ΓCMC: 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 dependant. In this paper we present ΓCMC, a new abstract machine, in which we transfer the control of the execution flow to C, as much as possible. ΓCMC 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, Götheburg, by Johnsson and his collegues [10], generated code in VAX-780 Assembly language, which made implementation extremely hard and machine dependant. 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 ΓCMC, a new abstract machine, in which we transfer the execution flow control to C, as much as possible. The key idea behind ΓCMC 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.

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

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

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

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 combinator c there is associated code c_r, we supress the environment ρ 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 $R^{x_0 \dots x_j}$ 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) $R^{[\,]} \underbrace{\lambda x_k \dots \lambda x_l}_{m} .a = \langle L^{m-1}(R^{x_k \dots x_l} a), () \rangle$

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

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

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

Combinator names are treated as constants.

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:

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

which by application of the compilation rules above translates to:

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

The expression to be evaluated is translated as

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

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) $\langle x_0 x_1 x_2 \ldots x_n, y\rangle \Rightarrow \langle x_0, y\rangle \ldots \langle x_n, y\rangle$

(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 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 ΓCMC

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

If one observes the rewriting rules for Categorical Multi-Combinators above we see that rule (M*.3) is equivalent to λ-Calculus β-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. $\langle\langle n, e_i\rangle.c, H[e_i = (x_m, \cdots, x_0)]\rangle \Rightarrow \langle x_n.c, H[e_i = (x_m, \cdots, x_0)]\rangle$
2. $\langle\langle x_0 \ldots x_n, e_i\rangle.c, H[e_i = \cdots]\rangle \Rightarrow \langle\langle x_0, e_i\rangle \ldots \langle x_n, e_i\rangle.c, H[e_i = \cdots]\rangle$
3. $\langle\langle L^n(y), e_i\rangle x_0 \cdots x_n.c, H\rangle \Rightarrow \langle\langle y, e_j\rangle.c, H[e_j = (x_0, \cdots, x_n)]\rangle$
4. $\langle\langle l, e_i\rangle.c, H\rangle \Rightarrow \langle\langle l_r, e_i\rangle.c, H\rangle$

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 β-reduction. We call the environment stack **E**.

1. $\langle n.c, H[e = (\langle x_m, e_m\rangle, \cdots, \langle x_0, e_0\rangle)], e.E\rangle \Rightarrow \langle x_n.c, H[e_i = (\langle x_m, e_m\rangle, \cdots, \langle x_0, e_0\rangle)], e_n.E\rangle$
2. $\langle\langle L^n(y), e_i\rangle x_0 \cdots x_n.c, H\rangle \Rightarrow \langle y.c, H[e_j = (\langle x_0, e_i\rangle, \cdots, \langle x_n, e_i\rangle)], e_j.E\rangle$
3. $\langle l.c, H, E\rangle \Rightarrow \langle\langle l_r, e_i\rangle.c, H, E\rangle$

Example of Evaluation

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

$$\begin{array}{lcl} S & \mapsto & L^2(2\ 0\ (1\ 0)) \\ K & \mapsto & L^1(1) \\ I & \mapsto & L^0(0) \end{array}$$

is evaluated as,

$$\begin{array}{rcl} \langle SKKI, H, E\rangle & \stackrel{3}{\Rightarrow} & \langle L^2(2\ 0\ (1\ 0))\ K\ K\ I, H, E\rangle \\ & \stackrel{2}{\Rightarrow} & \langle 2\ 0\ (1\ 0), H[e_1 = (K\ K\ I)], e_1.E\rangle \\ & \stackrel{1}{\Rightarrow} & \langle K\ 0\ (1\ 0), H[e_1 = (K\ K\ I)], e_1.E\rangle \\ & \stackrel{3}{\Rightarrow} & \langle L^1(1)\ 0\ (1\ 0), H[e_1 = (K\ K\ I)], e_1.E\rangle \\ & \stackrel{2}{\Rightarrow} & \langle 1, H[e_2 = (\langle 0, e_1\rangle, \langle (1\ 0), e_1\rangle)][e_1 = (K\ K\ I)], e_2.e_1.E\rangle \\ & \stackrel{2}{\Rightarrow} & \langle 0, H[e_2 = (\langle 0, e_1\rangle, \langle (1\ 0), e_1\rangle)][e_1 = (K\ K\ I)], e_1.e_2.e_1.E\rangle \\ & \stackrel{1}{\Rightarrow} & \langle I, H[e_2 = (\langle 0, e_1\rangle, \langle (1\ 0), e_1\rangle)][e_1 = (K\ K\ I)], e_1.e_2.e_1.E\rangle \end{array}$$

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* ΓCMC . All ΓCMC does is to prepare the arguments for them and receive the result. We introduce a new state transition law for special functions:

4. $\langle f^n\ x_0 \ldots x_n.c, H, e_i.E\rangle \Rightarrow \langle f_r.c, H, e_i.E\rangle$
   where $f_r = f^n(x'_0, \ldots, x'_n)$ and x'_i is the weak head normal form of x_i.

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:

$$\begin{array}{lcl} S' & \mapsto & L^2(2\ (f_1 1\ 0)) \\ f_1 & \mapsto & (1' + 0') \end{array}$$

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 $\mathsf{MKTvar(n)}$ 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:

$$\begin{array}{l} f_1 x_0^1 \ldots x_n^1 = body{-}of{-}f_1 \\ f_2 x_0^2 \ldots x_m^2 = body{-}of{-}f_2 \\ \vdots \\ f_z x_0^z \ldots x_y^z = body{-}of{-}f_z \\ expression? \end{array}$$

The expression to be evaluated is compiled by scheme $\mathcal{E}$ called as:

$$\mathcal{E}[expression]$$

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:

$$f_i x_0^i \ldots x_n^i = body{-}of{-}f_i \qquad\qquad f_i \mapsto \mathcal{S}[body{-}of{-}f_i]$$

Ordinary functions are compiled as:

$$f_j x_0^j \ldots x_l^j = body{-}{-}\ of{-}f_j \qquad\qquad f_j \mapsto \mathcal{T}^{x_0^j \ldots x_l^j}[body{-}of{-}f_j]$$

Scheme $\mathcal{E}$

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

1. $\mathcal{E}[k] = \mathsf{printf(k)};$ if k is a constant
2. $\mathcal{E}[a+b] = \mathsf{t1} = \mathcal{S}'[a]; \mathsf{t2} = \mathcal{S}'[b]; \mathsf{printf(t1+t2)};$
3. $\mathcal{E}[\mathit{if}\ a > b\ \mathit{then}\ c\ \mathit{else}\ d] = \mathsf{If\_true(A, C, D)};$
   where $\mathsf{A} \mapsto \mathcal{T}^{[]}[a]; \mathsf{t1} = (*(\mathsf{topT}--)) \rightarrow \mathsf{rem.value}; \mathcal{T}^{[]}[b]; \mathsf{t2} = (*(\mathsf{topT}--)) \rightarrow \mathsf{rem.value}; \mathsf{t1} > \mathsf{t2};$
   $\mathsf{C} \mapsto \mathcal{E}[c]\ \mathsf{D} \mapsto \mathcal{E}[d];$
4. $\mathcal{E}[f_i x_0 \ldots x_n] = \mathsf{printf}(\mathsf{f_i}(\mathcal{S}'[x_0], \ldots, \mathcal{S}'[x_n]));$ if f_i is a special function.
5. $\mathcal{E}[f_i \ldots] = \mathcal{T}^{[]}[f_i \ldots]\ \mathsf{print()};$

Scheme $\mathcal{S}$

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

1. $\mathcal{S}[k] = \mathsf{return(k);}$ if k is a constant
2. $\mathcal{S}[x] = \mathsf{return(x);}$ if x is a variable
3. $\mathcal{S}[a+b] = \mathsf{return}(\mathcal{S}'[a] + \mathcal{S}'[b]);$
4. $\mathcal{S}[\mathit{if}\ (a > b)\ \mathit{then}\ c\ \mathit{else}\ d] = \mathsf{if}\ (\mathcal{S}'[a] > \mathcal{S}'[b])\ \{\mathcal{S}[c]\}; \mathsf{else}\{\mathcal{S}[d]\};$
5. $\mathcal{S}[f_i x_0 \ldots x_j] = \mathsf{return}(\mathsf{f_i}(\mathcal{S}'[x_0], \ldots, \mathcal{S}'[x_j]));$ if f_i is a special function
6. $\mathcal{S}'[f_i x_0 \ldots x_j] = \mathcal{T}^{[]}(f_i x_0 \ldots x_j)\ \mathsf{return}(((*(\mathsf{topT}--)) \rightarrow \mathsf{rem.value}));$

