Citation for published version


DOI

Link to record in KAR

https://kar.kent.ac.uk/21043/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (e.g., Creative Commons), permissions for further reuse of content should be sought from the publisher, author, or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact: researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
TCMC: A Novel Way of Compiling Functional Languages

Rafael D. Lins & Bruno O. Lira

Dept. de Informática - Universidade Federal de Pernambuco - Recife - Brasil
Computing Laboratory - The University of Kent - Canterbury - England

Abstract

The efficient compilation of functional languages has been shown to be a difficult task. The most successful implementations so far generate code in assembly language. This makes implementation extremely hard and machine dependent. In this paper we present TCMC, a new abstract machine, in which we transfer the control of the execution flow to C, as much as possible. TCMC takes advantage of the extremely low costs of procedure calls in modern RISC architectures. This produces a substantial improvement in performance, as we show here.

Introduction

Due to their semantic elegance, expressive power and ease in proving the correctness of programs, functional languages have been pointed out as a possible solution for the problem of programming known as the Software Crisis. In such languages programs are written as a set of function definitions and an expression, whose value is the result of the program. The evaluation is accomplished through consecutively rewriting the expression according to the functions definitions. Functional languages seem to be harder to implement than conventional imperative ones. At execution time, we must maintain complicated structures, such as unevaluated function applications, which allow us to work with higher-order functions and infinite lists.

The traditional way to implement lazy functional languages was graph interpretation of combinators, as introduced by Turner in [12]. The understanding of the evaluation mechanisms of these languages allowed implementation to move from interpretation towards compilation, with substantial gain in performance. Cardelli’s abstract machine FAM [3] developed for the compilation of strict functional languages was an important step in this search for efficiency.

Johnsson [10] developed a strategy for compiling lazy functional languages, described as an abstract stack machine, called the G-Machine. The basic principle of the G-Machine is to avoid generating graphs. The code generated by the G-Machine when executed produces time and space performance at least an order of magnitude faster than interpreted functional languages. The original G-Machine implemented at Chalmers, Göteborg, by Johnsson and his colleagues [10], generated code in VAX-780 Assembly language, which made implementation extremely hard and machine dependent. It was common sense in the community of implementation of functional languages that assembly language implementation was the price to pay if one wanted efficiency. The Chalmers LML compiler is still a reference in terms of performance of lazy functional languages. The G-machine way of controlling the execution flow and evaluation was followed by most of other implementations, even the ones based on different abstract machines as the Spineless G-Machine [1], the Spineless Tagless G-machine [9], TIM [2], and GM-C [8].

The first author has made several implementations of compiled functional languages in C [8, 6, 11, 7], which were close, but worse, in performance to the best assembly language implementations. All these C-based implementations were portable and simpler than the assembly ones. C was used as a macro-assembler and all “execution flow control” was made on a higher-level abstract machine.

In this paper we present TCMC, a new abstract machine, in which we transfer the execution flow control to C, as much as possible. The key idea behind TCMC is to take advantage of efficient context switching in modern architectures based on RISC, which is able to implement function calls at a very low cost. We also observed that the object code generated by C compilers is extremely neat and very fast. These factors lead us to try to translate each function definition into a procedure in C. It is obvious that not all scripts could be translated into C, if we wanted to have a lazy functional
language. However, it is safe to translate strict functions on all arguments that produce results of
ground type as procedures in C. The same is also true for arithmetic expressions wherever they appear.
This is the key for the efficiency of ΓCMC. A higher-level abstract machine is still needed to glue
together procedure calls, unevaluated expressions and functions, data-structures, etc. Categorical
Multi-Combinators [4, 7] served as a basis for the evaluation model of the ΓCMC abstract machine.
Our experience with GM-C [8] and CM-CM [6, 11] was fundamental for the design, implementation,
and optimisation of ΓCMC.

In this paper we also compare the performance of ΓCMC with the Chalmers LML compiler and
with GMC [8].

**Categorical Multi-Combinators**

In this section we present a brief introduction to Categorical Multi-Combinators [4], a rewriting system
which provides the computational model for ΓCMC. Later, we show how to compile a functional
language directly into ΓCMC code.

**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.

\[
\begin{align*}
&c_1 \equiv \text{combinator}_1 \\
&\quad \ldots \\
&c_n \equiv \text{combinator}_n \\
&\text{main-expression}
\end{align*}
\]

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

\[
\rho = [c_1 \mapsto \text{combinator}_1] \\
\quad \ldots \\
\quad [c_n \mapsto \text{combinator}_n]
\]

The main-expression is compiled separately as,

\[
\text{[[main-expression]]}_\rho
\]

In order properly to interpret recursion, we assume that the environment \(\rho\) contains the definition of
all combinators, so that recursive combinators produce recursive references through the environment.
The notation we use is: with each combinator \(c\) there is associated code \(c_\rho\), we suppress the environment
\(\rho\) when no confusion is possible.

**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 \(R^{e \to x_j}\) 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 \(R^{1.}\). For a matter of uniformity combinators will be
represented as composed with a dummy frame, \(\varnothing\), which can be seen as the identity frame.

\[
\begin{align*}
(T \ .1) & \ R^{1.} \ \lambda x_k \ldots x_1. a = \langle L^{m-1}(R^{e \to x_1} a), \varnothing \rangle \\
(T \ .2) & \ R^{e \to x_j} a \ldots b = R^{e \to x_j} a \ldots R^{e \to x_j} b \\
(T \ .3) & \ R^{e \to x_j} b = b, \text{ if } b \text{ is a constant} \\
(T \ .4) & \ R^{e \to x_j} x_i = i
\end{align*}
\]

Combinator names are treated as constants.
Example of Compilation

The script:

\[
S = \lambda a. \lambda b. \lambda c. ac(bc) \\
K = \lambda k. \lambda l. k \\
I = \lambda i. i
\]

forms the following environment:

\[
S \mapsto R\overline{1} [\lambda a. \lambda b. \lambda c. ac (bc)] \\
K \mapsto R\overline{1} [\lambda k. \lambda l. k] \\
I \mapsto R\overline{1} [\lambda i. i]
\]

which by application of the compilation rules above translates to:

\[
S \mapsto \{L^2(2 [0 (1 0)], () )\} \\
K \mapsto \{L^1(1), ()\} \\
I \mapsto \{L^1(0), ()\}
\]

The expression to be evaluated is translated as

\[
R\overline{1} [SKKI]
\]

which generates \( SKKI \) as compiled code.

Categorical Multi-Combinator Rewriting Laws

The core of the Categorical Multi-Combinator machine is presented on page 71 of [4]. 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) \( \langle x_0 x_1 x_2 \cdots x_n, y \rangle \Rightarrow \langle x_0, y \rangle \cdots \langle x_n, y \rangle \)

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

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 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

\( \langle l, y \rangle \Rightarrow \langle l_r, y \rangle \)

From CM-C into ΠCMC

In this section we give an overview of the ΠCMC evaluation mechanism.

