A back-end for GHC based on Categorical Multi-Combinators

Ricardo Massa F. Lima  
Dept. Sistemas Computacionais  
Universidade de Pernambuco  
ricardo@upe.poli.br

Rafael Dueire Lins  
Dept. Eletrônica e Sistemas  
Universidade Federal de Pernambuco  
rdl@ee.ufpe.br

André L. M. Santos  
Centro de Informática  
Universidade Federal de Pernambuco  
alm@cin.ufpe.br

ABSTRACT

\( \mu \Gamma \text{CMC} \) is an abstract graph reduction machine for the implementation of lazy functional languages. Categorical multi-combinators served as a basis for the evaluation model of \( \mu \Gamma \text{CMC} \). This paper presents the implementation of a Haskell compiler, using the front-end of the Glasgow Haskell Compiler (GHC) and a new back-end based on the \( \mu \Gamma \text{CMC} \) abstract machine. A number of code optimisations are introduced to \( \mu \Gamma \text{CMC} \). The performance of our implementation is benchmarked against the Glasgow Haskell Compiler, one of the most efficient Haskell compilers available.

1. INTRODUCTION

Haskell [7] is a general purpose, pure functional programming language incorporating many recent innovations in programming language research, including higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, automatic garbage collection, pattern-matching, list comprehension, a module system, monads, and a rich set of primitive datatypes, including arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell has now become a de facto standard for the non-strict functional programming community, with several compilers available.

Functional languages seem to be harder to implement than conventional imperative ones. At execution time, they maintain complex structures, such as unevaluated function applications, which allow us to work with higher-order functions and infinite lists. Traditionally, lazy functional languages were implemented using graph interpretation of combinators, as introduced by Turner [22]. The understanding of the evaluation mechanisms of these languages allowed implementations to move from interpretation towards compilation, with gains in performance. Cardelli’s abstract machine FAM[1], developed for the compilation of strict functional languages, was an important step in this quest for efficiency. Johnsson [9] developed a strategy for compiling lazy functional languages, described as an abstract machine, called G-machine. The basic principle of the G-machine is to avoid generating graphs. The G-machine method of controlling the execution flow and evaluation was followed by other implementations, including those based on different abstract machines such as the spineless G-machine[18].

\( \mu \Gamma \text{CMC} \) is a new abstract machine for the implementation of functional languages which inherited features of the GCMC machine [14, 15] and introduces new characteristics and optimisations to support an efficient implementation of Haskell. In \( \mu \Gamma \text{CMC} \), C was used as a macro-assembler and the ‘execution flow control’ is transferred to C, as much as possible. The object code generated by C compilers is very fast. These factors lead us to translate some function definitions into procedures in C. It is obvious that not all functions may be translated into C if a lazy functional language is aimed at. However, it is safe to translate strict function on all its arguments that produce unboxed results of basic type (integer, floating-point, character, etc.) as procedures in C. A higher-level abstract machine is still needed to glue together procedure calls, unevaluated expressions, data-structures, etc. Categorical multi-combinators (CM-C) [21, 12] served as a basis for the evaluation model of the \( \mu \Gamma \text{CMC} \) abstract machine. The experience with GM-C [17], CM-CM[16, 21] and GCMC[14, 15] was fundamental for the design, implementation and optimisation of \( \mu \Gamma \text{CMC} \).

This work describes the new abstract machine \( \mu \Gamma \text{CMC} \). Section 2 introduces the categorical multi-combinators. The foundation of the \( \mu \Gamma \text{CMC} \) machine is presented in Section 3. The elimination of the environment cells by pushing the environment directly in the reduction stack distinguishes the new machine from the GCMC machine. Amongst other things, such a modification demands a special mechanism to compile higher-order functions, which is detailed in Section 4. The Haskell front-end used in our implementation is described in Section 5. Section 6 details an example of the compilation of a Haskell program into \( \mu \Gamma \text{CMC} \). The evaluation of the resulting \( \mu \Gamma \text{CMC} \) code is described in Section 7. Some optimisations and the performance evaluation are

Table 1: translating \( \lambda \)-expressions into categorical multi-combinators

