

Assessing the Evaluation Transformer Model of Reduction on the Spineless G-machine*

Sigbjørn Finne and Geoffrey Burn

Department of Computing, Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, United Kingdom

Abstract

This paper reports on our initial work on assessing how using the evaluation transformer model of reduction affects the performance of lazy functional programs. The model uses information about how much evaluation of an expression is required in order to evaluate an expression as much as possible, as early as possible. Our results show that there is a definite gain over just using strictness information, but that it is difficult to characterise exactly how much gain there will be, and in what programs it will occur.

1 Introduction

A large number of papers have been written on various analysis techniques for lazy functional languages, but there has been little systematic work determining how this information can be used most effectively in an implementation. The folklore says that exploiting strictness information is beneficial, especially when combined with passing values unboxed, but this conclusion has not been reached by careful experimentation, and without careful experiment we run the risk of ignoring other beneficial optimisations.

To the best of our knowledge, Hartel is the only person who *has* made any systematic study in this area [Har91a, Har91b]. Our work complements and extends the work of Hartel: it complements his work because we consider some of the same questions, but using a different abstract machine for our experiments; it extends it because we consider a more complicated computational model.

This paper reports on our initial work on assessing how the *evaluation transformer model of reduction* [Bur87, Bur91] affects the performance of programs run on the *Spineless G-machine* [BPR88].

The key idea of the evaluation transformer reduction model is that the amount of evaluation needed of an argument to a function depends on the amount of evaluation required of the application of that function. This model is described further in Section 2.1.

*Questions and comments on the paper should be directed to the second author (glb@doc.ic.ac.uk). This research was partially funded by SERC grant GR/H 17381 ("Using the Evaluation Transformer Model to make Lazy Functional Languages more Efficient").

The main differences between the Spineless G-machine and the earlier G-machine models of Augustsson and Johnson are that the compiler can specify when sharing might occur; that expressions are only updated when a WHNF is obtained; and that there is no need for (a chain of) spine nodes for an expression. We chose this machine because it is a good level of abstraction at which to express the alternative models of reduction. It is described further in Section 2.2.

As a first step in understanding how to use evaluation transformer information, we selected a benchmark suite of medium-sized programs, put them through an abstract interpretation system to determine the evaluation transformers [Hun89], and then generated four different variants of the code: a lazy variant; a variant which used only strictness information¹; and two variants which used evaluation transformer information: one which fixed the amount of evaluation for an expression when the closure was built; and one which chose this when beginning the evaluation of an expression.

Since the evaluation model changes the order in which expressions are evaluated, it was not sufficient to measure only how fast programs ran and how much heap they consumed, but we also measured some dynamic characteristics of programs: their maximum heap requirement; cell life-times and heap profiles. The dynamic measurements give insight into the load placed on the storage management system.

We describe our experimental methodology in more detail in Section 3. The experimental results and their analysis are presented in Section 4. We compare our work with that of Hartel in Section 5. The experiments have given a number of useful insights into using the evaluation model.

¹There is some confusion in the community about the meaning of 'strictness'. A function $f : \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$ is strict in its i th argument if $\forall 1 \leq j \leq n. \forall s_j. f s_1 \dots s_{j-1} \perp s_{j+1} \dots s_n = \perp$, for both first- and higher-order functions. The practical consequence of a function being strict in its i th argument is that it is safe to evaluate that argument *to (weak) head normal form* before (or in parallel with) evaluating the application. For structured data types such as tuples or pairs, a program analysis may find out more information about how an expression is used. This is *not* strictness, and does not even necessarily imply strictness. For example, the structure of the second argument to `append` can be evaluated if the structure of the result of applying `append` needs to be evaluated, but `append` is not strict in its second argument. When more information can be determined about data structures, the natural operational concepts are a *evaluators* and *evaluation transformers*, discussed in Section 2.1. Note that it is not sufficient to try to distinguish between these two cases by using terms such as 'flat' and 'non-flat' strictness, because the definition of strictness includes higher-order functions, which form a non-flat domain.

Directions for further work, which arise from the analysis of the results, are discussed in the final section.

2 Background

2.1 The Evaluation Transformer Model of Reduction

Evaluators

Expressions of type integer and boolean can only be evaluated as far as weak head normal form (WHNF), and functions are usually only evaluated as far as WHNF. Expressions of constructed type, pairs and lists for example, can have more complicated patterns of evaluation. Lists were the only constructed type in the language used for our experiments, and we only considered: evaluation to WHNF, denoted by ξ_{WHNF} ; evaluation of the structure of a list, denoted by ξ_{TS} (TS \equiv “tail-strict”); and evaluating the structure of the top-level list, and each element of list to WHNF, denoted by ξ_{HTS} (HTS \equiv “head- and tail-strict”). We use the term *evaluator* to refer to such a pattern of evaluation.

Evaluation Transformers

The amount of evaluation required of an argument expression in an application depends on the amount of evaluation required of the application. For example, if an application of `reverse` has to be evaluated with ξ_{WHNF} , then its argument can be evaluated with ξ_{TS} (since the whole of the structure of the list has to be traversed before the first `Cons`-cell can be returned), whilst the argument can be evaluated with ξ_{HTS} if the result of the application is being evaluated with ξ_{HTS} . An *evaluation transformer* for an argument to a function is a mapping from the evaluator being used to evaluate an application of the function to the evaluator which can be used to evaluate that argument of the function.

