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
alms@cin.ufpe.br

ABSTRACT

$\mu\Gamma$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 $\mu\Gamma$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 $\mu\Gamma$CMC abstract machine. A number of code optimisations are introduced to $\mu\Gamma$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].

$\mu\Gamma$CMC is a new abstract machine for the implementation of functional languages which inherited features of the ΓCMC machine [14, 15] and introduces new characteristics and optimisations to support an efficient implementation of Haskell. In $\mu\Gamma$CMC, C was used as a macro-assembler and the 'execution flow control' is transfered 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 $\mu\Gamma$CMC abstract machine. The experience with GM-C [17], CM-CM[16, 21] and ΓCMC[14, 15] was fundamental for the design, implementation and optimisation of $\mu\Gamma$CMC.

This work describes the new abstract machine $\mu\Gamma$CMC. Section 2 introduces the categorical multi-combinators. The foundation of the $\mu\Gamma$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 ΓCMC 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 $\mu\Gamma$CMC. The evaluation of the resulting $\mu\Gamma$CMC code is described in Section 7. Some optimisations and the performance evaluation are

SAC'04, March 14-17, Nicosia, Cyprus

Table 1: translating λ-expressions into categorical multi-combinators

(T .1)	$R^{[\,]}\lambda x_n \ldots \lambda x_0.a = L^n(R^{x_n \cdots x_0}a)$
(T .2)	$R^{x_n \cdots x_0}a \ldots b = R^{x_n \cdots x_0}a \ldots R^{x_n \cdots x_0}b$
(T .3)	$R^{x_n \cdots x_0}b = b$ *,if b is a constant*
(T .4)	$R^{x_n \cdots x_0}x_i = i$

Table 2: Compiling the λ-calculus into the categorical multi-combinators

λ-calculus code			Environment Construction			categorical multi-combinator		
S	=	λa.λb.λc.ac (bc)	S	$\rightarrow$	$R^{[\,]}$ [λa.λb.λc.ac (bc)]	S	$\rightarrow$	L^2 (2 0 (1 0))
K	=	λk.λl.k	K	$\rightarrow$	$R^{[\,]}$ [λk.λl.k]	K	$\rightarrow$	L^1 (1)
I	=	λi.i	I	$\rightarrow$	$R^{[\,]}$ [λi.i]	I	$\rightarrow$	L^0 (0)
		SKKI	SKKI	$\rightarrow$	$R^{[\,]}$ [SKKI]	SKKI	$\rightarrow$	SKKI

Table 3: Categorical multi-combinator rewriting laws

(M1) $\langle n, (x_m, \cdots, x_1, x_0)\rangle$	$\Rightarrow$	x_n
(M2) $\langle x_0 x_1 x_2 \ldots x_n, y\rangle$	$\Rightarrow$	$\langle x_0, y\rangle \ldots \langle x_n, y\rangle$
(M3) $L^n(y)x_0x_1\cdots x_n x_{n_+1}\cdots x_m$	$\Rightarrow$	$\langle y, (x_0, \cdots, x_n)\rangle\ x_{n+1}\cdots x_m$
(M4) $\langle f\ ,\ y\rangle$	$\Rightarrow$	$\langle f_b\ ,\ y\rangle$

presented in Section 8. The conclusions and future work are discussed in Section 9.

2. CATEGORICAL MULTI-COMBINATORS

This section briefly introduces the categorical multi-combinators [21, 12], a rewriting system which provides the computational model of $\mu\Gamma$CMC.

Categorical combinators represent a formal system similar to combinatory logic. The original system was developed by Curien [3] inspired by the equivalence of the theories of typed λ-calculus and cartesian closed categories as shown by Lambek [11] and Scott [20]. One approach to the execution of categorical combinators which uses a stack machine is described in [2].

