J. Functional Programming 4 (1): 47–63, January 1994 © 1994 Cambridge University Press

On the equivalence between CMC and TIM

RAFAEL D. LINS, SIMON J. THOMPSON
Departament de Informática, Universidade Federal de Pernambuco, Recife, Brazil
Computing Laboratory, The University of Kent, Canterbury, UK

SIMON PEYTON JONES
Department of Computing Science, University of Glasgow, Glasgow, Scotland

Abstract

In this paper we present an equivalence between TIM, a machine developed to implement non-strict functional programming languages, and the set of Categorical Multi-Combinators, a rewriting system developed with similar aims. These two models of computation at first appear to be quite different, but we show a direct equivalence between them, thereby adding some new structure to the 'design-space' of abstract machines for non-strict languages.

Capsule Review

In recent years a large number of abstract machines have been proposed for compiling functional languages. Although at first each machine appeared as an isolated 'island', it is now becoming clear that the various different designs can be related to one another in a larger design space.

In this paper the authors describe in detail the correspondence between two particular abstract machines: the reduction machine for Categorical Multi-Combinators and the Three Instruction Machine. The importance of their work is that it provides concrete evidence that two (seemingly very different) abstract machines can indeed be related to one another within a larger design space.

1 Introduction

A number of different abstract machines for the implementation of lazy functional languages have been developed in the last few years. Many of these machines were developed using different principles or even based on different theories of functions and seem to be unrelated. In our opinion, it is important to examine the similarities and differences between these machines, because this will provide a better understanding of their features.

In this paper, we investigate the relationship between TIM and the system of Categorical Multi-Combinators. Although these two abstract machines initially appear to be completely unrelated, we prove their equivalence.

The first section presents the small source language which we use for both implementations. The next two sections describe Categorical Multi-Combinators and the TIM respectively. For further details on TIM, and indeed on other machines, we refer readers to (Peyton Jones and Lester, 1992). The core of the paper is section 5, in which we present two functions $\mathcal{C}$ and $\mathcal{T}$ translating from TIM to CMC, and *vice versa*. We show in section 5.2 and 5.4 that each of the translation functions respects rewriting, in a sense which we explain, and in section 5.5 we show that $\mathcal{T}$ is a inverse of $\mathcal{C}$, and *vice-versa*.

We make no attempt to address the issues of sharing and graph updates. In effect our results apply only to tree-reduction versions of TIM and CMC. The choice of the sharing mechanism gives rise to different flavours of TIM and CMC. The original frame update mechanism in TIM finds its equivalent in CM-CM (Lins and Thompson, 1990b; Thompson and Lins, 1992). On the other hand, Guy Argo's version of TIM with graphs (G-TIM (Argo, 1989), bears some resemblance to GMC (Musicante and Lins, 1991). ΓCMC (Lins and Lira, 1993), a categorical multi-combinator machine that brings together C code (a result of the compilation of strict functions and arithmetic expressions) and abstract machine code, has no equivalent on the TIM side.

2 The source language

The source language is defined by the following grammar:

$$\begin{array}{lcll}
program & ::= & c_1 = rhs_1 & \\
 & & \vdots & \\
 & & c_n = rhs_n & \\
rhs & ::= & \lambda x_1 ... x_n . expr & (n \geq 0) \\
expr & ::= & c & \text{Combinator} \\
 & | & x & \text{Variable} \\
 & | & expr_1 \ldots expr_n & \text{Application } (n \geq 2)
\end{array}$$

A program is a set of combinator definitions, including a distinguished combinator `main` whose value is the result of the program. Lambdas only appear at the top of a combinator definition, i.e., the program has been lambda-lifted (Johnsson, 1987). Any realistic language will include constants and built-in operations over them, but we omit them for simplicity.

Both TIM and the Categorical Multi-Combinator machine execute a compiled version of the program. Each combinator is compiled separately; we use the notation c_c to denote the compiled CMC code for combinator c, and c_t to denote the compiled TIM code.

3 Categorical Multi-Combinator Machine

This section briefly introduces the Categorical Multi-Combinator (CMC) compilation and reduction schemes. For a much fuller presentation see Lins (1987) and Lins and Thompson (1990a).[†]

[†] To make presentation easier we adopt a slightly different notation for Categorical Multi-Combinators from that presented there. The multi-pair combinator is represented by a tuple $(x_0, \ldots, x_n)$, we use the empty tuple () to denote identity, and angle brackets stand for closures $\langle a, b \rangle$ (previously written $\circ\ a\ b$).

3.1 Compilation

Categorical Multi-Combinator code is defined as follows:

$CCode$	$::=$	$L^n(CCode)$	Abstraction ($n \geq 0$)
	$\mid$	$CCode_1 \ \ldots \ CCode_n$	Application ($n \geq 2$)
	$\mid$	c	Combinator
	$\mid$	i	Variable ($i \geq 0$)

In CMC code, variables are identified with their DeBruijn number, i, defined as the number of lambdas between the occurrence of the variable and its binding lambda.

The compilation algorithm for translating programs into Categorical Multi-Combinators is given by the functions $\mathscr{B}_c$and $\mathscr{E}_c$, whose types are given by

$$\mathscr{B}_c :: rhs \to CCode$$
$$\mathscr{E}_c :: expr \to Env \to CCode$$
$$Env :: var \to int$$

The translation rules are:

$$\begin{array}{llcl}
(T.1) & \mathscr{B}_c[\![e]\!] & = & \mathscr{E}_c[\![e]\!] \ [\,] \\
(T.2) & \mathscr{B}_c[\![\lambda x_1 \ldots \lambda x_n.e]\!] & = & L^{n-1}(\mathscr{E}_c[\![e]\!] \ [x_1 \mapsto n-1, \ldots, x_n \mapsto 0]) \\
(T.3) & \mathscr{E}_c[\![e_1 \ \ldots \ e_n]\!] \ \pi & = & (\mathscr{E}_c[\![e_1]\!] \ \pi) \ \ldots \ (\mathscr{E}_c[\![e_n]\!] \ \pi) \\
(T.4) & \mathscr{E}_c[\![c]\!] \ \pi & = & c \\
(T.5) & \mathscr{E}_c[\![x]\!] \ \pi & = & \pi(x)
\end{array}$$

Rule (T.1) deals with the case where the combinator right-hand side has no lambdas, while (T.2) handles combinators that do have lambdas. Rules (T.3) to (T.5) handle the various forms of expression. For example, the combinator definition

$$S = \lambda a.\lambda b.\lambda c.ac(bc)$$

would be translated to

$$S_c = L^2(2 \ 0 \ (1 \ 0))$$

3.2 Reduction

The reduction rules for CMCs are described by a *state transition system.* This requires us to say what a state is, give rules which transform each state into its successor, and say what the initial state is.

