# Heap Profiling of a Lazy Functional Compiler

Colin Runciman and David Wakeling University of York\*

# 1 Introduction

A significant problem with lazy functional programs is that they often demand a great deal of space. Multi-megabyte workstations are now commonplace, but serious users of functional programming systems have to equip even these machines with additional memory. The essence of laziness is to delay evaluation rather than compute values that may not be needed; and once values *are* computed to retain them if they may be needed again. This policy might save time, but it can easily lead to *space faults*: the accumulation or retention of large structures in memory, in ways that the programmer is unaware of, or does not fully understand, let alone intend. Hence *profiling tools*, by which programmers can obtain information about memory use in terms of the source program, are potentially of great value.

In a previous paper [5] we described the design, implementation and use of a prototype heap profiling tool for lazy functional programs. As an example application, we used clausify, an existing 130 line program to normalise logical formulae. The results were even better than we had hoped for: after five iterations of profiling and refinement, modifying both program and compilation rules, the cost of running the final version (measured in bytes  $\times$  seconds) was less than 0.5% of the cost of running the original. However, the information from our prototype tool related to individual definitions and constructor functions, and it was not clear whether a similar technique would work for larger and more complex programs.

This paper reports on the experimental application of heap profiling to Augustsson and Johnsson's Lazy ML (LML) compiler [1]. Since our implementation of heap profiling is itself based on the LML compiler, this amounts to a boot-strapping exercise. The LML compiler extends to some 16,500 lines of code in almost 200 modules, and is by any standard a large and sophisticated piece of software. As it has been developed over a period of almost a decade, with each successive version outperforming its predecessor, the sort of dramatic improvement obtained for the clausify program is hardly to be expected; but

<sup>\*</sup>Authors' address: Department of Computer Science, University of York, Heslington, York Y01 5DD, United Kingdom. Electronic mail: colin@minster.york.ac.uk, dw@minster.york.ac.uk

equally savings of only a few percent would not be very compelling. So we set ourselves the target of *a factor of two* — halving the cost of LML compilations.

The rest of this paper is organised as follows. Section 2 reviews the design of our prototype heap profiling tool, and describes how we have modified it to deal with larger programs. Section 3 outlines the structure of the LML compiler, identifying its main components and their role within the compiler. Section 4 describes the application of heap-profiling to the compiler, and the successive reduction of its memory consumption guided by profiling information. Section 5 discusses to what extent the kinds of space problems found in the compiler might be avoided in future programs by modifying implementation methods. Section 6 briefly considers some related and future work. Finally, Section 7 offers some conclusions.

# 2 A Heap Profiling System

Our heap profiling tool has two components. The first component is a modified implementation which generates profiling information during the execution of functional programs. When the programmer requests a heap profile, execution is suspended at specified regular intervals and the implementation traverses the program graph gathering information from each cell. This information is appended to a log file and execution is resumed. When execution is complete, the log file contains a profile of the graph nodes that were stored in the heap at each interval.

The second component of the tool is a program that generates a graph from a log file. Examples of these profile graphs will be found in Section 4. A profile graph shows how the amount of heap storage used by the program (measured in bytes) varies over the time that it takes to run (measured in seconds). Shaded bands are used to show how much of the total storage is associated with each identifier.

Recently, we have made our profiling tool more suitable for dealing with large programs. The modified implementation now attaches both *static* and *dynamic* tags to every cell in the heap. Static tags carry information determined at compile-time and dynamic tags carry information determined at runtime. For the static tags, space is reserved in each cell for a pointer to some tag information maintained by the compiler. For each dynamic tag, space is reserved for some tag information maintained by the run-time system. By way of example, Figure 1 shows how a list node is tagged.

In our first implementation there were only two static tags. These identified the function that produced the graph node, and the construction that it represented. When profiling larger programs, it is natural to want to extend this basic scheme. Thus, the producer is extended to a single module or a group of modules, and the construction is extended to a type. Each cell has space for one dynamic tag; in future, we plan to use this tag for the age of the node, but at the moment it is unused.

