SOFTWARE—PRACTICE AND EXPERIENCE. VOL. 21(3), 299–329 (MARCH 1991)

Performance of Lazy Combinator Graph Reduction

PIETER H. HARTEL*
Computer Systems Department, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

SUMMARY

The performance of program-derived combinator graph reduction is known to be superior to that of graph reduction based on a fixed set of standard combinator. The major advantage of program-derived combinator reduction is that it uses less transient store than standard combinator reduction. We show on what activities a combinator reduction algorithm spends its execution time. Based on this analysis we show that it depends to a large extent on the application how much faster a program will run if program-derived combinator are used instead of standard combinator. The analysis is based on experimental evidence obtained from a small bench-mark of medium-size functional programs. Performance gains of up to 11 × are reported for target architectures on which each memory reference consumes one unit of time. The results are valid for implementations of combinator graph reduction that use binary graphs.

KEY WORDS Turner's combinator G-machine Performance modelling Graph reduction Instruction level timing Small functional bench-mark

INTRODUCTION

Combinator graph reduction is a common implementation method for lazy functional programming languages. The performance of functional programs has improved over the years because many researchers have proposed improvements to the basic technique. In this paper we will review the problems with the basic technique and evaluate its performance. We will compare this to the performance of an improved version. The field, however, is large and we will be restrictive to be able to present concrete results. Only some commonly accepted improvements to the basic technique are therefore considered.

The first implementation of a functional programming language that was based on combinator graph reduction uses a fixed set of 'standard combinator'. The method was invented by Turner. [1] We will use it in a slightly modified form, as the basic technique and a yardstick to measure progress.

In most current implementations of functional programming languages, an optimal set of 'program-derived combinator' is created specifically for each application. This

* Current address: Department of Electronics and Computer Science, University of Southampton, Southampton S09 5NH, U.K.

0038–0644/91/030299–31$15.50

Received 24 July 1989
Revised 3 May 1990 and 24 October 1990

is the first improvement to the basic technique that we wish to consider. It was independently proposed by Johnsson [2] and Hughes. [3] In his Ph.D. thesis, Hughes presents a theoretical analysis of the efficiencies of both his own program-derived 'super combinator' method and Turner's standard combinator method. This analysis shows that the size of the combinator code produced by Turner's method is larger than super combinator code, at worst by a factor O(log *n*), where *n* is the size of the original code. [4] Hughes does not present details on his implementation of super combinator, which is unfortunate because complexity analysis at the combinator level is not enough to make statements about execution times. A super combinator may be arbitrarily complex, whereas most standard combinator are simple. In this paper we present experimental complexity measures based on decomposing any combinator into its elementary building blocks. Johnsson's program-derived combinator are different from super combinator because the former are not fully lazy. This means that under certain circumstances calculations have to be performed more than once. In a recent paper, [5] Augustsson and Johnsson have come back on this point of view and state that for a serious compiler full laziness is essential. In this paper we use the original non-fully-lazy program-derived combinator because they are the simplest to implement. To guarantee full laziness without a substantial loss in performance requires avoiding redundant full laziness. As yet this is too much an issue of debate to be considered a commonly accepted improvement. Turner's method is fully lazy.

The performance advantage of program-derived combinator over standard combinator is largely due to the difference in 'grain size' of the execution steps. In essence a single program-derived combinator performs the work of several standard combinator. To store the intermediate results requires heap cells. These are time consuming to allocate and reclaim. Standard combinator reduction produces more intermediate results than program-derived combinator reduction and is therefore generally slower.

The second improvement to the basic technique that we study is a reduction of the interpretative overhead incurred by naive graph reduction. Often the compiler can work out that if it were to generate code that builds the suspension of a function application at run time, the next thing after building the suspension would be to request its canonical form. So rather than generate code to build the suspension the compiler will output code to evaluate the expression directly. Johnsson calls this 'short-circuit' evaluation of graph reduction. The analysis involved is a very simple form of strictness analysis: it does not carry across function boundaries and is restricted to flat domains. Short-circuit analysis is reported to be effective in spite of its limitations.

In this paper we study the implementation of a lazy functional language through combinator graph reduction. The next section gives an overview of the two graph reduction methods that are the subject of the study. The third section presents a model of timing aspects in combinator graph reduction machines. In the fourth section we classify the various activities that a combinator graph reducer can be engaged in. Based on this classification we count the accesses to the store spent in most of these activities when running a benchmark of six medium-size application programs. The results obtained with Turner's standard combinator are compared to measurements obtained from an implementation of Johnsson's G-machine (last section).

AN OVERVIEW OF GRAPH REDUCTION WITH STANDARD COMBINATORS AND THE G-MACHINE

The variables that occur in functional programs are bound variables. Consider for example the function from defined in Figure 1. It computes the potentially infinite list of integers starting at the value specified by the parameter n. The colon (:) is the infix list construction operator.

A bound variable always represents the same value. Therefore, occurrences of such variables can be removed from the program text by a compilation process (bracket abstraction). The idea is to mark the place where a bound variable is removed, by a special function that will put the value of the variable in the right place during execution. The major difference between Turner's and Johnsson's methods is the way in which the abstraction removes bound variables from the source text. We will show how both methods compile and execute the function from.

Turner's method

Different abstraction rules are used, depending on the places where the bound variables occur. Associated with each abstraction rule is a combinator that has the inverse effect of the abstraction during execution. To remove the bound variable n from the function from, the three combinator Sp, B and I are needed. Their abstraction and reduction rules are shown in Figures 2(a) and 2(b) respectively. The symbols F_x and G_x represent arbitrary expressions that depend on x. The expression H does not depend on x. Abstraction is indicated by square brackets. Abstraction rule (iii) for example is to be interpreted as follows: to abstract the variable x out of the function application (H G,), replace the application by an occurrence of the combinator B, and continue to abstract x out of G_x

Figure 3 shows the abstractions necessary to remove the bound variable n from the function from. The final version of from in the last line no longer contains bound variables. What remains is an expression with the combinator Sp, B and I. The

```
def from=n: (from(plus 1 n))
```

Figure 1. The function that produces a list of integers starting at n

[x] F_x : G_x	⇒	Sp ([x] F_x) ([x] G_x)	Sp f g x	→	(f x) : (g x)
[x] x	⇒	I	I x	→	x
[x] H G_x	⇒	B H ([x] G_x)	B f g x	→	f (g x)
[x] H x	⇒	H			
(a) Abstraction rules			(b) Reduction rules		

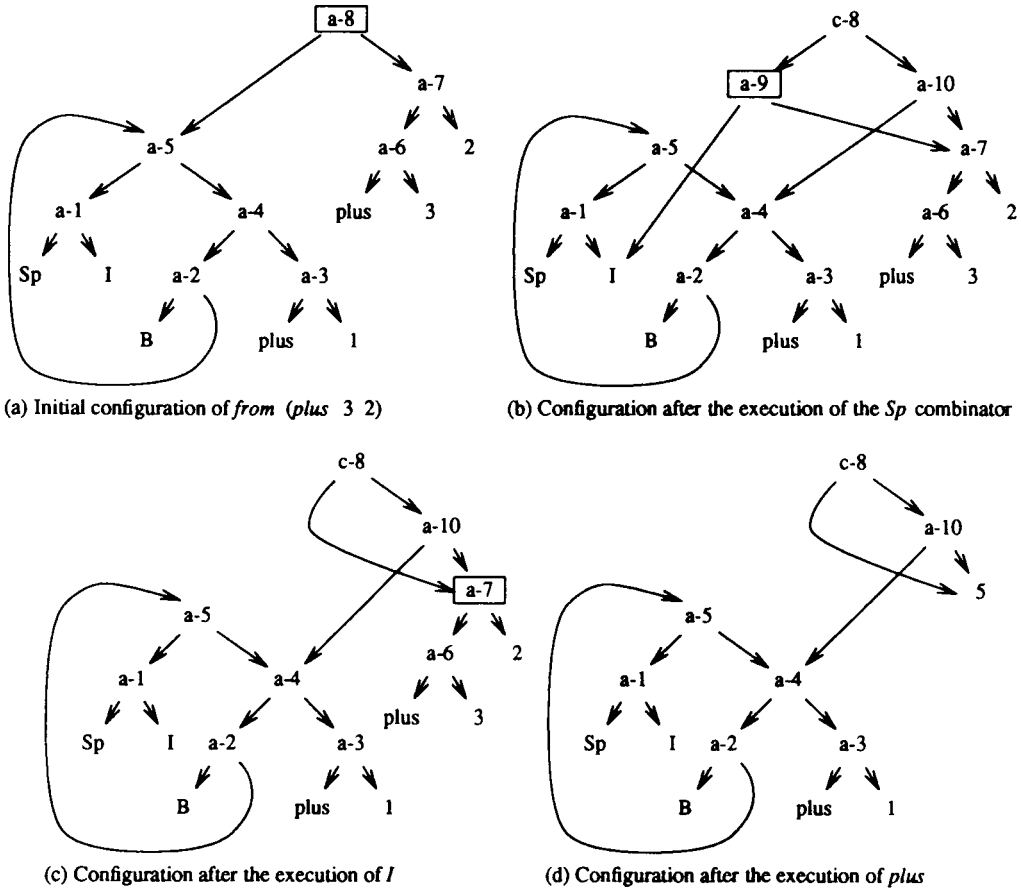
Figure 2. Abstraction and reduction with the standard combinator Sp, B *and* I

from	=	[n]	(	n)	:	(	from (	(plus 1) n))
	(i)⇒	Sp	([n]	n)		([n]	from (	(plus 1) n))
	(ii)⇒	Sp	(I	)		([n]	from (	(plus 1) n))
	(iii)⇒	Sp	(I	)		(B	from ([n]	(plus 1) n))
	(iv)⇒	Sp	(I	)		(B	from (	(plus 1)))

Figure 3. Abstraction of the function from *to standard combinator*

expression contains a circular reference to from, which can now be considered as a constant.