A CMC *state* consists of the application of a *function closure* to zero or more *argument closures*:

$$CState ::= CClo_f \ CClo_1 \ \ldots \ CClo_n \qquad (n \geq 0)$$

Each closure is a pair of a $CCode$ (introduced above) with a *frame* which gives the values of each of the free variables in the code:

$$\begin{array}{lll}
CClo & ::= & \langle CCode, CFrame \rangle \\
CFrame & ::= & (CClo_1, \ldots, CClo_n) \qquad (n \geq 0)
\end{array}$$

In papers about categorical machines, states are usually called *expressions*, closures are usually called *compositions*, and frames are usually called *multi-pairs*. We use different terminology to stress the correspondence with the TIM machine, something that is made explicit later.

Using this notation, the kernel of the Categorical Multi-Combinator rewriting laws, each of which maps a *CState* to the next *CState*, is:

$$\begin{array}{llcl}
(M.1) & \langle n, (x_m, \cdots, x_1, x_0)\ w_0 \ldots w_k\rangle & \Rightarrow & x_n w_0 \ldots w_k \\
(M.2) & \langle x_0 x_1 x_2 \ldots x_n, y\rangle'' w_0 \ldots w_k & \Rightarrow & \langle x_0, y\rangle \ldots \langle x_n, y\rangle\ w_0 \ldots w_k \\
(M.3) & \langle L^n(y), f\rangle\ w_0 \ldots w_k & \Rightarrow & \langle y, (w_0, \cdots, w_n)\rangle\ w_{n+1} \ldots w_k \\
(M.4) & \langle c, f\rangle\ w_0 \ldots w_k & \Rightarrow & \langle c_c, ()\rangle\ w_0 \ldots w_k
\end{array}$$

Rule (M.1) performs environment look-up, in which the value corresponding to a variable is looked up in the corresponding frame. Rule (M.2) is responsible for environment distribution. Rule (M.3) performs combinator application and environment formation. Finally, Rule (M.4) says that an occurrence of a combinator can be replaced by its code paired with an empty frame. (The frame is empty because a combinator has no free variables.)

The initial state of the machine consists of the combinator `main` paired with an empty frame:

$$\langle \texttt{main}, ()\rangle$$

4 TIM

Next, we briefly introduce TIM, the Three Instruction Machine (Fairbairn and Wray, 1987; Peyton Jones and Lester, 1992).

4.1 Compilation

TIM code is defined as follows:

$$\begin{array}{lcl}
TCode & ::= & \texttt{Take}\ int;\ TCode \\
 & | & \texttt{Push}\ TAmode;\ TCode \\
 & | & \texttt{Enter}\ TAmode \\
TAmode & ::= & \texttt{Code}\ TCode \\
 & | & \texttt{Comb}\ c \\
 & | & \texttt{Arg}\ n
\end{array}$$

Notice that this code is not linear, because the `Code` addressing mode contains a nested *TCode*, giving the code a tree structure. Real compilers, of course, flatten the code by introducing arbitrary labels, but that merely complicates matters here without adding anything.

The right-hand side of each combinator is translated into TIM code by the compilation schemes $\mathcal{B}_t$, using the auxiliary schemes $\mathcal{E}_t$and $\mathcal{A}$:

$$\begin{array}{l}
\mathcal{B}_t :: rhs \rightarrow TCode \\
\mathcal{E}_t :: expr \rightarrow Env \rightarrow TCode \\
\mathcal{A} :: expr \rightarrow Env \rightarrow TAmode
\end{array}$$

The compilation schemes are as follows:

$(C.1)$	$\mathscr{B}_t[\![e]\!]$	$=$	$\mathscr{E}_t[\![e]\!]\ [\,]$
$(C.2)$	$\mathscr{B}_t[\![\lambda x_1 \dots \lambda x_n.e]\!]$	$=$	$\texttt{Take}\ n;\ \mathscr{E}_t[\![e]\!]\ [x_1 \mapsto 1, \dots, x_n \mapsto n]$
$(C.3)$	$\mathscr{E}_t[\![e_1\ \dots\ e_n]\!]\ \rho$	$=$	$\texttt{Push}\ (\mathscr{A}[\![e_n]\!]\ \rho);\ \dots \texttt{Push}\ (\mathscr{A}[\![e_2]\!]\ \rho);\ \mathscr{E}_t[\![e_1]\!]\ \rho$
$(C.4)$	$\mathscr{E}_t[\![a]\!]\ \rho$	$=$	$\texttt{Enter}\ (\mathscr{A}[\![a]\!]\ \rho)$
$(C.5)$	$\mathscr{A}[\![x]\!]\ \rho$	$=$	$\texttt{Arg}\ \rho(x)$
$(C.6)$	$\mathscr{A}[\![c]\!]\ \rho$	$=$	$\texttt{Comb}\ c$
$(C.7)$	$\mathscr{A}[\![e]\!]\ \rho$	$=$	$\texttt{Code}\ (\mathscr{E}_t[\![e]\!]\ \rho)$

As in the case of the CMC machine, the environment ρ maps combinator arguments to the slot number they occupy in the frame. As an example, consider again the combinator definition

$$S = \lambda a.\lambda b.\lambda c.ac(bc)$$

It will compile to

```
S_t = Take 3;
      PushCode [PushArg 2; EnterArg 3];
      PushArg 3;
      EnterArg 1
```

4.2 Execution

Like the CMC machine, the execution of the TIM code is described by a state transition semantics.

The state of the TIM is a four-tuple of the form

$$\langle \text{TCode}, \text{addr}, (\text{TClo}_1 : \dots : \text{TClo}_n), \text{addr} \mapsto \text{TFrame} \rangle$$

The first component is the code being executed, while the second is the address of the frame which gives the values of its free variables. The third component is a stack of argument closures (the top of the stack is on the left), and the fourth is the heap which maps addresses to frames. A closure is a pair of a code and frame address, while a frame is a tuple of closures:

$$TClo \quad ::= \quad \langle TCode, addr \rangle$$

$$TFrame \quad ::= \quad (TClo_1, \dots, TClo_n)$$

The initial state of the machine is TState ::=

$$\langle \texttt{EnterComb main}, (), [\,], [\,] \rangle$$

The state transition laws for TIM are

(s.1) $\langle \texttt{Take}\ n; I, f_0, (a_1 : \dots : a_n : A), F \rangle \Rightarrow \langle I, f, A, F[f \mapsto (a_1, \dots, a_n)] \rangle$,
where $f \notin dom(F)$

(s.2) $\langle \texttt{PushArg}\ n; I, f, A, F[f \mapsto (\ldots, a_n, \ldots)]\rangle \Rightarrow$
$\qquad\qquad \langle I, f, (a_n : A), F[f \mapsto (\ldots, a_n, \ldots)]\rangle$