Determining Evaluation Transformers

Evaluation transformers can be determined using a safe program analysis phase in a compiler [Bur90, Bur91]. An analysis is safe if the set of expressions it says can be evaluated is a subset of those which are evaluated when the program is executed lazily.

Generating Different Versions of Code for Functions

A *version* of the code for a function has to be generated for every evaluator that could be used to evaluate an application of the function. What evaluators can be used to evaluate a function application? As long as the result type of a function is not a type variable, its type can be written as $(\sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau)$ where τ is not a function type. If such a function is applied to less than n arguments, then the result is of function type, and so the application can only be evaluated with ξ_{WHNF} . If it is applied to n arguments, then the application can be evaluated with any evaluator which can be used to evaluate an expression of type τ (ξ_{WHNF} for base types; and ξ_{WHNF} , ξ_{TS} , or ξ_{HTS} for lists). Any type will always have the evaluator ξ_{WHNF} associated with it, so no extra code versions need to be generated to deal with partial applications. If the result type of a function is a type variable, then a version of the code for the function needs to be generated for every possible evaluator that can be used for every possible instantiation of that type variable in the program (and this can be statically determined [Hol83]).

Choosing the Version of a Function

Each expression is evaluated with a particular evaluator. Suppose we are to compile the ξ version of `f` defined by:

```
> f x1 ... xm = g D1 ... Dn
```

then there are two ways that we use the evaluation information at compile time:

- the ξ version of the code for `g` is chosen;
- for each argument D_i to `g`, the evaluation transformer for that argument is applied to ξ to determine how much evaluation can be done to the expression D_i . If some evaluation is allowed, and $D_i = h P_1 \dots P_k$, then this version of `h` is chosen and the expression evaluated. Otherwise, a closure has to be built for D_i .

In the case that a closure must be built, which version of the code should be chosen for the function being applied? If the expression is ever evaluated, we can only guarantee that it will be evaluated with ξ_{WHNF} . Thus it is safe to build the closure so that the ξ_{WHNF} version of the function is chosen when the evaluation of the expression begins; the version is fixed *when the closure is built*. A second option is to associate an evaluator with each closure. This can be updated if it is subsequently discovered that more evaluation is required of the expression², and the version of the code is finally fixed *when evaluation of the expression begins*. In this case the compiler generates code to update the evaluator associated with a closure, so there is some run-time overhead in supporting this, but no run-time analysis. Details can be found in [Bur91, Chapter 6]. We call these two options respectively *compile-time* and *run-time* choice of versions, because the compiler generates code which fixes the version in the first case and allows the version to be changed at run-time in the second. Note that run-time choice of versions allows at least as much eager evaluation as compile-time choice of versions.

2.2 The Spineless G-machine

Code was produced for a version of the Spineless G-machine which uses variable-sized cells rather than binary application nodes to store closures. The machine is basically that presented in [Bur91, Chapter 6], modified to include a value stack for intermediate values during the computation of arithmetic expressions.

The Expected Benefits of Using Evaluation Transformer Information

There are two expected benefits of eagerly evaluating an expression on the Spineless G-machine:

- a saving in time because a closure never needs to be built for the unevaluated expression; the closure does not have to be read before starting the evaluation; and the closure does not need updating; and
- a saving in heap space because the closure does not need to be built.

Boxed versus Unboxed Values

When arguments have been evaluated, it is possible to pass the arguments ‘unboxed’ (on the stack or in registers) rather than ‘boxed’ (in an (evaluated) closure in the heap). Passing unboxed values from flat domains, such as integers and booleans, is known to be worthwhile. Using evaluation transformer information allows at least as many opportunities for passing such values unboxed as using only strictness

²This is possible because any static program analysis must lose information.

information, but we have not tried to quantify this in our experiments. Using evaluators stronger than ξ_{WHNF} opens up the possibility of passing more complicated data structures ‘unboxed’. This was not done in our experiments, but is discussed further in the final section.

We have not explored any other optimisations which are possible when it is known that an expression has been evaluated.

Supporting Evaluation Transformers

No modification needs to be made to the abstract machine in order to support compile-time choice of versions; only the compiler needs to be changed, as detailed in [Bur91, Chapter 6].

The advantage of using run-time choice of versions is that an it might be possible to evaluate an expression with a stronger evaluator than was known to be safe at compile-time. This means that there must be a mechanism to change the version of the code for a function that was selected when a closure was built. An expression of list type may have already been partially evaluated, and so the mechanism must first traverse the evaluated part of the list in order that the code version in the closure that forms the tail at some point, if there is one, can be changed³. So that the traversal only happens when more evaluation has been requested than all previous requests, each Cons node must have an evaluator associated with it, which is modified as the list is traversed to find the unevaluated tail.

Associating evaluators with nodes, and testing and updating them, can be done very efficiently. Rather than having a tag field and an evaluator field, these are combined into one field. We adapted this idea from [KLB91]. In our implementation tag-evaluators are addresses of jump tables (c.f. [Joh87, Chapter D]), and so we have a tag-evaluator table for every combination of tag and possible evaluator. Clearly there will only be one table for data types such as integers which have only one evaluator, or for values like Nil which cannot be evaluated further. As well as the usual entries for instructions like EVAL, there is an entry in each jump table for each possible update of an evaluator that might be requested. No comparison of evaluators needs to be done at run-time because attempts to update with an evaluator which is not stronger than the one already there just return.