We can now apply the compiled version of from to an expression, e.g. from (plus 3 2). We show how that works by drawing the graphs that represent the state of the computation during the reduction process. In these drawings, interior nodes of the graph are marked with a letter and a sequence number. Application nodes have the letter a, and the letter c is used for constructor nodes. The leaf nodes represent the combinator (Sp, B, I), primitive functions (plus) and data (1, 2 and 3). Each drawing shows the situation as it exists between two reduction steps. The node drawn in a box is the root of the graph that represents the current reducible expression. This node will be overwritten by the root of the graph that represents the result of the reduction. These root nodes should actually have been drawn as one and the same node. On paper this would make the drawings hopelessly tangled, but the reducer may just destroy the old graph to make place for the new configuration. Figure 4(a) shows the initial configuration of the graph. The root of the function from is marked a–5. The circularity of the definition shows in the circular path a–5, a–4. a–2, a–5, The node where the computation starts is a–8. Following the leftmost spine we find



(a) Initial configuration of *from* (*plus* 3 2)

(b) Configuration after the execution of the *Sp* combinator

(c) Configuration after the execution of *I*

(d) Configuration after the execution of *plus*

Figure 4. The first steps when reducing from (plus 3 2) *on the standard combinator machine*

that the first reduction is an Sp reduction (rule (i)). It claims two cells a–9 and a–10 and overwrites a-8 with a constructor node c–8 shown in Figure 4(b). The effect of the next reduction step I (rule (ii) is shown in Figure 4(c)). The configurations (b)-(d) illustrate the use that is made of sharing: node a-7 not only appears as the head of the list, but is also used in the recursive call to from.

Johnsson's method

Compilation for the G-machine proceeds in a similar way to the method of the previous section. For the function from we need a small subset of Johnsson's abstraction rules, which is shown in Figure 5. The notation [x=2] indicates a (singleton) list of associations of argument names and ordinal numbers. When such a list is applied to a particular argument name the corresponding ordinal number is returned. For example ([x=2] x) returns 2.

The compilation of the function from into G-machine code proceeds as shown in Figure 6.

Graph reduction on the G-machine proceeds along similar lines to the standard combinator machine. After the initial graph is loaded into the heap the machine discovers that the first reducible expression is an application of from. This is achieved by the UNWIND instruction, which represents the 'need to print'. Figure 7(a) not only shows the state of the graph at this point, but also the state of the stack that is used to record pointers into the graph. The standard combinator machine has a similar stack, but there is no need to expose it in an introductory presentation. In the G-machine, however, the stack plays a more fundamental role in representing the current state of the computation. Figure 7(b) shows the state of the G-machine after the first four G-machine instructions have been obeyed that we compiled for from, PUSH x pushes the contents of the stack at offset x onto the stack. The top of the stack is at offset 0. PUSHNUMB n stacks a pointer to a heap cell that contains the number n and PUSHFUN f has a similar effect on a function. The stack entries

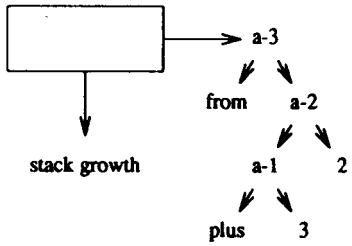
(i)	F [f x = e]		⇒	E [e] [x = 2] 2 ;	UPDATE 2 ;	RETURN 1
(ii)	E [e_1 : e_2]	r n	⇒	C [e_1] r n ;	C [e_2] r (n + 1) ;	CONS
(iii)	C [e_1 e_2]	r n	⇒	C [e_1] r n ;	C [e_2] r (n + 1) ;	MKAP
(iv)	C [x]	r n	⇒	PUSHFUN x		when x is a function
(v)	C [x]	r n	⇒	PUSHNUMB x		when x is a number
(vi)	C [x]	r n	⇒	PUSH (n - r x)		when x is an argument

Figure 5. Abstraction rules for the G-machine

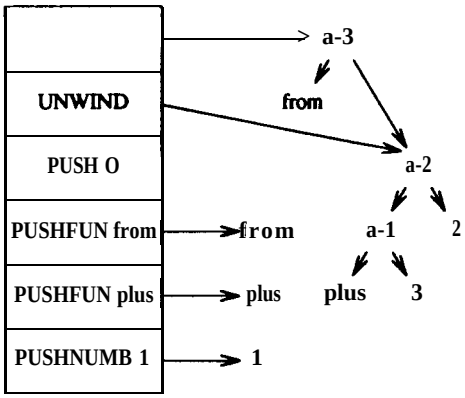
```
F [from n = n : from ( plus 1 n) ]          (i)⇒ E [n : from ( plus 1 n)] [n = 2] 2;   UPDATE 2;   RETURN 1
E [n : from ( plus 1 n )      ] [n = 2] 2  (ii)⇒ C [n] [n = 2] 2;     C [from ( plus 1 n )] [n = 2] 3;        CONS
C [n                          ] [n = 2] 2 (vi)⇒ PUSH 0
C [from ( plus 1 n )          ] [n = 2] 3 (iii)⇒ C [from] [n = 2] 3; C [plus 1 n] [n = 2] 4;                 MKAP
C [from                       ] [n = 2] 3 (iv)⇒ PUSHFUN from
C [plus 1 n                   ] [n = 2] 4  (*)⇒ PUSHFUN plus; PUSHNUMB 1; MKAP; PUSH 3; MKAP
```

from = PUSH 0; PUSHFUN from; PUSHFUN plus PUSHNUMB 1; MKAP;
PUSH 3; MKAP, MKAP; CONS; UPDATE 2; RETURN 1

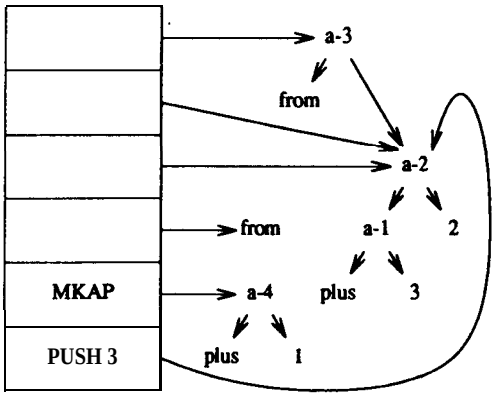
Figure 6. Compilation of the function from *to G-machine code*



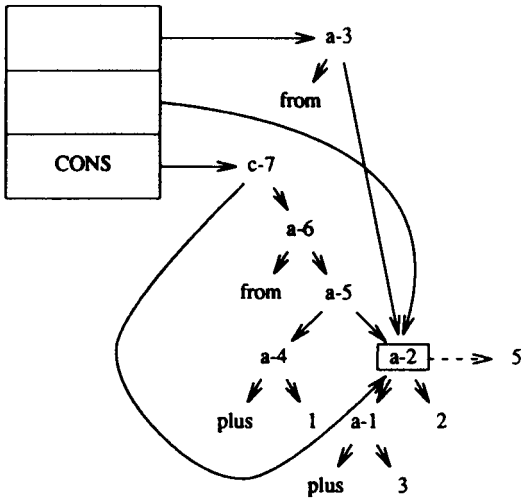
(a) Initial configuration of *from (plus* 3 2)



(b) Configuration after UNWIND, *PUSH* 0, *PUSHFUN from, PUSHFUN plus, PUSHNUMB*1



(c) **Configuration after the first MKAP and** *PUSH* 3



(d) Configuration after second and third MKAP and CONS

Figure 7. The first steps when reducing from (plus 3 2) *on the G-machine*

contain pointers into the graph, but each stack entry also shows the G-machine instruction that caused the pointer to be pushed onto the stack.

The G-machine instructions MKAP and CONS combine the top two elements of the stack into a new node as shown in Figures 7(c) and 7(d).

The reduction sequences of Figures 4 and 7 show some differences that influence performance aspects. The differences originate largely from the way instances are made of functions. The primitive functions in both methods operate in a similar way. Comparison of the reduction sequences in Figures 4 and 7 shows that the standard combinator machine requires fewer combinator steps to achieve the same result as the G-machine executes instructions, but claims more cells. The from example does not allow the G-machine compiler to short-circuit naive graph

reduction by using the B-compilation scheme. Johnsson has found that this improves the performance by a factor of 10 for some typical programs. In the remainder of the paper we will explore this matter in detail.

METHODOLOGY

We will present a model of a combinator graph reduction machine that is based on the assumption that there is a high correlation between the number of accesses to the store and execution time. The model captures the essential timing differences in Turner's and Johnsson's implementation methods. To a large extent it is independent of any particular architecture. From the model, performance parameters can be derived, which if multiplied by appropriate scale factors, add up to total execution time.

A model of combinator graph reduction

All combinator-based implementations of functional languages include a compiler to translate a program into a set of combinator. These may be regarded as an abstract machine code. Execution proceeds either by direct interpretation of the abstract machine code, or via further compilation of the abstract machine code into the native machine code of a target computer. A consequence of the difference in the abstract machine codes that are being used is that we cannot compare the performance of one implementation with that of another by comparing the number of executed combinator. However both improvements to the standard technique that we mentioned in the introduction reduce the number of heap cells required to execute a program. Therefore, measures based on store usage patterns may give reasonable performance estimates. In the three subsections below we discuss the main constituents of a combinator reduction machine. We assume that communication between them takes place via high speed registers.

Reduction strategy and execution of combinator

The semantics of a functional programming language determine a reduction strategy. This is an algorithm that decides, given the current state of the computation, which combinator must be executed next. A combinator can be executed only if it is supplied with enough arguments (it has a fixed arity). If there are no more combinator left to execute, the application program terminates. In this paper we restrict our attention to normal order reduction, which is the strategy used by Turner and Johnsson. Many combinators will cause other combinator to be activated recursively. To implement a reduction strategy, a record of the current state of the computation must be maintained. In most implementations of lazy graph reduction, there are two data structures present to support this: a graph that represents applications of combinator to arguments and a stack with pointers to the arguments of the active combinator. These arguments are subgraphs. The stack is commonly called the left ancestors stack, because it stores the ancestors of the currently active combinator. Some implementations use more than one stack for efficiency reasons, but that makes no difference to our performance evaluation.