If one observes the rewriting rules for Categorical Multi-Combinators above we see that rule (M*.3) is equivalent to \( \lambda \)-Calculus \( \beta \)-reduction in which substitutions are performed on demand. For
a matter of convenience we will structure the Categorical Multi-Combinator expression in two parts: the reduction stack $T$ and the heap $H$, where we place evaluation environments. The transition

$$\langle T, H \rangle \Rightarrow \langle T', H' \rangle$$

must be interpreted as: “when the machine arrives at state $\langle T, H \rangle$, it can get to state $\langle T', H' \rangle$. It is easy to see that the rewriting laws above can be rewritten as state transition rules:

1. $\langle \{y, e\}, c, H[e_i = (x_m, \ldots, x_0)] \rangle \Rightarrow \langle x_n \cdot c, H[e = (x_m, \ldots, x_0)] \rangle$
2. $\langle \{x_0 \ldots x_n, e\}, c, H[e_i = \ldots] \rangle \Rightarrow \langle \{x_0, e_i\} \ldots \{x_n, e_i\}, c, H[e_i = \ldots] \rangle$
3. $\langle \{L^n(y), e\}, x_0 \ldots x_n, c, H \rangle \Rightarrow \langle \{y, e\}, c, H[e_j = (x_0, \ldots, x_n)] \rangle$
4. $\langle \{l, e\}, c, H \rangle \Rightarrow \langle \{l, e\}, c, H \rangle$

Instead of manipulating references to environments directly as above we have a stack which keeps references to the current environment. Variables on the top position of the reduction stack fetch their values from the current environment. The current environment changes whenever a variable fetches a closure from the current environment or by creating a new environment via $\beta$-reduction. We call the environment stack $E$.

1. $\langle \{u, e\}, c, H[e_i = (x_m, e_m), \ldots, \{x_0, e_0\}] \rangle \Rightarrow \langle x_n \cdot c, H[e = (x_m, e_m), \ldots, \{x_0, e_0\}], e_0 \cdot E \rangle$
2. $\langle \{L^n(y), e\}, x_0 \ldots x_n, c, H \rangle \Rightarrow \langle \{y, e\}, H[e_j = (x_0, e_j), \ldots, \{x_n, e_j\}], e_j \cdot E \rangle$
3. $\langle \{l, e\}, c, H \rangle \Rightarrow \langle \{l, e\}, c, H \rangle$

**Example of Evaluation**

The expression $SKKI$, where $S$, $K$, and $I$ correspond to the following entries in the script

\[
\begin{align*}
S & \mapsto L^2(2 \ 0 \ 1 \ 0) \\
K & \mapsto L^1(1) \\
I & \mapsto L^3(0)
\end{align*}
\]

is evaluated as,

\[
\begin{align*}
\langle SKKI, H, E \rangle & \Rightarrow^3 \langle L^2(2 \ 0 \ 1 \ 0) \ K \ I, H, E \rangle \\
& \Rightarrow^3 \langle 2 \ 0 \ 1 \ 0, H[e_1 = (K \ K \ I)], e_1 \cdot E \rangle \\
& \Rightarrow^3 \langle K \ 0 \ 1 \ 0, H[e_1 = (K \ K \ I)], e_1 \cdot E \rangle \\
& \Rightarrow^3 \langle L^1(1) \ 0 \ 1 \ 0, H[e_1 = (K \ K \ I)], e_1 \cdot E \rangle \\
& \Rightarrow^3 \langle 1, H[e_2 = ((0, e_1), (1 \ 0, e_1))] \ [e_1 = (K \ K \ I)], e_2 \cdot e_1 \cdot E \rangle \\
& \Rightarrow^3 \langle 0, H[e_2 = ((0, e_1), (1 \ 0, e_1))] \ [e_1 = (K \ K \ I)], e_1 \cdot e_2 \cdot e_1 \cdot E \rangle \\
& \Rightarrow^3 \langle I, H[e_2 = ((0, e_1), (1 \ 0, e_1))] \ [e_1 = (K \ K \ I)], e_1 \cdot e_2 \cdot e_1 \cdot E \rangle
\end{align*}
\]

As there are no arguments on the evaluation stack we stop evaluation. The abstract machine presented above resembles the evaluation mechanism of CM-CM [6].

**Special Functions**

Strict functions on all arguments which produce results of ground type are called special. These functions will fetch their arguments from the evaluation stack and return the result of evaluation to the top of $T$. The evaluation of special functions happens outside $\GammaCMC$. All $\GammaCMC$ does is to prepare the arguments for them and receive the result. We introduce a new state transition law for special functions:

4. $\langle f^* x_0 \ldots x_n, c, H, e \cdot E \rangle \Rightarrow \langle f, c, H, e \cdot E \rangle$

where $f_r = f^*(x_0, \ldots, x_n)$ and $x_i$ is the weak head normal form of $x_i$. 

---

4
Arithmetic expressions, in general, will be lifted from the code and will be treated in a similar way to special functions. For instance,

\[ S' = \lambda a. \lambda b. \lambda c. a(b + c) \]

generates the following script:

\[ S' \mapsto L^2(2(f_1 1 0)) \]
\[ f_1 \mapsto (1' + 0') \]

From Interpretation to Compilation

The Categorical Multi-Combinator structures which appear on T are now replaced by code which when executed will generate a corresponding data structure on T. This data structure is interpreted by using the state transition laws above. A variable n, for example, is generated on top of T by using a MKTvar(n) instruction.

The code will always try to predict the behaviour of evaluation and avoid generating intermediate expressions, as much as possible. The novel aspect of ΓCMC if compared with its predecessors lies on translating special functions into procedures in C. In the next section we see ΓCMC in more details.

We must keep in mind the code sequences generated will perform operations equivalent to the naive ΓCMC machine in this section.

Compiling into ΓCMC Code

We present here the complete set of direct compilation rules for the kernel of ΓCMC.

A program in ΓCMC is formed by a set of function definitions plus an expression, which we want to evaluate, as follows:

\[ f_1 x_1 \ldots x_n = \text{body-of } f_1 \]
\[ f_2 x_1 \ldots x_n = \text{body-of } f_2 \]
\[ \vdots \]
\[ f_i x_1 \ldots x_n = \text{body-of } f_i \]

expression?

The expression to be evaluated is compiled by scheme E called as:

\[ E[expression] \]

Strict functions on all arguments which produce results of ground type are called special. These functions will be compiled directly as procedures in C. Special functions are compiled as:

\[ f_i x_1 \ldots x_n = \text{body-of } f_i \]
\[ f_i \mapsto S[\text{body-of } f_i] \]

Ordinary functions are compiled as:

\[ f_j x_1 \ldots x_n = \text{body-of } f_j \]
\[ f_j \mapsto T[x_1 \ldots x_n][\text{body-of } f_j] \]

Scheme E

This scheme is responsible for the printing routine and driving the evaluation mechanism.

1. \[ E[k] = \text{printf}(k); \quad \text{if } k \text{ is a constant} \]
2. \[ E[a + b] = \text{t1 = S'[a]; t2 = S'[b]; printf(t1 + t2);} \]
3. \[ E[\text{if } a > b \text{ then } c \text{ else } d] = \text{if_true(A, C, D);} \]
   where A \mapsto T^0[a]; t1 = \text{*(top T - 1)} \mapsto \text{rem.value; T}^0[b]; t2 = \text{*(top T - 1)} \mapsto \text{rem.value; t1 > t2; C} \mapsto E[c]; D \mapsto E[d]; \]
4. \[ E[f_i x_0 \ldots x_n] = \text{printf}(f_i[S'[x_0], \ldots, S'[x_n]); \quad \text{if } f_i \text{ is a special function.} \]
5. \[ E[f_i \ldots] = T^0[f_i \ldots] \text{ printf();} \]
Scheme $S$

This scheme is responsible for starting-up the compilation of special functions generating procedures in C.

1. $S[k] = \text{return}(k);$ if $k$ is a constant
2. $S[x] = \text{return}(x);$ if $x$ is a variable
3. $S[a+b] = \text{return}(S'[a] + S'[b]);$
4. $S[if \ (a > b) \ then \ c \ else \ d] = if \ (S'[a] > S'[b]) \ \{S[c]\}; else\{S[d]\};$
5. $S[f_1(x_0, \ldots, x_j) = \text{return}(f_1(S'[x_0], \ldots, S'[x_j]));$
6. $S'[f_1x_0, \ldots, x_j] = T^{\downarrow}(f_1x_0, \ldots, x_j) \ \text{return}(((\ast (\text{top}T - -)) \rightarrow \text{rem}.value));$

Scheme $S'$

This scheme is ancillary to $S$ and is responsible for the compilation of inner parts of the body of a special function, generating parts of procedure code in C.