3 Experimental Methodology

The Language

All programs were written in a simple functional language which has lets; whose base types are integers, booleans

³More complicated evaluators, such as one which evaluates the elements of the list with ξ_{TS} or ξ_{HTS} , also have to update the code versions selected in the closures for the elements of the list. Again this might require the traversal of a partially evaluated data structure. A little thought shows how the idea can be generalised to arbitrarily nested data structures.

It was suggested in [Bur91, Chapter 6] that the expression in the tail should be evaluated straight away, and this was implemented in our experiments. However, in a sequential machine, there are two good reasons for delaying the evaluation until some part of the tail is actually inspected. They are:

- it delays having the extra structure in the heap, and this is an advantage if the closure in the tail is small; and
- it may be discovered that the expression will need even more evaluation, saving the cost of building more closures.

The situation may of course be different in a parallel implementation.

and characters; and which has lists as its only compound data structure.

The Benchmark Suite

Both the available time and tools limited our experiments to the investigation of nine programs of small to medium size. The programs, their source, and their number of lines (excluding blank lines and comments, but including the input data) were:

calendar: prints the calendar for a given year (based on an example in [BW88]; 132 lines);

fft: fast Fourier transformation (from Hartel’s suite; 188 lines);

hanoi: the towers of hanoi problem (13 lines);

logsim: a clock driven simulator simulating a 4-bit adder, which makes extensive use of higher-order functions (due to David Bevan; 100 lines);

maxflow: Dinic’s algorithm to solve the problem of assigning maximal flow for a network (due to Paul Kelly; 100 lines);

qsort: quicksort on a data set of 100 random integers (33 lines).

scc: computes the strongly connected components of a given graph, in a ‘functional’ formulation (from John Launchbury; 53 lines);

wang: performs Gaussian elimination, by representing matrices as a list of lists (from Hartel’s suite; 146 lines); and

wave: predicts the tide in a rectangular estuary of the North Sea during a time interval (from Hartel’s suite; 189 lines);

Determining Evaluation Transformers

We used Hunt’s abstract interpretation system to determine the evaluation transformers for our programs [Hun89]. To use it, programs had to be translated into Hope, and then the output from the analyser was used to annotate programs with evaluation transformers by hand.

Generating Code

Four code *variants* were produced for each program. In order of increasing eagerness they were:

lazy: the normal lazy code;

ct.s: uses only strictness information (for first- and higher-order functions)⁴;

ct.et: uses compile-time choice of versions; and

rt.et: chooses versions at run-time.

⁴Note that using strictness information is just a degenerate case of using the evaluation transformer model of reduction where there is only one evaluator, ξ_{WHNF} . Also note that only compile-time choice of versions makes sense in this case because closures will be built selecting the ξ_{WHNF} version of the function being applied, and this cannot be updated with anything stronger because there is no stronger evaluator.

Both the `ct_et` and `rt_et` variants of the code evaluated the top-level expression with ξ_{HTS} if it was of list type.

The Spineless G-code was macro-expanded into C, instrumented to collect statistics, and then placed in a harness to form a complete C program.

Accounting for Garbage Collection

It was unclear how to take garbage collection overhead into account, especially when the variants of code have different space behaviours. The problem is that measurements are susceptible to edge effects. For example: if the number of garbage collections are small, then one more garbage collection may unfairly bias the results; and the maximum graph size of one variant of the code may be close to an arbitrary setting of the heap size, so that a large number of garbage collections occur, with a high heap occupancy level, for an insignificant part of the program. We therefore chose to separate the cost of doing the real work of the program from the cost of garbage collection.

Measuring Execution Time

We measured execution time by counting memory references (to the value and pointer stacks and the heap), as originally suggested by Hartel, because there is a strong correlation between this and the real execution time of lazy functional programs [Har91b]⁵.

Dynamic Measurements

Clearly the use of evaluation transformer information is going to change the space behaviour of programs because of the early evaluation of list structures by ξ_{TS} and ξ_{HTS} . In the absence of a good methodology for taking garbage collection into account, we opted to measure the maximum graph size for a program; its heap profile over time; and cell lifetimes. Although the code we generated for evaluating a list using ξ_{TS} built the list on the stack, flushing it into the heap when the end of the list has been found, we did not measure stack high-water marks. The reason is that a more sophisticated code generator could build `Cons` nodes with empty tails in the heap as each tail is evaluated to a `Cons` node, updating the previous empty tail with a pointer to the new `Cons` cell. Lists could then be evaluated in constant stack space.

4 Experimental Results and Their Analysis

4.1 Total Memory Accesses and Total Heap Allocation

The results for our nine programs can be found in Table 1. Each program is given two rows in the table: the first is the actual value of what was being measured; and the second is its ratio with respect to the figure for the lazy code.

In the observations that follow, recall that we use total memory accesses to measure time, so that “faster”, “uses less time”, and the suchlike are meant to be synonymous with “had fewer memory access”.

• **In terms of the number of heap cells allocated, lazy is worse than `ct_s`, `ct_s` is worse than `ct_et`, and `ct_et` is worse than `rt_et`.** This is to be expected because the information we are using only allows the early evaluation of expressions

⁵If we were to use unboxed objects and pass them in registers, then only measuring stack and heap references would not be a valid representation of the time, `nfib` would run in almost zero time using this metric for example, but we do not have this feature in our experiments, and so the measurement of time is valid.

which are eventually evaluated by the lazy code. In the Spineless G-machine, early evaluation of expressions saves the cost of building closures⁶.