Another improvement to the basic scheme reduces the overhead of profiling



Figure 1: A tagged list node

by performing a garbage collection while sampling the graph.

Usually, profile graphs are automatically scaled to fill the page, and the bands are automatically shaded and ordered so as to maximise the readability of the graph. However, for the purpose of comparison it is sometimes useful to fix certain parameters across several profile graphs. In this paper, all profile graphs share a common scale, shading and ordering.

### 3 An Overview of the LML Compiler

The LML compiler [1] consists of two programs which communicate via a text file. The *parser* is written in C with the aid of the Yacc parser generator. It checks the program syntax and outputs a prefix form of the parse tree. The *translator* is written in LML. It reads the prefix form of the parse tree and reconstructs it. After a number of transformations, it then produces the assembly language for the program. We are only concerned with profiling the translator, and from here on we will loosely refer to it as "the compiler".

The compiler source code is organised into 19 directories containing 198 modules; in total there are about 16,500 lines of code. In what follows, it will be useful to have an outline of the way that the compiler works. Below, a very brief description of each compiler pass is accompanied by the name of directory that contains the code for that pass.

- (1) The prefix form of the parse tree is read, and the tree is reconstructed (expr).
- (2) Conditional and ZF expressions are simplified (curry, zf).
- (3) Bound and imported identifiers are renamed if necessary (rename).

- (4) Pattern-matching is replaced by the use of case-expressions (transform).
- (5) Type checking is performed (type).
- (6) Expression simplifications, such as constant folding, are made (simpl).
- (7) Simple strictness analysis is performed (strict).
- (8) Nested functions are removed by lambda-lifting (llift).
- (9) G-machine code is generated and optimised (Gcode, Gopt).
- (10) M-machine code is generated and optimised (mcode, mopt).
- (11) Assembly code is generated (m\_68000).

Two other important components of the compiler, so far unaccounted for, are the functions in the standard library (lib) and the routines in the runtime system (runtime).

### 4 Profiling the Compiler

Before we can make any improvements to the LML compiler, we need to find out what makes it tick (or rather, clunk). The group profiles produced when the compiler recompiles each of its own source files are a good place to start. Leafing through them, a clear pattern emerges. Unfortunately, space precludes us from showing several of these profiles here. Instead we shall have to make do with a typical one, for the compilation of the 280 line module hcheck (see Figure 2). This profile shows that during a compilation taking a little over two and a half minutes, the memory demand exceeded two megabytes at its peak, and was above one and a half megabytes for most of the time. The "bytes x seconds" figure (top centre) corresponds to the total area under the graph, and is our overall measure of cost.

A key to the various shadings used is given on the right. The boxed labels 'A', 'B' and 'C' are not present in the output from the profiler; they have been added for ease of reference to three important regions of the graph. The region labelled 'A' represents graph structure produced by the M-code generator. The region labelled 'B' includes the three topmost bands; it represents graph structure produced by the expr and rename passes, and by some standard library functions lib. The region labelled 'C' represents graph structure produced by the run-time system. Together these three regions account for most of the compiler's demands on heap memory. We shall now discuss each of them in turn.

#### Version 0

Region 'A' indicates that the M-code generator produces a large amount of graph structure towards the end of the compilation. At first, this may not



Figure 2: A group profile for hcheck (compiler version 0)

seem particularly surprising: for any program written in a high level language, the machine code translation is bound to be quite large. Moreover, Johnsson's M-code generator is rather sophisticated, and so one might expect a significant amount of graph to be required to represent its internal data structures. However, this reasoning ignores the fact that the compiler is written in a *lazy* language. Since all of the clever optimisations described by Johnsson for translating G-code into M-code are *intra-functional*, and there are none which are *inter-functional*, we would actually expect G-code to be translated into M-code lazily, one function at a time. Clearly though, this is not happening. Indeed, it seems that the M-code for *every* function is generated before any assembly language is produced.

