The Categorical Multi-Combinator Machine: CMCM

Simon Thompson
Computing Laboratory, University of Kent at Canterbury,U.K.
(sjt@ukc.ac.uk)

Rafael Lins
Department of Informatics, Federal University of Pernambuco, Brazil

INTRODUCTION

Implementations of functional programming languages can take a number of different forms; the first implementations used the SECD machine of Landin [Lan], which is a generalisation of the abstract machine underlying implementations of imperative programming languages like Pascal. Making the implementation *lazy* can be done with some cost in efficiency (details can be found in Henderson's book [Hen]), but a more natural way of proceeding is provided by an implementation which uses rewriting of source-level expressions. Such implementations were pioneered by Wadsworth's work on graph-reduction for the λ-calculus [Wad], but only achieved reasonable performance after Turner's observation that the translation of programs into a variable-free, or *combinator*, form allowed for more efficient execution. Turner's work on the graph reduction of combinator code (together with that of Abdali [Abd]) was followed up by a number of researchers, including Hughes ([Hug]), Johnsson ([Joh]) and Lins ([Lins1) each of whom developed different systems of combinators, and algorithms for translating source programs into combinator form. Each of these implementations can be characterised as *interpretive*, as the reduction and transformation of the combinator graph can be seen as interpretation of the graph, and the obvious question raised by this is that of whether a true *compiler* for lazy functional programs can be produced. The first attempt to do this was the G-machine ('G' for graph) of Augustsson and Johnsson [Joh] in which many of the manipulations of the graph are transformed into abstract machine instructions, which can be rendered into machine code. Another approach is given by the Three Instruction Machine, or TIM [FaiWr], which comes full circle in using closures to represent the expressions under evaluation.

In this paper we introduce another abstract machine, Categorical Multi-Combinator Machine, (CMCM). In this paper we give a thoroughgoing introduction to the machine, in particular as far as the discussion of *sharing* of computational information is concerned. The approaches of both TIM and the CMCM depend upon the source code being λ-lifted before the translation takes place. This transformation, discovered by Johnsson [Joh], independently of related work by Hughes into supercombinators, has the effect of making *flat* the environments in which function bodies are interpreted. The transformation only came to light with the work on combinators, mentioned above, so perhaps reflecting the epigraph.
In another paper, [LiTh], we discuss in detail the close relationship between the TIM and the CMCM.

CATEGORICAL MULTI-COMBINATORS

The second author in his thesis [Lins1] introduces the system of categorical multi-combinators, which are based on Curien's categorical combinators [Cur], which in turn have their foundation in the theory of Cartesian Closed categories [Sco]. The major innovation of the multi-combinators is that a number of β-reductions can be performed in a single step of rewriting, rather than in a sequence of such steps. This offers the possibility of increasing the efficiency of a rewriting implementation considerably; a similar idea has been discussed by the implementors of the G-machine [Joh2].
The syntax of categorical multi-combinators consists of variables, $n, \ldots$ the constants P, $\circ$, L^n (where the superscript is a natural number) which are combined together by application, written as juxtaposition. The usual syntactic shorthand of omitting brackets in left-associated applications is adopted.
The rewriting laws which the combinators obey are as follows:

(M^*1)	$\circ n\,(P\,x_m \cdots x_1\,x_0)$	$\Rightarrow x_n$	
(M^*2)	$\circ\,(x_0 \ \ldots\ x_n)\ y$	$\Rightarrow (\circ x_0\ y)\ \ldots\ (\circ x_n\ y)$	
(M^*3)	$L^n(y)\,x_0\ \ldots\ x_z$	$\Rightarrow\ \circ y\ (P\ x_0 \ldots x_n)\,x_{n+1} \cdots\ x_z$	
(M^*4)	$\circ\ c\ y$	$\Rightarrow c$	(c constant or $L^n(y)$)

where we use the symbol '⇒' for 'rewrites to'. The notation $(P\ldots)$ is for a *multi-pair,* and the L^n is a $n+1$-ary abstraction. The first rule represents the selection of the value x_n of a variable (represented by a non-negative integer n) from a block (or environment or frame). In general an expression is combined with its frame information by means of the *composition* operator, $\circ$. The second law shows the distribution of the environment information y through a complex application $(x_0\ \ldots\ x_n)$. The third rule represents the formation of a frame which takes place when a function application is evaluated, and the final rule embodies the fact that neither constants nor lambda expressions (i.e. functions) need any environment information for their interpretation.