• **The fractional decrease in the number of heap-allocated cells is always greater than the fractional decrease in total memory accesses.**

• **Using strictness information always improved the performance of a program.** For `wang`, `calendar` and `hanoi` this improvement was negligible, but it almost halved the time taken for `wave`.

• **Using evaluation transformer information improved the performance of only four programs: `calendar`; `logsim`; `hanoi`; and `qsort`.** For programs where there was little improvement, a study of the code showed that most functions in such programs have evaluation transformers which map any evaluator to ξ_{WHNF} , and that these are on the critical computation path of the program, so little improvement is expected from using evaluation transformers. Furthermore, most of the functions in the programs which had more interesting evaluation transformers for their arguments were combinators that one would expect in a library, `map`, `foldr`, and `reverse` for example. If this is the case in general, then perhaps the program analysis phase of a compiler could be made very fast, particularly if programmers used the style of programming advocated in [BW88].

Of the four programs which did give an improvement, only two of them are realistic programs, or represent realistic programming paradigms (`calendar` and `logsim`), and even one of them (`logsim`) only had a minor improvement. Figure 1 shows that the `ct_et` variant of `hanoi` ran so much faster because no closures were created for function applications; the heap consisted entirely of `Integer`, `Char`, `Cons` and `Nil` nodes.

• **Using more complicated evaluators than ξ_{TS} and ξ_{HTS} can improve the performance of programs.** The `wang` program does a lot of matrix manipulation, where the matrices are represented as lists of lists. The elements of the new matrices are in general simple arithmetic expressions of some elements from old ones, which are selected by using `head` and `tail`. Further experiment has shown that no real speed-up is gained for this program unless these expressions get evaluated instead of building closures for them. This corresponds to using a new evaluator which evaluates the elements of a list with ξ_{HTS} . When this was done, we found that the total number of memory accesses was halved and the number of heap cells allocated was cut to one third. We draw two conclusions from these observations: (i) representing matrices as lists of lists can be highly inefficient; and (ii) it is worth investigating more complicated patterns of evaluation for nested data structures.

⁶The situation is a bit more complicated because, if an expression is not shared, and not evaluated until it is needed, then its value never needs to be written into the heap. So for a strict, unshared argument, we have two options: evaluate the expression early and store its value in the heap, saving the cost of building the closure; or build the closure and evaluate it when needed, saving the cost of building the data object in the heap. Since our compiler assumes that all argument expressions are shared, it does not have this choice, so that selecting the first option is always a win.

	Total Number of Memory Accesses				Number of Heap Cells Allocated			
	lazy	ct_s	ct_et	rt_et	lazy	ct_s	ct_et	rt_et
calendar	2 500 931 1.00	2 431 205 0.97	2 380 877 0.95	1 719 567 0.69	579 449 1.00	555 181 0.96	541 181 0.93	265 994 0.46
fft	76 288 108 1.00	43 283 670 0.57	43 283 670 0.57	43 331 812 0.57	16 722 098 1.00	7 505 613 0.45	7 505 613 0.45	7 505 613 0.45
hanoi	4 867 726 1.00	4 836 013 0.99	1 774 338 0.36	2 470 684 0.51	1 203 125 1.00	1 192 897 0.99	274 389 0.23	274 389 0.23
logsim	7 245 801 1.00	6 566 344 0.91	6 390 059 0.88	6 351 726 0.88	1 805 050 1.00	1 592 052 0.88	1 539 677 0.85	1 513 181 0.84
maxflow	415 199 1.00	307 446 0.74	307 446 0.74	307 254 0.74	112 176 1.00	77 142 0.69	77 142 0.69	77 002 0.69
qsort	230 892 1.00	208 942 0.90	165 633 0.72	173 783 0.75	49 615 1.00	43 004 0.87	30 143 0.61	25 737 0.52
scc	43 937 1.00	38 022 0.87	38 022 0.87	38 024 0.87	10 327 1.00	8 612 0.83	8 612 0.83	8 599 0.83
wang	138 940 1.00	137 162 0.99	136 837 0.98	137 250 0.99	33 746 1.00	33 015 0.98	32 923 0.98	32 879 0.97
wave	14 726 839 1.00	8 067 866 0.55	8 067 866 0.55	8 068 176 0.55	3 353 612 1.00	1 489 006 0.44	1 489 006 0.44	1 489 006 0.44