(s.3) $\langle \texttt{PushCode}\ B; I, f, A, F\rangle \Rightarrow \langle I, f, (\langle B, f\rangle : A), F\rangle$

(s.4) $\langle \texttt{PushComb}\ c; I, f, A, F\rangle \Rightarrow \langle I, f, (\langle c_t, ()\rangle : A), F\rangle$

(s.5) $\langle \texttt{EnterArg}\ n, f, A, F[f \mapsto (\ldots, \langle I_n, f_n\rangle, \ldots)]\rangle \Rightarrow$
$\qquad\qquad \langle I_n, f_n, A, F[f \mapsto (\ldots, \langle c, f_n\rangle, \ldots)]\rangle$

(s.6) $\langle \texttt{EnterComb}\ c, f, A, F\rangle \Rightarrow \langle c_t, (), A, F\rangle$

Note that in law (s.1) we use the notation $F[f \mapsto (a_1, \ldots, a_n)]$ to represent the heap F updated with a new frame f, consisting of a_1 to a_n. In all other rules $F[f \mapsto (a_1, \ldots, a_n)]$ means the heap F contains a particular frame f. The empty tuple, (), represents the empty frame.

5 CMC AND TIM

The close relationship between TIM (Fairbairn and Wray, 1987) and the original set of Categorical Multi-Combinators (Lins, 1986; Lins 1987) has been known to the first author for a long time, and has also been mentioned by others (Wraith and Bosley, 1988). This equivalence was also outlined in Lins and Thompson (1990b).

Our aim in this section is to make clear the relationship between TIM (Fairbairn and Wray, 1987) and the original set of Categorical Multi-Combinators (Lins, 1986; Lins 1987). We present two functions $\mathscr{C}$, translating from TIM states to CMC states, and $\mathscr{T}$ going in the reverse direction. These functions are defined in terms of others which establish correspondences between components of the states. Specifically, CMC closures correspond to TIM closures, and similarly for frames; indeed, these correspondences suggested our choice of terminology. The functions involved are:

$$\begin{tikzcd}
TState \arrow[r, shift left, "\mathscr{C}"] & CState \arrow[l, shift left, "\mathscr{T}"] \\
TClo \arrow[r, shift left, "\tau"] & CClo \arrow[l, shift left, "\theta"] \\
TCode \arrow[r, shift left, "\alpha"] & CCode \arrow[l, shift left, "\sigma"]
\end{tikzcd}$$

We then show that the translations given commute with rewriting. First we show that if a TIM state T_1 rewrites in one step to state T_2 then $\mathscr{C}(T_1)$, the Categorical Multi-Combinator equivalent rewrites in a sequence of zero or more steps to $\mathscr{C}(T_2)$ – **(Property I)**. We then show that if a CMC expression M_1 rewrites in one step to M_2 then $\mathscr{T}(M_1)$, the TIM equivalent rewrites in a sequence of zero or more steps to $\mathscr{T}(M_2)$ (modulo an operational equivalence, discussed in section 5.4.2). This is called **(Property II)**.

Property I **Property II**

$$\begin{tikzcd}
T_1 \arrow[r] \arrow[d, Rightarrow] & \mathscr{C}(T_1) \arrow[d, Rightarrow, "*"] \\
T_2 \arrow[r] & \mathscr{C}(T_2)
\end{tikzcd}
\qquad
\begin{tikzcd}
M_1 \arrow[r] \arrow[d, Rightarrow] & \mathscr{T}(M_1) \arrow[d, Rightarrow, "\wr *"] \\
M_2 \arrow[r] & \mathscr{T}(M_2)
\end{tikzcd}$$

Finally, we show that $\mathcal{T}$ is an inverse of $\mathcal{C}$, i.e., '$\mathcal{C}$ then $\mathcal{T}$' is the identity on TIM states. The other inverse relationship also holds, so '$\mathcal{T}$ then $\mathcal{C}$' is the identity on CMC expressions.

5.1 Translating TIM into CMC

The translation from TIM states to Categorical Multi-Combinator expressions is performed by the following functions:

(t.1) $\mathcal{C}(\langle I, f, (x_0 : \ldots : x_z), F[f \mapsto (y_0, \ldots y_n)]\rangle) =$
$$\langle \alpha I, (\tau_F y_0, \ldots, \tau_F y_n)\rangle\ \tau_F x_0 \ldots \tau_F x_z$$
(t.2) $\tau_F \langle c_n, f\rangle = \langle \alpha c_n, (\tau_F y_0, \ldots, \tau_F y_m)\rangle$,
where $f \mapsto (y_0, \ldots, y_m)$ *in* F
(t.3) $\alpha[\texttt{Take}\ n; I] = L^{n-1}(\alpha I)$
(t.4) $\alpha[\texttt{PushArg}\ n; I] = \alpha I\ (n-1)$
(t.5) $\alpha[\texttt{PushCode}B; I] = \alpha I\ (\alpha B)$
(t.6) $\alpha[\texttt{PushComb}\ c; I] = \alpha I\ c$
(t.7) $\alpha[\texttt{EnterArg}\ n] = (n-1)$
(t.8) $\alpha[\texttt{EnterComb}\ c] = c$

α translates code sequences, and τ_F translates closures, relative to the heap of frames F. For notational simplicity the subscript F in τ_F will be omitted in the sequel if no misunderstanding can arise. Rule **t.1** translates a TIM state into a top-level Categorical Multi-Combinator expression; it is used to translate the expression under evaluation. We apply α to each entry in the TIM code to generate the corresponding CMC code.

5.2 Proof of Property I

We show that if a state T_1 rewrites to a state T_2 then $\mathcal{C}(T_1)$, the Categorical Multi-Combinator equivalent expression to T_1, rewrites in a sequence of zero or more steps to $\mathcal{C}(T_2)$. The translation between TIM states and CMC expressions is performed by the algorithm above. The following subsections prove the result clause by clause.