We assume that communication between reductions is mediated by the graph, the stack and a few high-speed registers. All information that is needed by later reductions must be stored in the graph or on the stack by the current reduction. Some optimizations, in particular caches and pipelines, are excluded from the model. Each graph or stack access is assumed to consume the same amount of time. Caches, pipelines etc. may be used both with standard and program derived combinator reducers to improve the performance, but we will not be concerned with such devices.

Primitive operations embedded in combinator

A combinator has two tasks. The first is to instantiate a function and to provide access to the arguments of the function at the appropriate places in the function body. The second is to calculate new values from the arguments to be used by other combinator. The first task involves mainly references to the stack and the graph. Counting store references is therefore a good basis for the study of performance parameters associated with a reduction strategy.

The calculation of new values from current values by primitive operations cannot always be related easily to a machine-independent measure. A primitive that creates a data structure or accesses its components in essence creates a new value from existing ones by rearrangement. In this case it is sufficient to count the accesses to the store as a measure of the cost of the primitive. The remaining primitives are those that create new values from existing ones other than by rearrangement. Because non-rearranging primitives may spend an arbitrary amount of time in performing the required calculation, without even accessing the store, we cannot account for their cost just by counting accesses to the store. For instance the primitive division operation, once its operands have been copied from the store into the processors registers, needs a certain time to compute the quotient of the operands. The result is then copied from the result register to the store. Most primitives are sequential in the sense that little or nothing can be done before the operands are available. Similarly the result cannot be stored until the primitive operation is complete. Separating the retrieval and storage of operands from the actual operation on registers allows us to account for operand fetch and retrieval even though the actual non-rearranging primitive escapes from scrutiny.

The primitive operators, but also the reduction strategy, are usually implemented in terms of lower-level machine instructions. Without making further assumptions about particular target architectures we do not know which and how many low level machine instructions are responsible for a division primitive, for instance. Some machines provide an instruction for division, whereas others require a library function for that purpose. We circumvent this problem by arranging experiments such that once the operands of a primitive are loaded in registers, no further references to any store are counted until the result becomes available. The cost of a primitive operation itself is considered to depend on the target architecture only.

Storage management

The graph is stored in a heap, which is managed by a storage allocation and garbage collection procedure. We restrict our attention to binary graphs. Storage

allocation that supports variable-sized nodes is not required by a standard combinator reducer or the basic G-machine. Johnsson has proposed the use of variable-sized nodes as an optimization, but we do not consider it here. This restriction allows both implementations to use the same storage allocator. In this paper we may therefore ignore most of the performance issues in garbage collection. These have been the subject of another study. [6]

Timing of the model components

Figure 8 gives a schematic view of the timing aspects of our model of combinator graph reduction machines. The circles represent active components (i.e. implementations of the storage manager, reduction strategy etc.), and the boxes represent storage areas. The lines connecting circles and boxes represent the possible access paths. The combinator and the implementation of the reduction strategy (the mutator) have access to all available storage areas. The storage manager communicates via registers (e.g. the pointer to the top of the free list) with the mutator. The primitive operations have access to the operand and result registers. Register access is considered to be part of the instruction execution timing and as such not accounted for as a separate factor. The access paths to the registers are therefore marked with a zero. The execution time of an application program is equal to the time spent in moving data along the access paths shown in the diagram plus the time PrimTime to execute the application-specific primitives (see equation (l)). By application-specific (non-rearranging) primitives we mean those additions, divisions etc. that are visible in the text of the bench-mark program. The only non-rearranging primitives needed in the implementation of the reduction strategy and the storage manager are simple and fast operations such as addition of integers. We accept the error introduced by not including their timing explicitly in the model. Execution time can be represented by the following formula:

$$\text{time} = \text{RedAccTime} + \text{Red InsTime} + \text{CollAccTime} + \text{CollInsTime} + \text{PrimTime} + \text{LoadTime} \quad (1)$$

To deal with the parameters we propose the use of the following procedures:

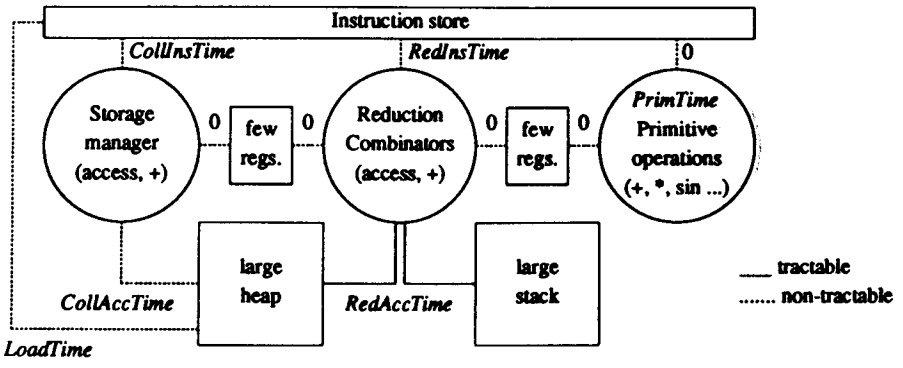


Figure 8. Model of a combinator graph reduction machine

RedAccTime

The time spent in performing 'tractable' accesses is RedAccTime. Tractable accesses are the references to the stack and the heap generated by the implementation of the reduction strategy and the combinator and the references to the stack and the heap caused by the operand fetches and stores generated by the primitives. The references are called tractable because they can be traced directly from the reducer. Examples of non-tractable references are those generated by the garbage collector or the underlying operating system. In Figure 8 tractable accesses proceed along the access paths drawn as solid lines. Dotted lines correspond to non-tractable accesses. Let Acc be the number of tractable accesses generated by a bench-mark program. Then RedAccTime = TimePerAcc x Acc, where TimePerAcc represents the unit of time per access. TimePerAcc is a constant since we assume that each access takes the same amount of time.

RedInsTime

The time spent in executing the instructions responsible for the tractable accesses is Red InsTime. Without a particular target architecture in mind we cannot know how many instructions are necessary to perform for instance an elementary 'unwind' step of the reduction algorithm. We will side-step the issue by assuming that for each reference to the graph or the stack a certain number of instructions must be executed. To indicate the magnitude of this parameter, we have measured the average number of instructions per store reference, witnessed by a highly efficient implementation of standard combinator. (Its 'nfib' number, the number of function calls per second, [7] exceeds 1000 on a VAX-11/750.) We found InsPerAcc, the average number of instructions per stack or graph reference, to vary (from bench-mark to bench-mark) between 0·72 and 0·80. We may thus approximate RedInsTime≈ InsPerAcc x RedAccTime = InsPerAcc x TimePerAcc x Ace, where InsPerAcc is a constant factor that depends on the target machine and on the instruction mix generated by the reducer.

CollAccTime + CollInsTime

The sum of CollAccTime and CollInsTime represents the time spent in allocation and reclamation of heap cells. The cost of allocating and reclaiming a cell can be made a constant that is independent of reduction method or bench-mark. This can be achieved for instance by using reference counting as the garbage collection method. With other garbage collection methods the standard combinator implementation must be given a larger heap than the scalar G-machine. With reference count garbage collection we found the average time per claimed and subsequently released cell (TimePerClaim) to vary (from bench-mark to bench-mark) between 36·5 and 43·1 μs on a 10 MHz MC68010 processor. This variation is small enough to treat TimePerClaim as a constant. [6] Hence CollAccTime + CollInsTime ≈ Time PerClaim x Claims, where Claims is the number of cells claimed by an application program and TimePerClaim is a constant factor that depends on the target machine and the instruction mix generated by the storage manager.

PrimTime

This is the total time to perform the primitive operations, such as division or reading from a file. The store references necessary to copy the operands and the result from stack or heap to the operand registers are included in the tractable references. Rather than make further assumptions about the timing requirements of primitive operators we will organize our experiments such that when running a bench-mark program on different graph reduction systems, the total time spent in executing application specific non-rearranging primitives is the same. Hence the value of PrimTime should not depend on the implementation method. In the section 'Bench-mark performance' we will see that this is not the case for all bench-mark programs.

LoadTime

The time to load the application programs is small compared to the total execution time. We will see later that LoadTime is about the same for both methods ('size of the compiled code' in Table IV). We may thus safely ignore LoadTime.

Under the above considerations we arrive at an approximation for the execution time as follows:

$$\begin{aligned} \text{time} \approx & (\text{InsPerAcc} + 1) \times \text{TimePerAcc} \times \text{Acc} \\ & + \text{TimePerClaim} \times \text{Claims} + \text{PrimTime} \end{aligned} \tag{2}$$

Here TimePerClaim and InsPerAcc are constants determined by instruction mix and target machine. TimePerAcc is the unit of time per store access.

Summarizing, we propose to count the number of claimed cells and all store accesses that we can trace from the reducer to the stack and the heap, as indicated by the solid lines in Figure 8.

IMPLEMENTATION OF LAZY COMBINATOR GRAPH REDUCTION

We give a sketch of the algorithm that performs lazy combinator graph reduction. The store references that are generated by the algorithm will serve as the raw material to base the cost measures on, hence the focus is on actions that involve store references. For instance we will make no provisions to cope with erroneous programs.

The first three steps provide the environment in which combinator can be executed:

initialize: initialize the heap and the stack by giving an initial value to the pointers that manage the heap and the stack.

load: Load the compiled functional program and the data it requires from any files into store. This includes building an application of the main function to its argument(s). A pointer PC is made to point at the root of the main application.

print: Evaluate the main application by calling unwind PC (see below). If the result is a constructor, recursively invoke print on the components of the constructor.