Table 1: Memory Accesses and Heap Allocations

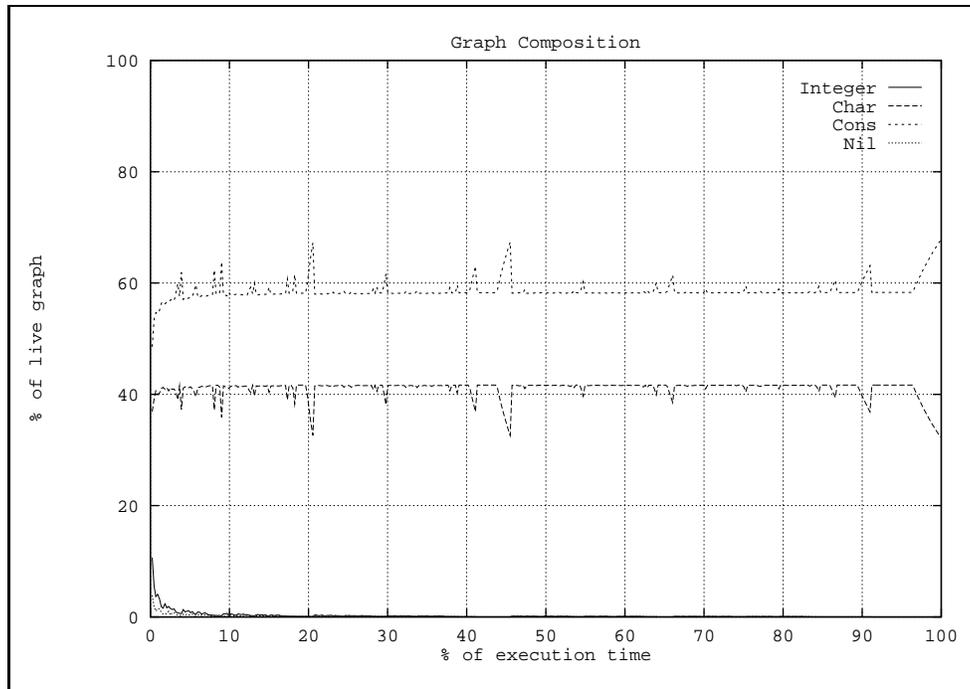


Figure 1: Graph Composition for both the ct_et and rt_et variants of hanoi

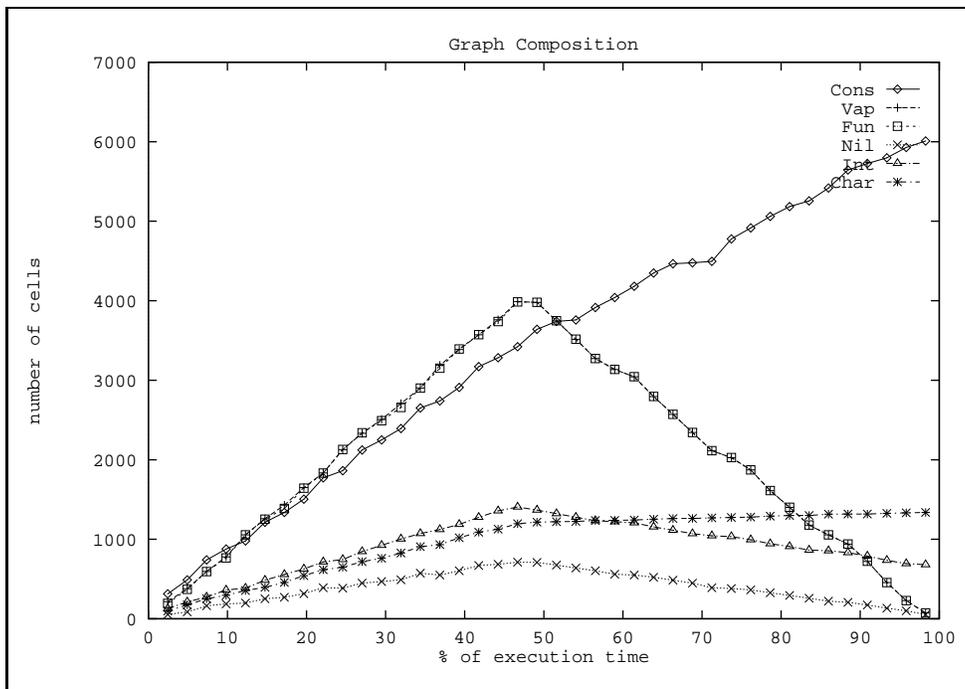


Figure 2: Graph Composition for the `rt_et` variant of `calendar`

• **Only one program, `calendar`, ran faster using `rt_et` instead of `ct_et`.** A plot of the composition of the graph over the life of the program for the `ct_et` variant of `calendar` gives a fairly flat profile for each of the node types in the heap. In contrast, the heap profile for the `rt_et` variant, given in Figure 2, shows that the number of closures (`Vap` and `Fun` nodes) decreases⁷, and the number of `Cons` nodes continues to increase from about half-way through executing the program. We suspect that this happens for the following reason. The `calendar` program is defined as a pipeline:

```
> calendar = display . block 3
>           . map picture . months
```

The evaluation transformer information tells us that the argument to `display` can be evaluated using ξ_{HTS} . A problem arises with the function `block`, which uses `zip`, defined by:

```
> zip :: ([*], [**]) -> [(*,**)]
> zip ([],ys)       = []
> zip (xs,[])       = []
> zip ((x:xs),(y:ys)) = (x,y):zip(xs,ys)
```

which takes a pair of lists and returns a list of pairs *which is as long as the shorter of its two arguments*. Because one argument list may be shorter than the other, and we do not know which, neither argument to `zip` can be evaluated with ξ_{TS} , even though the application of `zip` can be evaluated with ξ_{HTS} . We conjecture that once the `zip` operation has been completed, the run-time information is able to find

⁷Note that the numbers of `Vap` and `Fun` nodes are almost identical, so that their curves cannot be distinguished in Figure 2. Note also that we have only plotted one in ten of the sampled points during the execution of the program. Plotting more cluttered the graph without adding any extra information.

out that the result must be fully evaluated, and so is able to continue without creating any more closures.

Lazy data structures and general, reusable functions like `zip` are two things that make lazy functional programming nice [Hug89]. However, the problem raised in the previous paragraph leads us to ask whether we should be forced to use lazy data structures all the time, and whether it is always a good idea to use functions as general as `zip`, when there is a big cost in terms of a less efficient implementation. Perhaps data types like finite lists should be provided in a language, and libraries should contain some specialised versions of functions; a version of `zip` could be provided which only works if both of its arguments are the same length, with an obligation on the programmer to prove that it is safe to use it.