5.2.1 Multi β-Reduction

Let us start analysing the most important state transition law of both machines, the one which corresponds to β-reduction in the λ-Calculus. We can see that

$$\langle [\texttt{Take}\ n; I], f_0, (a_1 : \ldots : a_n : A), F\rangle \overset{\text{TIM}}{\Rightarrow} \langle I, f, A, F[f \mapsto (a_1, \ldots, a_n)]\rangle,$$

where f selects an unused frame and

$$\langle L^n(y), (w_0, \ldots, w_j)\rangle x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \overset{\text{CMC}}{\Rightarrow} \langle y, (x_0, \ldots, x_n)\rangle x_{n+1} \cdots x_z$$

perform an equivalent transformation to the code. This can be shown formally as follows:

$$\begin{array}{ccc}
\mathscr{C}(\langle[\texttt{Take } n; I], f, (x_0 : \ldots), F[f \mapsto (y_0, \ldots)]\rangle) & \overset{t.1}{=} & \langle\alpha[\texttt{Take } n; I], (\tau y_0, \ldots)\rangle\ \tau x_0 \ldots \\
\Downarrow s.1 & & \| \ t.3 \\
\mathscr{C}(\langle I, f_1, (x_n : \ldots), F[f_1 \mapsto (x_0, \ldots, x_{n-1})]\rangle) & & \langle L^{n-1}(\alpha I), (\tau y_0, \ldots)\rangle\ \tau x_0 \ldots \\
\| \ t.1 & & \Downarrow M^*.3 \\
\langle \alpha I, (\tau x_0, \ldots, \tau x_{n-1})\rangle\ \tau x_n \ldots & & \langle \alpha I, (\tau x_0, \ldots, \tau x_{n-1})\rangle\ \tau x_n \ldots
\end{array}$$

5.2.2 `PushArg` *as environment look-up*

The operation that allows a variable to fetch its value from its corresponding environment is expressed in TIM and CMC as

$$\langle[\texttt{PushArg } n; I], f, A, F[f \mapsto (\ldots, a_n, \ldots)]\rangle \overset{TIM}{\Rightarrow} \langle I, f, a_n, A, F[f \mapsto (\ldots, a_n, \ldots)]\rangle$$

and

$$\langle n, (x_m, \ldots, x_1, x_0)\rangle \overset{CMC}{\Rightarrow} x_n$$

Consider the behaviour of the rules

$$\begin{array}{ccc}
\mathscr{C}(\langle[\texttt{PushArg } n; I], f, (x_0 : \ldots), & \overset{t.1}{=} & \langle\alpha[\texttt{PushArg } n; I], (\tau a_i, \cdots)\rangle\ \tau x_0 \ldots \\
F[f \mapsto (a_i, \cdots)]\rangle) & & \\
\Downarrow s.2 & & \| \ t.4 \\
\mathscr{C}(\langle I, f, (a_n : x_0 \ldots), F[f \mapsto (a_i, \cdots)]\rangle) & & \langle \alpha I\ (n-1), (\tau a_i, \cdots)\rangle\ \tau x_0 \ldots \\
\| \ t.1 & & \Downarrow M^*.2 \\
\langle \alpha I, (\tau a_i, \cdots)\rangle\ \tau a_n\ \tau x_0 \ldots & & \langle \alpha I, (\tau a_i, \cdots)\rangle\langle (n-1), (\tau a_i, \cdots)\rangle \tau x_0 \ldots \\
 & & \Downarrow M^*.1 \\
 & & \langle \alpha I, (\tau a_i, \cdots)\rangle\ \tau a_n\ \tau x_0 \ldots
\end{array}$$

5.2.3 `PushCode` *as environment distribution*

This operation is performed by the following laws in TIM and CMC, respectively:

$$\langle[\texttt{PushCode } B; I], f, A, F\rangle \overset{TIM}{\Rightarrow} \langle I, f, \langle B, f\rangle : A, F\rangle$$

and

$$\langle x_0 x_1 x_2 \ldots x_n, y\rangle \overset{CMC}{\Rightarrow} \langle x_0, y\rangle\langle x_1, y\rangle \ldots \langle x_{n-1}, y\rangle\langle x_n, y\rangle$$

Right associated applications are translated into TIM code as `PushCode`. `PushCode` *B* builds a closure of the current frame and the code *B*.

Let us prove the operational equivalence between the laws above:

$$
\begin{array}{ccc}
\mathscr{C}(\langle[\texttt{PushCode}\ B;I], f, (x_0 : \ldots), & \overset{t.1}{=} & \langle\alpha[\texttt{PushCode}\ B;I], (\tau a_i, \cdots)\rangle\ \tau x_0 \ldots \\
F[f \mapsto (a_i, \cdots)]\rangle) & & \\
\Downarrow s.3 & & \| \ t.5 \\
\mathscr{C}(\langle I, f, [\langle B, f\rangle,\ x_0 \ldots], F[f \mapsto (a_i \cdots)]\rangle) & & \langle\alpha I\ (\alpha B), (\tau a_i, \cdots)\rangle\ \tau x_0 \ldots \\
\| \ t.1 & & \Downarrow M^{\bullet}.2 \\
\langle\alpha I, (\tau a_i, \cdots)\rangle\ \tau\langle B, f\rangle\ \tau x_0 \ldots & & \langle\alpha I, (\tau a_i, \cdots)\rangle\langle\alpha B, (\tau a_i, \cdots)\rangle\tau x_0 \ldots \\
\| \ t.2 & & \\
\langle\alpha I, (\tau a_i, \cdots)\rangle\langle\alpha B, (\tau a_i, \cdots)\rangle\tau x_0 \ldots & &
\end{array}
$$

5.2.4 `PushComb` *as script look-up*

In TIM and CMC functions are lambda lifted during compilation so that each function corresponds to a closed λ-expression or a combinator. Whenever a combinator is applied it will generate its own evaluation environment, binding actual parameters to formal parameters. In CMC, whenever a combinator name reaches the leftmost outermost position in the code we enter the corresponding code:

$$
\begin{array}{ccc}
\langle[\texttt{PushComb}\ c;I], f, A, F\rangle & \overset{TIM}{\Rightarrow} & \langle I, f, \langle c, ()\rangle : A, F\rangle \\
 & \text{and} & \\
\langle c, f\rangle\ w_0 \ldots w_k & \overset{CMC}{\Rightarrow} & \langle c_c, ()\rangle\ w_0 \ldots w_k
\end{array}
$$

Let us prove the operational equivalence between the rules above:

$$
\begin{array}{ccc}
\mathscr{C}(\langle[\texttt{PushComb}\ c;I], f, (x_0 : \ldots), & \overset{t.1}{=} & \langle\alpha[\texttt{PushComb}\ c;I], (\tau a_i, \cdots)\rangle\tau x_0 \ldots \\
F[f \mapsto (a_i, \cdots)]\rangle) & & \\
\Downarrow s.4 & & \| \ t.6 \\
\mathscr{C}(\langle I, f, (\langle c_t, ()\rangle : x_0, \ldots), F[f \mapsto (a_i, \cdots)]\rangle) & & \langle\alpha I\ c, (\tau a_i, \cdots)\rangle\tau x_0 \ldots \\
\| \ t.1 & & \Downarrow M^{\bullet}.2 \\
\langle\alpha I, (\tau a_i, \cdots)\rangle\tau\langle c_t, ()\rangle\tau x_0 \ldots & & \langle\alpha I, (\tau a_i, \cdots)\rangle\ \langle c, (\tau a_i, \cdots)\rangle\tau x_0 \ldots \\
\| \ t.2 & & \Downarrow M^{\bullet}.4 \\
\langle\alpha I, (\tau a_i, \cdots)\rangle\ \langle\alpha c_t, (\ )\rangle\tau x_0 \ldots & & \langle\alpha I, (\tau a_i, \cdots)\rangle\ \langle\alpha c_t, (\ )\rangle\tau x_0 \ldots
\end{array}
$$

(Strictly speaking the use of M.4 in the last step of the right hand column will only occur when (and if) the first argument of the state moves to the head position, but since none of the rules M.1...M.4 are affected by the structure of non-head sub-expressions, we can reason that the states are equivalent.)