Lins developed a new system of categorical combinators to implement lazy functional languages efficiently, called categorical multi-combinators. A project with similar aims to the system of categorical multi−combinator is Hughes' system of supercombinators [8]. Both systems have the power to perform the equivalent of several β-reductions in a single rewriting step and in both of them an expression needs to have all its arguments present before evaluation. There are, however, differences between these two systems. Categorical multi-combinators work with a fixed set of combinators which permits hardware implementation. On the other hand, supercombinators are generated during the compilation process. The compilation algorithm for categorical multi-combinators is extremely simple and generates expressions with size linear to the source code. Supercombinators use a more complex compilation algorithm due to the necessity of detecting maximal free expressions in the code. The supercombinator translation of a program of size N has size $O(NlogN)$ in the worst case. Supecombinators are fully lazy. This means that any sub-expression will be reduced at most once. Categorical multi-combinators are not fully lazy. If in our code we have a shared occurrence of a partial application of a function. This sub-expression cannot be evaluated before being copied and therefore it may be evaluated more than once. This drawback is removed by using partial categorical multi-combinators [13].

2.1 Compiling the λ-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 λ-expressions into categorical multi-combinators (Table 1) is given by the function $R^{x_0\cdots x_j}$ where x_i is a variable, and the corresponding i its depth in the environment, i.e. the corresponding DeBruijn [4] number. Top level expressions are translated using an empty environment, denoted 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. 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 $\langle a, b\rangle$, 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 λ-calculus β-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 $\mu\Gamma$CMC 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

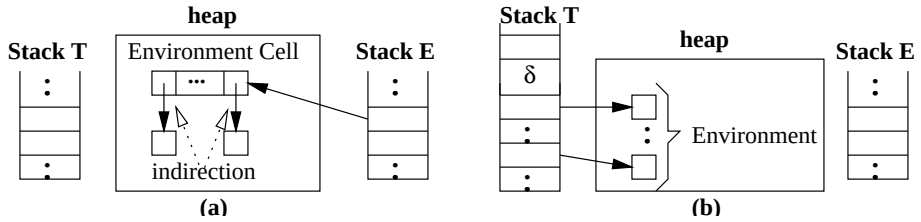

Figure 1: (a) ΓCMC: indirection to access the environment. (b) $\mu\Gamma$CMC: environment stored directly in the reduction stack

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

(1)	$\langle\langle n, e\rangle.c, H[e = (x_m, \cdots, x_0)]\rangle$	$\Rightarrow$	$\langle x_n.c, H[e = (x_m, \cdots, x_0)]\rangle$
(2)	$\langle\langle x_0 \cdots x_n, e\rangle.c, H[e = \cdots]\rangle$	$\Rightarrow$	$\langle\langle x_0, e\rangle \cdots \langle x_n, e\rangle.c, H[e = \cdots]\rangle$
(3)	$\langle\langle L^n(y), e_i\rangle x_0 \cdots x_n x_{n+1} \cdots x_m.c, H\rangle$	$\Rightarrow$	$\langle\langle y, e_j\rangle x_{n+1}, \cdots, x_m.c, H[e_j = \langle x_0 \cdots x_n, e_i\rangle]\rangle$
(4)	$\langle\langle f, e\rangle.c, H[e = \cdots]\rangle$	$\Rightarrow$	$\langle f_b, e\rangle.c, H[e = \cdots]\rangle$

Table 4: Example of evaluation

	$S\ K\ K\ I$
$\Rightarrow^{M4}$	$L^2(2\ 0\ (1\ 0))\ KKI$
$\Rightarrow^{M3}$	$\langle 2\ 0\ (1\ 0), KKI\rangle$
$\Rightarrow^{M2}$	$\langle 2, KKI\rangle\ \langle 0, KKI\rangle\ \langle 1\ 0, KKI\rangle$
$\Rightarrow^{M1}$	$K\ \langle 0, KKI\rangle\ \langle 1\ 0, KKI\rangle$
$\Rightarrow^{M4}$	$L^1(1)\ \langle 0, KKI\rangle\ \langle 1\ 0, KKI\rangle$
$\Rightarrow^{M3}$	$\langle 1, \langle 0, KKI\rangle\ \langle 1\ 0, KKI\rangle\ \rangle$
$\Rightarrow^{M1}$	$\langle 0, KKI\rangle$
$\Rightarrow^{M1}$	I

machines in which the execution flow control is transferred to C, as much as possible. A new abstract, $\mu\Gamma$CMC, has been created to support the implementation of programs written in the functional language Haskell.