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>( R^1 \lambda x_0 ... \lambda x_{A-1} a = L'(R^{x_0 ... x_{A-1}} a) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R^a_0 )</td>
<td>( R^a_{x_0} a ) → ( R^{x_0 ... x_{A-1}} a ) ... ( R^{x_0 ... x_{A-1}} b )</td>
</tr>
<tr>
<td>( R^a_{x_0} b )</td>
<td>( b ) if ( b ) is a constant</td>
</tr>
<tr>
<td>( R^a_{x_0} )</td>
<td>( i )</td>
</tr>
</tbody>
</table>
2.1 Compiling the $\lambda$-calculus into categorical multi-combinators

In categorical multi-combinators, function application is denoted by juxtaposition, taken to be left-associative. The compilation algorithm for translating $\lambda$-expressions into categorical multi-combinators (Table 1) is given by the function $R^{\mu\Gamma\text{CMC}}$, where $x_i$ 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, denoted by $R^{0}$. For a matter of uniformity, combinators will be represented as composed with a dummy frame, $(\ )$, which can be seen as the identity frame. Combinator names are treated as constants.

Table 2 presents an example of translations of a program into the categorical multi-combinators using the compilation schemes described.

2.2 Categorical multi-combinator rewriting laws

The core of the categorical multi-combinator machine is presented on page 71 of Reference [12]. For a matter of convenience, the multi-pair combinator, which forms evaluation environments, will be represented as $(x_0, \ldots, x_n)$ and closures will be written as $(a, b)$, where $b$ represents the environment. Using this notation the kernel of the categorical multi-combinator rewriting laws is described on Table 3.

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. It is equivalent to $\lambda$-calculus $\beta$-reduction, in which substitutions are performed on demand. If a combinator reaches the leftmost position of the code during rewriting, it proceeds with a script look-up and enters the corresponding code in the definition environment. (M.4) expresses this situation.

2.3 Example of evaluation

The expression SKKI, where S, K and I correspond to the CM-C code presented on Table 2, is evaluated as presented on Table 4.

3. THE $\mu$ΓCMC MACHINE

Amongst the implementations of compiled functional language based on categorical multi-combinators there are CM-CM [16, 21] and GM-C [17]. ΓCMC is an evolution of these...
The strategy of storing the environment in the reduction stack provides the depth of a value in the environment cell. One of the main innovations brought by the use of this stack creates an extra indirection level to access the top of which points to the current environment. The transition rules (4) and (5) were included to remove the environment from the evaluation stack in two distinct situations.

The evaluation of the expression $\text{SKKI}$ through the new machine $\mu\text{CMC}$ is presented on Table 7. The symbol ‘$e$’ designates the empty environment. A set of terms separated by the symbol ‘,’ indicates the order in which terms were pushed onto the stack. For instance, the sequence ‘$a,b,c$’ indicates that ‘$a$’ was pushed in front of $b$, and $b$ was pushed in front of $c$.

### 4. Higher-Order Functions

Due to the elimination of the environment cell by storing the environment directly in the reduction stack, the compilation of higher-order function requires special attention. Consider the function $f_{\#2}$, of arity 2, which is applied to three parameters: $f_{\#2} x y z$. The extra parameter $z$ is consumed by the partial function returned from the evaluation of $f_{\#2} x y$. However, the evaluation of $f_{\#2} x y$ may call other functions before returning the partial function. Therefore, when the evaluation finishes, the parameter $z$ will be in a position of the reduction stack which cannot be statically predicted. Consequently, the evaluation of the partial function must be postponed until the required parameter becomes accessible again at the top of stack $T$. $\mu\text{CMC}$ creates a closure for this partial function. The code of the closure fetches the missing parameters at the top of the reduction stack and applies the partial function to them.

In some cases it is not possible to know in advance the number of arguments a function will consume. For instance, consider the variable $x$ of type $A \rightarrow B$. One could think that $x$ consumes a single argument. However, since the type variables $A$ and $B$ can represent other functions, the application of $x$ can consume an unpredictable number of arguments. $\mu\text{CMC}$ uses a macro called $\text{eval}$ to force the evaluation stack has the advantage of reducing the access time. The index used to fetch a value in the environment is computed by adding the DeBruijn number to the distance, called $\delta$, between the environment and the top of stack $T$. $\delta$ changes whenever the expression on top of $T$ is computed. It is important to say that the control over $\delta$ is performed statically. Therefore, it does not affect the performance. Table 6 presents the rewriting laws of the categorical multi-combinator in this new evaluation model. The laws (4) and (5) were included to remove the environment from the evaluation stack in two distinct situations.

