GM-C
A Graph Categorical Multi-Combinator Machine

Martin A.Musicante & Rafael D.Lins *
Dept. de Informática - Universidade Federal de Pernambuco - Recife - Brazil
Computing Laboratory - The University of Kent - Canterbury - England.

Introduction

Semantic elegance, referential transparency, and expressive power are some of the important features which make lazy functional languages an interesting alternative to solve the problems of programming known as the software crisis [13]. The property of referential transparency brings an intuitive suitability for implementing functional languages in parallel machines. However, to make the use of functional languages viable today, we need to have fast implementations running on conventional von-Neumann computers.

A milestone in this quest for efficiency was Turner's graph reduction machine. Turner [12] showed how to compile functional languages into a set of combinators based on Combinatory Logic of Curry. His machine was based on graph interpretation. The performance of this machine was an order of magnitude faster than its ancestor, Landin's SECD machine [4]. It was still at least an order of magnitude slower than conventional imperative languages.

With the aim of providing an alternative to the efficient implementation of lazy functional languages we developed Categorical Multi-Combinators [7]. This system was the result of the evolution of several other rewriting systems [5, 6], which have similar aims, and are based on the original system of Categorical Combinators [1].

In Johnsson [2] an abstract machine for compiling lazy functional languages is presented. This machine puts together the best features of Turner's and Landin's machines with new code optimisation techniques. Johnsson's G-machine was a major achievement in the efficient implementation of lazy functional languages.

We present here GM-C, a Graph Categorical Multi-Combinator Machine, a new machine for the compilation of lazy functional languages. The performance figures presented in this work show that GM-C can be faster than a G-Machine, which was implemented with a much higher degree of sophistication [11].

1 Categorical Multi-Combinators

Categorical Multi-Combinators are a generalisation of Linear Categorical Combinators [6]. The code for a λ-expression compiled into Categorical Multi-Combinators is more compact

*Authors' permanent address: Dept.de Informática - U.F.PE. - 50.739 - Recife - PE - Brazil

than its Linear Categorical Combinators equivalent. Categorical Multi-Combinators reductions are of a coarser degree of computation than Linear Categorical Combinators. Each rewriting step of the Multi-Combinator code is equivalent to several rewritings of Linear Categorical Combinators. The core of the system of Categorical Multi-Combinators consists only of two rewriting laws with a very low pattern-matching complexity and avoids the generation of trivially reducible sub-expressions. In Categorical Multi-Combinators function application is denoted by juxtaposition. We take juxtaposition to be left-associative.

We assume that programs have already been λ-lifted [2] to remove non-local references from the bodies of functions. The compilation algorithm [7] for translating each function in the script into Categorical Multi-Combinators is:

(T .1) $[fun\ \underbrace{x_i \ldots x_j}_{n} \equiv a] = fun \mapsto L^{n-1}(R^{x_i,\ldots,x_j}_{n-1,\ldots,0}a)$

(T .2) $[a \ldots b] = [a] \ldots [b]$

(T .3) $[c] = c$, where c is a constant

(T .4) $R^{x_i \ldots x_j}_{n_i \ldots n_j}(a \ldots b) = (R^{x_i \ldots x_j}_{n_i \ldots n_j}a) \ldots (R^{x_i \ldots x_j}_{n_i \ldots n_j}b)$

(T .5) $R^{x_i \ldots x_j}_{n_i \ldots n_j}b = \begin{cases} b \text{ , } if\ \ b\ \ is\ \ a\ \ constant \\ n_k \text{ , } if\ \ b = x_k \end{cases}$