5.2.5 `EnterArg` *as environment look-up*

$$\langle[\texttt{EnterArg}\ n], f, A, F[f \mapsto (\ldots\langle I_n, f_n\rangle\ldots)]\rangle \Rightarrow \langle I_n, f_n, A, F[f \mapsto (\ldots\langle I_n, f_n\rangle\ldots)]\rangle$$

In this rule, `EnterArg` performs a similar transformation to the code as `PushArg` above, i.e., an environment look-up. Let us see the state transition this rule performs in CMC:

$$\begin{array}{cl}
\mathscr{C}(\langle[\texttt{EnterArg}\ n], f, (x_0 : \ldots), & \overset{t.1}{=}\ \langle\alpha[\texttt{EnterArg}\ n], (\cdots\tau\langle I_n, f_n\rangle\cdots)\rangle\tau x_0\ldots \\
\quad F[f \mapsto (\cdots\langle I_n, f_n\rangle\cdots)]\rangle) & \\
\Downarrow s.5 & \quad\quad \| \ t.7 \\
\mathscr{C}(\langle I_n, f_n, [x_0, \ldots], F[f_n \mapsto (y_0, \ldots)]\rangle) & \langle(n-1), (\cdots, \tau\langle I_n, f_n\rangle, \cdots)\rangle\tau x_0\ldots \\
\| \ t.1 & \quad\quad \Downarrow M^{*}.1 \\
\langle\alpha I_n, (\tau y_0, \cdots, \tau y_m)\rangle\tau x_0\ldots & \tau\langle I_n, f_n\rangle\tau x_0\ldots \\
 & \quad\quad \| \ t.2 \\
 & \langle\alpha I_n, (\tau y_0, \cdots, \tau y_m)\rangle\tau x_0\ldots
\end{array}$$

5.2.6 `EnterComb` *as script look-up*

The role of the `EnterComb` instruction is simply to read the code for a function definition from the script, performing a lazy linking of the code, by the following law:

$$\langle[\texttt{EnterComb}\ c], f, A, F\rangle \Rightarrow \langle c_t, (), A, F\rangle$$

This law is equivalent to the following state transformation in CMC:

$$\begin{array}{cl}
\mathscr{C}(\langle[\texttt{EnterComb}\ c], f, [x_0, \ldots], & \overset{t.1}{=}\ \langle\alpha[\texttt{EnterComb}\ c], (\tau a_m, \cdots)\rangle\ \tau x_0\ldots \\
\quad F[f \mapsto (a_m, \cdots)]\rangle) & \\
\Downarrow s.6 & \quad\quad \| \ t.8 \\
\mathscr{C}(\langle c_t, (), [x_0, \ldots], F[f \mapsto (a_m, \ldots)]\rangle) & \langle c, (\tau a_m, \cdots)\rangle\ \tau x_0\ldots \\
\| \ t.1 & \quad\quad \| \ M.4 \\
\langle\alpha c_t, (\ )\rangle\ \tau x_0\ldots & \langle c_c, (\ )\rangle\ \tau x_0\ldots \\
\| & \\
\langle c_c, (\ )\rangle\ \tau x_0\ldots &
\end{array}$$

5.3 Translating CMC into TIM

The translation between Categorical Multi-Combinator expressions and TIM states is performed by the following functions:

(r.1) $\mathscr{T}(\langle e, (y_0, \ldots, y_m)\rangle\ w_0\ldots w_k) = \langle\sigma e, f, [\theta w_0, \ldots, \theta w_k], F[f \mapsto (\theta y_0, \ldots, \theta y_m)]\rangle$
(r.2) $\theta\langle x, (y_0, \ldots, y_m)\rangle = \langle\sigma x, f\rangle, where\ f \mapsto (\theta y_0, \ldots, \theta y_m)$
(r.3) $\sigma n =$ `EnterArg` $(n+1)$
(r.4) $\sigma c =$ `EnterComb` c

(r.5) $\sigma(e_0 e_1 \ldots e_m) = \Psi e_m; \ldots ; \Psi e_1 ; \sigma e_0$
(r.6) $\sigma L^{n-1}(x) = [\texttt{Take } n; \sigma x]$
(r.7) $\Psi n = \texttt{PushArg } (n+1)$if n is a variable
(r.8) $\Psi c = \texttt{PushComb } c$if c is a combinator
(r.9) $\Psi(e_0 e_1 \ldots e_m) = \texttt{PushCode } [\sigma(e_0 e_1 \ldots e_m)]$

$\mathcal{T}$ translates a Categorical Multi-Combinator state into a TIM state, θ translates closures, and σ translates CMC code into TIM code. Ψ is an auxiliary function to σ, much as $\mathcal{A}$ is auxiliary to $\mathcal{E}_t$ in section 4.1.

F appears as an unbound variable in rule (r.1) – the meaning of this is 'the heap built by the recursive invocation of θ on the subexpressions to which it is applied'. When (r.2) is applied a new frame in the heap is generated, and we can see that the traversal of the Categorical Multi-Combinator expression gives rise to a collection of frames (F) in the heap.

The corresponding TIM script is generated by applying σ to each of the entries of the CMC script, thus:

$$c_t = \sigma c_c$$

5.4 *Proof of Property II*

We show here that if a Categorical Multi-Combinator expression M_1 rewrites to expression M_2 in one step, then the TIM state $\mathcal{T}(M_1)$ rewrites in a sequence of one or more steps to $\mathcal{T}(M_2)$, modulo the proviso in Section 5.4.2. The translation between CMC expressions and TIM states is performed by the algorithm above.