1. $S'[k] = k$ if $k$ is a constant
2. $S'[x] = x$ if $x$ is a variable
3. $S'[a+b] = (S'[a] + S'[b])$
4. $S'[if \ (a > b) \ then \ c \ else \ d] = if \ (S'[a] > S'[b]) \ \{S[c]\}; else\{S[d]\};$
5. $S'[f_1(x_0, \ldots, x_j) = f_1(S'[x_0], \ldots, S'[x_j]);$
6. $S'[f_1x_0, \ldots, x_j] = T^{\downarrow}(f_1x_0, \ldots, x_j) \ \text{return}(((\ast (\text{top}T - -)) \rightarrow \text{rem}.value));$

Example of Compilation

Let us show an example of special function compilation. If we have the script:

```plaintext
fib n = if n<2 then 1 else fib(n-1) + fib(n-2)
fib 20
```

it will be compiled as,

```plaintext
C[fib 20] \xrightarrow{c_1^4} \text{printf}(\text{fib}(S'[20]));
\xrightarrow{c_1^1} \text{printf}(\text{fib}(20));
```

Now we compile

```plaintext
fib n = if n<2 then 1 else fib(n-1) + fib(n-2)
```

by using scheme $S$.

```plaintext
fib \mapsto S[if \ n < 2 \ then \ 1 \ else \ fib(n-1) + fib(n-2)]
```

```plaintext
\mapsto c_1^4 \ (S'[n] < S'[2]) \ \{S[1]\}; else\{S[fib(n-1) + fib(n-2)]\};
\mapsto c_1^2 \ (n < S'[2]) \ \{S[1]\}; else\{S[fib(n-1) + fib(n-2)]\};
\mapsto c_1^1 \ (n < 2) \ \{S[1]\}; else\{S[fib(n-1) + fib(n-2)]\};
\mapsto \ (n < 2) \ \{return(1)\}; else\{S[fib(n-1) + fib(n-2)]\};
\mapsto c_1^3 \ (n < 2) \ \{return(1)\}; else\{return(S'[fib(n-1)] + S'[fib(n-2)])\};
\mapsto c_1^5 \ (n < 2) \ \{return(1)\}; else\{return(fib(S'[n-1]) + S'[fib(n-2)])\};
\mapsto c_1^6 \ (n < 2) \ \{return(1)\}; else\{return(fib(S'[n-1]) + fib(S'[n-2]))\};
\mapsto c_1^3 \ (n < 2) \ \{return(1)\}; else\{return(fib(n-S'[1])) + fib(S'[n-2]))\};
\mapsto c_1^2 \ (n < 2) \ \{return(1)\}; else\{return(fib(n-S'[1])) + fib(S'[n-2]))\};
\mapsto c_1^3 \ (n < 2) \ \{return(1)\}; else\{return(fib(n-1) + fib(S'[n-2]))\};
\mapsto c_1^2 \ (n < 2) \ \{return(1)\}; else\{return(fib(n-1) + fib(n-S'[2]))\};
\mapsto c_1^3 \ (n < 2) \ \{return(1)\}; else\{return(fib(n-1) + fib(n-2))\};
```
As we can see the result of compilation is a procedure in C, which needs only a heading with type declarations to be compiled and executed by the C machine.

Scheme \( T \)
This scheme is responsible for the compilation of ordinary functions and generates code which is handled by the abstract machine. We assume the arity of a function \( f_n \) to be \( n + 1 \).

1. \( T_{\text{gen}}^{g_{n-1}}[f_n x_0 \ldots x_n] = T_{\text{gen}}^{g_{n-1}}[x_0] \ldots T_{\text{gen}}^{g_{n-1}}[x_n] \)
2. \( T_{\text{gen}}^{g_{n-1}}[f_n x_0 \ldots x_n] = \text{MKTcte}(f_n(Z^{g_{n-2}} x_0, \ldots, Z^{g_{n-2}} x_n)) \); if \( f_n \) is a special function
3. \( T_{\text{gen}}^{g_{n-1}}[f_n x_0 \ldots x_n] = \text{MKEcomp}(g_{n-2}[x_0] \ldots g_{n-2}[x_n] \text{Pushfun}(f_n); \text{Popenv}; \)
4. \( T_{\text{gen}}^{g_{n-1}}[f_n] = \text{eval}(0) \ldots \text{eval}(n); \text{MKT}(n, f_n((\text{topT} \rightarrow \text{rem.value}), \ldots, (((\text{topT} - n) \rightarrow \text{rem.value}) \ldots) \text{Popenv}; \)
5. \( T_{\text{gen}}^{g_{n-1}}[f_n] = \text{MKEenv}(n + 1); \text{Pushfun}(f_n); \text{Popenv}; \)

Scheme \( G \)
This scheme generates code which when executed fills the fields of a cell in an evaluation environment.

1. \( G^{g_{n-2}}[k] = \text{MKEcte}(k); \)
2. \( G^{g_{n-2}}[x_i] = \text{MKTvar}(x_i); \)
3. \( G^{g_{n-2}}[a + b] = \text{MKEcte}(Z^{g_{n-1}}[a] + Z^{g_{n-1}}[b]); \)
4. \( G^{g_{n-2}}[a + b] = \text{MKTcomp}(a, i); \)
5. \( G^{g_{n-2}}[a + b] = \text{MKTEcomp}(a, i); \)
6. \( G^{g_{n-2}}[a + b] = \text{MKEpc}(a, i); \)
7. \( G^{g_{n-2}}[b] = \text{MKEcomp}(b, i); \)
8. \( G^{g_{n-2}}[f_n] = \text{MKEcte}(f_n); \)

Scheme \( T' \)
This scheme produces code which when executed generates cells on the top of the T-stack.

1. \( T_{\text{gen}}^{g_{n-1}}[k] = \text{MKTcte}(k); \)
2. \( T_{\text{gen}}^{g_{n-1}}[x_i] = \text{MKTvar}(i); \)
3. \( T_{\text{gen}}^{g_{n-1}}[a + b] = \text{MKTcte}(Z^{g_{n-1}}[a] + Z^{g_{n-1}}[b]); \)
4. \( T_{\text{gen}}^{g_{n-1}}[a + b] = \text{MKTcomp}(A); \)
5. \( T_{\text{gen}}^{g_{n-1}}[a + b] = \text{MKTEcomp}(A); \)
6. \( T_{\text{gen}}^{g_{n-1}}[a + b] = \text{MKEpc}(A); \)
7. \( T_{\text{gen}}^{g_{n-1}}[a + b] = \text{MKTcomp}(A); \)
8. \( T_{\text{gen}}^{g_{n-1}}[f_n] = \text{MKEcte}(f_n); \)
Scheme $\mathcal{E}$

This scheme makes parameters ready for special functions or arithmetic expressions whenever called inside an ordinary function.

1. $\mathcal{E}[x^0] = k$;
2. $\mathcal{E}[x] = ((\ast \text{topE} + (\ast \text{topE}) \rightarrow \text{tipo} - 1 - i \rightarrow \text{rem.graph}) \rightarrow \text{rem.value})$ if $x_i$ is already evaluated.
3. $\mathcal{E}[x] = \text{eval.env}(i); \text{return}(\ast \text{topT} \rightarrow \text{rem.value});$
4. $\mathcal{E}[f\ a\ \ldots\ b] = f_i(\mathcal{E}[a],\ldots,\mathcal{E}[b])$ if $f_i$ is a special function.
5. $\mathcal{E}[a\ \ldots\ b] = T^x[a\ \ldots\ b] \text{return}((\ast \text{topT} \rightarrow \text{rem.value}))$.
6. $\mathcal{E}[a + b] = \mathcal{E}[a] + \mathcal{E}[b]$.

