GAML: a Parallel Implementation of Lazy ML

Luc Maranget *

Abstract

We present a new parallel implementation of lazy ML. Our scheme is a direct extension of the G-machine-based implementation of lazy ML. Parallelism is introduced by *fork* annotations inserted by the programmer. We discuss the interference of such user annotations with strictness annotations generated by our compiler. The system has been implemented on a Sequent Balance computer. We also address the main practical issues involved, including stack and heap management.

1 Introduction

The object of this work is to demonstrate that a lazy runtime system such as the G-machine may be implemented efficiently on a parallel shared-memory computer.

It is natural to consider parallelism in a language without side effects. Its semantics guarantee that it is possible to evaluate subexpressions in any order (Church-Rosser property). From a more operational point of view it can be observed that a lazy runtime system already provides facilities for delaying evaluations and that it should be relatively easy to adapt them to the concurrent evaluation of delayed tasks.

Therefore, our starting point is to allow several G-machines to reduce the same graph concurrently. We choose "standard" G-machines for their elegance and simplicity. The performance of our system is competitive with similar implementations [4, 13], taking memory management time into account.

2 A brief tour of GAML

The language implemented by GAML is the ML language with lazy semantics, very close to lazy ML of Göteborg [3, 8]. The syntax is actually closer to the one of CAML [14].

Unlike LML, GAML implements lazy pattern matching as defined by [9, 12]. LML relies on a more classical left-to-right evaluation order of the discriminating expression [3]. Lazy pattern matching, adapts the evaluation order to each set of patterns, guaranteeing termination whenever possible. GAML also has a simple module system which allows type-safe separate compilation.

Compiling lazy ML language to the G-machine has been extensively described in the literature [3, 8, 11]. However, we shall examine this notions first, in order to make our extensions easier to understand.

2.1 Simplified GAML

We consider the language defined by the following grammar:

programs :

$$
\begin{array}{lll}
p & ::= & \textbf{let rec } f_1^{n_1}\ [!]x_1 \ldots [!]x_{n_1} = e_1 \\
 & & \textbf{and} \ldots \\
 & & \ldots \\
 & & \textbf{and} \quad f_m^{n_m}\ [!]x_1 \ldots [!]x_{n_m} = e_m \\
 & & \textbf{in } e
\end{array}
$$

expressions :

$$
\begin{array}{lll}
e & ::= & x\ e_1 \ldots e_n \\
 & | & f\ e_1 \ldots e_n \\
 & | & c_k^n\ e_1 \ldots e_n \\
 & | & \textbf{match } e \textbf{ with } {c_1}^{k_1}\ x_1 \ldots x_n \texttt{ -> } e_1 \mid \ldots \\
 & | & \textbf{let } [\textbf{rec}]\ x_1 = e_1 \textbf{ and} \ldots \textbf{ in } e \\
 & | & !\,e \\
 & | & ?x
\end{array}
$$

We write f^n for a function of arity n, x for a local variable and c_k^n for a constructor of arity n, identified by its constructor number k. The notation [item] expresses an optional item. This language is the intermediate language between the front end of the compiler and the G-code generator. Actually, this language is a subset

*INRIA Rocquencourt, BP 105, 78153 LE CHESNAY CEDEX, FRANCE. This work was partially funded by DRET under grant N°8780814.

of the GAML source language. The front end consists of a sequence of source-to-source transformations, such as pattern matching compilation, strictness analysis and lambda-lifting. Some restrictions should be emphasized: The λ-lifting transformation has been applied to the original source code, as a consequence functions exist at the program level only and constructors cannot be partially applied. Strictness annotations, written "!" and "?", are generated by the strictness analyzer. The exclamation mark "!" has two meanings:

- In **let rec** $f\ !x = \ldots$, it means that the function f is strict in its argument x.
- As an expression, $!\,e$ means that e can be evaluated eagerly.

The annotation ? applied to a variable x means that the expression bound to x is already reduced.

GAML is a lazy language. Expressions are reduced to weak head normal form, or canonical form. A canonical form is either the partial application of a function or a constructed value:

canonical forms :

$$\begin{array}{lcll} r & ::= & f^n\ e_1 \ldots e_m & 0 \le m < n \\ & | & c_k^n\ e_1 \ldots e_n & 0 \le n \end{array}$$

Arguments to functions and constructors are left unevaluated.

2.2 Compilation

Figure 1 explicits the GAML compilation scheme. There are four different compilation functions $\mathcal{F}$, $\mathcal{R}$, $\mathcal{E}$ and $\mathcal{C}$. All these functions take an expression or program as argument and give a sequence of G-code as result. The $\mathcal{R}$, $\mathcal{E}$ and $\mathcal{C}$ functions take an environment ρ and a stack depth d as additional arguments, to manage bindings:

1. $\mathcal{F}[\![\mathrm{f}\ x_1 \ldots x_n = e]\!]$: Compile the definition of a supercombinator f. There are two entry points to a function f, labeled f and $\_f$. When entered as $\_f$, the function assumes that its strict arguments are already reduced. When entered as f, it must reduce its possibly non-canonical strict arguments in a prelude generated by the $\mathcal{A}$ scheme (rule $\mathbf{c}_{1b}$). The strictness analyzer replaces a call to f by a call to $\_f$ whenever appropriate.
2. $\mathcal{R}[\![e]\!]\ \rho\ d$: Evaluate e and return from a function (rule $\mathbf{c}_2$).
3. $\mathcal{E}[\![e]\!]\ \rho\ d$: Evaluate to weak head normal form. This scheme is initially called by $\mathcal{R}$. It is recursively called on the strict parts of the language constructs ($\mathbf{c}_4$ to $\mathbf{c}_6$), on primitives (such as arithmetic operations) and on expressions annotated by the ! annotation (rule $\mathbf{c}_{15}$).
4. $\mathcal{C}[\![e]\!]\ \rho\ d$: Construct a graph. This scheme is invoked on the arguments to functions and constructors, as well as on expressions bound by a **let** (rules $\mathbf{c}_{11}$ and $\mathbf{c}_5$–$\mathbf{c}_6$). While compiling function application, when the arity of the function is known at compile time, RAP nodes are introduced to represent partial applications (rule $\mathbf{c}_9$),

The actual compiler implements more sophisticated schemes, that perform most usual optimizations: tail calls in the $\mathcal{R}$ scheme by the DISPATCH instruction, direct calls in the $\mathcal{E}$ scheme by the CALL instruction [11]. Neither do we consider functions of zero arguments.

A program **let rec** $f \ldots$ **in** e is compiled by calling the $\mathcal{F}$ scheme on the function definitions f. Its execution consists of evaluating the expression e to normal form. This is done by the usual "printing" mechanism, which basically is a repeated application of evaluation to weak head normal form in a deep-first, left-to-right traversal.

2.3 The G-machine

The nodes in the graph are:

- Binary application nodes, AP $n_1\ n_2$. The real implementation also uses vector application nodes, which represent functions applied to the right number of arguments in a compact way. In the simplified model, AP nodes are the only non-canonical nodes.
- Reduced application nodes, RAP $n_1\ n_2$. They represent partial application, which are canonical forms (rule $\mathbf{e}_2$ in figure 2).
- Function nodes, FUN (C, a), where C is the compiled code of the function and a its arity.
- Constructor nodes CONSTR $c_k^a\ n_1 \ldots n_a$.

The G-machine reduces expression graphs to weak head normal form. Its state is described by a 4-tuple $<C,\ s,\ G,\ D>$, where C is the code being executed, s is the main stack, G is the graph and D is the *dump* —a stack saving machine states when the G-machine is called recursively by an EVAL instruction. The initial state of the machine is $<$EVAL.(), n.(), $G[n = N]$, ()$>$, where n is a pointer to a graph representation N of the expression to be reduced, $G[n = N]$ is a graph containing N and () stands for the empty sequence.

