The Chalmers Lazy-ML Compiler

L. AUGUSTSSON AND T. JOHNSSON
Department of Computer Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden

We present the principles and pragmatics of a compiler for Lazy ML, a lazy and purely functional variant of ML, developed at Chalmers University of Technology. The aim has been to develop an implementation that enables efficient execution on today's computers. The compiler itself has been written almost entirely in Lazy ML.

We first briefly describe the Lazy ML language, in particular with respect to pattern matching, modules and separate compilation, and input/output. A programming example is given. We outline the general structure of the LML compiler and then describe in some depth the technical details of program transformation phases and the G-machine, the abstract machine underlying the implementation. Performance issues are also discussed. We describe some application programs written in LML, and the experience gained from them.

Received November 1988

1. INTRODUCTION

Functional languages, and in particular lazy ones, have many appealing properties; this is amply demonstrated elsewhere in this issue. However, wider use of functional languages has been hampered by the lack of good implementations.

Unfortunately functional languages, and in particular lazy ones, are much harder to implement than conventional imperative ones. The reason for this should come as no surprise: the design of functional languages focuses entirely on expressiveness and semantic simplicity, while paying little or no attention to the peculiarities of present computer hardware. This is in contrast to conventional imperative languages, like FORTRAN, Pascal and Ada. They have grown out of a tradition of computer hardware development, being abstractions of the machines they run on: a variable being an abstraction of a memory cell; assignment statements being an abstraction of load and store machine instructions; FOR statements, WHILE statements, etc. being abstractions of machine instruction sequencing and jump instructions. Consequently, such languages are fairly easy to implement efficiently, since they map well on to present computers.

As a result of the big 'semantic gap' between functional languages and present-day computer hardware, many researchers are actively seeking novel ways of organising hardware (see Ref. 11). Many researchers also seek to exploit the inherent parallelism due to the lack of side effects in the language, and design multiprocessor architectures capable of executing functional programs in parallel without the programmer telling it how to subdivide the computation into processes.[3,13,37] But even when this research results in useful architectures, it will still be a long time before such machines become widely available. In the meantime, we are stuck with the conventional von Neumann computer.

Our research has focused on implementing lazy functional languages on present-day computers, rather than trying to invent new hardware. Our work stems from work on combinator graph reduction.[31] By a process called *variable abstraction*, variables are removed entirely from the program by translating it into an expression containing the combinators *S*, *K*, *I*, *B* and *C* from combinatory logic, augmented with curried functions for the predefined operators of the source language, such as *add*, *sub*, *if*, etc. Execution of the program is then carried out in the *SKI-reduction machine*, which performs graph reduction on combinator expressions. While the method is very elegant, the performance from a straightforward implementation of it is dismaying: perhaps two orders of magnitude slower than conventional code for a function. The reason is that the combinators subdivide the computation into a large number of tiny reduction steps, each of which only pushes a function argument a single level further down in the abstract syntax tree of the function.

Our lazy evaluation method is similar to Turner's combinator reduction method, but instead of using a standard, fixed set of combinators, each user-defined function is used as a 'combinator', i.e. a rewrite rule in the graph. The graph manipulation of each function is expressed as code for the *G-machine*, from which target code generation is rather straightforward. In effect, the compiler constructs a specialised, machine-language coded combinator interpreter from each program.

One of our aims has been to make lazy functional languages practical, everyday tools of programming, enabling efficient execution on today's computers. We have implemented a compiler for Lazy ML, or LML for short, which emits target machine code to perform graph reduction. The rest of this paper describes the LML language and the principles and pragmatics of this compiler. For an excellent introduction to the basic problem and different approaches to graph reduction, see Ref. 29.

This paper is organised as follows. Section 2 briefly describes the LML language, and in particular pattern matching, modules and separate compilation, and input/output. Section 3 first outlines the general structure of the LML compiler and then describes in some depth the technical details of program transformation phases and the abstract machine underlying the implementation. Performance issues are also discussed. Section 4 discusses some examples of applications written in LML and the experience gained from this. Section 5 contains a brief discussion of related work.

2. THE LAZY ML LANGUAGE

Lazy ML, or LML for short, is a strongly typed, polymorphic, purely functional language, in the tradition

of ML,[16] Hope,[6] SASL[30] and Miranda.[33]* Like SASL and Miranda but unlike Hope and ML, LML has call-by-name semantics. The first version of the LML language was a lazy variant of the purely functional subset of ML; hence the name Lazy ML. Later, user-definable concrete data types and pattern matching were added to the LML language.

Lazy ML is a fairly rich functional language; we only describe some of the pertinent features here. A complete description of the language is given in Ref. 1 and a denotational semantics for the language is given in Ref. 5.

From the outset, LML has been designed for the conventional batch compile–load–execute cycle (on UNIX™ in our case). Hence a program is an expression (possibly subdivided into several separately compiled programs), and to execute the program means to print the value of the expression. (This is in contrast to ML or Miranda, for example, where programs are sequences of declarations and expressions entered incrementally into an interactive system.) LML thus fits well into the UNIX environment, with Make and other tools. LML programs may be fitted together with other UNIX programs using pipes or communication via the file system.

2.1 Pattern matching

Pattern matching is an essential part of modern functional languages. It allows succinct definitions of functions by several equations. For instance, the function for computing the length of a list is written in LML as follows

$length\,[\,] = 0 \parallel$

$length\,(x\,.\,xs) = length\,xs + 1$

Pattern matching can be used to define a function by several equations, but also to scrutinize a value with a **case** expression. The patterns are built from constructors and variables and may be arbitrarily complex. The pattern matching in LML is sequential; if there are several equations they are tested top-down, and each one is tested from left to right. Pattern matching is always strict, even if there is only one equation or if the type contains only one constructor (e.g. pairs).

It is also possible to attach a guard to a pattern. An adorned pattern will only match if the pattern matches and the guard, which is an expression of boolean type, evaluates to true. Patterns must be linear, i.e. variable names may not be duplicated, but guarded patterns can be used to simulate non-linear patterns. A pattern which is non-linear, e.g. "(x, x)" (the intention being that for the pattern to match the two components must be equal), is rewritten to a pattern with several distinct variables and a guard that checks that they are equal, "$(x1, x2)\&(x1 = x2)$".

2.2 Declarations

LML is a structured language, à la ISWIM[25] or ML. This means that the definitions of names, for instance functions, can have limited scope, and there is also control over what definitions are recursive. This approach is the opposite to the one in KRC,[32] for example, which is a 'flat' language where all definitions are global.

There are two basic forms of declaration, non-recursive value definitions such as

$square\,x = x * x$

and non-recursive type definitions for example

type $Colour = red + green + blue$

Declarations may be combined into more complex declarations by three operators on declarations: **and, rec** and **local.** The **and** operator combines two declarations into one, and defines what was defined by them. The **rec** operator takes a single declaration and makes it recursive, i.e. the identifiers defined in it may be used inside the declaration itself. For instance, the declaration

$square\,x = x * x$
and
rec $fac\,n =$ **if** $n = 0$ **then** 1 **else** $n * fac(n-1)$

declares *square* and *fac*, where *fac* is recursive. The operator **local** is useful, for instance, when an auxiliary function is needed in more than one function, but the auxiliary function itself is not to be visible outside the declaration. Example:

local $h\,x = \cdots$
in
$f\,x = \cdots h \cdots$
$g\,x = \cdots h \cdots$
end

2.3 Modules and separate compilation

For practical reasons separate compilation is an absolute necessity when developing large programs. LML is no exception to this; the compiler is simply not fast enough that substantial applications could be compiled without it. The unit of separate compilation is called a "module", which consists of declarations (of types and values), or the main expression being the value of the program.

However, separate compilation makes some program analysis techniques much more difficult, since they often require that the whole program should be available for analysis – an example of this is strictness analysis. Another example is polymorphic type checking; here the current LML implementation does not allow mutually recursive modules, i.e. mutually recursive functions defined in different modules (the reasons will become apparent below). Thus we have favoured practical usability at the expense of ease of global analysis.

The module mechanism is built very much using a standard UNIX approach: use text files as far as possible,

```
module
infixr "::";
export Signal, Shd, Stl;
rec (type Signal *a = *a :: (Signal *a))
and Shd (h::t) = h
and Stl (h::t) = t
end
```

Figure 1. The file signal.m**, a module defining the type Signal and operations on it.**

* Miranda is trademark of Research Software Ltd.

```
module
#include "signal.t"
export gnand, pr, osc;
rec pr l = map (\x. if Shd x then '1' else '0') l @ "\n" @ pr (map Stl l)
and gnand' (x::xs) (y::ys) = (~(x & y)) :: gnand' xs ys
and gnand x y = false :: gnand' x y
and osc n m = gen true n (gen false m (osc n m))
and gen _ 0 s = s
|| gen x n s = x :: gen x (n-1) s
end
```

Figure 2. The file gates.m, **a module defining a nand gate, an oscillator, and a function to print a signal.**

```
#include "signal.t"
#include "gates.t"

let RSflipflop r_ s_ =
    let rec q  = gnand r_ q_
    and     q_ = gnand q  s_
    in q
in
let s1 = osc 11 2
and s2 = osc 9  3
in  pr [s1; s2; RSflipflop s1 s2]
```

Figure 3. The file main.m, **a main program defining and simulating an R-S flip-flop circuit.**

use standard tools such as "make", the assembler and the linker. This has the advantage that there is much less work in implementing the system, and that it integrates well into the UNIX environment. There is no need to write special syntax-directed editors, persistent storage, etc.

Figs 1, 2 and 3 show a sample LML program, which simulates a digital circuit using infinite streams of boolean values. The *n*th boolean stream element signifies the logical value of that signal at the *n*th time unit.

