# Some Performance Figures for the G-Machine and its Optimisations

Rafael D.Lins and Patricia G.Soares[a]

[a]Departamento de Informática, Universidade Federal de Pernambuco,
Cidade Universitária, 50.739, Recife, PE, Brazil

Computing Laboratory, The University of Kent at Canterbury,
Canterbury, CT2 7NF, Kent, England

Johnsson's G-Machine is a major achievement in the efficient implementation of lazy functional languages. In this paper we provide figures of time and space performance of the original G-Machine and each optimisation step. The figures presented not only help to understand the G-Machine, but can also serve as a basis for choosing which optimisations to use in a different machine for the implementation of lazy functional languages.

## 1. Introduction

The G-Machine [1] was developed by Johnsson and Augustsson, in the *Chalmers Institute of Technology*, Göteborg, Sweden, with the aim of providing efficient implementation of lazy functional languages in von-Neumann machines. The code generated by the G-Machine when executed produces time and space performance comparable with conventional implementations of imperative languages, such as C. This improvement has been attributed to many different aspects of the machine in isolation. But not much is said about how each of these aspects affects the behaviour of the machine as a whole. The only data available is the execution time of 3 benchmark programs in LML (page C-20 of reference [1]).

In our opinion, some quantification is extremely important to develop a "feel" for the gains obtained in each of the optimisation steps of the G-Machine presented by Johnsson [1] and Peyton-Jones [6]. In this work we present time and space performance figures for the G-machine and each optimisation. These optimisations are not applicable only to the G-Machine. They can be adapted to other lazy functional machines. The figures presented here can also serve as a basis for choosing which optimisations to use in a different machine. This knowledge was used with success in optimising GM-C [5], CM-CM [7] and ΓCMC [3], abstract machines for the implementation of lazy functional languages based on Categorical Multi-Combinators [2, 4].

## 2. The G-Machine

The original G-Machine is very simple. We can say that this machine works as an interpreter in which the original graph is replaced by code. This code when executed generates a graph to be interpreted.

Suppose, for example, that we want to evaluate the following expression, which returns the list of the squares of each Natural number.

$$\begin{array}{l} list\ 0,\ \text{where} \\ list\ n = square\ n : list\ (suc\ n) \end{array}$$

In the definition above *square* and *suc* are predefined functions as follows,

$$\begin{array}{rcl} square\, x & = & x \times x \\ suc\, x & = & x + 1 \end{array}$$

The expression, $list\ 0$, will be represented in the G-Machine as in figure (a) below. Using the

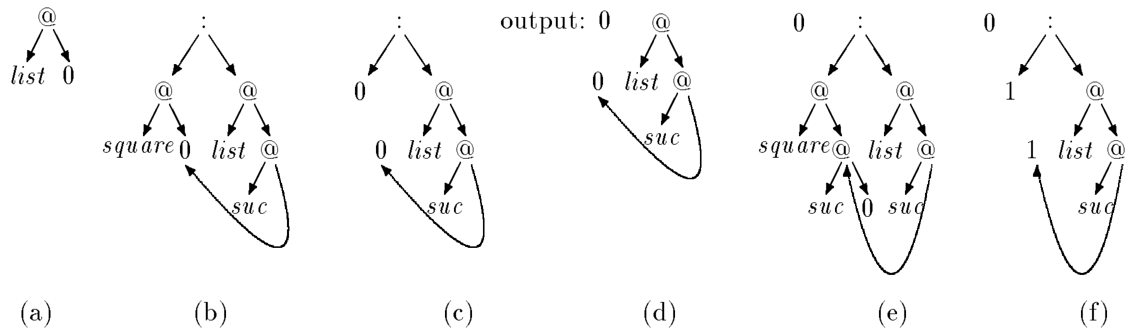


Figure 1: Example of the Evaluation Mechanism

definition of the function *list* as a rewriting law, this expression is reduced to figure (b). The integer 0 is the shared argument to *square* and *suc*. The resulting list expression is in canonical form, but neither its head nor its tail are. After reducing a graph to its canonical form, the next step is printing the result. Printing a list means to print its head and then to print its tail. As only ground type expressions can be printed, the machine will first reduce the head of the list, print it and then reduce its tail and print it. Using the definition of *square* as a rewriting law, the head of the list is then rewritten to the square of the integer 0, that is 0. This rewriting step can be seen in figure (c).

Now, as the head of list is in canonical form, it can be printed and removed from the graph (figure (d)). The evaluation of the tail continues in a similar fashion(figure (e)). Computation does not terminate. Now, the argument to *square* and *suc* is the expression *suc* 0. Again, the machine will try to print it. The definition of *square* reduces the expression to the square of the integer denoted by the expression *suc* 0. First we reduce the graph representing *suc* 0 to its canonical form, the integer 1. As *suc* 0 is shared, all expressions that reference it will benefit from the reduction accomplished. Then, *square* is applied on the resulting expression, yielding the integer 1. The graph after these rewritings is as in figure (f). Again, as the head is now in canonical form, it can be printed and removed from the graph. Reduction continues in the same manner.