3.1 Translating CM-C into $\mu\Gamma$CMC

For convenience, the categorical multi-combinator expression will be structured in two parts: the reduction stack T and the heap H, where evaluation environments are placed. The transition $\langle T, H\rangle \Rightarrow \langle T', H'\rangle$ must be interpreted as: 'whenever 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 of the categorical multi-combinators can be expressed as state transition rules (Table 5).

Instead of manipulating references to the environment directly, there is 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.

The ΓCMC abstract machine contains a stack, called E, the top of which points to the current environment. The use of this stack creates an extra indirection level to access the environment. One of the main innovations brought by the machine $\mu\Gamma$CMC was to avoid such indirection level by eliminating the environment cell. The environment is now stored directly onto the reduction stack T (see Figure 1).

In ΓCMC, an index corresponding to the DeBruijn number provides the depth of a value in the environment cell. The strategy of storing the environment in the reduction 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 δ, between the environment and the top of stack T. δ changes whenever the expression on top of T is computed. It is important to say that the control over δ 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 **SKKI** through the new machine $\mu\Gamma$CMC is presented on Table 7. The symbol 'ε' 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. $\mu\Gamma$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 $\mathbf{A} \rightarrow \mathbf{B}$. 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.

$\mu\Gamma$CMC uses a macro called *eval* to force the evaluation

Table 6: New rewriting laws of the categorical multi-combinators

(1)	$\langle_{(n+\delta)}.y_1.y_2.\ldots.y_\delta.e.c, H[e_{=(\langle x_0,e_0\rangle.\langle x_1,e_1\rangle\cdots\langle x_m,e_m\rangle)}]\rangle$	$\Rightarrow$	$\langle x_n.e_n.y_1.y_2\ldots y_\delta.e.c,\ H[e_{=(\langle x_0,e_0\rangle.\langle x_1,e_1\rangle.\ldots.\langle x_m,e_m\rangle)}]\rangle$
(2)	$\langle\langle L^n(y), e_i\rangle.x_0\cdots x_n.x_{n+1}\cdots x_m.e_i.c, H\rangle$	$\Rightarrow$	$\langle y.e_j.x_{n+1}\cdots x_m.e_i.c, H[e_{j=(\langle x_n,e_i\rangle\cdots\langle x_1,e_i\rangle.\langle x_0,e_i\rangle)}]\rangle$
(3)	$\langle\langle f, e\rangle.c, H\rangle$	$\Rightarrow$	$\langle\langle f_b, e\rangle.c, H\rangle$
(4)	$\langle n.e.c, H[e = (\langle x_0, e_0\rangle.\langle x_1, e_1\rangle\cdots\langle x_m, e_m\rangle)]\rangle$	$\Rightarrow$	$\langle x_n.e_n.c, H[e = (\langle x_0, e_0\rangle.\langle x_1, e_1\rangle\ldots\langle x_m\ e_m\rangle)]\rangle$
(5)	$\langle\langle f, e\rangle x_0\cdots x_n.e.c, H\rangle$	$\Rightarrow$	$\langle\langle L^n(y), e\rangle.x_0\cdots x_n.c, H\rangle$

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. *eval* starts an iterative process which finishes only when the element on top of stack **T** is completely evaluated. Notice that the semantics of *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. Such partial application cannot be evaluated in the next iteraction step of *eval* because the missing arguments are on unknown (statically unpredictable) positions of stack *T*. In this case, $\mu\Gamma$CMC uses the macro *eval'*, which avoids the iterative evaluation strategy adopted by *eval*. *eval'* performs a single evaluation step, removes the garbage left by the previous evaluated expression from stack **T**, and invokes *eval* to open the closure representing the partial function. Notice that it is safe to use *eval* after the first iteration, because at this point it is already known where the missing parameters are located.

