

Problems & Proposals for Time & Space Profiling of Functional Programs

Colin Runciman and David Wakeling
University of York*

1 Problems

Many functional programs are quick and concise to express but large and slow to run. A few critical parts of the program may account for much of the time and space used: replacing these with something cheaper, or applying them less often, significantly improves overall performance of the program. The repetition of this refinement process, starting from a deliberately artless prototype, is a popular strategy for software development. But its effectiveness depends on the ability to identify the “critical parts”, and this is far from easy in a lazy functional language. Commenting on the experiences of a colleague developing a large application with their LML system, Augustsson and Johnsson note that

After long and painful searching in the program he finally found three places where full laziness was lost, each of which accounted for an *order of magnitude* in time (and it is not guaranteed that all such places were found) ... If a program does not run fast enough it can be quite difficult to find out why [1].

(The italics are ours.) There is now general agreement in the functional programming community that a major weakness of lazy functional languages is the difficulty of reasoning about their space and time behaviour [2].

1.1 Why Conventional Profilers May be Inapplicable

Conventional programming systems have tools for profiling such as *gprof* [3] and *mprof* [4]. These automate the installation and reading of *meters* to record information such as the number of times each procedure is called for each static point of call, the proportion of time spent in each procedure (estimated by regular sampling of the program counter), and the amount of memory dynamically allocated by each procedure. Costs incurred by lower level procedures are also aggregated appropriately so that they can be attributed to calling procedures higher up in the program call graph. The availability of such information greatly eases the task of identifying the critical parts of a program. However, these profiling tools of procedural programming systems may be of limited use to the functional programmer, for reasons such as the following.

*Authors' address: Department of Computer Science, University of York, Heslington, York YO1 5DD, United Kingdom. Electronic mail: colin@uk.ac.york.minster, dw@uk.ac.york.minster

1. *The semantic gap*: Program counter sampling may give good estimates of the proportion of time spent in different parts of the *object* code, but this accuracy is not very useful if these “parts” cannot be identified in the *source* code. Functional programs, unlike procedural ones, do not map very directly into computer instructions; functional language implementations involve radical transformation. Measurements of a run-time profiler may therefore be difficult to associate with structural units of the source code, and programs tuned under one implementation may need significant re-tuning to run well under others.
2. *Hidden routines*: Not only may the structure of compiled code depart radically from the source, the auxiliary routines of a functional run-time system may carry out a significant proportion of the computational work. Storage management in general, and garbage collection in particular, may be expensive. Simply passing responsibility up the call graph, as a conventional profiler would, unreasonably charges the full cost of a garbage collection to the program part whose request for memory allocation happened to trigger it.
3. *Global laziness*: Lazy evaluation makes it difficult to assess the costs of isolated program parts. Laziness is a global policy not a local programming technique. The extent of evaluation and the degree to which computational work is shared or avoided is context dependent in subtle ways. For example, a supposed refinement that avoids the evaluation of a costly expression may only shift the point of evaluation to a different context where the result was previously shared.
4. *Space leaks*: In view of the current multi-megabyte storage requirements of many functional programming systems, space costs are perhaps even more important to address than time costs. The lazy evaluation strategy can cause large memory demands of a distinctive kind. Laziness cuts the number of reductions by evaluating expressions only if needed, and then only once. However, this means expressions may be held unevaluated even though their results would occupy much less space: conversely, large results of evaluating small expressions may be retained for a long time until they are used again (or until it is realised that they are no longer needed).
5. *Recursion & cycles*: Profilers for conventional languages, such as *gprof* [3] and *mprof* [4] actually attach measurements to a *condensed* call graph, collapsing strong components to a single point. Since most functional programs use recursive and mutually recursive functions heavily, such condensation could involve the loss of a great deal of information.

The functional programming world has been aware of some of these difficulties for several years — pertinent observations go back at least as far as [5] — but there have not been many suggestions about how to resolve them. Current implementations of purely functional languages typically provide only very basic computational statistics such as the number of reductions performed (on some abstract machine), the number of memory cells allocated (ditto) and the number of garbage collections.

2 Proposals

2.1 Source-level Evaluation with Parametrised Costs?

It is often assumed that to overcome the *semantic gap* existing implementations must be adapted to preserve more source level information in (or about) compiled code. An alternative approach is to construct an interpretive profiler using a *source level graph reduction model*. The idea is to *approximate* the characteristics of various implementations, estimating the time and space costs by a formula combining raw measurements (*eg.* product of reduction count & number of new graph nodes per contractum for each rule). A particular target implementation is represented by a set of values for parametric weights in these formulae, and the formulae can always be refined by generalising them to take into account additional parameters (*eg.* distinctions between *constructors* and other nodes, or between *binary* and *vector* application nodes).

Advantages? The problem of attaching profiling information to units of the source code is greatly simplified. A *single* development of a comparatively simple system approximates *many* separate modifications of already complex implementations. An interpreter with parametrised profiling might also be a useful tool for comparing the relative importance of implementation strategies for particular applications, and for assessing the potential gains from new implementation ideas.