Scheme $\mathcal{S}'$

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

1. $\mathcal{S}'[k] = \mathsf{k}$ if k is a constant
2. $\mathcal{S}'[x] = \mathsf{x}$ if x is a variable
3. $\mathcal{S}'[a+b] = (\mathcal{S}'[a] + \mathcal{S}'[b])$
4. $\mathcal{S}'[\mathit{if}\ (a > b)\ \mathit{then}\ c\ \mathit{else}\ d] = \mathsf{if}\ (\mathcal{S}'[a] > \mathcal{S}'[b])\ \{(\mathcal{S}[c])\}; \mathsf{else}\{(\mathcal{S}[d])\};$
5. $\mathcal{S}'[f_i x_0 \ldots x_i] = \mathsf{f_i}(\mathcal{S}'[x_0], \ldots, \mathcal{S}'[x_i])$ if f_i is a special function.
6. $\mathcal{S}'[f_i x_0 \ldots x_i] = \mathcal{T}^{[]}[f_i x_0 \ldots x_i]\ \mathsf{return}((*(\mathsf{topT}--)) \rightarrow \mathsf{rem.value});$

Example of Compilation

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

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

it will be compiled as,

$$\begin{aligned} \mathcal{E}[\texttt{fib 20}] &\overset{\mathcal{E},4}{\Rightarrow} \mathsf{printf(fib}(\mathcal{S}'[20])); \\ &\overset{\mathcal{S}',1}{\Rightarrow} \mathsf{printf(fib(20));} \end{aligned}$$

Now we compile

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

by using scheme $\mathcal{S}$.

$$\begin{aligned}
\mathsf{fib} &\mapsto \mathcal{S}[\mathit{if}\ n < 2\ \mathit{then}\ 1\ \mathit{else}\ \mathit{fib}(n-1) + \mathit{fib}(n-2)] \\
&\overset{\mathcal{S},4}{\Rightarrow} \mathsf{if}\ (\mathcal{S}'[n] < \mathcal{S}'[2])\ \{\mathcal{S}[1])\}; \mathsf{else}\{\mathcal{S}[\mathit{fib}(n-1) + \mathit{fib}(n-2)]\}; \\
&\overset{\mathcal{S}',2}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < \mathcal{S}'[2])\ \{\mathcal{S}[1]\}; \mathsf{else}\{\mathcal{S}[\mathit{fib}(n-1) + \mathit{fib}(n-2)]\}; \\
&\overset{\mathcal{S}',1}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathcal{S}[1\}; \mathsf{else}\{\mathcal{S}[\mathit{fib}(n-1) + \mathit{fib}(n-2)]\}; \\
&\overset{\mathcal{S},1}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathcal{S}[\mathit{fib}(n-1) + \mathit{fib}(n-2)]\}; \\
&\overset{\mathcal{S},3}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathcal{S}'[\mathit{fib}(n-1)] + \mathcal{S}'[\mathit{fib}(n-2)])\}; \\
&\overset{\mathcal{S}',5}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathcal{S}'[(n-1)]) + \mathcal{S}'[\mathit{fib}(n-2)])\}; \\
&\overset{\mathcal{S}',5}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathcal{S}'[(n-1)]) + \mathsf{fib}(\mathcal{S}'[(n-2)]))\}; \\
&\overset{\mathcal{S}',3}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathcal{S}'[n] - \mathcal{S}'[1])]) + \mathsf{fib}(\mathcal{S}'[(n-2)]))\}; \\
&\overset{\mathcal{S}',2}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathsf{n}{-}\mathcal{S}'[1])]) + \mathsf{fib}(\mathcal{S}'[(n-2)]))\}; \\
&\overset{\mathcal{S}',1}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathsf{n}-1) + \mathsf{fib}(\mathcal{S}'[(n-2)]))\}; \\
&\overset{\mathcal{S}',3}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathsf{n}-1) + \mathsf{fib}(\mathcal{S}'[n] - \mathcal{S}'[2]))\}; \\
&\overset{\mathcal{S}',2}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathsf{n}-1) + \mathsf{fib}(\mathsf{n}{-}\mathcal{S}'[2]))\}; \\
&\overset{\mathcal{S}',1}{\Rightarrow} \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathsf{n}-1) + \mathsf{fib}(\mathsf{n}-2))\};
\end{aligned}$$

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 $\mathcal{T}$

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. $\mathcal{T}^{y_0\cdots y_j}[f_n x_0 \ldots x_n z_1 \ldots z_m] = \mathcal{T}'^{x_0\cdots x_n}[z_m] \ldots \mathcal{T}'^{x_0\cdots x_n}[z_1]\mathcal{T}^{x_0\cdots x_n}[f_n x_0 \ldots x_n]$
2. $\mathcal{T}^{y_0\cdots y_j}[f_n x_0 \ldots x_n] = \mathsf{MKTcte(f_n(}\mathcal{Z}^{y_0\cdots y_j}x_0, \ldots, \mathcal{Z}^{y_0\cdots y_j}x_n));$ if f_n is a special function
3. $\mathcal{T}^{y_0\cdots y_j}[f_n x_0 \ldots x_n] = \mathsf{MKEcell}(n+1); \mathcal{G}^{x_0\cdots x_n}[x_0]n \ldots \mathcal{G}^{x_0\cdots x_n}[x_n]0$ $\mathsf{Pushfun(f_n); Popenv;}$
4. $\mathcal{T}^{y_0\cdots y_j}[f_n] = \mathsf{eval'(0); \ldots eval'(n); MKTk(n, f_n((*(topT) \rightarrow rem.value), \ldots, (*(topT - n) \rightarrow rem.value)));}$ if f_i is a special function.
5. $\mathcal{T}^{y_0\cdots y_j}[f_n] = \mathsf{MKenv(n+1); Pushfun(f_n); Popenv;}$
6. $\mathcal{T}^{y_0\cdots y_j}[k] = \mathsf{MKTcte(k);}$ if k is a constant
7. $\mathcal{T}^{y_0\cdots y_j}[y_i] = \mathsf{MKTvar(i);}$ if x_i is evaluated
8. $\mathcal{T}^{y_0\cdots y_j}[y_i] = \mathsf{eval\_env(i);}$
9. $\mathcal{T}^{y_0\cdots y_j}[\mathit{if}\ a > b\ \mathit{then}\ c\ \mathit{else}\ d] = \mathsf{If\_true(A, C, D);}$
   where $\mathsf{A} \mapsto \mathcal{T}^{y_0\cdots y_j}[a]\ \mathsf{t1 = (*(topT--)) \rightarrow rem.value;}\ \mathcal{T}^{y_0\cdots y_j}[b]\ \mathsf{t2 = (*(topT--)) \rightarrow rem.value; t1 > t2;}$
   $\mathsf{C} \mapsto \mathcal{T}^{y_0\cdots y_j}[c]$ $\quad \mathsf{D} \mapsto \mathcal{T}^{y_0\cdots y_j}[d]$
10. $\mathcal{T}^{y_0\cdots y_j}[a+b] = \mathsf{t1} = \mathcal{T}^{y_0\cdots y_j}[a]\ \mathsf{t2} = \mathcal{T}^{y_0\cdots y_j}[b]\ \mathsf{MKTcte(t1 + t2);}$
11. $\mathcal{T}^{y_0\cdots y_j}[x_0(x_1 \ldots x_m)] = \mathsf{MKTcomp(A);}\mathcal{T}^{y_0\cdots y_j}[x_0]$ where $\mathsf{A} \mapsto \mathcal{T}^{y_0\cdots y_j}[x_1 \ldots x_m]$
12. $\mathcal{T}^{y_0\cdots y_j}[x_0 \ldots x_m] = \mathcal{T}'^{y_0\cdots y_j}[x_1] \ldots \mathcal{T}'^{y_0\cdots y_j}[x_m]\ \mathcal{T}^{y_0\cdots y_j}[x_0]$

Scheme $\mathcal{G}$

This scheme generates code which when executed fills the fields of a cell in an evaluation environment.