Each module must reside in its own file. The program consists of three separately compiled modules: two modules proper, the file signal.m (Fig. 1), the file gates.m (Fig. 2), and the file main.m containing the main expression. Importation of a module into another one is done by including the compiler-generated ".t" file of the imported module (see below). Lines like "#include "signal.t"" are in effect directives to the C preprocessor which is run prior to parsing in the compiler.

When an LML file (with name suffix ".m") is compiled, two files are produced: an object file (".o"), which is used when the final program is produced by linking all object files, and a type file (".t"), which contains information about all identifiers that are exported from the file. The file contains type and arity information that is needed to compile other modules that import the first module. The compiler-generated ".t" files in the example are given in 4.

An LML module is usually rather short – 20–200 lines is the typical size. It is usually a good idea to structure the program in small modules, for programming methodology reasons of course, but also to keep recompilation time down when changes are made.

There is also an (extensible) library of standard functions. It would be a nuisance to have to import explicitly a commonly used predefined function, like *length* or *reduce*, for every module where they are used. Therefore the functions in the standard library are implicitly imported into all modules.

2.4 Input/output

The input and output facilities in LML are quite simple, but adequate for many applications. A program can access the standard input (normally connected to the keyboard) as a lazy list of characters. Lazy evaluation makes writing interactive programs possible; since the complete input list may not be needed to compute part of the output, a program can produce, say, a prompt and then wait for the input characters to arrive.

To read arbitrary files there is a "read" function. Read takes a list of characters, the name of a file, and returns the contents of the file as a list of characters.

An LML program normally produces a list of characters as its result. This list is printed on the standard output (normally the screen) as it is computed. To produce other files this list of characters can contain special characters that are not printed, but instead

file signal.t:

```
infixr "::";
import type Signal *a = *a :: (Stream *a);
import Shd: Signal *a->*a;
import Stl: Signal *a->Signal *a;
```

file gates.t:

```
import gnand: Signal Bool->Signal Bool->Signal Bool;
import pr: List (Signal Bool)->List Char;
import osc: Int->Int->Signal Bool;
```

Figure 4. The compiler-generated files signal.t **and** gates.t.

redirects the characters that follow to a different file. This method of output is somewhat crude, but simple and powerful enough to allow many programs to be written. The function tofile is used to get a string that, when produced as output, redirects the following characters. The program ""Hello"@tofile "foo"@"world"" will produce the characters Hello on the standard output and the characters world in a file named foo.

It is also possible to read the input in a non-blocking mode. In this mode a special character, a "hiaton",[27] is read whenever there is no input character available. The non-blocking input is essential for writing things like real-time applications (e.g. games).

3. THE COMPILER

3.1 Overview of the compiler passes

The LML compiler, being written (mostly) in LML itself, is organised into about 25 passes, of which most are syntax tree-to-syntax tree transformations. The result of each such pass could in principle be written out again as a valid LML program. This has proved to be a very convenient approach, since it is easier to verify (both informally and formally) that a transformation is correct when it is performed at a high level.[5]

Below follows a brief description of most of the passes; a more complete description can be found in Ref. 5. The passes have different purposes: some transform the program from the full language to the subset of the language that the code generator can handle, others try to make performance-improving transformations.

(1) *Parse*. The parser reads the program and prints the syntax tree, which is then read by the compiler proper. This is the only compiler pass not written in LML; it uses the standard UNIX tool, Yacc, instead.

(2) *Rename*. The rename pass renames all bound variables so that they have unique names. This makes later transformations much easier, since there is no longer any risk of name clashes. The rename pass also checks for undefined identifiers, etc.

(3) *Flattening*. As described in Section 2.2, LML allows complicated nested declarations. This pass flattens these.

(4) *Pattern transformation*. All kinds of pattern matching are transformed into **case** expressions, which are then transformed into **case** expressions with only simple patterns (as described in Section 3.2).

(5) *Recursion removal*. A program may have declarations that are indicated to be recursive (with **rec**) without really being recursive. These declarations may have been there in the source program, but they are also introduced by earlier transformations. This pass removes the unnecessary recursiveness.

(6) *Type inference*. This pass checks that the program is type-correct. LML has a type system very similar to ML.[26]

(7) *Application transformation*. This pass transforms the program so that the (leftmost) function part of an application always is an identifier.

(8) *Reductions*. This pass evaluates the program to some degree, for example, by constant folding, by reducing **case** expressions with a known scrutinised part, and by β-reduction.

(9) *Strictness analysis*. A very simple strictness analysis is performed. It mostly keeps track of which variables have already been evaluated (to avoid re-evaluation).

(10) *Lambda lifting*. Removes all nested functions (as described in Section 3.3).

(11) *G-code generation*. Generate code for the *G-machine*, the abstract machine underlying the implementation. The G-code is used as an intermediate code in the compiler. The G-machine is described in detail in Sections 3.5–3.9.

(12) *G-code improvement*. The G-code is broken into basic blocks and control flow improvements are made, for example dead code elimination and jump removal.

(13) *M-code generation*. Generates M-code, an intermediate code that captures the characteristics of common register machines (as described in Section 3.10).

(14) *Machine-code generation*. Generates target machine code (in assembly language form).

Which of these passes are important to performance? This is difficult to answer, since they are in some ways interrelated.

Removing unnecessary recursion in a program has a surprisingly large impact on performance. The reason for this is that the pattern transformation may introduce a number of unnecessary recursive declarations. All passes before the reduction may introduce inefficient constructions, such as a **let**-bound variable that is only used once, and these will disappear during the simplification. Except for the things introduced by the compiler itself, this pass would have very little impact on performance. The strictness analysis makes a big difference on some programs, namely those that do a lot of arithmetic. On more 'normal' programs it has little or no impact. The performance of the LML compiler does not depend on very complicated analysis such as strictness analysis or sharing analysis. The G-code improvement mostly improves the control flow of the program, many jumps being eliminated. Depending on the target machine at hand, this may be of small to medium importance.

3.2 Pattern-matching transformation

Pattern matching allows succinct and readable function definitions. When using pattern matching, function definitions are made up of a number of cases covering different alternatives. The fundamental pattern-matching mechanism in LML is the **case** expression. Other kinds of matching, i.e. function definitions and **let** expressions, are translated into **case** expressions.

A **case** expression (see Fig. 5) has an expression part and a number of pattern-expression pairs. Evaluating a case expression entails scrutinising the expression to find out which of the given patterns it matches (the patterns are tried in succession, and from left to right). When a matching patterns is found the value of the case expression is the value of the expression corresponding to that pattern. The expression is evaluated with all the variables in the pattern bound to the corresponding subparts of the scrutinised expression.

A pattern is built up from constructors and variables

$$\begin{aligned} \mathit{caseexpr} &::= \textbf{case}\ e\ \textbf{of}\ \mathit{pat}_1 : e_1 \,||\, \mathit{pat}_2 : e_2 \,||\, \cdots \\ \mathit{pat} &::= x \mid c_k\ \mathit{pat}_1 \cdots \mathit{pat}_m \end{aligned}$$

Figure 5. Syntax of case expressions and patterns.

and it can be arbitrarily complex (as long as it follows the grammar and is type-correct). To make compilation easier all case expressions are transformed into **case** expressions (possibly nested) that contain only *simple patterns*. A simple pattern is a pattern that has a constructor as its outermost form, with all the subpatterns being variables. The reason for this translation is that matching simple patterns can be compiled into very efficient code in a fairly straightforward manner (see below).

The algorithm for turning arbitrary **case** expressions into nested **case** expressions proceeds as follows.

(1) Rearrange the order of the patterns into groups, each group having the same top-level constructor. The rearrangement requires some care since the order of the patterns in **case** expressions matters, thus only non-overlapping patterns may be interchanged without any change of semantics.

(2) Each of these groups will form a single pattern and a nested **case** expression to determine which of the patterns actually match.

(3) Each of the **case** expressions introduced in step 2 is handled by this algorithm until only simple patterns are left.

The complete transformation algorithm is described in Refs 4 and 29, and we will only show an example here. Given the function definition

$$\begin{array}{l} last\ nil = error \\ \| \ last\ (cons\ x\ nil) = x \\ \| \ last\ (cons\ x\ xs) = last\ xs \end{array}$$

the translated function definition will be

$$\begin{array}{l} last\ zs = \mathbf{case}\ zs\ \mathbf{of} \\ \qquad nil : error \\ \qquad \| \ cons\ x\ xs : \mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad nil : x \\ \qquad\qquad \| \ cons\ y\ ys : last\ xs \end{array}$$

A simple pattern can be compiled very efficiently. Two things happen when a **case** expression is evaluated: first the matching pattern is determined, then the variables in the pattern are bound to their corresponding expressions. The G-machine has one instruction for each of these. There is a CASE instruction which examines a constructor node (it has to be evaluated before it can be examined) and performs a multi-way jump depending on which constructor it is. The other instruction is SPLIT, which takes a constructor node and pushes all the subparts of it on the stack. Since the stack is used for accessing variables this accomplishes the binding of the variables in the pattern; they are simply taken from the stack after this instruction has been executed.

3.3 Lambda lifting

The language we can compile directly to G-code is a 'flat' language, i.e. all function definitions occur at the global level. We do know how to compile some **let** or **where** expressions into efficient G-code directly (see Section 3.9); however, the **let**- or **where** expressions are not allowed to define functions (i.e. no lambda expressions in the function bodies). Together with pattern matching, this flat language is a very powerful and succinct programming language in its own right – KRC,[32] for instance, is of this sort. However, LML like most other functional languages permit local function declarations in **let**- or **where** expressions, and to compile such programs, they must be transformed to the flat form. This transformation step is called *lambda lifting*.