The eight remaining steps are performed by or on behalf of the combinator and generate the tractable accesses from the reducer to stack and heap.

unwind: Unwind the leftmost spine, beginning at the node PC, until a combinator or a data value is found. A data value causes a previously suspended reduction to be resumed (see return below). When a combinator is found, PC will point at the combinator. Pointers to the application nodes that have been visited during the unwinding are pushed onto the left ancestors stack.

call: Execution of the combinator at PC may now begin. This may include a test to see whether enough arguments are present and some rearrangements of the left ancestors stack.

access: During its invocation a combinator accesses its arguments via the left ancestors stack.

claim: Some combinator claim new cells from the storage manager and the new cells will be filled in.

evaluate: A strict argument of a combinator is evaluated by recursively invoking unwind with a pointer to the argument. The information necessary to resume the current combinator invocation is pushed onto the stack.

combine: The purpose of a combinator is to combine arguments and/or constants into a new graphical structure that will replace the current structure in the graph. The actions performed by a combinator that cannot be covered by any of the other steps listed here fall in the category combine, e.g. accessing the operand values by strict operators. We will assume that pointers to the arguments and/or constants that are necessary to make the appropriate combination are available in high speed registers. The time necessary to transfer the pointers into the registers is accounted for by unwind and access.

update: When an invocation of a combinator terminates, it will return a result to its caller. The topmost application node of the current invocation is overwritten with the root of the result. In some implementations combinator exist that do not require any arguments. Such a combinator is called a constant applicative form or CAF for short. To avoid recalculation of a CAF each time it is used, special arrangements may have to be made to allow for CAFS to be replaced by their results.

return: Execution resumes with the suspension that is stored on top of the stack. This may either cause reduction to resume combinator invocations that were suspended earlier, or to enter unwind again or to stop the entire reduction process.

The eleven steps above are performed in an order that is determined by the combinator used and the functional program under execution. The functionality of a program-derived combinator is conveniently described using Johnsson's abstract G-machine. Some of the elementary steps in the reduction algorithm are also G-machine instructions (e.g. UPDATE, EVAL). Other activities require several G-machine instructions (e.g. claim). The time spent in executing a G-machine instruction depends to a large extent on the store references that it generates. The same holds for standard combinator. Before we present the details of the counting procedures in

the last section of the paper, we must have a closer look at the primitive operators and data types that are used in both methods.

Primitive operators

Standard combinator include several more mathematical functions (e.g. SIN, LOG) than the G-machine. We have chosen to extend the G-machine with the mathematical functions required by the bench-marks, because it is laborious to express them using the available operators. The second category of primitive functions that are part of the standard combinator abstract machine but not of the G-machine instruction repertoire are: list append, equality test on lists and list projection. For both implementations we will implement these functions as part of the application programs, using head, tail, etc. These two modifications ensure that the sets of primitive operators in the two implementations are identical, because addition, multiplication etc. and also the printing of the results are part of any reduction machine. Now all primitive operators either manipulate scalar values or single constructor nodes; never lists. We will call our version of the G-machine the scalar G-machine.

Pattern matching

The bench-mark of functional programs that we have available uses pattern matching extensively. To run the bench-mark implies that efficient support for pattern matching has to be present in our reduction machines. The bench-mark programs use top-down left-to-right pattern matching, sometimes with repeated variables.

Pattern matching in Turner's method is supported for efficiency reasons by special combinators. [6] Similarly Augustsson [8] has proposed special abstract machine instructions to support efficient pattern matching in the G-machine. The only difference between Turner's and Augustsson's methods is that the latter combines common subpatterns of alternative clauses into one, whereas Turner does not. We have decided not to combine common patterns. The primitives used in both methods are then the same and pattern matching can be compiled into conditionals that perform selection of clauses and let expressions to bind subpatterns to variables. Both features can be compiled efficiently into scalar G-machine code. In the remainder of this section we will show that as far as pattern matching is concerned, the two implementation methods will be a close match. We start with an example of the way pattern matching operates with standard combinator and on the scalar G-machine. The function f of Figure 9 has two alternatives. The first applies when the actual argument is a list constructor, whose head is equal to 1. By default the second alternative applies. Figure 9(b) shows that the two alternatives of Figure 9(a) compile into two separate function definitions: f and f_1. The else-clauses of both generated conditionals of f specify that evaluation is to be resumed at f_1 but with the same argument x.

In the scalar G-machine a modified version of Augustsson's SPLIT instruction provides access to the head and tail fields of a list constructor. The instruction SPLIT is similar to PUSH, but instead of pushing a pointer to an argument, SPLIT pushes pointers on the stack, to the head and the tail of an argument that is a binary constructor. Note that in the scalar G-machine no vector application or vector constructor nodes exist. The JFUN instruction joins alternatives by jumping to the

```
f (l : b) = b                f x = if (pair x)
                                   (let a=head x and b=tail x in
                                        if(a=1) b (f1 x) )
                                   (f1 x)
f x       = 100              f1 x = l 0 0
```

(a) pattern matching (b) Conditionals and a *let*-expression

Figure 9. Compilation of pattern matching into conditionals

code of the next alternative in a function definition, while maintaining the current stack frame. The extensions to the G-machine must be matched by extensions to Johnsson's compilation schemes. Figure 10 presents the complete R-compilation scheme as it is used by our compiler to support pattern matching. The extensions to the E-, B- and C-schemes follow along the same lines.

Rules 2-6 propagate the R-scheme down the branches of conditionals and into let and letrec expressions. Rule 7 applies when the (lexically) last alternative of a function may still fail. Rule 9 applies when none of the other clauses apply. Finally rules 1 and 8 are there in support of pattern matching. Rule 1 of the R-scheme generates much better code for a let expression as introduced by the pattern-matching compiler than a combination of rules 2 and 5 would. Rule 8 represents a restricted optimization for tail calls that preserve the first k arguments of the current stack frame. The forms of rules 1 and 8 exactly fit the code produced by the pattern-matching compiler and should therefore be considered as a necessity to unify the primitives of both methods. Figure 11 represents the standard combinator code and the G-machine instructions for the function f.

```
1. R [if (pair x_i) (let y = head x_i and z = tail x_i in e_2) e_3] r n ⇒
                        PUSH (n - r x_i); EVAL; PAIR;      JFALSE l_1;   SPLIT (n - r x_i);
                        R [e_2] r' n';             LABEL l_1;
                        R [e_3] r n
                        where r', n' = (concatenate r [y = n+1, z = n+2] , n+2)
2. R [if e_1 e_2 e_3 ] r n ⇒ B [e_1] r n;          JFALSE l_1;
                        R [e_2] r n;               LABEL l_1;
                        R [e_3] r n
3. R [and e_1 e_2 ] r n ⇒ R [if e_1 e_2 false] r n
4. R [or e_1 e_2 ] r n ⇒ R [if e_1 true e_2] r n
5. R [let d in e ] r n ⇒ Clet d r n;               R [e] r' n'
                        where r', n' = Xr [d] r n
6. R [letrec d in e ] r n ⇒ Cletrec d r' n';       R [e] r' n'
                        where r', n' = Xr [d] r n
7. R [fail f ] r n ⇒ FAIL f
8. R [g x_1 ... x_k ] r n ⇒ POP (n-k-1);           JFUN g
                        where k < n
9. R [e ] r n ⇒ E [e] r n;                         UPDATE n; RETURN (n-1) otherwise

Xr     [v_1 = e_1 and... v_m = e_m] r n ⇒ (concatenate r [v_1 = n+1,... v_m = n+m], n+m)
Clet   [v_1 = e_1 and... v_m = e_m] r n ⇒ C [e_1] r n;... C [e_m] r (n+m-1)
Cletrec [v_1 = e_1 and... v_m = e_m] r n ⇒ ALLOC m; C [e_1] r n; UPDATE m;... C [e_m] r n; UPDATE 1
```

Figure 10. R-compilation scheme for F [f x, x_m = e] = R [e] [x_i = m+l ...x_m = 2] (m+ 1)

```
f=      TRY (Us (MATCH 1 I)) (K 100)
```

(a) Standard combinator code for *f*

```
f=      PUSH 0; EVAL, PAIR JFALSE l1;
           SPLIT 0 ;PUSH 1; EVAL; GET; PUSHBASIC 1; EQ; JFALSE 12;
              PUSH 0; EVAL; UPDATE 4; RETURN 3;
           LABEL 12; POP 2; JFUN f1;
        LABEL 11; JFUN f2;
f1=     PUSHNUMB 100 UPDATE 2; RETURN 1;
```

(b) Scalar G-machine code for *f*

Figure 11. Compiled versions of the pattern matching function f

We will now trace the execution of both compiled versions of f when applied to the tuple (2: 3). The relevant configurations of the graphs are represented for convenience as strings in Figures 12 and 13. In the actual graph reduction implementation pointers to the tuple (2:3) are duplicated rather than the tuple itself as suggested by the traces. The reducible standard combinator expressions are shown underlined in Figure 12. The TRY combinator at step 1 takes three arguments and builds applications of the first two arguments to the third. TRY then starts evaluation of its new first argument, such that the application of Us becomes the next redex (line 2). The invocation of TRY is then suspended. The Us combinator 'un-curries' its second argument after having verified that it is a constructor. Line 3 shows that the next combinator to be executed is MATCH. This combinator compares its first and third arguments, here to discover that they do not match. The application of FAIL, as generated by MATCH, is interpreted in such a way by the resumed TRY combinator (line 4) that it rewrites the current expression to the application of K. The latter simply discards the tuple and returns 100 as the net result. The indirection shown are generated to allow for one application to be overwritten with another (existing) application. In this case the print function of the reducer elides the indirection.