1. $\mathcal{G}^{x_0\cdots x_j}[k]i = \mathsf{MKEcte(k, i);}$
2. $\mathcal{G}^{x_0\cdots x_j}[x_i]j = \mathsf{MKEvar(x_i, j);}$
3. $\mathcal{G}^{x_0\cdots x_j}[a+b]j = \mathsf{MKEcte(}\mathcal{Z}'^{x_0\cdots x_j}[a] + \mathcal{Z}'^{x_0\cdots x_j}[b], \mathsf{j});$ if a and b are evaluated.
4. $\mathcal{G}^{x_0\cdots x_j}[a+b]i = \mathsf{MKEcomp(A, i);}$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i\cdots x_j}[a+b];$
5. $\mathcal{G}^{x_0\cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c]j = \mathsf{IfE\_true(}\mathcal{Z}'^{x_0\cdots x_j}[a], \mathsf{B, C, j});$ if a is evaluated
   where $\mathsf{B} \mapsto \mathcal{T}^{x_0\cdots x_j}[b]$ $\ \mathsf{C} \mapsto \mathcal{T}^{x_0\cdots x_j}[c]$
6. $\mathcal{G}^{x_0\cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c]i = \mathsf{MKEcomp(A, i);}$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i\cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c];$
7. $\mathcal{G}^{x_0\cdots x_j}[a \ldots b]i = \mathsf{MKEcomp(A, i);}$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i\cdots x_j}[a \ldots b];$
8. $\mathcal{G}^{x_0\cdots x_j}[f_n]i = \mathsf{MKEpc(A, i);}$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i\cdots x_j}[f_n];$

Scheme $\mathcal{T}'$

This scheme produces code which when executed generates cells on the top of the T-stack.

1. $\mathcal{T}'^{x_0\cdots x_j}[k] = \mathsf{MKTcte(k);}$
2. $\mathcal{T}'^{x_0\cdots x_j}[x_i] = \mathsf{MKTvar(i);}$
3. $\mathcal{T}'^{x_0\cdots x_j}[a+b] = \mathsf{MKTcte(}\mathcal{Z}^{x_0\cdots x_j}[a] + \mathcal{Z}^{x_0\cdots x_j}[b]);$ if a and b are evaluated.
4. $\mathcal{T}'^{x_0\cdots x_j}[a+b] = \mathsf{MKTcomp(A);}$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i\cdots x_j}[a+b]$
5. $\mathcal{T}'^{x_0\cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c] = \mathsf{If\_true(}\mathcal{Z}^{x_0\cdots x_j}[a], \mathsf{B, C});$ if a is evaluated.
   where $\mathsf{B} \mapsto \mathcal{T}^{x_0\cdots x_j}[b], \mathsf{C} \mapsto \mathcal{T}^{x_0\cdots x_j}[c]$
6. $\mathcal{T}'^{x_0\cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c] = \mathsf{MKTcomp(A);}$ where $\mathsf{A} \mapsto \mathcal{T}^{x_0\cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c]$
7. $\mathcal{T}'^{x_0\cdots x_j}[a \ldots b] = \mathsf{MKTcomp}(A);$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i\cdots x_j}[a \ldots b]$
8. $\mathcal{T}'^{x_0\cdots x_j}[f_i] = \mathsf{MKTpc(A);}$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i\cdots x_j}[f_n]$

Scheme $\mathcal{Z}$

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

1. $\mathcal{Z}^{x_0 \cdots x_j}[k] = \mathsf{k};$
2. $\mathcal{Z}^{x_0 \cdots x_j}[x_i] = ((*\mathsf{topE} + (*\mathsf{topE}) \rightarrow \mathsf{tipo} - 1 - \mathsf{i}) \rightarrow \mathsf{rem.graph}) \rightarrow \mathsf{rem.value};$
   if x_i is already evaluated.
3. $\mathcal{Z}^{x_0 \cdots x_j}[x_i] = \mathsf{eval\_env(i); return(}*(\mathsf{topT}--) \rightarrow \mathsf{rem.value});$
4. $\mathcal{Z}^{x_0 \cdots x_j}[f_i a \ldots b] = \mathsf{f_i}(\mathcal{Z}^{x_0 \cdots x_j}[a], \ldots, \mathcal{Z}^{x_0 \cdots x_j}[b]);$ if f_i is a special function.
5. $\mathcal{Z}^{x_0 \cdots x_j}[a \ldots b] = \mathcal{T}^{x_0 \cdots x_j}[a \ldots b]\ \mathsf{return}((*(\mathsf{topT}--)) \rightarrow \mathsf{rem.value});$
6. $\mathcal{Z}^{x_0 \cdots x_j}[a + b] = \mathcal{Z}^{x_0 \cdots x_j}[a] + \mathcal{Z}^{x_0 \cdots x_j}[b];$

Scheme $\mathcal{Z}'$

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

1. $\mathcal{Z}'^{x_0 \cdots x_j}[k] = \mathsf{k};$
2. $\mathcal{Z}'^{x_0 \cdots x_j}[x_i] = (((*\mathsf{topE} - 1) + (*\mathsf{topE} - 1) \rightarrow \mathsf{tipo} - 1 - \mathsf{i}) \rightarrow \mathsf{rem.graph}) \rightarrow \mathsf{rem.value};$
   if x_i is already evaluated.
3. $\mathcal{Z}'^{x_0 \cdots x_j}[x_i] = \mathsf{eval\_env(i); return(}*(\mathsf{topT}--) \rightarrow \mathsf{rem.value});$
4. $\mathcal{Z}'^{x_0 \cdots x_j}[f_i a \ldots b] = \mathsf{f_i}(\mathcal{Z}'^{x_0 \cdots x_j}[a], \ldots, \mathcal{Z}'^{x_0 \cdots x_j}[b]);$ if f_i is a special function.
5. $\mathcal{Z}'^{x_0 \cdots x_j}[a \ldots b] = \mathcal{T}^{x_0 \cdots x_j}[a \ldots b]\ \mathsf{return}((*(\mathsf{topT}--)) \rightarrow \mathsf{rem.value});$
6. $\mathcal{Z}'^{x_0 \cdots x_j}[a + b] = \mathcal{Z}'^{x_0 \cdots x_j}[a] + \mathcal{Z}'^{x_0 \cdots x_j}[b];$

Example of Compilation

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

```
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,

$$
\begin{array}{lll}
\mathcal{E}[\texttt{twice fib 5}] & \overset{\mathcal{E}.5}{\Rightarrow} & \mathcal{T}^{[]}[\mathit{twice\ fib\ 5}]; \mathsf{print}(); \\
& \overset{\mathcal{T}.3}{\Rightarrow} & \mathsf{MKEcell}(2); \mathcal{G}^{[]}[fib]; \mathcal{G}^{[]}[5]0; \mathsf{Pushfun(twice)}; \mathsf{Popenv}; \mathsf{print}(); \\
& \overset{\mathcal{G}.8}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEpc}(\mathsf{A}, 1); \mathcal{G}^{[]}[5]0; \mathsf{Pushfun(twice)}; \mathsf{Popenv}; \mathsf{print}(); \\
& \overset{\mathcal{G}.1}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEpc}(\mathsf{A}, 1); \mathsf{MKEcte}(5, 0); \mathsf{Pushfun(twice)}; \mathsf{Popenv}; \mathsf{print}();
\end{array}
$$

where A is:

$$
\begin{array}{lll}
\mathsf{A} & \mapsto & \mathcal{T}^{f,x}[\mathit{fib}] \\
& \overset{\mathcal{T}.4}{\Rightarrow} & \mathsf{eval}'(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value}));
\end{array}
$$

Now we compile