The basic idea is to turn the lambda expressions into global functions, but before doing so the free variables of the lambda expressions need to be abstracted out: ($e[u]$ denotes an expression e with a free variable u),

$$\begin{array}{c} \cdots\lambda u.\cdots(\lambda x.\lambda y.e[u])\cdots \\ \Downarrow \\ \cdots\lambda u.\cdots(\lambda u.\lambda x.\lambda y.e[u])\,u\cdots \\ \Downarrow \\ \begin{cases} f\,u\,x\,y = e[u] \\ \cdots\lambda u.\cdots f u\cdots \end{cases} \end{array}$$

However, the above method would treat a locally defined function in a let expression as follows (a function definition $f\,x = e$ is regarded as syntactic sugar for $f = \lambda x.e$).

$$\begin{array}{c} \cdots\lambda u.\cdots \mathbf{let}\ f\,x\,y = e[u] \\ \qquad\qquad \mathbf{in}\cdots f e_1 e_2\cdots \\ \Downarrow \\ \begin{cases} f'\,u\,x\,y = e[u] \\ \cdots\lambda u.\cdots \mathbf{let}\ f = f'\,u \\ \qquad\qquad \mathbf{in}\cdots f e_1 e_2\cdots \end{cases} \end{array}$$

i.e. the function f will appear as a local variable (bound to $f'u$). Now for reasons of making the program susceptible to tail-call and direct-call optimisations (see Section 3.9) we would prefer the function f to be a global function definition, rather than a local variable. This can be achieved by modifying the lambda-lifting strategy for **let** expressions as follows: abstract out the free variables of the function by adding them as parameter to the function, and then apply the variables at each use of the function. Our example now becomes:

$$\begin{array}{c} \cdots\lambda u.\cdots \mathbf{let}\ f\,x\,y = e[u] \\ \qquad\qquad \mathbf{in}\cdots f e_1 e_2\cdots \\ \Downarrow_1 \\ \cdots\lambda u.\cdots \mathbf{let}\ f u\,x\,y = e[u] \\ \qquad\qquad \mathbf{in}\cdots f u e_1 e_2\cdots \\ \Downarrow_2 \\ \begin{cases} f u\,x\,y = e[u] \\ \qquad\qquad \cdots\lambda u.\cdots f u e_1 e_2\cdots \end{cases} \end{array}$$

Note that the **let** expression is eliminated. However, because of substitutions like $e[fu/f]$, in general free variables u may be introduced where there were none before. So the abstraction step $\Leftarrow_1$ above must be repeated until there are no free variables in functions definitions. To avoid repeated transformations of the program, we instead compute the final set of variables that will have to be abstracted out of the functions, by solving a system of set equations – for details see Ref. 21.

3.4 Normal-order graph reduction

The semantics of lazy functional languages imply that programs should be evaluated using *normal-order reduction*, i.e. the leftmost outermost reducible expression (or

redex) should be reduced first. For various reasons, a linear 'string' representation of the expression, such as one would use when calculating by hand, is unsuitable for use in a computer – one reason is that sharing of computation would be lost (see below). We therefore use another representation, directed graphs, on which we are to perform *normal-order graph reduction*. We will interpret each function definition as a graph rewrite rule. With graph reduction, a rewrite step entails replacing the application of the next redex with the graph of the right-hand side, with *pointers to* argument graphs substituted for formal parameters. The replacement must be done by actually updating the root of the redex with the root of the new expression graph.

Fig. 6 shows graph reduction of *square*(2+3), where *square* $x = x * x$. In the graphs, application is denoted by @. To get a uniform representation we represent expressions using the binary operators +, *, etc. by applications to predefined functions *add*, *mul*, etc.

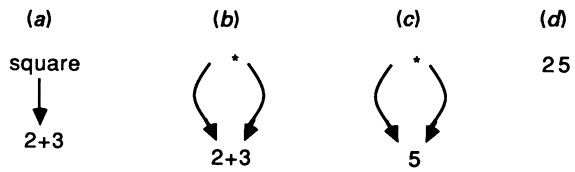

Figure 6. Graph reduction of *square* (2+3).

Note that if a parameter occurs more than once in the right-hand side (as is the case in Fig. 6) the corresponding argument graph becomes shared. A shared subexpression is reduced to its value the first time it is needed, and all further use of it benefits from the first evaluation. On the other hand, if the value is not needed at all, no time is wasted reducing it. (This is a property that string reduction does not have.) This mode of passing function arguments is called *call-by-need*.

A normal-order graph reduction interpreter would repeatedly perform the following steps: (1) find the leftmost outermost redex, (2) instantiate the function body of the function applied in the redex, i.e. build a graph of the right-hand side of the function body with pointers to the actual arguments substituted for the formal parameters, and (3) update the root of the redex with the root of the instance. Step (1) is simplified by keeping a stack of pointers to the apply nodes of the path from the root of the expression to the leftmost outermost redex. Thus the function and the arguments of the current redex can always be found via the topmost pointers of the stack.

3.5. Compiled graph reduction and the G-machine

The efficiency of the instantiation step (2) of an implementation of the interpreter can be increased considerably by tailoring the interpreter to each set of functions we want to instantiate. Further, by compiling the interpreter to machine code for the target machine at hand, we have essentially compiled each user-defined function of the source program into target machine code. This is the idea behind the G-machine and our implementation of Lazy ML.

We want to avoid discussing details of the machine code of a particular target machine. The G-machine is an abstraction of the above idea of compiled graph reduction, and we will describe it in much the same way as the SECD machine.[25] Thus the instantiation of a right-hand side in the G-machine is described in terms of stack machine code; actual target machine code for doing this rarely needs the stack for temporary storage.

Figure 7 illustrates G-machine execution of the rewrite from (*square* (*add* 2 3)) to (*mul* (*add* 2 3) (*add* 2 3)) (reduction step (a) to (b) in Fig. 6).

Before the start of the reduction a pointer to the expression graph to be reduced is at the top of the pointer stack (a). Reduction is started by the execution of the G-machine instruction EVAL (e.g. by the print machinery, or another part of a computation needing the value of this expression). The EVAL instruction causes a new stack frame to be created, (b) saving the old one on another stack called the dump (not shown in Fig. 7). The G-machine now enters the UNWIND state in order to find the function node of the application and to push pointers to the apply nodes on the way, (c). To make arguments more easily accessible, the stack is then rearranged so that the topmost pointer points to the argument of *square*, the second pointer from the top is left untouched and will thus point to the apply node to be updated later. The G-machine now starts to execute the code for *square*, which is

```
square: PUSH 0; PUSH 1; PUSHGLOB mul; MKAP;
        MKAP; UPDATE 2; POP 1; UNWIND.
```

Except for the last three instructions, this is essentially a (reverse) postfix representation of the right hand side of *square*. The PUSH *m* instruction pushes the *m*th pointer of the stack relative to the top (the top has offset 0).* The PUSHGLOB mul instruction pushes a pointer to a FUN node for *mul*. MKAP constructs an application node with the two topmost pointers as subparts. Having constructed the graph of the right-hand side of the definition of *square*, the root node of this graph is copied on to the apply node of the original application, (j), having thereby rewritten (*square* (*add* 2 3)) to (*mul* (*add* 2 3) (*add* 2 3)) in the graph.

The final UNWIND instruction is to re-enter the unwind state. So the machine will here end up in the code for mul, which reduces the two arguments to their values, computes the product, updates the root of the application to an integer node with the product, restores the old stack from the dump, and returns from EVAL.

3.6 Some technical details of the G-machine

In this section we explicate the G-machine by defining a small language, give compilation rules to G-code for the language which performs repeated graph rewrites, and give semantics for the G-machine instructions.

The language we will give compilation rules for is shown in Fig. 8. It is a simple combinator language, i.e. a set of possibly mutually recursive functions. The right-hand side consists of parameters, functions, applications

* We could just as well have chosen to address stack entries via a base pointer pointing to, say, the stack entry pointing to the node to be updated; however, this would introduce other complications in the G-machine.

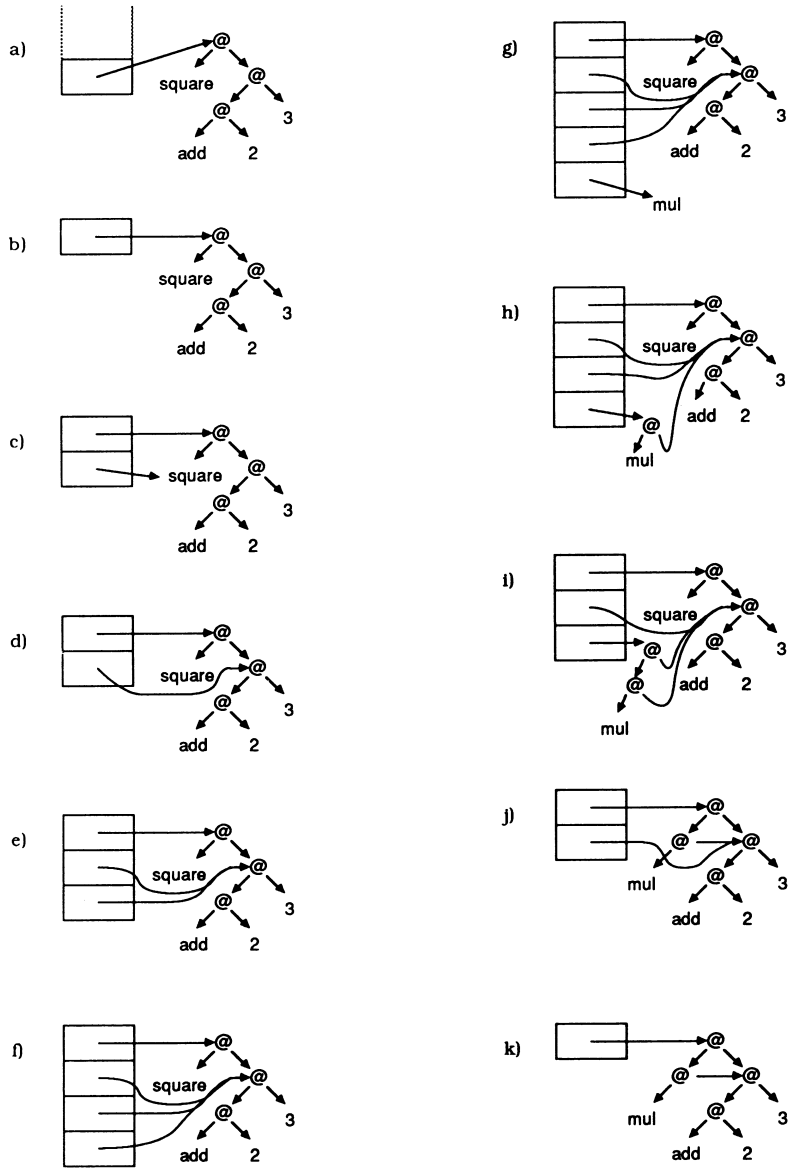

Figure 7. G-machine rewrite of *square (add 2 3)*.