5. REUSING A HASKELL FRONT-END

The Haskell functional language has several higher level features [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 $\mu\Gamma$CMC onto an existing Haskell front-end, namely the Glasgow Haskell Compiler (GHC) version 0.29 front-end [23]. The intermediate language *Shared Term Graph* (STG) [10] has been used as the interface between the GHC front-end and the $\mu\Gamma$CMC. 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 threfore 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 [19]. Reimplementing all these optimisations would also be a major programming effort.

6. COMPILING HASKELL INTO $\mu\Gamma$CMC

The compilation of Haskell programs into the kernel of the $\mu\Gamma$CMC abstract machine is presented in this section through an example. The compilation schemes to translate STG programs into $\mu\Gamma$CMC are available at the URL http://www.upe.poli.br/*sim*dsc/mgcmc.

Table 7: Example of evaluation using the $\mu\Gamma$CMC machine

$\langle S\ K\ K\ I, H\rangle$
$\Rightarrow^3\ \langle\langle L^2(2\ 0\ (1\ 0)), \varepsilon\rangle\ KKI, H\rangle$
$\Rightarrow^2\ \langle 2.0.(1\ 0).e_1, H[e_1 = I.K.K]\rangle$
$\Rightarrow^1\ \langle K.0.(1\ 0).e_1, H[e_1 = I.K.K]\rangle$
$\Rightarrow^5\ \langle\langle L^1(1), e_1\rangle.0.(1\ 0), H[e_1 = I.K.K]\rangle$
$\Rightarrow^2\ \langle 1.e_2, H[e_2 = \langle(1\ 0), e_1\rangle.\langle 0, e_1\rangle][e_1 = I.K.K]\rangle$
$\Rightarrow^4\ \langle 0.e_1, H[e_2 = \langle(1\ 0), e_1\rangle.\langle 0, e_1\rangle][e_1 = I.K.K]\rangle$
$\Rightarrow^4\ \langle I, H[e_2 = \langle(1\ 0), e_1\rangle.\langle 0, e_1\rangle][e_1 = I.K.K]\rangle$

Consider the following Haskell program:

```
twice f x = f (f x)
succ n = n +# 1#
main = print (I# (twice twice twice succ 3#))
```

Figure 2 presents the equivalent program written on the intermediate language STG. The translation of functions *main* and $f2$ into $\mu\Gamma$CMC is shown in Figure 3[1].

7. EXAMPLE OF EVALUATION

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

During the evaluation, $\mu\Gamma$CMC instructions are referred by their respective lines in the $\mu\Gamma$CMC code presented in Figure 3. Due to the lack of space, only a few evaluation steps are presented[2]. The initial state of the $\mu\Gamma$CMC machine for the code in Figure 3 is given by: $\langle_{26.27.28}, T, C, FV, H, O\rangle$. 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 - *nvol* - is located on the 26th line of the $\mu\Gamma$CMC code. Therefore, the 2nd transition law (see Appendix A) must be applied (this is indicated by the symbol $\Downarrow_2$)[3]. The next steps of the evaluation process are depicted on Table 8.

8. OPTIMISATIONS AND PERFORMANCE

This section describes a set of optimisations introduced to $\mu\Gamma$CMC yielding better performance figures. The execution time of the final version of $\mu\Gamma$CMC is compared against the Glasgow Haskell compiler which uses the *spineless tagless G-machine*[18].

[1]The remaining code is available at the URL http://www.upe.poli.br/∼dsc/mgcmc.
[2]The complete sequence of execution may be obtained at the URL http://www.upe.poli.br/∼dsc/mgcmc.
[3]the symbol $\Downarrow$ without index denotes the execution of a C instruction.