5.4.1 *Environment look-up*

$$\begin{tikzcd}
\mathcal{T}(\langle n,(\ldots,\langle y,(x_j,\cdots x_l)\rangle,\ldots)\rangle w_0\ldots w_k) \arrow[r, phantom, "\overset{r.1}{=}"] \arrow[d, Rightarrow, "M^{\bullet}.1"] & \begin{array}{l}\langle \sigma n, f, [\theta w_0,\ldots,\theta w_k],\\ F[f\mapsto(\ldots,\theta\langle y,(x_j,\cdots x_l)\rangle,\ldots)]\rangle\end{array} \arrow[d, equal, "r.3"] \\
\mathcal{T}(\langle y,(x_j,\cdots,x_l)\rangle w_0\ldots w_k) \arrow[d, equal, "r.1"] & \begin{array}{l}\langle \texttt{EnterArg}\ (n+1), f, [\theta w_0,\ldots],\\ F\begin{bmatrix} f\mapsto(\ldots,\langle\sigma y,f'\rangle,\ldots)\\ f'\mapsto(\theta x_j,\cdots,\theta x_l)\end{bmatrix}\rangle\end{array} \arrow[d, Rightarrow, "s.5"] \\
\begin{array}{l}\langle\sigma y, f', [\theta w_0,\ldots,\theta w_k],\\ F[f'\mapsto(\theta x_j,\cdots,\theta x_l)]\rangle\end{array} & \begin{array}{l}\langle\sigma y, f', [\theta w_0,\ldots,\theta w_k],\\ F[f\mapsto(\ldots,\langle\sigma y,f'\rangle,\ldots)]\rangle\end{array}
\end{tikzcd}$$

The translation rules give rise to different heaps on the left and right hand sides. Note, however, that the only difference is the presence of an additional frame f on the right hand side. As rewriting is not affected by the presence of this extra frame, the two expressions above are equivalent.

5.4.2 *Environment distribution*

$$\begin{array}{ccc}
\mathscr{T}(\langle x_0\ldots x_n,(y_m,\ldots)\rangle w_0\ldots) & \overset{r.1}{=} & \langle\sigma(x_0\ldots x_n),f,[\theta w_0,\ldots],F'\rangle \\
\Downarrow M^{*}.2 & & \|\; r.5 \\
\mathscr{T}(\langle x_0,(y_m,\ldots)\rangle\ldots\langle x_n,(y_m,\ldots)\rangle w_0\ldots) & & \langle\Psi x_n\ldots\sigma x_0,f,[\theta w_0,\ldots],F'\rangle \\
\|\; r.1 & & \\
\langle\sigma x_0,f,[\theta\langle x_1,(y_m,\ldots)\rangle,\ldots,\theta w_0,\ldots],F'\rangle & &
\end{array}$$

where we define $F' = F[y \longmapsto (\theta y_m,\ldots)]$.

There are three cases to be considered depending on the form of the expressions x_i. The appropriate case is used to transform the closures on the TIM stack (in the left-hand column) and the Push instructions in the TIM instruction stream (in the right-hand column). x_i **is a combinator** c:

$$\begin{array}{cc}
\|\; r.7 & \|\; r.8 \\
\langle\sigma x_0,f,[\ldots,\langle\sigma c,(y_m,\ldots)\rangle,\theta w_0,\ldots],F'\rangle & \langle\texttt{PushComb}\ c\ldots\sigma x_0,f,[\theta w_0,\ldots],F'\rangle \\
\| & \Downarrow s.4 \\
\langle\sigma x_0,f,[\ldots,\langle\texttt{EnterComb}\ c,(y_m,\ldots)\rangle, & \langle\Psi x_{n-1}\ldots\sigma x_0,f,[\langle c_t,()\rangle,\ \theta w_0,\ldots],F'\rangle \\
\theta w_0,\ldots],F\rangle & \\
 & \Downarrow *(s.2,s.3,or\ s.4) \\
 & \langle\sigma x_0,f,[\ldots,\langle c_t,()\rangle,\theta w_0,\ldots],F'\rangle
\end{array}$$

Now, the TIM closure $\langle\texttt{EnterComb}\ c,(y_m,\ldots)\rangle$ behaves exacly like the closure $\langle c_t,()\rangle$, so the two states are operationally equivalent.

x_n **is a variable** a:

$$\begin{array}{cc}
\|\; r.6 & \|\; r.7 \\
\langle\sigma x_0,f,[\ldots,\theta\langle a,(y_m,\ldots)\rangle, & \langle\texttt{PushArg}\ (a+1)\ldots\sigma x_0,f,[\theta w_0,\ldots],F'\rangle \\
\theta w_0,\ldots],F'\rangle & \\
 & \Downarrow s.2 \\
 & \langle\Psi x_{n-1}\ldots\sigma x_0,f,[\theta y_a\ ,\theta w_0,\ldots],F'\rangle \\
 & \Downarrow *(s.2,s.3,\ or\ s.4) \\
 & \langle\sigma x_0,f,[\ldots,\theta y_a\ ,\theta w_0,\ldots],F'\rangle
\end{array}$$

The final states here are equivalent, operationally, as in whichever situation the closure $\theta\langle a,(y_m,\ldots)\rangle$ is invoked, it will behave in exactly the same way as y_a. It is in this sense that the result is modulo an operational equivalence.

x_n **is an application** (e_0e_1):

$$\begin{array}{ll}
\Vert\ r.6 & \Vert\ r.9 \\
\langle \sigma x_0, f, [\ldots, \langle\sigma(e_0e_1), f\rangle, & \langle \texttt{PushCode}\ [\sigma(e_0e_1)]\ldots\sigma x_0, f, [\theta w_0, \ldots], F'\rangle \\
\quad \theta w_0, \ldots], F'\rangle & \\
 & \Downarrow s.3 \\
 & \langle \Psi x_{n-1}\ldots\sigma x_0, f, [\langle\sigma(e_0e_1), f\rangle,\ \theta w_0, \ldots], F'\rangle \\
 & \Downarrow *(s.2,\ s.3,\ or\ s.4) \\
 & \langle \sigma x_0, f, [\ldots, \langle\sigma(e_0e_1), f\rangle, \theta w_0, \ldots], F'\rangle
\end{array}$$

5.4.3 Multi β-reduction

$$\begin{array}{lcl}
\mathscr{T}(\langle L^{n-1}(y), (w_0, \ldots, w_j)\rangle x_0 \ldots x_z) & \overset{r.1}{=} & \langle \sigma L^{n-1}(y), f', [\theta x_0, \ldots, \theta x_z], \\
 & & \quad F[f' \mapsto (\theta w_0, \ldots, \theta w_j)]\rangle \\
\Downarrow M^*.3 & & \Vert\ r.6 \\
\mathscr{T}(\langle y, (x_0, \ldots, x_{n-1})\rangle\ x_n \ldots x_z) & & \langle [\texttt{Take}\ n; \sigma y], f', [\theta x_0, \ldots, \theta x_z], \\
 & & \quad F[f' \mapsto (\theta w_0, \ldots, \theta w_j)]\rangle \\
\Vert\ r.1 & & \Downarrow s.1 \\
\langle \sigma y, f, [\theta x_n, \ldots, \theta x_z], & & \langle \sigma y, f, [\theta x_n, \ldots, \theta x_z], \\
F[f \mapsto (\theta x_0, \ldots, \theta x_{n-1})]\rangle & & F[f \mapsto (\theta x_0, \ldots, \theta x_{n-1})]\rangle
\end{array}$$

The heap in the right hand side has an additional frame f' if compared with the heap in the left hand side. As this does not affect rewriting we can say that the two expressions above are equivalent.