Although we could investigate further by producing a module profile restricted to the mcode group, it is unnecessary to do so. The profile tells us that the problem is some form of lazy pipeline blockage, and that information alone is enough for us to identify the cause. The blockage cannot be at either of the later stages in the pipeline: the M-code optimiser does little more than improve the aesthetics of the M-code, and the assembly language generator for the MC68000 is really just an elaborate pretty printer. By inspection, both are lazy, so the problem must be with the M-code generator itself. Somewhat embarrassingly, it turns out that it is one of our own modifications to the M-code generator that makes it use so much space.

Recall from Section 2 that our modified compiler attaches static and dynamic tags to every cell in the heap, and that the static tags carry information maintained by the compiler. In practice, this means that during code generation the compiler must place appropriate vectors of strings in the assembly language program, to be used at run-time when new nodes are created. This seems simple enough. However, there is a slight complication: for practical reasons it is important to ensure that neither the vectors or the strings are duplicated. Otherwise the resulting code becomes excessively large.

One way to do this is to record the vectors and strings arising during code generation in a table. When all of the M-code has been generated, any duplicates can be purged from the table and the remainder can be output along with the M-code. Another (very similar) way is to add new vectors and strings to the table only if they are not already there. This eliminates the need to purge the table before it is output. As far as coding is concerned, there is not much to choose between the two alternatives. As far as efficiency is concerned, however, the difference is rather large. The first alternative generates all of the M-code in order to create the table of vectors and strings. But none of this M-code can be output until any duplicates have been purged from the table and the correct labels for the remainder have been determined. The second alternative allows the correct labels for the vectors and strings to be determined when they are added to the table, and so the M-code can be output without delay. As Figure 3 shows, the second alternative costs 50Mbs less than the first.



Figure 3: A group profile for hcheck (compiler version 1)

#### Version 1

Let us now turn our attention to region 'B'. Figure 4 shows a profile by type, restricted to the module groups expr, rename and lib that make up this region. Here we can see that half of the graph structure is of either the List or Id types. A glance at the definition of the Id type sheds further light on the problem:

```
type Id = mkids String String
+ mkidi String String (Option (Modinfo # String))
+ mkid Int String String Idinfo Origname
```

Every identifier requires two strings: one is the name used by the profiler, and the other is the name used by the ordinary compiler. Usually, these names are identical; they differ only for identifiers that are renamed during compilation. Since strings are just lists of characters, Figure 4 suggests that identifiers could account for more than 90Mbs of the cost of our typical compilation. This seems rather high.



Figure 4: A type profile for the expr, rename and lib groups only

An obvious improvement would be to use a more compact string representation than the usual list of characters. Although the standard compiler affords compile-time string literals a compact representation, one cannot form a compact string at run-time. Let us rectify this by introducing a primitive function

```
pack :: [Char] -> [Char]
```

Semantically, this function is the identity for finite and fully defined lists of characters. Pragmatically, it converts an ordinary list of characters into a compact one. The idea is to use this function in the compiler to pack up the name strings whenever a new Id is constructed. Somewhat surprisingly, using **pack** as described makes no discernible difference to the behaviour of the compiler. This is because the standard evaluation machinery causes packed strings to be unpacked. By instrumenting the run-time system to print the context in which string unpacking was performed, we discovered four operations in the compiler that cause name strings to be unpacked:

- hashing;
- comparison;
- translation to an assembly language label;
- translation to an assembly language string.

To make these operations work directly on the packed representation, we rewrote them in M-code. As Figure 5 shows, packed strings reduce the cost of compiling our example file by another 50Mbs.