• **Two programs, `hanoi` and `qsrt`, ran slower using run-time instead of compile-time choice of versions.** The reason why `hanoi` ran slower using run-time choice of versions can be seen from Figure 1: no closures were created in the `ct_et` code, and so any operations to support the run-time choice of versions were redundant. The compiler we used made no attempt to detect redundant operations. Instead we counted them, giving an upper bound on what could be saved by a clever compiler. For `hanoi` redundant updates of the evaluator field and extra calls to evaluate an already evaluated expression account for about seventy percent of the extra memory references. For `qsrt`, there are approximately 17000 extra memory references due to these operations, indicating that the `rt_et` variant of `qsrt` could be more efficient than the `ct_et` variant if a compiler could spot most of the redundant operations.

	Maximum Graph Size				Ratio			
	lazy	ct_s	ct_et	rt_et	lazy	ct_s	ct_et	rt_et
calendar	13 649	13 521	13 638	55 864	1.00	0.99	1.00	4.09
fft	706 834	745 470	743 245	745 755	1.00	1.05	1.05	1.06
hanoi	829	666	58 308	58 308	1.00	0.80	70.34	70.34
logsim	4 345	4 291	346 534	347 375	1.00	0.99	79.75	79.95
maxflow	806	923	923	973	1.00	1.15	1.15	1.21
qsort	2 593	2 593	4 777	4 745	1.00	1.00	1.84	1.83
scc	1 187	1 159	1 159	1 159	1.00	0.98	0.98	0.98
wang	6116	6139	14563	14563	1.00	1.00	2.38	2.38
wave	17 135	17 025	16 986	17 012	1.00	0.82	0.99	0.99

Table 2: Maximum Graph Sizes for the Programs

4.2 Dynamic Measurements

- **The use of strictness information had little effect on the maximum graph size.** The maximum graph sizes varied from 0.8 to 1.15 times the maximum size for the lazy case.

- **There is no correlation between the increase in speed of a program and its maximum graph size.** For example, the `ct_s` variant of `wave` ran almost twice as fast as the lazy variant, and had a decrease in its maximum heap size for the `ct_s` variant, but `fft`, which had almost the same increase in speed for its `ct_s` variant, increased its maximum heap size in this case. Similarly the `rt_et` variant of `calendar` ran 1.37 times as fast as the `ct_et` variant for a four-fold increase in the maximum heap size, whereas a 1.13 times improvement in speed between the `ct_s` and `ct_et` variants of `logsim` resulted in an eighty-fold increase in maximum heap size⁸.

- **For the programs in our test suite, if the `ct_et` or the `rt_et` variant of a program runs faster than the `ct_s` variant, then it produces a bigger maximum heap size than that produced by the lazy variant.** This happens for: `calendar` (`rt_et` variant); and `logsim`, `hanoi`, and `qsort` (`ct_et` and `rt_et` variants).

- **The increase in maximum heap size is dramatic in some cases.** For example, there was an eighty-fold increase for the `ct_et` variant of `logsim`. So far we have no way of predicting when this will happen.

- **If a variant of the code increases the efficiency of the program, then the cell-lifetime profile of the program changes.** This can be seen from the example graph in Figure 3, where the curve for the `rt_et` variant is markedly different from that for the other variants. Further experiment is needed in order to characterise exactly how it changes.

- **The amount of live heap, measured as a percentage of the maximum graph size, sometimes changes between the different variants of the code.** For example, in Figure 4 we compare the heap profile for the `ct_et` version of `calendar` (little increase in speed over just using strictness information) with that for `rt_et` (large increase in speed over just using strictness information). It is important to note that

⁸Recall that we do not include the memory accesses due to garbage collection in our experiments.

neither the horizontal or vertical scales on the two graphs are the same. From looking at other heap profiles, this does not always happen, so some further investigation needs to be done here.

5 Comparison with the Work of Hartel

Hartel has made a comprehensive study of a number of different optimisations (strictness, tail-strictness, unboxing, inlining, and evaluating arithmetic expressions even if they may not be needed) in the compiler being developed at Southampton [Har91a]. The computational model adopted by Hartel can be described as compile-time choice of versions with a restricted number of evaluators (only ξ_{WHNF} and ξ_{TS}). Time was measured by counting the number of cell allocations. Whilst we have not considered such a range of optimisations, our work extends this in the following ways:

- we have considered more complicated patterns of evaluation than tail-strictness;
- we have investigated run-time choice of versions; and
- our experiments show that Hartel’s choice to count only cell allocations as a measure of time may not be sufficiently accurate, especially when run-time choice of versions is allowed (c.f. the `ct_et` and `rt_et` variants of `hanoi` and `qsort`, where the number of cells stayed the same, or decreased, but there was an increase in the number of memory accesses, and hence an increase in execution time)⁹.

6 Conclusion and Further Work

Our experimental results for the *Spineless G-machine* can be summarised as follows:

- using strictness information always gives better performance, but maybe not much better;
- using evaluation transformer information sometimes gives good speed-ups, but sometimes at the cost of a dramatically increased maximum heap size;
- only one of the programs from the test suite had significant gains in using run-time choice of versions; and

⁹Note that in his original experiments reported in [Har91b], Hartel counted both the total memory accesses and the cell allocations. We also did this in our experiments.

