Efficient Compilation of Lazy Evaluation

Thomas Johnsson

Abstract

This paper describes the principles underlying an efficient implementation of a lazy functional language, compiling to code for ordinary computers. It is based on combinator-like graph reduction: the user defined functions are used as rewrite rules in the graph. Each function is compiled into an instruction sequence for an abstract graph reduction machine, called the G-machine, the code reduces a function application graph to its value. The G-machine instructions are then translated into target code. Speed improvements by almost two orders of magnitude over previous lazy evaluators have been measured; we provide some performance figures.

1 Background

Functional programming is emerging as an alternative to the conventional imperative style of programming [Lan66], [Bac78]. Lazy evaluation (call by need, normal order evaluation) has been proposed as a method for executing functional programs, the advantages being, among others, that unbound data structures, e.g. infinite lists, can easily be handled, and further that it makes interactive input/output possible in functional programs [Fri76]. Though functional programming languages have many pleasing properties, an obstacle to their wider use has been the lack of efficient implementations.

Our work is based on Turner's combinator approach [Tur79], where programs are transformed into expressions containing the combinators S, K, I etc from combinatory logic, thus removing all variables from the program. A combinator expression is evaluated in the 'SKI-machine' using normal order graph reduction. A problem with combinators is that each combinator defines rather a small interpretative step, and combinator expressions have a tendency to become very cumbersome for non-trivial programs.

Our lazy evaluation method is similar to the combinator reduction regime, but instead of using a standard, fixed set of combinators, each user defined function is used as a 'combinator', i.e., a rewrite rule for the graph. Functions are compiled into code sequences for an abstract graph reduction machine, called the *G-machine*, with instructions that explicitly construct and manipulate expression graphs to reduce expressions to their values; both shared and cyclic graphs can be directly constructed. Target code generation for ordinary computers from the G-machine code is rather straight-forward. One might say that the compiler constructs a specialised, machine-language coded combinator interpreter from each program.

2 Graph reduction

In our graph reduction approach a program is an expression whose value will appear as, in general, a stream of basic values (integers, booleans etc) on the output file. Expressions are evaluated using normal order graph reduction, and is carried out by performing transformations on the graph to reduce it to its *canonical form*. A canonical form is an expression which cannot be further reduced on the outermost level (even though subexpressions may be further reducible). In this paper canonical forms are integer and boolean constants, list expressions $e_1.e_2$ with arbitrary expressions e_1 and e_2, and function applications $f\ e_1 \cdots e_m$ where f is a function that takes more than m curried arguments; a reduction of an application can take place only if all curried arguments to the functions are present. Thus in general for an expression to become completely reduced, subexpressions must also be reduced to canonical form, for instance the elements of a list. Evaluation of a function application amounts to using the corresponding function definition as a graph rewrite rule, repeatedly rewriting the application graph to an instance of the right hand side of the function definition, with arguments substituted for formal parameters, until having reached a canonical form.

For illustration, consider the following functional program, its value being the infinite list of natural numbers.

$$\textbf{letrec}\, \mathit{from}\ n = n.\mathit{from}(\mathit{succ}\ n)\ \textbf{in}\ \mathit{from}\ 0$$

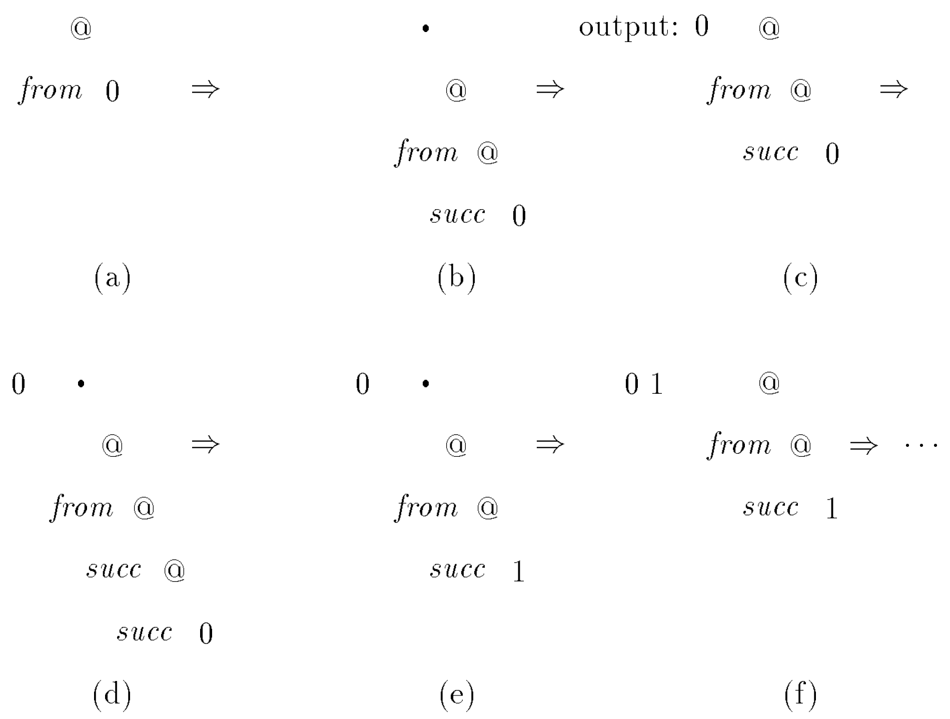

Figure 1: Graph reduction of *from* 0. The output is shown to the left of each graph.

succ is the predefined successor function, '.' is the infix list construction operator and juxtaposition denotes function application. Graph reduction of this program is shown in figure 1, In the figures function application is denoted by @.

The start expression 1(a) is transformed to 1(b) using the rewrite rule for the function *from* as defined above, with a pointer to the integer 0 substituted for the parameter n. In 1(b) the expression is on canonical form, and so is also its head part 0. The head value can now be output and dropped from the graph, 1(c). Again the the rewrite rule for *from* is applied to the graph, 1(d), and is now on the form $e.e'$, which is canonical. The next step is to reduce the head part *succ* 0 to its canonical form using the rewrite rule for *succ*, 1(e). The resulting integer 1 is then written on output and dropped from the graph.

The execution continues in this way ad infinitum. Note that the shared expression *succ* 0 has been replaced in 1(e) by its value. In general expression graphs are evaluated, i.e., reduced to their canonical forms, at most once and all expressions that share a particular subexpression benefit from the evaluation (call by need).

3 An introductory example of G-machine execution

In our graph reduction scheme each function definition is compiled into a sequence of G-machine instructions. Each graph rewrite, according to a function definition, is carried out by executing the code for that function. We here illustrate execution of the G-machine with the reduction step (c)–(d) from figure 1. The G-machine state transitions are shown in figure 2.

Before the start of the reduction a pointer to the expression graph is at the top of a pointer stack, figure 2(a) (the stack grows downwards). Reduction is started by execution of the G-machine instruction EVAL, in this case by the print mechanism. EVAL causes a new stack frame to be created with the previously topmost pointer as its single entry, saving the old stack on another stack called the dump (not shown in figure 2), then an unwind state pushes pointers to the application nodes of the left 'backbone' until having reached a function node, (a) – (c). The stack is then rearranged so that the topmost pointer of the stack points to the argument of *from*, the second pointer from the top is left untouched and will thus point to the apply node which is to be updated by the code with the result of the application. The G-machine now starts to execute the code for *from*, which is (see section 4.2 and table 3 how we obtain this)

```
from:  PUSH 0; PUSHFUN from;
       PUSHFUN succ; PUSH 3; MKAP;
       MKAP; CONS; UPDATE 2; RET 1.
```

Except for the last two instructions, this instruction sequence is essentially a postfix representation of the right hand side of *from*. The PUSH m instruction pushes the mth pointer of the stack relative to the top and starting with 0; note that different offsets have to be used to push pointers to the formal parameter m, depending on the current depth of the stack (the reason for this is explained in section 5.5). The PUSHFUN succ instruction pushes a pointer to a *succ* function node. MKAP constructs an application node with the to topmost as subparts; similarly for CONS. After having constructed the graph for the right hand side of the definition of *from*, figure 2(k), the cons node is copied onto the result apply node by the UPDATE 2 instruction, having thereby transformed $\mathit{from}(\mathit{succ}\,0)$ to $(\mathit{succ}\,0).\mathit{from}(\mathit{succ}(\mathit{succ}\,0))$ in the graph, which is a canonical expression. The RET 1 instruction pops one element from the stack, and since the top graph is now on canonical form, the old stack is restored from the dump and control is returned to the instruction following EVAL. In general, had the top graph been not on canonical form but an application node or a function node, instead of restoring the old stack and returning the G-machine would have reentered the unwind state to continue the reduction of the new expression graph.

4 Short-circuiting graph reduction

We have previously indicated that we do graph reduction by repeatedly rewriting the graph to the right hand sides of functions. Indeed we can use G-machine code that does precisely this; in most cases, however, we can take considerable shortcuts an do away with many intermediate graph rewritings.

