{; Enter comb } \mathrm{Plus}$
$\mathbf{C} \llbracket\ \mathsf{PACK}\text{-}a\text{-}t\ e_1 \ldots e_m\ \rrbracket\ \mathrm{r}$	$= \mathbf{L} \llbracket\ e_m\ \rrbracket\ \mathrm{r}; \ldots \mathbf{L} \llbracket\ e_1\ \rrbracket\ \mathrm{r}\texttt{; Construct } a\ t\texttt{; Enter top}$
$\mathbf{C} \llbracket\ \mathsf{UNPACK}\text{-}n\ c\ e\ \rrbracket\ \mathrm{r}$	$= \mathbf{E} \llbracket\ e\ \rrbracket\ \mathrm{r}\texttt{; Deconstruct } n\texttt{; } \mathbf{C} \llbracket\ c\ \rrbracket\ \mathrm{r}$
$\mathbf{C} \llbracket\ \mathsf{CASE}\text{-}m\ c_1 \ldots c_m\ x\ \rrbracket\ \mathrm{r}$	$= \mathbf{E} \llbracket\ x\ \rrbracket\ \mathrm{r}\texttt{; Switch } (l_1 \ldots l_m)$ (and $\texttt{Label } l_i\texttt{; } \mathbf{C} \llbracket\ c_i\ \rrbracket\ \mathrm{r}$)
$\mathbf{C} \llbracket\ \mathsf{LET}\ v\ e\ \rrbracket\ \mathrm{r}$	$= \mathbf{V} \llbracket\ v\ \rrbracket\ \mathrm{r};\ \mathbf{C} \llbracket\ e\ \rrbracket\ r'$ (where $r' = \mathbf{X} \llbracket\ v\ \rrbracket\ \mathrm{r}$)
$\mathbf{C} \llbracket\ \mathsf{LETREC}\ v\ e\ \rrbracket\ \mathrm{r}$	$= \mathbf{V} \llbracket\ v\ \rrbracket\ r';\ \mathbf{C} \llbracket\ e\ \rrbracket\ r'$ (where $r' = \mathbf{X} \llbracket\ v\ \rrbracket\ \mathrm{r}$)

The E Scheme (Evaluate)

$\mathbf{E} \llbracket\ e\ \rrbracket$ r generates code which evaluates the expression e.

$\mathbf{E} \llbracket\ n\ \rrbracket\ \mathrm{r} = \texttt{Push num } n$

$\mathbf{E} \llbracket\ e\ \rrbracket\ \mathrm{r} = \texttt{Push label } l_1\texttt{; } \mathbf{C} \llbracket\ e\ \rrbracket\ \mathrm{r}\texttt{; Label } l_1$

The L Scheme (Label)

$\mathbf{L}[\![e]\!]$ r generates code for an expression e, and leaves a label for this code at the top of the stack.

$\mathbf{L}[\![n]\!]$ r = `Push num` n
$\mathbf{L}[\![c]\!]$ r = `Push comb` c
$\mathbf{L}[\![x]\!]$ r = `Push arg` r(x)
$\mathbf{L}[\![e]\!]$ r = `Push label` l_1;
(and `Label` l_1; $\mathbf{C}[\![e]\!]$ r)

The X and V Schemes (Local Definitions)

These two schemes are used to handle the local definitions in a LET(REC) expression.

$\mathbf{X}[\![(v_1 \ldots v_m)\,(e_1 \ldots e_m)]\!]$ r = r$[v_1 = n{+}1 \ldots v_m = n{+}m]$
(where n = length r)
$\mathbf{V}[\![(v_1 \ldots v_m)\,(e_1 \ldots e_m)]\!]$ r = `Push into arg` $(n{+}1)$ l_1;...`Push into arg` $(n{+}m)$ l_m
(and `Label` l_1; $\mathbf{C}[\![e_1]\!]$ r;...`Label` l_m; $\mathbf{C}[\![e_m]\!]$ r
and n = length r)

The T Scheme (Take)

$\mathbf{T}[\![c]\!]$ = `Take unshared` n

where n is the arity of c.