The transitions of the sequential G-machine are summarized in figure 2. The typical reduction process, started by EVAL applied to a "root" AP node ($\mathbf{e}_1$), repeatedly applies rule $\mathbf{e}_3$ to "unwind the spine" on the S

$\mathbf{c}_1$ $\mathcal{F}[\![f\ x_1 \ldots x_n = e]\!] =$
  LABEL f; $\mathcal{A}[\![x_1, \ldots x_n]\!]$; LABEL _$f$; $\mathcal{R}[\![e]\!]\ [x_1 = n \ldots x_n = 1]\ n \quad (n > 0)$

$\mathbf{c}_{1b}$ $\mathcal{A}[\![x_1, \ldots\ !x_i, \ldots\ x_n]\!] = \ldots$; PUSH $i-1$; EVAL; POP 1; $\ldots$

$\mathbf{c}_2$ $\mathcal{R}[\![e]\!]\ \rho\ d = \mathcal{E}[\![e]\!]\ \rho\ d$; UPDATE $(d+1)$; POP d; UNWIND

$\mathbf{c}_3$ $\mathcal{E}[\![c_k\ e_1\ \ldots\ e_n]\!] = \mathcal{C}[\![c_k\ e_1\ \ldots\ e_n]\!] \quad (n \geq 0)$

$\mathbf{c}_4$ $\mathcal{E}[\![$**match** e **with** $c_1{}^n\ x_1 \ldots x_n$ -> $e_1 \ldots]\!]\ \rho\ d =$
  $\mathcal{E}[\![e]\!]$; CASE $(l_1, l_2, \ldots)$;
  LABEL l_1; SPLIT; $\mathcal{E}[\![e_1]\!]\ \rho[x_1 = d+n, \ldots x_n = d+1]\ (d+n)$;
  SQUEEZE $1\ n$; JUMP l ; $\ldots$ LABEL l

$\mathbf{c}_5$ $\mathcal{E}[\![$**let** $x_1 = e_1$ **and** $\ldots\ x_n = e_n$ **in** $e]\!]\ \rho\ d =$
  $\mathcal{C}[\![e_1]\!]\ \rho\ d; \ldots; \mathcal{C}[\![e_n]\!]\ \rho\ (d+n-1)$;
  $\mathcal{E}[\![e]\!]\ \rho[x_1 = d+1, \ldots x_n = d+n]\ (d+n)$; SQUEEZE $1\ n$

$\mathbf{c}_6$ $\mathcal{E}[\![$**let rec** $x_1 = e_1$ **and** $\ldots\ x_n = e_n$ **in** $e]\!]\ \rho\ d =$
  ALLOC^{n†}; $\mathcal{C}[\![e_1]\!]\ \rho_r\ (d+n)$; UPDATE n; $\ldots\ \mathcal{C}[\![e_n]\!]\ \rho_r\ (d+n)$; UPDATE 1;
  $\mathcal{E}[\![e]\!]\ \rho_r\ (d+n)$; SQUEEZE $1\ n$ (where $\rho_r = \rho[x_1 = d+1, \ldots x_n = d+n]$)

$\mathbf{c}_7$ $\mathcal{E}[\![?x]\!]\ \rho\ d =$ PUSH $d - \rho(x)$

$\mathbf{c}_8$ $\mathcal{E}[\![e]\!]\ \rho\ d = \mathcal{C}[\![e]\!]\ \rho\ d$; EVAL

$\mathbf{c}_9$ $\mathcal{C}[\![f^n\ e_1 \ldots e_m]\!]\ \rho\ d =$
$$\begin{cases} \mathcal{C}[\![e_m, \ldots e_1]\!]\ \rho\ d;\ \mathsf{PFUN}\ f;\ \mathsf{MKRAP}^{m\dagger} & (m < n) \\ \mathcal{C}[\![e_m, \ldots e_1]\!]\ \rho\ d;\ \mathsf{PFUN}\ f; \mathsf{MKRAP}^{(n-1)\dagger};\ \mathsf{MKAP}^{(m-n+1)\dagger} & (m \geq n) \end{cases}$$

$\mathbf{c}_{10}$ $\mathcal{C}[\![e_f\ e_1 \ldots e_n]\!] = \mathcal{C}[\![e_n, \ldots e_1]\!]\ \rho\ d; \mathcal{C}[\![e_f]\!]\ \rho\ (d+n)$; MKAP^n

$\mathbf{c}_{11}$ $\mathcal{C}[\![e_n, \ldots e_1]\!]\ \rho\ d = \mathcal{C}[\![e_n]\!]\ \rho\ d; \ldots \mathcal{C}[\![e_1]\!]\ \rho\ (d+n-1)$

$\mathbf{c}_{12}$ $\mathcal{C}[\![c_k^n\ e_1\ \ldots\ e_n]\!]\ \rho\ d = \mathcal{C}[\![e_n, \ldots e_1]\!]\ \rho\ d$; MKCONSTR c_k^n

$\mathbf{c}_{13}$ $\mathcal{C}[\![x]\!]\ \rho\ d =$ PUSH $d - \rho(x)$

$\mathbf{c}_{14}$ $\mathcal{C}[\![f]\!]\ \rho\ d =$ PFUN f

$\mathbf{c}_{15}$ $\mathcal{C}[\![!e]\!]\ \rho\ d = \mathcal{E}[\![e]\!]\ \rho$

† ALLOC^n, MKRAP^n and MKAP^n stand for ALLOC, MKRAP and MKAP repeated n times.

Figure 1: Simplified compilation scheme of GAML

stack. Once the function at the bottom of the spine is reached, and if there are enough arguments on the stack ($\mathbf{e}_4$), the code for this function is entered. This code, as compiled by rule $\mathbf{c}_2$ in figure 1, constructs a new graph in canonical form, updates the root ($\mathbf{e}_7$) and resumes the unwinding process.

The G-code is expanded into native machine code in the efficient way presented in [3]. Our sequential compiler is similar to the Chalmers LML compiler [3, 8], in design and performance.

3 Parallelism at the source level

The parallel extension preserves the deterministic lazy semantics of GAML. The reduction of expressions is organized in a different way, so that some parts of the evaluation process may be performed simultaneously.

3.1 Annotations

Idealy, the compiler should determine where parallel computations can be started, without user intervention. As a first step, we have chosen to rely on user annotations in the source program, indicating subcomputations to be performed in parallel. Automatic parallelization can be added later on as a source to source transformation. The advantage of source to source transformations is that they give the user a precise idea of what the compiler is doing.

