On The Equivalence Between CM-C and TIM

Rafael D.Lins & Simon J.Thompson
Dept. de Informática - Universidade Federal de Pernambuco - Recife - Brazil
Computing Laboratory - The University of Kent - Canterbury - England.

Abstract

In this paper we present the equivalence between TIM, a machine developed to implement lazy functional programming languages, and the set of Categorical Multi-Combinators, a rewriting system developed with similar aims.
Keywords: Categorical Multi-Combinators, lambda calculus, functional programming.

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 seem to be completely unrelated we prove their equivalence.

The method of compilation of functional languages into combinators, first explored by Turner in[18], provides a way of removing the variables from a program, transforming it into an applicative combination of constant functions or **combinators**. Turner used a set of combinators based on Curry's Combinatory Logic. To each combinator there is associated a rewriting law. In rewriting a combinator expression, Turner rewrites the leftmost-outermost reducible subexpression (or *redex*) at each stage. When no further rewriting can take place the expression is said to be in **normal form**.

Another theory of functions is provided by Category Theory [4], and we can see the notation used herein as providing an alternative set of combinators. The original system of Categorical Combinators was developed by Curien [1]. This work was inspired by the equivalence of the theories of typed λ-calculus and Cartesian Closed Categories as shown by Lambek [4] and Scott [15].

Aiming to implement lazy functional languages in an efficient way using rewriting of Categorical Combinators we developed a number of optimisations [5, 7] of the naïve system, the most refined of which was the system of Linear Categorical Combinators [7]. The modifications introduced reduce the number of rewriting laws and increase the efficiency of the system by reducing the number of rewriting steps involved in taking an expression to normal form, whilst leaving the complexity of the pattern matching algorithm unchanged.

Categorical Multi-Combinators are a generalisation of Linear Categorical Combinators. Each rewriting step of the Multi-Combinator code is equivalent to several rewritings of Linear Categorical Combinators, since an application of a function to several arguments can be reduced in a single step. The core of the system of Categorical Multi-Combinators consists only of two rewriting laws with a very low pattern-matching complexity and avoids the generation of trivially reducible sub-expressions.

Independently, there has been much interest in compiled versions of functional languages which run much more quickly on von Neumann machines than do interpreters. Johnsson, with his implementation of Lazy ML [3], showed that it is possible to get fast implementations of lazy functional languages. Johnsson's implementation model was described as the G-Machine [13, 3]. The basic principle of the G-Machine is to avoid generating graph nodes when it is unnecessary. Several optimisations to the G-Machine are suggested in [13, 3]. In [16] there is an analysis of these optimisations and their performance figures obtained with several different benchmark programs.

Categorical Multi-Combinators served as basis for two compiled machines: GMC [12] and CM-CM [11, 17]. GMC is inspired by the G-Machine, in the sense that it generates graph lazily. The implementation of GMC has shown performance close, but slower, than the G-machine. CM-CM is a stack based machine which served as a basis for ΓCMC, a lower level abstract machine suitable for efficient implementation of functional languages on RISC architectures. The implementation of ΓCMC, still in progress has shown performance figures which in the best case is several times faster and in the worst case it is 10% slower than Chalmers LML compiler based on the G-machine.

At the same time, independent work on the Ponder abstract machine by Fairbairn and Wray developed into a more sophisticated system, the Three Instruction Machine, or TIM [2] which can be thought of as a lazy SECD machine.

In this paper we investigate the relationship between TIM and the system of Categorical Multi-Combinators. The first section presents the source language for generating Categorical Multi-Combinator expressions and TIM code. To make presentation easier we adopted a slightly different notation for Categorical Multi-Combinators from the one presented in [8]. 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 $\langle a, b\rangle$ (which we previously wrote $\circ\ a\ b$). We follow this by explaining the evaluation mechanism in Categorical Multi-Combinators [8] and TIM [2]. For further details on TIM, and indeed on other machines we refer readers to [14]. The core of the paper is section 3, in which we present two functions $\mathcal{C}$ and $\mathcal{T}$ translating from TIM to CMC and vice versa. We show in 3.2 and 3.4 that each of the translation functions respects rewriting, in a sense which we explain, and in 3.5 we show that $\mathcal{T}$ is a left inverse of $\mathcal{C}$, and that $\mathcal{C}$ is a left inverse of $\mathcal{T}$ *modulo rewriting.*

1 The Source Language

A program is taken to be a sequence of combinator definitions together with an expression to be evaluated, which will involve these combinators.

$$\begin{array}{rcl} c_1 & =_{def} & combinator_1 \\ & \ldots & \\ c_n & =_{def} & combinator_n \\ & main-expression & \end{array}$$

A program when compiled will generate a script which is formed by a sequence of combinators linked to their code thus,

$$\rho = \left[\begin{array}{lcl} c_1 & \mapsto & [\![combinator_1]\!] \\ & \vdots & \\ c_n & \mapsto & [\![combinator_n]\!] \end{array}\right]$$

In order to flatten the source code the compilation algorithms for Categorical Multi-Combinators and TIM will extract right-parenthesised expressions and replace each of them by a unique label. These labels will also be part of the script, and as with combinators they have their name linked to their code.

$$\rho = \left[\begin{array}{lcl} c_1 & \mapsto & [\![combinator_1]\!] \\ & \vdots & \\ c_n & \mapsto & [\![combinator_n]\!] \\ l_1 & \mapsto & [\![expression_1]\!] \\ & \vdots & \\ l_m & \mapsto & [\![expression_m]\!] \end{array}\right]$$

The *main-expression* is compiled separately as,

$$[\![main-expression]\!]\rho$$

In order properly to interpret recursion, we assume that the environment ρ contains the definition of all combinators, so that recursive combinators produce recursive references through the environment. The notation we use is: with each label l there is associated code l_r and with each combinator c there is associated code c_r, we supress the environment ρ when no confusion is possible.