Scheme $\mathcal{E}'$

This scheme makes parameters ready for special functions or arithmetic expressions whenever called inside a cell generating scheme.

1. $\mathcal{E}'[x^0] = k$;
2. $\mathcal{E}'[x] = ((\ast \text{topE} - 1) + (\ast \text{topE} - 1) \rightarrow \text{tipo} - 1 - i \rightarrow \text{rem.graph} \rightarrow \text{rem.value})$ if $x_i$ is already evaluated.
3. $\mathcal{E}'[x] = \text{eval.env}(i); \text{return}(\ast \text{topT} \rightarrow \text{rem.value});$
4. $\mathcal{E}'[f\ a\ \ldots\ b] = f_i(\mathcal{E}'[a],\ldots,\mathcal{E}'[b])$ if $f_i$ is a special function.
5. $\mathcal{E}'[a\ \ldots\ b] = T^x[a\ \ldots\ b] \text{return}((\ast \text{topT} \rightarrow \text{rem.value}))$.
6. $\mathcal{E}'[a + b] = \mathcal{E}'[a] + \mathcal{E}'[b]$.

Example of Compilation

Let us show an example of compilation of an ordinary function. If we have the script:

```python
fib n = if n<2 then 1 else fib(n-1) + fib(n-2)
twice f x = f (f x)
twice fib 5
```

it will be compiled as,

$$\mathcal{E}[\text{twice fib 5}] \xrightarrow{\mathcal{T}} 1[\text{twice fib 5}]; \text{print}();$$

where $A$ is:

$$A \mapsto T^{J}[\text{fib}]$$

and for $\mathcal{E}'$

$$\mathcal{E}'[\text{twice fib 5}] \xrightarrow{\mathcal{T}'} 1[\text{twice fib 5}]; \text{print}();$$

where $A'$ is:

$$A' \mapsto T^{J}[\text{fib}]$$

fib is as in the previous example of compilation above.
State Transition Laws

We present ΓCMC as a state transition machine. A state of ΓCMC is a 5-uple
\[ \langle C, T, H, O, E \rangle \]
in which each component is interpreted in the following way:

C: The code to be executed.
   This code is generated by the translation rules presented by the compilation schemes above.
T: The reduction stack. The top of T points to the part of the graph to be evaluated.
H: The heap where graphs are stored. The notation \( H[d = c_1 \ldots c_n] \) means that there is in \( H \) a
   \( n \)-component cell named \( d \). The fields of \( d \) are filled with \( c_1 \ldots c_n \), in this order.
   Cells are fully-boxed.
O: The output.
E: The environment stack. Its top contains a reference to the current environment.

ΓCMC is defined as a set of transition rules. The transition
\[ \langle C, T, H, O, E \rangle \Rightarrow \langle C', T', H', O', E' \rangle \]
must be interpreted as: “when the machine arrives at state \( \langle C, T, H, O, E \rangle \), it can get to state
\( \langle C', T', H', O', E' \rangle \)”.

We present here the complete set of state transition laws for the kernel of ΓCMC:

```
1. (print c, d.T, H[d=k], O, E) \Rightarrow \langle c, T, H[d=k], k.O, E \rangle
2. (eval c, d.T, H[d=k], O, E) \Rightarrow \langle c, d.T, H[d=k], O, E \rangle
3. (eval c, d.T, H[d=(A,c)], O, E) \Rightarrow \langle A.Popenv c, T, H[d=(A,c)], O, c.E \rangle
4. (eval c, d.T, H[d=f_n], O, E) \Rightarrow \langle f_n c, T, H[d=f_n], O, E \rangle
5. (eval c, d.T, H[d=k], O, E) \Rightarrow \langle c, d.T, H[d=k], O, E \rangle
6. (eval c, d.T, H[d=(A,c)], O, E) \Rightarrow \langle A.Pop(i).c, d.T, H[d=(A,c)], O, c.E \rangle
7. (eval c, d.T, H[d=f_n], O, E) \Rightarrow \langle c, d.T, H[d=f_n], O, E \rangle
8. (eval c, d.T, H[e..f_n], O, c.E) \Rightarrow \langle f_n.T, H[e..f_n], O, c.E \rangle
9. (eval c, d.T, H[e..f_n], O, c.E) \Rightarrow \langle eval c, u.T, H[e..f_n], O, c.E \rangle
10. (MKenv n).c, d_1 \ldots d_n \ldots d_m.T, H, O, E) \Rightarrow \langle c, d_1 \ldots d_n \ldots d_m.T, H[e=d_1 \ldots d_n], O, E \rangle
11. (Popenv c, T, H, O, c.E) \Rightarrow \langle c, T, H, O, E \rangle
12. (Pop(i).c, d_1 \ldots d_i \ldots d_k.T, H, O, E) \Rightarrow \langle c, d_1 \ldots d_i \ldots d_k.T, H, O, E \rangle
13. (Pushfun(f).c, T, H, O, E) \Rightarrow \langle f.c, T, H, O, E \rangle
14. (If_true=True, A, B, c, T, H, O, E) \Rightarrow \langle A.c, T, H, O, E \rangle
15. (If_true=False, A, B, c, T, H, O, E) \Rightarrow \langle B.c, T, H, O, E \rangle
16. (MKTvar k).c, T, H[e=c_0 \ldots c_n], O, c.E) \Rightarrow \langle c, d.T, H[d=c_0 \ldots c_n], O, c.E \rangle
17. (MKTcomp(P).c, T, H, O, c.E) \Rightarrow \langle c, d.T, H[d=(P,c)], O, c.E \rangle
18. (MKTpc(A, c, T, H, O, E) \Rightarrow \langle c, d.T, H[d=A], O, E \rangle
19. (MKTcte(k).c, T, H, O, E) \Rightarrow \langle c, d.T, H[d=k], O, E \rangle
20. (MKT((n,A).c, d_0 \ldots d_n, T).H[d_0 \ldots d_n], O, E) \Rightarrow \langle c, d_0 \ldots d_n, T, H[d_0 \ldots d_n], O, E \rangle
21. (MKEval(i).c, T, H[e_0 \ldots e_n], O, c_0, c.E) \Rightarrow \langle c, T, H[e_0 \ldots e_n], O, c_0, c.E \rangle
22. (MKEval(i).c, T, H[e_0 \ldots e_n], O, c_0, c.E) \Rightarrow \langle c, T, H[e_0 \ldots e_n], O, c_0, c.E \rangle
23. (MKCell(n).c, T, H, O, E) \Rightarrow \langle c, d.T, H[e=u_0 \ldots u_n], O, E \rangle
24. (MKCell((A,c).c, T, H[e_0 \ldots e_n], O, c_0, c.E) \Rightarrow \langle c, T, H[e_0 \ldots e_n], O, c_0, c.E \rangle
25. (MKEp(A, c, T, H[e_0 \ldots e_n], O, c.E) \Rightarrow \langle c, d.T, H[e_0 \ldots e_n], O, c.E \rangle
26. (If_true(True, B, C, i).c, T, H[e_0 \ldots e_n], O, c_0, c.E) \Rightarrow \langle c, T, H[e_0 \ldots e_n], O, c_0, c.E \rangle
27. (If_true(False, B, C, i).c, T, H[e_0 \ldots e_n], O, c_0, c.E) \Rightarrow \langle c, T, H[e_0 \ldots e_n], O, c_0, c.E \rangle
```
Example of Evaluation