Parallelism annotations come in addition to strictness annotations. Consider, for instance, Takeushi's function:

```
let rec tak x y z = match  x < y with
  true -> z
| false ->
    tak
      (tak (x-1) y z)
```

e_1	$<\mathsf{EVAL}.C,\ n.s,\ G[n = \mathrm{AP}\ n_1\ n_2],\ D> \quad \Rightarrow <\mathsf{UNWIND}.(),\ n.(),\ G[\ldots],\ (C,s).D>$
e_2	$<\mathsf{EVAL}.C,\ n.s,\ G,\ D>$, and n canonical $\Rightarrow <C,\ n.s,\ G,\ D>$
e_3	$<\mathsf{UNWIND}.(),\ n.s,\ G[n = (\mathrm{R})\mathrm{AP}\ n_1\ n_2],\ D> \Rightarrow <\mathsf{UNWIND}.(),\ n_1.n.s,\ G[\ldots],\ D>$
e_4	$<\mathsf{UNWIND}.(),\ n_0.n_1\ldots n_k.s,\ G[n_0 = \mathrm{FUN}\ (C_f, k),\ n_1 = \mathrm{AP}\ n'_1\ n''_1,\ \ldots n_k = \mathrm{AP}\ n'_k\ n''_k\ ],\ D>$ $\Rightarrow\ <C_f,\ n''_1\ldots n''_k.n_k.s,\ G[\ldots],\ D>$
e_5	$<\mathsf{UNWIND}.(),\ n_0.n_1\ldots n_k.(),\ G[n_0 = \mathrm{FUN}\ (C_f, a)],\ (C,s).D>$ and $a < k \quad \Rightarrow$ $<C,\ n_k.s,\ G[n_0 = \mathrm{FUN}\ (C_f, a)],\ D>$
e_6	$<\mathsf{UNWIND}.(),\ n.(),\ G[n = \mathrm{CONSTR}\ c^a_k\ n_1\ \ldots\ n_a],\ (C,s).D> \ \Rightarrow\ <C,\ n.s,\ G[\ldots],\ D>$
e_7	$<\mathsf{UPDATE}\ d.C,\ n_0.n_1\ldots n_d.s,\ G[n_0 = N_0,\ n_d = N_d],\ D> \quad \Rightarrow$ $<C,\ n_1\ldots n_d.s,\ G[n_0 = N_0,\ n_d = N_0],\ \mathrm{D}>$
e_8	$<\mathsf{PUSH}\ d.C,\ n_0\ldots n_d.s,\ G,\ D> \quad \Rightarrow <C,\ n_d.n_0\ldots n_d.s,\ G,\ D>$
e_9	$<\mathsf{PFUN}\ f.C,\ s,\ G[n = \mathrm{FUN}\ (C_f, a)],\ D> \Rightarrow <C,\ n.s,\ G[n = \mathrm{FUN}\ (C_f, a)],\ D>$
e_{10}	$<\mathsf{POP}\ d.C,\ n_1\ldots n_d.s,\ G,\ D> \quad \Rightarrow <C,\ s,\ G,\ D>$
e_{11}	$<\mathsf{SQUEEZE}\ d\ m.C,\ n_1\ldots n_{d+m}.s,\ G,\ D> \Rightarrow <C,\ n_1\ldots n_d.s,\ G,\ D>$
e_{12}	$<\mathsf{ALLOC}.C,\ s,\ G,\ D> \quad \Rightarrow <C,\ n.s,\ G[n = \mathrm{HOLE}],\ D>$
e_{13}	$<\mathsf{MKAP}.C,\ n_1.n_2.s,\ G,\ D> \quad \Rightarrow <C,\ n.s,\ G[n = \mathrm{AP}\ n_1\ n_2],\ D>$
e_{14}	$<\mathsf{MKCONSTR}\ c^a_k.C,\ n_1\ldots n_a.s,\ G,\ D> \quad \Rightarrow <C,\ n.s,\ G[n = \mathrm{CONSTR}\ c^a_k\ n_1\ \ldots\ n_a],\ D>$
e_{15}	$<\mathsf{CASE}\ (l_1, l_2\ldots)\ldots\mathsf{LABEL}\ l_k.C,\ n.s,\ G[n = \mathrm{CONSTR}\ c^a_k\ n_1\ \ldots n_a],\ D> \quad \Rightarrow$ $<C,\ n.s,\ G[\ldots],\ D>$
e_{16}	$<\mathsf{SPLIT}.C,\ n.s,\ G[n = \mathrm{CONSTR}\ c^a_k\ n_1\ \ldots n_a],\ D> \Rightarrow <C,\ n_1\ldots n_a.s,\ G[\ldots],\ D>$

Figure 2: Transitions of the sequential G-machine

```
      (tak (y-1) z x)
      (tak (z-1) x y) in
  tak 18 12 6
```

It is compiled as follows:

```
let rec tak !x !y !z = match  ?x < ?y with
  false ->
   _tak
     !(_tak !(int._sub ?x 1) ?y ?z)
     !(_tak !(int._sub ?y 1) ?z ?x)
     !(_tak !(int._sub ?z 1) ?x ?y)
| true -> ?z in
_tak 18 12 6
```

Strictness analysis successfully infers that the `tak` function is strict in its three arguments. As a consequence, all calls to the `tak` function are made directly without constructing a graph, since the optimized $\mathcal{E}$ scheme, induced by the `!` annotation compile them with a CALL instruction. Furthermore all calls are made through the `_tak` entry of the function, so that the arguments are not tested for canonical form.

3.2 An annotation for parallelism

We use a *fork* annotation "#", such an annotation is also called *spark* following [11]. The expression $\#e$ means that e is to be reduced in parallel with the current thread of computation. This makes sense only if $\#e$ can be referenced later. A first case is when $\#e$ is an argument to a constructor. This constructor can be destructured later, by a **match** construct or by the printing mechanism, so that the value of e can be retrieved. A second case is when $\#e$ is bound to a variable. That is, $\#e$ is an argument in a function call, or bound by a **let** construct. Consider, for instance:

```
let rec fib n =  match n with
  0 -> 1
| 1 -> 1
| _ -> let v = #(fib (n-1)) in 1 + fib (n-2) + v in
fib 30
```

The programmer here specifies that the call `(fib (n-1))` should be done in parallel. When the computation needs the result of this call, it will find it bound to variable `v`. Synchronization is easily expressed by the syntax of the expressions itself.

It is not always obvious to decide when to fork exactly. Consider:

```
let K x y = x
and F x = fib x + fib (x/2) in
K 0 (F #(F (F 2)))
```

It is not clear when #(F (F 2)) should be forked: while constructing the graph for the whole expression (F #(F (F 2))) ? Or while reducing it, that is while performing the outermost call to F (which, in this case, will not happen) ? The second solution gives the programmer more precise control on fork time. Thus, $\#e$ means "start reducing e in parallel at the time when this expression is bound". An expression is bound either by a **let** construct or by reducing a function call. To do so, the compiler introduces a new function G and rewrites the example above as:

```
let K x y = x
and F x = fib x + fib (x/2)
and G x = F #(F (F x)) in
K 0 (G 2)
```

This transformation allows to choose the "fork at binding time" semantics without any special node in the graph. It also applies to annotated **let** bindings. The $\mathcal{E}$ scheme is the only one that deals with parallel bindings, $f \ \ldots \ \#e \ \ldots$ and **let** $x = \ \#e$ **in**….

The case of the constructors is less ambiguous, the expression $c_k^n \ldots \#e \ldots$ just means: "fork at the time when the value is actually constructed", both the $\mathcal{C}$ and $\mathcal{E}$ compilation schemes construct this value.

3.3 Treatment of # by the compiler

The presence of the spark annotation does not inhibit strictness-based optimizations. Moreover, these optimizations are combined with another source to source transformation expressing load control at runtime.

On a real computer with a limited number of CPUs, it is likely that an expression will not be reduced as soon as it is forked. It can happen that a forked expression is not yet reduced when its parent computation thread needs it. In this case, the parent thread reduces the unevaluated expression as it would do in the sequential case. This is the simplest and most efficient solution. Waiting for someone else to do the reduction is both wasteful and difficult to implement safely.