1.1 Compiling into Categorical Multi-Combinators

In Categorical Multi-Combinators function application is denoted by juxtaposition, taken to be left-associative. The compilation algorithm for translating λ-expressions into Categorical Multi-Combinators is given by the function $R^{x_0 \ldots x_j}$ where each x_i is a variable and the corresponding i its depth in the environment, i.e. the corresponding DeBruijn number. Top level expressions are translated using an empty environment, so by $R^{[\ ]}$. For a matter of uniformity combinators will be represented as composed with a dummy frame, (), which can be seen as the identity frame.

(T .1) $R^{[\ ]}\underbrace{\lambda x_k \ldots \lambda x_l}_{m}.a = \langle L^{m-1}, ()\rangle(R^{x_k \ldots x_l}a)$

(T .2) $R^{x_0 \ldots x_j}a \ldots b = R^{x_0 \ldots x_j}a \ldots R^{x_0 \ldots x_j}b$

(T .2') $R^{x_0 \ldots x_j}(a \ldots b) = l_i$
where l_i is a new unique label in the script, such that
$l_i \mapsto (R^{x_i \ldots x_j}a) \ldots (R^{x_i \ldots x_j}b)$

(T .3) $R^{x_0 \ldots x_j}b = b$ *, if b is a constant*

(T .4) $R^{x_0 \ldots x_j}x_i = i$

Combinator names and labels are treated as constants.

1.1.1 Example of Compilation

The script:

$$\begin{array}{rcl} S & = & \lambda a.\lambda b.\lambda c.ac(bc) \\ K & = & \lambda k.\lambda l.k \\ I & = & \lambda i.i \\ & SKKI & \end{array}$$

forms the following environment:

$$\begin{array}{rcl} S & \mapsto & R^{[\ ]}[\![\lambda a.\lambda b.\lambda c.acl_1]\!] \\ K & \mapsto & R^{[\ ]}[\![\lambda k.\lambda l.k]\!] \\ I & \mapsto & R^{[\ ]}[\![\lambda i.i]\!] \end{array}$$

which by application of the compilation rules above translates to:

$$\begin{array}{rcl} S & \mapsto & \langle L^2(2\ 0\ l_1), ()\rangle \\ K & \mapsto & \langle L^1(1), ()\rangle \\ I & \mapsto & \langle L^0(0), ()\rangle \\ l_1 & \mapsto & 1\ 0 \end{array}$$

The expression to be evaluated is translated as

$$R^{[\,]}[\![SKKI]\!]$$

which generates $SKKI$ as compiled code.

1.2 Generating TIM Code

Now we present the compilation algorithm for TIM.
The script,

$$\begin{bmatrix} c_1 & \mapsto & [\![combinator_1]\!] \\ & \vdots & \\ c_n & \mapsto & [\![combinator_n]\!] \\ main\!-\!expression & & \end{bmatrix}$$

Compiles into TIM code as,

$$\rho = \begin{bmatrix} c_1 & \mapsto & B[\![combinator_1]\!] \\ & \vdots & \\ c_n & \mapsto & B[\![combinator_n]\!] \\ C[\![main\!-\!expression]\!] & & \end{bmatrix}$$

where compilation schemes B and C are given below:

(C.1) $B[\![\lambda a_1, \ldots a_n.body]\!] \Rightarrow$ [Take n; $C[\![body]\!][a_1, \ldots, a_n]$]

(C.2) $C[\![e_1\ e_2]\!][a_1, \ldots, a_n] \Rightarrow [P[\![e_2]\!][a_1, \ldots, a_n]; C[\![e_1]\!][a_1, \ldots, a_n]]$

(C.3) $C[\![atom]\!][a_1, \ldots, a_n] \Rightarrow E[\![atom]\!][a_1, \ldots, a_n]$

(C.4) $P[\![a_m]\!][a_1, \ldots, a_n] \Rightarrow$ [Push arg m]

(C.5) $P[\![c_i]\!][a_1, \ldots, a_n] \Rightarrow$ [Push combinator c_i]

(C.6) $P[\![e]\!][a_1, \ldots, a_n] \Rightarrow$ [Push label l], where l is a new (unique) label and the rule side-effects ρ thus $\rho := \rho[e' \mapsto (C[\![e]\!])]$ means ρ with entry $[e' \mapsto (C[\![e]\!])]$.

(C.7) $E[\![a_m]\!][a_1, \ldots, a_n] \Rightarrow$ [Enter arg m]

(C.8) $E[\![c_i]\!][a_1, \ldots, a_n] \Rightarrow$ [Enter combinator c_i]

The ";" used in rules (C.1) and (C.2) is overloaded. In rule (C.1) semi-colon is equivalent to *cons* (:) in a functional language, while in rule (C.2) semi-colon stands for *append* (++). Compilation of an expression into TIM generates a flat sequence of code, always. In rules C.4 and C.7 a variable a_m is replaced by m, its position in the list of variables $[a_1, \ldots, a_n]$.

1.2.1 Example of Compilation

The script:

$$\begin{aligned} S &= \lambda a.\lambda b.\lambda c.ac(bc) \\ K &= \lambda k.\lambda l.k \\ I &= \lambda i.i \\ &SKKI \end{aligned}$$

forms the following environment:

$$\begin{array}{rcl} S & \mapsto & B[\![\lambda a.\lambda b.\lambda c.acl_1]\!] \\ K & \mapsto & B[\![\lambda k.\lambda l.k]\!] \\ I & \mapsto & B[\![\lambda i.i]\!] \end{array}$$

which by application of the compilation rules above translates as:

$$\begin{array}{rcl} S & \mapsto & [\text{Take } 3; \text{Push label } l_1; \text{Push arg } 1; \text{Enter arg } 3] \\ K & \mapsto & [\text{Take } 2; \text{Enter arg } 1] \\ I & \mapsto & [\text{Take } 1; \text{Enter arg } 1] \\ l_1 & \mapsto & [\text{Push arg } 2; \text{Enter arg } 3] \end{array}$$

The expression to be evaluated generates the following TIM-code:

$C[\![SKKI]\!]\rho \mapsto$ Push Combinator I; Push Combinator K; Push Combinator K; Enter Combinator S;

2 Executing the Code

In this section we show how Categorical Multi-Combinators and TIM execute the code compiled by the compliation schemes above.