As we saw in the examples of compilation above, the program:

```
fib n = if n<2 then 1 else fib(n-1) + fib(n-2)
twice f x = f (f x)
twice fib 5
```

compiled as,

```
twice fib 5 ← MKEcell(2); MKEpct(A,1); MKEcte(5,0); Pushfun(twice); Popenv; print();
A ← eval'(0); MKT(0, fib((*(topT)) ← rem.value));
twice ← MKTcomp(A'); eval_env(1);
A' ← MKTvar(0); eval_env(1);
fib ← if (n < 2) {return(1)}; else {return(fib(n-1) + fib(n-2))};
```

The initial state of the machine is:

```
{MKEcell(2); MKEpct(A,1); MKEcte(5,0); Pushfun(twice); Popenv; print(); ,T,H,O,E}
```

executing this code using the state transition laws above we have,

```
1) {MKEpct(A,1); MKEcte(5,0); Pushfun(twice); Popenv; print(); ,T,H[c_0 = 1 \ 0],O,c_0,E}
2) {MKEcte(5,0); Pushfun(twice); Popenv; print(); ,T,H[c_0 = A \ d_0],O,c_0,E}
3) {Pushfun(twice); Popenv; print(); ,T,H[d_0 = 5][c_0 = A \ d_0],O,c_0,E}
4) {MKTcomp(A'); eval_env(1); Popenv; print(); ,T,H[d_0 = 5][c_0 = A \ d_0],O,c_0,E}
5) {eval'(0); MKTk(0, fib((*(topT)) ← rem.value)); Popenv; print(); ,
d_1,T,H[d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
6) {MKTvar(0); eval_env(1); Pop(0); MKTk(0, fib((*(topT)) ← rem.value)); Popenv; print(); ,
d_1,T,H[d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
7) {eval'(0); MKTk(0, fib((*(topT)) ← rem.value)); Pop(0); MKTk(0, fib((*(topT)) ← rem.value)); Popenv; print(); ,
d_0,d_1,T,H[d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
8) {MKTk(0, fib((*(topT)) ← rem.value)); Pop(0); MKTk(0, fib((*(topT)) ← rem.value)); Popenv; print(); ,
d_0,d_1,T,H[d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
9) {Pop(0); MKTk(0, fib((*(topT)) ← rem.value)); Popenv; print(); ,
d_2,d_1,T,H[d_2 = 8][d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
10) {MKTk(0, fib((*(topT)) ← rem.value)); Popenv; print(); ,
d_2,d_1,T,H[d_2 = 8][d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
11) {Popenv; print(); ,d_2,T,H[d_2 = 34][d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
12) {print(); ,d_2,T,H[d_2 = 34][d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],O,c_0,E}
13) { ,T,H[d_3 = 34][d_2 = 8][d_1 = (A',c_0)][d_0 = 5][c_0 = A \ d_0],34,O,E}
```

Compiling Lists

Now we enrich ICMC with lists. A new compilation scheme, called L, is introduced. Some of the previous compilation schemes need to be extended.
Scheme $\mathcal{E}$

6. $\mathcal{E}([]) = \text{printf}([])$;
7. $\mathcal{E}[a : b] = \mathcal{E}[a] \mathcal{E}[b] \text{ print}();$
8. $\mathcal{E}[\text{Hd}(a \ldots b)] = \mathcal{T}^2[a \ldots b] \text{ Hd; printf}((s(\text{top} T = -i)) \rightarrow \text{ rem value});$
9. $\mathcal{E}[\text{Tl}(a \ldots b)] = \mathcal{T}^3[a \ldots b] \text{ Tl; print}();$
10. $\mathcal{E}[\text{Hd}(a : b)] = \mathcal{E}[a]$
11. $\mathcal{E}[\text{Tl}(a : b)] = \mathcal{E}[b]$

Scheme $\mathcal{T}$

13. $\mathcal{T}^{x_0 \ldots x_j}[a] = \text{ MKTlv};$
14. $\mathcal{T}^{x_0 \ldots x_j}[\text{Hd} x_i] = \text{ Ehdi}(i);$
15. $\mathcal{T}^{x_0 \ldots x_j}[\text{Hd}(a : b)] = \mathcal{T}^{x_0 \ldots x_j}[a]$
16. $\mathcal{T}^{x_0 \ldots x_j}[\text{Hd}(a \ldots b)] = \mathcal{T}^{x_0 \ldots x_j}[a \ldots b] \text{ Hd};$
17. $\mathcal{T}^{x_0 \ldots x_j}[a \text{ (Hd } b \ldots)] = \mathcal{T}^{x_0 \ldots x_j}[b \ldots] \text{ Hd; } \mathcal{T}^{x_0 \ldots x_j}[a]$
18. $\mathcal{T}^{x_0 \ldots x_j}[\text{Tl} x_i] = \text{ Etl}(i);$
19. $\mathcal{T}^{x_0 \ldots x_j}[\text{Tl}(a : b)] = \mathcal{T}^{x_0 \ldots x_j}[b]$
20. $\mathcal{T}^{x_0 \ldots x_j}[\text{Tl}(a \ldots b)] = \mathcal{T}^{x_0 \ldots x_j}[a \ldots b] \text{ Tl};$
21. $\mathcal{T}^{x_0 \ldots x_j}[a \text{ (Tl } b \ldots)] = \mathcal{T}^{x_0 \ldots x_j}[b \ldots] \text{ Tl; } \mathcal{T}^{x_0 \ldots x_j}[a]$
22. $\mathcal{T}^{x_0 \ldots x_j}[a : b] = \mathcal{T}^{x_0 \ldots x_j}[a] \mathcal{T}^{x_0 \ldots x_j}[b] \text{ MKcons;}$
23. $\mathcal{T}^{x_0 \ldots x_j}[\text{if } x_i = [ \text{ then } b \text{ else } c] = \text{ Enull}(i, i_1); \mathcal{T}^{x_0 \ldots x_j}[b] \text{ JMP } i_2; i_2 : \mathcal{T}^{x_0 \ldots x_j}[c] \text{ l_2};$

Scheme $\mathcal{G}$

9. $\mathcal{G}^{x_0 \ldots x_j}[a] = \text{ MKEv}(a);$
10. $\mathcal{G}^{x_0 \ldots x_j}[\text{Hd} x_i] = \text{ Ehdi}(i, j);$
11. $\mathcal{G}^{x_0 \ldots x_j}[\text{Hd}(a : b)] = \mathcal{G}^{x_0 \ldots x_j}[a]$
12. $\mathcal{G}^{x_0 \ldots x_j}[\text{Tl} x_i] = \text{ Etl}(i, j);$
13. $\mathcal{G}^{x_0 \ldots x_j}[\text{Tl}(a : b)] = \mathcal{G}^{x_0 \ldots x_j}[b]$j
14. $\mathcal{G}^{x_0 \ldots x_j}[a : b] = \mathcal{L}^{x_0 \ldots x_j}[a] \mathcal{L}^{x_0 \ldots x_j}[b] \text{ MKEv}(b);$

Scheme $\mathcal{T'}$

9. $\mathcal{T'}^{x_0 \ldots x_j}[a] = \text{ MKTlv};$
10. $\mathcal{T'}^{x_0 \ldots x_j}[\text{Hd} x_i] = \text{ Ehdi}(i);$
11. $\mathcal{T'}^{x_0 \ldots x_j}[\text{Hd}(a : b)] = \mathcal{T'}^{x_0 \ldots x_j}[a]$
12. $\mathcal{T'}^{x_0 \ldots x_j}[\text{Tl} x_i] = \text{ Etl}(i);$
13. $\mathcal{T'}^{x_0 \ldots x_j}[\text{Tl}(a : b)] = \mathcal{T'}^{x_0 \ldots x_j}[b]$
14. $\mathcal{T'}^{x_0 \ldots x_j}[a : b] = \text{ MKTcomp}(A);$