The four rules above can be combined into two rules thus:

$$(M1) \qquad \circ(x_0\ \ldots\ x_n)\ (P\,y_m \cdots y_1\,y_0) \Rightarrow (x'_0\ \ldots\ x'_n)$$

where

$$x'_i = \begin{cases} x_i & x_i \text{ constant or } L^n(y) \\ y_k & x_i \text{ is variable } k \\ \circ x_i\ (P\,y_m \cdots y_1\,y_0) & \text{otherwise} \end{cases}$$

$$
\begin{array}{llll}
(M2) & L^n(y)\,x_0 \ldots x_z & & \\
 & \Rightarrow\ {}^\circ y\ x_{n+1} \ldots x_z & & y \text{ constant} \\
 & \Rightarrow\ {}^\circ x_{n-k}\ x_{n+1} \ldots x_z & & y \text{ is variable } k \\
 & \Rightarrow\ {}^\circ y\ (P\ x_0 \ldots x_n)\, x_{n+1} \ldots x_z & & \text{otherwise}
\end{array}
$$

General λ-expressions are compiled into categorical multi-combinators in two stages. First the expressions are λ-lifted, removing non-local references from function bodies; variables are then replaced by their distance from their binding λ (the distance being the number of intervening λs), and blocks of $n+1$ λs are replaced by L^n.

For example, we compile the combinator G defined by

$$G = \lambda a.\lambda b.\lambda c.\ (ab)(ac)$$

into the code

$$L^2((21)(20))$$

The identity function, I, or $\lambda x.x$, compiles into the expression $L^0(0)$. We can now follow the evaluation of an expression using the rewriting rules above.
The expression

$$G\ (II)\, I\, I$$

compiles to

$$L^2((21)(20)) \quad (L^0(0)L^0(0)) \quad L^0(0) \quad L^0(0)$$

This will rewrite thus:

$L^2((21)(20)) \quad (L^0(0)L^0(0)) \quad L^0(0) \quad L^0(0) \Rightarrow$ by (M2.3)

(where we use (M2.*i*) to denote the *i*th clause of rule (M2))

$\circ\ ((21)(20))\ \ (P\ \ (L^0(0)L^0(0))\ L^0(0)\ L^0(0)\,) \Rightarrow$ by (M1)

$(\circ(21)\ \ (P\ \ (L^0(0)L^0(0))\ L^0(0)\ L^0(0)\,)\ \ \ (\circ(20)\ \ (P\ \ (L^0(0)L^0(0))\ L^0(0)\ L^0(0)))$

now we rewrite the redex in the first bracket using the rule (M1), noting that this is a situation in which one of the optimisations applies.

$(\ (L^0(0)L^0(0))\ L^0(0)\,)\ (\circ(20)\,(P\ \ (L^0(0)L^0(0))\ L^0(0)\ L^0(0)\,)) \Rightarrow$ by (M2.1)

$(\,L^0(0)\ L^0(0)\,)\ (\circ(20)\,(P\ \ (L^0(0)L^0(0))\ L^0(0)\ L^0(0)\,)) \Rightarrow$ by (M2.1)

$L^0(0)\ (\circ(20)\,(P\ \ (L^0(0)L^0(0))\ L^0(0)\ L^0(0)\,)\,) \Rightarrow$ by (M2.1)

$(\circ(20)\,(P\ \ (L^0(0)L^0(0))\ L^0(0)\ L^0(0)\,))$

Note that in the last three cases we have used the optimised form of β-reduction, and we have thus avoided the formation of a frame each time. Note also that the redex

$$(L^0(0)L^0(0))$$

which appears in the frame is reduced to $L^0(0)$ in the first of the three steps above. If we use graph reduction, then this redex would be updated to have this new value in the graph, and thus in the frame. Resuming evaluation, assuming the update, we have,

$(\circ(20)\,(P\ L^0(0)\ L^0(0)\ L^0(0)\,)\,) \Rightarrow$ by (M1)

$(\ L^0(0)\ L^0(0)\ ) \Rightarrow$ by (M2.1)

$L^0(0)$

at which point evaluation halts.

EXTENDING THE COMBINATOR SET

We saw in the previous section that we could modify the rewriting rules, from the set M* to the set M by making the observation that a constant or a variable in the body of a lambda abstraction behaves in a particularly simple way. We can indeed observe this at compile time, and so we acknowledge this by defining two new combinators which have the appropriate effect in these two cases.

$$K^n_y\ x_0\ \dots\ x_z \quad\Rightarrow\quad y\ x_{n+1} \cdots\ x_z$$

$$V^n_k\ x_0\ \dots\ x_z \quad\Rightarrow\quad x_{n\text{-}k}\ x_{n+1} \cdots\ x_z$$

EXAMINING THE EFFECT OF THE β-REDUCTION RULE

What is the effect of the general form of the rule (M2)?

$$L^n(\,y)\,x_0\ \dots\ x_z \quad\Rightarrow\quad \circ y\ (P\ x_0 \dots x_n\,)\,x_{n+1} \cdots\ x_z$$

First we create a composition, in which the body of the *L*-expression, *y*, is combined with the appropriate frame, $(P\ \ x_0 \dots x_n\ )$; after this the rule (M1) will cause this information about the values of variable to be distributed through the body of the abstraction. We can see this in action in the example of *S' (II) I I* we examined above.