The evaluation of the expression $\text{SKKI}$ through the new machine $\mu\text{CMC}$ is presented on Table 7. The symbol ‘$e$’ designates the empty environment. A set of terms separated by the symbol ‘,’ indicates the order in which terms were pushed onto the stack. For instance, the sequence ‘$a,b,c$’ indicates that ‘$a$’ was pushed in front of $b$, and $b$ was pushed in front of $c$.

Table 5: State transition rules for the categorical multi-combinators

<table>
<thead>
<tr>
<th>No.</th>
<th>Expression</th>
<th>Transition Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\langle n, e \rangle \cdot c, H[e = (x_m, \ldots, x_0)]$</td>
<td>$\Rightarrow \langle x_{n}, c, H[e = (x_m, \ldots, x_0)] \rangle$</td>
</tr>
<tr>
<td>2</td>
<td>$\langle x_0 \cdot \ldots \cdot x_n, e \rangle \cdot c, H(e = \cdots)$</td>
<td>$\Rightarrow \langle x_0, e \cdot \cdots \cdot (x_n, e) \cdot c, H(e = \cdots) \rangle$</td>
</tr>
<tr>
<td>3</td>
<td>$\langle L^n(y), e \rangle x_0 \cdot \ldots \cdot x_n x_{n+1} \cdots x_{m}, c, H \rangle$</td>
<td>$\Rightarrow \langle y, e \rangle x_{n+1} \cdots x_{m}, c, H[e_j = (x_0 \cdots x_n, e_j)] \rangle$</td>
</tr>
<tr>
<td>4</td>
<td>$\langle f, e \rangle \cdot c, H(e = \cdots) \rangle$</td>
<td>$\Rightarrow \langle f_n, e \rangle \cdot c, H(e = \cdots) \rangle$</td>
</tr>
</tbody>
</table>

Table 6: Example of evaluation

<table>
<thead>
<tr>
<th>No.</th>
<th>Expression</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S K K I$</td>
<td>$\Rightarrow^{M4} L^2(2 \ 0 \ (1 \ 0)) \ K K I$</td>
</tr>
<tr>
<td>2</td>
<td>$K K I$</td>
<td>$\Rightarrow^{M3} (2 \ 0 \ (1 \ 0)) \ K K I$</td>
</tr>
<tr>
<td>3</td>
<td>$2 \ K K I$</td>
<td>$\Rightarrow^{M2} (0, K K I) \ K K I \ (1 \ 0, K K I)$</td>
</tr>
<tr>
<td>4</td>
<td>$K \ K K I$</td>
<td>$\Rightarrow^{M1} K \ (0, K K I) \ (1 \ 0, K K I)$</td>
</tr>
<tr>
<td>5</td>
<td>$L^1(1) \ (0, K K I) \ (1 \ 0, K K I)$</td>
<td>$\Rightarrow^{M2} 1 \ (0, K K I) \ (1 \ 0, K K I)$</td>
</tr>
<tr>
<td>6</td>
<td>$0, K K I$</td>
<td>$\Rightarrow^{M1} 0, K K I$</td>
</tr>
<tr>
<td>7</td>
<td>$I$</td>
<td>$\Rightarrow^{M1}$</td>
</tr>
</tbody>
</table>
of structures at the top of stack T. This instruction is invoked whenever it is not possible to predict the behaviour of the program. It can be thought as a transition from the compilation to the interpretation level. \textit{eval} starts an iterative process which finishes only when the element on top of stack T is completely evaluated. Notice that the semantics of \textit{eval} is not compatible with the case where the number of arguments consumed by an application cannot be predicted. If the function does not consume the complete set of applied parameters, its evaluation will return a partial application.

\section{Reusing a Haskell Front-End}

The Haskell functional language has several higher level features \cite{7}. Thus, constructing a Haskell front-end from scratch is a very complex task. In addition, we are mainly interested in evaluating a new abstract machine which implements the back-end of a Haskell compiler. Therefore, it has been decided to plug the abstract machine \(\mu\GammaCMC\) onto an existing Haskell front-end, namely the Glasgow Haskell Compiler (GHC) version 0.29 front-end \cite{23}. The intermediate language \textit{Shared Term Graph} (STG) \cite{10} has been used as the interface between the GHC front-end and the \(\mu\GammaCMC\). The GHC front-end was slightly modified to incorporate type information and function arity in the STG code. This intermediate code is output to a file, and read by the back-end.