Scheme $\mathcal{L}$

1. $\mathcal{L}^{x_0 \ldots x_j}[x_i] = \text{ MKTvar}(i);$
2. $\mathcal{L}^{x_0 \ldots x_j}[k] = \text{ MKTcet}(k);$
3. $\mathcal{L}^{x_0 \ldots x_j}[a + b] = \text{ MKTcet}(\mathcal{L}^{x_0 \ldots x_j}[a] + \mathcal{L}^{x_0 \ldots x_j}[b]);$
4. $\mathcal{L}^{x_0 \ldots x_j}[a + b] = \text{ MKTcomp}(A);$  \hspace{1cm} if $a$ and $b$ are evaluated.
5. $\mathcal{L}^{x_0 \ldots x_j}[a] = \text{ MKTcomp}(A);$  \hspace{1cm} where $A \Rightarrow \mathcal{T}^{x_0 \ldots x_j}[a + b];$
6. $\mathcal{L}^{x_0 \ldots x_j}[a + \text{ if } a \text{ then } b \text{ else } c] = \text{ MKTcomp}(A);$  \hspace{1cm} if $a$ is evaluated.
7. $\mathcal{L}^{x_0 \ldots x_j}[a : b] = \mathcal{L}^{x_0 \ldots x_j}[a] \mathcal{L}^{x_0 \ldots x_j}[b] \text{ MKEv}(b);$  \hspace{1cm} if $a$ is evaluated.
Example of Compilation

Let us present an example of compilation involving lists. If we have the script:

map f x = if x != [] then [] else f(Hd x):map f (Tl x)

fib n = if n < 2 then 1 else fib(n-1) + fib(n-2)
twice f x = f (f x)

map (twice fib) (1:(2:[]))

it will be compiled as,

\[
\begin{align*}
\text{map (twice fib) (1:(2:[]))} & \\
\mathcal{L} \mapsto T^3[\text{map (twice fib) (1:(2:[]))}] & \\
\end{align*}
\]

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \) if \( f \) is a special function.

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \)

Example of Compilation

Let us present an example of compilation involving lists. If we have the script:

\[
\begin{align*}
\text{map f x} & = \text{if } x = \square \text{ then } \square \text{ else } f(\text{Hd } x) : \text{map } f (\text{Tl } x) \\
\text{fib n} & = \text{if } n < 2 \text{ then } 1 \text{ else } \text{fib(n-1)} + \text{fib(n-2)} \\
\text{twice f x} & = f (f x) \\
\text{map (twice fib) (1:(2:[]))} & \\
\end{align*}
\]

it will be compiled as,

\[
\begin{align*}
\text{map (twice fib) (1:(2:[]))} & \\
\mathcal{L} \mapsto T^3[\text{map (twice fib) (1:(2:[]))}] & \\
\end{align*}
\]

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \) if \( f \) is a special function.

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \)

Example of Compilation

Let us present an example of compilation involving lists. If we have the script:

\[
\begin{align*}
\text{map f x} & = \text{if } x = \square \text{ then } \square \text{ else } f(\text{Hd } x) : \text{map } f (\text{Tl } x) \\
\text{fib n} & = \text{if } n < 2 \text{ then } 1 \text{ else } \text{fib(n-1)} + \text{fib(n-2)} \\
\text{twice f x} & = f (f x) \\
\text{map (twice fib) (1:(2:[]))} & \\
\end{align*}
\]

it will be compiled as,

\[
\begin{align*}
\text{map (twice fib) (1:(2:[]))} & \\
\mathcal{L} \mapsto T^3[\text{map (twice fib) (1:(2:[]))}] & \\
\end{align*}
\]

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \) if \( f \) is a special function.

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \)

Example of Compilation

Let us present an example of compilation involving lists. If we have the script:

\[
\begin{align*}
\text{map f x} & = \text{if } x = \square \text{ then } \square \text{ else } f(\text{Hd } x) : \text{map } f (\text{Tl } x) \\
\text{fib n} & = \text{if } n < 2 \text{ then } 1 \text{ else } \text{fib(n-1)} + \text{fib(n-2)} \\
\text{twice f x} & = f (f x) \\
\text{map (twice fib) (1:(2:[]))} & \\
\end{align*}
\]

it will be compiled as,

\[
\begin{align*}
\text{map (twice fib) (1:(2:[]))} & \\
\mathcal{L} \mapsto T^3[\text{map (twice fib) (1:(2:[]))}] & \\
\end{align*}
\]

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \) if \( f \) is a special function.

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \)

Example of Compilation

Let us present an example of compilation involving lists. If we have the script:

\[
\begin{align*}
\text{map f x} & = \text{if } x = \square \text{ then } \square \text{ else } f(\text{Hd } x) : \text{map } f (\text{Tl } x) \\
\text{fib n} & = \text{if } n < 2 \text{ then } 1 \text{ else } \text{fib(n-1)} + \text{fib(n-2)} \\
\text{twice f x} & = f (f x) \\
\text{map (twice fib) (1:(2:[]))} & \\
\end{align*}
\]

it will be compiled as,

\[
\begin{align*}
\text{map (twice fib) (1:(2:[]))} & \\
\mathcal{L} \mapsto T^3[\text{map (twice fib) (1:(2:[]))}] & \\
\end{align*}
\]

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \) if \( f \) is a special function.

where \( \Lambda \mapsto T^{x_0 \ldots x_n} [a : b] \)
\[ T_0 \Rightarrow \text{Eval}(0, 1); \text{MKTvw}; \text{Impl}_2; T_1 : T^{f,x}[f[(\text{Hd} \ x)]T^{f,x}[\text{map } f (T_1 \ x)]]; \text{MKcons}; l_2 : \]

where,

\[ C \Rightarrow T^{f,x}[\text{map } f (T_1 \ x)] \]

\[ \Rightarrow \text{MKEccell}(2); T^{f,x}[f] G^{f,x}[T_1 \ x] \text{ Pushfun(map); Popenv; } \]

\[ \Rightarrow \text{MKKvar}(2); \text{MKEvar}(1, 1); T^{f,x}[T_1 \ x] \text{ Pushfun(map); Popenv; } \]

\[ \Rightarrow \text{MKKvar}(2); \text{MKEvar}(1, 1); \text{Etl}(0, 0); \text{Pushfun(map); Popenv; } \]

### New State Transition Laws

ICMC with lists also makes use of the following state transition laws:

28. \((\text{printc}, d, T, H[d = a \ b], O, E) \Rightarrow \{(\text{eval.print.eval.print.c}, a, b, T, H[d = a:b], O, E)\} \]

29. \((\text{printc}, d, T, H[d = []], O, E) \Rightarrow \{(c, T, H[d = []], O, E)\} \]

30. \((\text{Hd}, c, d, T, H[d = a \ b], O, E) \Rightarrow \{(c, a, T, H[d = a:b], O, E)\} \]

31. \((\text{Tl}, c, d, T, H[d = a \ b], O, E) \Rightarrow \{(c, b, T, H[d = a:b], O, E)\} \]

32. \((\text{MKKcons}, c, d_1, d_2, T, H[d_2 = \text{d}_1, H[d_1 = a], O, E) \Rightarrow \{(c, d, T, H[d_1 = \text{d}_1, O, E)\} \]

33. \((\text{MKTvw}, c, T, H[d = []], O, E) \Rightarrow \{(c, d, T, H[d = []], O, E)\} \]

34. \((\text{MKEccell}(1), c, T, H[\text{c}_0 = \ldots \text{w}_i], O, E) \Rightarrow \{(c, T, H[d = \text{c}_0 = \ldots \text{w}_i], O, E)\} \]

35. \((\text{MKKcons}(1), c, d_1, d_2, T, H[d_1 = \text{d}_1, O, \text{c}_1, E) \Rightarrow \{(c, d_1, d_2, H[d_1 = \text{d}_1, \text{c}_1 = \ldots \text{w}_i], O, \text{c}_1, E)\} \]