2.1 Categorical Multi-Combinator Rewriting Laws

The core of the Categorical Multi-Combinator machine is presented on page 71 of [8]. For a matter of convenience we will represent the multi-pair combinator, which forms evaluation environments as $(x_0, \ldots, x_n)$ and compositions, which represent closures, will be written as $\langle a, b\rangle$. Using this notation the kernel of the Categorical Multi-Combinator rewriting laws is:

(M*.1) $\langle n, (x_m, \cdots, x_1, x_0)\rangle \Rightarrow x_n$

(M*.2) $\langle x_0 x_1 x_2 \ldots x_n, y\rangle \Rightarrow \langle x_0, y\rangle \ldots \langle x_n, y\rangle$

(M*.3) $\langle L^n(y), (w_0, \ldots, w_m)\rangle x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \Rightarrow \langle y, (x_0, \cdots, x_n)\rangle\ x_{n+1} \cdots x_z$

The state of computation of a Categorical Multi-Combinator expression is represented by the expression itself. Rule (M*.1) performs environment look-up, this is the mechanism by which a variable fetches its value in the corresponding environment. (M*.2) is responsible for environment distribution. The rule (M*.3) performs environment formation: if during rewriting a label or a combinator reaches the leftmost position of the code we proceed a script look-up and enter the corresponding code in the definition environment. This can be expressed as

$$\langle l, y\rangle \Rightarrow \langle l_r, y\rangle$$

2.2 TIM states

The state of a TIM computation is a tuple

$$\langle \mathit{Code}, \mathit{Current\ Frame}, \mathit{Argument\ Stack}, \mathit{Frames}\rangle$$

The *Code* part is a sequence of TIM instructions. The *Current Frame* is the label (pointer) to a frame in *Frames*, which will be used for the evaluation of the *Code*. Specifically it is used to hold the values of free variables in the code. These values might be literal values, or closures represented by code-frame pairs. The *Argument Stack* is a stack of values, which are arguments to functions. *Frames* is a heap in which frames are stored. We use Miranda list notation to represent stacks.

The initial state of the machine is

$$\langle Code, (), [\,], [\,]\rangle$$

The state transition laws for TIM presented on page 36 of [2] are,

(s.1) $\langle[\text{Take } n; I], f_0, (a_1 : \ldots : a_n : A), F\rangle \Rightarrow \langle I, f, A, F[f \mapsto (a_1, \ldots, a_n)]\rangle$,
where f selects an unused frame

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

(s.3) $\langle[\text{Push label } l; I], f, A, F\rangle \Rightarrow \langle I, f, (\langle l, f\rangle : A), F\rangle$

(s.4) $\langle[\text{Push combinator } c; I], f, A, F\rangle \Rightarrow \langle I, f, (\langle c, ()\rangle : A), F\rangle$

(s.5) $\langle[\text{Enter arg } n], f, A, F[f \mapsto (\ldots, \langle c, f\rangle, \ldots)]\rangle \Rightarrow \langle c_r, f_n, A, F[f \mapsto (\ldots, \langle c, f_n\rangle, \ldots)]\rangle$

(s.6) $\langle[\text{Enter combinator } c], f, A, F\rangle \Rightarrow \langle c_r, (), A, F\rangle$

Note that in law (s.1) above 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.

3 C.M-C & TIM

The close relationship between TIM [2] and the original set of Categorical Multi-Combinators [6, 8] has been known to the first author for a long time, and has also been mentioned by other people [19]. This equivalence was also outlined in [11].

Our aim in this section is to make clear the relationship between TIM [2] and the original set of Categorical Multi-Combinators [6, 8]. We present two functions $\mathcal{C}$, translating from TIM to CMC and $\mathcal{T}$ going in the reverse direction. The translation functions and equivalence proofs we supply depend upon a number of simple properties of the form of the state and expressions produced by rewriting or executing compiled lambda expressions.

- All lambda expressions rewritten are of ground (non-functional) type. This is implicit in the rewriting rule for *Take* in TIM where it is assumed that there are always sufficient arguments upon the stack to perform a function application when required.
- All lambda expressions are assumed to be lambda-lifted before compilation (c.f. [3]), since this is intrinsic to the rewriting rules for Categorical Multi-Combinators. Examining the form of rewritten lambda expressions in CMC, it is safe to assume that in any composition $\langle l, r\rangle$, l is not a composition and that r is a multi-pair or tuple $(x_0, \ldots, x_n)$.

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 $\mathcal{C}(T_1)$, the Categorical Multi-Combinator equivalent rewrites in a sequence of zero or more steps to $\mathcal{C}(T_2)$ – **(Property I)**. We then show that if a CMC expression M_1 rewrites in one step to M_2 then $\mathcal{T}(M_1)$, the TIM equivalent rewrites in a sequence of zero or more steps to $\mathcal{T}(M_2)$ – **(Property II)**.

Property I **Property II**

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

Finally we show that $\mathcal{T}$ is a left inverse of $\mathcal{C}$, i.e. '$\mathcal{C}$ then $\mathcal{T}$' is the identity on TIM states. The other inverse relationship does not hold. We exhibit an example to show this, but we also show that it is an inverse modulo rewriting.