The idea of reusing GHC’s front-end therefore has three objectives: first to avoid reimplementing all the front-end of a Haskell compiler, a daunting task itself; second, to allow the compilation of real, large programs, which would be difficult to develop with a simple compiler that did not implement full Haskell; and finally to allow a direct, fair comparison, using the same source program, of the two back-ends, taking the same benefits from higher level program optimisations that exist in GHC’s front-end \cite{19}. Reimplementing all these optimisations would also be a major programming effort.

\section{Compiling Haskell into \(\mu\GammaCMC\)}

The compilation of Haskell programs into the kernel of the \(\mu\GammaCMC\) abstract machine is presented in this section through an example. The compilation schemes to translate STG programs into \(\mu\GammaCMC\) are available at the URL \url{http://www.upe.poli.br/simdsc/mgcmc}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Table 6: New rewriting laws of the categorical multi-combinators} & \\
\hline
(1) \((\langle n \mapsto \ast \rangle . y_1 . y_2 . \ldots . y_s . c . e, H[e_{\langle (x_0, e_0), (x_1, e_1), \ldots (x_m, e_m) \rangle}]) \Rightarrow (x_n . e_n . y_1 . y_2 . \ldots . y_s . c . e, H[e_{\langle (x_0, e_0), (x_1, e_1), \ldots (x_m, e_m) \rangle}]) & \\
(2) \((\langle L^2(y_1 . e) . e \rangle . x_0 . x_1 . x_2 . \ldots . x_m . c . e, H) \Rightarrow (y_1 . e . x_1 . x_2 . \ldots . x_m . c . e, H[e_{\langle (x_0, e_0), (x_1, e_1), (x_2, e_2) \rangle}]) & \\
(3) \((\langle e . c . e \rangle . H) \Rightarrow (e . c . e) & \\
(4) \((n . e . c . H[e = (\langle x_0, e_0 \rangle . (x_1, e_1) \cdots (x_m, e_m) \rangle)] \Rightarrow (x_n . e_n . c . H & \\
(5) \((\langle f . e \rangle . x_0 . \ldots . x_n . c . e, H) \Rightarrow (\langle L^2(y, e) . x_0 . \ldots . x_n . c . e, H) & \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Table 7: Example of evaluation using the \(\mu\GammaCMC\) machine} & \\
\hline
\((S \ K \ K \ I, H) \Rightarrow \langle (\langle L^2(2 \ 0 \ (1 \ 0)), e \rangle . K(KI, H) \Rightarrow \langle 2 \ (0 \ ((0 \ 1), e_1, H[e_1 = I.K.K]) \Rightarrow \langle K(0 \ ((0 \ 1), e_1, H[e_1 = I.K.K) \Rightarrow \langle \langle L^1(1), e_1, 0 \ (1 \ 0), H[e_1 = I.K.K]\rangle) \Rightarrow \langle 1 \ e_2, H[e_2 = \langle (1 \ 0), e_1, 0 \ (1 \ 0), e_1 \rangle e_1 = I.K.K]\rangle) \Rightarrow \langle 0 \ e_1, H[e_2 = \langle (1 \ 0), e_1, 0 \ (1 \ 0), e_1 \rangle e_1 = I.K.K]\rangle) \Rightarrow \langle I, H[e_2 = \langle (1 \ 0), e_1, 0 \ (1 \ 0), e_1 \rangle e_1 = I.K.K]\rangle & \\
\hline
\end{tabular}
\end{table}

Consider the following Haskell program:

\begin{verbatim}
twice f x = f (f x)
succ n = n + 1
main = print (I# (twice twice twice succ 3#))
\end{verbatim}

Figure 2 presents the equivalent program written on the intermediate language STG. The translation of functions \textit{main} and \textit{f} into \(\mu\GammaCMC\) is shown in Figure 3.\footnote{The remaining code is available at the URL \url{http://www.upe.poli.br/~dsc/mgcmc}.}

\section{Example of Evaluation}

Appendix A presents the \(\mu\GammaCMC\) machine as a state transition machine. The kernel of \(\mu\GammaCMC\) is defined by a set of state transition laws. This section applies these laws to evaluate the \(\mu\GammaCMC\) code described in Figure 3.