Consider the function definition $\mathit{succ}\,n = n + 1$. If we compile it into code that constructs the graph for the right hand side, $\mathit{add}\,n\,1$, then when executed the expression graph $\mathit{succ}\,e$ will be rewritten into $\mathit{add}\,e\,1$, thus leaving over the task of further reduction to *add*, which will reduce the expression to its integer value. Much efficiency can be gained if we compile *succ* into code that first reduces its parameter n, computes the value of $n + 1$, and then remakes the apply node to a integer node with this value. This avoids the construction of the intermediate graph $\mathit{add}\,e\,1$. A code sequence for the function *succ* is accordingly

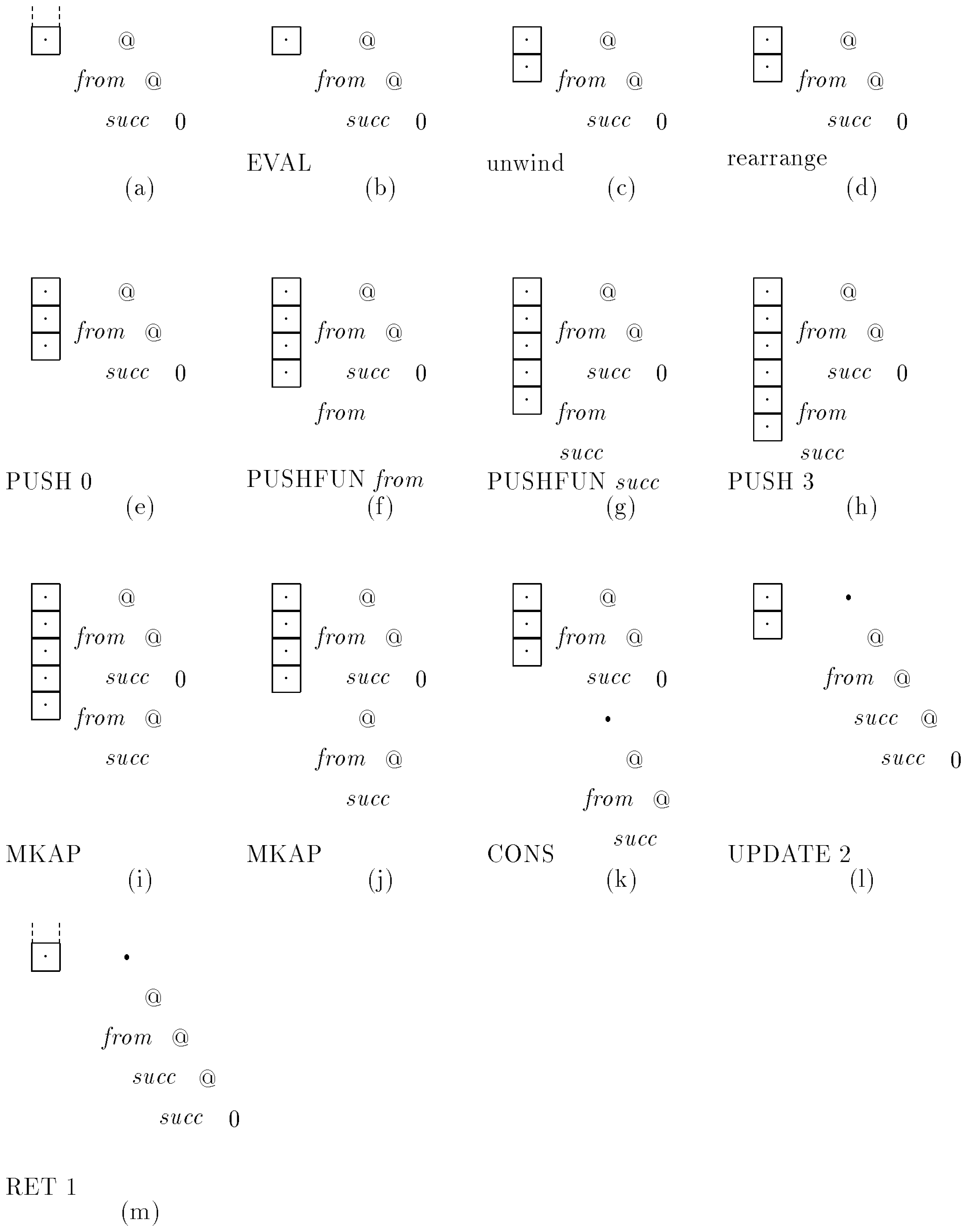

Figure 2: G-machine reduction of *from* (*succ* 0).

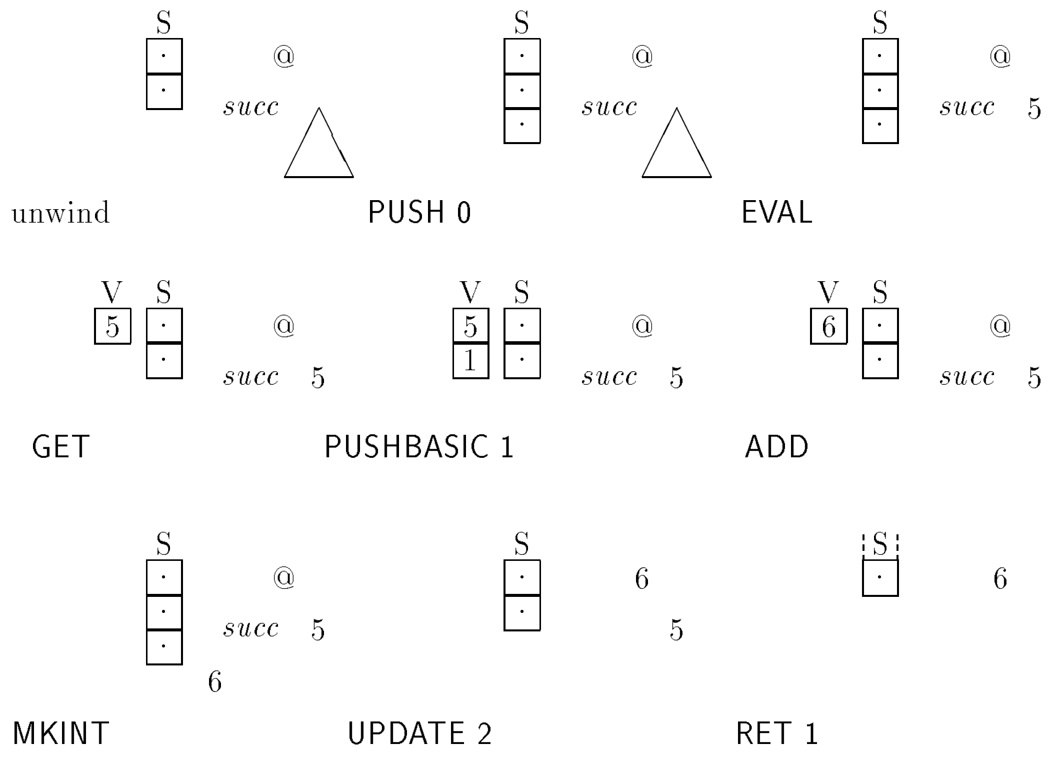

Figure 3: Shortcut evaluation of function *succ*.

```
succ:  PUSH 0; EVAL; GET; PUSHBASIC 1; ADD;
       MKINT; UPDATE 2; RET 1.
```

The execution of this code sequence is shown in figure 3. The addition is done on a separate stack for basic values, called V, with instructions MKINT and GET for transfering values to and from the graph. PUSHBASIC pushes a basic value constant on the V stack.

Similar reasoning can be applied to all other predefined primitive functions; if the right hand side is an **if**-expression, for example, then the code would do the following: compute the value of the condition, and if true the proper apply node is to be updated with the value of the **then**-expression, else updated with the value of the **else**-expression.

This line of reasoning is systematized in the next section by having different compilation schemes, one giving code that computes the value of an expression, and one giving code that constructs the graph of an expression. This more direct method is significantly faster; in our compiler implementation we have measured a speedup of about a factor of ten for some typical programs, compared to naive graph reduction.

5 Technical details of the abstract machine and compiler

In this section we give a complete set of compilation rules for a simple functional language, compiling to G-machine code. We also give an abstract description of the G-machine, describing the the effects of the G-machine instructions on a machine state.

5.1 Source Language

A program in the language described here consists of a set of recursive functions and an expression whose value is the value of the program, as summarised in table 1. Normal order evaluation is assumed. Each function f_i takes n_i curried arguments and the free variables of e_i are in the set $\{x_1 \cdots x_{n_i}\}$. Operators $+$, $-$ etc are viewed as syntactic sugar for applications to predefined functions *add*, *sub* etc, of which we deal with the ones given in table 2.