```
twice f x = f (f x)
```

by using scheme $\mathcal{T}$ as,

$$
\begin{array}{lll}
\mathsf{twice} & \mapsto & \mathcal{T}^{f,x}[f\ (f\ x)] \\
& \overset{\mathcal{T}.11}{\Rightarrow} & \mathsf{MKTcomp}(\mathsf{A}'); \mathcal{T}^{f,x}[f] \\
& \overset{\mathcal{T}.8}{\Rightarrow} & \mathsf{MKTcomp}(\mathsf{A}'); \mathsf{eval\_env}(1);
\end{array}
$$

where A$'$ is:

$$
\begin{array}{lll}
\mathsf{A}' & \mapsto & \mathcal{T}^{f,x}[f\ x] \\
& \overset{\mathcal{T}.12}{\Rightarrow} & \mathcal{T}'^{f,x}[x]; \mathcal{T}^{f,x}[f]; \\
& \overset{\mathcal{T}'.2}{\Rightarrow} & \mathsf{MKTvar}(0); \mathcal{T}^{f,x}[f]; \\
& \overset{\mathcal{T}.8}{\Rightarrow} & \mathsf{MKTvar}(0); \mathsf{eval\_env}(1);
\end{array}
$$

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 $H[d = e_1 \ldots e_n]$ means that there is in H a n-component cell named d. The fields of d are filled with $e_1 \ldots e_n$, 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. $\langle$ print.c, d.T, H[d=k], O, E $\rangle \Rightarrow \langle$ c, T, H[d=k], k.O, E $\rangle$
2. $\langle$ eval.c, d.T, H[d=k], O, E $\rangle \Rightarrow \langle$ c, d.T, H[d=k], O, E $\rangle$
3. $\langle$ eval.c,d.T,H[d=(A,e)], O, E $\rangle \Rightarrow \langle$ A.Popenv.c, T, H[d=(A,e)], O, e.E $\rangle$
4. $\langle$ eval.c,d.T,H[d=fn], O, E $\rangle \Rightarrow \langle$ f_n.c, T, H[d= fn], O, E $\rangle$
5. $\langle$ eval'(i).c, $\ldots d_i \ldots$.T, H [d_i=k], O, E $\rangle \Rightarrow \langle$ c, $\ldots d_i \ldots$T, H [d=k] , O, E $\rangle$
6. $\langle$ eval'(i).c,$\ldots d_i \ldots$.T,H[d_i=(A,e)], O, E $\rangle \Rightarrow \langle$ A.Pop(i).c, $\ldots d_i \ldots$T, H [d=(A,e)], O, e.E $\rangle$
7. $\langle$ eval'(i).c,$\ldots d_i \ldots$.T,H[d_i = fn], O, E $\rangle \Rightarrow \langle$ c, $\ldots d_i \ldots$T, H [d= fn], O, E $\rangle$
8. $\langle$ eval_env(i).c, T, H[e=…fn …], O, e.E $\rangle \Rightarrow \langle$ f_n, T, H[e=…fn …], O, e.E $\rangle$
9. $\langle$ eval_env(i).c, T, H[e=…u_i …], O, e.E $\rangle \Rightarrow \langle$ eval.c, u_i.T, H[e=…u_i …], O, e.E $\rangle$
10. $\langle$ MKenv(n).c, $d_1 \ldots . d_n \ldots d_m$.T, H,O, E $\rangle \Rightarrow \langle$ c,$d_{n+1} \ldots d_m$.T, H [e=$d_1 \ldots d_n$], O, e.E $\rangle$
11. $\langle$ Popenv.c, T, H, O, e.E $\rangle \Rightarrow \langle$ c, T, H, O, E $\rangle$
12. $\langle$ Pop(i).c,$d_0 d_1 \ldots d_i \ldots$T, H, O, e.E $\rangle \Rightarrow \langle$ c,$d_1 \ldots d_0 \ldots$T, H, O, E $\rangle$
13. $\langle$ Pushfun(f_i).c, T, H, O, E $\rangle \Rightarrow \langle$ f_i.c, T, H, O, E $\rangle$
14. $\langle$ If_true(True,A,B).c, T, H, O, E $\rangle \Rightarrow \langle$ A.c, T, H, O, E $\rangle$
15. $\langle$ If_true(False,A,B).c, T, H, O, E $\rangle \Rightarrow \langle$ B.c, T, H, O, E $\rangle$
16. $\langle$ MKTvar(k).c, T, H[e=$e_0 \ldots e_m$], O, e.E $\rangle \Rightarrow \langle$ c, d.T, H[d = e_{m-k}][e=$e_0 \ldots e_m$], O, e.E $\rangle$
17. $\langle$ MKTcomp(P).c, T, H, O, e.E $\rangle \Rightarrow \langle$ c, d.T, H[d=(P,e)], O, e.E $\rangle$
18. $\langle$ MKTpc(A).c, T, H, O, E $\rangle \Rightarrow \langle$ c, d.T, H[d= A], O, E $\rangle$
19. $\langle$ MKTcte(k).c, T, H, O, E $\rangle \Rightarrow \langle$ c, d.T, H[d= k], O, E $\rangle$
20. $\langle$ MKTk(n,A).c, $d_0 \ldots d_n \ldots d_m$ .T, H [d_n =…], O, E $\rangle \Rightarrow \langle$ c, $d_n \ldots d_m$ .T, H [d_n=A] , O, E $\rangle$
21. $\langle$ MKEvar(j,i).c, T, H[$e_1 = \ldots u_i \ldots$] [$e_0 = \ldots a_j \ldots$], O, $e_1.e_0$.E $\rangle \Rightarrow$
    $\langle$ c, T, H[e_1=… a_j …] [$e_0 = \ldots a_j \ldots$], O, $e_1.e_0$.E $\rangle$
22. $\langle$ MKEcte(k,i).c, T, H[$e_1 = \ldots u_i \ldots$] O, e_1.E $\rangle \Rightarrow \langle$ c, T, H[e_1=…k…], O, e_1.E $\rangle$
23. $\langle$ MKEcell(n).c, T, H, O, E $\rangle \Rightarrow \langle$ c, d.T, H[e=$u_1 \ldots u_n$], O, E $\rangle$
24. $\langle$ MKEcomp(A,i).c, T, H[$e_1 = \ldots u_i \ldots$],O, $e_1.e_0$.E $\rangle \Rightarrow \langle$ c,T,H[e_1=…(A,e_0)…], O, $e_1.e_0$.E $\rangle$
25. $\langle$ MKEpc(A,i).c, T, H[e=…i …], O, e.E $\rangle \Rightarrow \langle$ c, d.T, H[e=…A …], O, e.E $\rangle$
26. $\langle$ IfE_true(True,B,C,i).c, T, H[$e_1 = \ldots u_i \ldots$],O, $e_1.e_0$.E $\rangle \Rightarrow \langle$ c, T, H[e_1=… (B,e_0) …], O, $e_1.e_0$.E $\rangle$
27. $\langle$ IfE_true(False,B,C,i).c, T, H[$e_1 = \ldots u_i \ldots$],O, $e_1.e_0$.E $\rangle \Rightarrow \langle$ c, T, H[e_1=… (C,e_0) …], O, $e_1.e_0$.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,

$$\begin{array}{rcl}
\texttt{twice fib 5} & \mapsto & \mathsf{MKEcell}(2); \mathsf{MKEpc}(\mathsf{A},1); \mathsf{MKEcte}(5,0); \mathsf{Pushfun}(\mathsf{twice}); \mathsf{Popenv}; \mathsf{print}(); \\
\mathsf{A} & \mapsto & \mathsf{eval}'(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \rightarrow \mathsf{rem.value})); \\
\mathsf{twice} & \mapsto & \mathsf{MKTcomp}(\mathsf{A}'); \mathsf{eval_env}(1); \\
\mathsf{A}' & \mapsto & \mathsf{MKTvar}(0); \mathsf{eval_env}(1); \\
\mathsf{fib} & \mapsto & \mathsf{if}\ (\mathsf{n} < 2)\ \{\mathsf{return}(1)\}; \mathsf{else}\{\mathsf{return}(\mathsf{fib}(\mathsf{n}-1) + \mathsf{fib}(\mathsf{n}-2))\};