and constructor values with constructor number k and m sub-expressions. Values in this language are (1) constructor values with a constructor number and a sequence of (unevaluated) subexpression, and (2) function values consisting of curried applications of the functions f_i with too few arguments. (For the time being we ignore the predefined operators *add*, etc.; see Section 3.7.)

The compilation will be described by the following three compilation schemes:

$\mathscr{F}[\![f\,x_1 \cdots x_m = e]\!]$ denotes the code sequence for an entire function definition.

$\mathscr{R}[\![e]\!]\,\rho\,d$ denotes the code for a right-hand-side e.

$\mathscr{C}[\![e]\!]\,\rho\,d$ denotes the code to construct the graph of e and leave the result on the top of the stack.

ρ is a mapping from names of parameters to their location on the stack, and d is the current depth of the stack. Thus the offset from the top of the stack for a variable x is given by $d-\rho(x)$. The compilation rules are given in Fig. 9. (There is only one case for $\mathscr{R}[\![\,]\!]$ in Fig. 9, but we shall elaborate $\mathscr{R}[\![\,]\!]$ a great deal later.)

$$
\begin{aligned}
p ::= \;& f_1\, x_1 \cdots x_{m_1} = e_1 \\
& \vdots \\
& f_n\, x_1 \cdots x_{m_n} = e_n \\
& e_0 \\
e ::= \;& x_i \mid f_i \mid e\ e \mid c_k\, e_1 \cdots e_m
\end{aligned}
$$

Figure 8. Syntax of compiled programs.

Fig. 10 shows the exact meaning of the instructions used in the compilation rules. A state in the G-machine is a tuple $\langle c, s, G, D\rangle$, where c is the currently executed code sequence, s is the pointer stack, G is the graph, i.e. a mapping from node names to nodes. Possible nodes are FUN $a\,c$, where a is the arity of the function and c its code; AP $n_1\,n_2$, where n_1 is a pointer (i.e. node name) to the function and n_2 to the argument; and CONSTR k $n_1 \cdots n_m$, where k is the constructor number and $n_1 \cdots n_m$ are pointers to its sub-expressions. D is the dump. [ ] denotes an empty code sequence or an empty stack.

The initial state of the G-machine for a specific program is shown in Fig. 11. It starts out with a code

$$
\begin{aligned}
\mathcal{F}[\![f\ x_1 \cdots x_m = e]\!] &= \mathcal{R}[\![e]\!][x_1 = m, \cdots x_m = 1]m \\
\mathcal{R}[\![e]\!]\,\rho\,d &= \mathcal{C}[\![e]\!]\,\rho\,d; \mathsf{UPDATE}\ (n+1); \mathsf{POP}\ n; \mathsf{UNWIND} \\
\mathcal{C}[\![f]\!]\,\rho\,d &= \mathsf{PUSHGLOB}\ f \\
\mathcal{C}[\![x]\!]\,\rho\,d &= \mathsf{PUSH}\ (n - r(x)) \\
\mathcal{C}[\![e_1 e_2]\!]\,\rho\,d &= \mathcal{C}[\![e_2]\!]\,\rho\,d; \mathcal{C}[\![e_1]\!]\,\rho\,(d+1); \mathsf{MKAP} \\
\mathcal{C}[\![c_k e_1 \cdots e_m]\!]\,\rho\,d &= \mathcal{C}[\![e_m]\!]\,\rho\,d; \cdots \mathcal{C}[\![e_1]\!]\,\rho\,(d+m-1); \mathsf{CONSTR}\ k\ m
\end{aligned}
$$

Figure 9. Compilation rules.

$$
\begin{aligned}
&\langle \mathsf{EVAL}.c, n.s, G[n = \mathrm{CONSTR}\ k\ n_1 \cdots n_m], D\rangle \Rightarrow \langle c, n.s, G[n = \mathrm{CONSTR}\ k\ n_1 \cdots n_m], D\rangle \\
&\langle \mathsf{EVAL}.c, n.s, G[n = \mathrm{FUN}\ a\ c], D\rangle \Rightarrow \langle \mathsf{EVAL}.c, n.s, G[n = \mathrm{FUN}\ a\ c], D\rangle \\
&\langle \mathsf{EVAL}.c, n.s, G[n = \mathrm{AP}\ n_1\ n_2], D\rangle \Rightarrow \langle \mathsf{UNWIND}.[], n.[], G[n = \mathrm{AP}\ n_1\ n_2], (c,s).D\rangle \\
&\langle \mathsf{UNWIND}.c, n.s, G[n = \mathrm{AP}\ n_1\ n_2], D\rangle \Rightarrow \langle \mathsf{UNWIND}.c, n_1.n.s, G[n = \mathrm{AP}\ n_1\ n_2], D\rangle \\
&\langle \mathsf{UNWIND}.c, n.n_1 \cdots n_k.[], G[n = \mathrm{FUN}\ a\ c_f], (c_r, s_r).D\rangle \Rightarrow \\
&\qquad \begin{cases} \langle c_f, \mathcal{A}(G, n_1). \cdots \mathcal{A}(G, n_a).n_a. \cdots n_k.[], G[n = \mathrm{FUN}\ a\ c_f], (c_r, s_r).D\rangle & \text{if } k \geq a \\ \qquad \text{where} \mathcal{A}(G[n = \mathrm{AP}\ n_f\ n_a], n) = n_a & \\ \langle c_r, n_k.s_r, G[n = \mathrm{FUN}\ a\ c_f], D\rangle & \text{if } k < a \end{cases} \\
&\langle \mathsf{UNWIND}.c, n.[], G[n = \mathrm{CONSTR}\ k\ n_1 \cdots n_m], (c_r, s_r).D\rangle \Rightarrow \langle c_r, n.s_r, G[n = \mathrm{CONSTR}\ k\ n_1 \cdots n_m], D\rangle \\
&\langle \mathsf{PUSH}\ m.c, n_0. \cdots n_m.s, G, D\rangle \Rightarrow \langle c, n_m.n_0. \cdots n_m.s, G, D\rangle \\
&\langle \mathsf{POP}\ m.c, n_1. \cdots n_m.s, G, D\rangle \Rightarrow \langle c, s, G, D\rangle \\
&\langle \mathsf{UPDATE}\ m.c, n_0.n_1. \cdots n_m.s, G[n_0 = N_0, n_m = N_m], D\rangle \Rightarrow \langle c, n_1. \cdots n_m.s, G[n_0 = N_0, n_m = N_0], D\rangle \\
&\langle \mathsf{MKAP}.c, n_1.n_2.s, G, D\rangle \Rightarrow \langle c, n'.s, G[n' = \mathrm{AP}\ n_1\ n_2], D\rangle \\
&\langle \mathrm{CONSTR}\ k\ m.c, n_1. \cdots n_m.s, G, D\rangle \Rightarrow \langle c, n'.s, G[n' = \mathrm{CONSTR}\ k\ n_1 \cdots n_m], D\rangle
\end{aligned}
$$

Figure 10. Semantics of instructions.

$$
\begin{aligned}
&\langle c_0, [], G_0, []\rangle \\
&\quad \textit{where}\ \ c_0 = \mathcal{C}[\![e_0]\!][]0; \mathsf{EVAL} \\
&\quad \textit{and}\ \ G_0 = \{\ n_{f_1} = \mathrm{FUN}\ m_1\ \mathcal{F}[\![f_1 x_1 \cdots x_{m_1} = e_1]\!] \\
&\qquad\qquad\qquad \vdots \\
&\qquad\qquad\ n_{f_n} = \mathrm{FUN}\ m_n\ \mathcal{F}[\![f_n x_1 \cdots x_{m_n} = e_n]\!] \\
&\qquad\ \}
\end{aligned}
$$

Figure 11. Initial state of the G-machine.

sequence to build and evaluate the graph of the value of the program e_0, an empty stack, a graph with function nodes one for each function f_i, and an empty dump.

If the value of e_0 is a constructor, on the top level the G-machine evaluates e_0 to constructor form, leaving the sub-expressions unevaluated. This is a simplification of presentation: one wants a real implementation to evaluate and print the entire data structure. It is straightforward to extend the G-machine with a PRINT instruction which evaluates and prints each sub-expression in turn.

The careful reader may have discovered a slight flaw in the above schemes. If the right-hand side is a variable only, the root of the graph that the variable stands for is *copied* onto the root of the redex – thus work is copied as well. To avoid this possibility the variable has to be evaluated before being copied. We will not remedy this now, the problem will go away in the next section in a grander scheme of things. (The problem can also be solved using indirection nodes.)