C State Transition Rules

The formal description of our TIM with parent frame sharing appears below:

$<$[`Construct` n t; I], f_1, $(a_1, \ldots, a_n, A)$, $F$$>$
$\Rightarrow <I, f_1, (<$ `Struct`$, f >, A), F\,[f \mapsto (a_1, \ldots, a_n, t)]>$

$<$[`Deconstruct` n; I], f_1, $(<$`Struct`$,f>, A)$, $F\,[f \mapsto (a_1, \ldots, a_n, t)]>$
$\Rightarrow <I, f_1, (a_1, \ldots, a_n, A), F>$

$<[\texttt{Enter comb}\ c], f, A, F> \Rightarrow <c, f, A, F>$
$<[\texttt{Enter arg}\ n], f, A, F\,[f \mapsto (\ldots, <c_n, f_n>, \ldots)]> \Rightarrow <c_n, f_n, (\le f, n \ge, A), F>$
$<[\texttt{Enter top}], f, (<c_1, f_1>, A), F> \Rightarrow <c_1, f_1, A, F>$
$<[\texttt{Enter label}\ l], f, A, F> \Rightarrow <l, f, A, F>$
$<[\texttt{Num}], f, (<c_1, f_1>, A), F> \Rightarrow <c_1, f_1, (<\texttt{Num}, f>, A), F>$
$<[\texttt{Push arg}\ n; I], f, F, A> \Rightarrow <I, f, (<[\texttt{Enter arg'}\ n], f>, A), F>$
$<[\texttt{Push num}\ n; I], f, A, F> \Rightarrow <I, f, (<\texttt{Num}, f>, A), F>$
$<[\texttt{Push label}\ l; I], f, A, F> \Rightarrow <I, f, (<l, f>, A), F>$
$<[\texttt{Push comb}, c; I], f, A, F> \Rightarrow <I, f, (<c, 0>, A), F>$
$<[\texttt{Share label}\ l; I], f_1, (\le f, m \ge, A), F\,[f \mapsto (\ldots, a_m, \ldots)]>$
$\qquad\qquad \Rightarrow <[\texttt{Share label}\ l; I], f_1, A, F\,[f \mapsto (\ldots, <l, f_1>, \ldots)]>$

$<[\texttt{Share label}\ l; I], f, A, F> \Rightarrow <I, f, A, F>$
$<[\texttt{Switch}\ (l_1, l_2, \ldots, l_m)], f_1, (<\texttt{Struct}, f>, A), F\,[f \mapsto (a_1, \ldots, a_n, t)]>$
$\qquad\qquad \Rightarrow <l_t, f_1, A, F>$

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

References

[1] G. Argo. Improving the Three Instruction Machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 100–115. ACM Press, September 1989.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[3] J. Fairbairn and S. Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, pages 34–45. Springer-Verlag, September 1987. LNCS 274.

[4] B. Goldberg. Detecting sharing of partial applications in functional programs. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, pages 408–425. Springer-Verlag, September 1987. LNCS 274.

[5] R. J. M. Hughes. Super-combinators: A New Implementation Method for Applicative Languages. In *1982 ACM Symposium on LISP and Functional Programming*, pages 1–10, August 1982.

[6] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, September 1985. LNCS 201.

[7] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, S-412 96 Göteborg, February 1987.

[8] S. L. Peyton Jones and D. Lester. *Iplementing Functional Languages: A Tutorial.* Prentice-Hall, 1992.

[9] S. L. Peyton Jones. FLIC – a functional language intermediate code. *ACM SIG-PLAN Notices*, 23(8):30–48, August 1988.

[10] S. Wray and J. Fairbairn. Non-strict Languages — Programming and Implementation. *Computer Journal*, 32(2):142–151, April 1989.