\end{array}$$

The initial state of the machine is:

$$\langle\, \mathsf{MKEcell}(2);\ \mathsf{MKEpc}(\mathsf{A},1);\ \mathsf{MKEcte}(5,0);\ \mathsf{Pushfun}(\mathsf{twice});\ \mathsf{Popenv};\ \mathsf{print}();\ ,\mathrm{T,H,O,E}\,\rangle$$

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

$$\begin{array}{ll}
\overset{23}{\Rightarrow} & \langle \mathsf{MKEpc}(\mathsf{A},1); \mathsf{MKEcte}(5,0); \mathsf{Pushfun}(\mathsf{twice}); \mathsf{Popenv}; \mathsf{print}(); , T, H[e_0 = 1\ 0], O, e_0.E\rangle \\
\overset{25}{\Rightarrow} & \langle \mathsf{MKEcte}(5,0); \mathsf{Pushfun}(\mathsf{twice}); \mathsf{Popenv}; \mathsf{print}(); , T, H[e_0 = \mathsf{A}\ d_0], O, e_0.E\rangle \\
\overset{22}{\Rightarrow} & \langle \mathsf{Pushfun}(\mathsf{twice}); \mathsf{Popenv}; \mathsf{print}(); , T, H[d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.E\rangle \\
\overset{13}{\Rightarrow} & \langle \mathsf{MKTcomp}(\mathsf{A}'); \mathsf{eval_env}(1); \mathsf{Popenv}; \mathsf{print}(); , T, H[d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.E\rangle \\
\overset{17}{\Rightarrow} & \langle \mathsf{eval_env}(1); \mathsf{Popenv}; \mathsf{print}(); , d_1.T, H[d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.E\rangle \\
\overset{8}{\Rightarrow} & \langle \mathsf{eval}'(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \rightarrow \mathsf{rem.value})); \mathsf{Popenv}; \mathsf{print}(); , \\
& \qquad d_1.T, H[d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.E\rangle \\
\overset{6}{\Rightarrow} & \langle \mathsf{MKTvar}(0); \mathsf{eval_env}(1); \mathsf{Pop}(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Popenv}; \mathsf{print}(); , \\
& \qquad d_1.T, H[d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.e_0.E\rangle \\
\overset{16}{\Rightarrow} & \langle \mathsf{eval_env}(1); \mathsf{Pop}(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Popenv}; \mathsf{print}(); , \\
& \qquad d_0.d_1.T, H[d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.e_0.E\rangle \\
\overset{8}{\Rightarrow} & \langle \mathsf{eval}'(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Pop}(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Popenv}; \mathsf{print}(); , \\
& \qquad d_0.d_1.T, H[d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.e_0.E\rangle \\
\overset{5}{\Rightarrow} & \langle \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Pop}(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Popenv}; \mathsf{print}(); , \\
& \qquad d_0.d_1.T, H[d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.e_0.E\rangle \\
\overset{20}{\Rightarrow} & \langle \mathsf{Pop}(0); \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Popenv}; \mathsf{print}(); , \\
& \qquad d_2.d_1.T, H[d_2 = 8][d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.e_0.E\rangle \\
\overset{12}{\Rightarrow} & \langle \mathsf{MKTk}(0, \mathsf{fib}((*(\mathsf{topT})) \Rightarrow \mathsf{rem.value})); \mathsf{Popenv}; \mathsf{print}(); , \\
& \qquad d_2.T, H[d_2 = 8][d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.E\rangle \\
\overset{20}{\Rightarrow} & \langle \mathsf{Popenv}; \mathsf{print}(); , d_3.T, H[d_3 = 34][d_2 = 8][d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, e_0.E\rangle \\
\overset{11}{\Rightarrow} & \langle \mathsf{print}(); , d_3.T, H[d_3 = 34][d_2 = 8][d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], O, E\rangle \\
\overset{1}{\Rightarrow} & \langle\ , T, H[d_3 = 34][d_2 = 8][d_1 = (\mathsf{A}', e_0)][d_0 = 5][e_0 = \mathsf{A}\ d_0], 34.O, E\rangle
\end{array}$$

Compiling Lists

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

Scheme $\mathcal{E}$

6. $\mathcal{E}[[]] = \mathsf{printf}([]);$
7. $\mathcal{E}[a : b] = \mathcal{E}[a]\ \mathcal{E}[b]\ \mathsf{print}();$
8. $\mathcal{E}[\mathsf{Hd}(a \ldots b)] = \mathcal{T}^{[]}[a \ldots b]\ \mathsf{Hd}; \mathsf{printf}((*(\mathsf{topT}--)) \rightarrow \mathsf{rem.value});$
9. $\mathcal{E}[\mathsf{Tl}(a \ldots b)] = \mathcal{T}^{[]}[a \ldots b]\ \mathsf{Tl}; \mathsf{print}();$
10. $\mathcal{E}[\mathsf{Hd}(a : b)] = \mathcal{E}[a]$
11. $\mathcal{E}[\mathsf{Tl}(a : b)] = \mathcal{E}[b]$

Scheme $\mathcal{T}$

13. $\mathcal{T}^{x_0 \cdots x_j}[\ [\ ]\ ] = \mathsf{MKTlv};$
14. $\mathcal{T}^{x_0 \cdots x_j}[\mathsf{Hd}\ x_i] = \mathsf{Ehd}(i);$
15. $\mathcal{T}^{x_0 \cdots x_j}[\mathsf{Hd}(a : b)] = \mathcal{T}^{x_0 \cdots x_j}[a]$
16. $\mathcal{T}^{x_0 \cdots x_j}[\mathsf{Hd}(a \ldots b)] = \mathcal{T}^{x_0 \cdots x_j}[a \ldots b]\ \mathsf{Hd};$
17. $\mathcal{T}^{x_0 \cdots x_j}[a\ (\mathsf{Hd}\ b \ldots)] = \mathcal{T}^{x_0 \cdots x_j}[b \ldots]\ \mathsf{Hd}; \mathcal{T}^{x_0 \cdots x_j}[a]$
18. $\mathcal{T}^{x_0 \cdots x_j}[\mathsf{Tl}\ x_i] = \mathsf{Etl}(i);$
19. $\mathcal{T}^{x_0 \cdots x_j}[\mathsf{Tl}(a : b)] = \mathcal{T}^{x_0 \cdots x_j}[b]$
20. $\mathcal{T}^{x_0 \cdots x_j}[\mathsf{Tl}(a \ldots b)] = \mathcal{T}^{x_0 \cdots x_j}[a \ldots b]\ \mathsf{Tl};$
21. $\mathcal{T}^{x_0 \cdots x_j}[a\ (\mathsf{Tl}\ b \ldots)] = \mathcal{T}^{x_0 \cdots x_j}[b \ldots]\ \mathsf{Tl}; \mathcal{T}^{x_0 \cdots x_j}[a]$