36. \((\text{MKEccell}(1), c, T, H[\text{c}_0 = \ldots \text{w}_i], O, \text{c}_1, \text{c}_0, E) \Rightarrow \{(\text{c.d.T}, H[d = \text{w}_i], \text{c}_0 = \ldots \text{w}_i], O, \text{c}_1, \text{c}_0, E)\} \]

37. \((\text{MKTcomp}(A), c, T, H[\text{c}_0 = \ldots \text{w}_i], O, \text{c}_1, \text{c}_0, E) \Rightarrow \{(\text{c.d.T}, H[d = \text{A}. \text{c}_0], O, \text{c}_1, \text{c}_0, E)\} \]

38. \((\text{End}(i,c), T, H[\text{c}_1 = \ldots (a \ b) \ldots], O, \text{c}_1, E) \Rightarrow \{(\text{c.a.T}, H[\text{c}_1 = \ldots], O, \text{c}_1, E)\} \]

39. \((\text{Etl}(i,c), T, H[\text{c}_1 = \ldots (a \ b) \ldots], O, \text{c}_1, E) \Rightarrow \{(\text{c.b.T}, H[\text{c}_1 = \ldots], O, \text{c}_1, E)\} \]

40. \((\text{ETh}(i,c), T, H[\text{c}_1 = \ldots (a \ b) \ldots], O, \text{c}_1, \text{c}_0, E) \Rightarrow \{(\text{c.a.T}, H[\text{c}_0 = \ldots], O, \text{c}_1, \text{c}_0, E)\} \]

41. \((\text{Etl}(i,c), T, H[\text{c}_1 = \ldots (a \ b) \ldots], O, \text{c}_1, \text{c}_0, E) \Rightarrow \{(\text{c.b.T}, H[\text{c}_0 = \ldots], O, \text{c}_1, \text{c}_0, E)\} \]

42. \((\text{End}(i), c, T, H[\text{c}_1 = \ldots \text{w}_i], O, \text{c}_1, \text{c}_0, E) \Rightarrow \{(\text{c.T}, H[\text{c}_1 = \ldots \text{w}_i], O, \text{c}_1, \text{c}_0, E)\} \]

43. \((\text{Etl}(i), c, T, H[\text{c}_1 = \ldots \text{w}_i], O, \text{c}_1, \text{c}_0, E) \Rightarrow \{(\text{c.T}, H[\text{c}_1 = \ldots \text{w}_i], O, \text{c}_1, \text{c}_0, E)\} \]

44. \((\text{Eval}(n, l), l \times x, T, H[d_0 = \ldots \text{c}_0 = \ldots \text{d}_n \ldots], c_0, E) \Rightarrow \{(\text{c.T}, H[d_0 = \ldots \text{c}_0 = \ldots \text{d}_n \ldots]. O, \text{c}_1, \text{c}_0, E)\} \]

45. \((\text{Jmp} 1 \ldots \text{ic}, T, H, O, E) \Rightarrow \{(\text{c.T}, H, O, E)\} \]

46. \((\text{Jfalse} \text{False}, 1 \ldots \text{ic}, T, H, O, E) \Rightarrow \{(\text{c.T}, H, O, E)\} \]

47. \((\text{Jfalse} \text{True}, 1 \ldots \text{ic}, T, H, O, E) \Rightarrow \{(\text{c.T}, H, O, E)\} \]

### Example of Evaluation

As we saw in the examples of compilation above, the program:

\[
\text{map } f x = \text{if } x = \square \text{ then } \square \text{ else } f(\text{Hd} \ x) : \text{map } f (T_1 x) \\
\text{fib } n = \text{if } n \neq 2 \text{ then } 1 \text{ else fib}(n-1) + \text{fib}(n-2) \\
\text{twice } f x = f (f x) \\
\text{map } (\text{twice fib})(1:(2:\square))?
\]
compiled as,

\[
\text{map } \text{twice fib} \ (1 : (2 : []) ) \mapsto \text{MK Ecell}(2) ; \text{MKE comp}(B, 1) ; \text{MKT cte}(1) ; \text{MKT cte}(2) ; \text{MKT lv} ;
\]

\[
\text{MK cons} ; \text{MK cons} ; \text{MK cons} ; \text{MK cons} ; \text{Push fun}(\text{map}) ; \text{Pop env} ; \text{print}(1) ;
\]

\[
B \mapsto \text{MK Tpc}(A) ; \text{MK Env}(2) ; \text{Push fun}(\text{twice}) ; \text{Pop env} ;
\]

\[
\text{map} \mapsto \text{Enull}(0, 1) ; \text{MKT lv} ; \text{Jmpl}(2) : 1 = \text{MK Tvar}(0) ; \text{Hd} ; \text{Eva l env}(1) ; \text{MKT comp}(C) ; \text{MK cons} ; l_2 ;
\]

\[
C \mapsto \text{MK Ecell}(2) ; \text{MKE var}(1, 1) ; \text{E t a l}(0, 0) ; \text{Push fun}(\text{map}) ; \text{Pop env} ;
\]

\[
A \mapsto \text{eval}(0) ; \text{MT K t k}(0, \text{fib}(* (\text{stop T})) \mapsto \text{rem value}) ;
\]

\[
twice \mapsto \text{MK T comp}(A') ; \text{eval env}(1) ;
\]

\[
A' \mapsto \text{MK T var}(0) ; \text{eval env}(1) ;
\]

\[
\text{fib} \mapsto \text{if} (n < 2) \{ \text{return}(1) \} \text{else} \{ \text{return}(\text{fib}(n - 1) + \text{fib}(n - 2)) \};
\]

The initial state of the machine is:

\[
\langle \text{MK Ecell}(2) ; \text{MKE comp}(B, 1) ; \text{M KT cte}(1) ; \text{MKT cte}(2) ; \text{MKT lv} ; \ldots, T, H, O, E \rangle
\]

executing this code using the state transition laws above we have,

\[
\langle \text{Push fun}(\text{map}) ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_1 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]]
\]

\[
[d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{Enull}(0, 1) ; \text{M KT lv} ; \text{J mpl}(2) : 1 = \text{MK Tvar}(0) ; \text{H d} ; \text{E va l env}(1) ; \text{MK T comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{MK T var}(0) ; \text{H d} ; \text{E va l env}(1) ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{H d} ; \text{E va l env}(1) ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{E va l env}(1) ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{E va l}(1) ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{E va l}(1) ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{M KTpc}(A) ; \text{MK Env}(2) ; \text{Push fun}(\text{twice}) ; \text{Pop env} ; \text{Pop env} ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{MK Env}(2) ; \text{Push fun}(\text{twice}) ; \text{Pop env} ; \text{Pop env} ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[d_4 = A][d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][c_1 = (B, c_0)][c_0 = c_1, O, c_0, E]
\]

\[
\langle \text{Push fun}(\text{twice}) ; \text{Pop env} ; \text{Pop env} ; \text{M KT comp}(C) ; \text{MK cons} ; l_2 ; \text{Pop env} ; \text{print}(1) ; d_4, T, H[e_2 = d_0 : d_2][e_2 = A][d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [] [ ]][d_1 = 2][d_0 = 1][e_1 = (B, c_0)][c_0 = e_1, O, c_0, E]
\]

Now the code for \texttt{twice} is called taking as arguments \texttt{fib} and 1, which are referenced by the frame on the top of the environment stack. The reduction sequence above gives an idea of how \texttt{GC}C\texttt{M}C evaluates lists.
**Optimisations**

A number of code optimisations should be introduced to CMC in order to obtain a better performance. In this section we present the most important of them.