Table 1: **Syntax of programs**

$$
\begin{array}{lll}
\textit{program} ::= & f_1 x_1 \cdots x_{n_1} = e_1 & \text{(function definitions)} \\
 & \vdots & \\
 & f_m x_1 \cdots x_{n_m} = e_m & \\
 & e_0 & \text{(the value of the program)}
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & \textit{identifiers} \mid \textit{constants} \mid e\ e \\
 & \mid & \textbf{let}\ x_1 = e_1\ \textbf{and} \cdots \textbf{and}\ x_m = e_m\ \textbf{in}\ e \\
 & & \qquad \text{(multiple simultaneous local definitions)} \\
 & \mid & \textbf{letrec}\ x_1 = e_1\ \textbf{and} \cdots \textbf{and}\ x_m = e_m\ \textbf{in}\ e \\
 & & \qquad \text{(multiple simultaneous local recursive definitions)}
\end{array}
$$

Table 2: **Predefined functions**

add sub mul div	(binary arithmetic operators)
neg	(unary negation)
lt le eq ne ge gt	(binary relational operators)
and or	(conditional and, or)
not	(logical negation)
cons	(binary list construction)
hd tl	(unary head and tail of a list)
null	(unary test on empty list)
if	(ternary if-then-else)

Note that there is no lambda expression in the syntax of expressions, functions are defined only globally. Functional programs with local function definitions and lambda expressions with free variables can be transformed into the form above, using *super combinators* [Hug82]; an algorithm to the same effect is used in our compiler implementation, however, the program resulting from our transformation does not exhibit 'full laziness', as is the main issue in Hughes' work.

5.2 Compilation rules

The abstract compiler given in table 3 is subdivided into 4 compilation schemes:

$\mathcal{F}[\![f\ x_1 \cdots x_m = e]\!]$ gives the code for a function which reduces the graph of an application to canonical form.

$\mathcal{C}[\![e]\!]\ r\ n$ gives code that constructs the graph of e and leaves a pointer to the result on the top of the stack.

$\mathcal{E}[\![e]\!]\ r\ n$ gives code that computes the value, i.e. canonical form, of e and leaves a pointer to the value on the top of the stack. It yields the sames result as $\mathcal{C}[\![e]\!]\ r\ n$ followed by an EVAL instruction and embodies the short-circuiting described in section 4.

$\mathcal{B}[\![e]\!]\ r\ n$ computes the basic value of e and leaves the result on the basic value stack V, yielding the same result as $\mathcal{E}[\![e]\!]\ r\ n$ followed by a GET instruction. The idea behind $\mathcal{B}$ is to avoid construction of a new node for each intermediate result in an arithmetic or logical expression. The value is transferred to the graph only when the entire expression has been evaluated, by the MKINT or MKBOOL instruction.

Table 3: **Compilation rules**

$\mathcal{F}[\![f\ x_1 \cdots x_m = e]\!] = \mathcal{E}[\![e]\!]\ r\ (m+1)$; UPDATE $(m+1)$; RET m,
where $r = [x_1 = m+1,\ x_2 = m, \cdots, x_m = 2]$

Scheme $\mathcal{E}$: Evaluate

1.	$\mathcal{E}[\![i]\!]\ r\ n$	= PUSHINT i
2.	$\mathcal{E}[\![b]\!]\ r\ n$	= PUSHBOOL b
3.	$\mathcal{E}[\![nil]\!]\ r\ n$	= PUSHNIL
4.	$\mathcal{E}[\![x]\!]\ r\ n$	= PUSH $(n - r(x))$; EVAL
5.	$\mathcal{E}[\![f]\!]\ r\ n$	= PUSHFUN f
6.	$\mathcal{E}[\![add\ e_1\ e_2]\!]\ r\ n$	$= \mathcal{B}[\![add\ e_1\ e_2]\!]\ r\ n$; MKINT, and similarly for *sub*, *mul*, *div*
7.	$\mathcal{E}[\![neg\ e]\!]\ r\ n$	$= \mathcal{B}[\![neg\ e]\!]\ r\ n$; MKINT
8.	$\mathcal{E}[\![eq\ e_1\ e_2]\!]\ r\ n$	$= \mathcal{B}[\![eq\ e_1\ e_2]\!]\ r\ n$; MKBOOL, and similarly for *lt*, *gt*, *ne*, *ge*, *le*
9.	$\mathcal{E}[\![not\ e]\!]\ r\ n$	$= \mathcal{B}[\![not\ e]\!]\ r\ n$; MKBOOL
10.	$\mathcal{E}[\![and\ e_1\ e_2]\!]\ r\ n$	$= \mathcal{E}[\![if\ e_1\ e_2\ false]\!]\ r\ n$
11.	$\mathcal{E}[\![or\ e_1\ e_2]\!]\ r\ n$	$= \mathcal{E}[\![if\ e_1\ true\ e_2]\!]\ r\ n$
12.	$\mathcal{E}[\![cons\ e_1\ e_2]\!]\ r\ n$	$= \mathcal{C}[\![e_1]\!]\ r\ n$; $\mathcal{C}[\![e_2]\!]\ r\ (n+1)$; CONS
13.	$\mathcal{E}[\![null\ e]\!]\ r\ n$	$= \mathcal{E}[\![e]\!]\ r\ n$; NULL; MKBOOL
14.	$\mathcal{E}[\![hd\ e]\!]\ r\ n$	$= \mathcal{E}[\![e]\!]\ r\ n$; HD; EVAL, similarly for *tl*
15.	$\mathcal{E}[\![if\ e_1\ e_2\ e_3]\!]\ r\ n$	$= \mathcal{B}[\![e_1]\!]\ r\ n$; JFALSE l_1; $\mathcal{E}[\![e_2]\!]\ r\ n$; JMP l_2; LABEL l_1; $\mathcal{E}[\![e_3]\!]\ r\ n$; LABEL l_2 where l_1 and l_2 are new unique labels
16.	$\mathcal{E}[\![$**let** d **in** $e]\!]\ r\ n$	$= \mathcal{C}let[\![d]\!]\ r\ n$; $\mathcal{E}[\![e]\!]\ r'\ n'$; SLIDE $(n'-n)$, where $(r', n') = \mathcal{X}\,r[\![d]\!]\ r\ n$
17.	$\mathcal{E}[\![$**letrec** $d\ in\ e]\!]\ r\ n$	$= \mathcal{C}letrec[\![d]\!]\ r'\ n'$; $\mathcal{E}[\![e]\!]\ r'\ n'$; SLIDE $(n'-n)$, where $(r', n') = \mathcal{X}\,r[\![d]\!]\ r\ n$
18.	$\mathcal{E}[\![e]\!]\ r\ n$	$= \mathcal{C}[\![e]\!]\ r\ n$; EVAL otherwise

Scheme $\mathcal{B}$: Evaluate basic value

1.	$\mathcal{B}[\![i]\!]\ r\ n$	= PUSHBASIC i
2.	$\mathcal{B}[\![b]\!]\ r\ n$	= PUSHBASIC b
3.	$\mathcal{B}[\![add\ e_1\ e_2]\!]\ r\ n$	$= \mathcal{B}[\![e_1]\!]\ r\ n$; $\mathcal{B}[\![e_2]\!]\ r\ (n+1)$; ADD, similarly for *sub*, *mul*, *div*, *eq*, *ne*, *lt*, *gt*, *ge*, *le*.
4.	$\mathcal{B}[\![neg\ e]\!]\ r\ n$	$= \mathcal{B}[\![e]\!]\ r\ n$; NEG
5.	$\mathcal{B}[\![not\ e]\!]\ r\ n$	$= \mathcal{B}[\![e]\!]\ r\ n$; NOT
6.	$\mathcal{B}[\![null\ e]\!]\ r\ n$	$= \mathcal{E}[\![e]\!]\ r\ n$; NULL
7.	$\mathcal{B}[\![if\ e_1\ e_2\ e_3]\!]\ r\ n$	$= \mathcal{B}[\![e_1]\!]\ r\ n$; FALSE l_1; $\mathcal{B}[\![e_2]\!]\ r\ n$; JMP l_2; LABEL l_1; $\mathcal{B}[\![e_3]\!]\ r\ n$; LABEL l_2 where l_1 and l_2 are new unique labels
8.	$\mathcal{B}[\![$**let** d **in** $e]\!]\ r\ n$	$= \mathcal{C}let[\![d]\!]\ r\ n$; $\mathcal{B}[\![e]\!]\ r'\ n'$; POP $(n'-n)$ where $(r', n') = \mathcal{X}\,r[\![d]\!]\ r\ n$
9.	$\mathcal{B}[\![$**letrec** d **in** $e]\!]\ r\ n$	$= \mathcal{C}letrec[\![d]\!]\ r'\ n'$; $\mathcal{B}[\![e]\!]\ r'\ n'$; POP $(n'-n)$ where $(r', n') = \mathcal{X}\,r[\![d]\!]\ r\ n$
10.	$\mathcal{B}[\![e]\!]\ r\ n$	$= \mathcal{E}[\![e]\!]\ r\ n$; GET, otherwise