This "advisory fork" mechanism has additional benefits. Since a possibly forked expression can be reduced by its parent, it is not necessary to actually fork it when the computer load is too high. This can be expressed at the source level. The compiler transforms the Fibonacci example above:

```
let rec fib !n = match ?n with
      0 -> 1
    | 1 -> 1
    | _ -> match ?sys.forkNow with
        true ->
         let v_3 = #(_fib !(int._sub ?n 1)) in
         let b_5 = !(_fib !(int._sub ?n 2)) in
         let b_6 = !v_3 in
         int._add (int._add 1 ?b_5) ?b_6
       | false ->
         let b_7 = !(_fib !(int._sub ?n 2))
         and b_8 = !(_fib !(int._sub ?n 1)) in
         int._add (int._add 1 ?b_7) ?b_8 in
_fib 30
```

Here sys.forkNow is a boolean system variable set to true by the runtime system when the load is low. This example shows the main advantage of expressing load control at the source level. In the case when no fork is not done (the false branch), strictness optimizations apply, and the call !(_fib (int._sub ?n 1)) is compiled into a direct call by the optimized $\mathcal{E}$ scheme. This would not be possible if the decision whether or not to fork were done by the runtime system alone.

Another point is that the programmer does not need to worry about the order of evaluation of strict expressions. In the Fibonnaci example, the programmer specifies that the expression (_fib !(int._sub ?n 1)) should be forked. The compiler introduces annotated **let** bindings, expressing the fork (binding v_3), evaluate the other needed subexpression (binding b_5) and synchronize (binding b_6) evaluation order. This order is implicitly meant by the programmer when he puts the # annotation.

4 The parallel G-machine

4.1 Compilation to the parallel G-machine

A parallel G-machine is best described as one of several G-machines operating on the same graph.

A given subgraph cannot be reduced by more than one G-machine at a time. This could lead to inconsistent graphs. And such duplication of work wastes resources. Therefore, when applied to an AP node, the EVAL and UNWIND instructions change its tag to a new tag, the RUN tag. Another machine will not perform an EVAL or UNWIND instruction on a RUN node. As several machines may attempt to change an AP node into a RUN node simultaneously, the tag test and change sequence must be atomic. This is done by using a lock associated to the memory address of the tag. Locking is always expensive, even on the architectures where any memory address can be viewed as a lock, which is not the case in our Sequent Balance system. Thus, we try to avoid this penalty as much as possible.

$$
\begin{array}{lll}
\mathbf{c'}_1 & \mathcal{C}[\![\#e]\!]\ \rho\ d & = \mathcal{C}[\![e]\!];\ \mathsf{FORK} \\
\mathbf{c'}_2 & \mathcal{C}[\![f^n\ e_1\ \ldots e_m]\!]\ \rho\ d = & \\
& \left\{ \begin{array}{ll} \mathcal{C}[\![e_m, \ldots e_1]\!]\ \rho\ d;\ \mathsf{PFUN}\ f;\ \mathsf{MKRAP}^m & (m < n) \\ \mathcal{C}[\![e_m, \ldots e_1]\!]\ \rho\ d;\ \mathsf{PFUN}\ f;\ \mathsf{MKRAP}^{(n-1)};\ \mathsf{MKAP}^{(m-n)};\ \mathsf{MKSAP} & (m \geq n) \end{array} \right. & \\
\mathbf{c'}_3 & \mathcal{C}[\![e_f\ e_1\ \ldots e_n]\!]\ \rho\ d & = \mathcal{C}[\![e_n, \ldots e_1]\!]\ \rho\ d;\ \mathcal{C}[\![e_f]\!]\ \rho\ (d+n);\ \mathsf{MKAP}^{(n-1)};\ \mathsf{MKSAP}
\end{array}
$$

Figure 3: Compilation of parallel GAML

The atomic change from AP to RUN state is not needed if the AP node is not shared. This leads us to distinguish between possibly shared application nodes SAP and certainly unshared application nodes AP. It does not matter whether a partial application RAP node is shared or not, since these nodes are never modified.

Possibly shared expressions are the ones bound by a **let**, the arguments in an application, and the arguments to a constructor. All these expressions are compiled with the $\mathcal{C}$ scheme (figure 1, rules $\mathbf{c}_5$, $\mathbf{c}_6$, and $\mathbf{c}_9$ to $\mathbf{c}_{12}$). Thus, the $\mathcal{C}$ scheme is modified: rules $\mathbf{c}_9$ and $\mathbf{c}_{10}$ are replaced by the new rules $\mathbf{c'}_2$ and $\mathbf{c'}_3$ (figure 3). The new rules generate a SAP node when called on a (possibly) non-canonical expression.

Obviously this criterion is not optimal. In some cases, a bound expression is not shared. Some sharing analysis could be used to determine unshared variables [7, 5]. This analysis seems not very useful in our case, because the spark annotation introduces a lot of sharing. For instance, in the Fibonacci function, variable $\mathbf{v}$ is shared between the parent thread and the child thread, whereas it is obviously not shared in a sequential implementation.

4.2 A parallel G-machine model

In this section we explain how several G-machines cooperate. A concurrent G-machine state is a 6-tuple $<C, s, D, G, R, F>$. The components C, s, G and D are the same as in the sequential case. We will explain the *run pool* R and the *fork pool* F later. The components C, s and D are local to each G-machine, whereas G, R and F are shared between the cooperating G-machines. The transitions of the parallel G-machine are shown in figure 4. Transitions $\mathbf{e'}_1$–$\mathbf{e'}_4$ and $\mathbf{e'}_8$–$\mathbf{e'}_{10}$ comes in addition to the transitions of the sequential G-machine. Whereas transition $\mathbf{e'}_5$ replaces transition $\mathbf{e}_5$ and transitions $\mathbf{e'}_6$–$\mathbf{e'}_7$ replaces transition $\mathbf{e}_7$.

The graph G has two new types of nodes:

- the shared application node, SAP n_1 n_2, created by the MKSAP instruction.
- the RUN node, RUN p n_2. A RUN node is created by overwriting a SAP node. This is the reason why a RUN node has two fields, though only the first one p is used.

When a machine forks an expression, it just puts the graph representing it in the *fork pool* F (transition $\mathbf{e'}_8$), where idle machines can pick it up ($\mathbf{e'}_{10}$). When a machine attempts to EVAL or UNWIND a node being processed by another machine, it finds a node with a RUN tag. Evaluation of the current task cannot be pursued. Therefore, its state is saved as a process structure $<C, s, D>$ and queued on the p field of the RUN node ($\mathbf{e'}_2$ and $\mathbf{e'}_4$). The machine then enters the idle state $<(), (), (), G, R, F>$. When the machine processing the RUN node overwrites it with its canonical form (instruction UPDATE), it will awake the processes on this node (transition $\mathbf{e'}_6$). These awakened processes become runnable, they are put in the *run pool* R, where idle machines can pick them up (transition $\mathbf{e'}_9$).

In the parallel G-machine, UPDATE is a generic instruction just like EVAL or UNWIND: its effect depends upon the nodes it operates on (transitions $\mathbf{e'}_6$ and $\mathbf{e'}_7$). A new UPDATE instruction has to be inserted when an UNWIND instruction fails (transition $\mathbf{e'}_5$). In this case, the root node to be updated (N_k) can be a RUN node, so its tag must be changed into RAP and the suspended processes queued in the p field must be awakened.

A complete system is made of several G-machines. Initially, one machine is in the state $<\mathsf{EVAL}.(), n.(), D, G[\ n = \ldots], (), ()>$, where n is a pointer to the expression to compute. The others machines are in idle state, waiting for the initiator machine to put something in the fork pool.