We can see that an alternative strategy suggests itself when we perform the evaluation of the body of a lambda-abstraction: we can preserve the frame information separately, and perform lookups into this information when and if it is necessary. Examining the example we looked at earlier, we can say that

$$L^2((21)(20))\ \ (V^0_0\,V^0_0)\ \ V^0_0\ V^0_0$$

will rewrite thus:

expression	**frame**
$(21)(20)$	$(P\ \ (V^0_0\,V^0_0)\ \ V^0_0\ V^0_0)$

where we picture the expression and the current frame separately. We only fetch values from the frame when necessary.

$(\ (V^0_0\,V^0_0)\ \ 1)\,(20) \Rightarrow$

$(\,V^0_0\ 1)\ \ (20)\ \Rightarrow$

$1\ \ (20) \Rightarrow$

At this point we again have to fetch a value from the frame,

$$V^0_0(20) \Rightarrow (20)$$

and so we continue. Note that under this procedure, we *do not* update the frame with the reduced value of the variable 0, which is reduced from $(V^0_0 V^0_0)$ to V^0_0. We will return to the issue of sharing later in this paper. Obviously we do not simply create a single frame during the course of an evaluation. How are we to maintain a structure of frames, corresponding to the various redexes which we reduce during the course of an evaluation?

A STACK OF FRAMES

The natural structure for the frames we generate is a *stack*. Each time we begin to evaluate a function application, we *push* onto the frame or *multi-pair stack* the frame formed from the arguments of the application. If we deal with pure, untyped λ-expressions, without any ground values like numbers, then we find that the result of each function call is another expression, the reduction of (the leftmost-outermost redex of) which gives rise itself to another function call, creating another frame. In other words, we seem not to need ever to *pop* a frame from the stack. Why then do we need to preserve the earlier frames we formed, rather than simply keeping track of the current frame? There are two reasons for this. The first is in the case when we add base values, which we explain below, and the second is to keep track of information about free variables. Consider the following situation:

expression	**frame stack**
$L^2((21)(20))\ \ 1\ \ V^0_0\ \ 0$	$(P\ \ (V^0_0 V^0_0)\ \ V^0_0\ V^0_0)$

This calls for the formation of a new frame,

$$(P\ \ 1\ \ V^0_0\ 0)$$

but in this frame we have the free variables 1 and 0 which are references to the previous frame. Two possibilities suggest themselves

- We can resolve the references of any free variables at the time that we form the frame. This can be done either by copying the entry in the appropriate frame, or by copying a *pointer* to the (position in the) frame. One difficulty with the latter solution is that it allows the possibility that frames may have a longer extent than suggested by the stack discipline.
- In the situation outlined above, the situation is simple: to find the values of the free variables in a frame, we only have to look in the frame *below* that frame in the stack. For instance, our stack will now be

$$(P\ \ 1\ \ V^0_0\ 0)\ ,\ (P\ (V^0_0 V^0_0)\ \ V^0_0\ V^0_0)$$

with the values of the variables 1 and 0 to be found in the frame below the top frame.

Are the values of free variables always to be found one frame down in the stack? Let us consider a variant of the previous example:

expression	**frame stack**
$L^2((21)(20))\ \ 1\ \ V^0_0\ \ 0\ \ 2$	$(P\ (V^0_0 V^0_0)\ \ V^0_0\ V^0_0)$

Formation of a frame produces the following situation:

$$(21)\ \ (20)\ \ 2 \qquad\qquad (P\ \ 1\ \ V^0_0\ 0\,)\quad (P\ (V^0_0 V^0_0)\ \ V^0_0\ V^0_0)$$

We now have a problem. The sub-expression *(21) (20)* refers to the newly formed frame, whereas the final 2 refers to the previous frame. This is because we have included no information about the extent of the function body in the expression we produced. We can do this by placing a mark in the expression, *F*, which delimits the extent of the function body, thus:

$$(21)\ \ (20)\ \ F\ \ 2 \qquad\qquad (P\ \ 1\ \ V^0_0\ 0\,)\quad (P\ (V^0_0 V^0_0)\ \ V^0_0\ V^0_0)$$