Scheme $\mathcal{C}$: Construct graph

1.	$\mathcal{C}[\![i]\!]\ r\ n$	= PUSHINT i
2.	$\mathcal{C}[\![b]\!]\ r\ n$	= PUSHBOOL b
3.	$\mathcal{C}[\![nil]\!]\ r\ n$	= PUSHNIL
4.	$\mathcal{C}[\![f]\!]\ r\ n$	= PUSHFUN f
5.	$\mathcal{C}[\![x]\!]\ r\ n$	= PUSH $(n - r(x))$
6.	$\mathcal{C}[\![cons\ e_1\ e_2]\!]\ r\ n$	$= \mathcal{C}[\![e_1]\!]\ r\ n$; $\mathcal{C}[\![e_2]\!]\ r\ (n+1)$; CONS
7.	$\mathcal{C}[\![e_1\ e_2]\!]\ r\ n$	$= \mathcal{C}[\![e_1]\!]\ r\ n$; $\mathcal{C}[\![e_2]\!]\ r\ (n+1)$; MKAP, if not matched above
8.	$\mathcal{C}[\![$**let** d **in** $e]\!]\ r\ n$	$= \mathcal{C}let[\![d]\!]\ r\ n$; $\mathcal{C}[\![e]\!]\ r'\ n'$; SLIDE $(n'-n)$ where $(r', n') = \mathcal{X}\,r[\![d]\!]\ r\ n$
9.	$\mathcal{C}[\![$**letrec** d **in** $e]\!]\ r\ n$	$= \mathcal{C}letrec[\![d]\!]\ r'\ n'$; $\mathcal{C}[\![e]\!]\ r'\ n'$; SLIDE $(n'-n)$ where $(r', n') = \mathcal{X}\,r[\![d]\!]\ r\ n$

Miscellaneous schemes for local definitions

$\mathcal{X}\,r[\![v_1 = e_1$ **and** $\cdots v_i = e_i \cdots$ **and** $v_m = e_m]\!]\ r\ n = (r[v_i = n+1, \cdots v_i = n+i, \cdots v_m = n+m],\ n+m)$
$\mathcal{C}let[\![v_1 = e_1$ **and** $\cdots v_i = e_i \cdots$ **and** $v_m = e_m]\!]\ r\ n = \mathcal{C}[\![e_1]\!]\ r\ n; \cdots \mathcal{C}[\![e_i]\!]\ r\ (n+i-1); \cdots \mathcal{C}[\![e_m]\!]\ r\ (n+m-1)$
$\mathcal{C}letrec[\![v_1 = e_1$ **and** $\cdots v_i = e_i \cdots$ **and** $v_m = e_m]\!]\ r\ n =$ ALLOC m; $\mathcal{C}[\![e_1]\!]\ r\ (n+m)$; UPDATE $m; \cdots$
$\mathcal{C}[\![e_i]\!]\ r\ (n+m)$; UPDATE $(m+1-i); \cdots \mathcal{C}[\![e_m]\!]\ r\ (n+m)$; UPDATE 1

In addition, there are 3 help-functions used for local definitions: $\mathcal{X}r$ returns a pair of the extended environment and the new stack depth, $\mathcal{C}let$ and $\mathcal{C}letrec$ gives code to extend the stack with local definitions. In the translation schemes r is a mapping from identifiers of parameters to their location on the stack, and n is the current depth of the stack. Below we show compilation of the function $f\ x = x.f\ x$.

$\mathcal{F}[\![f\ x = cons\ x\ (f\ x)]\!] =$
$\mathcal{E}[\![cons\ x\ (f\ x)]\!]\ [x = 2]\ 2$; UPDATE 2; RET 1 =
$\mathcal{C}[\![\mathrm{x}]\!]\ [x = 2]\ 2$; $\mathcal{C}[\![\mathrm{f\ x}]\!]\ [x = 2]\ 3$; CONS; UPDATE 2; RET 1 =
PUSH 0; $\mathcal{C}[\![\mathrm{f}]\!]\ [\mathrm{x}{=}2]\ 3$; $\mathcal{C}[\![x]\!]\ [x = 2]\ 4$; MKAP; CONS; UPDATE 2; RET 1 =
PUSH 0; PUSHFUN f; PUSH 2; MKAP; CONS; UPDATE 2; RET 1.

5.3 The abstract machine

A state in the abstract G-machine is a 7-tuple $\langle O, C, S, V, G, E, D\rangle$ where

O	is the output produced so far, as shown in the example in figure 1. It consists of a sequence of integers and booleans. In an actual implementation O is printed on standard output.
C	is the G-code sequence currently being executed.
S	is a stack of node names, i.e., pointers into the graph.
V	is a stack of basic values, i.e., integers and booleans on which the arithmetic and logical operations are performed, as shown in section 4.
G	is the graph: a mapping from node names to nodes. We have nodes of the following types:

INT i	integer nodes,
BOOL b	boolean nodes,
NIL	empty list nodes,
CONS n_1 n_2	list nodes, where n_1 is a pointer to the head graph and n_2 is a pointer to the tail graph,
AP n_1 n_2	application nodes, where n_1 is a pointer to the function graph and n_2 is a pointer to the argument graph,
FUN f	a node with a reference to the compiled function f,
HOLE	a node which is to be filled in with another value later; it is used while constructing cyclic graphs for letrec expressions.

E	is a global environment, which is a mapping from function names to pairs consisting of the number of curried arguments of the function, and its code sequence. E corresponds to the code segment in conventional machines and is constant throughout the execution of the program.
D	is a dump used for recursive calls to EVAL: a stack of pairs consisting of

- a stack of node names: S before EVAL,
- a G-code sequence: C before EVAL.

Table 4 summarises the state transition rules for the G-machine instructions used in the compilation rules given in table 3.

Table 4: **State transition rules for G-machine instructions**

1. $\langle o, \mathsf{PRINT}.c, n.s, v, G[n = \mathrm{INT}\ i], E, D\rangle \Rightarrow \langle o;i, c, s, v, G[n = \mathrm{INT}\ i], E, D\rangle$

2. $\langle o, \mathsf{PRINT}.c, n.s, v, G[\mathrm{n} = \mathrm{BOOL\ b}], E, D\rangle \Rightarrow \langle o;b, c, s, v, G[\mathrm{n} = \mathrm{BOOL\ b}], E, D\rangle$

3. $\langle o, \mathsf{PRINT}.c, n.s, v, G[n = \mathrm{CONS}\ n_1\ n_2], E, D\rangle \Rightarrow$
$\langle o, \mathsf{EVAL.PRINT.EVAL.PRINT}.c, n_1.n_2.s, v, G[n = \mathrm{CONS}\ n_1\ n_2], E, D\rangle$

4. $\langle o, \mathsf{PRINT}.c, n.s, v, G[n = \mathrm{NIL}], E, D\rangle \Rightarrow \langle o, c, s, v, G[n = \mathrm{NIL}], E, D\rangle$

5. $\langle o, \mathsf{EVAL}.c, n.s, v, G[n = \mathrm{AP}\ n_1\ n_2], E, D\rangle \Rightarrow$
$\langle o, \mathsf{UNWIND}.(), n.(), v, G[n = \mathrm{AP}\ n_1\ n_2], E, (c,s).D\rangle$

6. $\langle o, \mathsf{EVAL}.c, n.s, v, G[n = \mathrm{INT}\ i], E, D\rangle \Rightarrow \langle o, c, n.s, v, G[n = \mathrm{INT}\ i], E, D\rangle$,
similarly for nodes BOOL b, NIL, CONS $n_1\ n_2$ and FUN f.

7. $\langle o, \mathsf{UNWIND}.(), n.s, v, G[n = \mathrm{AP}\ n_1\ n_2], E, D\rangle \Rightarrow \langle o, \mathsf{UNWIND}.(), n_1.n.s, G[n = \mathrm{AP}\ n_1\ n_2], E, D\rangle$

8. $\langle o, \mathsf{UNWIND}.(), n_0.n_1 \cdots n_k.s, v, G[n_0 = \mathrm{FUN}\ f,$
$n_1 = \mathrm{AP}\ n_1'\ n_1'', \cdots n_k = \mathrm{AP}\ n_k'\ n_k''], E[f = (k,c)], D\rangle \Rightarrow$
$\langle o, c, n_1'' \cdots n_k''.n_k.s, v, G[n_0 = \mathrm{FUN}\ f, n_1 = \mathrm{AP}\ n_1'\ n_1'', \cdots n_k = \mathrm{AP}\ n_k'\ n_k''], E[f = (k,c')], D\rangle$

9. $\langle o, \mathsf{UNWIND}.(), n_0.n_1 \cdots n_k.(), v, G[n_0 = \mathrm{FUN}\ f], E[f = (a,c')], (c',s').D\rangle$ *and* $k < a \Rightarrow$
$\langle o, c', n_k\ .s', v, G[n_0 = \mathrm{FUN}\ f], E[f = (k,c')], D\rangle$

10. $\langle o, \mathsf{RET}\ m.c, v, n_1 \cdots\ n_m.n.(), G[n = \mathrm{INT}\ i], E, (c',s').D\rangle \Rightarrow$
$\langle o, c', n.s', v, G[n = \mathrm{INT}\ i], E, D\rangle$, *similarly for nodes* BOOL b, NIL *and* CONS $n_1\ n_2$.