The reduction of f applied to the same tuple (2:3) on the scalar G-machine is somewhat easier to trace. Figure 13 shows successive states of the machine by listing the next instruction to be executed and relevant contents of the pointer and value stacks (s and v). The first two execution steps make sure that the argument of f is a weak head normal form (line 2). The PAIR instruction decides whether we have a constructor node and since this is the case it pushes TRUE onto the value stack (v). The JFALSE instruction skips to the next instruction, because the top of the value stack contains TRUE. The boolean is popped and we arrive at the SPLIT instruction (line 5). It pushes the pointers to the head and tail fields of our tuple onto the

```
1. TRY (Us (MATCH 1 I ) ) (K 100 ) (2 : 3)
2. TRY (Us (MATCH 1 I ) (2 : 3)) (K 100 (2 : 3))
3. TRY (MATCH 1 I 2 3) (K 100 (2 : 3))
4. TRY (FAIL 3) (K 100 (2 : 3))
5. I (K 100 (2 : 3))
6. I (I 100 )
```

Figure 12. Reduction of (f (2:3)) *with standard combinator*

	instruction	argument	pointer stack	value stack
1.	PUSH	0	((2 : 3) : @ : s)	(v)
2.	EVAL		((2 : 3) : (2 : 3) : @ : s)	(v)
3.	PAIR		((2 : 3) : (2 : 3) : @ : s)	(v)
4.	JFALSE	l_1	((2 : 3) : @ : s)	(TRUE : v)
5.	SPLIT	0	((2 : 3) : @ : s)	(v)
6.	PUSH	1	(3 : 2 : (2 : 3) : @ : s)	(v)
7.	EVAL		(2 : 3 : 2 : (2 : 3) : @ : s)	(v)
8.	GET		(2 : 3 : 2 : (2 : 3) : @ : s)	(v)
9.	PUSHBASIC	1	(3 : 2 : (2 : 3) : @ : s)	(2 : v)
10.	EQ		(3 : 2 : (2 : 3) : @ : s)	(1 : 2 : v)
11.	JFALSE	l_2	(3 : 2 : (2 : 3) : @ : s)	(FALSE : v)
12.	LABEL	l_2	(3 : 2 : (2 : 3) : @ : s)	(v)
13.	POP	2	(3 : 2 : (2 : 3) : @ : s)	(v)
14.	JFUN	f_1	((2 : 3) : @ : s)	(v)
15.	PUSHNUMB	100	((2 : 3) : @ : s)	(v)
16.	UPDATE	2	(100 : (2 : 3) : @ : s)	(v)
17.	RETURN	1	((2 : 3) : 100 : s)	(v)
18.			(100 : s)	(v)

Figure 13. Reduction of (f (2:3)) *on the scalar G-machine*

pointer stack. The next sequence of instructions (6-10) evaluate and compare the head of the tuple to 1 and push FALSE onto the value stack. The JFALSE instruction at line 11 transfers control to label l_2. The top two elements of the pointer stack, which were put there by SPLIT, are discarded by POP. The JFUN instruction at line 14 transfers control to the first instruction of the function f_1. This is a PUSHNUMB instruction that delivers a node with the return value 100. The UPDATE instruction overwrites the application of the original f to the tuple. This is schematically shown by replacing the stack element marked '@' by 100. In the actual graph-reduction machine, the appropriate node in the graph is overwritten. Finally RETURN discards the current stack frame.

To conclude the discussion on pattern matching we show that our example function f with both standard and program-derived combinator uses the same primitive functions. Us and PAIR, respectively, perform the first test, whereas MATCH and EQ perform the second. On both implementations (pointers to) the components of the tuple are accessed in one step (by Us and SPLIT). In the standard combinator approach TRY switches from the first to the second clause by overwriting a node in the graph with an indirection, whereas the scalar G-machine instruction JFUN must access the code for the function f_1.

Some of the advantages of the scalar G-machine are that with standard combinator the evaluator is called four times (once by TRY, Us and twice by MATCH) whereas the scalar G-machine only calls EVAL twice. Standard combinator require four new nodes in the graph (three by TRY and one by Us) whereas the scalar G-machine only requires one (by PUSHNUMB). After a short introduction into some implementation details we will return to these issues and present several more performance parameters.

Primitive data types and the implementation of unwind

The unwind step in a combinator graph reducer is executed often, because it provides the combinator access to their arguments. We will briefly review an efficient implementation method for unwind. For this method a node in the graph must be large enough to hold a tag and either two pointers or a scalar. There are two types of nodes that require pointers: application nodes and constructor nodes. A scalar can be a character, a boolean, NIL or a floating point number. Strings are represented as lists of characters. We will assume that a representation of a floating point number that takes as much space as two pointers is satisfactory. For the tag of a node enough space is reserved to store an instruction that can be executed by the target machine. On a machine such as the VAX, two bytes are sufficient to store the tag. The tag of an application node is the opcode of a 'jump to subroutine' instruction (jsb) and that of a scalar or constructor node a 'return from subroutine' instruction (rsb). Figure 14 shows the two unwind steps that are necessary to invoke an application of the combinator PLUS to another application and a numeric argument 10. Initially we assume that the program counter holds the address of the top left jsb instruction, which is the same as saying that the reducer is given the address of the top left application node to evaluate. Executing the jsb instruction causes the address of the next word to be pushed onto the stack, which holds a pointer to the *rsb* instruction. Execution then continues at the middle jsb instruction. After this instruction has been executed, the program counter will point at the first instruction of the PLUS combinator. The stack now holds pointers to the right fields of both application nodes, which look like ordinary return addresses. However, these return addresses will never by used by any rsb instruction. Instead, the PLUS combinator uses them to access its arguments via two indirection, one in the stack and one in the heap.

In the target machine code for the combinator PLUS there must be two calls to evaluate to make sure that both arguments are indeed numbers before we add them up. Such calls can be implemented by a single jump to subroutine instruction, where the pointer to the appropriate argument of the combinator is the subroutine. The numeric argument will immediately return control. The other argument will be unwound into recursively, until at some point in time it becomes a weak head normal form. Control then returns to PLUS for the addition.

For correctly typed programs, there is no need to perform an argument sufficiency check. Once a function application becomes needed, it must have all its arguments present. If not this is an error. That this must be the case in a system that implements lazy evaluation can be shown as follows. No application will ever be evaluated unless

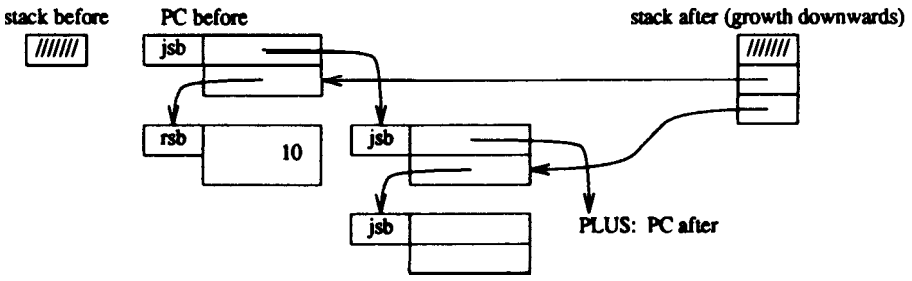


Figure 14. Two unwind steps that invoke (PLUS (. . .) 10)

a primitive function demands its value. We assume here that the 'need to print' also acts as a primitive function in the sense that it demands values. In the G-machine no primitive function exists that will accept a partial application as one of its strict arguments. In Turner's combinator machine, since it is run-time typed, type testing functions are provided that will accept partial applications on strict argument positions. For example an expression such as 'if (function x) (x 1) x' first reduces x to weak head normal form and if x is indeed a function, it is applied to 1 and reduced again. Our bench-mark applications however, do not use type testing functions in this way. As a consequence lazy evaluation of correctly typed programs never requires argument sufficiency checks. For some implementation purposes it may be useful to perform argument sufficiency checks after all. For instance to report the name of the function in case it is applied to insufficient arguments. In the experiments described here, we have run properly working programs only, so the argument sufficiency test was not needed and indeed not implemented.

Standard combinator such as S, which return an application rather than a scalar or a constructor node, also benefit from this method of implementing unwind. Such combinator change the pointer field(s) of their root application node and then in principle return. This causes the pointer to the root application node to be popped off the stack, together with the return address that was put there by evaluate. Since evaluate must insist on a data value rather than an application, all it can do is unwind into the same application node again. evaluate and unwind will push the same pointers back on the stack that were there before. Rather than return, a combinator that yields an application should 'jump' directly to the address it just stored in the left field of its top application node (see Figure 15).

Short-circuiting the return / unwind combination not only saves store references, it also avoids complication and time penalties in the reducer. Without short-circuiting, evaluate has to check whether the evaluation of an application node results in a data object or an application again. Furthermore evaluation has to be retried until finally a data object emerges. Both the repetition and the check can be avoided by the short-circuit implementation. Note that short-circuiting can hardly be considered an optimization, because it is both easier to implement and more intuitive: if we know what to do next then why forget it and immediately discover it again? Another advantage of the so-called 'threaded code' implementation of unwind is that the only store references made are to the graph and the stack, but not to the code. The idea was first put forward by Augusteijn,[9] although he did not use a return from subroutine instruction but a routine call to return the appropriate value explicitly to the caller.

Interaction with the storage manager

Although the cost of storage allocation and reclamation is not a major issue in this paper, there is some interaction between the allocator and the reducer that

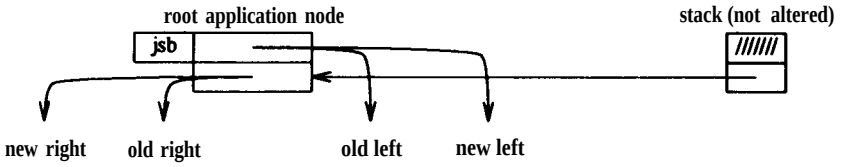


Figure 15. Short-circuit implementation of return / unwind

requires consideration. First we have decided to use boxed objects only (i. e. each data object is stored in a cell all by itself) and scalar objects of enumeration types are hyper-shared. This means that there is never more than one node around with, for instance, the boolean value TRUE, the character 'a', a standard combinator S, a program derived combinator etc. Such an organization is easy to implement cheaply except for numerals.[6] The PUSHNUMB instruction of the scalar G-machine carries a pointer to the node with the required number rather than the number itself. The number is allocated in the graph during the initial loading of the program, in much the same way as numbers are allocated during loading of a standard combinator program. Similarly the PUSHFUN instruction carries a pointer to the code of the appropriate function.