4.3 Implementation

The fork pool is a statically allocated area of memory holding pointers to forked expressions. It is not accessed in an exclusive manner, so that some forked expressions may be lost (if two machines forks simultaneously) or picked up more than once. This does not compromise the correctness of the implementation, since spark is advisory and the RUN tag already prevents several machine from reducing the same sub-graph. This non-locked access mechanism is crucial for the efficiency of

$$\mathbf{e'}_1 \quad <\mathsf{EVAL}.C,\ n.s,\ D,\ G[n = \mathrm{SAP}\ n_1\ n_2],\ R,\ F> \ \Rightarrow$$
$$<\mathsf{UNWIND}.(),\ n_1.n.(),\ (C,s).D,\ G[n = \mathrm{RUN}\ ()\ n_2],\ R,\ F>$$

$$\mathbf{e'}_2 \quad <\mathsf{EVAL}.C,\ n.s,\ D,\ G[n = \mathrm{RUN}\ p],\ R,\ F> \Rightarrow$$
$$<(),\ (),\ (),\ G[n = \mathrm{RUN}\ (<\mathsf{EVAL}.C,\ n.s,\ D>.p)],\ R,\ F>$$

$$\mathbf{e'}_3 \quad <\mathsf{UNWIND}.(),\ n.s,\ D,\ G[n = \mathrm{SAP}\ n_1\ n_2],\ R,\ F> \ \Rightarrow$$
$$<\mathsf{EVAL}.(),\ n.(),\ (\mathsf{UNWIND}.(), n.s).D,\ G[\ \ldots],\ R,\ F>$$

$$\mathbf{e'}_4 \quad <\mathsf{UNWIND}.(),\ n.s,\ D,\ G[n = \mathrm{RUN}\ p],\ R,\ F> \ \Rightarrow$$
$$<(),\ (),\ (),\ G[n = \mathrm{RUN}\ (<\mathsf{UNWIND}.(),\ n.s,\ D>.p)],\ R,\ F>$$

$$\mathbf{e'}_5 \quad <\mathsf{UNWIND}.(),\ n_0.\ldots.n_k.(),\ (C,s).D,\ G[n_0 = \mathrm{FUN}\ (C_f, a)\ ],\ R,\ F> \text{ and } a < k \Rightarrow$$
$$<\mathsf{UPDATE}\ 1.C,\ n.n_k.s,\ D,\ G[\ n_0 = \ldots,\ n = \mathrm{RAP}\ n_{k-1} *(n_k + 2)],\ R,\ F>$$

$$\mathbf{e'}_6 \quad <\mathsf{UPDATE}\ d.C,\ n_0.n_1\ldots n_d.s,\ D,\ G[n_0 = N_0,\ n_d = \mathrm{RUN}\ p],\ R,\ F> \Rightarrow$$
$$<C,\ n_1\ldots n_d.s,\ D,\ G[n_0 = N_0,\ n_d = N_0],\ R[p],\ F>$$

$$\mathbf{e'}_7 \quad <\mathsf{UPDATE}\ d.C,\ n_0.n_1\ldots n_d.s,\ D,\ G[n_0 = N_0,\ n_d \neq \mathrm{RUN}\ p],\ R,\ F> \Rightarrow$$
$$<C,\ n_1\ldots n_d.s,\ D,\ G[n_0 = N_0,\ n_d = N_0],\ R,\ F>$$

$$\mathbf{e'}_8 \quad <\mathsf{FORK}.C,\ n.s,\ D,\ G,\ R,\ F> \quad \Rightarrow <C,\ n.s,\ D,\ G,\ R,\ F[n]>$$

$$\mathbf{e'}_9 \quad <(),\ (),\ (),\ G,\ R[p = <C,\ s,\ D>],\ F> \quad \Rightarrow <C,\ s,\ D,\ G,\ R,\ F>$$

$$\mathbf{e'}_{10} \quad <(),\ (),\ (),\ G[n = \mathrm{SAP}\ n_1\ n_2],\ R,\ F[n]> \ \Rightarrow <\mathsf{EVAL}.(),\ n.(),\ (),\ G[\ldots],\ R,\ F>$$

Figure 4: Transition rules of the concurrent G-machine

the parallel G-machines. Another important point is that the fork pool is statically allocated, so that the FORK operation does not allocate memory. These two points make a FORK instruction very cheap: it takes between 9 and 20 machine instructions (including testing buffer overflow).

Unlike the fork pool F, the run pool R needs exclusive access. Once started, a reduction must be completed, otherwise the whole computation could stop before completion. The run pool is implemented as a linked list of processes. A lock can be used to guard a global pool, or each machine can have its own private RUN pool. This choice is not critical, since few EVAL or UNWIND instructions find nodes in RUN state. Therefore the RUN pool is empty most of the time.

4.4 Scheduling

In figure 4 no scheduling policy is imposed idle machines can take either transaction $\mathbf{e'}_9$ or $\mathbf{e'}_{10}$.

In the best case, all forked expressions are eventually required (conservative parallelism) and the scheduling strategy is only a matter of efficiency. If a process does not terminate, neither will the complete program, since the non-terminating process computes a needed value.

Experimentation shows that the scheduler should look at the run pool first to avoid creating a new process when an already existing process is runnable (non-running processes waste stack and heap resources). The strategy governing the run pool itself has little impact, because the run pool is empty most of the time. If the run pool is local to each machine, then it can be checked just once per call to the scheduler.

The best strategy for the fork pool is a LIFO strategy: the fork buffer is a bounded stack of 128 pointers. Other alternatives tested are the linked list, the circular buffer and various "random" policies. Experiments indicate that the evaluation of forked expressions should not be delayed too much. This seems to help to keep the number of suspended tasks low and avoid reductions of forked tasked by their parents.

When some forked expression may not be needed (speculative parallelism), we must make sure that the processes computing the normal reduction are not blocked for ever by speculative tasks. This can be done by pre-emptive fair scheduling. We plan to design a more clever implementation, where a "token" is passed through the running processes, so that the process holding the token is computing the normal order branch of reduction, and that the process holding the token is always running.

Another important point is the management of the `forkNow` variable. A simple policy gives good results. When an idle machine finds the fork pool empty it sets `forkNow` to `true`. When a forking machine finds the fork pool full it sets `forkNow` to `false`.

5 Heap management

Heap management is the hidden cost the previous parallel G-machine implementations. Usually they use a sequential garbage collector. That is, one processor is collecting the memory using a traditional algorithm while the others are idle. This is a waste of CPU power.

5.1 Calling the parallel collector

We want to keep the efficiency and simplicity of stop & copy garbage collectors. In this section we present a parallel version of this algorithm. Our algorithm is parallel in the sense that all running processors take part in the collection. Collection is not performed concurrently with the main program however. Therefore, this approach does not require complicated synchronization between processors executing user code and processors collecting memory as in [2].

The heap space is divided in two spaces, the from-space and the to-space. The processors request to-space pages from a global memory manager, which is a critical section. After getting a page, each processor allocates in it without any further synchronization. As described in [8, page C-18], a request for the heap space needed by a basic block is made at the beginning of the block. When there is not enough memory left in the current page to satisfy a request, the processor asks for a new heap page. Thus, large heap pages are desirable, to minimize synchronization. If a memory request is more than one page large, then it should be broken into smaller ones, by shortening basic blocks. If large objects such as arrays are to be allocated (this is not the case in GAML), then several contiguous pages can be given by the memory manager.

If a request for a new page fails, then the requesting machine enters collector state. All the others allocating machines will end up having consumed their own pages. We experimentally checked that it was not worth the trouble for the allocating processors to try to enter collector state before running out of space naturally. However, when a machine is polling on the fork queue or performing a non-allocating computation, it does not allocate any memory. Such machines periodically check a shared variable. When a machine enters collector state, it sets this variable, so that non-allocating machines join it for starting collection. We cannot use very large pages, because the larger the page, the longer the delay for starting garbage collection. In practice $4Kb$ is a good tradeoff.