11. $\langle o, \mathsf{RET}\ m.c, n_1.\cdots n_m.n.s, v, G[n = \mathrm{AP}\ n_1\ n_2], E, D\rangle \Rightarrow$
$\langle o, \mathsf{UNWIND}.(), n.s, v, G[n = \mathrm{AP}\ n_1\ n_2], E, D\rangle$, *similarly for* $n = \mathrm{FUN}\ f$.

12. $\langle o, \mathsf{PUSHINT}\ i.c, s, v, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{INT}\ i], E, D\rangle$

13. $\langle o, \mathsf{PUSHBOOL}\ b.c, s, v, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{BOOL}\ b], E, D\rangle$

14. $\langle o, \mathsf{PUSHNIL}.c, s, v, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{NIL}], E, D\rangle$

15. $\langle o, \mathsf{PUSHFUN}\ f.c, s, v, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{FUN}\ f], E, D\rangle$

16. $\langle o, \mathsf{PUSH}\ m.c, n_0.\cdots.n_m.s, v, G, E, D\rangle \Rightarrow \langle o, c, n_m.n_0.\cdots n_m.s, v, G, E, D\rangle$

17. $\langle o, \mathsf{MKINT}.c, s, i.v, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{INT}\ i], E, D\rangle$

18. $\langle o, \mathsf{MKBOOL}.c, s, b.v, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{BOOL}\ b], E, D\rangle$

19. $\langle o, \mathsf{MKAP}.c, n_1.n_2.s, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{AP}\ n_2\ n_1], E, D\rangle$

20. $\langle o, \mathsf{CONS}.c, n_1.n_2.s, G, E, D\rangle \Rightarrow \langle o, c, n'.s, v, G[n' = \mathrm{CONS}\ n_2\ n_1], E, D\rangle$

21. $\langle o, \mathsf{ALLOC}\ m.c, s, v, G, E, D\rangle \Rightarrow$
$\langle o, c, n_1' \cdots n_m'.s, v, G[n_1' = \mathrm{HOLE}, \cdots n_m' = \mathrm{HOLE}], E, D\rangle$

22. $\langle o, \mathsf{UPDATE}\ m.c, n_0.\cdots n_m.s, v, G[n_0 = N_0\ , n_m = N_m], E, D\rangle \Rightarrow$
$\langle o, c, n_1.\cdots n_m.s, v, G[n_0 = N_0\ , n_m = N_0], E, D\rangle$

23. $\langle o, \mathsf{SLIDE}\ m.c, n_0.\cdots.n_m.s, v, G, E, D\rangle \Rightarrow \langle o, c, n_0.s, v, G, E, D\rangle$

24. $\langle o, \mathsf{GET}.c, n.s, v, G[n = \mathrm{INT}\ i], E, D\rangle \Rightarrow \langle o, c, s, i.v, G[n = \mathrm{INT}\ i], E, D\rangle$

25. $\langle o, \mathsf{GET}.c, n.s, v, G[n = \mathrm{BOOL}\ b], E, D\rangle \Rightarrow \langle o, c, s, b.v, G[n = \mathrm{BOOL}\ b], E, D\rangle$

26. $\langle o, \mathsf{PUSHBASIC}\ i.c, s, v, G, E, D\rangle \Rightarrow \langle o, c, s, i.v, G, E, D\rangle$

27. $\langle o, \mathsf{ADD}.c, s, i_2.i_1.v, G, E, D\rangle \Rightarrow \langle o, c, s, (i_1 + i_2).v, G, E, D\rangle$,
similarly for SUB, MUL, DIV, EQ, NE, LT, GT, LE and GE,
the last six putting boolean values on V.

28. $\langle o, \mathsf{NEG}.c, s, i.v, G, E, D\rangle \Rightarrow \langle o, c, s, (-i).v, G, E, D\rangle$

29. $\langle o, \mathsf{NOT}.c, s, b.v, G, E, D\rangle \Rightarrow \langle o, c, s, (\neg b).v, G, E, D\rangle$

30. $\langle o, \mathsf{JFALSE}\ l.c, s, \mathrm{true}.v, G, E, D\rangle \Rightarrow \langle o, c, s, G, E, D\rangle$

31. $\langle o, \mathsf{JFALSE}\ l.c, s, \mathrm{false}.v, G, E, D\rangle \Rightarrow \langle o, \mathsf{JMP}\ l.c, s, v, G, E, D\rangle$

32. $\langle o, \mathsf{JMP}\ l.\cdots \mathsf{LABEL}\ l.c, s, v, G, E, D\rangle \Rightarrow \langle o, c, s, v, G, E, D\rangle$

33. $\langle o, \mathsf{LABEL}\ l.c, s, v, G, E, D\rangle \Rightarrow \langle o, c, s, v, G, E, D\rangle$

34. $\langle o, \mathsf{HD}.c, n.s, v, G[n = \mathrm{CONS}\ n_1\ n_2], E, D\rangle \Rightarrow \langle o, c, n_1.s, v, G[n = \mathrm{CONS}\ n_1\ n_2], E, D\rangle$,
similarly for TL

35. $\langle o, \mathsf{NULL}.c, n.s, v, G[n = \mathrm{CONS}\ n_1\ n_2], E, D\rangle \Rightarrow \langle o, c, s, \mathrm{false}.v, G[n = \mathrm{CONS}\ n_1\ n_2], E, D\rangle$

36. $\langle o, \mathsf{NULL}.c, n.s, v, G[n = \mathrm{NIL}], E, D\rangle \Rightarrow \langle o, c, s, \mathrm{true}.v, G[n = \mathrm{NIL}], E, D\rangle$

37. $\langle o, \mathsf{POP}\ m.c, n_1.\cdots.n_m.s, v, G, E, D\rangle \Rightarrow \langle o, c, s, v, G, E, D\rangle$

$$\begin{array}{lll}
\multicolumn{3}{l}{\langle (), c_0, (), (), \{\}, E_0, () \rangle} \\
\text{where} & c_0 = & \mathcal{E}[\![e_0]\!]\ r_0\ 0;\ \mathsf{PRINT} \\
\text{and} & E_0 = & \{ \quad \begin{array}{ll}
f_0 & : (n_1,\ \mathcal{F}[\![f_0\ x_1 \cdots x_{n_1} = e_0]\!]\ )\ , \\
\vdots & \\
f_m & : (n_m,\ \mathcal{F}[\![f_m\ x_1 \cdots x_{n_m} = e_m]\!]\ ), \\
add & : (2,\ \mathcal{F}[\![\ p\ x\ y = add\ x\ y]\!]\ ), \\
sub & : (2,\ \mathcal{F}[\![\ p\ x\ y = sub\ x\ y]\!]\ ), \\
\vdots & \\
\end{array} \\
 & & \}
\end{array}$$

Figure 4: Initial state of the G-machine.

In a G-machine state, () denotes an empty stack or an empty code sequence. The semicolon appends values onto an output sequence. Period is used as infix cons for instruction sequences and push for stacks. Updating of the graph is written as e.g. $G[n = \mathrm{INT}\, i]$. If there is a node named n previously in G, then the node n is updated with a new value, otherwise a new node is created. This notation is also used in pattern matching situations, for instance state transition rule 1 is applicable if the top of the stack points to an integer node. For instructions with parameters, e.g. PUSH $m.c$ binds as (PUSH $m).c$. A node name that occurs only in the right hand side of a transition rule is considered to be new and unique, e.g. n' in transition rule 12. G-machine states that do not match any rule are considered to be run time errors.

The definition of the G-machine has certain similarities with the definition of the SECD machine [Lan64], new in our model is that we describe how we do lazy output, and handle updating and sharing in a graph, in the framework of the abstract machine.

5.4 Initial and final state of the machine

The initial configuration of the machine for a given program is shown in figure 4. The machine starts with an empty output, a code sequence c_0 for evaluating and printing the start expression e_0, an empty pointer stack and an empty basic value stack, an empty graph, an environment E_0 containing the compiled code for the functions together with their arity, and an empty dump. Since the operators $+$, $-$ etc are represented with applications to predefined functions *add*, *sub* etc in unevaluated expression graphs, the code for these functions must also be present in E_0. The machine stops when the state $\langle o, (), (), (), G, E, () \rangle$ has been reached.

5.5 The evaluation mechanism

The evaluation of the program is driven by PRINT, which in case of a list starts the evaluation of the head and the tail part of the list, see transition rules 1–4. Only the leaves of the printed data structure appears on the output, for instance the list $(2.3.nil).(5.nil).nil$ gives the output sequence 2 3 5.