Communication between the reducer and the storage allocator takes place via registers that contain pointers to the next free cell in the heap etc. Both a test to check that enough free cells are available and the allocation of a cell can therefore be implemented cheaply and we do not account for this cost here. The store accesses required to fill the newly allocated cells with data are the responsibility of the reducer and are accounted for.

MEASUREMENTS

To summarize our efforts so far, we have selected two implementation methods of combinator graph reduction. Both were modified to make sure that application programs use the same set of primitive data objects and operators. The performance differences that will be experienced by an application program can thus be attributed entirely to the two differences in the reduction methods: fully lazy standard combinator with naive graph reduction versus non-fully lazy program derived combinator with short-circuiting graph reduction.

Our efforts to reconcile the primitives of both methods immediately pay off, because the time spent in execution of a primitive does not depend on the particular implementation method. We have been careful, though, not to include the time spent to fetch the operands and to store the result in the time to perform, for instance, a multiplication or a read from a file. The total time spent in executing primitives provides us with a lower bound on the execution time of a particular application; only a change in the application itself may change it. We would like to see all housekeeping costs as low as possible. In the next three paragraphs we will introduce the performance parameters associated with the housekeeping cost that are common to both implementation methods, the standard combinator specific parameters and those that apply only to the scalar G-machine.

Common performance measures

The eleven steps that we introduced before are responsible for the total execution time of a functional program. The first three steps (initialize, load and print) all take a certain amount of time, but we assume this to be negligible compared to the total execution time. We will therefore concentrate on the eight remaining steps. The eight steps require execution of instructions and accessing operands in store. Without a particular target architecture in mind we cannot say much about how many instructions are necessary to perform one of the eight steps. We will distinguish

between accesses to the stack(s), a tag of a cell in the graph or a (pointer) field in a cell. An access to a floating point number counts as two field accesses. Read and write accesses are counted separately.

unwind: Unwinding a single application node as in Figure 14 causes one tag, one field and one stack access.

call: When the (scalar) G-machine starts executing a program-derived combinator, it rearranges the current stack frame such that the stack contains pointers to the arguments rather than to the application nodes that point at the arguments. Each unwound application node incurs the cost of two stack accesses (one to read and one to write an entry) and one field access. With standard combinator there is no need to rearrange the stack frame; the cost of the indirection are attributed to the category access below. With program-derived combinator it is necessary to know the number of arguments, because of the stack rearrangement. As we argued before, sufficient arguments will always be present in lazily evaluated, correct programs. Similarly all standard combinator (except the variable arity TRY combinator) will assume that the required arguments are present.

access: Accessing a pointer to an argument on the (scalar) G-machine requires one stack read. A standard combinator generates one stack and one field access per argument. The cost of accessing the contents of an argument depends on its type and is separately accounted for in the category combine,

claim: The cost of allocating a cell depends on the particular storage allocator used and the size of the heap. The influence of the application program on the relevant cost factors is minimal.[6] This implies that for all applications a suitable heap size can be found such that the cost of allocating a cell is always the same. It is therefore enough to record the total number of claimed cells only. Filling a new cell requires one tag and two field accesses.

evaluate: To start reducing an expression to weak head normal form requires two stack accesses to push the current PC and the pointer to the expression to be evaluated.

combine: Under this heading we have collected all operations that would not fit into any of the other categories. It therefore depends on the particular abstract machine instruction how many store references fall into this category (see Tables II and III).

update: An update on the (scalar) G-machine requires two argument pointer accesses, two tag accesses and four field accesses, because the contents of one node in the graph has to be transferred to another. Pointers to both nodes reside in the stack. Standard combinator that produce an application (such as K and TRY) use an indirection node to update one application with another application. Standard combinator that produce a data value or constructor (such as PLUS and Sp) store their result directly into the appropriate application node and update the tag of the application node. The cost of an update thus depends on the particular standard combinator and is separately accounted for.

return: The RETURN instruction of the scalar G-machine, or execution by a standard combinator that returns a scalar or constructor, pops the PC and the pointer to the evaluated expression from the stack. Standard combinator that return an application short-circuit the return / unwind by a jump (see Figure 15). This saves an unwind and a return, which explains the origin of the negative numbers in Table (I).

Table I provides a summary of the numbers of read and write accesses generated by the eight steps discussed above.

Performance measures for standard combinator

The common performance parameters will now be added to standard combinator specific costs. The PLUS combinator, for instance, must access the contents of its two arguments and update the tag and the fields of the result. In addition to the two unwind, two evaluate and two access steps and the single return step, the PLUS combinator generates 4+2 field accesses and 0+1 tag access as part of the update and combine costs. Adding these all up we obtain a total of 4+6 stack, 8+2 field and 2+1 tag accesses. Table II summarizes the costs involved in each of the standard combinator. The data under the heading combine & update cover the (read + write) store accesses that a combinator generates when building the required graphical structure. Pointers to existing structure are supposed to be available in high-speed registers. The cost of setting these may be found in the columns access and unwind. The column 'other' shows whether a combinator returns control to its caller or short-circuits the next unwind. The associated costs can be found in Table I. Some combinator have more than one entry in the table, because they can follow an execution path that depends on the arguments. For instance the equality test (=) generates four more field accesses when the tags of its argument are equal.

Performance measures for scalar G-machine instructions

In the scalar G-machine the instructions UNWIND, EVAL, UPDATE and RETURN are devoted to one or more steps in the reduction algorithm. The remaining instructions

Table I. Cost summary of the major eight steps of the reduction algorithm (read + write)

	Standard combinator			Program derived combinator		
	stack	field	tag	stack	field	tag
unwind	0+1	1+0	1+0	0+1	1+0	1+0
call				1+1	1+0	
access	1+0	1+0		1+0		
claim		0+2	0+1		0+2	0+1
evaluate	0+2			0+2		
combine		see Table II			see Table III	
update		see Table II			see Table III	
return	2+0			2+0		
jump	-(0+1)	-(1+0)	-(1+0)		not applicable	

Table II. Cost summary of standard combinator (read + write)

Standard combinator	unwind	access	claim	evaluate	combine & update			other
					stack	field	tag	
=,~= different tags	2	2		2			2+1	return
=,~= same tags	2	2		2		4+0	2+1	return
>=, >, <=, <	2	2		2		4+0	0+1	return
AND FALSE, OR TRUE	2	1		1			1+1	return
AND TRUE, OR FALSE	2	1		1		0+1	1+0	jump
B, C	3	3	1			0+2		jump
Bp, cp	3	3	1			0+2	0+1	return
B1, C1	4	4	2			0+2		jump
CHAR, LIST, NUMBER, PAIR	1	1		1			1+1	return
CODE	1	1		1		1+2	0+1	return
COND	1	1		1		0+2	1+0	jump
DECODE	1	1		1		2+1	0+ 1	return
DIV, MOD, +, –, *, /	2	2		2		4+2	0+1	return
FAIL applied to argument	1						0+1	return
HD, TL	1	1		1		1+2		jump
I	1	1			2+0	0+1		
K	2	1			1+0	0+2		jump
MATCH fails and different tags	3	3		2			2+1	return
MATCH fails and same tags	3	3		2		4+0	2+1	return
MATCH succeeds	3	3		2		4+2	2+0	jump
NEG, SIN	1	1		1		2+2	0+1	return
NOT	1	1		1			1+1	return
S	3	3	2			0+2		jump
Sp	3	3	2			0+2	0+1	return
S1	4	4	3			0+2		jump
TRY with three arguments	3	3	3		1+0	0+2		jump
TRY with two and first fails	2	1		1		0+1	1+0	jump
TRY with two and first succeeds	2	1		1		0+2	1+0	jump
U	2	2	3			0+2		jump
Us on pair	2	2	1	1		0+2	1+0	jump
Us fails	2	2		1			1+1	return
Y	1	1				0+2		jump

perform store accesses that fall in the categories shown in the header of Table III. The negative numbers in the entry for ALLOC originate from the assumption made in Table I that a cell when claimed is immediately initialized. This is not necessary for the 'holes' claimed by ALLOC, hence the compensation of two field and a tag write per hole. The cost assigning values to the fields of a hole is accounted for by UPDATE.

The data in Tables I–III pertain to particular implementations of the standard combinator and G-machines. Other implementations will give different numbers. For instance, Turner's own implementation does not use threaded code to implement unwind, which makes his unwind more time-consuming. On the other hand Turner

Table III. Cost summary of scalar G-machine instruction (read + write)

Scalar G-machine instruction	access	claim	combine				other
			value	pointer	field	tag	
>=, >, <=, <, =,~=			4+1				
ALLOC of one cell		1		0+1	-(0+2)	-(0+1)	
CHAR, LIST, NULL, NUMBER, PAIR	1		0+1			1+0	
CODE			1+2				
DECODE			2+1				
DIV , MOD, +, –, *, /			4+2				
EVAL	1						evaluate
GET boolean	1		0+1			1+0	
GET character	1		0+1		1+0		
GET number	1		0+2		2+0		
HD, TL	1			0+1	1+0		
JFALSE			1+0				
JUMP, JFUN, POP							
MKAP, MKCONS	2	1		0+1			
MKBOOL			1+0	0+1			
MKCHAR			1+0	0+1	1+0		
MKNUMB		1	2+0	0+1			
NEG, SIN			2+2				
NOT			1+1				
PUSH, SLIDE	1			0+1			
PUSH BASIC boolean or character			0+1				
PUSH BASIC number			0+2				
PUSHBOOL, PUSHCHAR, PUSHFUN				0+1			
PUSHNIL, PUSHNUMB				0+1			
RETURN							return
SPLIT	1			0+2	2+0		
UNWIND							unwind + call
UPDATE	2				2+2	1+1	

reduces fewer indirection (applications of the | combinator), for instance, when TRY or MATCH return fail. This makes our implementation more expensive. The tables provide the basis for a comparison between the methods and care has been taken to make the implementations as similar as possible while exploiting to the full their specific advantages.