During the evaluation, \(\mu\GammaCMC\) instructions are referred by their respective lines in the \(\mu\GammaCMC\) code presented in Figure 3. Due to the lack of space, only a few evaluation steps are presented\footnote{The complete sequence of execution may be obtained at the URL \url{http://www.upe.poli.br/~dsc/mgcmc}.}. The initial state of the \(\mu\GammaCMC\) machine for the code in Figure 3 is given by: \((26, 27, 28, T, C, FV, H, O)\). Starting at this point, the evaluation takes place by applying, step-by-step, the state transition law associated with the leftmost instruction in the sequence of code. For instance, the first instruction to be executed - \textit{ndol} - is located on the 26th line of the \(\mu\GammaCMC\) code. Therefore, the 2nd transition law (see Appendix A) must be applied (this is indicated by the symbol \((\downarrow)\)\footnote{the symbol \(\downarrow\) without index denotes the execution of a C instruction}). The next steps of the evaluation process are depicted in Table 8.

\section{Optimisations and Performance}

This section describes a set of optimisations introduced to \(\mu\GammaCMC\) yielding better performance figures. The execution time of the final version of \(\mu\GammaCMC\) is compared against the Glasgow Haskell compiler which uses the \textit{spineless tagless G-machine}\cite{18}.
Table 8: Example of evaluation

\[
\begin{align*}
\{ & 26.27.28, T, C, FV, H, O \} \downarrow_2 \\
\{ & 27.28, c_0, T, C, FV, H[c_0=(f.t.)], O \} \downarrow_3 \\
\{ & 4.5.6.7.8.9.10.11.12.13...28, c_0, T, C, FV, H[c_0=(f.t.)], O \} \downarrow_4 \\
\{ & 5.6.7.8.9.10.11.12.13...28, c_0, T, c_1, C, FV, H[c_0=(f.t.)], O \} \downarrow_5 \\
\{ & 6.7.8.9.10.11.12.13...28, c_0, T, c_2, c_1, C, FV, H[c_0=(f.t.)], O \} \downarrow_6 \\
\{ & 7.8.9.10.11.12...28, c_0, T, c_2, c_1, C, FV, H[c_0=(f.t.)], O \} \downarrow_7 \\
\{ & 8.9.10.11.12...28, c_0, T, c_3, c_2, C, FV, H[c_0=(f.t.)], O \} \downarrow_8 \\
\{ & 9.10.11.12...28, c_0, T, c_3, c_2, c_1, C, FV, H[c_0=(f.t.)], O \} \downarrow_9 \\
\{ & 10.11.12...28, c_0, T, c_3, c_2, c_1, C, FV, H[c_0=(f.t.)], O \} \downarrow_{10} \\
\end{align*}
\]

Table 9: Figures of performance for pseudoknot

<table>
<thead>
<tr>
<th>Compiler</th>
<th>time (sec.)</th>
<th>No garbage collection calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mu) CMC (original)</td>
<td>8.5</td>
<td>19</td>
</tr>
<tr>
<td>(\mu) CMC (otm. Section 8.1)</td>
<td>7.1</td>
<td>19</td>
</tr>
<tr>
<td>(\mu) CMC (otm. Section 8.2)</td>
<td>5.5</td>
<td>19</td>
</tr>
<tr>
<td>(\mu) CMC (otm. Section 8.3)</td>
<td>4.5</td>
<td>14</td>
</tr>
<tr>
<td>GHC 0.29</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>GHC 4.08</td>
<td>1.5</td>
<td></td>
</tr>
</tbody>
</table>

8.1 Avoiding unnecessary bindings

In the intermediate STG language, dictionaries containing methods of a type-class are represented as tuples. Whenever a given method is required, the complete set of methods in the dictionary is bound to variables of a type constructor. Then the method is selected by referencing the corresponding variable. This strategy may lead to inefficiencies since it often binds variables that are not going to be used. For example, consider the implementation of the list equality class in Haskell:

\[
\text{Eq.List dictEq} = \\
(\text{Eq.List.(==)} \text{dictEq}, \text{Eq.List.(/=)} \text{dictEq})
\]

The overloaded operation `==` in the context \([x] = [y]\) is transformed into `==` \((\text{Eq.List dictEq}) x y\). Here, equality operation is selected from the dictionary \(\text{Eq.List dictEq}\) using the selector verb `+(==)+`. The operation is then applied to operands \(x\) and \(y\). The STG code for this selection is:

```haskell
case (\text{Eq.List dictEq}) of 
    \{ \text{Tup2} [eq_fn, diff_fn] \mapsto \text{eq_fn} x y \}
```

Only the \(\text{eq_fn}\) function is required in the right-hand side expression, but both \(\text{eq_fn}\) and \(\text{diff_fn}\) are bound to the respective operations in the list equality class dictionary.