The presence of the *F* shows that any references to its right are not to the top frame but to the frame 1 down in the stack. Similarly, if variable has *n* *F*s before it, lookups should be made *n* frames down in the stack. Taking an analogous problem, now, we see the difficulty in frame formation.

expression	**frame stack**
$L^1(\ldots)\ \ (20)\ \ F\ \ 2$	$(P\,1\ \ V^0_0\ 0\,)\quad (P\ (V^0_0 V^0_0)\ \ V^0_0\ V^0_0)$

This calls for the formation of a frame. First we note that *F* is obviously not an argument, so that we don't incorporate it into a frame. We do need to note that the references in the first argument are to the top frame, whilst those in the second, to the right of the *F*, are to the next to top. We can form the frame in either of the ways indicated above, *i.e.*,

- Resolve the references of any free variables at the time that we form the frame.
- Annotate the position in the frame under formation with the number of *F*s that occur to the left of the corresponding argument in the expression. In fact we could do this even more easily: we could copy the Fs into the frame as well as leaving them in the code. In the example above this would mean the new frame would have the form

  $$(P\ \ (20)\ \ F\ \ 2\,)$$

  (In this case we must make sure we do not confuse the copied *F* with an in the code — we therefore mark them with a prime, thus:

  $$(P\ \ (20)\ \ F'\ \ 2\,)$$

Either of these strategies is sufficient to give the correct information for the variable bindings.

STRUCTURED EXPRESSIONS → A SEQUENCE OF INSTRUCTIONS

Our next design decision is to turn the expression under evaluation into a *flat* sequence of *code* instructions. We turn the expressions we are familiar with into a sequence of code instructions thus:

- Left-associated applications are simply treated as a sequence of instructions;
- other (bracketed) applications are replaced by labels;
- bodies of abstractions are labelled.

Considering the example we saw above, of, *S' (II) I I* which produced the code

$$L^2((21)(20))\ \ (V^0_0\, V^0_0)\ \ V^0_0\ V^0_0$$

this code would be replaced by

$$\begin{array}{lll} fun & \to & L^2(body)\ \ l_2\ \ V^0_0\ V^0_0 \\ body & \to & 2\,1\,l_1 \\ l_1 & \to & 2\,0 \\ l_2 & \to & V^0_0\ V^0_0 \end{array}$$

How do we obey these instructions? The behaviour of the instructions L^n, V^n_m K^n_y is exactly as before. The occurrence of a label as the first instruction results in the code for the label replacing the label itself. This is can obviously be implemented by a jump to the code labelled, with a *return address* held on a stack of continuations, for example. This is after all exactly what the code to the right of the first instruction represents: the *continuation* of the computation.

There is one other effect which results from us adding labels to the code. When we move some code into a frame, we have to resolve any references it might contain, in the form of free variables. The same comment will apply to a piece of code which contains a label which refers to code containing free variables such as the labels *body* and l_1 which appear in the example above.

Two alternatives for this sort of reference resolution were mentioned above. In the second, we made a note of the frame referenced, but in the former, we simply replaced the reference by its referent. The second (former) possibility is still available here, but the latter causes difficulty. We may have a number of references to a particular label, with different instantiations of the variables, and so we *cannot* change the code referenced by the label. We have instead to create a new instance of the labelled code, instantiating the variables in the appropriate way. This imposes an overhead, comparable to the overhead of *template instantiation* in other implementations. It also means that the code is no longer static: we generate code as execution proceeds.

RECURSION

Once we have labels then we are able to treat recursion in a completely straightforward way: recursive functions are compiled into functions with circular definitions, that is functions which refer to their own labels in their code.

A STACK OF BASIC VALUES

If we want to add to our system values of base type, such as integers, characters and so on, we can do this by means of a stack which will store the intermediate results of calculations, exactly as is done in the postfix expression evaluators we are familiar with from conventional machines. This means that values are stored in two places in the machine. Basic values reside on the *ground stack* and functional values are represented by the state of the expression and multi-pair stack.
In detail, we handle the base types thus:

- If the first instruction in the sequence of instructions is a number it is transferred to the ground stack;
- If the instruction is an operation, like +, it is applied to the appropriate number of arguments (in the case of +, two) taken from the ground stack, with the result replaced on the ground stack.

A *conditional* expression like

$$\textit{if } e \textit{ then } g \textit{ else } h$$

is treated in a similar way, compiling to the code sequence

$$[[e]] \;\; C \;\; l_1 \; l_2$$

where $[[e]]$ is the code compiled for e, and l_1, l_2 label the code for g and h. The instruction C will branch to whichever of l_1, l_2 labels the code to be chosen.This is one point at which the machine is different from TIM, in which there is no explicit stack of base values, rather they use the '*self*' combinator.