If whenever applying rule T.5 above a variable b can be associated with more than one x_k then one must choose the minimum correspondent n_k. In doing so we preserve locality of binding because a greater n_k means that a more internal binder is connected to the variable x_k. There follows an example of the translation of a function into Categorical Multi-Combinators using the algorithm above:

$$\begin{aligned}
[tw\ f\ x \equiv f\ (f\ x)] &\overset{\text{T.1}}{=} tw \mapsto L^1(R^{f,x}_{1,0}(f\ (f\ x))) \\
&\overset{\text{T.4}}{=} tw \mapsto L^1((R^{f,x}_{1,0}f)\ (R^{f,x}_{1,0}(f\ x))) \\
&\overset{\text{T.5}}{=} tw \mapsto L^1(1\ (R^{f,x}_{1,0}(f\ x))) \\
&\overset{\text{T.4}}{=} tw \mapsto L^1(1\ ((R^{f,x}_{1,0}f)\ (R^{f,x}_{1,0}x))) \\
&\overset{\text{T.5}}{=} tw \mapsto L^1(1\ (1\ (R^{f,x}_{1,0}x))) \\
&\overset{\text{T.5}}{=} tw \mapsto L^1(1\ (1\ 0))
\end{aligned}$$

The core of the Categorical Multi-Combinator machine enriched with arithmetic operations is expressed by the following rewriting laws,

(M.1) $\circ (x_0 x_1 x_2 \ldots x_n)\ (P\ y_m \ldots y_1 y_0) \Rightarrow (x'_0 x'_1 \ldots x'_n)$,

$$where \begin{cases} x'_i & = x_i\ if\ x_i\ is\ a\ constant\ or\ of\ type\ L^a(b) \\ & = y_k\ if\ x_i\ is\ a\ variable\ k \\ & = \circ\, x_i\ (P\ y_m \ldots y_1 y_0), otherwise \end{cases}$$

$$
\textbf{(M.2)}\ L^n(y)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \begin{cases} \Rightarrow y\ x_{n+1}\ldots x_z\ \textit{if}\ y\ \textit{is a constant} \\ \Rightarrow x_{n-y}x_{n+1}\ldots x_z\ \textit{if}\ y\ \textit{is a variable} \\ \Rightarrow (\circ\, y\ (P\ x_0\cdots x_n))x_{n+1}\cdots x_z, \quad \textit{otherwise} \end{cases}
$$

$$
(+)\ +xy \Rightarrow x+y
$$

$$
(=)\ \ Cond\ x\ m\ n \begin{cases} \Rightarrow m,\ \textit{if}\ x = \textit{True} \\ \Rightarrow n,\ \textit{otherwise} \end{cases}
$$

The + rule above, which stands for all arithmetic and binary boolean operators. It means evaluate the first argument, evaluate the second argument, and then add them together. In the case of = it means evaluate the condition (x) first, test the result of evaluation, and then branch.

Let us now present an example of the execution of a "program" using Categorical Multi-Combinators. The expression

$$tw\ \ id\ \mathbf{1}$$

with *tw* and *id* defined as, (we use boldface to represent constants),

$$
\begin{aligned} tw\ f\ x &\equiv f\ (f\ x) \\ id\ y &\equiv y \end{aligned}
$$

translates into Categorical Multi-Combinators using the compilation algorithm above as

$$L^1(1\ (1\ 0))\ L^0(0)\ \mathbf{1}$$

which using the laws above can be rewritten as,

$$
\begin{aligned}
&\overset{M.2}{\Rightarrow} \ \circ(1\ (1\ 0))\ (P\ L^0(0)\ \mathbf{1}) \\
&\overset{M.1}{\Rightarrow} \ L^0(0)\ (\circ(1\ 0)\ (P\ L^0(0)\ \mathbf{1}) \\
&\overset{M.1}{\Rightarrow} \ L^0(0)\ (L^0(0)\ \mathbf{1}) \\
&\overset{M.2}{\Rightarrow} \ L^0(0)\ \mathbf{1} \\
&\overset{M.2}{\Rightarrow} \ \mathbf{1}
\end{aligned}
$$

As one can observe in the sequence of reductions above the Categorical Multi-Combinator code suffers a metamorphosis under rewriting. The application of rule (M.2) changes the structure of the code and generates an "environment", in which to each variable there is associated a (local) value. This "evaluation environment" is distributed through the body of the multi-abstraction (L^n) and then variables fetch their value by successive application of rule (M.1).