5.2 Safe copying in parallel

Once all the processors are in collector state, the collection begins. The to-space and the from-space are swapped. Then each processor starts a traditional stop & copy algorithm, allocating memory page by page in the new to-space. To preserve the structure of the graph, no object can be copied by more than one processor. Therefore synchronization is needed. More precisely, while tracing referenced objects, a collector repeatedly invokes the `copyObj` function to copy object into its own to-space pages. We give the C code for the `copyObj` function, that copies objects, and then, overwrite their old locations with a forward reference to their new locations.

```
word pointer *last;  /* Next free position  */

void INCRLAST(n);    /* Handle memory allocation */

word MemLock(obj);   /* Lock obj, return its tag */

/* UnLock obj, change its tag */
word MemUnlock(obj,newTag);

word *copyObj(obj)
word *obj;
{
 word *tag,*newObj,oldObj;

 tag = (word *)*obj;
 switch (tag) {
   ....
  case SAP:                    /* not yet copied */
   tag = (word *)MemLock(obj);
   if (tag == MOVED) {         /* lost race */
    MemUnlock(obj,MOVED);
       /* return forward pointer */
    return((word *)*(obj+1));
   } else {               /* won race */
    INCRLAST(3);          /* three words needed */
    newObj = last; oldObj = obj++;
       /* fill first two words */
    *last++ = SAP;  *last++ = *obj;
    *obj++ = newObj;      /* set forward pointer */
    MemUnlock(oldObj,MOVED);
    *last++ = *obj;       /* copy last word */
    return(newObj);
  }
   ....
  case MOVED:            /* already copied  */
      /* return forward pointer */
   return ((word *)*(obj+1));
    ....
 }
}
```

The tag check and the setting of the forwarding pointer have to be performed atomically: This is done by using the already present mechanism of tags as locks from section 4.1 (cf functions `MemLock` and `MemUnlock`). In the source code above, the SAP tag is changed this way into a new MOVED tag identifying pointers to moved nodes. The overhead for locking is paid only when the object has a chance to be copied, i.e. no mutual exclusion is necessary to access the forwarding pointer in a MOVED node, or to retrieve the statically allocated version of a node, as it is the case for FUN nodes and CONSTR nodes with zero argument.

5.3 The roots for the GC

At the beginning of the GC, the roots considered are the stacks of the running processes, plus the runnable processes in the run pool. As the GC proceeds, the processes suspended on RUN nodes are discovered . The stacks of suspended processes are also scanned as roots. All theses processes are said to be active, their stacks have to be scanned since they are running or may run in the future.

The fork pool also has to be scanned. But it is not part of the roots, the pointers to the graph in the fork pool are "weak pointers": if an object is pointed to from the fork pool only, then it is garbage, since it is not referenced to by any active process. This is done by scanning the fork pool after the main collection has been completed, not collecting objects not copied at this stage and replacing pointers to them by a pointer to a dummy object. The scanning of the fork pool itself is done in parallel, which is easy since the fork pool is a contiguous zone of memory.

5.4 Distributing the work

As all machines cooperate to collect unused space it is important to distribute the work to be done as evenly as possible, to maximize parallelism. We do this by distributing the roots to be scanned between the processors.

A first idea is just to have each processor scan its own running process (if any), and then let it scan the suspended processes as they are found in the heap, queued on RUN nodes. This solution does not lead to good balancing: some processors are idle while others are scanning large stacks. To improve on this, we divide the job of scanning a stack into "scanning tasks". We fix a granularity constant T. A scanning task consists of a base pointer sp and of an offset o less than T. Completing a scanning task is scanning a stack, starting at location $sp + o$, skipping T by T.

When a processor has a stack to scan, it puts the corresponding scanning tasks in a global pool. As this pool is not accessed with mutual exclusion (for efficiency reasons), the initiator of tasks must then try to complete them, in case they are lost in the pool. This absence of synchronization also implies that several processors may start completing the same scanning task. But it is easy for a processor completing a scanning task to know that this task has already been completed or is being completed by another processor: as long as it has not yet been examined, a pointer to the heap is a pointer to the from-space ; once the pointed object has been copied, the pointer is changed into a pointer to the to-space. So, a processor scanning a portion of stack gives up when it encounters a pointer to the to-space.

The choice of the granularity constant T is critical. If it is too small, the distribution of the work is poor. If it is too big there are too many tasks and the stacks are not efficiently scanned. Our benchmarks gives best results with T between 8 and 16.

6 Implementation of the stacks

The theoretical G-machine has tree stacks: The S stack, that is used to unwind the spine, the V stack, for the intermediate results of arithmetic computations, and the dump or D stack. In our parallel implementations, stack space is given to a G-process on its explicit demand. This implies that a machine has to check stack overflow, so that it can ask for more stack space when needed. In our parallel implementation, this mechanism is needed for one real stack only, and the stack overflow check is not done so often.

6.1 The V stack

Traditionally, arithmetic expressions are compiled with a stack model in mind. That is, arithmetic operators pop their arguments from a stack and push their results on it. The G-machine has the V stack for this purpose. We omitted the V stack from our description of the G-machine for the sake of simplicity.

The V stack is a stack of integers or other basic values, which the garbage collector must not consider as pointers. This is the reason why it is convenient to implement the V and S stacks as separate stacks: the garbage collector simply ignore the V stack. If we want to have one stack only, and still not use a conservative garbage collector, then the garbage collector must be able to distinguish a pointer from a non-pointer value. This can be achieved by *tagging*: one bit per word is reserved to indicate its pointer or non-pointer status, see [6, p. 50] for instance. This information can be stored somewhere else than in the word itself [1, 4, 13]. These solutions imply important modifications of the runtime system.