The EVAL instruction reduces the graph pointed to by the pointer at the top of the

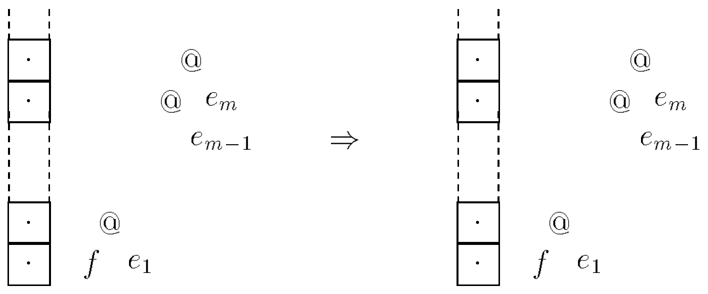

Figure 5: Rearrangement of the stack after unwind.

stack to canonical form. If the top of stack is an apply node, transition rule 5, the rest of the code sequence and the stack except for the top element is pushed onto the dump, and the unwind state is entered, following the function parts of apply nodes, pushing the function pointers on the way (transition rule 7). When a function node has been reached, and the stack is deep enough to contain all curried arguments to the function, rule 8, the stack is arranged according to figure 5. The top m elements of the stack now points to the m curried arguments of the function, and below them there is a pointer to the apply node which is to be updated with the value of the application. The reason for remaking the stack in this manner is firstly to make the function arguments easily accessible, and secondly to access function arguments and local variables introduced by **let** and **letrec** expressions uniformly.

After the stack rearrangement the function code is executed; see also compilation rule $\mathcal{F}$. If there were too few curried arguments in the application then a premature return is performed, rule 9.

The RET instruction performs a return from EVAL if the function code has updated the apply node for the return value with an integer, boolean, nil or cons node, rule 10. If the updated node is an apply node or function node then the UNWIND state is reentered, to continue the reduction of the new graph; an example when this happens is shown in figure 6 which illustrates reduction of the expression $f\ (g.nil)\ 3$, where $f\ l = hdl$ and $g\ x = 2 \times x$. The value of $f\ (g.nil)$ is the function g, and f has one 'extra' argument supplied. After EVAL and two unwind transitions we have the configuration shown in 6(b), the top of the stack is then made to point ot the argument of f, figure 6(c). The code for f then computes the value of $hd\,l$, which is the function g, and updates the apply node of the application $f : (g.nil)$ with the function node g, figure 6(d). Since the entire graph for which EVAL was called for is not yet fully reduced, the RET 1 instruction of the code for f makes the machine reenter the unwind state, figure 6(e), and the top of of the stack is made to point to the argument of g, figure 6(f). The code for g then computes the value of $2 + x$ and updates the top apply node with the integer 6. The RET 1 instruction of the function g finally performs a proper return from EVAL, figure 6(h).

The fact that 'extra' curried arguments can be applied to function in this manner, and in general we cannot know in advance how many extra, is the reason for accessing parameters and variables relative to the top of the stack (instead of relative to the bottom which perhaps at first sight would seem more natural).

$f\ l = hd\ l$	f:	PUSH 0; EVAL; HD; EVAL; UPDATE 2; RET 1.
$g\ x = 2 \times x$	g:	PUSHBASIC 1; PUSH 0; EVAL;
		GET; MUL; MKINT; UPDATE 2; RET 1

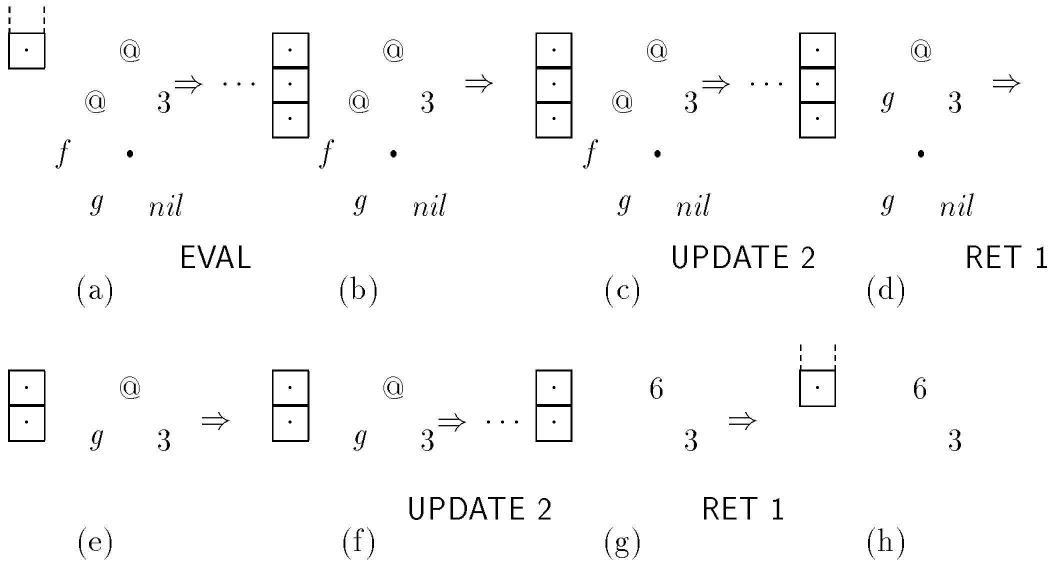

Figure 6: Graph reduction when a function returns a function.

5.6 Let and letrec expressions

The code for a **let** or **letrec** expression constructs the graphs for the locally defined expressions and puts pointers to these graphs onto the stack. When leaving the code for the **let** or **letrec** expression these stack elements are removed by the SLIDE instruction; see compilation rules E16, E17 etc. The recursive local definitions in **letrec** expressions are implemented by constructing cyclic graphs, see scheme $\mathcal{C}letrec$ in table 3 As an example consider the code sequence

$\mathcal{C}[\![\textbf{letrec}\, x = f\ x\ \textbf{in}\ x\ x]\!]\ r\ n =$
$\mathcal{C}letrec[\![\ x = f\ x\ ]\!]\ r[x = n+1]\ (n+1);\ \mathcal{C}[\![x\ x]\!]\ r[x = n+1]\ (n+1);$ SLIDE 1 =
ALLOC 1; PUSHFUN f; PUSH 1; MKAP; UPDATE 1;
PUSH 0; PUSH 1; MKAP; SLIDE 1.

Figure 7 shows some of the intermediate machine states when executing this code sequence. To construct the graph for x we must have a pointer to x, for this purpose a HOLE node is allocated by the ALLOC 1 instruction; when $f\ x$ has been constructed the HOLE node is updated with this graph.

6 Further improvements of the G-machine code

This section discusses two kinds of improvements of the G-machine code, which is not embodied in the compiler given in the previous section: improved tail recursive behaviour and exploiting the knowledge that a variable has been previously evaluated. Both kinds of improvements are included in our compiler implementation.

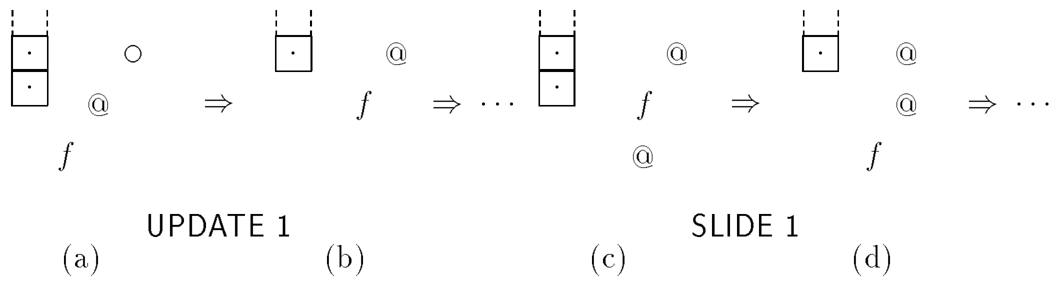

Figure 7: Construction of a cyclic graph.

6.1 Tail recursion

Graph reduction by succesive rewritings to right hand sides gives us a loop-like behavior for tail recursive calls. However, this desirable property is not preserved by the compilation scheme given in table 3, because compilation scheme $\mathcal{F}$ emits code for computing the value of the right hand side, before updating with the result. Thus using scheme $\mathcal{E}$ in $\mathcal{F}$ is advantageous if the right hand side is an application to a primitive predefined function such as *add*, *sub* etc, but does not bring out the proper tail recursive behaviour if the right hand side is an application to a user defined function. For instance, using the compilation rules in table 3, we have

$$\mathcal{F}[\![\ g\ x = f\ 5\ ]\!] = \mathcal{E}[\![\ f\ 5\ ]\!]\ [x = 2]\ 2;\ \textsf{UPDATE 2; RET 1} = \\ \textsf{PUSHFUN f; PUSHINT 5; MKAP; EVAL; UPDATE 2; RET 1.}$$

Here the EVAL instruction is unnecessary, and in fact harmful, in that it will create another stack frame for the evaluation of $f5$. If the EVAL instruction is removed from the code above the UPDATE instruction will update with the apply node of $f5$, and the RET instruction will make the machine reenter the unwind state; no additional stack frame is created.