The extra cost to bind unused variables to the respective parts of the dictionary may be avoided by analysing the right-hand expression. \(\mu\) CMC identifies those variables which are used in the scope of the right-hand side, and only binds those variables that are used. In the example above, the \(\text{eq_fn}\) is bound to the first element of the tuple, but won’t bind \(\text{diff_fn}\)

This optimisation can always be applied whenever a given pattern-matching binding is not used on its the right-hand side expression, therefore it is not restricted to dictionaries or tuples.

8.2 Automatic generation of garbage collection code

The \(\mu\) CMC machine implements the copy algorithm by Fenichel and Yochelson [5]. The implementation considers basic type cells (integer, floating-point, character, etc.), type constructor cells and closure cells. The two latter are structured cells containing subcells. A generic recursive routine was used to copy different type of cells. Such routines test the cell type in order to apply the appropriate copy procedure. In the case of structured cells, a recursive call to the copy routine is performed for each subcell bound to them.

This strategy was adopted because it is impossible to predict the complete set of different cells which will be actually required during program execution.

Such a homogeneous treatment is a source of inefficiency. In order to avoid additional costs for recursive calls and tag testing one must provide specific routines for different types of cells. The information provided by the type-checker was used to generate specific copy routines for each type of cell that may be created in the heap. Such specific copy routines avoid recursive calls and type tests performed by the previous generic routine during the copying of structured cells.

8.3 Simplifying structured cells

The automatic generation of specific copy routines brought the opportunity for a new optimisation, which aims to decrease the demand for heap space by reducing the size of structured cells.

In the original \(\mu\) CMC abstract machine structured cells contain tags to identify types of subcells. These tags were only checked in the generic copy routine to decide which copy procedure should be applied for each subcell. As the subcell checking was eliminated in the new copy routines, subcells’ tags may be removed from structured cells. This simplification in the structured cell organization decreases the demand for heap space and, consequently, reduces the number of calls to the garbage collector.
8.4 Performance Analysis

A floating-point intensive application taken from molecular biology named pseudoknot is used in the performance analysis. Over 25 implementations of different functional languages were benchmarked using pseudoknot[6]. The performance figures for the initial version of μΓCMC along with results after each optimisation are presented on Table 9. The performance is measured in terms of execution time - given in seconds - and heap space demand - given in numbers of garbage collection calls.

The measurements were taken on a machine with a Pentium II 300Mhz processor, 64MB of RAM, running Linux (Debian 2.1). For GHC versions 0.29 and 4.08 optimisation -O2 was used. To compile μΓCMC the option gcc -O3 was used. A heap of 30MB was adopted.

On Table 9, the final version of μΓCMC compiler is also compared against GHC versions 0.29 and 4.08. The comparison with GHC version 0.29 is the fairest, since in this case GHC and μΓCMC share the same front-end, which is from GHC 0.29 itself. Results demonstrate that GHC 0.29 is twice as fast as the μΓCMC compiler. One must emphasize that the version of GHC used in the test exploits the size of the registers of the target machine and includes assembly routines in its runtime system. On the other hand, μΓCMC uses ordinary C variables instead of registers. In addition to that, the runtime system of μΓCMC is completely implemented in C. Moreover, μΓCMC consists of a six man-month work and there is much room for new optimisations. Taking into account these observations, one may consider the performance of μΓCMC rather satisfactory.

9. CONCLUSIONS

In this paper, a new back-end for GHC based on the categorical multi-combinators was presented. The abstract machine for graph reduction named μΓCMC implements this new back-end. It has been shown how to express the rewriting rules of the categorical multi-combinators as state transition rules and, then, how such state transition rules can be defined in the evaluation model of the μΓCMC machine. In this new machine, the environment is stored directly onto the reduction stack. In particular, the compilation of higher-order functions into this new scenario was detailed. In general, the gains obtained by eliminating the indication level to access the environment outweigh the need for special treatment for higher-order functions.