Bench-mark performance

Our bench-mark applications are written in SASL, a run-time typed lazy functional language. [10] The bench-mark set contains six medium size programs, varying in size from 170 to 2700 lines of program text (excluding comments, blank lines etc). Most programs are run on small input data sets: em script runs a simple script through a

Table IV. Global characteristics of the bench-mark programs

		em	gcode	lambda	qsort	sched	wave
Defined functions		48	198	114	14	35	84
Executed function calls		139,913	64,769	8157	51,150	13,373	152,400
Size of the compiled code							
Standard	(nodes)	4024	23,566	4777	2985	1598	1692
G-machine	(instructions)	5040	22,826	6465	2514	1521	2309
Number of tractable accesses of the stack(s) and the graph							
Standard	(Acc_s)	16,792,387	18,703,812	1,729,067	16,994,803	7,608,393	9,466,054
G-machine	(Acc_g)	6,856,242	4,665,480	553,278	3,991,856	1,052,316	11,120,830
Number of claimed cells							
Standard	($Claims_s$)	733,742	966,963	77,128	1,131,047	477,243	424,623
G-machine	($Claims_g$)	245,008	144,709	18,058	162,199	42,246	403,645
Ratio standard/G-machine of store accesses and cell claims							
Access ratio	(Acc_s/ Acc_g)	2·4	4·0	3·1	4·3	7·2	0·9
Claim ratio	($Claims_s$/ $Claims_g$)	3·0	6·7	4·3	7·0	11·3	1·1
Number of claimed cells per 100 tractable accesses of the store							
Standard	($Claims_s$ x 100/ Acc_s)	4·37	5·17	4·46	6·66	6·27	4·49
G-machine	($Claims_g$ x 100/ Acc_g)	3·57	3·10	3·26	4·06	4·01	3·63
Number of arithmetic operations per 100 tractable accesses of the store							
Standard	(Arith. ops_s x 100/ Acc_s)	0·13	0·05	0·17	0·14	0·01	1·02
G-machine	(Arith. ops_g x 100/Acc_g)	0·31	0·22	0·56	0·56	0·06	1·21

functional implementation of the UNIX text editor; [11] gcode compiles the qsort program into scalar G-machine code according to the compilation schemes as described in Johnsson's paper, lambda (S K K) evaluates to I on an implementation of the λ–K calculus; [12] qsort (sin 1, ... sin 1024) sorts a list of 1024 real numbers using quick sort; sched 7 calculates an optimum schedule of seven parallel jobs with a branch and bound algorithm; [13] wave 3 predicts the tides in a rectangular estuary of the North Sea over a period of 3×20 minutes. [14] Qsort, sched and wave are programs, suitable for parallel job reduction, [13] but evaluated sequentially. Table IV provides a further indication of the properties of the bench-mark programs. One remarkable observation that can be made from the table is that both methods generate about the same number of 'codes' (scalar G-machine instructions or application, constructor or scalar nodes in case of standard combinator). Loaded in store a G-code program would be smaller than a standard combinator program because a G-machine instruction generally requires less space than an application node. From the experimental data we guess that the order of complexity for G-machine code is the same as for standard combinator.

We ran the bench-mark programs on both reducers and counted the basic steps of the reduction algorithm and the number of executed standard combinator or scalar G-machine instructions. These counts were weighted and added using Tables I, II and III to yield the total number of tractable accesses of the store made by the implementation of the reduction strategy and the combinator. With our bench-marks the standard combinator machine generates up to seven times more store accesses. It also claims up to eleven times more cells than the scalar G-machine.

The performance of the wave program is sensitive to the fact that program derived combinator (according to Johnsson) are not fully lazy. The wave program heavily relies on operations on two-dimensional matrices, which can only be implemented in SASL as lists of lists. Figure 16 shows the definition of the access function of an array element (sub_1) and that of a matrix element (sub_2). The last line of the figure shows a typical use of sub_2; it selects the first, second, fourth and eighth elements of the third row of a matrix. The G-machine implementation treats sub_2 as a combinator. In the case of the example it is therefore instantiated four times, rather than once as the standard combinator machine does. This implies that the redex (sub, matrix 3) appears four times as well. It cannot be shared and therefore causes a non-fully lazy implementation to do more work than a fully lazy one.

If we take the increase in the number of arithmetic operations performed as an indication for the amount of recalculation we find that the wave program does 39 per cent more work on the scalar G-machine. On top of this there is additional housekeeping for the reduction strategy etc. The lambda program also suffers from this effect, but to a much lesser extent: it needs 4 per cent more arithmetic operations. For the remaining programs full laziness is irrelevant.

```
sub1 (head:tail)      1          = head
sub1 (head:tail)      index      = sub1 tail (index - 1)
sub2 matrix      row  column     = sub1 (sub1 matrix row) column

map (sub2 matrix 3) (1,2,4,8)
```

Figure 16. Array and matrix operations used by wave

An estimate of the maximum speed-up of the G-machine over standard combinator

Now that we know the number of tractable accesses of the stack and the heap, and the number of claimed cells for both implementation methods, we can use these figures (as shown in Table IV) with equation (2) to estimate an upper bound for the ratio (3) below. This yields the maximum attainable performance increase of the scalar G-machine over standard combinator:

$$\text{ratio} \approx \frac{(\text{InsPerAcc}_s + 1) \times \text{TimePerAcc} \times \text{Acc}_s + \text{TimePerClaim}_s \times \text{Claims}_s + \text{PrimTime}_s}{(\text{InsPerAcc}_g + 1) \times \text{TimePerAcc} \times \text{Acc}_g + \text{TimePerClaim}_g \times \text{Claims}_g + \text{PrimTime}_g} \tag{3}$$

Although (3) represents a good approximation to the performance gain that we are interested in, it contains too many unknown parameters to use it directly. However, by making some estimates we can determine an upper bound for (3). We need an auxiliary result first: it can be proved by induction on n that when all variables involved are positive numbers we have

$$\frac{\sum_{i=1}^{n} x_i}{\sum_{i=1}^{n} y_i} \leqslant \max_{i=1}^{n} \frac{x_i}{y_i} \tag{4}$$

Combining this inequality with (3) we obtain

$$\text{ratio} \leqslant \max\left[\frac{(\text{InsPerAcc}_s + 1)}{(\text{InsPerAcc}_g + 1)} \times \frac{\text{Acc}_s}{\text{Acc}_g}, \frac{\text{TimePerClaim}_s}{\text{TimePerClaim}_g} \times \frac{\text{Claims}_s}{\text{Claims}_g}, \frac{\text{PrimTime}_s}{\text{PrimTime}_g}\right]$$

Because of the way the experiments have been set up, $\text{PrimTime}_s = \text{PrimeTime}_g$ when full laziness is not important. This is the case for the bench-mark programs em, gcode, qsort and sched. Therefore, restricting our attention to these programs, we may replace the PrimTime ratio by 1.

The remaining unknown parameters are InsPerAcc and TimePerClaim, whose values depend on the particular target architecture and instruction mix generated by reducer and garbage collector. As our purpose is to compare two implementation methods of lazy graph reduction, we must keep the other parameters that influence performance constant. Therefore we must use the same underlying target architecture, operating system and garbage collector. By using the same garbage collector for both methods we would expect $\text{TimePerClaim}_s \approx \text{TimePerClaim}_g$. This means that TimePerClaim is expected to be insensitive to changes in the structure of the graphs that the garbage collector operates on. This is confirmed by the result reported earlier that TimePerClaim does not vary by more than 20 per cent as a result of running different bench-mark programs on the same reducer.

The ratio of the average number of instructions per access depends on the target architecture and the instruction mix generated by the reducers. As we do not change the target architecture, this ratio must be near 1, unless the instruction mixes are rather different. This, however, is unlikely as the instruction mix in all cases mainly

consists of 'move' instructions to manipulate graph or stack and 'jump/return from subroutine' instructions for unwind and return. Most of the work inherent in such instructions is in the data movement. If a 'jump to subroutine' instruction for instance takes twice as long to execute as a 'move' instruction, it is because the former moves twice as much data around. The instructions that do more than simple data movement are those generated by the primitive operations, which are accounted for separately by the terms $PrimTime_s$ and $PrimTime_g$. As a result we may assume that $(InsPerAcc_s + 1) \approx (InsPerAcc_g + 1)$. Again we take the fact that InsPerAcc does not vary by more than 20 per cent when running different bench-marks on the same reducer as an indication that this assumption is realistic.

Combining the above considerations with the fact that for all bench-mark programs the claim ratio exceeds the access ratio, we arrive at the following upper bound of the performance gain:

$$\text{ratio} \leq \frac{Claim_s}{Claims_g} \tag{5}$$

The accuracy of the upper bound (5) is determined by the weights of the ratios used in (4). For instance from (4) we have $(10 + 2)/(9 + 1) < 2$, which is not very close to the real value 1-2 of the ratio. The upper bound (5) we found for (3) corresponds to the ratio with the largest weight, because the cost of allocation and reclamation of cells, even on the G-machine often exceeds the cost of reduction. When comparing the performance of reference counting, mark-scan and two-space copying garbage collection,[6] we found the number of stack, heap and instruction store accesses per claimed and subsequently recovered cell to range from 10 to 200. For reference counting it is almost a constant (100), whereas the cost of the other two methods is proportional to the reciprocal of the heap size. The figure of 10 accesses per cell applies when no garbage collection is performed. For the G-machine, the number of claimed cells per 100 tractable accesses to the store lies between 3 and 4 (see Table IV). Therefore the storage manager generates between 3×10 and 4×200 references for each 100 references generated by the reducer. Unless heap space is abundant, the cost of storage management will dominate the cost of reduction.

Concluding, we may state that the maximum performance gain of the scalar G-machine over the standard combinator machine is determined by the claim ratio, which has values of up to 11 $\times$. This is the first conclusion we may draw from our experiments.