Table 8: Exemple of evaluation

$\langle_{26.27.28}, T, C, FV, H, O\rangle \Downarrow_2$
$\langle_{27.28}, e_0.T, C, FV, H[e_{0=(f.t.[])}], O\rangle \Downarrow_3$
$\langle_{4.5.6.7.8.9.10.11.12.13...23.28}, e_0.T, C, FV, H[e_{0=(f.t.[])}], O\rangle \Downarrow_{13}$
$\langle_{5.6.7.8.9.10.11.12.13...23.28}, e_0.T, e_1.C, FV, H[e_{0=(f.t.[])}][e_{1=P0}], O\rangle \Downarrow_{13}$
$\langle_{6.7.8.9.10.11.12.13...23.28}, e_0.T, e_2.e_1.C, FV, H[e_{0=(f.t.[])}][e_{1=P0}][e_{2=\langle P2,\emptyset\rangle}], O\rangle \Downarrow$
$\langle_{7.8.9.10.11.12.13...23.28}, e_0.T, e_2.e_1.C, FV, H[e_{0=(f.t.[])}][e_{1=P0}][e_{2=\langle P2,e_1\rangle}], O\rangle \Downarrow_{13}$
$\langle_{8.9.10.11.12.13...23.28}, e_0.T, e_3.e_2.e_1.C, FV, H[e_{0=(f.t.[])}][e_{1=P0}][e_{2=\langle P2,e_1\rangle}][e_{3=\langle P3,\emptyset\rangle}], O\rangle \Downarrow$
$\langle_{9.10.11.12.13...23.28}, e_0.T, e_3.e_2.e_1.C, FV, H[e_{0=(f.t.[])}][e_{1=P0}][e_{2=\langle P2,e_1\rangle}][e_{3=\langle P3,e_2\rangle}], O\rangle \Downarrow_{21}$
$\langle_{10.11.12.13...23.28}, e_0.T, e_3.e_2.e_1.C, e_3.FV, H[e_{0=(f.t.[])}][e_{1=P0}][e_{2=\langle P2,e_1\rangle}][e_{3=\langle P3,e_2\rangle}], O\rangle \Downarrow_3$
...

Table 9: Figures of performance for pseudoknot

Compiler	time (sec.)	No garbage collection calls
$\mu\Gamma$CMC (original)	8.5	19
$\mu\Gamma$CMC (otm. Section 8.1)	7.1	19
$\mu\Gamma$CMC (otm. Section 8.2)	5.5	19
$\mu\Gamma$CMC (otm. Section 8.3)	4.5	14
GHC 0.29	2.3	–
GHC 4.08	1.5	–

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 inneficiencies since it often binds variables that are not going to be used. For example, consider the implementation of the list equality class in Haskell:

```
Eq.List dictEq =
   (Eq.List.(==) dictEq, Eq.List.(/=) dictEq)
```

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

```
case (Eq.List dictEq) of
      { Tup2 [eq_fn , diff_fn] ->  eq_fn x y }
```

Only the `eq_fn` function is required in the right-hand side expression, but both `eq_fn` and `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. $\mu\Gamma$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, `eq_fn` is bound to the first element of the tuple, but won't bind `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 $\mu\Gamma$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 $\mu\Gamma$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.

```
succ{1}(Int#->Int#) =
  [] \r [n#{0}(Int#)]  plusInt# [n# 1#];

f1{2}((Int#->Int#)->Int#->Int#) =
  [] \r [f{0}(Int#->Int#) x#{0}(Int#)]
     case f x# of { v#{0}(Int#) -> f v#;};

f2{0}(Int) =
  [] \r []
     let { l1{1}(Int#->Int#) =
       [] \r [v1#{0}(Int#)] f1 f1 succ v1#;}
     in let { l2{1}(Int#->Int#) =
          [l1] \r [v2#{0}(Int#)] f1 l1 v2#;}
        in let { l3{1}(Int#->Int#) =
             [l2] \r [v3#{0}(Int#)] f1 l2 v3#;}
           in case l3 3# of {
                v4#{0}(Int#) ->
                  case l3 v4# of {
                    v5#{0}(Int#) ->  I# v5#;};};

main{0}([Response]->[Request]) =
  [] \u [] print_Int f2{0}(Int);
```