Proper tail recursive behaviour can be reinstated into our compilation schemes by introducing yet another compilation scheme, $\mathcal{R}$ for return value, which preserves the context that the result is to be returned as the value of the current function evaluation. Starting with the compilation function $\mathcal{F}$, we then have

$$\mathcal{F}[\![\ fx_1 \cdots x_m = e\ ]\!] = \mathcal{R}[\![e]\!]\ [x_1 = m+1, \cdots x_m = 2]\ (m+1)$$

where the code emitted by $\mathcal{R}$ also performs the updating and returning. To return the value of an application to a user defined function we can do a simplistic graph rewite, by

$$\mathcal{R}[\![\ fe_1 \cdots e_m\ ]\!]\ r\ n = \mathcal{C}[\![\ fe_1 \cdots e_m\ ]\!]\ r\ n;\ \textsf{UPDATE}\ n;\ \textsf{RET}\ (n-1).$$

$\mathcal{R}$ can also be made to propagate down the branches of an **if** expression, by

$$\mathcal{R}[\![\textbf{if}\ e_1\ e_2\ e_3]\!]\ r\ n = \mathcal{B}[\![e_1]\!]\ r\ n;\ \textsf{JFALSE}\ l_1;\ \mathcal{R}[\![e_2]\!]\ r\ n;\ \textsf{LABEL}\ l_1;\ \mathcal{R}[\![e_3]\!]\ r\ n$$

and down into the **in**-expression in **let** and **letrec** expressions, by

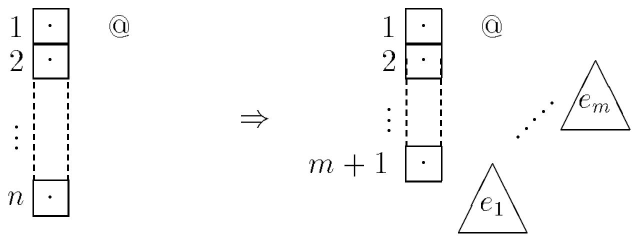

Figure 8: Rearranging the stack for tail calls.

$$\begin{aligned}\mathcal{R}[\![\mathbf{let}\ d\ \mathbf{in}\ e]\!]\ r\ n &= \ \mathcal{C}let[\![d]\!]\ r\ n;\ \mathcal{R}[\![e]\!]\ r'\ n'\\ \mathcal{R}[\![\mathbf{letrec}\ d\ \mathbf{in}\ e]\!]\ r\ n &= \ \mathcal{C}letrec[\![d]\!]\ r'\ n';\ \mathcal{R}[\![e]\!]\ r'\ n'\\ &\quad\ \text{where } (r',n') = \mathcal{X}r[\![d]\!]\ r\ n.\end{aligned}$$

The default case for $\mathcal{R}$ is

$$\mathcal{R}[\![e]\!]\ r\ n = \mathcal{E}[\![e]\!]\ r\ n;\ \mathsf{UPDATE}\ n;\ \mathsf{RET}\ (n-1).$$

To return the value of the application $f e_1 \cdots e_m$, we can do even better by shortcircuiting the unwind action which in this case follows the RET instruction. Provided the arity of f is m, which is a condition for that the same apply node will be updated both by the calling function and f, we can use

$$\mathcal{R}[\![f\ e_1 \cdots e_m]\!]\ r\ n = \mathcal{S}[\![\ e_1 \cdots e_m\ ]\!]\ r\ n;\ \mathsf{JFUN}\ f.$$

The new scheme $\mathcal{S}$ emits a code sequence to rearrange the stack in the manner shown in figure 8, and then a direct jump is performed to the first instruction of f, thus turning tail recursion into loops in the G-machine code. Using this method on our little example above, assuming f only takes one argument, we would get

$$\mathcal{F}[\![\ g\ x = f\ 5\ ]\!] = \mathsf{PUSHINT\ 5;\ MOVE\ 1;\ JFUN\ f.}$$

The new instructions MOVE and JFUN are defined by

$$\begin{aligned}&\langle o,\ \mathsf{MOVE}\, m.c,\ n_0 \cdots n_{m-1}.n_m.s,\ v,\ G,\ E,\ D\rangle \Rightarrow \langle o,\ c,\ n_1 \cdots n_{m-1}.n_0.s,\ v,\ G,\ E,\ D\rangle\\ &\langle o,\ \mathsf{JFUN\ f}.c,\ s,\ (),\ G,\ E[f=(a,c')],\ D\rangle \Rightarrow \langle o,\ c',\ G,\ E[f=(a,c')],\ D\rangle.\end{aligned}$$

6.2 On evaluated variables

The first time EVAL is executed for a particular variable, that graph is reduced to canonical form, and subsequent EVALs on the same variable has no effect. By keeping count of when variables are being evaluated in each function we can avoid emitting EVAL instructions more than once for each variable. For example, to compute the basic value of the expression $x \times x$, table 3 gives us the code sequence

$$\begin{aligned}&\mathcal{B}[\![mul\ x\ x]\!]\ [x=1]\ 1 =\\ &\mathcal{B}[\![x]\!]\ [x=1]\ 1;\ \mathcal{B}[\![\ x\ ]\!]\ [x=1]\ 1;\ \mathsf{MUL} =\\ &\mathsf{PUSH\ 0;\ EVAL;\ GET;\ PUSH\ 0;\ EVAL;\ GET;\ MUL.}\end{aligned}$$

Here the second EVAL instruction is clearly useless and can be eliminated. Apart from having removed a useless EVAL instruction, conditions also become better for target code generation from the G-code, since we get longer code sequences unbroken by calls to EVAL and may thus keep things in machine registers a bit longer.

Because of the cost involved in construction and reduction of expression graphs, it is cheaper to evaluate some expressions directly than to construct their graphs, even if the value is not going to be used. This is the case for expressions involving constants, variables which has been evaluated previously, and arithmetic and logical primitive functions (we ignore the problem of overflow and other exceptions). As an example, consider construction of the expression $2 \times x + y$. The compilation rules in table 3 gives us

$$\mathcal{C}[\![add(mul\, 2\, x)\, y]\!]\ [x = 2,\ y = 1]\ 2 =$$

```
PUSHFUN add; PUSHFUN mul; PUSHINT 2;
MKAP; PUSH 2; MKAP; PUSH 3; MKAP.
```

If the variable x has been previously evaluated, it is safe to compute the value of $2 \times x$, and instead we can use the code sequence

```
PUSHFUN add; PUSHBASIC 2; PUSH 1; GET; MUL; MKINT PUSH 3; MKAP.
```

and if both x and y have been previously evaluated, we can use the code sequence

```
PUSHBASIC 2; PUSH 0; GET; MUL; PUSH 1; GET; ADD; MKINT.
```

When dealing with expressions with list values the situation is similar. For instance, consider construction of the expression *tl l*, as in the function definition

$$f\ l = \textbf{if}\ null\ l\ \textbf{then}\ \cdots\ \textbf{else}\ g\ (tl\ l)$$

Because of the test in the condition part of the **if**-expression, not only can we know for sure that *l* has been evaluated, in the else part of the **if**-expression we can also assert that l is on cons form. To construct the expression *tl l*, instead of using

```
PUSHFUN tl; PUSH 2; MKAP
```

we can use the code sequence

```
PUSH 1; TL.
```

Not only does this avoid allocation of an apply node, it also removes the overhead of executing the code for the *tl* function when function g calls for evaluation of its argument.

When a variable with a list value cannot be determined statically to be on cons-form, we can test for this dynamically, with instructions MKHD and MKTL, used in the following compilation rules.

$$\begin{aligned} \mathcal{C}[\![\ hd\ e\ ]\!]\ r\ n &= \mathcal{C}[\![\ \mathrm{e}\ ]\!]\ r\ n;\ \mathsf{MKHD} \\ \mathcal{C}[\![\ tl\ e\ ]\!]\ r\ n &= \mathcal{C}[\![\ \mathrm{e}\ ]\!]\ r\ n;\ \mathsf{MKTL} \end{aligned}$$

MKHD and MKTL test whether the top of stack is on cons-form, and if this is the case then behaves as the HD and TL instructions respectively, otherwise constructs the graphs. These instruction are defined by

$$\begin{array}{l}\langle o,\ \mathsf{MKHD}.c,\ n.s,\ v,\ G[n = \mathrm{CONS}\, n_1\, n_2],\ E,\ D\rangle \Rightarrow \\ \langle o,\ c,\ n_1.s,\ v,\ G[n = \mathrm{CONS}\, n_1\, n_2],\ E,\ D\rangle\end{array}$$

otherwise:

$$\begin{array}{l}\langle o,\ \mathsf{MKHD}.c,\ n.s,\ v,\ G,\ E,\ D\rangle \Rightarrow \\ \langle o,\ c,\ n_1.s,\ v,\ G[n_1 = \mathrm{AP}\, n_2\, n,\ n_2 = \mathrm{FUN}\, hd],\ E,\ D\rangle\end{array}$$

and similarly for MKTL.