Figure 5: A group profile for the hcheck (compiler version 2)

#### Version 2

In all of the group profiles that we have seen so far, there has been a large block of graph structure produced by the run-time system (labelled 'C' in Figure 2).

With the aid of a producer profile restricted to the **runtime** group (see Figure 6), we can quickly establish that almost all of this structure is produced by the built-in routine which reads the list of characters in a file.



Figure 6: A producer profile restricted to the runtime group

Recall from Section 3 that the LML compiler consists of two programs, the *parser* and the *translator*, which communicate via a text file. During a compilation, the translator reads several files, but the only one of significant size is the text file created by the parser. From Figure 6 it seems that the contents of this file are retained by the translator as a list of characters for well over half the total compilation time. This is really most surprising. We would expect the translator to discard the character list in the process of constructing the abstract syntax tree. Clearly this does not happen. Yet the list is not dragged along until the very end of the compilation, so what triggers its release? In the "valley" of Figure 2 the critical computational event allowing source to be discarded is the onset of code generation. More specifically, it is the opening of the output stream for the assembly code.

Once we knew that the input stream was being retained, we made the above diagnosis by simply examining the code concerned with input and output. It looked something like this:

```
(finput, ftype,...fasm) =
case basename in
  No msg : fail msg
  || Yes n : (read (n@".p"), tofile (n@".t"),...tofile (n@".s"))
end
```

The intention here is to bind finput to the list of characters read from the input stream, and ftype and fasm to the streams to be used for the type and assembly language files. Unfortunately, this rather arcane piece of code suffers from a space leak caused by the implementation of lazy pattern matching. Delayed selection means that none of the components of (finput, ftype,...fasm) can be released until all of them have been evaluated. As a result, the list of characters read from the input stream is retained until the assembly language stream is required, which is not until well over half the total compilation time has elapsed. A fix for this particular problem is trivial: just make separate definitions of finput, ftype and fasm. The gain is yet another 50Mbs (see Figure 7).



Figure 7: A group profile for the hcheck (compiler version 3)

# 5 Cure or Prevention?

Heap profiling is a diagnostic tool for programmers who want to detect, understand and hence cure space faults in their programs. Although this tool is quite effective, we are bound to ask whether the use of other techniques might prevent the introduction of space faults in the first place.

Where a space fault is due to *excessive laziness*, as with the dragging problem of region 'C', it may be cured by forcing a little more evaluation in just the right place. But it is not always as easy as it was in this case to determine an appropriate source level reformulation. Many problems of this kind can be avoided at the implementation level, by introducing quasi-parallel reduction rules into the garbage collector. Wadler [10] has described such a scheme for the common case of selection from a tuple of multiple results, and this scheme is generalised in von Dorrien's *stingy evaluator* [9]. Indeed, a stingy evaluator is distributed with the LML compiler — but it is not used for the bootstrap compilation!

The fault of region 'A' was caused by a definition that was not lazy enough, being over-strict in a (large) argument value. To help avoid such faults, implementations might check and/or generate strictness declarations. Similar support for type declarations has been available, and widely valued, for some time. The comparison is pertinent not only because both forms of declaration can characterise useful properties of defined functions, but also because strictness analysis can be approached as a type inference problem [3].

Packed representations of character lists can bring about substantial savings in comparison with the usual "cons-cell" chains, even in the context of lazy evaluation, as Stove [7] observed several years ago. The use of this technique to reduce the size of region 'B' might easily be dismissed as a specific low-level optimisation, not linked to any general principle. But packed strings were already present in the LML implementation, with lazy unpacking as the interface to normal strings, and no unpacking at all in the context of top-level concatenation. So the issue is not just whether or not an implementation employs a special representation (for example, strings, unboxed values or dictionaries) but under what circumstances, and how easily the effect can be predicted or specified by the programmer. We suggest that if a compiler is capable of using a special representation in some circumstances, the programmer should have the opportunity to specify its use in other circumstances — a form of equal opportunity [4]. Recall also that we chose to rewrite some string operations in M-code to avoid the unpacking machinery: an alternative approach repacks strings during garbage collection — a special case of Turner's idea [8] that expressions should revert to an earlier, smaller form when space is short.