3.1 Translating TIM into C.M-C

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

(t.1) $\mathcal{C}(\langle I, f, [x_0, \ldots, x_z], F[f \mapsto (y_0, y_1, \ldots y_n)]\rangle) = \langle \tau_F I, (\tau_F y_0, \tau_F y_1, \ldots, \tau_F y_n)\rangle\ \tau_F x_0 \ldots \tau_F x_z$

(t.2) $\tau_F[\langle c_n, f\rangle] = \langle \tau_F c_n, (\tau_F y_0, \ldots, \tau_F y_m)\rangle$, *where* $f \mapsto (y_0, \ldots, y_m)$ *in* F

(t.3) $\tau_F[\text{Take } n; I] = L^{n-1}(\tau_F I)$

(t.4) $\tau_F[\text{Push arg } n; I] = \tau_F I\ (n-1)$

(t.5) $\tau_F[\text{Push label } l; I] = \tau_F I\ l'_r$

(t.6) $\tau_F[\text{Push combinator } c; I] = \tau_F I\ c'_r$

(t.7) $\tau_F[\text{Enter arg } n] = (n-1)$

(t.8) $\tau_F[\text{Enter combinator } c] = c'_r$

As we can observe τ_F in rules (t.3) to (t.8) is recursively invoked only on code sequences without any need for heap information, which is carried by F. For notational simplicity the subscript F, such as in τ_F, which stands for the heap of frames in TIM states, will be omitted in the sequel, if no misunderstanding can arise. Rule **t.1** above translates a TIM state into a top-level Categorical Multi-Combinator expression it is used to translate the expression under evaluation. In this case τ_F is ancillary to $\mathcal{C}$ and translates a code sequence into Categorical Multi-Combinator sub-expressions. We also apply τ_F to each entry in the TIM script in order to generate the corresponding C.M-C script thus,

$$\tau_F \begin{bmatrix} c & \mapsto & [\![combinator_1]\!] \\ & \vdots & \\ l & \mapsto & [\![label_1]\!] \\ & \vdots & \end{bmatrix} = \begin{bmatrix} c' & \mapsto & \tau_F[\![combinator_1]\!] \\ & \vdots & \\ l' & \mapsto & \tau_F[\![label_1]\!] \\ & \vdots & \end{bmatrix}$$

3.2 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 C.M-C. expressions is performed by the algorithm above. The following sub-sections prove the result clause by clause.

3.2.1 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

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

where f selects an unused frame

and

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

perform exactly the same transformation to the code. This equivalence can be shown formally as follows,

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

3.2.2 Push arg as Environment Look-up

The operation which allows a variable to fetch its value from its corresponding environment is expressed in TIM and C.M-C. as,

$$\begin{array}{rcl}
\langle[\text{Push arg } n; I], f, A, F[f \mapsto (\ldots, a_n, \ldots)]\rangle & \Rightarrow & \langle I, f, a_n, A, F[f \mapsto (\ldots, a_n, \ldots)]\rangle \\
\langle n, (x_m, \ldots, x_1, x_0)\rangle & \Rightarrow & x_n
\end{array}$$

Consider the behaviour of the two rules:

$$\begin{array}{rcl}
\mathcal{C}(\langle[\text{Push arg } n; I], f, (x_0 : \ldots), F[f \mapsto (a_m, \cdots)]\rangle) & \overset{t.1}{=} & \langle \tau[\text{Push arg } n; I], (\tau a_m, \cdots)\rangle\ \tau x_0 \ldots \\
\Downarrow s.2 & & \| \ t.4 \\
\mathcal{C}(\langle I, f, (a_n : x_0 \ldots), F[f \mapsto (a_m, \cdots)]\rangle) & & \langle \tau I\ (n-1), (\tau a_m, \cdots)\rangle\ \tau x_0 \ldots \\
\| \ t.1 & & \Downarrow M^*.2 \\
\langle \tau I, (\tau a_m, \cdots)\rangle\ \tau a_n\ \tau x_0 \ldots & & \langle \tau I, (\tau a_m, \cdots)\rangle\ \langle (n-1), (\tau a_m, \cdots)\rangle\ \tau x_0 \ldots \\
 & & \Downarrow M^*.1 \\
 & & \langle \tau I, (\tau a_m, \cdots)\rangle\ \tau a_n\ \tau x_0 \ldots
\end{array}$$

3.2.3 Push label as Environment Distribution

This operation is performed by the following laws in TIM and C.M-C, respectively,

$$\begin{array}{rcl}
\langle[\text{Push label } l; I], f, A, F\rangle & \Rightarrow & \langle I, f, \langle l, f\rangle : A, F\rangle \\
\langle x_0 x_1 x_2 \ldots x_n, y\rangle & \Rightarrow & \langle x_0, y\rangle\langle x_1, y\rangle \ldots \langle x_{n-1}, y\rangle\langle x_n, y\rangle
\end{array}$$

Right associated applications are removed from the TIM code and replaced by a label. Push label l builds a closure of the current frame and the label l.

Let us prove the operational equivalence between the laws above.

$$\begin{array}{rcl}
\mathcal{C}(\langle[\text{Push label } l; I], f, [x_0, \ldots], F[f \mapsto (a_m \cdots)]\rangle) & \overset{t.1}{=} & \langle \tau[\text{Push label } l; I], (\tau a_m, \cdots)\rangle\ \tau x_0 \ldots \\
\Downarrow s.3 & & \| \ t.5 \\
\mathcal{C}(\langle I, f, [\langle l, f\rangle,\ x_0 \ldots], F[f \mapsto (a_m \cdots)]\rangle) & & \langle \tau I\ l', (\tau a_m, \cdots)\rangle\ \tau x_0 \ldots \\
\| \ t.1 & & \Downarrow M^*.2 \\
\langle \tau I, (\tau a_m, \cdots)\rangle\ \tau\langle l, f\rangle\ \tau x_0 \ldots & & \langle \tau I, (\tau a_m, \cdots)\rangle\langle l', (\tau a_m, \cdots)\rangle\tau x_0 \ldots \\
\| \ t.2 & & \\
\langle \tau I, (\tau a_m, \cdots)\rangle\langle \tau l, (\tau a_m, \cdots)\rangle\tau x_0 \ldots & & \\
\| & & \\
\langle \tau I, (\tau a_m, \cdots)\rangle\langle l', (\tau a_m, \cdots)\rangle\tau x_0 \ldots & &
\end{array}$$

We recall that l' is the TIM label corresponding to l.