Figure 2: STG Code

```
01  void f2() {
02    int v4;
03    int v5;
04    MKC_comp(&P0, 0);
05    MKC_comp(&P2, 1);
06    (*top_C+1) → graph = (*(top_C-1));
07    MKC_comp(&P3, 1);
08    (*top_C+1) → graph = (*(top_C-1));
09    push_FV(*(top_C-0));
10    p_l3(3);
11    pop_FV;
12    v4 = (*top_T) → Ivalue;
13    pop_T;
14    push_FV(*(top_C-0));
15    p_l3(v4);
16    pop_FV;
17    v5 = (*top_T) → Ivalue;
18    pop_T;
19    MKT_cons(15, 2);
20    MKT_Ictel(v5, 1);
21    pop_C_n(1);
22    pop_C_n(1);
23    pop_C_n(1);
24    }
25
26  void main() { nvol;
27    f2();
28    printf("%d", (*top_T+1) → Ivalue); pop_T; pop_T; }
```

Figure 3: Compiling an STG program into $\mu\Gamma$CMC

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 $\mu\Gamma$CMC 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 $\mu\Gamma$CMC the option `gcc -O3` was used. A heap of 30MB was adopted.

On Table 9, the final version of $\mu\Gamma$CMC 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 $\mu\Gamma$CMC share the same front-end, which is from GHC 0.29 itself. Results demonstrate that GHC 0.29 is twice as fast as the $\mu\Gamma$CMC compiler. One must emphasize that the version of GHC used in the test exploits the set of registers of the target machine and includes *assembly* routines in its runtime system. On the other hand, $\mu\Gamma$CMC uses ordinary C variables instead of registers. In addition to that, the runtime system of $\mu\Gamma$CMC is completely implemented in C. Moreover, $\mu\Gamma$CMC 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 $\mu\Gamma$CMC 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 $\mu\Gamma$CMC 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 $\mu\Gamma$CMC 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 indirection level to access the environment outweight 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 $\mu\Gamma$CMC as described. The resulting code was then evaluated using the state transition laws which implement the kernel of $\mu\Gamma$CMC.

$\mu\Gamma$CMC 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 $\mu\Gamma$CMC 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 $\mu\Gamma$CMC is only twice as slow as that