A profile of combinator graph reduction

Some more detailed results of the experiments are summarized in Tables V and VI. Both present the number of accesses to store sorted by particular categories. The percentages shown are averages taken over the entire bench-mark suite and all percentages fall within the pleasingly narrow intervals as shown in the tables. The percentage of accesses spent in each of the eight major basic steps of the reduction algorithms is presented in Table V. In the column marked return we report the sum of the cost of returning from function calls and returning after redundant calls to evaluate (i.e. calls to evaluate when the object to be evaluated is already in weak head normal form). The basic step that requires most accesses to the store is combine on the G-machine and unwind on the standard combinator machine.

Table V. Percentages of store accesses sorted by basic steps of the reduction algorithm

	unwind	call	access	claim	evaluate	combine	update	return
Standard	32±1	—	28±2	16±4	3±2	7±3	11±1	3±1
G-machine	8±2	7±2	22±2	11±2	7±1	27±4	11±5	7±1

Table VI. Percentages of store accesses sorted by store area

	Value stack	Pointer stack	Graph field	Graph tag	Total
Standard	—	36±4	46±3	18±1	$Acc_s = 100$
read + write	—	22+14	25+21	12+6	59+41
G-machine	10±5	58±5	22±2	10±1	$Acc_g = 100$
read + write	5+5	33+25	12+10	5+5	55+45

An optimization that Johnsson presents in his paper is how to avoid redundant calls to evaluate. With standard combinator we found that in 25 to 50 per cent of the cases, depending on the bench-mark, evaluate was indeed applied to a weak head normal form. The G-machine figures range from 50 to 70 per cent. Avoiding redundant calls to evaluate (and the associated return) may reduce accesses to the store by at most 10 per cent. Combined with another optimization that Johnsson proposes (compile time simulation of the contents of the pointer stack) we find that the maximum improvement can be 16 per cent, if all calls to evaluate (and therefore also to return) are avoided as well as all stack updates by the PUSH instructions. In practice the benefit will be less.

An interesting peephole optimization to G-machine code is described by Kieburtz and Agapiev. [15] When the result of a reduction is a pair or a basic value held in the value stack, a new heap cell is not allocated to hold it. Instead the result is used to update the application node at the root of the redex directly. This requires additional instructions in the abstract G-machine to replace instruction patterns such as MKNUMB / UPDATE / RETURN. Other commonly occurring patterns, such as EVAL / UPDATE / RETURN cannot be optimized out in this fashion, because the node to update with may come from anywhere. The maximum gain from the peephole optimization is therefore attained if all update activity can be avoided (16 per cent). This is also the result that may be obtained if we can avoid redundant updates, which is the aim of the spineless G-machine. [16] The designers of the spineless G-machine report results for small programs only.

Table VI shows another way of sorting store accesses. The data show that the G-machine spends about 10 per cent of its accesses on the value stack, where it stores the operands that it performs most primitive operations on.

Apart from software-oriented efforts, there are projects involved in designing special hardware for the G-machine. Here the data of Table VI may provide useful information because an option that a hardware designer has is to use caches to speed up access to various regions of store. For instance keeping (the top of) the pointer stack in a cache may save up to half the memory references generated by the G-

machine [17] Recent work of Kieburtz [15] contains a detailed analysis of simulated G-machine RISC performance that is similar to ours, but based on assumptions that make the results less applicable to conventional architectures. The G-machine RISC is inherently a tagged architecture with hardware support for stack operations. The processor is not always required to wait for a write operation to the graph to complete. This means that the weight of all categories in Table VI except the read operations on the graph is lower in Kieburtz' analysis than in ours.

In a project at our own institute a distributed G-machine is being designed that exploits parallelism at the program-derived combinator and G-machine instruction levels [19] For instance the distributed G-processor can perform an update in one read and one write cycle instead of the eight cycles that a conventional architecture requires (see Table I). This implementation of update may improve performance by up to 16 per cent.

CONCLUSIONS

The performance of two implementations of lazy combinator graph reduction has been compared. Johnsson's implementation based on program-derived combinator and short circuiting of graph reduction generally gives better results than Turner's standard combinator implementation because both having program-derived combinator and avoiding naive graph reduction reduce the number of cell claims.

For a bench-mark of medium-sized programs we found that Turner's method causes up to eleven times more cell claims than the G-machine and that Turner's reducer alone (i.e. exclusive of the storage manager) generates up to seven times more references to the store. Unless heap space is abundant we found that allocation and reclamation of heap cells requires at least as many references to the store as reduction proper. Assuming that there is a high correlation between the number of store accesses and execution time, we conclude that programs may run up to about 10 times faster on the G-machine. Our G-code compiler does not use any of the optimizations that have been published since Johnsson's 1984 paper and it uses a restricted form of Johnsson's optimization of tail recursion.

One of our bench-mark programs causes the G-machine to access the store more often than Turner's machine because the G-machine compiler does not use fully lazy lambda lifting. The performance of the other bench-mark programs is not affected significantly by the loss of full laziness.

In both methods of combinator graph reduction a considerable fraction of the number of store accesses is spent on function calls and parameter passing. Turner's reducer spends over 60 per cent of its accesses to the store on function calls and parameter access. The G-machine uses 30 per cent of its accesses to the store for this purpose.

The major difference between the current work and that reported by other researchers is that we have used medium size as opposed to small programs. Our bench-mark of functional programs could be used only because we have spent considerable effort in reconciling the primitive operands and operations used in both reduction methods. With real programs, pattern matching is an important part of the set of primitives. Without a common set of primitives such as the one that we have developed, there is no hope for sensible performance comparisons of any but small programs.

We use a model of reduction that captures all time-consuming aspects of an implementation in such a way that time can be measured in memory accesses. Not included in our measurement are the primitive operations, because the time spent in, for instance, a floating point division is not easy to relate to a number of memory accesses in a machine independent way. As a result our model gives an upper bound on the performance gain that may be achieved with better implementation techniques.

ACKNOWLEDGEMENTS

I am grateful to Marcel Beemster, Hugh Glaser, R.B. Kieburtz, Keen Langendoen, Rob Milikowski, Wim Vree and the referees for their comments on drafts of the paper.

REFERENCES

1. D. A. Turner, 'A new implementation technique for applicative languages', *Software—Practice and Experience*, **9,** (1) 31–49 (1979).
2. T. Johnsson, 'Efficient compilation of lazy evaluation', *SIGPLAN* Notices, **19,** (6) 58–69 (1984).
3. R. J. M. Hughes, 'Super combinators—a new implementation method for applicative languages', *ACM Symposium on Lisp and functional programming,* ACM, Pittsburg, Pennsylvania, August 1982, pp. 1-10.
4. R. J. M. Hughes, 'The design and implementation of programming languages', *Ph.D. Thesis,* Oxford University, U. K., July 1983.
5. L. Augustsson and T. Johnsson, 'The Chalmers Lazy-ML compiler', *The Computer Journal,* **32,** (2), 127–141 (1989).
6. P. H. Hartel, 'Performance analysis of storage management in combinator graph reduction', *Ph.D. Thesis,* Department of Comp. Syst, University of Amsterdam, October 1988.
7. T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer and M. J. Plasmeijer, 'CLEAN: a language for functional graph rewriting', in G. Kahn (cd.), *Third Conference on Functional Programming Languages and Computer Architecture, LNCS 274* Springer Verlag, Portland, Oregon, September 1987, pp. 366–384.
8. L. Augustsson, 'Compiling pattern matching', in J.-P. Jouannaud (ed.), *Second Conference on Functional Programming Languages and Computer Architecture, LNCS 201,* Springer-Verlag, Nancy, France, September 1985, pp. 368–381.
9. L. Augusteijn, 'A translation of combinator graphs into machine code', in *Symposium on the 'Stare of the Art',* Nederlands genootschap voor informatica, Utrecht, Netherlands, April 1984, pp. 362–369.
10. D. A. Turner, 'SASL language manual', *Technical report,* Computing Laboratory, University of Kent at Canterbury, August 1979.
11. P. W. M. Koopman, 'Interactive programs in a functional language: a functional implementation of an editor', *Software—Practice and Experience,* **17,** (9), 609–622 (1987).
12. P. H. Hartel and A. H. Veen, 'Statistics on graph reduction of SASL programs', *Software—Practice and Experience,* **18,** (3), 239–253 (1988).
13. W. G. Vree, 'Design considerations for a parallel reduction machine', *Ph.D. Thesis,* Department of Comp. Syst, University of Amsterdam, December 1989.
14. W. G. Vree, 'The grain size of parallel computations in a functional program', in E. Chiricozzi and A. d'Amico, (eds), *Conference on Parallel Processing and Applications,* Elsevier Science Publishing, L'Aquila, Italy, September 1987, pp. 363–370.
15. R. B. Kieburtz and B. Agapiev, 'Optimizing the evaluation of suspensions', in T. Johnsson, S. L. Peyton Jones and K. Karlsson (eds), *Proc. of the Workshop on Implementation of Functional Languages,* Department of Computer Science Chalmers University of Technology, Goteborg, Sweden, September 1988, pp. 267–282.
16. G. L. Burn, S. L. Peyton Jones and J. D. Robson, 'The spineless G-machine', in *ACM Symposium on Lisp and Functional Programming,* ACM, Snowbird, Utah, July 1988, pp. 244–258.

17. R. B. Kieburtz, 'The G-machine: a fast, graph-reduction evaluator', in J.-P. Jouannaud, (ed.), *Second Conference on Functional Programming Languages and Computer Architecture, LNCS 201*, Springer-Verlag, Nancy, France, September 1985, pp. 400–413.
18. R. B. Kieburtz, 'A risc architecture for symbolic computation', *SIGPLAN Notices*, **22,** (10), 146–155 (1987).
19. R. Milikowski, 'A distributed G-processor', *PRM Project internal report D-30*, Department of Comp. Syst, University of Amsterdam, August 1989.