3.2.4 Push combinator **as Script Look-up**

In TIM and C.M-C 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 C.M-C whenever a combinator name reaches the leftmost outermost position in the code we enter the corresponding code.

$$\langle [\text{Push combinator } c; I], f, A, F\rangle \quad \Rightarrow \quad \langle I, f, \langle c, ()\rangle : A, F\rangle$$

Let us prove the operational equivalence between the laws above.

$$\begin{array}{rcl}
\mathcal{C}(\langle [\text{Push combinator } c; I], f, [x_0, \ldots], F[f \mapsto (a_m, \cdots)]\rangle) & \overset{t.1}{=} & \langle \tau[\text{Push combinator } c; I], (\tau a_m, \cdots)\rangle \; \tau x_0 \ldots \\
\Downarrow s.4 & & \| \; t.6 \\
\mathcal{C}(\langle I, f, [\langle c, ()\rangle, \; x_0, \ldots], F[f \mapsto (a_m, \cdots)]\rangle) & & \langle \tau I \; c', (\tau a_m, \cdots)\rangle \; \tau x_0 \ldots \\
\| \; t.1 & & \Downarrow M^*.2 \\
\langle \tau I, (\tau a_m, \cdots)\rangle \; \tau\langle c, ()\rangle \; \tau x_0 \ldots & & \langle \tau I, (\tau a_m, \cdots)\rangle \; \langle c', (\tau a_m, \cdots)\rangle \; \tau x_0 \ldots \\
\| \; t.2 & & \\
\langle \tau I, (\tau a_m, \cdots)\rangle \; \langle \tau c, \tau()\rangle \; \tau x_0 \ldots & & \\
\| & & \\
\langle \tau I, (\tau a_m, \cdots)\rangle \; \langle c', \tau()\rangle \; \tau x_0 \ldots & &
\end{array}$$

where c' is the TIM combinator corresponding to c. As combinators discharge the environments they are composed with we have both sides above operationally equal.

3.2.5 Enter arg **as Environment Look-up**

In the law,

$$\langle [\text{Enter arg } n], f, A, F[f \mapsto (\ldots, \langle c, f'\rangle, \ldots)]\rangle \Rightarrow \langle c_r, f', A, F[f \mapsto (\ldots, \langle c, f'\rangle, \ldots)]\rangle$$

Enter performs a similar transformation to the code as *Push arg n* above, i.e. an environment look-up. Let us see the state transition this law performs in C.M-C.

$$\begin{array}{rcl}
\mathcal{C}(\langle [\text{Enter arg } n], f, [x_0, \ldots], F[f \mapsto (\cdots \langle c, f'\rangle \cdots)]\rangle) & \overset{t.1}{=} & \langle \tau[\text{Enter arg } n], (\cdots, \tau\langle c, f'\rangle, \cdots)\rangle \; \tau x_0 \ldots \\
\Downarrow s.5 & & \| \; t.7 \\
\mathcal{C}(\langle c_r, f', [x_0, \ldots], F[f' \mapsto (y_0, \ldots, y_n)]\rangle) & & \langle (n-1), (\cdots, \tau\langle c, f'\rangle, \cdots)\rangle \; \tau x_0 \ldots \\
\| \; t.1 & & \Downarrow M^*.1 \\
\langle \tau c_r, (\tau y_0, \cdots, \tau y_m)\rangle \; \tau x_0 \ldots & & \tau\langle c, f'\rangle \; \tau x_0 \ldots \\
\| & & \| \; t.2 \\
\langle c_r', (\tau y_0, \cdots, \tau y_m)\rangle \; \tau x_0 \ldots & & \langle \tau c, (\tau y_0, \cdots, \tau y_m)\rangle \; \tau x_0 \ldots \\
& & \| \\
& & \langle c', (\tau y_0, \cdots, \tau y_m)\rangle \; \tau x_0 \ldots \\
& & \| \\
& & \langle c_r', (\tau y_0, \cdots, \tau y_m)\rangle \; \tau x_0 \ldots
\end{array}$$

3.2.6 Enter combinator **as Script Look-up**

The other role of the *Enter* combinator is simply to read the code for a function definition from the script, performing a lazy linking of the code, by the following law,

$$\langle [\text{Enter combinator } c], f, A, F \rangle \Rightarrow \langle c_r, (), A, F \rangle$$

This law is equivalent to the following state transformation in C.M-C.

$$\begin{tikzcd}
\mathcal{C}(\langle [\text{Enter combinator } c], f, [x_0,\ldots], F[f \mapsto (a_m,\cdots)] \rangle) \arrow[r, phantom, "\overset{t.1}{=}"] \arrow[d, Rightarrow, "s.6"] & \langle \tau[\text{Enter combinator } c], (\tau a_m,\cdots)\rangle\ \tau x_0 \ldots \arrow[d, equal, "t.8"] \\
\mathcal{C}(\langle c_r, (), [x_0,\ldots], F[f \mapsto (a_m,\ldots)] \rangle) \arrow[d, equal, "t.1"] & \langle c'_r, (\tau a_m,\cdots)\rangle\ \tau x_0 \ldots \\
\langle \tau c_r, (\tau a_m,\cdots)\rangle\ \tau x_0 \ldots \arrow[d, equal] & \\
\langle c'_r, (\tau a_m,\cdots)\rangle\ \tau x_0 \ldots &
\end{tikzcd}$$

where c'_r is the TIM code associated with combinator c'.