**Sharing**

Sharing of computation can bring substantial improvement to the performance of the machine. There is a number of ways sharing can be incorporated to CMC. Although the authors are still experimenting to know the best possible way, the sharing mechanism implemented at the moment is similar to the one in CMCM [6, 11], which is inspired in the frame update mechanism of TFM [2]. Now, the user provides annotations ($U$ combinator) to specify variables one wants to share.

The $U$ combinator performs the following state transition:

$$ (U(i).c.d.T, H[c_0 = \ldots c_i], O, c_0.E) \Rightarrow (c, d.T, H[c_0 = \ldots c_i], O, c_0.E) $$

As Categorical Multi-Combinators do not allow for partial applications to be reduced we think of to the one in CMCM [6, 11]. Now, the

Sharing of computation can bring substantial improvement to the performance of the machine. There is a number of ways sharing can be incorporated to CMC. Although the authors are still experimenting to know the best possible way, the sharing mechanism implemented at the moment is similar to the one in CMCM [6, 11], which is inspired in the frame update mechanism of TFM [2]. Now, the user provides annotations ($U$ combinator) to specify variables one wants to share.

The $U$ combinator performs the following state transition:

$$ (U(i).c.d.T, H[c_0 = \ldots c_i], O, c_0.E) \Rightarrow (c, d.T, H[c_0 = \ldots c_i], O, c_0.E) $$

As Categorical Multi-Combinators do not allow for partial applications to be reduced we think of using Partial Categorical Multi-Combinators [5] to deal with sharing of partial applications.

**Tail Recursion**

Functions over lists recursively defined as

$$ f_n x_0 \ldots x_n = \text{if } a \text{ then } b \text{ else } z : (f_n y_0 \ldots y_n) $$

such as `map`, are of widespread use in functional programs. The compilation schemes we have generate an environment every time we make a recursive call and discard the environment used for the previous call. To increase the performance of CMC we avoid garbage generation by compiling tail recursive functions as

$$ [f_n x_0 \ldots x_n = \text{if } x_i = [ ] \text{ then } b \text{ else } z : (f_n y_0 \ldots y_n)] $$

by the following entries in the script:

$$ f_n \mapsto \text{Enull}(i, l_1); T^{x_0 \ldots x_n}[b]; \text{Jmp} l_2; \text{JKTcomp}(A); \text{MKenv}(n + 1); \text{MKcons}; l_2 : A \mapsto G^{x_0 \ldots x_n}[y_n \ldots G^{x_0 \ldots x_n}[y_0]0 \text{ Swap}; \text{Pushfun}(f_n); \text{Popenv}; $$

$$ f_n \mapsto \text{Enull}(i, l_1); T^{x_0 \ldots x_n}[b]; \text{Jmp} l_2; l_1 : T^{x_0 \ldots x_n}[z] \text{ MKTcomp}(A); \text{MKcons}; l_2 ;$$

The state transition law for `Swap` is:

$$ (\text{Swap}.c, T, H, O, c_1, c_0.E) \Rightarrow (c, A.T, H, O, c_0, c_1, E) $$

**Recursive Functions**

Recursion is fundamental for functional programming languages. Many functions are not special thus can not benefit from the very efficient handling of recursion made by the C compiler, which takes advantage of the fast context switching mechanism of RISC architectures. Better performance can be obtained if we introduce a stack to handle recursion. Thus we translate functions defined as

$$ f_n x_0 \ldots x_n = \text{if } a < b \text{ then } c \text{ else } f_n y_0 \ldots y_n $$

by using the following scheme:

$$ f_n \mapsto \text{pushR}(0); $$

$$ \text{LP1 : } \text{Jfalse}(T^{x_0 \ldots x_n}[s]; t1 = (*\text{topT}---) \text{ rem.value}; $$

$$ T^{x_0 \ldots x_n}[b]; t2 = (*\text{topT}---) \text{ rem.value}; t1 > t2; l_1); T^{x_0 \ldots x_n}[c]; $$

$$ \text{Jmp} l_2; $$

$$ l_1 : \text{MKcell}(n + 1); $$

$$ G^{x_0 \ldots x_n}[y_n]n $$

$$ ; $$

$$ G^{x_0 \ldots x_n}[y_0]0 $$

$$ \text{pushR}(1); \text{Jmp}(\text{LP1}); $$

$$ \text{LP2 : } \text{Popenv}; $$

$$ l_2 : \text{if}(*\text{topR}---) == 1) \text{Jmp}(\text{LP2}); $$
Avoiding Indirections

Functions which take only one parameter are frequent. Because we adopted a fully-boxed representation the cell which represents the environment of a function to one argument works as an indirection cell. One can avoid the generation of this indirection cell by making the environment stack point directly to its argument. New operators are needed for this optimisation. For a matter of simplicity we will call them as before suffixed by 0. For instance, instruction Ehd(i) becomes Ehd0.

Monomorphic Print

Instead of having a general (polymorphic) printing routine, which at run-time tests the data produced to output it, we use information provided by the type-checker to choose statically which printing function is suitable for printing the output.

Performance

In this section we present the performance figures obtained for the benchmark programs below running on a SUN Sparcstation II under UNIX.

- **Fib 30**: the Fibonacci number of 30
- **Rev**: reverse reverse reverse of a list of 300 numbers.
- **Sieve**: generates a list of prime numbers smaller than 2000 by using Erathosthenes’ sieve.
- **Insord**: sorting by insertion of a list of 600 random numbers.
- **Simlog**: takes a list of 4000 random numbers and produces 4000 boolean values.
- **Map**: maps (twice twice twice successor) on a list of 2000 integers.
- **Tak**: Takeiushi function of 30 25 15.

<table>
<thead>
<tr>
<th></th>
<th>Fib 30</th>
<th>Rev</th>
<th>Sieve</th>
<th>Insord</th>
<th>Simlog</th>
<th>Map</th>
<th>Tak</th>
</tr>
</thead>
<tbody>
<tr>
<td>GM-C</td>
<td>10.6</td>
<td>—</td>
<td>2.8</td>
<td>5.0</td>
<td>1.6</td>
<td>2.2</td>
<td>2.382</td>
</tr>
<tr>
<td>ΓC M C</td>
<td>1.5</td>
<td>2.5</td>
<td>1.4</td>
<td>2.9</td>
<td>0.2</td>
<td>0.5</td>
<td>2.8</td>
</tr>
<tr>
<td>LML</td>
<td>8.9</td>
<td>4.0</td>
<td>1.6</td>
<td>2.5</td>
<td>0.6</td>
<td>0.4</td>
<td>42.8</td>
</tr>
</tbody>
</table>

 ΓM-C corresponds to the last version of GM-C [8] done by Musicante and Lins. ΓC M C refers to our best implementation of ΓC M C. LML presents the performance of the Chalmers Lazy ML compiler version 0.99.3.

As we can observe from the table above the performance of ΓC M C is far better than GM-C, for all our benchmark programs. ΓC M C presented a performance close to LML in the benchmark programs which made intensive use of higher-order-functions and lazy evaluation. In the case of the use of strict functions under recursion ΓC M C performed far better than LML.

Further Work & Conclusions

The authors are currently working on a front-end for ΓC M C and also analysing a number of optimisations to the back-end. We believe the benchmark figures we presented are representative of the performance of ΓC M C and that they scale-up for larger programs.

In our opinion, ΓC M C has already shown that transferring the flow of execution from a higher-level abstract machine to C, by using procedure calls, brings not only portability but also efficiency on RISC architectures. The performance of ΓC M C is in the worst case as good as the Chalmers LML compiler. In the best case, ΓC M C can run several times faster than LML.
Acknowledgements

We express gratitude for several discussions with Ricardo Massa and Genésio Neto. Research reported herein has been sponsored jointly by The British Council, CNPq (Brazil) grants 40.9110/88-4, 46.0782/89.4, and 80.4520/88-7, and CAPES (Brazil) grant 2487/91-08.

References