3.7 The predefined functions

So far the machinery exposed in Figs 8–11 does not cater for the predefined functions. So let us now discuss two important representatives of them: *add* and *if*.

First, let us consider the predefined function *add* of two curried arguments. *add* cannot be expressed as a simple graph rewrite, since *add* needs to evaluate its arguments. We would like to express *add* in G-machine code. To evaluate its two arguments we already have the EVAL instruction at our disposal. But we need to introduce an instruction ADD to perform the addition. Let us assume that integers are represented by CONSTR nodes with zero arguments, the integer value being the constructor number. The ADD instruction then computes the sum of the constructor numbers of the CONSTR nodes on the top of the stack, and constructs a CONSTR node with a constructor number being the computed sum, thus:

$$
\begin{aligned}
&\langle \mathsf{ADD}.c, n_1.n_2.s, G[n_1 = \mathrm{CONSTR}\ i_1, \\
&\qquad\qquad n_2 = \mathrm{CONSTR}\ i_2], D\rangle \Rightarrow \\
&\langle c, n'.s, G[n_1 = \mathrm{CONSTR}\ i_1, n_2 = \mathrm{CONSTR}\ i_2, \\
&\qquad n' = \mathrm{CONSTR}\ i_1 + i_2], D\rangle
\end{aligned}
$$

The predefined function *add* can now be expressed with the following G-machine code.

```
add: PUSH 0; EVAL; PUSH 2; EVAL; ADD;
     UPDATE 3; POP 2; UNWIND.
```

Let us now consider the predefined function *if*, of three arguments. *if* should reduce its first argument to either the value true or false; if true, the application should reduce to the value of the second arguments, and to the value of the third arguments otherwise. To express *if* in G-code, we introduce instructions for conditional jumping. The JFALSE *l* instruction pops the stack, and if the node pointed to is false (represented by CONSTR 0), it jumps to a position indicated by the LABEL *l* pseudo-instruction. The code for *if* can be written as follows.

```
if: PUSH 0; EVAL; JFALSE L1;
    PUSH 1; EVAL; UPDATE 4; POP 3; UNWIND;
    LABEL L1;
    PUSH 2; EVAL; UPDATE 4; POP 3; UNWIND;
```

3.8 By-passing graph reduction

The code emitted by the compilation schemes evaluates an application by repeatedly rewriting the graph expression until a normal form is reached. An evaluator using this mode of computation will be a heavy consumer of heap space. Thus there is a lot of speed to be gained by optimising away heap allocations. It turns out that we actually can bypass a considerable amount of the graph construction, and instead evaluate expressions in a more conventional manner.

Consider our *square*(1+2) example. The code for *square* rewrites the graph to the graph equivalent of *mul*(*add* 1 2) (*add* 1 2), where *add* 1 2 is shared. *square* then hands over the task of further reduction to *mul* by executing the UNWIND instruction. Hence execution continues with the code for *mul*, which reduces the shared expression to its value and updates the root of the redex with the value of the product, i.e. the square of (1+2).

In this case we could do much better by compiling *square* to code much like the code in the predefined functions. The code for *square* would then evaluate its argument, compute the square, and update the root of the redex with the value of the square. Thus it is not necessary to actually build the graph of a right-hand side; instead we may compile it to code to evaluate it to normal form directly. The code for *square* would then look like this:

```
square: PUSH 0; EVAL;      push and evaluate the
                           argument
        PUSH 1;            push the arg again; now
                           evaluated
        MUL                compute the square
        UPDATE 2;
        POP 1; UNWIND      update and return from
                           EVAL.
```

This motivates us to modify the compilation schemes, and to introduce the compilation scheme $\mathscr{E}[\![e]\!]\rho d$, which computes e to normal form and leaves a pointer to the result on top of the stack. The effect is the same as $\mathscr{C}[\![e]\!]\rho d$ followed by an EVAL, except that $\mathscr{E}[\![e]\!]\rho d$ executes faster and uses less heap space.

First we modify $\mathscr{R}[\![\ ]\!]$ to use $\mathscr{E}[\![\ ]\!]$ instead of $\mathscr{C}[\![\ ]\!]$:

$$\mathscr{R}[\![e]\!]\rho d = \mathscr{E}[\![e]\!]\rho d; \mathsf{UPDATE}\,(d+1); \mathsf{POP}\,d; \mathsf{UNWIND}$$

The compilation scheme $\mathscr{E}[\![\ ]\!]$ can be written as follows.

$$\begin{aligned}
\mathscr{E}[\![\mathit{if}\,e_1\,e_2\,e_3]\!]\rho d &= \mathscr{E}[\![e_1]\!]\rho d; \mathsf{FALSE}\,l_1; \mathscr{E}[\![e_2]\!]\rho d; \mathsf{JUMP}\,l_2;\\
&\quad \mathsf{LABEL}\,l_1; \mathscr{E}[\![e_3]\!]\rho d; \mathsf{LABEL}\,l_2;\\
\mathscr{E}[\![\mathit{add}\,e_1\,e_2]\!]\rho d &= \mathscr{E}[\![e_2]\!]\rho d; \mathscr{E}[\![e_1]\!]\rho(d+1); \mathsf{ADD}\\
\mathscr{E}[\![e_1\,e_2]\!]\rho d &= \mathscr{C}[\![e_1\,e_2]\!]\rho d; \mathsf{EVAL}\\
\mathscr{E}[\![x]\!]\rho d &= \mathscr{C}[\![x]\!]\rho d; \mathsf{EVAL}\\
\mathscr{E}[\![f]\!]\rho d &= \mathscr{C}[\![f]\!]\rho d\\
\mathscr{E}[\![c_k\,e_1\cdots e_m]\!]\rho d &= \mathscr{C}[\![c_k\,e_1\cdots e_m]\!]\rho d
\end{aligned}$$

The above optimisation constitutes an order-of-magnitude improvement in the efficiency of compiled programs! But in one respect the above optimisation actually performs worse than the original compilation schemes, namely in the usage of stack space at tail (recursive) calls. Consider, for example, what happens if the right-hand side is a function application:

$$f\,x = g\,x\,x.$$

With the original schemes, the code for f simply rewrites the graph of $f\,e$ into the graph of $g\,e\,e$, and the execution is then passed over to the code for g – this is a tail call to g. However, with the modified schemes, $\mathscr{E}[\![\ ]\!]$ builds the graph of $g\,e\,e$ and EVALuates it, before updating with the value – this is a 'call' to g which uses another stack frame (assuming the arity of g is two or less).

A similar thing happens if the right hand side is an if-expression (i.e, an application to ternary predefined *if*), e.g.,

$$f\,x = \mathit{if}\,e_1(g_1\,x\,x)\,e_2$$

With the modified compilation schemes, $\mathscr{E}[\![(g_1\,x\,x)]\!]\rho d$ is used which constitutes an ordinary 'call'. Using $\mathscr{R}[\![(g_1\,x\,x)]\!]\rho d$ instead would also in this case reinstate proper tail recursive behaviour.

To remedy this we must make $\mathscr{R}[\![\ ]\!]$ more selective, and not just call $\mathscr{E}[\![\ ]\!]$ in all cases right away. We thus modify $\mathscr{R}[\![\ ]\!]$ into the following:

$$\begin{aligned}
\mathscr{R}[\![\mathit{if}\,e_1\,e_2\,e_3]\!]\rho d &= \mathscr{E}[\![e_1]\!]\rho d; \mathsf{JFALSE}\,l'; \mathscr{R}[\![e_2]\!]\rho d;\\
&\quad \mathsf{LABEL}\,l'; \mathscr{R}[\![e_3]\!]\rho d;\\
\mathscr{R}[\![e_1\,e_2]\!]\rho d &= \mathscr{C}[\![e_1\,e_2]\!]\rho d; \mathsf{UPDATE}\,(n+1); \mathsf{POP}\,n;\\
&\quad \mathsf{UNWIND}
\end{aligned}$$

otherwise

$$\mathscr{R}[\![e]\!]\rho d = \mathscr{E}[\![e]\!]\rho d; \mathsf{UPDATE}\,(n+1); \mathsf{POP}\,n; \mathsf{UNWIND}$$

We shall consider one more improvement of the compilation schemes. The way the basic valued instructions ADD, etc. work at this stage when computing the value of an expression that requires basic valued intermediate results, a new cell is allocated each time for every intermediate result. This seems quite wasteful. The normal thing to do would be to put the intermediate result in a machine register or on the runtime stack. We want to mimic this behaviour in the G-code, in order to be able to generate more efficient target code from the G-machine code. To do this, we separate the operation of computing the value of the arithmetic operation from the operation of allocating a new cell and putting the basic value there. Consider a big arithmetic expression: we want to compute the value of the expression using register or stack temporaries, and put the result in the stack only when we are done. This is the motivation for introducing the compilation scheme $\mathscr{B}[\![e]\!]$, evaluate to basic value. But where shall we put the basic value result? We have three options: (1) put it on the pointer stack (the stack will then contain both pointers and basic values); (2) introduce a new stack in the abstract machine; (3) put the value on the dump.

For reasons of memory management and garbage collection in the implementation, we want to avoid alternative (1). Choosing between (2) and (3) is purely a matter of notational and descriptional convenience. The least modification to the G-machine as it stands will be to assume that the dump is used for this purpose. We thus change the ADD instruction into the following:

$$\langle \mathsf{ADD}.c, s, G, i_1.i_2.D\rangle \Rightarrow \langle c, s, G, (i_1+i_2).D\rangle$$

The complete set of modified compilation rules are given in Fig. 12.