5.5 $\mathscr{C}$ and $\mathscr{T}$

We show that the two translation functions $\mathscr{C}$ and $\mathscr{T}$ are related to each other, and that $\mathscr{T}$ and $\mathscr{C}$ are inverse mappings.

5.5.1 $\mathscr{T} \circ \mathscr{C} = Identity$

Here we prove that $\mathscr{T}(\mathscr{C}x) = x$, when x is a TIM state by structural induction over the structure of x.

$$\begin{array}{lcl}
\mathscr{T}(\mathscr{C}\langle I, f, [x_0, \ldots, x_z], & \overset{t.1}{=} & \mathscr{T}(\langle \alpha I, (\tau y_0, \ldots, \tau y_n)\rangle\ \tau x_0 \ldots \tau x_z) \\
F[f \mapsto (y_0, \ldots, y_n)]\rangle) & & \\
 & \overset{r.1}{=} & \langle \sigma(\alpha I), f, [\theta(\tau x_0), \ldots, \theta(\tau x_z)], \\
 & & \quad F[f \mapsto (\theta(\tau y_0), \ldots, \theta(\tau y_n))]\rangle
\end{array}$$

Assuming that $\sigma(\alpha z) = z$ and $\theta(\tau x) = x$

$$\overset{\theta\tau}{=} \quad \langle I, f, [x_0, \ldots, x_z], F[f \mapsto (y_0, \ldots, y_n)]\rangle$$

Now we prove that $\theta(\tau x) = x$ by induction over the structure of x:

$$\begin{aligned}
\theta(\tau[\langle c_n, f\rangle]) &\overset{t.2}{=} \theta(\langle \alpha c_n, (\tau y_0, \ldots, \tau y_m)\rangle) \\
&\overset{r.2}{=} \langle \sigma(\alpha c_n), f\rangle \\
&\quad \text{where} f \mapsto (\theta(\tau y_0), \ldots, \theta(\tau y_m))
\end{aligned}$$

by induction $\theta(\tau I) = I$ and assuming $\sigma(\alpha z) = z$,

$$\overset{\theta\tau,\sigma\alpha}{=} \langle c_n, f\rangle$$

We need to prove that $\sigma(\alpha z) = z$

$$\begin{aligned}
\sigma(\alpha[\texttt{Take } n; I]) &\overset{t.3}{=} \sigma(L^{n-1}\ \alpha I) \\
&\overset{r.6}{=} [\texttt{Take } n; \sigma(\alpha I)] \\
\text{by induction } \sigma(\alpha I) = I, \text{ so } &\overset{\sigma\alpha}{=} [\texttt{Take } n; I]
\end{aligned}$$

$$\begin{aligned}
\sigma(\alpha[\texttt{PushArg } n; I]) &\overset{t.4}{=} \sigma(\alpha I\ n-1) \\
&\overset{r.5}{=} [\Psi(n-1); \sigma(\alpha I)] \\
&\overset{r.7}{=} [\texttt{PushArg } n; \sigma(\alpha I)] \\
\text{by induction } \sigma(\alpha I) = I, \text{so } &\overset{\sigma\alpha}{=} [\texttt{PushArg } n; I]
\end{aligned}$$

$$\begin{aligned}
\sigma(\alpha[\texttt{PushCode } [B]; I]) &\overset{t.5}{=} \sigma(\alpha I\ (\alpha B)) \\
&\overset{r.5}{=} [\Psi(\alpha B); \sigma(\alpha I)] \\
&\overset{r.9}{=} [\texttt{PushCode } [\sigma(\alpha B)]; \sigma(\alpha I)] \\
\text{by induction } \sigma(\alpha I) = I, &\overset{\sigma\alpha}{=} [\texttt{PushCode } [B]; I]
\end{aligned}$$

$$\begin{aligned}
\sigma(\alpha[\texttt{PushComb } c; I]) &\overset{t.6}{=} \sigma(\alpha I; c'_r) \\
&\overset{r.5}{=} [\Psi c'_r; \sigma(\alpha I)] \\
&\overset{r.8}{=} [\texttt{PushComb } c; \sigma(\alpha I)] \\
\text{by induction } \sigma(\alpha I) = I, \text{so } &\overset{\sigma\alpha}{=} [\texttt{PushComb } c; I]
\end{aligned}$$

$$\begin{aligned}
\sigma(\alpha[\texttt{EnterArg } n]) &\overset{t.7}{=} \sigma(n-1) \\
&\overset{r.3}{=} [\texttt{EnterArg } n] \\
\sigma(\alpha[\texttt{EnterComb } c]) &\overset{t.8}{=} \sigma c'_r \\
&\overset{r.4}{=} [\texttt{EnterComb } c]
\end{aligned}$$

5.5.2 $\mathscr{C} \circ \mathscr{T} = $ *Identity*

We will show that $\mathscr{C}(\mathscr{T}x) = x$, where x is a Categorical Multi-Combinator expression, also holds.

$$\begin{aligned}
\mathscr{C}(\mathscr{T}(\langle e,(y_0,\ldots,y_m)\rangle w_0\ldots w_k)) &\overset{r.1}{=} \mathscr{C}\langle \sigma e, f, [\theta w_0,\ldots,\theta w_k], \\
&\quad\; F[f\mapsto(\theta y_0,\ldots,\theta y_m)]\rangle \\
&\overset{t.1}{=} (\langle \alpha[\sigma e],(\tau[\theta y_0],\ldots,\tau[\theta y_m])\rangle \\
&\quad\; \tau[\theta w_0]\ldots\tau[\theta w_m] \\
\text{If } \alpha[\sigma e]=e \text{ and } \tau[\theta x]=x,\ \text{then} &\overset{\alpha\sigma}{=} (\langle e,(y_0,\ldots,y_m)\rangle\ w_0\ldots w_k
\end{aligned}$$

Now we prove that $\tau[\theta x]=x$, where x is a Categorical Multi-Combinator expression by induction over the structure of x:

$$\begin{aligned}
\tau[\theta(\langle x,(y_0,\ldots,y_m)\rangle)] &\overset{r.2}{=} \tau[\langle \sigma x, f\rangle] \\
&\overset{t.2}{=} \langle \alpha[\sigma x],(\tau[\theta y_0],\ldots,\tau[\theta y_m])\rangle \\
&\overset{\alpha\sigma,\tau\theta}{=} \langle x,(y_0,\ldots,y_m)\rangle
\end{aligned}$$

Now we prove that $\alpha(\sigma x)=x$, again by induction over the structure of x.