Table 10: State transition laws

1. $\langle \varepsilon.c, T, C, FV, H, O\rangle \Rightarrow \langle c, T, C, FV, H, O\rangle$
2. $\langle nvol.c, T, C, FV, H, O\rangle \Rightarrow \langle c, e.T, H[e{=}false.true.[]], C, FV, O\rangle$
3. $\langle f.c, T, C, FV, H, O\rangle \Rightarrow \langle f_b.c, T, H, C, FV, O\rangle$
4. $\langle print.c, e.T, C, FV, H[e{=}cte], O\rangle \Rightarrow \langle c, T, C, FV, H[e{=}cte], cte.O\rangle$
5. $\langle eval.c, e.T, C, FV, H[e{=}cte], O\rangle \Rightarrow \langle c, e.T, C, FV, H[e{=}cte], O\rangle$
6. $\langle eval.c, e.T, C, FV, H[e{=}\langle S, fv, p_0\rangle][p_1], O\rangle \Rightarrow \langle S.c, T, C, FV, H[e{=}\langle S, fv, p_0\rangle][p_1], O\rangle$
7. $\langle MKT_{PC}(P).c, T, C, FV, H, O\rangle \Rightarrow \langle c, P.T, C, FV, H, O\rangle$
8. $\langle MKT_{cte}(cte).c, T, C, FV, H, O\rangle \Rightarrow \langle c, e.T, C, FV, H[e{=}cte], O\rangle$
9. $\langle MKC_{cte}(cte).c, T, C, FV, H, O\rangle \Rightarrow \langle c, T, e.C, FV, H[e{=}cte], O\rangle$
10. $\langle MKT_{cons}(id, n).c, T, C, FV, H, O\rangle \Rightarrow \langle c, e.T, C, FV, H[e{=}(id\ \emptyset_1..\emptyset_n)], O\rangle$
11. $\langle MKC_{cons}(id, n).c, T, C, FV, H, O\rangle \Rightarrow \langle c, T, e.C, FV, H[e{=}(id\ \emptyset_1..\emptyset_n)], O\rangle$
12. $\langle MKT_{comp}(S, n).c, T, C, FV, H, O\rangle \Rightarrow \langle c, e.T, C, FV, H[e{=}\langle S, \emptyset_1..\emptyset_n\rangle], O\rangle$
13. $\langle MKC_{comp}(S, n).c, T, C, FV, H, O\rangle \Rightarrow \langle c, T, e.C, FV, H[e{=}\langle S, \emptyset_1..\emptyset_n\rangle], O\rangle$
14. $\langle MKT_{ctel}(cte, i).c, e.T, C, FV, H[e{=}(..\emptyset_i..)], O\rangle \Rightarrow \langle c, e.T, C, FV, H[e{=}(..cte..)], O\rangle$
15. $\langle MKC_{ctel}(cte, i).c, T, e.C, FV, H[e{=}(..\emptyset_i..)], O\rangle \Rightarrow \langle c, T, e.C, FV, H[e{=}(..cte..)], O\rangle$
16. $\langle MKT_{consl}(id, i).c, e.T, C, FV, H[e{=}(..\emptyset_i..)], O\rangle \Rightarrow \langle c, e.T, C, FV, H[e{=}(..id..)], O\rangle$
17. $\langle MKC_{consl}(id, i).c, T, e.C, FV, H[e{=}(..\emptyset_i..)], O\rangle \Rightarrow \langle c, T, e.C, FV, H[e{=}(..id..)], O\rangle$
18. $\langle MKT_{comp\_PC}(P, i).c, e.T, C, FV, H[e{=}(..\emptyset_i..)], O\rangle \Rightarrow \langle c, e.T, C, FV, H[e{=}(..P..)], O\rangle$
19. $\langle MKC_{comp\_PC}(P, i).c, T, e.C, FV, H[e{=}(..\emptyset_i..)], O\rangle \Rightarrow \langle c, T, e.C, FV, H[e{=}(..P..)], O\rangle$
20. $\langle push_T(v).c, T, C, FV, H[v], O\rangle \Rightarrow \langle c, v.T, C, FV, H[v], O\rangle$
21. $\langle push_{FV}(v).c, T, C, FV, H[v], O\rangle \Rightarrow \langle c, T, C, v.FV, H[v], O\rangle$
22. $\langle pop_T.c, v.T, C, FV, H[v], O\rangle \Rightarrow \langle c, T, C, FV, H[v], O\rangle$
23. $\langle pop_{C\_n}(n).c, T, e.C, FV, H[e{=}v_1.v_2..v_n], O\rangle \Rightarrow \langle c, T, C, FV, H[v1.v_2..v_n], O\rangle$
24. $\langle pop_{FV}.c, d.T, C, v.FV, H[v], O\rangle \Rightarrow \langle c, T, C, FV, H[v], O\rangle$
25. $\langle pop_{TU\_n}(n).c, v.b_1.b_2..b_n.T, C, FV, H[vb_1..b_n], O\rangle \Rightarrow \langle c, v.T, C, FV, H[vb_1..b_n], O\rangle$

of the GHC compiler which adopts the *spineless tagless G-machine* [18]. GHC is probably the most efficient Haskell Compiler available. In addition, $\mu\Gamma$CMC 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 man-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 been 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 $\mu\Gamma$CMC are being studied.

The abstract machine $\mu\Gamma$CMC, along with other informations, is available on the Web page:
http://www.upe.poli.br/~dsc/mgcmc.

10. REFERENCES

[1] L. Cardelli. The Functional Abstract Machine. *Polimorphism*, 1(1), 1983.