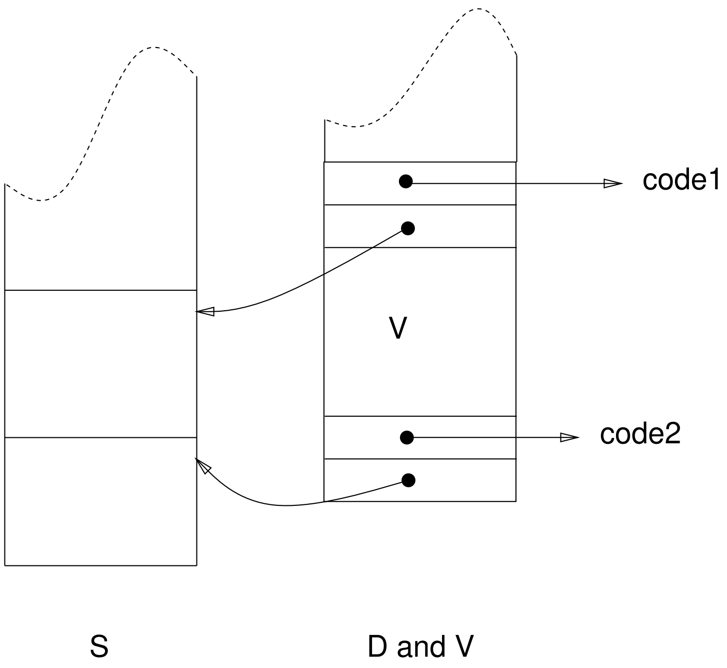

Figure 5: Stacks in the sequential implementation

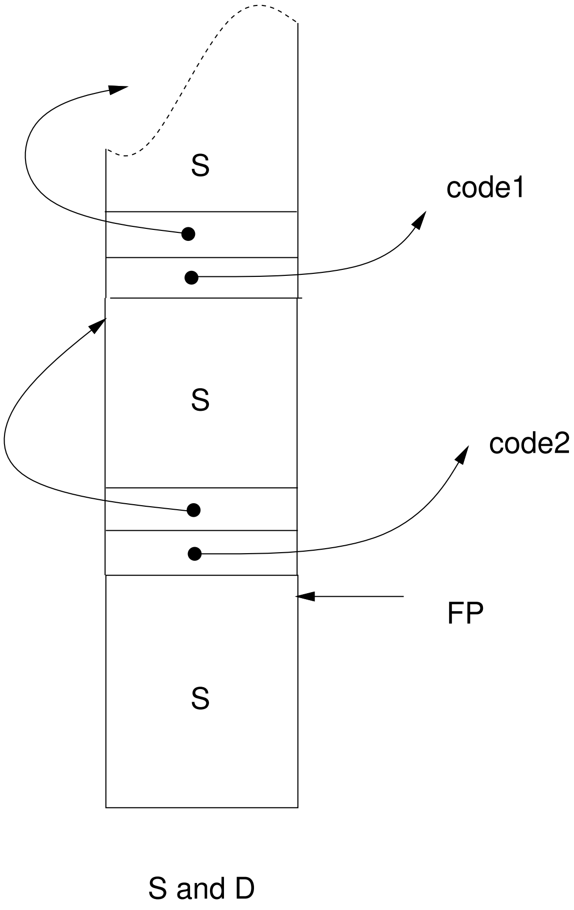

Figure 6: Stacks in the parallel implementation

Fortunately, a simpler solution is already given in [3, p. C-27–C-29]: the V stack can be made empty each time the heap is checked. More precisely, intermediate arithmetic results are stored in the S stack when they survive basic blocks. This is achieved by a source-to-source transformation: the V stack elimination. It lifts expressions that compile into an EVAL out of arithmetic expressions, replacing them by fresh variables bound by a strict **let**:

```
fib (n-1) + fib (n-2) + 1
```

will be transformed into:

```
let a = !(fib (n-1))
and b = !(fib (n-2)) in
?a+?b+1
```

Since a process gets suspended by evaluating a RUN node, the V stack will always be empty at that time. Therefore, the V stack is not part of the saved state of a process.

6.2 The D stack

The D stack is used to remember G-machine states, that is, pairs (S, C) of S stacks and return addresses. In the sequential implementation, all the S stacks are allocated contiguously on one real stack. A saved state is two entries on the dump, one pointing to the code and the other pointing to the beginning of the S stack starting just below the saved S stack (cf figure 5, stacks are growing from top to bottom). In our parallel implementation, we store a pointer to the beginning of the current S stack in the FP (frame pointer) register. Of course, when a new S stack is started, the EVAL instruction has to save the old frame pointer, it just pushes it on the real stack with the return address. Therefore, the only real stack used will look as depicted in figure 6: a sequence of S stacks separated by pairs of saved frame pointers and program counters. The garbage collector easily retrieves the pointers to the heap from such a stack, either by following the saved frame pointers or by just checking all words in the stack against heap bounds.

6.3 Checking stack overflow

Thus, there is only one real stack per process. We

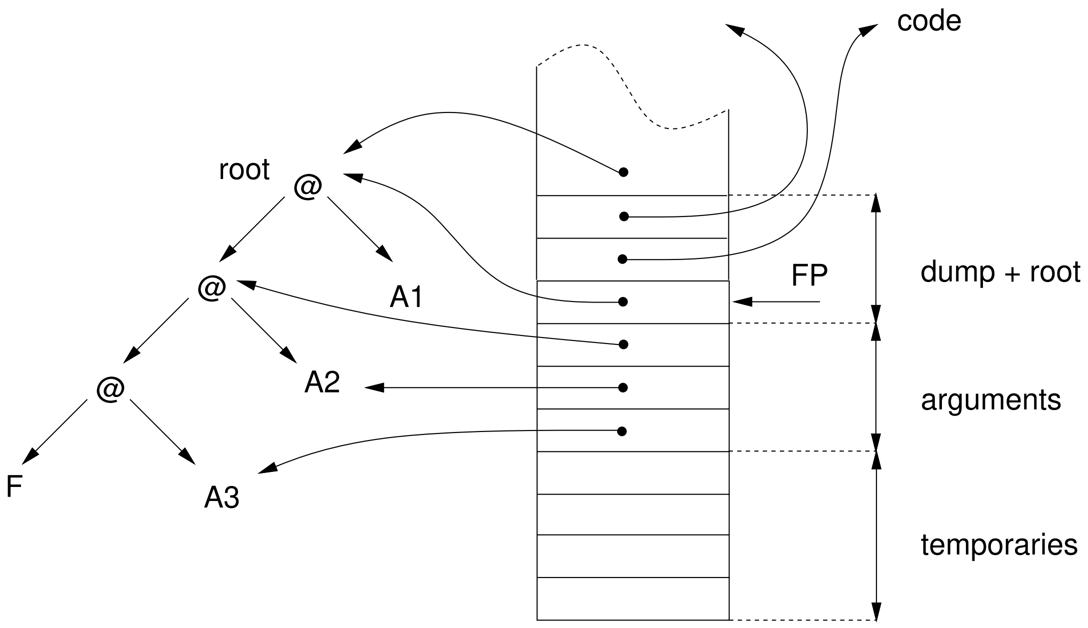

Figure 7: A typical S stack

simply call it "the stack". In a strict language checking stack overflow needs be performed at the beginning of the functions only: each function checks whether there is enough room on the stack to hold its local variables and temporaries. This does not work in GAML, because the EVAL and UNWIND instructions may use an amount of stack that cannot be bounded by a local analysis. But, for a given program, the depth of the S stacks can bounded by a constant depending upon the program only.

Consider a typical S stack, such as the one in figure 7. Assume that this stack was started to compute the expression $E = F\ A_1\ A_2 \ldots A_n$, where F is a supercombinator and A_1,A_2,$\ldots A_n$ are arbitrary expressions. From top to bottom, first, there are three slots for the old frame pointer, the return address and a pointer to the root node of E. Then, n stack entries hold the arguments that were *given* to F. These arguments were put on the stack while unwinding the spine. The remaining entries, up to the stack pointer, are filled with the temporaries and local variables of F. This description also applies when F is invoked from another supercombinator via UNWIND.

The code generator can easily compute an upper bound temp(F) of the number of temporaries used by the function F. In ML, which is a typed language, an upper bound args(F) of the number of arguments than can be given to F, can also be computed. Suppose that F has a non-polymorphic type, then F cannot be invoked with more arguments then specified by its type. For instance, the curried addition on integers, of type int $\rightarrow$ int $\rightarrow$ int, cannot be applied to more than two arguments. But types in ML may be polymorphic. Fortunately, as observed by Andrew Appel [1], each function is used with a finite number of different ground types. David Lester [10] states that these monomorphic types can be computed while type-checking the program, by conserving all the instances of polymorphic type schemes. Consider the program:

```
let I x = x in I I 1
```

The polymorphic function `I` of type $\forall\alpha \rightarrow \alpha$ is used with the type instances "(int $\rightarrow$ int) $\rightarrow$ int $\rightarrow$ int" and "int $\rightarrow$ int". Here, args(I) = 2. Some generic variables may remain in the type instances. As they cannot be instantiated any further, this has no impact on the computation of args(F). For instance, in `let I x = x in I I`, the type instances are $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ and $\alpha \rightarrow \alpha$, and args($I$) = 2.