SUMMARY

We can summarise the behaviour of the machine thus. Note first that there are two sorts of instruction. The operators and constants affect the ground stack but not the multi-pair stack; the others do the reverse. We first summarise the former. Note that we use the convention that **boldface** symbols like '**k**' represent constants, and lightface symbols like '*k*' are variables.

code	**multi-pair stack**	**ground stack**		**code**	**multi-pair stack**	**ground stack**
k …	……	…	→	…	……	**k** …
+ …	……	**p q**	→	…	……	(**p**+**q**) …
C x y …	……	**True**…	→	x …	……	…
C x y …	……	**False**…	→	y …	……	…
$K^n_{\mathbf{k}}$ …	……	…	→	…	……	**k** …

In summarising the other instructions we omit the ground stack which is not affected by their action.

code	**multi-pair stack**		**code**	**multi-pair stack**
$L^n(y)\, x_0 \ldots x_z$	$\ldots\ldots$	$\rightarrow$	$y\ F\ x_{n+1} \ldots x_z$	$(P\ x_0 \ldots x_n)\ldots\ldots$
$V_k^n\ x_0 \ldots x_z$	$\ldots\ldots$	$\rightarrow$	$x_{n-k}\ x_{n+1} \ldots x_z$	$\ldots\ldots$
$y\ z_0 \ldots z_l$	$(P \ldots x_i \ldots)\ldots\ldots$	$\rightarrow$	$x_{n-y}\ z_0 \ldots z_l$	$(P \ldots x_i \ldots)\ldots\ldots$
$F \ldots$	$(P \ldots x_i \ldots)\ldots\ldots$	$\rightarrow$	$\ldots$	$\ldots\ldots$

Note that in the execution of the first instruction we treat the mark F in a different way. Fs are left on the stack, and are not moved into frames. For example,

$$L^2(y)\, x_0\, F\, x_1\, x_2 \ldots \quad \ldots\ldots \quad \rightarrow \quad y\ F\, F\, x_3 \ldots \quad (P\ x_0 \ldots x_2)\ldots\ldots$$

The first *F* comes from the entry of the function body, *y*, and the second is that which originally lay between x_0 and x_1.

EXAMPLE

In this section we look at an example of the operation of the machine. The expression

$$(\lambda x.\lambda y.\lambda z.(xy+xz))((\lambda v \lambda u.\lambda w.v+w)\ (\mathbf{5+2})\ \mathbf{0})\ \mathbf{3}\ \mathbf{7}$$

will compile into the following code:

$l_1 \rightarrow L^2\,(body_1)\ ex_1\ \mathbf{3}\,\mathbf{7}$

$body_1 \rightarrow 2\ 1\ ex_2\ +$

$ex_1 \rightarrow L^2\,(body_2)\ ex_3\ \mathbf{0}$

$ex_2 \rightarrow 2\,0$

$body_2 \rightarrow 2\ 0\ +$

$ex_3 \rightarrow \mathbf{5}\ \mathbf{2}\ \mathbf{+}$

and execution will proceed thus:

code	**multi-pair stack**	**ground**
$L^2\,(body_1)\ ex_1\ \mathbf{3}\,\mathbf{7}$	..	..
$body_1\ \ F$	$(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..
$2\ 1\ ex_2 + F$	$(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..
$ex_1\ 1\ ex_2 + F$	$(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..
$L^2\,(body_2)\ ex_3\ \mathbf{0}\ 1\ ex_2 + F$	$(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..
$body_2\ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\,(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..

Note that in the frame we have just formed, we have a free variable, *1*; this refers to the frame below. We have not recorded this explicitly, but we keep informal track of the fact, and in the next sequence of moves, when we look up the value, we are careful to take it from the right frame.

$2\ 0 +\ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\,(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..
$ex_3\ 0 +\ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\,(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..
$\mathbf{5}\ \mathbf{2}\ \mathbf{+}\ 0 +\ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\,(P\ ex_1\ \mathbf{3}\,\mathbf{7})$	..