A few words about some of the instructions in Fig. 12: JFALSE now pops a boolean value from the dump, and jumps if it is equal to 0. BONSTR pops an integer from

the dump and constructs a CONSTR node (with no arguments) with this constructor number. ADD adds values on the dump, as noted earlier. GET is the inverse of BCONSTR: it pushes the constructor number of the inspected CONSTR node on to the dump. PUSHBASIC pushes a constant onto the dump.

3.9 Various other optimisation techniques

There are a host of other techniques and optimisations used in the compiler – we mention some here in passing that relate to the G-machine code.

It is possible to compile a restricted form of **let** and **let rec** expressions directly into G-machine code. The restriction is that what is being defined must not be a function (i.e. no lambda expression must occur in the function bodies, all functions must be defined on the global level). Consider $\mathscr{R}[\![\textbf{let}\ x = e\ \textbf{in}\ e']\!]\rho d$. To compile this, we simply construct and push a pointer to the graph for e, as if it had been put there originally as a function argument:

$$\mathscr{R}[\![\textbf{let}\ x = e\ \textbf{in}\ e']\!]\rho d = \mathscr{C}[\![e]\!]\rho d; \mathscr{R}[\![e']\!]\rho[x = d](\delta+1)$$

Similar simple rules apply for the other compilation schemes. If the local definition is a **let rec** expression, a cyclic graph can be built for the recursive definition.[20]

Consider $\mathscr{R}[\![f e_1 e_2]\!]\rho d$, which according to the compilation rules in Fig. 12 builds the application, updates the root of the redex, and re-enters the unwind state. If f is a global function of two arguments, we can foresee what will happen: when unwind has reached the function node for f, the stack will be rearranged so that the three topmost pointers point to the graph for e_1, the graph for e_2, and the root of the redex. A better code sequence for $\mathscr{R}[\![f e_1 e_2]\!]\rho d$ would simply construct the graphs for e_1 and e_2, update the stack, and jump to the code for f – thus it is not necessary to construct the two apply nodes of the application $f e_1 e_2$. This optimisation is obviously valid when the function applied is a global one of known arity.

A similar improvement of $\mathscr{E}[\![f e_1 e_2]\!]\rho d$ can be done: first allocate a node for the result and push a pointer to it (it will play the rôle of the root of the redex), then construct and push pointers to the graphs for e_2 and e_1 in turn, then do a conventional call to f.

In some cases it is useful to do "peep-hole code improvement" on the G-machine code, by considering pairs of instructions and merging each pair into one. Consider the pair MKAP;UPDATE n. The MKAP instruction constructs an apply node, which is copied onto another node by UPDATE. This can be done more efficiently by an instruction UPDATEAP n, which would remake the updated node directly into an apply node with two pointers taken from the stack.

In the LML compiler this peephole code improvement is done in the target code generator, by letting it pattern-match on such pairs of G-code instructions, and generate target code directly for the G-code pair.

Similar optimisations can be done in many other cases.[22]

As we have seen, when building the graph for an application an apply node is built for each argument. This seems a rather wasteful way of storing the pointer to the arguments. A justification is of course that one of the apply nodes in the application is the root of the redex and thus gets updated with the value of the function application, but in general at compile-time we do not know which one (for instance when the function is a local variable). But very often we do know which apply node will be updated, namely when the application is of a global function of known arity. When the number of arguments is equal to the arity, our LML compiler builds a *vector apply node*. This restriction simplifies the target code involved immensely, while still covering a large

$$
\begin{aligned}
\mathcal{F}[\![f\ x_1 \cdots x_m = e]\!] &= \mathcal{R}[\![e]\!][x_1 = m, \cdots x_m = 1]m \\
\\
\mathcal{R}[\![if\ e_1\ e_2\ e_3]\!]\rho d &= \mathcal{B}[\![e_1]\!]\rho d; \text{JFALSE } l'; \mathcal{R}[\![e_2]\!]\rho d; \text{LABEL } l'; \mathcal{R}[\![e_3]\!]\rho d; \\
\mathcal{R}[\![e_1 e_2]\!]\rho d &= \mathcal{C}[\![e_1 e_2]\!]\rho d; \text{UPDATE } (n+1); \text{POP } n; \text{UNWIND} \\
\mathcal{R}[\![e]\!]\rho d &= \mathcal{E}[\![e]\!]\rho d; \text{UPDATE } (n+1); \text{POP } n; \text{UNWIND} \quad \text{otherwise} \\
\\
\mathcal{E}[\![x]\!]\rho d &= \mathcal{C}[\![x]\!]\rho d; \text{EVAL} \\
\mathcal{E}[\![f]\!]\rho d &= \mathcal{C}[\![f]\!]\rho d \\
\mathcal{E}[\![if\ e_1\ e_2\ e_3]\!]\rho d &= \mathcal{B}[\![e_1]\!]\rho d; \text{JFALSE } l_1; \mathcal{E}[\![e_2]\!]\rho d; \text{JUMP } l_2; \\
&\quad \text{LABEL } l_1; \mathcal{E}[\![e_3]\!]\rho d; \text{LABEL } l_2; \\
\mathcal{E}[\![add\ e_1\ e_2]\!]\rho d &= \mathcal{B}[\![add\ e_1\ e_2]\!]\rho d; \text{BCONSTR} \\
\mathcal{E}[\![e_1\ e_2]\!]\rho d &= \mathcal{C}[\![e_1 e_2]\!]\rho d; \text{EVAL} \\
\mathcal{E}[\![c_k\ e_1 \cdots e_m]\!]\rho d &= \mathcal{C}[\![c_k\ e_1 \cdots e_m]\!]\rho d \\
\\
\mathcal{B}[\![x]\!]\rho d &= \mathcal{E}[\![x]\!]\rho d; \text{GET} \\
\mathcal{B}[\![if\ e_1\ e_2\ e_3]\!]\rho d &= \mathcal{B}[\![e_1]\!]\rho d; \text{JFALSE } l_1; \mathcal{B}[\![e_2]\!]\rho d; \text{JUMP } l_2; \\
&\quad \text{LABEL } l_1; \mathcal{B}[\![e_3]\!]\rho d; \text{LABEL } l_2; \\
\mathcal{B}[\![add\ e_1\ e_2]\!]\rho d &= \mathcal{B}[\![e_1]\!]\rho d; \mathcal{B}[\![e_1]\!]\rho d; \text{ADD} \\
\mathcal{B}[\![e_1\ e_2]\!]\rho d &= \mathcal{E}[\![e_1\ e_2]\!]\rho d; \text{GET} \\
\mathcal{B}[\![c_k]\!]\rho d &= \text{PUSHBASIC } k \\
\\
\mathcal{C}[\![f]\!]\rho d &= \text{PUSHGLOB } f \\
\mathcal{C}[\![x]\!]\rho d &= \text{PUSH } (n - r(x)) \\
\mathcal{C}[\![e_1\ e_2]\!]\rho d &= \mathcal{C}[\![e_2]\!]\rho d; \mathcal{C}[\![e_1]\!]\rho(d+1); \text{MKAP} \\
\mathcal{C}[\![c_k\ e_1 \cdots e_m]\!]\rho d &= \mathcal{C}[\![e_m]\!]\rho d; \cdots \mathcal{C}[\![e_1]\!]\rho(d+m-1); \text{CONSTR } k\ m
\end{aligned}
$$

Figure 12. Improved compilation schemes.

number of common cases. If there are more arguments in a function application, the extra arguments are put in apply nodes as usual. Vector apply nodes are faster to build and consume less space, and UNWIND is very simple: just push the arguments of the node on to the stack and start executing the code.

According to the previous discussion, to construct the graph of an expression involving the primitive operators $+$, $\times$, etc., we build a graph with applications to the corresponding primitive functions; for example, to build the graph for the expression $x \times y + x$, we would then build the application $add\,(mul\,x\,y)\,x$. This has the effect that evaluation of this expression by EVAL causes evaluation to work in an 'interpretive mode', the evaluation machinery interpreting the graph to compute the value. This is hardly a very efficient evaluation method, compared to 'real code' to evaluate the same expression. However, there is a way to regain this efficiency to some extent, namely by wrapping the expression up in a function $f\,x\,y = add\,(mul\,x\,y)\,x$, and replace the arithmetic expression with $f\,x\,y$. Thus, instead of building a potentially large graph involving the primitive functions *add*, etc., the expression will now be represented by 'pure code' being executed when the value is demanded.

Combined with the vector apply node representation, things begin to look very much like conventional closures (i.e. code pointer–environment pointer pairs) except that what gets put in the environment are exactly the variables occurring in the expression.

3.10 M-code and target code

To make target-code generation easier, another intermediate code is used in the LML compiler. It is called M-code. M-code looks very much like machine code for modern machines such as VAX and MC68020. It uses a somewhat idealised instruction set and addressing modes, but it is essentially an ordinary machine. Producing target code from the M-code is relatively easy; for some machines (e.g. VAX) it is mostly a matter of pretty-printing the M-code. For other machines (e.g. Intel 80386) it involves some transformations to eliminate some instructions and addressing modes that are not available. Writing a code generator for a reasonable machine takes a few man-weeks. A large part of the run-time system (e.g. the garbage collector) is written in M-code. This achieves both speed and portability.

The M-code generator tries to avoid moving data around. The G-code for making, for example, an application node first moves the parts to the stack and then moves them into the new node. The M-code generator avoids this by keeping a simulated stack during code generation, and it emits code that moves the data to the right place at once. This is accomplished by an attribute grammar-based code generator, which is described in Ref 22 and 23.

3.11 Performance

The performance of the compiler itself is acceptable, despite the large number of passes. When the compiler compiles itself and all library functions, it reads some 17,000 lines* of LML code from 270 modules – this takes 70 minutes on a SUN3/180. This is about 250 lines/minute.