22. $\mathcal{T}^{x_0 \cdots x_j}[a : b] = \mathcal{T}'^{x_0 \cdots x_j}[a]\ \mathcal{T}'^{x_0 \cdots x_j}[b]\ \mathsf{MKcons};$
23. $\mathcal{T}^{x_0 \cdots x_j}[\mathit{if}\ x_i = [\ ]\ \mathit{then}\ b\ \mathit{else}\ c] = \mathsf{Enull}\mathit{1}(\mathsf{i}, \mathsf{l}_1); \mathcal{T}^{x_0 \cdots x_j}[b]\ \mathsf{Jmp}\ \mathsf{l}_2; \mathit{l1} : \mathcal{T}^{x_0 \cdots x_j}[c]\ \mathsf{l}_2 :$

Scheme $\mathcal{G}$

9. $\mathcal{G}^{x_0 \cdots x_j}[\ [\ ]\ ]j = \mathsf{MKElv(j)};$
10. $\mathcal{G}^{x_0 \cdots x_j}[\mathsf{Hd}\ x_i]j = \mathsf{Ehd1}(i, j);$
11. $\mathcal{G}^{x_0 \cdots x_j}[\mathsf{Hd}(a : b)]j = \mathcal{G}^{x_0 \cdots x_j}[a]j$
12. $\mathcal{G}^{x_0 \cdots x_j}[\mathsf{Tl}\ x_i]j = \mathsf{Etl1}(i, j);$
13. $\mathcal{G}^{x_0 \cdots x_j}[\mathsf{Tl}(a : b)]j = \mathcal{G}^{x_0 \cdots x_j}[b]j$
14. $\mathcal{G}^{x_0 \cdots x_j}[a : b]j = \mathcal{L}^{x_0 \cdots x_j}[a]\ \mathcal{L}^{x_0 \cdots x_j}[b]\ \mathsf{MKEcons1}(j);$

Scheme $\mathcal{T}'$

9. $\mathcal{T}'^{x_0 \cdots x_j}[\ [\ ]\ ] = \mathsf{MKTlv};$
10. $\mathcal{T}'^{x_0 \cdots x_j}[\ \mathsf{Hd}\ x_i\ ] = \mathsf{Ehd}(\mathrm{i});$
11. $\mathcal{T}'^{x_0 \cdots x_j}[\mathsf{Hd}(a : b)] = \mathcal{T}'^{x_0 \cdots x_j}[a]$
12. $\mathcal{T}'^{x_0 \cdots x_j}[\ \mathsf{Tl}\ x_i\ ] = \mathsf{Etl}(\mathrm{i});$
13. $\mathcal{T}'^{x_0 \cdots x_j}[\mathsf{Tl}(a : b)] = \mathcal{T}'^{x_0 \cdots x_j}[b]$
14. $\mathcal{T}'^{x_0 \cdots x_j}[a : b] = \mathsf{MKTcomp(A)};$ $\quad$ *where* $\mathsf{A} \mapsto \mathcal{T}^{x_0 \cdots x_j}[\ \mathrm{a} : \mathrm{b}\ ]$

Scheme $\mathcal{L}$

1. $\mathcal{L}^{x_0 \cdots x_j}[\ x_i] = \mathsf{MKETvar}(\mathrm{i});$
2. $\mathcal{L}^{x_0 \cdots x_j}[\ \mathrm{k}\ ] = \mathsf{MKTcte}(\mathrm{k});$
3. $\mathcal{L}^{x_0 \cdots x_j}[a + b] = \mathsf{MKTcte}(\mathcal{Z}'^{x_0 \cdots x_j}[a] + \mathcal{Z}'^{x_0 \cdots x_j}[b]);$ $\quad$ if a and b are evaluated.
4. $\mathcal{L}^{x_0 \cdots x_j}[a + b] = \mathsf{MKTcomp(A)};$ $\quad$ where $\mathsf{A} \mapsto \mathcal{T}^{x_i \cdots x_j}[a + b];$
5. $\mathcal{L}^{x_0 \cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c] = \mathsf{IfE\_true}(\mathcal{T}^{x_0 \cdots x_j}[\mathrm{a}], \mathrm{B}, \mathrm{C})$ $\quad$ if a is evaluated.
   where $\mathsf{B} \mapsto \mathcal{T}^{x_0 \cdots x_j}[\mathrm{b}] \quad \mathsf{C} \mapsto \mathcal{T}^{x_0 \cdots x_j}[\mathrm{c}]$
6. $\mathcal{L}^{x_0 \cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c] = \mathsf{MKETcomp(A)};$ $\quad$ $\mathrm{A} \mapsto \mathcal{T}^{x_0 \cdots x_j}[\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c]$
7. $\mathcal{L}^{x_0 \cdots x_j}[\ \mathrm{a} : \mathrm{b}\ ] = \mathcal{L}^{x_0 \cdots x_j}[\ \mathrm{a}\ ]\ \mathcal{L}^{x_0 \cdots x_j}[b]\ \mathsf{MKcons};$ $\quad$ if a is evaluated.

8. $\mathcal{L}^{x_0 \cdots x_j}[\text{ a : b }] = \mathsf{MKETcomp}(\mathrm{A});$ where $\mathrm{A} \mapsto \mathcal{T}^{x_0 \cdots x_j}[\text{ a : b }]$

9. $\mathcal{L}^{x_0 \cdots x_j}[f_i x_0 \ldots x_m] = \mathsf{MKTcte}(\mathsf{fi}(\mathcal{Z}'^{x_0 \cdots x_j} x_0, \ldots, \mathcal{Z}'^{x_0 \cdots x_j} x_n));$ if f_i is a special function

10. $\mathcal{L}^{x_0 \cdots x_j}[f_i] = \mathsf{MKTpc}(\mathsf{f_i});$

11. $\mathcal{L}^{x_0 \cdots x_j}[\,[\,]\,] = \mathsf{MKTlv};$

12. $\mathcal{L}^{x_0 \cdots x_j}[\mathsf{Hd}\ x_i] = \mathsf{EThd(i)};$

13. $\mathcal{L}^{x_0 \cdots x_j}[\mathsf{Hd}(a : b)] = \mathcal{L}^{x_0 \cdots x_j}[a]$

14. $\mathcal{L}^{x_0 \cdots x_j}[\mathsf{Tl}\ x_i] = \mathsf{ETtl(i)};$

15. $\mathcal{L}^{x_0 \cdots x_j}[\mathsf{Tl}(a : b)] = \mathcal{L}^{x_0 \cdots x_j}[b]$

16. $\mathcal{L}^{x_0 \cdots x_j}[\text{ a} \ldots \text{b }] = \mathsf{MKETcomp}(\mathrm{A});$ where $\mathrm{A} \mapsto \mathcal{T}^{x_0 \cdots x_j}[\text{ a} \ldots \text{b }]$

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,

$$
\begin{array}{ll}
 & \mathcal{E}[\texttt{map (twice fib)}\ (1 : (2 : []))] \\
\overset{\mathcal{E}.5}{\Rightarrow} & \mathcal{T}^{[]}[map\ (twice\ fib)\ (1 : (2 : []))]; \mathsf{print}(); \\
\overset{\mathcal{T}.3}{\Rightarrow} & \mathsf{MKEcell}(2); \mathcal{G}^{[]}[twice\ fib]1; \mathcal{G}^{[]}[(1 : (2 : []))]0; \mathsf{Pushfun(map)}; \mathsf{Popenv}; \mathsf{print}(); \\
\overset{\mathcal{G}.7}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEcomp}(\mathsf{B}, 1); \mathcal{G}^{[]}[(1 : (2 : []))]0; \mathsf{Pushfun(map)}; \mathsf{Popenv}; \mathsf{print}(); \\
\overset{\mathcal{G}.11}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEcomp}(\mathsf{B}, 1); \mathcal{L}^{[]}[1]\ \mathcal{L}^{[]}[(2 : [])] \\
 & \qquad \mathsf{MKEcons1}(0); \mathsf{Pushfun(map)}; \mathsf{Popenv}; \mathsf{print}(); \\
\overset{\mathcal{L}.2}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEcomp}(\mathsf{B}, 1); \mathsf{MKTcte}(1);\ \mathcal{L}^{[]}[(2 : [])] \\
 & \qquad \mathsf{MKEcons1}(0); \mathsf{Pushfun(map)}; \mathsf{Popenv}; \mathsf{print}(); \\
\overset{\mathcal{L}.6}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEcomp}(\mathsf{B}, 1); \mathsf{MKTcte}(1);\ \mathcal{L}^{[]}[2]\ \mathcal{L}^{[]}[[]] \\
 & \qquad \mathsf{MKcons}; \mathsf{MKEcons1}(0); \mathsf{Pushfun(map)}; \mathsf{Popenv}; \mathsf{print}(); \\