Now we have to place these constant values on the ground stack:

$\mathbf{2} + 0 + \ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\ (P\ ex_1\ \mathbf{3}\ 7)$	**5**
$+ 0 + \ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\ (P\ ex_1\ \mathbf{3}\ 7)$	**2 5**

and then we add them

$0 + \ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\ (P\ ex_1\ \mathbf{3}\ 7)$	**7**

Here we have to fetch the value from the previous frame:

$\mathbf{3} + \ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\ (P\ ex_1\ \mathbf{3}\ 7)$	**7**
$+ \ F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\ (P\ ex_1\ \mathbf{3}\ 7)$	**3 7**
$F\ ex_2 + F$	$(P\ ex_3\ \mathbf{0}\ 1)\ (P\ ex_1\ \mathbf{3}\ 7)$	**10**
$ex_2 + F$	$(P\ ex_1\ \mathbf{3}\ 7)$	**10**
$2\ 0 + F$	$(P\ ex_1\ \mathbf{3}\ 7)$	**10**
$ex_1\ 0 + F$	$(P\ ex_1\ \mathbf{3}\ 7)$	**10**
$L^2\ (body_2)\ \ ex_3\ \mathbf{0}\ 0 + F$	$(P\ ex_1\ \mathbf{3}\ 7)$	**10**
$body_2\ \ F\ + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**10**

Note that in the frame we have just formed, we again have a free variable, 0.

$2\ 0 + \ F + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**10**
$ex_3\ 0 + \ F\ + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**10**
$\mathbf{5}\ \mathbf{2} + 0 + \ F\ + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**10**

Now we have to place these constant values on the ground stack:

$\mathbf{2} + 0 + \ F\ + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**5 10**
$+ 0 + \ F + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**2 5 10**
$0 + \ F + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**7 10**

Here we have to fetch the value from the previous frame:

$\mathbf{7} + \ F\ + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**7 10**
$+ \ F\ + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**7 7 10**
$F + F$	$(P\ ex_3\ \mathbf{0}\ 0)\ (P\ ex_1\ \mathbf{3}\ 7)$	**14 10**
$+ F$	$(P\ ex_1\ \mathbf{3}\ 7)$	**14 10**
F	$(P\ ex_1\ \mathbf{3}\ 7)$	**24**
..	..	**24**

The example exhibits the main features of the machine, including the fact that it is not lazy. The computation of **5**+**2**, labelled ex_3 is repeated, even though it could be saved and performed only once. In the next section we look at the means by which we can add sharing to the machine, making it lazy.

SHARING

The only feature which the machine described so far lacks is the ability to *share* the results of computations. Graph reduction of λ-lifted expressions [Joh1] ensures that all arguments which become duplicated will have their computation shared — this is called *lazy* evaluation. There remains the possiblity that there are sub-expressions of shared arguments whose computation is not shared under this transformation. One way to achieve this full laziness in a graph reduction machine for categorical multi-combinators is to perform the *supercombinator* transformation of maximal free expressions (*mfe*s) before compiation.

A disadvantage of this compilation scheme is that the granularity of computation is thereby much reduced.
In contrast to graph reduction, we ensure this full laziness by a run-time strategy, inspired by the mechanism in TIM.
As an example of a situation in which we have to update more than the arguments which are explicitly repeated consider the expression

$$(\lambda f.\lambda x.\lambda y.fx+fy)((\lambda a.\lambda q.(a+q))\ (\mathbf{3}+\mathbf{5}))\ \mathbf{7}\ \mathbf{9}$$

even though the argument a is not repeated in the expression $(a+q)$ the result of the evaluation

$$\lambda q.\ (\mathbf{3}+\mathbf{5})+q$$

is a shared argument of the function $(\lambda f.\lambda x.\lambda y.fx+fy)$. Because of this we need to update the slot containing the argument a, which will contain the value $\mathbf{8}$, as well as the slot containing the argument f itself.

We therefore need to decide what mechanism we will use to achieve the sharing we require. A number of options present themselves. We might decide to label every occurrence of every variable as requiring updating when it is evaluated; we might perform some compile-time analysis to decide which occurrences of which variable could ever be shared (there is a lot of work in this area: see, for instance, [Gol] and also [AbHa] for a survey of the general techniques involved); the final option which we adopt here is to mark all occurrences of repeated variables, distributing marks to further variables as evaluation proceeds.
The mechanism for sharing is relatively straightforward. We have to arrange that it works for the two kinds of value we have in the machine. These are

- Basic values, like numbers, etc. These are transferred to the ground stack once evaluated, so it should be simple to arrange that the appropriate frame slot is updated at the appropriate time.
  If we think of placing a mark in the code at the end of an expression, carrying the information about where the argument lies in a frame, then an update should take place when that mark reaches the head of the code. We update the slot mentioned with the value we just transferred to the ground stack, and carry on.
- The other kind of value is functional.