2 GM-C

We will present GM-C as a state transition machine. The state of the GM-C Machine is a 6-uple

$$\langle C, B, T, H, O, E\rangle$$

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 in Appendix A.

B: The basic-value stack, used for the evaluation of arithmetic and logic expressions.

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.

O: The output.

E: The environment stack. Its top contains a reference to the current environment.

GM-C is defined as a set of transition rules. The transition

$$\langle C, B, T, H, O, E \rangle \Rightarrow \langle C', B', T', H', O', E' \rangle$$

must be interpreted as: "when the machine arrives at state $\langle C, B, T, H, O, E \rangle$, it can get to state $\langle C', B', T', H', O', E' \rangle$".

The full set of transition laws is presented in Appendix B. In [9] we formalise the notion that GM-C "implements" Categorical Multi-Combinators.

2.1 Compiling into GM-C Code

The compilation of Categorical Multi-Combinator expressions into GM-C code is performed by using three different schemes:

Scheme $\mathcal{E}$: This compilation scheme drives the evaluation process, besides construction the graph of expressions.
It is the first scheme called for the compilation of an expression to be evaluated. If we have a program e its compilation will be performed by

$$\mathcal{E}[e]; \mathsf{print}$$

Scheme $\mathcal{T}$: constructs a cell containing a representation of a term, pushing its address onto the stack **T**.

Scheme $\mathcal{G}$: is called by the other schemes to fill in the fields of the cells.

The complete set of compilation rules for the schemes above can be found in Appendix A.

2.1.1 Example of Compilation

The identity function

$$id\ y \equiv y$$

translates into Categorical Multi-Combinators as

$$id \mapsto L^0(0)$$

The expression,

$$id\ \mathbf{7}$$

compiles into GM-C code as,

$$
\begin{aligned}
\mathcal{E}[L^0(0)\ \mathbf{7}];\mathsf{print} \stackrel{xi}{=}\ & \mathsf{MKcell}(1);\mathcal{G}[\mathbf{7}]\mathbf{0};\\
& \mathsf{TransTE};\mathcal{E}[0];\\
& \mathsf{popENV};\mathsf{print};\\
\stackrel{vii}{=}\ & \mathsf{MKcell}(1);\mathsf{MKcteN}(\mathbf{7})\mathbf{0};\\
& \mathsf{TransTE};\mathcal{E}[0];\\
& \mathsf{popENV};\mathsf{print};\\
\stackrel{xiii}{=}\ & \mathsf{MKcell}(1);\mathsf{MKcteN}(\mathbf{7})\mathbf{0};\\
& \mathsf{TransTE};\mathsf{MKcell}(1);\mathcal{G}[0]0;\\
& \mathsf{popENV};\mathsf{print};\\
\stackrel{viii}{=}\ & \mathsf{MKcell}(1);\mathsf{MKcteN}(\mathbf{7})\mathbf{0};\\
& \mathsf{TransTE};\mathsf{MKcell}(1);\mathsf{MKvar}(0)0;\\
& \mathsf{popENV};\mathsf{print};
\end{aligned}
$$

2.2 Running GM-C Code

Now let us use the expression above as an example of execution of GM-C. The initial state of the machine is,

$$
\langle\ \begin{array}{l}\mathsf{MKcell}(1);\mathsf{MKcteN}(\mathbf{7})\mathbf{0};\\ \mathsf{TransTE};\mathsf{MKcell}(1);\mathsf{MKvar}(0)0;\\ \mathsf{popENV};\mathsf{print};\end{array}, B[\ ], T[\ ], H[\ ], O[\ ], E[\ ]\rangle
$$