Therefore, for a program P, the S stack will never grow to more than $N = 3 + \max(\text{args}(F)) + \max(\text{temp}(F))$ elements, where F ranges over the supercombinators defined by P. This upper bound can be computed at compile time. Thus, before starting a new S stack, the EVAL instruction checks that there is at least N free entries on the real stack. If not, it asks a new stack chunk from global memory manager, and changes the return address in the new S stack, so that

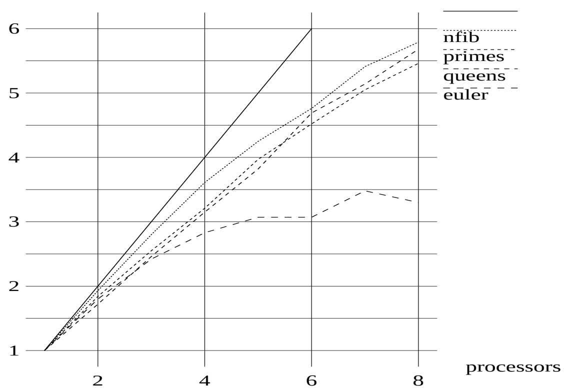

Figure 8: Speedups relative to one processor.

the stack chunk will be freed while returning from EVAL. In this way, the activation records of the functions are stack-allocated.

Unfortunately, the computation of N in the presence of separate compilation requires to put *all* the instances of polymorphic types in the public part of the modules. This appears quite inconvenient, if not prohibitive. In practice, we only use the fact that the N upper bound exists, and fix it to a hopefully large enough value. In our benchmarks 32 is large enough. If more safety is required, the stack must be checked each time EVAL or UNWIND pushes something on it, and at the beginning of the functions.

7 Performance

7.1 Overall performance

There are four benchmarks. The sources are given in the appendix. The *nfib30*, *primes* and *queens* benchmarks exhibit the same good performance in speedup (figure 8). The *euler* test gives a disappointing result. Namely, not only the *euler* test uses more memory than the others, but it also generate parallelism in a different way.

In table 1, we give the wall clock time performance of the parallel implementation relatively to the sequential implementation (comparable with LML 0.99). Combined with the speedup figures, these data show that parallelism leads to real benefits, as soon as more than one processor is used.

The figure 9 shows how the total CPU time and its distribution evolves as the number of processors increases. Total CPU time is the sum of the CPU time of all the cooperating processors. This time is divided in three categories, according to where it is spent:

- **user** corresponds to the code of the user functions,
- **gc** is the garbage collector,
- **par** accounts for the scheduler loop and the management of the Unix processes implementing the cooperating G-machines.

In all the tests, the **user** time slightly increases with the number of processors, because of some synchronization costs in the EVAL, UNWIND and UPDATE instructions. In addition, for the *nfib30* the strictness optimizations (in the `false` branch of the `forkNow` test, section 3.3) may apply less often.

For the first three tests, the **par** time stays marginal. There is little polling on the fork queue, because these programs can fork tasks evenly. For the *euler* test the parallelism is generated by scanning the list of the first `1000` integers. For each cons in the list, two tasks are forked, one for a large computation and the other for treating the next cell in the list. If a scanning task gets lost —which is likely when the fork buffer has less than twice as many elements as the list—, the scanning of the list is stopped. Scanning will restart once the normal reduction order has reached the point where scanning stopped. This introduces delays.

7.2 Efficiency of the GC

We deliberately choose a relatively small heap size (one megabyte), to get closer to the behavior of real programs, for which the time spent in garbage collection is significant.

As the number of processors increases, so does the percentage of running time spent in garbage collection. This is not only because garbage collection is not parallelized as efficiently as the GAML programs themselves. Parallelism has a cost in terms of memory. The number of calls to the GC does not increase much with parallelism though. The slight increase in the total memory allocated comes from process structures (but there are few of them), and extra graphs created to be put in the fork pool. But the increase in the number of processes has an indirect effect: there are more stacks to scan, furthermore there are more live objects in the heap, i.e., the heap occupancy rate increases as shown in the table below:

procs	nfib30	primes	queens	euler
1	0.10	7.09	11.92	5.74
4	0.36	7.07	12.41	9.63
8	0.62	7.04	13.94	14.04

Hence, our stop & copy-based GC, has more objects to copy. This effect depends upon the benchmark considered.

Acknowledgments

I thank Xavier Leroy and Georges Gonthier for their editorial work. I also thank Robert Ehrlich who made the *gproff* Unix profiling utility work for parallel programs.

References

[1] Andrew W. Appel, *Runtime Tags Aren't Necessary*, Lisp and Symbolic Computation, vol 2 1989.

[2] Andrew W. Appel, John R. Ellis, and Kai Li, *Real-time Concurrent Collection on Stock Multiprocessors*, PLDI'88.

[3] Lennart Augustsson, *Compiling Lazy Functional Languages Part II*, Ph. D. Thesis, Chalmers University of Technology, 1987.

[4] Lennart Augustsson and Thomas Johnsson, *Parallel Graph Reduction with the* $<\nu, G>$*-machine*, FPCA'89.

[5] Alain Deutsch, On determining lifetime and aliasing of dynamically allocated data in higher order functional specifications POPL'90.

[6] *Report on the Programming Language Haskell*, Version 1.0, YALEU/DCS/RR-77.

[7] Paul Hudak, *A Semantic Model of Reference Counting and its Abstraction*, in *Abstract Interpretation of Declarative Languages*, Ellis Horwood 1987.

[8] Thomas Johnsson, *Compiling Lazy Functional Languages*, Ph. D. Thesis, Chalmers University of Technology, 1987.

[9] Alain Laville, *Implementation of Lazy Pattern Matching Algorithms*, ESOP'88, LNCS 300.

[10] David Lester, *Stacklessness: Compiling Recursion for a Distributed Architecture*, FCPA'89.

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

[12] Laurence Puel, Ascánder Suárez, Compiling Pattern Matching by Term Decomposition, LFP'90.

[13] Simon L. Peyton-Jones, John Salkild, *The Spineless Tagless G-machine*, FPCA'89.

[14] Pierre Weis, *The CAML Reference manual* Version 2.6.1, INRIA Technical Report 121.

Figure 9: Distribution of CPU time.

	nfib30	primes	queens	euler
sequential	130 (1.00)	139 (1.00)	337 (1.00)	314 (1.00)
parallel	213 (1.63)	199 (1.43)	467 (1.39)	430 (1.37)

Table 1: Performance of parallel GAML relatively to sequential GAML

Appendix: source of the benchmarks

primes: primes numbers up to 1000.

```
let filter f =  let rec f_rec l = match l with
      []    -> []
    | x::xs -> if f x then x::#(f_rec xs) else f_rec xs in f_rec in

let rec sieve l = match l with
    []    -> []
  | p::xs ->  p::sieve (filter (fun x ->x mod p <> 0) xs) in

sieve (count 2 10000)
```

queens : the 10 queens problem

```
let concmap f =  let rec c_rec m = match m with
      []    -> []
    | x::xs -> append (f x) #(c_rec xs) in  c_rec in

let filter f = let rec filter_rec l =  match l with
    []    -> []
  | x::xs -> if f x then x::#(filter_rec xs) else filter_rec xs in filter_rec in

let rec safe x d l = match l with
  []    -> true
| q::qs -> x<>q  & x<>q+d & x<>q-d & safe x (d+1) qs in

let ok l =  match l with  []    -> true | x::xs -> safe x 1 xs in

let rec gen n = match n with
  0 -> [[]]
| _ -> concmap
         (fun b ->filter ok (map (fun q ->q::b) (count 1 10)))
         (gen (n-1)) in

length (gen 10)
```

euler : the euler totient function

```
let map f =  let rec map_rec l = match l with
    []    -> []
  | x::xs -> #(f x)::#(map_rec xs) in map_rec in

let rec gcd x y = match y with 0 -> x | _ -> gcd y (x mod y) in
let relprime x y = gcd x y = 1 in
let euler n =  length (filter (relprime n) (fromto 1 (n-1))) in

sum (map euler (count 1 1000))
```