\overset{\mathcal{L}.2}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEcomp}(\mathsf{B}, 1); \mathsf{MKTcte}(1); \mathsf{MKTcte}(2);\ \mathcal{L}^{[]}[[]] \\
 & \qquad \mathsf{MKcons}; \mathsf{MKEcons1}(0); \mathsf{Pushfun(map)}; \mathsf{Popenv}; \mathsf{print}(); \\
\overset{\mathcal{L}.12}{\Rightarrow} & \mathsf{MKEcell}(2); \mathsf{MKEcomp}(\mathsf{B}, 1); \mathsf{MKTcte}(1); \mathsf{MKTcte}(2); \mathsf{MKTlv}; \\
 & \qquad \mathsf{MKcons}; \mathsf{MKEcons1}(0); \mathsf{Pushfun(map)}; \mathsf{Popenv}; \mathsf{print}();
\end{array}
$$

where B is:

$$
\begin{array}{lll}
\mathsf{B} & \mapsto & \mathcal{T}^{[]}[twice\ fib] \\
 & \overset{\mathcal{T}.12}{\Rightarrow} & \mathcal{T}'^{[]}[fib]\ \mathcal{T}^{[]}[twice] \\
 & \overset{\mathcal{T}'.8}{\Rightarrow} & \mathsf{MKTpc(A)}; \mathcal{T}^{[]}[twice] \\
 & \overset{\mathcal{T}.5}{\Rightarrow} & \mathsf{MKTpc(A)}; \mathsf{MKenv}(2); \mathsf{Pushfun(twice)}; \mathsf{Popenv};
\end{array}
$$

and A is as in the last example of compilation. Now we translate `map` by using scheme $\mathcal{T}$.

$$
\begin{array}{lll}
\mathsf{map} & \mapsto & \mathcal{T}^{f,x}[if\ x = []\ then\ []\ else\ f(Hd\ x) : map\ f(Tl\ x)] \\
 & \overset{\mathcal{T}.23}{\Rightarrow} & \mathsf{Enull}(0, \mathsf{l_1}); \mathcal{T}^{f,x}[[]]\ \mathsf{Jmpl_2}; \mathsf{l_1} : \mathcal{T}^{f,x}[f(Hd\ x) : map\ f(Tl\ x)]\ \mathsf{l_2} : \\
 & \overset{\mathcal{T}.13}{\Rightarrow} & \mathsf{Enull}(0, \mathsf{l_1}); \mathsf{MKTlv}; \mathsf{Jmpl_2}; \mathsf{l_1} : \mathcal{T}^{f,x}[f(Hd\ x) : map\ f(Tl\ x)]\ \mathsf{l_2} : \\
 & \overset{\mathcal{T}.22}{\Rightarrow} & \mathsf{Enull}(0, \mathsf{l_1}); \mathsf{MKTlv}; \mathsf{Jmpl_2}; \mathsf{l_1} : \mathcal{T}^{f,x}[f(Hd\ x)]\mathcal{T}'^{f,x}[map\ f(Tl\ x)]; \mathsf{MKcons};\ \mathsf{l_2} :
\end{array}
$$

$\stackrel{\mathcal{T}.9}{\Rightarrow}$ Enull(0, l_1); MKTlv; Jmpl_2; l_1 :$\mathcal{T}^{f,x}[f(Hd\ x)]\mathcal{T}'^{f,x}[map\ f(Tl\ x)]$; MKcons; l_2 :

$\stackrel{\mathcal{T}.18}{\Rightarrow}$ Enull(0, l_1); MKTlv; Jmpl_2; l_1 :$\mathcal{T}^{f,x}[x]$ Hd; $\mathcal{T}^{f,x}[f]\ \mathcal{T}'^{f,x}[map\ f(Tl\ x)]$; MKcons; l_2 :

$\stackrel{\mathcal{T}.7}{\Rightarrow}$ Enull(0, l_1); MKTlv; Jmpl_2; l_1 :MKTvar(0); Hd; $\mathcal{T}^{f,x}[f]\ \mathcal{T}'^{f,x}[map\ f(Tl\ x)]$; MKcons; l_2 :

$\stackrel{\mathcal{T}.8}{\Rightarrow}$ Enull(0, l_1); MKTlv; Jmpl_2; l_1 :MKTvar(0); Hd; eval_env(1); $\mathcal{T}'^{f,x}[map\ f(Tl\ x)]$; MKcons; l_2 :

$\stackrel{\mathcal{T}'.7}{\Rightarrow}$ Enull(0, l_1); MKTlv; Jmpl_2; l_1 :MKTvar(0); Hd; eval_env(1); MKTcomp(C); MKcons; l_2 :

where,

C $\mapsto$ $\mathcal{T}^{f,x}[map\ f(Tl\ x)]$

$\stackrel{\mathcal{T}.3}{\Rightarrow}$ MKEcell(2); $\mathcal{G}^{f,x}[f]1\ \mathcal{G}^{f,x}[(Tl\ x)]0$ Pushfun(map); Popenv;

$\stackrel{\mathcal{G}.2}{\Rightarrow}$ MKEcell(2); MKEvar(1, 1); $\mathcal{G}^{f,x}[(Tl\ x)]0$ Pushfun(map); Popenv;

$\stackrel{\mathcal{G}.10}{\Rightarrow}$ MKEcell(2); MKEvar(1, 1); Etl1(0, 0); Pushfun(map); Popenv;

New State Transition Laws

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

28. ⟨ print.c, d.T, H[d= $a:b$], O, E ⟩ ⇒ ⟨ eval.print.eval.print.c, $a.b$.T, H[d=$a:b$], O, E ⟩
29. ⟨ print.c, d.T, H[d=[ ]], O, E ⟩ ⇒⟨ c, T, H[d=[ ]], O, E ⟩
30. ⟨ Hd.c, d.T, H[d=$a:b$], O, E ⟩ ⇒ ⟨ c, a.T, H[d=$a:b$], O, E ⟩
31. ⟨ Tl.c, d.T, H[d=$a:b$], O, E ⟩ ⇒ ⟨ c, b.T, H[d=$a:b$], O, E ⟩
32. ⟨ MKcons.c, $d_1.d_2$.T, H[d_2= b][d_1= a], O, E ⟩ ⇒ ⟨ c, d.T, H[d=d_2:d_1], O, E ⟩
33. ⟨ MKTlv.c, T, H, O, E ⟩ ⇒ ⟨ c, d.T, H[d= [ ]], O, E ⟩
34. ⟨ MKElv(i).c, T, H[$e_0 = \ldots u_i \ldots$], O, e_0.E ⟩ ⇒ ⟨ c, T, H[d=[ ]][e_0=$\ldots d \ldots$], O, e_0.E ⟩
35. ⟨ MKEcons1(i).c, $d_1.d_2$.T, H[d_1= a][d_2=b][$e_1 = \ldots u_i \ldots$], O, e_1.E ⟩ ⇒ ⟨ c, T, H[d=$d_2 : d_1$][e_1=$\ldots d \ldots$], O, e_1.E ⟩
36. ⟨ MKETvar(i).c, T, H[$e_0 = \ldots u_i \ldots$], O,e_1, e_0.E ⟩ ⇒ ⟨ c.d.T, H[$d = u_i$][$e_0 = \ldots u_i \ldots$], O, e_1, e_0.E ⟩
37. ⟨ MKETcomp(A).c, T, H, O,e_1, e_0.E ⟩ ⇒ ⟨ c.d.T, H[$d = (A, e_0)$], O, e_1, e_0.E ⟩
38. ⟨ Ehd(i).c, T, H[$e_1 = \ldots (a:b) \ldots$], O, e_1.E ⟩ ⇒⟨ c, a.T, H[e_1=$\ldots$], O, e_1.E ⟩
39. ⟨ Etl(i).c, T, H[$e_1 = \ldots (a:b) \ldots$], O, e_1.E ⟩ ⇒⟨ c, b.T, H[e_1=$\ldots$], O, e_1.E ⟩
40. ⟨ EThd(i).c, T, H[$e_0 = \ldots (a:b) \ldots$], O, $e_1.e_0$.E ⟩ ⇒⟨ c, a.T, H[e_0=$\ldots$], O, $e_1.e_0$.E ⟩
41. ⟨ ETtl(i).c, T, H[$e_0 = \ldots (a:b) \ldots$], O, $e_1.e_0$.E ⟩ ⇒⟨ c, b.T, H[e_0=$\ldots$], O, $e_1.e_0$.E ⟩
42. ⟨ Ehd1(i,j).c, T, H[$e_1 = \ldots x_j \ldots$] [$e_0 = \ldots (a:b) \ldots$], O, $e_1.e_0$.E ⟩ ⇒⟨ c, T, H [$e_1 = \ldots a \ldots$] [$e_0 = \ldots$], O, $e_1.e_0$.E ⟩
43. ⟨ Etl1(i,j).c, T, H[$e_1 = \ldots x_j \ldots$] [$e_0 = \ldots (a:b) \ldots$], O, $e_1.e_0$.E ⟩ ⇒⟨ c, b.T, H [$e_1 = \ldots b \ldots$][e_0=$\ldots$], O, $e_1.e_0$.E ⟩
44. ⟨ Enull(n,l_1) $\ldots l_1$:c, T, H[d_0=[ ]][e_0= $\ldots d_n \ldots$], e_0.E ⟩ ⇒ ⟨ c,T,H[d_0=[ ]][e_0= $\ldots d_n \ldots$], e_0.E ⟩
45. ⟨ Jmp l $\ldots$l:c, T, H, O, E ⟩ ⇒ ⟨ c, T, H, O, E ⟩
46. ⟨ Jfalse(False,l) $\ldots$l:c, T, H, O, E ⟩ ⇒ ⟨ c, T, H, O, E ⟩
47. ⟨ Jfalse(True,l) .c, T, H, O, E ⟩ ⇒ ⟨ c, T, H, O, E ⟩

Example of Evaluation

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