which yields

$$
\begin{aligned}
&\stackrel{1}{\Rightarrow} \langle\ \begin{array}{l}\mathsf{MKcteN}(\mathbf{7})\mathbf{0};\\ \mathsf{TransTE};\mathsf{MKcell}(1);\mathsf{MKvar}(0)0;\\ \mathsf{popENV};\mathsf{print};\end{array}, B[\ ], d_0, H[d_0 = u_0], O[\ ], E[\ ]\rangle\\
&\stackrel{2}{\Rightarrow} \langle\ \begin{array}{l}\mathsf{TransTE};\mathsf{MKcell}(1);\mathsf{MKvar}(0)0;\\ \mathsf{popENV};\mathsf{print};\end{array}, B[\ ], d_0, H[d_0 = \mathbf{7}], O[\ ], E[\ ]\rangle\\
&\stackrel{34}{\Rightarrow} \langle\ \begin{array}{l}\mathsf{MKcell}(1);\mathsf{MKvar}(0)0;\\ \mathsf{popENV};\mathsf{print};\end{array}, B[\ ], T[\ ], H[d_0 = \mathbf{7}], O[\ ], d_0\rangle\\
&\stackrel{1}{\Rightarrow} \langle \mathsf{MKvar}(0)0;\mathsf{popENV};\mathsf{print};, B[\ ], d_1, H[d_1 = u_0]\ [d_0 = \mathbf{7}], O[\ ], d_0\rangle\\
&\stackrel{3}{\Rightarrow} \langle \mathsf{popENV};\mathsf{print};, B[\ ], d_1, H[d_1 = \mathbf{7}]\ [d_0 = \mathbf{7}], O[\ ], d_0\rangle\\
&\stackrel{12}{\Rightarrow} \langle \mathsf{print};, B[\ ], d_1, H[d_1 = \mathbf{7}]\ [d_0 = \mathbf{7}], O[\ ], E[\ ]\rangle\\
&\stackrel{31}{\Rightarrow} \langle [\ ], B[\ ], d_1, H[d_1 = \mathbf{7}]\ [d_0 = \mathbf{7}], \mathbf{7}, E[\ ]\rangle
\end{aligned}
$$

3 GM-C & the G-Machine

There are both differences and similarities between the GM-C machine and the G-Machine. The G-machine places arguments to a function on the evaluation stack. Variables are represented by offsets in relation to the top of this stack. We can say that the environment to a given function is simulated by the evaluation stack. GM-C follows the tradition of Categorical Multi-Combinators and works with environments in an explicit manner.

The G-machine uses a dump for saving and restoring the machine state. GM-C uses closures for this purpose.

The philosophy of the G-machine is to evaluate expressions eagerly, whenever safe, to avoid generating graphs. GM-C may in some cases use closures to avoid evaluating expressions. This is a compromise with evaluating eagerly something which may not be needed for computation, as in the G-machine, and generating the whole graph.

GM-C and the G-machine also differs in the way they represent graphs. Our machine uses an unboxed variable-length cell representation. The G-machine adopts a fully-boxed representation for cells. These cells are of fixed size. The G-machine uses variable-length cells only as an optimisation to represent a sequence of nested applications. These are called *vector apply nodes*. Vector apply nodes are used only to represent the application of a function to a number of arguments equal to its arity.

Amongst the similarities between the two machines there is the way predefined operators are implemented.

4 An Implementation of GM-C

GM-C was implemented in C [3], as follows:

C is loaded with the GM-C code, which is obtained from the application of the translation laws in Appendix A to the original expression.
Each of the GM-C machine instruction was implemented as a macro written in C.

B is implemented as an integer stack. In future implementations the system stack and stack pointers may be used.

T is a stack of references to the heap.

H is a large heap area divided into two equally sized halves used one at a time.

O is the standard output.

E was implemented together with **T**.

The process of evaluating an expression is driven by the printing routine. We used a copying garbage collector[10].

4.1 Performance

We present here four simple programs which make extensive use of the most important features of lazy functional languages, such as recursion, higher-order functions, and lazy evaluation. They are:

Fibonacci: the Fibonacci number of 20.

Sieve: Erathosthenes' sieve to find all prime numbers up to 300.

InsOrd: sorting by insertion of a list of 100 numbers generated at random.

SimLog: a program which takes a list of 100 random numbers and produces 100 random boolean values.