The graph transformations above illustrate the behaviour of original G-Machine, in which expressions are mostly interpreted. Optimisations replace interpretation by compilation.

### 2.1. Compilation Schemes

Our first implementation of the G-Machine uses the following compilation schemes (pg C-8 of [1]):

$\mathcal{F}[\![f x_1 \cdots x_m = e]\!]$ produces code to reduce the graph of function $f$ applied to arguments $x_1$ to $x_m$.

$\mathcal{E}[\![e]\!]$ generates code that evaluates $e$. It is used if we know statically that $e$ is needed.

$\mathcal{C}[\![e]\!]$ produces code that constructs the graph of $e$, if scheme $\mathcal{E}$ is not applicable.

$\mathcal{B}[\![e]\!]$ compiles arithmetic or logical expressions.

Using these schemes the code for *square* is:
*square* $\rightarrow$ PUSH(0); EVAL; GET;
PUSH(0); EVAL; GET; MUL; MKINT

### 2.2. The Abstract Machine

The G-code generated is executed in the abstract G-Machine [1]. A state in this machine is described as a 7-tuple $\langle O, C, S, V, G, E, D\rangle$, where

- $O$: is the output ever produced, i.e. a sequence of basic values (integers and booleans).
- $C$: is the the G-code sequence being executed
- $S$: is the stack of pointers to the graph stored in the heap
- $V$: is the stack of values for operands of arithmetic and logical operations
- $G$: is the the graph formed by fully-boxed cells stored in an area of memory called heap.

$E$: is the stores the name, arity and code for each function defined in the script.

$D$: is a dump used to save the current state during recursive calls to EVAL.

## 3. Performance

In order to increase the efficiency of the original G-Machine, Johnsson optimises the compiled code to avoid generating graphs as much as possible. This reduces the interpretative part of the execution of programs. Most of these optimizations are suggested and described by Johnsson [1] and Peyton-Jones [6].
The benchmark programs are:

**Fibonacci:** the Fibonacci number of 20.

**Sieve:** generates a list of prime numbers smaller than 300 by using Erathostenes' sieve.

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

**Simlog:** takes a list of 100 random numbers and produces 100 boolean values.

**TwSuc:** maps (twice twice twice successor) onto a list of 600 numbers.

The original implementation of the G-Machine by Johnsson [6] is made in Vax 780 Assembly Language. For the sake of simplicity and portability our implementations of the G-Machine use C as a macro assembler. The data presented was from a VAX 750 with arithmetic co-processor. A copying algorithm for garbage collection was used, each heap of size 43,750 cells. The table below summarizes the results for the benchmark programs.

**KRC** corresponds to Turner's KRC implemented by Simon Croft as an interpreter written in C.

**00** Presents the performance of the G-Machine without optimisations.

**01** Introduces non-volatile cells, which are not reachable by the garbage collector.

**02** Optimises pre-defined functions (i.e. arithmetic and logical operations) with the optimisations described in lines **08**, **10** and **11** for user-defined functions.

**03** Saves function calls by storing return addresses on the $V$ stack and defines UNWIND as a macro.

**04** Uses schemes $\mathcal{R}$, $\mathcal{RS}$ & $\mathcal{ES}$ as described in chapter 20 of reference [6].

**05** Optimizes scheme RS as described on pages 367-377 of [6]. This compilation scheme avoids the generation of vertebrae which will become garbage soon after its generation.

**06** Optimization of scheme $\mathcal{ES}$ as presented on pages 377 and 378 of [6].

**07** Performs symbolic evaluation of expressions. In most cases, this optimization transforms call-by-need into call-by-value without losing laziness, because this is done locally within a function.

**08** Presents the result of the peephole optimisation for instructions UPDATE, SLIDE, and POP.

**09** The numbers and characters found in user defined functions are stored in the non-volatile area, avoiding to copy them every time they are needed, as in the case of recursive functions.