The GHC front-end was connected to the back-end through the intermediate language Shared-Term Graph (STG). The translation of a Haskell program into μΓCMC as described. The resulting code was then evaluated using the state transition laws which implement the kernel of μΓCMC.

μΓCMC is the first abstract machine based on the categorical multi-combinator to be used in the implementation of a powerful functional language like Haskell. With the implementation of this compiler it was possible to migrate from small and unrealistic benchmarks like fibonacci, sieve, prime, queens, etc. onto programs containing thousands of lines, created for real applications. Therefore, now it is possible to evaluate with more accuracy the qualities of the μΓCMC machine for the implementation of lazy functional languages.

The NoFib Haskell benchmark suite was used for the performance analysis. In particular the performance figures for the pseudoknot, a floating point intensive application taken from molecular biology, were presented. After optimisations the performance of μΓCMC is only twice as slow as that.
of the GHC compiler which adopts the spineless tagless G-machine [18]. GHC is probably the most efficient Haskell Compiler available. In addition, μTCMC does not exploit target machine features. It is completely implemented in the C language and has been easily migrated to distinct architectures (Sun SparcStation, IBM AIX/RS6000 and Intel Pentium). Moreover, our back-end is the result of a six months’ work and there is much room for new optimisations.

The extension of the IO library, which is required to support a larger number of Haskell applications is currently being worked on. This is a very time consuming task, since such a library must be carefully hand crafted. New optimisations to improve the performance of the new Haskell back-end based on μTCMC are being studied.

The abstract machine μTCMC, along with other informations, is available on the Web page: http://www.upe.polito.it/~dsc/mgcemc.

10. REFERENCES

APPENDIX

A. STATE TRAITION LAWS

\( \mu \Gamma CMC \) is now presented as a state transition machine. A state of \( \mu \Gamma CMC \) is a 6-tuple \((c, T, H, C, FV, O)\) in which each component is interpreted in the following way:

- \( c \): the \( \mu \Gamma CMC \) code sequence currently being executed.
- \( T \): the reduction stack. The top of \( T \) points at the part of the graph to be evaluated. \( T \) also holds references to environment cells (see Section 3).
- \( H \): the heap of cells where graphs are stored. \( H[d = e_1...e_n] \) means that there is in \( H \) a \( n \)-component cell named \( d \). The fields of \( d \) are filled with \( e_1...e_n \) in this order.
- \( C \): the closure stack. This stack also holds references to closures representing local functions defined in \texttt{let}-expressions.
- \( FV \): the free variable stack. The top of \( FV \) points at a cell with the fields are filled with free variables of the current redex.
- \( O \): the output stack. It can be thought of as a standard output terminal.

\( \mu \Gamma CMC \) is defined as a set of transition rules. The transition: \((c, T, H, C, FV, O) \Rightarrow (c', T', H', C', FV', O')\) is interpreted as when the machine reaches the state \((c, T, H, C, FV, O)\), it can get to state \((c', T', H', C', FV', O')\). A particular notation is adopted in the set of transition rules described below:

- \( \varepsilon \): the empty code;
- \( \phi_i \): an empty cell in the \( i \)-th field of a closure/type constructor cell.
- \( \texttt{cte} \): stands for a constant value;
- \( f \): invokes function \( f \);
- \( f_b \): the body of function \( f \);
- \( f_v \): are free variables;
- \( S \): is a suspension;
- \( P \): is a pointer to a static code block;
- \( p_0 \): partially applied parameters;
- \( p_i \): stands for the complementary parameters a partial application expects;
- \( p_n \): is a set of parameters applied to a saturated function;
- \( \texttt{id} \): is a type constructor identifier;
- \( b_i \): \( i \)-th element bound to a type constructor;
- \( n \): stands for a constant integer value representing the number of elements to be removed from a stack; and also the arity of functions and type constructors;
- \( i \): is the index of a field in a closure/type constructor cell;
- \( v \): is a generic value bound to a variable;
- \( . \): separates elements stored in the stack; \((a.b.c)\) indicates that \('a'\) was pushed in front of \( b \), and \( b \) was pushed in front of \( c \).

The complete set of state transition laws for the kernel of \( \mu \Gamma CMC \) is presented on Table 10.