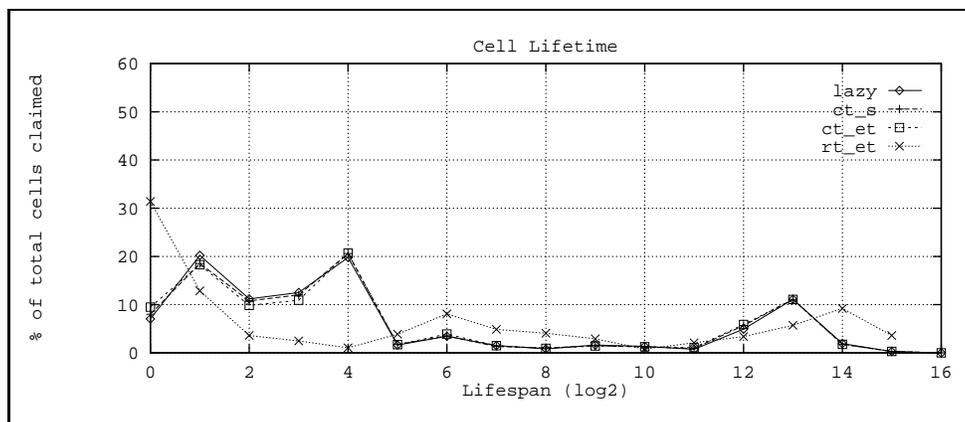


Figure 3: Cell Lifetimes for calendar

- there seems to be no correlation between performance gains and the maximum graph size.

We believe that the most important contribution of this paper is to clarify the issues which need further investigation. Some of these are:

- Clearly not all programs gain from using evaluation transformer information. We would like to try to characterise the style of programs for which there is a performance gain. Furthermore, we would like to try to determine the sort of programs which benefit from runtime choice of versions. If such programs could be determined statically, then we could develop a set of heuristics to be used by the compiler.
- We are unsatisfied at not being able to take garbage collection overhead into account in a satisfactory way, and plan to develop an experimental methodology to rectify this situation.
- None of the experiments tried to measure the changes in performance as a function of the program input size. In the absence of theory about complexity of programs written in lazy languages, it is important to try to do this experimentally.
- Passing objects unboxed is important for integers, and what notions of ‘unboxedness’ are useful for objects such as lists is still an open question. Hall has shown that using n -ary rather than binary cons cells can make programs more efficient [Hal92]. This may be a sensible alternative to attempting to store an unboxed list as some sort of an array, as has been tried by Hartel [Har91a].
- Our results are from running the programs on the Spineless G-machine. It is not clear how the machine model underlying a compiler affects the results of the experiments. For example, if a compiler is very good at implementing strict functions but bad at lazy ones, then the savings due to using strictness information could be more dramatic.
- Our observations about `wang` suggest that more complicated patterns of evaluation will be useful for some

programs using nested data structures, and this needs investigation in more detail.

- Finally, no account of parallelism was taken in these experiments. The original motivation for evaluators like ξ_{HTS} was that they opened up opportunities for parallel evaluation. Some preliminary experiments in this area have been reported in [KLB91, HKL92]. However, the first can only be regarded as an existence proof, and an indication that the evaluation transformer model of reduction may have a pay-off in a parallel implementation; and the second suffers from the problem that the code is interpreted, and so the real cost of communication is not taken into account.

Howe is currently working on some of these questions for real programs compiled with the Glasgow Haskell compiler. Some initial results can be found in [HB92].

Acknowledgements

Paul Kelly provided very useful input at various stages of the project, in particular suggesting the adoption of Pieter Hartel’s methodology for conducting these experiments. Sebastian Hunt made his abstract interpretation system available to us, and helped us interpret the results, and Pieter Hartel kindly sent us the source-code of the benchmarks that he has often used in his experiments. Denis Howe has provided much help in running the experiments and commenting on drafts of this paper. Andrew Bennet, Pieter Hartel, Martin Kohler, and Daniel Le Métayer have also made helpful comments on drafts of this paper.

References

- [BPR88] G.L. Burn, S.L. Peyton Jones, and J.D. Robson. The spineless G-machine. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 244–258, Snowbird, Utah, 25–27 July 1988.
- [Bur87] G. L. Burn. Evaluation transformers – A model for the parallel evaluation of functional languages (extended abstract). In G. Kahn, editor, *Proceedings*