**10** Tests memory availability per function instead of per cell needed.

**11** Uses simulation stacks to avoid redundant PUSHes and POPs (pages D-22/33 of [1]).

**12** The printing procedure is made particular to the type of the output (monomorphic).

**13** Nested applications are flatened whenever all arguments to a function are present at compile-time.

**14** The stack and heap pointers were represented as pointers instead of integers indexing an array.

**ML** is the Edinburgh implementation of Standard ML by FAM version 3.3.

**C** corresponds to programs implemented in C in a functional style.

| Program | Fibonacci | | Sieve | | InsOrd | | SimLog | | TwSuc | |
|---|---|---|---|---|---|---|---|---|---|---|
| Implemt. | time | cells | time | cells | time | cells | time | cells | time | cells |
| KRC | 65.84 | ———- | 30.40 | ———- | 30.21 | ———- | 4.11 | ———- | 18.07 | ———- |
| 00 | 21.97 | 417,720 | 14.79 | 96,275 | 4.71 | 57,797 | 1.39 | 11,962 | 14.40 | 133,301 |
| 01 | 20.53 | 329,411 | 14.13 | 70,149 | 4.51 | 41,725 | 1.30 | 9,148 | 14.30 | 121,447 |
| 02 | 19.45 | 285,473 | 13.57 | 68,851 | 4.27 | 40,721 | 1.26 | 8,548 | 14.29 | 121,447 |
| 03 | 16.58 | 285,473 | 10.82 | 68,851 | 3.70 | 40,271 | 1.14 | 8,548 | 14.20 | 121,447 |
| 04 | 14.73 | 285,473 | 9.17 | 68,851 | 3.44 | 40,271 | 1.06 | 8,548 | 10.14 | 119,857 |
| 05 | 13.91 | 285,473 | 9.05 | 64,290 | 3.41 | 39,864 | 1.04 | 7,645 | 8.91 | 84.654 |
| 06 | 10.22 | 285,473 | 8.21 | 57,902 | 3.00 | 39,864 | 0.87 | 7,189 | 8.28 | 84,654 |
| 07 | 5.45 | 197,627 | 2.34 | 33,548 | 2.02 | 25,673 | 0.84 | 6,781 | 7.71 | 78,064 |
| 08 | 4.81 | 153,663 | 1.85 | 21,859 | 1.86 | 17,678 | 0.80 | 5,981 | 7.43 | 55,434 |
| 09 | 3.92 | 109,775 | 1.78 | 21,259 | 1.83 | 16,866 | 0.75 | 4,869 | 7.31 | 54,144 |
| 10 | 3.89 | 109,775 | 1.82 | 21,259 | 1.84 | 16,866 | 0.77 | 4,869 | 7.20 | 54,144 |
| 11 | 3.01 | 109,775 | 1.48 | 21,259 | 1.54 | 16,866 | 0.71 | 4,869 | 6.70 | 54,144 |
| 12 | 2.88 | 109,775 | 1.44 | 21,259 | 1.55 | 16,866 | 0.71 | 4,869 | 6.58 | 54,144 |
| 13 | 2.88 | 109,775 | 1.18 | 17,025 | 1.42 | 11,614 | 0.73 | 4,067 | 6.70 | 54,144 |
| 14 | 1.74 | 109,775 | 1.03 | 17,025 | 1.16 | 11,614 | 0.65 | 4,067 | 5.73 | 54,144 |
| ML | 8.58 | ———- | 6.68 | ———- | —— | ———- | —— | ———- | 16.99 | ———- |
| C | 0.83 | ———- | 1.28 | ———- | —— | ———- | —— | ———- | —— | ———- |

## 4. Conclusions

In this paper, we show how the G-Machine provides fast implementations of functional languages, comparable in performance to imperative ones. The simplicity and modularity of the original G-Machine are the key for allowing simple optimisations, which in some cases, increased the performance of an order of magnitude.

The figures for the optimisations steps presented in this paper serve to give a better understanding of the G-Machine, quantify gains in each of them and allow implementors of lazy functional languages to make a choice of which optimisations to use in their own implementation. This knowledge was used with success in optimising Γ-CMC [3], a machine that produced performance figures ranging from as good to several times faster than Johnsson's implementation of the LML compiler based on the G-Machine.


## Acknowledgements

We express gratitude for several discussions with Danilo Florissi and Martin Musicante.

Research reported herein has been sponsored jointly by The British Council, CNPq (Brazil) grants 40.9110/88-4 and 80.4520/88-7, and CAPES (Brazil) grant 2487/91-08.


## References


[1] T.Johnsson. *Compiling Lazy Functional Languages*. Ph.D.Thesis, Chalmers Tekniska Högskola, Göteberg, Sweden, January 1987.

[2] R.D.Lins. Categorical Multi-Combinators. *Functional Prog. Lang. and Comp. Architecture*, Portland, USA, September 1987, LNCS 274, pp. 60-79, Springer Verlag.

[3] R.D.Lins & B.O.Lira. ΓCMC: Fast Lazy Functional Languages. to appear: *Microprocessing and Microprogramming*.

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

[5] M.A.Musicante & R.D.Lins. GMC: A graph Multi-Combinator Machine. *Microprocessing & Microprogramming*, vol 31(1-5):81-84, North-Holland, 1991.

[6] S.Peyton-Jones. *The Implementation of Functional Languages*. Prentice Hall, 1987.

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