3.3 Translating C.M-C into TIM

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

(r.1) $\mathcal{T}(\langle e, (y_0,\ldots,y_m)\rangle\ w_0 \ldots w_k) = \langle \theta e, f, [\theta w_0,\ldots,\theta w_k], F[f \mapsto (\theta y_0,\ldots,\theta y_m)]\rangle$

(r.2) $\theta L^{n-1}(x) = [\text{Take } n; \theta x]$

(r.3) $\theta n = \text{Enter arg } (n+1)$

(r.4) $\theta c'_r = \text{Enter combinator } c$

(r.5) $\theta(e_0 e_1 \ldots e_m) = \Psi e_m; \ldots; \Psi e_1; \theta e_0$

(r.6) $\theta\langle n, (y_0,\ldots,y_m)\rangle = \theta y_n$

(r.7) $\theta\langle x, (y_0,\ldots,y_m)\rangle = \langle \theta x, f\rangle, \mathit{where}\ f \mapsto (\theta y_0,\ldots,\theta y_m)$

(r.8) $\Psi n = \text{Push arg } (n+1),$ if n is a variable

(r.9) $\Psi c'_r = \text{Push combinator } c,$ if c is a combinator

(r.10) $\Psi l'_r = \text{Push label } l$

$\mathcal{T}$ translates a top-level Categorical Multi-Combinator expression into a TIM state. θ and Ψ are ancillary functions which translate the code of a Categorical Multi-Combinator sub-expression into TIM-code. As we can observe in rule (C.2) above for compiling TIM code each subterm in an application is translated depending on its position in the term. Ψ is needed to reflect this difference, which does not exist in Categorical Multi-Combinators, into TIM. 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.5) 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 θ to each of the entries of the C.M-C script, thus

$$\theta\begin{bmatrix} c' & \mapsto & [\![combinator_1]\!] \\ & \vdots & \\ l' & \mapsto & [\![label_1]\!] \\ & \vdots & \end{bmatrix} = \begin{bmatrix} c & \mapsto & \theta[\![combinator_1]\!] \\ & \vdots & \\ l & \mapsto & \theta[\![label_1]\!] \\ & \vdots & \end{bmatrix}$$

The syntax of Categorical Multi-Combinator expressions which can arise from compilation or rewriting of compiled expressions shows us that in rule (r.1) e can either be a variable, an application, or an abstraction ($L^n(a)$ or c_i). We use this in proving property II below.

3.4 Proof of Property II

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

3.4.1 Environment Look-up

$$\begin{array}{ccc}
\mathcal{T}(\langle n, (\ldots, \langle y, (x_j, \cdots, x_l)\rangle, \ldots)\rangle w_0 \ldots w_k) & \overset{r.1}{=} & \langle \theta n, f, [\theta w_0, \ldots, \theta w_k], F[f \mapsto (\ldots, \langle \theta y, f'\rangle, \ldots)]\rangle \\
\Downarrow M^*.1 & & \| \; r.3 \\
\mathcal{T}(\langle y, (x_j, \cdots, x_l)\rangle w_0 \ldots w_k) & & \langle \text{Enter arg } (n+1), f, [\theta w_0, \ldots, \theta w_k], F[f \mapsto (\ldots, \langle \theta y, f'\rangle, \ldots)]\rangle \\
\| \; r.1 & & \Downarrow s.5 \\
\langle \theta y, f', [\theta w_0, \ldots, \theta w_k], F\rangle & & \langle \theta y, f', [\theta w_0, \ldots, \theta w_k], F[f \mapsto (\ldots, \langle \theta y, f'\rangle, \ldots)]\rangle
\end{array}$$

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 we can say that the two expressions above are equivalent.

3.4.2 Environment Distribution

$$\begin{array}{ccc}
\mathcal{T}(\langle x_0 \ldots x_n, (y_m, \ldots)\rangle w_0 \ldots) & \overset{r.1}{=} & \langle \theta(x_0 \ldots x_n), f, [\theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)]\rangle \\
\Downarrow M^*.2 & & \| \; r.5 \\
\mathcal{T}(\langle x_0, (y_m, \ldots)\rangle \ldots \langle x_n, (y_m, \ldots)\rangle w_0 \ldots) & & \langle \Psi x_n \ldots \theta x_0, f, [\theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)]\rangle \\
\| \; r.1 & & \\
\langle \theta x_0, f, [\theta\langle x_1, (y_m, \ldots)\rangle, \ldots, \theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)]\rangle & &
\end{array}$$

There are three cases to be considered depending on the form of the expression x_n,
x_n is a combinator c':

$$\begin{array}{cc}
\| \; r.7 & \| \; r.10 \\
\langle \theta x_0, f, [\ldots, \langle \theta c', ()\rangle, \theta w_0, \ldots], F\rangle & \langle \text{Push combinator } c \ldots \theta x_0, f, [\theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)]\rangle \\
\| & \Downarrow s.4 \\
\langle \theta x_0, f, [\ldots, \langle c, ()\rangle, \theta w_0, \ldots], F\rangle & \langle \Psi x_{n-1} \ldots \theta x_0, f, [\langle c, ()\rangle,\ \theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)]\rangle \\
 & \Downarrow *(s.2, s.3, \mathit{or}\ s.4) \\
 & \langle \theta x_0, f, [\ldots, \langle c, ()\rangle, \theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)]\rangle
\end{array}$$

x_n is a variable a:

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

x_n is a label l':

$$\begin{array}{cc}
\| \; r.6 & \| \; r.10 \\
\langle \theta x_0, f, [\ldots, \langle \theta l'_r, f \rangle, \theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)] \rangle & \langle \text{Push label } l \ldots \theta x_0, f, [\theta w_0, \ldots], F[f \mapsto (\theta, y_m \ldots)] \rangle \\
\| & \Downarrow s.3 \\
\langle \theta x_0, f, [\ldots, \langle l_r, f \rangle, \theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)] \rangle & \langle \Psi x_{n-1} \ldots \theta x_0, f, [\langle l_r, f \rangle, \; \theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)] \rangle \\
 & \Downarrow *(s.2, \; s.3, \; or \; s.4) \\
 & \langle \theta x_0, f, [\ldots, \langle l_r, f \rangle, \theta w_0, \ldots], F[f \mapsto (\theta y_m, \ldots)] \rangle
\end{array}$$