$$\begin{aligned}
\alpha[\sigma L^{n-1}(I)] &\overset{r.6}{=} \alpha[\mathtt{Take}\ n;\sigma I] \\
&\overset{t.3}{=} L^{n-1}(\alpha[\sigma I]) \\
&\overset{\alpha\sigma}{=} L^{n-1}(I) \\
\alpha[\sigma n] &\overset{r.3}{=} \alpha[\mathtt{EnterArg}\ (n+1)] \\
&\overset{t.7}{=} (n+1)-1 \\
&= n
\end{aligned}$$

$$\begin{aligned}
\alpha[\sigma(e_0e_1\ldots e_m)] &\overset{r.5}{=} \alpha[\Psi e_m;\ldots;\Psi e_1;\sigma e_0] \\
\text{if } e_m \text{ is a variable } n &\overset{r.7}{=} \alpha[\mathtt{PushArg}\ (n+1);\ldots;\Psi e_1;\sigma e_0] \\
&\overset{t.4}{=} \alpha[\ldots;\Psi e_1;\sigma e_0]n \\
&\overset{r.5,\alpha\sigma}{=} e_0e_1\ldots e_m
\end{aligned}$$

The induction is in the last step, which also uses (r.5) backwards. The other two cases are similar:

$$\begin{aligned}
\text{if } e_m \text{ is an application } (w_0w_1) &\overset{r.9}{=} \alpha[\mathtt{PushCode}\ [\sigma(w_0w_1)];\ldots;\Psi e_1;\sigma e_0] \\
&\overset{t.5}{=} \alpha[\ldots;\Psi e_1;\sigma e_0](\alpha[\sigma(w_0w_1)]) \\
&\overset{\alpha\sigma}{=} \alpha[\ldots;\Psi e_1;\sigma e_0](w_0w_1) \\
&\overset{r.5,\alpha\sigma}{=} e_0e_1\ldots e_m \\
\text{if } e_m \text{ is a combinator } c &\overset{r.8}{=} \alpha[\mathtt{PushComb}\ c;\ldots;\Psi e_1;\sigma e_0] \\
&\overset{t.6}{=} \alpha[\ldots;\Psi e_1;\sigma e_0]c_t \\
&\overset{r.5,\alpha\sigma}{=} e_0e_1\ldots e_m
\end{aligned}$$

6 Related work

There is a fair amount of related work which derives an abstract machine design (or designs) from the semantic specification of a (usually functional) language (Wand, 1991; Kelsey and Hudak, 1989; Lester, 1989; Meijer, 1992). There seems to be much less literature which directly relates to abstract machine designs arising from different roots, such as that presented in this paper. Meijer explores several abstract machine designs within a single formal framework (Meijer, 1989), while Hannan presents a framework for reasoning about the 'concretisation' of an abstract machine (Hannan, 1991). Peyton Jones and Lester (1992, Chap. 4) give an informal transformation which shows the relationship between the G-machine and TIM.

7 Conclusions

In this paper we have shown the equivalence between the operational semantics of the TIM machine and rewriting of Categorical Multi-Combinator expressions: every TIM state is equivalent to a Categorical Multi-Combinator expression and *vice versa*; equivalent expressions are transformed into equivalent expressions by rewriting. In effect, we can see TIM and CMC as notational re-packagings of each other. The only real difference is that the CMC's environment-distribution rule (M.2) does in one step what is done by a sequence of `Push` instructions in TIM.

A notable omission from our treatment is the issue of sharing and updates, which we leave as an open, and quite difficult, problem.

We see the main interest of our result as an extra landmark in the 'design space' of abstract machines.

Acknowledgements

We would like to thank David Wakeling for his helpful comments.

Research reported herein has been sponsored jointly by the British Council, C.N.Pq. (Brazil) grants No 40.9110/88.4. and 46.0782/89.4, and CAPES (Brazil) grant 2487/91-08.

References

Argo, G. (1989) Improving the three instruction machine. *Functional Programming Languages and Computer Architecture*. Addison-Wesley

Fairbairn, J. and Wray, S. (1987) Tim: A simple, lazy abstract machine to execute supercombinators. In *Proc. 3rd Int. Conf. on Functional Programming and Computer Architecture*. Volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag, pp. 34–45.

Hannan, J. (1991) Making abstract machines less abstract. *Functional Programming Languages and Computer Architecture (FPCA)*. Volume 523 of *Lecture Notes in Computer Science*. Springer-Verlag.

Johnsson, T. (1989) *Compiling Lazy Functional Languages*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden.

Kelsey, R. and Hudak, P. (1989) Realistic compilation by program transformation. In *Proc. ACM Conf. on Principles of Programming Lang.*, pp. 281–292.

Lester, D. (1989) *Combinator graph reduction: a congruence and its applications.* PhD Thesis, Programming Research Group, Oxford.

Lins, R. D. (1986) *On the Efficiency of Categorical Combinators in Applicative Languages.* PhD thesis, University of Kent at Canterbury.

Lins, R. D. (1987) Categorical multi-combinators. In Kahn, G., editor, *Functional Programming Languages and Computer Architecture.* Volume 274 of *Lecture Notes in Computer Science.* Springer-Verlag, pp. 60–79.

Lins, R. D. and Lira, B. O. (1993) ΓCMC: A Novel Way to Implement Functional Languages. *Journal of Programming Language Design and Implementation.* (To appear.)

Lins, R. D. and Thompson, S. J. (1990a) Implementing SASL using categorical multi-combinators. *IEEE Trans. Softw. — Practice and Experience* **20**(11):1137–1165.

Lins, R. D. and Thompson, S. J. (1990b) CM-CM: A categorical multi-combinator machine. In *Proc. XVI Latino-American Conf. on Informatics*, vol(1) - pp. 181–198, Asunción, Paraguay.

Meijer, E. (1989) *Cleaning up the design space of function evaluating machines.* Department of Informatics, University of Nijmegen, March.

Meijer, E. (1992) *Calculating Compilers.* PhD thesis, University of Nijmegen, February.

Musicante, M. A. and Lins, R. D. (1991) GMC – a graph multi-combinator machine. *Microprocess. and Microprogram.* **31**(1–5):81–84, April.

Peyton Jones, S. (1987) *The Implementation of Functional Languages.* Prentice-Hall.

Peyton Jones, S. and Lester, D. (1992) *Implementing Functional Languages: A Tutorial.* Prentice-Hall.

Thompson, S. J. and Lins, R. D. (1992) The Categorical Multi-Combinator Machine: CM-CM. *The Computer Journal* **35**(2):170–176, April.

Wand, M. (1991) *Correctness of procedure representations.* Department of Computer Science, Northeastern University, March.

Wraith, G. and Bosley, D. (1988) Private communication.