Functions are represented in CMCM by portions of code, together with sufficient frame information to interpret the free variables. For instance, the function $\lambda x.x$ is represented by the code $V^{0}_{\ \ 0}$ and the function

$$\lambda x\ .\ (\mathbf{13}+x)$$

may be represented in a number of different ways, depending upon the way that it is generated during evaluation. The simplest representation is by the code

$$L^{0}\ (body)$$

where *body* labels the code $(\mathbf{13}\ 0\ +)$ but it may also be represented by the code for the following λ-expression, where b is bound to $\mathbf{13}$.

$$\lambda a.\lambda x.(a+x)\ \ b$$

In terms of categorical combinator code, this might be

$$L^{1}\ (body)\ \ 0$$

where *body* labels the code *(*1 *0 +)* and the current frame binds the variable *0* to ***13***.

What is meant by *sufficient* frame information? We may find that the code part of a function representation contains one or more *F*s. In such a case, we have to refer to more than one frame to gain enough information. In fact, when we perform this update we replace all free variables with their values. This seems an expensive option, but will not be so if we make sure also that *every value in a frame is in fact a label (or pointer).* The effect of this update then is just to replace an index into a frame by a pointer.

An alternative method is provided by grouping together the variables and (pointers to) the frames from which they come. This forms what looks like a closure, and makes the machine look closer to the TIM.

When do we perform an update to a functional value? In terms of the λ-calculus we perform an update when a function reaches weak head normal form:

$$\lambda x.e$$

To do any further reduction we need to supply the expression with another argument value; in other words we have a result, or *canonical form,* when we find a function with too few arguments.

How is this manifested in the CMCM? We will have a marker in the code, signifying that what lies to its left will eventually represent the canonical value of an expression. This *is* a canonical value when it has the form

$$L^n\ (body)\ \ a_0 \dots a_m$$

with m less than n. When the code takes the form

$$L^n\ (body)\ \ a_0 \dots a_m\ \ U^{\alpha} \dots\dots$$

then we will update the label α with the code to the left of the U, (with the variables replaced with labels or pointers as we explained above). Now we examine two examples to show how sharing works in the CMCM. Compiling the expression

$$\lambda x,y((x+y)*x)\ (\lambda x.x\ \mathbf{5})\ \mathbf{4}$$

we obtain the script

$$\begin{array}{lll} ex_0 & \rightarrow & ex_2\ \ ex_1\ \ ex_4 \\ ex_1 & \rightarrow & V^0_0\, ex_3 \\ ex_2 & \rightarrow & L^1(\mathbb{1}0 + \mathbb{1}*) \\ ex_3 & \rightarrow & \mathbf{5} \\ ex_4 & \rightarrow & \mathbf{4} \end{array}$$

in which shared occurrences of variables appear in outline font ($\mathbb{1}$, etc.)

Let us execute an example

code	multi-pair stack	ground
ex_0	..	..
$ex_2\ ex_1\ ex_4$	..	..
$L^2(\mathbb{1}0 + \mathbb{1}*)\ \ ex_1\ ex_4$	..	..
$\mathbb{1}0 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	..
$ex_1\, U^{ex1}\ 0 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	..

Note that all the slots in the multi-pair are occupied with labels appearing in the script.

$V^0_0\, ex_3\, U^{ex1}\ 0 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	..
$ex_3\, U^{ex1}\ 0 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	..
5 $U^{ex1}\ 0 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	..
$U^{ex1}\ 0 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	**5**

At this point we update the ex_1 entry in the script, so making the link

$$ex_1 \quad \rightarrow \quad \mathbf{5}$$

If we continue execution, we have,

$0 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	**5**
$ex_4 + \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	**5**
4 $+ \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	**5**
$+ \mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	**4 5**
$\mathbb{1}*F$	$(P\ \ ex_1\ \ ex_4)$	**9**
$ex_1\, U^{ex1}*F$	$(P\ \ ex_1\ \ ex_4)$	**9**
5 $U^{ex1}*F$	$(P\ \ ex_1\ \ ex_4)$	**9**
$U^{ex1}*F$	$(P\ \ ex_1\ \ ex_4)$	**5 9**

This second update is unnecessary - we should mark the label to this effect.