3.4.3 Multi β-Reduction

$$\begin{array}{ccc}
\mathcal{T}(\langle L^{n-1}(y), (w_0, \ldots, w_j) \rangle x_0 \ldots x_z) & \overset{r.1}{=} & \langle \theta L^{n-1}(y), f', [\theta x_0, \ldots, \theta x_z], F[f' \mapsto (\theta w_0, \ldots, w_j)] \rangle \\
\Downarrow M^*.3 & & \| \; r.2 \\
\mathcal{T}(\langle y, (x_0, \ldots, x_{n-1}) \rangle \; x_n \ldots x_z) & & \langle [\text{Take } n, \theta y], f', [\theta x_0, \ldots, \theta x_z], F[f' \mapsto (\theta w_0, \ldots, w_j)] \rangle \\
\| \; r.1 & & \Downarrow s.1 \\
\langle \theta y, f, [\theta x_n, \ldots, \theta x_z], F[f \mapsto (\theta x_0, \ldots, \theta x_{n-1})] & & \langle \theta y, f, [\theta x_n, \ldots, \theta x_z], F[f \mapsto (\theta x_0, \ldots, \theta x_{n-1})] \rangle
\end{array}$$

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.

3.5 $\mathcal{C}$ and $\mathcal{T}$

We show that the two translation functions $\mathcal{C}$ and $\mathcal{T}$ are related to each other. In particular we show that $\mathcal{T}$ is a left inverse of $\mathcal{C}$, but the reverse is not true. However, it is an inverse modulo expression rewriting, as explained in section 3.5.2.

3.5.1 $\mathcal{T} \circ \mathcal{C} = \mathit{Identity}$

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

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

Assuming that $\theta(\tau x) = x$, this equals

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

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

$$\begin{aligned}
\theta(\tau[\text{Take n}; I]) &\overset{t.3}{=} \theta(L^{n-1}(\tau I)) \\
&\overset{r.2}{=} [\text{Take n}; \theta(\tau I)] \\
\text{by induction } \theta(\tau I) = I, \text{so} \\
&\overset{\theta\tau}{=} [\text{Take n}; I]
\end{aligned}$$

$$\begin{aligned}
\theta(\tau[\text{Push arg n}; I]) &\overset{t.4}{=} \theta(\tau I; (n-1)) \\
&\overset{r.5}{=} [\Psi(n-1); \theta(\tau I)] \\
&\overset{r.8}{=} [\text{Push arg n}; \theta(\tau I)] \\
\text{by induction } \theta(\tau I) = I, \text{so} \\
&\overset{\theta\tau}{=} [\text{Push arg n}; I]
\end{aligned}$$

$$\begin{aligned}
\theta(\tau[\text{Push combinator c}; I]) &\overset{t.6}{=} \theta(\tau I; c'_r) \\
&\overset{r.5}{=} [\Psi c'_r; \theta(\tau I)] \\
&\overset{r.8}{=} [\text{Push combinator c}; \theta(\tau I)] \\
\text{by induction } \theta(\tau I) = I, \text{so} \\
&\overset{\theta\tau}{=} [\text{Push combinator c}; I]
\end{aligned}$$

$$\begin{aligned}
\theta(\tau[\text{Push label l}; I]) &\overset{t.5}{=} \theta(\tau I; l'_r) \\
&\overset{r.5}{=} [\Psi l'_r; \theta(\tau I)] \\
&\overset{r.8}{=} [\text{Push label l}; \theta(\tau I)] \\
\text{by induction } \theta(\tau I) = I, \text{so} \\
&\overset{\theta\tau}{=} [\text{Push label l}; I]
\end{aligned}$$

$$\begin{aligned}
\theta(\tau[\text{Enter arg } n]) &\overset{t.7}{=} \theta(n-1) \\
&\overset{r.3}{=} [\text{Enter arg } n]
\end{aligned}$$

$$\begin{aligned}
\theta(\tau[\text{Enter combinator } c]) &\overset{t.8}{=} \theta c'_r \\
&\overset{r.4}{=} [\text{Enter combinator } c]
\end{aligned}$$

$$\begin{aligned}
\theta(\tau[\langle c_n, f\rangle]) &\overset{t.2}{=} \theta\langle \tau c_n, (\tau y_0, \ldots, \tau y_m)\rangle, \mathit{where}\ f \mapsto (y_0, \ldots, y_m) \\
&\overset{r.6}{=} (\theta(\tau c_n), f), \mathit{where}\ f \mapsto (\theta(\tau y_0), \ldots, \theta(\tau y_m)) \\
\text{by induction } \theta(\tau c_n) = c_n, \text{so} \\
&\overset{\theta\tau}{=} \langle c_n, f\rangle
\end{aligned}$$

3.5.2 $\mathcal{C} \circ \mathcal{T} \approx$ *Identity*

We will show that $\mathcal{C}(\mathcal{T}\, x) = x$, where x is a Categorical Multi-Combinator expression does not hold, but if $\mathcal{C}(\mathcal{T}\, x) = x'$ then x rewrites to x' in a finite sequence of steps.

Firstly let us try to prove that $\mathcal{C}(\mathcal{T}\, x) = x$.