Table 1. Some performance figures

	LML	C	Pascal	SML	SML-NJ	Liszt	Miranda
8queens	6.7	1.3	3.8	75	9	47	549
fib 20	0.78	0.46	0.92	3.4	0.4	1.1	72
prime	0.23	0.2	—	2.1	0.38	1.1	12.3
kwic	1.4	—	0.9	—	—	—	—

Measuring performance of the compiled programs is very difficult, but to give an indication of the performance of the code produced by the LML compiler, Table 1 presents timings for some benchmark programs. All measurements were made on a VAX/780,** and are given in seconds of CPU time.

The following benchmark programs were used in the comparison in Table 1.

8queens	Counts the number of solutions to the 8 queens problem.
fib 20	Computes the 20th Fibonacci number.
prime	Computes all prime numbers up to 300.
kwic	Produces a permuted index of a number of titles. The Pascal version is written in an imperative style.

The implementations compared in Table 1 refer to the following:

LML	The LML compiler described in this paper.
C	The ordinary UNIX BSD 4.3 C compiler.
Pascal	The ordinary UNIX BSD 4.3 Pascal compiler.
SML	The Edinburgh SML compiler. Compiles into FAM code,[10] which is interpreted.
SML-NJ	The New Jersey SML compiler, version 0.18. Compiles into native code.
Liszt	Ordinary UNIX BSD 4.3 Franz LISP compiler.
Miranda	Miranda™ interpreter version 1.009.

4. EXPERIENCES FROM USING LML

Since the compiler produces normal executable UNIX programs it can be used to make "real" programs, i.e. programs that can be used by other people without any knowledge of the implementation language. For many applications the speed of the programs is quite adequate. By now a large number of non-trivial applications have been written in LML, both by us and others. We list some of them below.

(1) The LML compiler itself. All of the LML compiler, except the parser, is written in LML. The LML language has evolved hand in hand with the implementation, and very often the language has developed from perceived needs while programming the compiler. Examples of this are guards and the **local** construct. This project has taught us a great deal about functional programming – for instance, the attribute-grammar paradigm was dis-

* Actually, the compiler consists of some 6000 lines of source, but due to all the # includes of common type files the compiler has to read 17000 lines.

** The VAX timings are used, since this is a widely available machine.

covered while developing the lambda-lifting algorithm.[23]

(2) An LR(1) parser generator similar to Yacc, producing LML programs.[34] This was a major undertaking, and Uddeborg found his way into many hitherto undiscovered odd corners in the LML implementation. The major experience was that the absence of full laziness[19] can seriously damage your performance; more about this below.

(3) A simple real-time interactive video game. This program can be run in a heap of only 10 Kbytes, which gives several garbage collections per second, without any noticeable slowdown.

(4) A simulator for digital circuits using streams (cf. Figs 1–3), simulating a simple microprogrammed CPU, displaying the signal output in graphical form. The program manages to simulate several time steps per second.

(5) A simple object-oriented picture editor (a very stripped-down version of the Macintosh MacDraw program). The graphics is done by outputting Display Postscript to a NeWS™ graphics interface program for the SUN workstation. The major performance hurdle here turned out not to be LML, but the graphics interface.

(6) A Go-playing program (it plays a poor game of Go, of course, but so do all computer programs). The main reason for doing this in LML was to test different playing strategies – speed was of secondary concern.

(7) A theorem prover for minimal logic.

As almost anyone going from an untyped to a (polymorphically) typed functional language has experienced, the type system is a real help when writing even the smallest program. But when there is a type error, it is very difficult to give sensible error messages – the place where unification failed the first time is usually not the place of the error. This is a general problem in all implementations of languages that use type inferencing. This problem is further accentuated in the LML implementation: because of the many transformation steps in the compiler, it is difficult to relate the program fragment shown in the error message (pretty-printed abstract syntax tree) to the source program. This applies not only to type errors but to all kinds of errors. So far we have not spent much effort here, so there is ample room for improvement.

Sometimes compilation may take an unexpectedly long time. One possible cause for this is the polymorphic type checking, which seems to have difficulties with types with a large number of constructors, or a large number of type variables, or both. The functional parser generator (FPG) quite often generates programs with such "difficult" types.

Another situation where compilation may take a very long time is with functions defined using a fair number of equations, but where the patterns in the equations are complex. The pattern-matching transformation phase in the compiler expands such programs considerably (see Section 2.1), and it is the further compilation of this big program that takes time. An example of this is the G-code to M-code generator in the compiler.

Previously, we had thought that the issue of full laziness[19] was not particularly important, since it would always be possible to rewrite a non-fully lazy program to a fully lazy one, and it would not be very difficult to see where this was needed. When developing the functional parser generator (FPG),[34] Uddeborg found that this was not the case, and that loss of full laziness can be damaging. After long and painful searching in the program, he finally found three places where full laziness was lost, each of which accounted for an order of magnitude in time (and it is not guaranteed that all such places were found). Hence we are now of the opinion that a transformer to full laziness is an essential part of a serious compiler.

If a program does not run fast enough it can be quite difficult to find out why. One reason for this is that it is much more difficult to get a good operational understanding of a lazy functional program than of a strict functional one or an imperative one. This problem is emphasised further by the fact that the current LML implementation lacks the full laziness property (see above), which means that some expressions get re-evaluated in a non-intuitive way. There is also a lack of tools for analysing program behaviour; the usual UNIX tools for profiling programs, like "prof", do not work so well in a lazy evaluation context, or with higher-order functions. When programming in a style making much use of the predefined higher-order functions, like *map*, *reduce*, etc., the profiler may well say that most of the time is spent in *map* or *reduce* – hardly a big help when trying to pinpoint the bottlenecks in one's program.

Even though the compiler is not very fast, the separate compilation facility makes it possible to write large programs. The very simple machinery for modules and separate compilation, with text-file inclusion of compiler-generated type signature files, is for most cases quite adequate. On rare occasions, however, the need has been felt for something more substantial, that allows recursive modules. On the wish-list we have a somewhat more elaborate linking step, which performs the final unification of the types of imported and exported identifiers. This would remove the need to compile modules in a specific order because of the inclusion ordering.

A general problem with purely functional languages is that 'tracing' and debugging can be quite difficult. In programs in conventional languages one can insert print statements at interesting places to get a view of what is happening. With a purely functional language this is not possible (since they have no notion of statements and side-effects). At one time, we used a 'dirty' function called *trace* of two arguments which returns the value of the first argument and as a side-effect prints the value of the second argument. However, it generally turned out to be very difficult to decipher the output from this, since quite often (due to lazy evaluation) the evaluation by *trace* of its arguments caused other instances of *trace* to be evaluated. The result was generally an indecipherable mish-mash of output from different instances of *trace*.

This is an interesting and important problem area worth of much more attention.

5. RELATED WORK

There have been several attempts at speeding up graph reduction by supporting the G-machine more directly in hardware, through microcoding,[18] or as special-purpose VLSI processor designs.[24,35] However, the abstract G-machine being geared towards conventional machines, it seems eminently difficult to beat the von Neumann

processor architecture at its own game! Vasell reports a projected speed-up of about a factor of two using a special-purpose VLSI G-machine architecture.[35]

The G-machine has evolved considerably since its original conception, and a number of variants of the G-machine have been developed both by us and by others. We will briefly mention some of them here.

Graph reduction implies that when the value of a function application has been computed, the apply node being the root of the redex should be updated with the value. However, updating is only useful when the value may be reused, i.e. if the node is shared in the graph. In the *spineless G-machine*[7] needless updating is avoided by distinguishing between two kinds of apply nodes, *shared apply nodes* (SAPs), which should be updated with the value of the application for which the SAP is the root, and normal apply nodes (APs), which cannot be shared but just serve as placeholders for arguments. This arrangement greatly reduces the transfer of data between the pointer stack and the heap, as reduction of a function application proceeds entirely on the stack until the number of arguments on the stack is less than the arity of the function applied in the current term, at which point the updating of the SAP node is done.

A further refinement is the *spineless tagless G-machine*, where there is only one kind of node, a multi-argument apply node. It consists of a pointer to the code of the function being applied, and some arguments applied to it. The code for the function reduces the application to its value, updates the apply node, *and* pushes the constituent parts on to the stack if the value is a constructor. This representation is used also for nodes representing values, e.g. constructor nodes – here the code just pushes the constituent parts of the constructor on to the stack and returns. This is very similar to the evaluation paradigm of TIM.[15]

A potentially serious problem with the use of lazy functional languages is that of *space leaks*.[36] This problem comes from the appearance of graphs for expressions like $snd(e_1, e_2)$, $tl(e_1, e_2)$, etc.; although the expressions e_1 will never be used, the graph for them nevertheless occupies space. In general, the problem might appear with application to any function where some part or all of an argument will never be used.

This problem is attacked with the *Stingy G-machine*.[9] The general approach is to compile two pieces of code for each function: the normal evaluation code, used for evaluation of an application and the *stingy code*, used for building the graph of an application. This code may be called in two circumstances: when building the graph of an application, and during garbage collection. Consider the (built-in) function *tl*. To build the graph for the application $tl\,x$, push x and call the stingy code for *tl*. This code examines the argument, and if it is evaluated to canonical form *tl* returns the tail part, otherwise *tl* builds the graph of the application and returns a pointer to it. The stingy code may also be called during garbage collection to reduce applications of the above form.

In general, the stingy code looks just like the normal code, except that where the normal code would do EVAL, the stingy code does TEVAL *l*, which jumps to location label *l* if it is not evaluated, to build the application. To avoid non-termination the stingy code may have to give up at other points as well.