These programs were used to compare the performance of GM-C with the G-machine described in [2, 11], and also with Simon Croft's implementation of Turner's KRC [13] and ML [8], a strict functional language. We also provide performance figures for two of chosen benchmark programs in C. These programs were implemented in a functional style. A different implementation of these algorithms in C may bring a better performance.

InsOrd and **SimLog** make use of lazy evaluation, for this reason we do not produce their performance figures in ML and C.

For the sake of simplicity and portability our implementations use C under VMS. All data presented here was obtained from a Vax 750. For each test program at least five time measures were taken. We present the worst ones.

The heap is of 235,200 bytes. The number of cells we present below correspond to the number of units of information used. A data structure with two fields, for instance, counts as two cells. During garbage collection all cells copied count as a new cell. The time figures below correspond to c.p.u. user time in seconds.

Program	*Fibonacci*		*Sieve*		*InsOrd*		*SimLog*	
Implem.	*time*	*cells*	*time*	*cells*	*time*	*cells*	*time*	*cells*
KRC	65.84	—	30.40	—	30.21	—	4.11	—
G	2.88	109,775	1.18	17,025	1.42	11,614	0.73	4,067
GM-C	2.19	21,892	2.89	18,011	2.85	20,990	1.05	4,003
ML	8.58	—	6.68	—	—	—	—	—
C	0.83	—	1.28	—	—	—	—	—

In this implementation lists were introduced in a similar way to the G-machine [2, 10, 11]. However, we think we can optimise the way GM-C works with lists. For this reason we have omitted the part which deals with data structures from the description of GM-C. The performance figures obtained for the programs which use lists are close to the ones for the non-optimised G-machine [11].

In the case of Fibonacci GM-C presented a performance of around 25% better than the G-machine. We would like to stress that many important optimisations for the G-machine, as described in [2, 10, 11], are still to be tested in GM-C.

5 Conclusions

GM-C is a simple and efficient machine for implementing lazy functional languages. The performance figures presented here show that GM-C can, in some cases, be 25% faster than a G-machine implemented with a much higher degree of sophistication. In the authors' opinion, GM-C still gives room for several optimisations, which we hope will bring great improvements on its time and space performance.

References

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

2. T.Johnsson, Compiling Lazy Functional Languages, Ph.D. Thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, January 1987.

3. B.W.Kernighan & D.M.Ritchie, The C Programming Language, Prentice Hall, Englewood Cliffs, N.J., 1978.

4. P.J.Landin, The Mechanical Evaluation of Expressions, Comput.J.6, (1963-4).

5. R.D.Lins, A New Formula for The Execution of Categorical Combinators, Proceedings of 8th. International Conference on Automated Deduction, Oxford, England, July 1986, LNCS 230, p 89 - 98, Springer Verlag.

6. R.D.Lins, On the Efficiency of Categorical Combinators as a Rewriting System, Software Practice & Experience, Vol 17(8), 547-559, (August 1987).

7. R.D.Lins, Categorical Multi-Combinators, Proceedings of Third International Conference on Functional Programming and Computer Architecture, Portland, USA, September 1987, LNCS 274, p 60-79, Springer Verlag.

8. R.Milner, Standard ML proposal, Polimorphism: The ML/LCF/Hope Newsletter, 1(3), January 1984.

9. M.A.Musicante & R.D.Lins, On optimising and proving the correctness of GC-MC, in preparation.

10. S.Peyton-Jones, The Implementation of Functional Languages, Prentice Hall, (1987).

11. P.G.Soares & R.D.Lins, Implementing the G-Machine, UKC Computing Laboratory Report N.66, The University of Kent at Canterbury, August 1989. (submitted for publication in Software Practice & Experience).

12. D.A.Turner, A New Implementation Technique for Applicative Languages, Software Practice and Experience, Vol 9, 31-49 (1979).

13. D.A.Turner, Recursion Equations as a Programming Language, Functional Programming and its Applications, J.Darlington, P.Henderson, and D.A.Turner eds., Cambridge University Press (1982).