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

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

$$\begin{aligned}
\tau[\theta L^{n-1}(I)] &\overset{r.2}{=} \tau[\text{Take } n, \theta I] \\
&\overset{t.3}{=} L^{n-1}(\tau[\theta I]) \\
\text{by induction } \tau[\theta I] = I & \\
&\overset{\tau\theta}{=} L^{n-1}(I)
\end{aligned}$$

$$\begin{aligned}
\tau[\theta n] &\overset{r.3}{=} \tau[\text{Enter arg } (n+1)] \\
&\overset{t.7}{=} (n+1) - 1 \\
&= n
\end{aligned}$$

$$\begin{aligned}
\tau[\theta(e_0 e_1 \ldots e_m)] &\overset{r.5}{=} \tau[\Psi e_m; \ldots; \Psi e_1; \theta e_0] \\
\text{if } e_m \text{ is a variable } n &\overset{r.8}{=} \tau[\text{Push arg } (n+1); \ldots; \Psi e_1; \theta e_0] \\
&\overset{t.4}{=} \tau[\ldots; \Psi e_1; \theta e_0] n \\
&\overset{t.4-6}{=} e_0 e_1 \ldots e_m \\
\text{if } e_m \text{ is the code linked to a label } l' &\overset{r.10}{=} \tau[\text{Push label } l; \ldots; \Psi e_1; \theta e_0] \\
&\overset{t.5}{=} \tau[\ldots; \Psi e_1; \theta e_0] l'_r \\
&\overset{t.4-6}{=} e_0 e_1 \ldots e_m \\
\text{if } e_m \text{ is the code linked to a combinator } c' &\overset{r.9}{=} \tau[\text{Push combinator } c; \ldots; \Psi e_1; \theta e_0] \\
&\overset{t.6}{=} \tau[\ldots; \Psi e_1; \theta e_0] c'_r \\
&\overset{t.4-6}{=} e_0 e_1 \ldots e_m
\end{aligned}$$

$$\begin{aligned}
\tau[\theta(\langle x, (y_0, \ldots, y_m)\rangle)] &\overset{r.7}{=} \tau[\langle \theta x, f\rangle] \\
&\overset{t.2}{=} \langle \tau[\theta f], (\tau[\theta y_0], \ldots, \tau[\theta y_m])\rangle \\
&\overset{\tau\theta}{=} \langle x, (y_0, \ldots, y_m)\rangle
\end{aligned}$$

$$\begin{aligned}\tau[\theta(\langle n,(y_0,\ldots,y_m)\rangle)] &\overset{r.6}{=} \tau[\theta y_n]\\ &\overset{\tau\theta}{=} y_n\end{aligned}$$

In the last case we saw that $\tau[\theta x] \neq x$. However, we can see that if $\tau[\theta x] = x'$ then x rewrites to x' in a finite sequence of rewriting steps, so we have $\tau\theta x \approx x$, and $\mathcal{CT}\, x \approx x$, as required.

4 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.

The point of similarity of the two systems which distinguishes them from others is their coarse granularity of computation – a number of β-reductions can be performed in a single step in both systems. Both perform formation, distribution, look-up and deletion of multi-element environments as single computation steps.

The result shows that we can see Categorical Multi-Combinators as describing machine computations at a high level of abstraction, and also indicates that efficient implementations of this system are feasible. The authors are currently investigating a novel abstract machine, ΓCMC [9], based on Categorical Multi-Combinators and CM-CM [11, 17].

Acknowledgements

The authors are most grateful to Simon Peyton Jones for several corrections to and comments on earlier versions of this paper.

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

[1] P-L.Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman Publishing Ltd., 1986.

[2] J.Fairbairn and S.Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of Third International Conference on Functional Programming and Computer Architecture*, pages 34–45. LNCS 274, Springer Verlag, 1987.

[3] T.Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, January 1987.

[4] J.Lambek. From lambda-calculus to cartesian closed categories. In J.P.Seldin and J.R.Hindley, editors, *in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism.* Academic Press, 1980.

[5] R.D.Lins. A new formula for the execution of categorical combinators. In *Proceedings of 8th. International Conference on Automated Deduction*, pages 89–98. Springer Verlag, July 1986. LNCS 230.

[6] R.D.Lins. *On the Efficiency of Categorical Combinators in Applicative Languages.* PhD thesis, The University of Kent at Canterbury, October 1986.

[7] R. D. Lins. On the efficiency of categorial combinators as a rewriting system. *Software — Practice and Experience*, 17(8):547–559, August 1987.

[8] R.D.Lins. Categorical multi-combinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 60–79. Springer-Verlag, September 1987. LNCS 274.

[9] R.D.Lins & B.O.Lira. ΓCMC: A Novel Way to Implement Functional Languages. *in preparation*

[10] R.D.Lins & S.J.Thompson. Implementing SASL using categorical multi-combinators. *Software — Practice and Experience*, 20(11):1137–1165, November 1990.

[11] R.D.Lins & S.J.Thompson. CM-CM: A categorical multi-combinator machine. In *Proceedings of XVI LatinoAmerican Conference on Informatics*, vol(1) - pages 181-198, Assunsion - Paraguay, September 1990.

[12] M.A.Musicante & R.D.Lins. GMC - a graph multi-combinator machine. , *Euromicro Journal - Microprocessing and Microprogramming*, 31(1-5):81–84, North-Holland, April 1991.

[13] S.Peyton Jones. *The Implementation of Functional Languages.* Prentice Hall, 1987.

[14] S.Peyton Jones & D.Lester. *Implementing Functional Languages: A Tutorial.* Prentice Hall, 1992.

[15] D.Scott. Relating theories of the lambda-calculus. In J.P.Seldin and J.R.Hindley, editors, *in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism.* Academic Press, 1980.

[16] P.G.Soares & R.D.Lins. Implementing the G-machine. Technical Report 66, UKC Computing Lab. Report, The University of Kent at Canterbury, August 1989.

[17] S.J.Thompson & R.D.Lins. The Categorical Multi-Combinator Machine: CM-CM. to appear in *The Programming Journal*, April 92.

[18] D.A.Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9, 1979.

[19] G.Wraith and D.Bosley, November 1988. private communication.