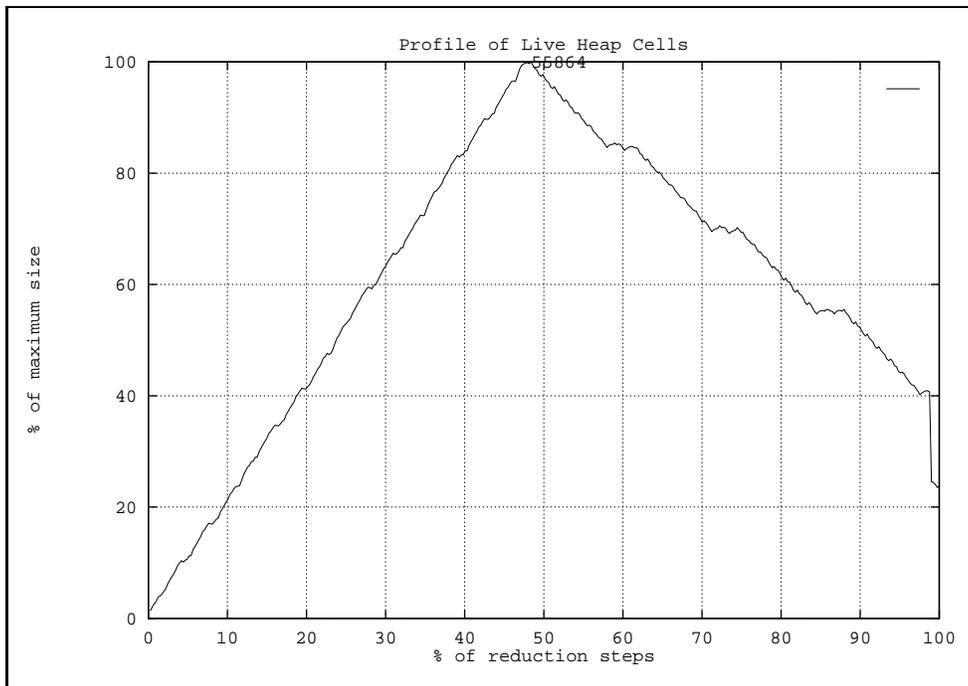
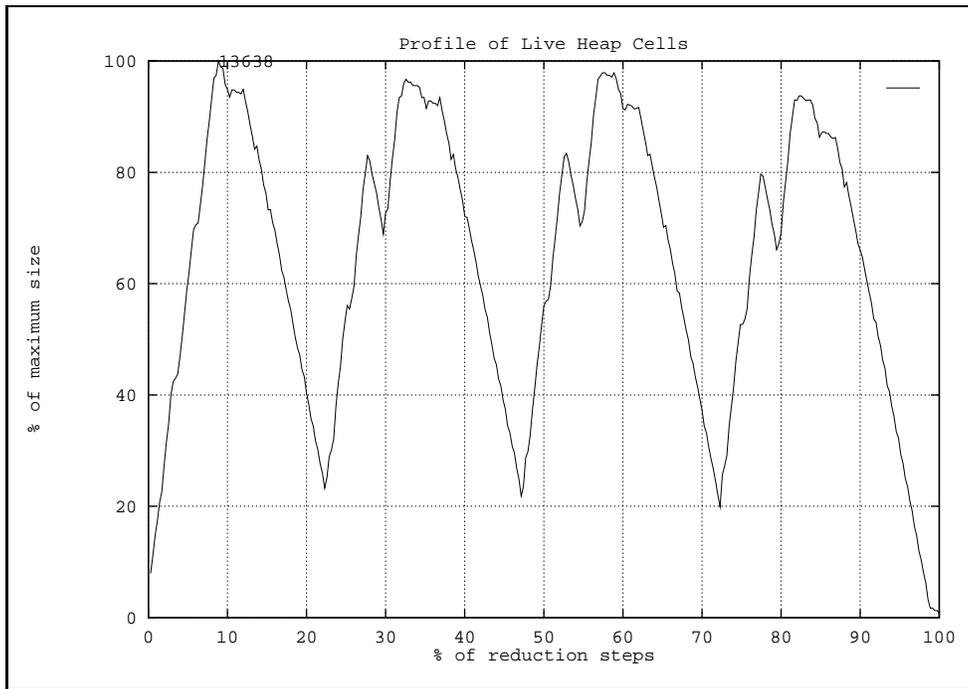


Figure 4: Comparing the Graph Growth for the ct_et and rt_et Versions of calendar

of the *Functional Programming Languages and Computer Architecture Conference*, pages 446–470. Springer-Verlag LNCS 274, September 1987.

- [Bur90] G.L. Burn. Using projection analysis in compiling lazy functional programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 227–241, Nice, France, 27–29 June 1990.
- [Bur91] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman in association with MIT Press, 1991. 238pp.
- [BW88] R. Bird and P.L. Wadler. *An Introduction to Functional Programming*. Prentice-Hall Series in Computer Science. Prentice-Hall International (UK) Ltd., Hemel Hempstead, Hertfordshire, England, 1988.
- [Hal92] C.V. Hall. An optimist’s view of life: Transforming list expressions. Technical report, Department of Computer Science, University of Glasgow, 1992.
- [Har91a] P.H. Hartel. On the benefits of different analyses in the compilation of lazy functional languages. In *3rd Informal International Workshop on the Parallel Implementation of Functional Languages*, pages 123–145, Southampton, 1991.
- [Har91b] P.H. Hartel. Performance of lazy combinator graph reduction. *Software — Practice and Experience*, 21(3):299–329, March 1991.
- [HB92] D.B. Howe and G.L. Burn. Experiments with strict STG-code. In *4th Informal International Workshop on the Parallel Implementation of Functional Languages*, Aachen, 28–30 September 1992.
- [HKL92] G. Hogen, A. Kindler, and R. Loogen. Automatic parallelisation of lazy functional programs. In *Proceedings of ESOP92*, pages 254–268. Springer-Verlag LNCS582, 1992.
- [Hol83] S. Holmström. *Polymorphic Type Systems and Concurrent Computation in Functional Languages*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1983.
- [Hug89] R.J.M Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.
- [Hun89] S. Hunt. Frontiers and open sets in abstract interpretation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, 11–13 September 1989.
- [Joh87] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.
- [KLB91] H. Kingdon, D.R. Lester, and G.L. Burn. A transputer-based HDG-machine. *The Computer Journal*, 34(4):290–301, August 1991.