As the G-machine has been successful as an evaluation model for sequential execution of lazy function languages, it is natural to try to use the same techniques for parallel execution on multiprocessor computers. Researchers at several places are engaged in such work, among them Burn at GEC Hirst Research Centre,[8] and ourselves. Peyton Jones at University College London with the GRIP architecture has contemplated various evaluation models for GRIP,[12,28] among them a parallel G-machine. Below we briefly describe our own approach.

It would be possible to design an abstract parallel G-machine by simply extending the sequential G-machine to allow multiple threads of control – this has been done, for example, by Augustsson *et al.*[2] The sequential G-machine implementation (see Ref. 22) has two stacks: the dump and the pointer stack). Thus, in a parallel version of this machine, we are faced with the (hardly appealing) prospect of having to implement management of an arbitrary number of big stack pairs, one stack pair per evaluation process! Yet in the heap management for the graph we already have a general and efficient memory-allocation machinery. Therefore the thought is not far off of allocating the stack segments in the heap. But rather than allocating arbitrarily large stack nodes, it seems more sensible to allocate 'stack frames', i.e. small stacks for single activations of EVAL. This is the basic idea behind the $\langle v, G\rangle$-machine. In a parallel implementation of the $\langle v, G\rangle$-machine, multiple processors successively rewrite a frame node, and as the last step in a computation the frame node becomes a node for a constructor value. Although it is potentially more expensive to allocate frames from the heap than to re-use stack space, this cost appears to be cancelled by not having to move function arguments from apply nodes to the stack (the 'unwind' and 'rearrange' in the conventional G-machine).

A first implementation of the parallel $\langle v, G\rangle$-machine is now up and running on our Sequent Symmetry™, a 16-processor shared-memory multiprocessor. Although this first implementation has a number of rough edges, initial benchmark tests show that real and substantial speed-up over a sequential implementation should be fairly easy to achieve with this technique.

There are now several compilers for lazy functional languages that use techniques related to the one described in this paper.

Fairbairn and Wray have, independently of us, developed techniques which are very similar to ours.[14] In the Ponder compiler, super-combinators are compiled into code for the *Ponder Abstract Machine*, which is quite similar to the G-machine.

Hudak and Kranz[17] have implemented a compiler where they use SKI combinators as a convenient intermediate form for program optimisation. The optimised combinator expression is then converted back to lambda expression form, which is further compiled to machine code using a Scheme compiler, using environment-based techniques.

It is also interesting to compare the representation of the graph using vector application nodes to that of a closure-based implementation, such as FAM.[10] A closure consists of a pointer to code and a pointer to an environment. A compact representation of this is a node with a pointer to the code and pointers to all variables used in that code. This is exactly the same as the compact

representation of the vector application described earlier. The graph reduction and the closed-based implementation, in spite of their conceptual differences, end up being quite similar.

6. CONCLUDING REMARKS

A decade ago it was thought that radically new architectures were needed to execute functional languages, not only for taking advantage of the inherent parallellism is these languages, but even for efficient serial execution. What our research has shown is that the conventional von Neumann architecture is not so bad after all for executing functional languages. As mentioned in the introduction, our aim has been to make lazy functional languages practical on today's computers. We believe that we've come a long way towards that goal.

Parallel computers, consisting of dozens to hundreds of conventional processors connected to a message network or a shared memory, are now becoming available in the marketplace. As shown by Peyton Jones' paper in this issue, there is good evidence that these machines will be well suited for executing functional languages in parallel. There is also evidence that the techniques we have developed to improve efficiency on von Neumann architectures will still be relevant for many of the new parallel architectures.

Acknowledgements

We wish to thank Staffan Truvé and Phil Wadler for many helpful comments on earlier drafts of this paper. We also thank Göran Uddeborg for implementing the functional parser generator (FPG), and for many interesting remarks on LML and its implementation. The Swedish Board of Technical Development (S.T.U.) provided financial support for a large part of the research reported in this paper.

REFERENCES

1. L. Augustsson and T. Johnsson, *Lazy ML User's Manual*. Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology Göteborg (1987). (To be distributed with the LML compiler.)
2. L. Augustsson, C. Nilsson and S. Truvé, *The PG-machine: a Machine for Parallel Graph Reduction*. Technical Report Memo 57, Department of Computer Sciences, Chalmers University of Technology, Göteborg (1988).
3. Arvind, *A Data Flow Architecture with Tagged Tokens*. Technical Report, MIT, Cambridge, Massachusetts (1980).
4. L. Augustsson, Compiling pattern matching. In *Proceedings, 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy, France*.
5. L. Augustsson, Compiling Lazy Functional Languages, Part II. *PhD thesis*, Department of Computer Science, Chalmers University of Technology, Göteborg (1987).
6. R. M. Burstall, D. B. McQueen and D. T. Sannella, Hope: an experimental applicative language. In *Proceedings, 1980 ACM Symposium on Lisp and Functional Programming, Stanford, CA*, pp. 136–143.
7. G. Burn, J. Robson and S. Peyton Jones, The spineless G-machine. In *Proceedings, 1988 ACM Symposium on Lisp and Functional Programming, Snowbird, Utah*.
8. G. L. Burn, A shared memory parallel G-machine based on the evaluation transformer model of computation. In *Proceedings, Workshop on Implementation of Lazy Functional Languages, Programming Methodology Group, Göteborg*, (to appear).
9. C. von Dorrien, Stingy evaluation. *Licentiate thesis*, in preparation (1989).
10. L. Cardelli, The functional abstract machine. *Polymorphism: The ML/LCF/Hope Newsletter*, **1** (1) (1983).
11. T. J. W. Clarke, P. J. S. Gladstone, C. D. Maclean and A. C. Norman, SKIM: the S, K, I reduction machine. In *Proceedings, 1980 Lisp Conference, Stanford, CA*.
12. C. Clack and S. L. Peyton Jones, The four-stroke reduction engine. In *Proceedings, 1986 ACM Conference on Lisp and Functional Programming*, pp, 220–232.
13. J. Darlington, Alice: a multi-processor reduction machine for the parallel evaluation of applicative languages. In *Proceedings, 1981 Conference on Functional Languages and Computer Architecture, Wentworth-by-the-Sea, Portsmouth, NH*.
14. J. Fairbairn and S. C. Wray, Code generation techniques for functional languages. In *Proceedings, 1986 ACM Symposium on Lisp and Functional Programming, Cambridge, MA*.
15. J. Fairbairn and S. C. Wray, TIM: a simple, lazy abstract machine to execute supercombinators. In *Proceedings, 1987 Conference on Functional Programming Languages and Computer Architecture, Portland, OR*. 1987.
16. M. Gordon, R. Milner and C. Wadsworth, *Edinburgh LCF*. Volume 78 of Lecture Notes in Computer Science no. 78. Springer, Heidelberg (1979).
17. Paul Hudak and D. Kranz, A combinator-based compiler for a functional language. In *Proceedings, 11th ACM Symposium on Principles of Programming Languages*, pp. 122–132 (1984).
18. J.-M. Hauttecoeur and D. Suk, *Microcoded G-Machine under Unix 4.2*. Report 42, Department of Computer Engineering, Chalmers University of Technology, Göteborg (1987).
19. R. J. M. Hughes, Super combinators – a new implementation method for applicative languages. In *Proceedings, 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, PA*, pp. 1–10.
20. T. Johnsson, Efficient compilation of lazy evaluation. In *Proceedings, SIGPLAN '84 Symposium on Compiler Construction, Montreal*, pp. 58–69.
21. T. Johnsson, Lambda lifting: transforming programs to recursive equations. In *Proceedings, 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy*. Springer, Heidelberg.
22. T. Johnsson, Code generation from G-machine code. In *Proceedings, Workshop on Graph Reduction, Santa Fe, NM, 1986*. Springer, Heidelberg.
23. T. Johnsson, Attribute grammars as a functional programming paradigm. In *Proceedings, 1987 Conference on Functional Programming Languages and Computer Architecture, Portland, OR*. Springer, Heidelberg.
24. R. B. Kieburtz, The G machine: a fast graph-reduction evaluator. In *Proceedings, 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy*, pp. 400–413.
25. P. J. Landin, The mechanical evaluation of expressions. *The Computer Journal*, **6** (4), 308–320 (1964).
26. Robin Milner, A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, **17**, 348–375 (1978).
27. David Park, The 'fairness' problem and nondeterministic computing networks. In *Foundations of Computer Science*,

vol. IV (2), 133–161. Mathematical Centre Tract 159, Amsterdam (1983).
28. S. L. Peyton Jones, GRIP: A parallel graph reduction machine. In *Proceedings, 1987 Conference on Functional Programming Languages and Computer Architecture, Portland, OR.*
29. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ (1987).
30. D. A. Turner, *SASL Language Manual.* Technical report, University of St. Andrews (1976).
31. D. A. Turner, A new implementation technique for applicative languages. *Software – Practice and Experience*, **9**, 31–49 (1979).
32. D. A. Turner, *KRC Language Manual.* Technical report, University of Kent at Canterbury (1981).
33. D. A. Turner, Miranda: a non-strict language with polymorphic types. In *Proceedings, 1985 Conference on Functional Programming Languages and Computer Architecture, Nancy*, pp. 1–16.
34. G. Uddeborg, A functional parser generator. *Licentiate Thesis*, Chalmers University of Technology, Göteborg (1988).
35. J. Vasell, RISC-*implementing av G-maskinen*. Report 302-86/87, Department of Computer Engineering, Chalmers University of Technology, Göteborg (in Swedish) (1987).
36. P. Wadler, Fixing some leaks with a garbage collector. *Software Practice and Experience* (September 1987).
37. P. Watson and I. Watson, Evaluating functional programs on the FLAGSHIP Machine. In *Proceedings, 1987 Conference on Functional Programming Languages and Computer Architecture, Portland, OR*, pp. 80–97.