The analysis shown above can detect call-by-name to call-by-value transformations only locally within a function. A more general method would be to use a global analysis method, as described in [Myc80]. A future version of our compiler may include such an analysis phase.

7 Implementation

This section discusses some features of our compiler implementation of the G-machine concept. The source language is a completely function variant of ML [GMW79], with call-by-name semantics. The last phase of the compiler translates the G-machine code into target code for the VAX-11 computer.

7.1 Compiler organisation

The compiler is organised into the following parts:

Syntax analysis: Builds an abstract syntax tree of the program.

Type checking: Checks that the program is well-typed, using a polymorphic type checking algorithm [Mil78].

Program transformation: Transforms the program into a set of functions, possibly mutually recursive, as described in section 5.1. Also, the user defined data types and pattern matching is transformed into simpler constructs.

Value analysis: Performs the analysis on evaluated variables as discussed in section 6.2.

G-code generation: Translates the functions into G-machine code.

Target code generation: Translates the G-machine code into assembly code for the VAX-11 computer.

The entire compiler, except for the syntax analysis, has been written in *fc* [Aug82], a functional language with lazy evaluation, a forerunner to the present implementation based on our earlier ideas of compiled graph reduction [Joh81]. We are currently in the process of rewriting the compiler into its own language.

7.2 Target code generation

For target code generation, the components of the G-machine state is mapped onto the target computer in the following way:

- O is printed on standard output.
- C is the target code of the currently executed function, and the program counter.
- S is a data area for the pointer stack, and a stack pointer register (called *ep*).
- V is the system stack and stack pointer (*sp*).
- G is a large heap area divided into two equally sized halves, and a register (called *hp*) as heap pointer, pointing to the next free location (see below).
- E is the target code for the functions, with code that performs nr-of-arguments-check.
- D is the system stack and stack pointer (*sp*). Only pointers into the S stack and into the system stack are pushed, not entire stacks and dumps as description of the abstract machine suggests.

Both the V stack and the dump D is mapped onto the same stack in the target machine, which is possible because things pushed onto the V stack are only used locally in functions which pushed the value.

The garbage collector is a variant of Fenichel-Yochelson's copying garbage collector [FY69], but for vary-sized cells, and works as follows. The heap is divided into two equally sized areas. Memory is allocated from one area at a time by simply incrementing the heap pointer *hp*, and when running out of memory in one area the entire graph is copied into the other heap area, leaving the garbage behind, also updating the pointers on the pointer stack S. In the target code, before an instruction sequence that allocates a certain amount of memory, a check is made if that amount of memory is available on the heap, if not the garbage collector is invoked. A disadvantage of this method of memory management is that only half of the total available memory can be utilised; however on computers with large virtual address spaces this is not a serious problem. To its advantage, the time used for garbage collection is proportional to the size of the graph, (not the size of the heap area, as it is for mark-scan methods) thus taking little time for small graphs.

The target code generation is done by deferring some operations on the pointer stack S and basic value stack V, and instead simulate the contents of the topmost elements. Thus instructions PUSHINT, PUSHFUN, PUSHBASIC, etc, which pushes constants, will in the code generator push these constants on the simulated stacks. The instruction MKAP, for instance, will thus take two arguments from the simulated stack if nonempty, otherwise from the real stack. To bring out the main idea, a simple example of target code generation is shown in figure 9, which constructs the graph for the expression $3.f\ 5$. In the simulated stack *fun f* refers to a pointer to a function node *f*, *int i* refers to an integer node with value i, and *heap n* refers to a pointers into the heap at location n. In the code, newly created nodes on the heap are referred to relative to the *hp* register, and since node allocation changes the value of *hp*, we also need to carry along a current relative value of *hp*, called HP. Function nodes, integer nodes, boolean nodes and the nil node are not allocated each time on the heap; instead pointers to nodes in a constant area are used. (The simulated V stack is irrelevant for this example and is not shown.)

A further possibility which is not shown in this example is to allocate machine registers for entries into the simulated stacks, particularly the for V stack entries for the result of the usual arithmetic operations.

G-code	*VAX assembler code*	*HP*	*Simulated S stack*	*Remark*
		0	()	Start configuration
PUSHINT 3		0	int 3.()	Push pointer to integer constant 3
PUSHFUN f		0	fun f.int 3.()	Push pointer to function node for f
PUSHINT 5		0	int 5.fun f.int 3.()	Push pointer to integer constant 5
MKAP	`movl $APPLY,(hp)+`	4		Tag of apply node to heap ...
	`movl $C_F,(hp)+`	8		Fun. part = fun f to heap ...
	`movl $I_5,(hp)+`	12	heap 0.int 3.()	Arg. part = int 5 to heap.
CONS	`movl $CONS,(hp)+`	16		Tag of cons node to heap ...
	`movl $I_3,(hp)+`	20		Head part = int 3 to heap ...
	`moval -20(hp),(hp)+`	24	heap 12.()	Tail part = result of MKAP to heap ...
	`moval -12(hp),-(ep)`	24	()	Move result to real S stack.

Figure 9: Target code generation from graph construction code.

The target code is assembled in the usual manner, and loaded together with the runtime system to make an executable file. The runtime system contains code for PRINT, EVAL, unwind, the garbage collector, and also target code for the primitive predefined functions *add*, *sub* etc.

7.3 Performance

We have compared our implementation with a couple of other implementations of functional languages that have been available to us, both with strict and lazy evaluation. The implementations in the table below are the following:

1. Our implementation; lazy evaluation, executes VAX-11 code.
2. Cardelli's ML system [Car84]; strict evaluation, executes VAX-11 code.
3. The Liszt Lisp compiler under UNIX; strict evaluation, executes VAX-11 code.
4. The ML implementation in the LCF system; strict evaluation, interprets Lisp.
5. SASL, based on the SECD machine [Tur75]; lazy evaluation, interpretative.
6. C compiler under UNIX (applies only to the Fibonacci program).

The table below shows the execution time in seconds for three programs: $fib(20)$ using $fib(n) = \textbf{if } n < 2 \textbf{ then } 1 \textbf{ else } fib(n-1) + fib(n-2)$, primes up to 300 using sieve of Erathostenes, and insertion sort of 100 random elements.

	1.	2.	3.	4.	5.	6.
Fibonacci	0.92	0.5	1.1	46	31	0.46
Primes	0.50	1.2	1.1	29	20	-
Insert sort	0.37	1.0	0.8	15	12	-

The programs above have been chosen so that the results are the same independent of whether lazy or strict evaluation is used, but in general lazy evaluation permits a more direct programming style. It should be noted that in our Fibonacci program, in the recursive call to *fib* the arguments are passed by value, due to the analysis on evaluated variables described in section 6.2.

8 Related work

Jones and Muchnick [JM82] gives an alternative evaluation mechanism for combinator expressions, with a compilation algorithm which translates combinators to fixed-program code for a stack machine.

Hudak's combinator based compiler [HK84] resembles our work in many respects. He uses the standard combinators as a convenient intermediate language for performing program transformations and optimisations. The program is the converted into one containing macro-combinators, which is similar to Hughes' super-combinators [Hug82] and our global function definitions. Each macro-combinator is then translated into code for a conventional machine.

Dick Kieburtz et. al at Oregon Graduate Center is currently in the process of designing and implementing a VLSI chip for the G-machine.

9 Acknowledgements

This work was supported by the Swedish Board for Technical development (STU). The compiler has been implemented together with Lennart Augustsson, and many ideas has grown out of this close cooperation. I also wish to thank Alan Mycroft for helpful comments on earlier drafts of this paper, and the members of the Programming Methodology Group for numerous nice cake parties.

References

[Aug82] L. Augustsson. *FC mamual.* Technical Report Memo 13, Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1982.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:280–294, August 1978.

[Car84] L. Cardelli. ML under UNIX. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[Fri76] D. P. Friedman. *Cons should not evaluate its arguments*, pages 257–284. Edinburgh University Press, 1976. In the book *Automata, languages and Programming.*

[FY69] R. Fenichel and J. Yochelson. A lisp garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[HK84] P. Hudak and D. Kranz. A combinator-based compiler for a functional language. In *Proceedings 11th ACM Symposium on Principles of Programming Languages*, pages 122–132, 1984.

[Hug82] J. Hughes. Super combinators - a new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.

[JM82] N. D. Jones and S. S. Muchnick. A fixed-program machine for combinator expression evaluation. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 11–20, Pittsburgh, 1982.

[Joh81] T. Johnsson. *Code Generation for Lazy Evaluation.* Technical Report Memo 22, Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1981.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, (6):308–320, 1964.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–164, 1966.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Myc80] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings 4th International Symposium on Programming, Lecture Notes in Computer Science 83*, pages 269–281, Springer Verlag, Paris, April 1980.

[Tur75] D. A. Turner. *An implementation of SASL.* Technical report 4, University of St. Andrews, 1975.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.