[2] G. Cousineau, P. L. Curien, and M. Mauny. The Categorical Abstract Machine. *Functional Programming Languages and Computer Architecture, LNCS 201*, 1985.

[3] P. L. Curien. Combinateurs Categoriques, Algorithmes Sequentiels et Programmacion Applicative. *Thése de Doctorat d'Etat, Université Paris VII, LITP*, 1983.

[4] N. G. De Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Manipulation. *Indag. Math.*, (34):381–392, 1972.

[5] R. R. Fenichel and J. C. Yochelson. A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communication of ACM*, 11(12):611–612, November 1969.

[6] P. H. Hartel and et al. Benchmarking Implementation of Functional Languages with 'Pseudoknot', a Floating-Intensive Benchmark. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[7] P. Hudak, J. Peterson, and J. Fasel. A Gentle Introduction to Haskell. , 1997.

[8] R.J.M. Hughes. The design and implementation of programming languages. *Ph.D. Thesis, The Oxford University Comp. Lab.*, 1983.

[9] T. Johnsson. Compiling Lazy Functional Languages. *PhD Thesis, Chalmers Tekniska Högskola, Göteborg, Sweden*, 1987.

[10] S.Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[11] J. Lambek. From Lambda-Calculus to Cartesian Closed Categories. *Haskell B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980.

[12] R. D. Lins. Categorical Multi-Combinators. *Functional Programming Languages and Computer Architecture, Springer-Verlag, LNCS 274*, pages 60–79, 1987.

[13] R. D. Lins. Partial Categorical Multi-Combinators and Church-Rosser Theorems. *submitted for publication.*

[14] R. D. Lins and B. O. Lira. ΓCMC: A Novel Way of Implementing Functional Languages. *Journal of Programming Languages*, 1:19-39, Chapmann & Hall, January 1993.

[15] R.D. Lins, G. Neto, and R. MLima. Implementing and Optimising ΓCMC. *Euromicro'94, IEEE Computer Society Press*, pages 353–361, 1994.

[16] R. D.Lins and S.J. Thompson. Implementing SASL using Categorical Multi-Combinators. *Software — Practice and Experience*, 20(8):163–166, 1990.

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

[18] S. Peyton Jones and J. Salkild. The Spineless Tagless Gmachine. *In MacQueen, editor, Functional Programming and Computer Architecture. Addisson Wesley*, 1989.

[19] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computer Science, University of Glasgow, July 1995.

[20] D. Scott. Relating Theories of the Lambda-Calculus. *Haskell B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980.

[21] S. Thompson and R. D. Lins. The Categorical Multi-Combinator Machine: CMCM. *The Computer Journal*, 35(2):170–176, 1992.

[22] D. A. Turner. A New Implementation Technique for Aplicative Languages. *Software Practice and Experience*, 9:31–49, 1979.

[23] GHC Team. The Glasgow Haskell Compiler User's Guide, Version 4.01. 1998.

APPENDIX

A. STATE TRASITION LAWS

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

c	is the $\mu\Gamma$CMC code sequence currently being executed.
T	is 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	is 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	is the closure stack. This stack also holds references to closures representing local functions defined in *let*-expressions.
FV	is 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	is 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: $\langle c, T, H, C, FV, O\rangle \Rightarrow \langle c', T', H', C', FV', O'\rangle$ is interpreted as *when the machine reaches the state* $\langle c, T, H, C, FV, O\rangle$, *it can get to state* $\langle c', T', H', C', FV', O'\rangle$. A particular notation is adopted in the set of transition rules described below:

ε	is the empty code;
$\emptyset_i$	is an empty cell in the ith field of a closure/type constructor cell.
cte	stands for a constant value;
f	invokes function f;
f_b	is the body of function f;
fv	are free variables;
S	is a suspension;
P	is a pointer to a static code block;
p_0	are partially applied parameters;
p_1	stands for the complementary parameters a partial application expects;
p_n	is set of parameters applied to a saturated function;
id	is a type constructor identifier;
b_i	is ith 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.