### 6 Related and Future Work

Although there have been many implementations of lazy functional languages, it seems there has been comparatively little work on profiling. However, we know of at least two other profiling systems that have been constructed recently [2, 6]. In comparison with the work on these other systems, the distinctive features of our own approach include:

- 1. putting all the emphasis on profiling *memory space*, rather than *processor time*, and defining overall cost in byte seconds;
- 2. multi-dimensional profiling (the two dimensions in our present profiler being producers and constructions) allowing sectioned or product profiles to be obtained;

3. stressing *application* of the technique by using the profiler to reduce the costs of existing programs.

Since the Ayr workshop, Augustsson has added heap profiling, as described in this paper, to the latest version of the LML compiler.

Although various extensions and refinements of our heap profiling system could increase its effectiveness — for example, the addition of an *age* dimension by tagging graph nodes with their time of creation — we have no immediate plans to develop it further. Rather, we plan to move on from profiling memory use to profiling *parallelism*, applying similar techniques to reveal and to improve the degree of parallelism in lazy functional programs.

# 7 Conclusions

We claim a successful outcome from the experimental self-application of our heap profiling version of the LML compiler. A target factor of two reduction in execution cost was reached after about three week's work (and this does *not* include the reduction obtained by fixing the M-code generator fault that we ourselves had introduced!). Based on our own experiments with heap profiling, and also on the experience of a small number of other users of our system, we suspect that *most* lazy functional programs of more than a couple of pages have space faults. The diagnosis and cure of such faults are neglected problems so much so that even rather simple techniques can lead to significant improvements. Perhaps surprisingly, similar techniques apply to both small and large programs — the compiler has more source *modules* than clausify has *lines* of code!

### Acknowledgements

Our thanks to Lennart Augustsson and Thomas Johnsson, whose work on the LML compiler forms the basis of our own, and to Simon Peyton Jones with whom we have had some useful discussions about heap profiling. We are also grateful to the referees for their comments.

This work was funded by the Science and Engineering Research Council.

#### References

- L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. Computer Journal, 32(2):127-141, April 1989.
- [2] S. Clayman, D. Parrot, and C. Clack. A Profiling Technique for Lazy, Higher-Order Functional Programs. Technical report, Department of Computing Science, University College London, November 1991.
- [3] T-M. Kuo and Mishra. Strictness Analysis: A New Perspective Based on Type Inference. In *Proceedings of the 1989 Conference on Functional*

Programming Languages and Computer Architecture, pages 260–272. ACM Press, September 1989.

- [4] C. Runciman and H. W. Thimbleby. Equal Opportunity Interactive Systems. International Journal of Man-Machine Studies, 25:439-451, 1986.
- [5] C. Runciman and D. Wakeling. Heap Profiling of Lazy Functional Programs. Technical Report 172, Department of Computer Science, University of York, April 1992.
- [6] P. M. Sansom and S. L. Peyton Jones. Profiling Lazy Functional Programs. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, *Glasgow 1992.* Springer-Verlag, Workshps in Computing, 1992.
- [7] W. Stoye. The Implementation of Functional Languages Using Custom Hardware. PhD thesis, University of Cambridge Computer Laboratory, December 1986. Technical Report No. 81.
- [8] D. A. Turner. A new implementation technique for applicative languages. SOFTWARE — Practice and Experience, 9(1):31–50, January 1979.
- [9] C. von Dorrien. Stingy Evaluation. Licentiate Dissertation, Chalmers University of Technology, S-412 96 Göteborg, 1989.
- [10] P. Wadler. Fixing Some Space Leaks with a Garbage Collector. Software — Practice and Experience, 17(9):595-608, September 1987.