$*F$	$(P\ \ ex_1\ \ ex_4)$	**5 9**
F	$(P\ \ ex_1\ \ ex_4)$	**45**
		45

Now we consider a second example in which we share a partial function application. The λ-expression

$$(\lambda a.\lambda b.\lambda c(ac)(bc))(\lambda i.i)(\lambda j.j)((\lambda k.k)(\lambda l.l))\ \mathbf{3}$$

translates into the expression ex_5 in the following script.

$$\begin{array}{lll}
ex_0 & \rightarrow & ex_1\, ex_2\, ex_2\, ex_3\, ex_4 \\
ex_1 & \rightarrow & L^2(2\ \mathbb{0}\, fex) \\
fex & \rightarrow & 1\ \mathbb{0} \\
ex_2 & \rightarrow & V^0_0 \\
ex_3 & \rightarrow & V^0_0\ V^0_0 \\
ex_4 & \rightarrow & \mathbf{3}
\end{array}$$

Now we look at the evaluation

code	multi-pair stack	ground
ex_0	..	..
$L^2(2\ \mathbb{O}\, fex)\ ex_2\ ex_2\ ex_3\ ex_4$	..	..
$2\ \mathbb{O}\, fex\ F\ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..
$ex_2\ \mathbb{O}\, fex\ F\ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..
$V_0^0\ \mathbb{O}\, fex\ F\ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..
$\mathbb{O}\, fex\ F\ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..
$V_0^0\ V_0^0\ U^{ex3}\, fex\ F\ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..
$V_0^0\ U^{ex3}\, fex\ F\ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..

At this point there are not enough arguments on the left-hand side of the U^{ex3} to perform β-reduction. We therefore update the label ex_3 with the code to the left of the U^{ex3}

$$ex_3 \quad \rightarrow \quad V_0^0$$

and continue. Note that in this simple example, we had no free variables in the code which updates the label. If they are present, recall that we replace them with the labels to which they refer. Evaluation proceeds thus:

$V_0^0\ fex\ F\ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..
$fex\ F\ \ ex_4$	$(P\ ex_2\ ex_2\ ex_3)$	..

and so on.

CONCLUSION

We have given an account of a new machine, based on the ideas of categorical multi-combinators which first appeared in [Lins 2]. The machine has links with the TIM abstract machine, which we have explored in the companion paper [LiTh]. During the development of the machine we found the functional programming language Miranda[1] the ideal tool for machine prototyping; we have also written a prototype of much of the machine in C — obviously the next step is to build a complete implementation of the machine in an efficient manner.

We are grateful to Richard Jones and David Turner, both of the University of Kent, for discussions of the ideas in this paper; any defects are, of course, our responsibility.

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

BIBLIOGRAPHY

[Abd] A. Abdali *An abstraction algorithm for Combinatory Logic*, Journal of Symbolic Logic 41 (1976) pp 222-224.

[AbHa] S. Abramsky & C. Hankin (eds.) *Abstract Interpretation of Declarative Languages*, Ellis-Horwood, 1987.

[Cur] P.-L. Curien *Categorical Combinators, Sequential Algorithms and Functional Programming,* Pitman, 1986.

[FaiWr] J. Fairbairn & S. Wray *The Three Instruction Machine* in [Kahn]

[Gol] B. Goldberg *Detecting Sharing of Partial Applications in Functional Programs* in [Kahn]

[Hen] P. Henderson *Functional Programming: Application and Implementation,* Prentice-Hall, 1980.

[Hug] J. Hughes *The Design and Implementation of Programming Languages,* D. Phil thesis,Oxford University, 1984.

[Joh] T. Johnsson *Efficient Compilation of Lazy Evaluation*, in *Proceedings of ACM SIGPLAN '84 Symposium on Compiler Construction, Montreal,* ACM Press 1984.

[Joh1] T. Johnsson *Lambda Lifting —transforming programs to recursive equations,* in Jouannaud (ed.) *Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science 201, 1985.

[Joh2] T.Johnsson *Compiling Lazy Functional Languages* Ph. D. thesis Chalmers Tekniska Hogskola, 1987.

[Kahn] G. Kahn (ed.) *Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science 274, Springer-Verlag,1987.

[Lan] P. J. Landin *The mechanical evaluation of expressions,* The Computer Journal 6 (1964), pp 308-320.

[Lins1] R. D. Lins *On the Efficiency of Categorical Combinators in Applicative Languages* Ph. D. Thesis, University of Kent at Canterbury, 1986.

[Lins2] R. D. Lins *Categorical Multi-Combinators* in [Kahn]

[LiTh] R. D. Lins & S. Thompson *On the Equivalence Between CM-C and TIM* Computing Laboratory Report 67,University of Kent at Canterbury,1989, revised 1990. (Submitted for publication in Journal of Functional Programming.)

[Sco] D. S. Scott *Relating theories of the lambda calculus* in J.P. Seldin & J. R. Hindley (eds.) *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisation,* Academic Press, 1980.

[Tur] D.A. Turner *An Overview of Miranda* in D.A. Turner (ed.) Research Topics in Functional Programming, Addison-Wesley, 1990.

[Wad] C. P. Wadsworth *Semantics and Pragmatics of the Lambda Calculus,* D. Phil thesis, Oxford University, 1971.