```
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:[]))?
```

compiled as,

$$
\begin{array}{rcl}
\texttt{map (twice fib) (1 : (2 : []))} & \mapsto & \mathsf{MKEcell(2); MKEcomp(B,1); MKTcte(1); MKTcte(2); MKTlv;} \\
 & & \mathsf{MKcons; MKcons; MKEcons1}(0)\mathsf{; Pushfun(map); Popenv; print();} \\
\mathsf{B} & \mapsto & \mathsf{MKTpc(A); MKenv(2); Pushfun(twice); Popenv;} \\
\mathsf{map} & \mapsto & \mathsf{Enull(0,l_1); MKTlv; Jmpl_2; l_1 :MKTvar(0); Hd; eval\_env(1); MKTcomp(C); MKcons;\ \ l_2 :} \\
\mathsf{C} & \mapsto & \mathsf{MKEcell(2); MKEvar(1,1); Etl1(0,0); Pushfun(map); Popenv;} \\
\mathsf{A} & \mapsto & \mathsf{eval'(0); MKTk(0, fib((*(topT)) \rightarrow rem.value));} \\
\mathsf{twice} & \mapsto & \mathsf{MKTcomp(A'); eval\_env(1);} \\
\mathsf{A'} & \mapsto & \mathsf{MKTvar(0); eval\_env(1);} \\
\mathsf{fib} & \mapsto & \mathsf{if\ (n < 2)\ \{return(1)\}; else\{return(fib(n-1) + fib(n-2))\};}
\end{array}
$$

The initial state of the machine is:

$$\langle\ \mathsf{MKEcell(2);\ MKEcomp(B,1);\ MKTcte(1);\ MKTcte(2);\ MKTlv;}\ \ldots,\mathrm{T,H,O,E}\rangle$$

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

$$
\begin{array}{ll}
\overset{23}{\Rightarrow} & \langle \mathsf{MKEcomp(B,1); MKTcte(1); MKTcte(2); MKTlv;} \ldots, T, H[e_0 = 10], O, e_0.E\rangle \\
\overset{24}{\Rightarrow} & \langle \mathsf{MKTcte(1); MKTcte(2); MKTlv;} \ldots, T, H[e_1 = (B, e_0)][e_0 = e_1 0], O, e_0.E\rangle \\
\overset{19}{\Rightarrow} & \langle \mathsf{MKTcte(2); MKTlv;} \ldots, d_0.T, H[d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 0], O, e_0.E\rangle \\
\overset{19}{\Rightarrow} & \langle \mathsf{MKTlv;} \ldots, d_1.d_0.T, H[d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 0], O, e_0.E\rangle \\
\overset{33}{\Rightarrow} & \langle \mathsf{MKcons; MKEcons1}(0)\mathsf{; Pushfun(map); Popenv; print();}, d_2.d_1.d_0.T, \\
 & \qquad H[d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 0], O, e_0.E\rangle \\
\overset{32}{\Rightarrow} & \langle \mathsf{MKEcons1}(0)\mathsf{; Pushfun(map); Popenv; print();}, d_3.d_0.T, \\
 & \qquad H[d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 0], O, e_0.E\rangle
\end{array}
$$

at this point of execution the graph for the list is complete and we enter the code for `map`.

$$
\begin{array}{ll}
\overset{35}{\Rightarrow} & \langle \mathsf{Pushfun(map); Popenv; print();}, d_4.T, H[d_4 = d_1 : d_3][d_3 = d_1 : d_2][d_2 = [\ ]] \\
 & \qquad [d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 0], O, e_1.e_0.E\rangle \\
\overset{13}{\Rightarrow} & \langle \mathsf{Enull(0,l_1); MKTlv; Jmpl_2; l_1 :MKTvar(0); Hd; eval\_env(1); MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, d_4.T, \\
 & \qquad H[d_4 = d_0 : d_3][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.E\rangle \\
\overset{44}{\Rightarrow} & \langle \mathsf{MKTvar(0); Hd; eval\_env(1); MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, d_4.T, \\
 & \qquad H[d_4 = d_0 : d_3][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.E\rangle \\
\overset{16}{\Rightarrow} & \langle \mathsf{Hd; eval\_env(1); MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, d_4.T, \\
 & \qquad H[d_4 = d_0 : d_3][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.E\rangle \\
\overset{30}{\Rightarrow} & \langle \mathsf{eval\_env(1); MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, d_0.T, \\
 & \qquad H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.E\rangle \\
\overset{9}{\Rightarrow} & \langle \mathsf{eval; MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, d_0.T, \\
 & \qquad H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.E\rangle \\
\overset{3}{\Rightarrow} & \langle \mathsf{MKTpc(A); MKenv(2); Pushfun(twice); Popenv; Popenv; MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, d_0.T, \\
 & \qquad H[d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.e_0.E\rangle \\
\overset{18}{\Rightarrow} & \langle \mathsf{MKenv(2); Pushfun(twice); Popenv; Popenv; MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, d_5.d_0.T, \\
 & \qquad H[d_5 = A][d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.e_0.E\rangle \\
\overset{10}{\Rightarrow} & \langle \mathsf{Pushfun(twice); Popenv; Popenv; MKTcomp(C); MKcons;\ \ l_2 :Popenv; print();}, e_2.T, \\
 & \qquad H[e_2 = d_5 d_0][d_5 = A][d_4 = d_0 : d_2][d_3 = d_1 : d_2][d_2 = [\ ]][d_1 = 2][d_0 = 1][e_1 = (B, e_0)][e_0 = e_1 d_4], O, e_0.e_0.E\rangle
\end{array}
$$

Now the code for `twice` is called taking as arguments `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 ΓCMC evaluates lists.

Optimisations

A number of code optimisations should be introduced to ΓCMC 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 ΓCMC . 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 TIM [2]. Now, the user provides annotations (U combinator) to specify variables one wants to share.
The U combinator performs the following state transition:

48. $\langle$ U(i).c,d.T, H$[e_0 = \ldots a_i \ldots]$, O, e_0.E$\rangle$ $\Rightarrow$$\langle$ c, d.T, H$[e_0{=}\ldots d \ldots]$, O, e_0.E$\rangle$

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

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

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 ΓCMC we avoid garbage generation by compiling tail recursive functions as

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

by the following entries in the script:

$f_n \mapsto$ Enull(i, $\mathsf{l_1}$); $\mathcal{T}^{x_0 \cdots x_n}[b]$ Jmp $\mathsf{l_2}$; $\mathsf{l_1}$:$\mathcal{T}^{x_0 \cdots x_n}[z]$ MKTcomp(A); MKenv(n + 1); MKcons; $\mathsf{l_2}$:
A $\mapsto \mathcal{G}^{x_0 \cdots x_n}[y_n]n \ldots \mathcal{G}^{x_0 \cdots x_n}[y_0]0$ Swap; Pushfun($\mathsf{f'_n}$); Popenv;
$\mathsf{f'_n} \mapsto$ Enull(i, $\mathsf{l_1}$); $\mathcal{T}^{x_0 \cdots x_n}[b]$ Jmp $\mathsf{l_2}$; $\mathsf{l_1}$:$\mathcal{T}^{x_l \cdots x_j}[z]$ MKTcomp(A); MKcons; $\mathsf{l_2}$:

The state transition law for Swap is:

49. $\langle$ Swap.c, T, H, O, $e_1.e_0$.E$\rangle$ $\Rightarrow$$\langle$ c, A.T, H, O, $e_0.e_1$.E$\rangle$

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

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

by using the following scheme:

$f_n \mapsto$ pushR(0);
LP1 :Jfalse($\mathcal{T}^{x_0 \cdots x_j}[a]$; t1 = (∗(topT−−)) → rem.value;
$\mathcal{T}^{x_0 \cdots x_j}[b]$; t2 = (∗(topT−−)) → rem.value; t1 > t2;, $\mathsf{l_1}$); $\mathcal{T}^{x_0 \cdots x_j}c$;
Jmp l_2;
$\mathbf{l_1}$:MKEcell(n + 1);
$\mathcal{G}^{x_0 \cdots x_n}[y_n]n$
⋮
$\mathcal{G}^{x_0 \cdots x_n}[y_0]0$
pushR(1); Jmp(LP1);
LP2 : Popenv;
$\mathbf{l_2}$: if(∗(topR−−)) == 1) Jmp(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 Sparckstation 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.

Prog	*Fib 30*	*Rev*	*Sieve*	*Insord*	*Simlog*	*Map*	*Tak*
GM-C	10.6	—	2.8	5.0	1.6	2.2	2:38.2
Γ*CMC*	1.5	2.5	1.4	2.9	0.2	0.5	2.8
LML	8.9	4.0	1.6	2.5	0.6	0.4	42.8

GM-C corresponds to the last version of GM-C [8] done by Musicante and Lins. Γ*CMC* refers to our best implementation of ΓCMC. *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 ΓCMC is far better than GM-C, for all our benchmark programs. ΓCMC 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 ΓCMC performed far better than LML.

Further Work & Conclusions

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

In our opinion, ΓCMC 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 ΓCMC is in the worst case as good as the Chalmers LML compiler. In the best case, ΓCMC 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

[1] G.L.Burns, S.L.Peyton Jones and J.D.Robson. The spineless g-machine. In *Proc.ACM Conference on Lisp and Functional Programming*, pages 244–258, Snowbird, USA, 1988.

[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] L.Cardelli. The functional abstract machine. *Polymorphism*, 1, 1983.

[4] 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.

[5] R.D.Lins. Partial Categorical Multi-Combinators. UKC-Computing Lab.Report 7/92, The University of Kent at Canterbury, April 1992

[6] R.D.Lins & S.J.Thompson. CM-CM: A Categorical Multi-Combinator machine. In *Proceedings of XVI LatinoAmerican Conference on Informatics*, Assuncion, Paraguay, September 1990.

[7] R.D.Lins & S.J.Thompson. Implementing SASL using Categorical Multi-Combinators. *Software — Practice and Experience*, 20(8):1137–1165, November 1990.

[8] M.A.Musicante & R.D.Lins. GMC: A Graph Multi-Combinator Machine. *Microprocessing and Microprogramming*, 31:31–35, April 1991.

[9] S.L.Peyton Jones and J.Salkild. The spineless tagless g-machine. In *Proc.ACM Conference on Functional Programming Lannguages and Computer Architecture*, pages 184–201, Snowbird, USA, 1989.

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

[11] S.J.Thompson & R.D.Lins. The Categorical Multi-Combinator Machine:CM-CM. *The Programming Journal*, vol 35(2):170-176, BCS, Cambridge University Press, April 1992.

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