Disadvantages? An interpretive profiler might only give crude approximations of costs for some kinds of implementation (*eg.* those employing *deforestation* [6]) but even these crude profiles may be better than no profiling information at all. The low speed of an interpreter might limit its application: but even a profiler two orders of magnitude slower than implementations designed for speed can still perform substantial computations while the programmer is eating, sleeping or otherwise occupied. Besides, a single profiling run generates enough information to engage the human programmer for some time in consequent program refinements. Lastly, perhaps some elements of modern functional languages cannot be modelled conveniently using source-level graph reduction: the issue is contentious, but in any case tackling a simplified version of the problem first may be quite sensible. Even for rather limited forms of functional program — say in a first order subset of Haskell [7] with no “extras” such as local definitions or comprehensions — the pragmatics of pattern matching and lazy evaluation pose plenty of difficulties.

2.2 Profiles of Object Program Space?

The main concern of profiling is the changing pattern of resource consumption during a computation, and for most implementation methods space taken up by the program *as distinct from the graph* is fixed throughout. (In interpretive implementations based on fixed combinators, however, the program and the graph may be treated as one; the program is “self-modifying”, and indeed “self-optimising” [5]). But recent implementation methods involve a significant expansion of program size, and the degree of expansion varies for different kinds of expression:

... functions defined using a fair number of equations, where the patterns in the equations are complex. The pattern matching transformation phase in the compiler *expands such programs considerably* ... [1]

... though the above partial evaluation strategy is safe, it may induce a certain degree of “*code explosion*” ... it may be very efficient with respect to execution time ... but it may be *very inefficient with respect to code size* [7].

(Again, italics are ours.) As Thomas Johnsson has pointed out to us, if a program uses a heap of several megabytes, then it is not unreasonable to expand its size from say 100K to 1M bytes, particularly if this also reduces heap usage. But expanding 1M to 10M bytes would be more problematic! It may therefore be useful to know the size of object code generated for each function definition (say).

2.3 Aggregate Profiles ?

The whole of a functional computation can be regarded as nothing but the production and consumption of pieces of graph. To understand why a graph becomes so large, or why the process of reducing it is so long, one might therefore begin by identifying the high-volume *producers* and *consumers* of nodes.

The graph representing the state of a functional computation may grow and shrink dramatically during its history. It is typically small at the start, representing a “main expression” whose value is to be computed. In mid-computation, it may become a very complex structure occupying several megabytes of memory. In the end it will reduce to nothing as the last reduction permits the output driver to complete whatever remains of its value-printing task. This is illustrated nicely in the paper by Hartel and Veen [8].

Every node in the graph at any point must either have been present in the original graph (representing the “main expression”), or else it must have been created by some previous reduction step. The *producer* of a graph node (let us say) is the reduction rule, whether primitive or a program clause, that caused it to be introduced into the graph — a distinguished producer *main* can be defined for nodes in the original graph.

It is not so easy to define *consumers*. It is difficult even in a simple first order programming system without “apply” nodes — in which, for example, `tail (x:xs)` is represented not by the graph shown in Figure 1(a), but by the the one shown in Figure 1(b).

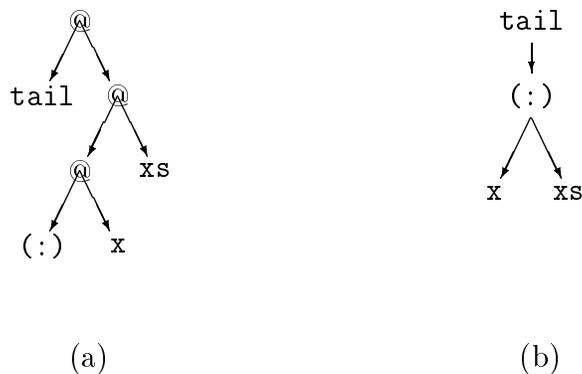


Figure 1: Two alternative representations of `tail (x:xs)`

In such a system, reduction by the rule `tail (x:xs) = xs` consumes the graph node for `tail` in the twin sense that it is *used* (it is necessary for the redex to exist) and it is *disposed of* (after the reduction it is no longer part of the graph). But what of the `(:)` node? Though necessary for the reduction it may not disappear from the graph since it may be shared — it may be used but not disposed of. And what of the graph corresponding to `x`? It may be disposed of without being used.

It seems necessary to distinguish between the two forms of consumption. Let's call a primitive or programmed reduction rule r a *user* of a graph node n if some reduction using the rule depends on the occurrence of n in the redex. And let's call it the *disposer* of n if n is finally detached from the graph as an immediate consequence of a reduction using r . If the *output driver* is regarded as a distinguished user and disposer of graph nodes, then each node has zero or more users but exactly one disposer.

Producer-consumer matrices

We envisage a profiling implementation that maintains information about the production and consumption of every graph node. This information can be summarised in a *producer-consumer matrix*, with a column for each producer and a row for each user or disposer. Entries at (i, j) present three figures about nodes *produced* by i : (1) the number of such nodes *disposed of* by j , (2) their *average lifetime* (in reductions), and (3) the number of *uses* of such nodes by j . Rather than have only a single matrix summarising production and consumption of *all* graph nodes, it may be useful to classify nodes in some way and to compute a matrix for *each class*.

To show which classes of graph nodes account for most space used by a program, and which program clauses are most heavily involved in their production, use and disposal, matrices could be ranked (*eg.* by the product of total volume and overall average lifetime of nodes they represent). Similarly, for rows or columns within each matrix. To reduce the amount of information presented to the programmer, matrices/rows/columns representing a low volume of graph nodes could be suppressed.

Notice that both call-frequency and call-graph information is present in producer-consumer matrices since each function call involves the production of an application node tagged with the identity of the “calling” function and its subsequent use and disposal.

It is not obvious how to build a profiler that can produce all this information efficiently. A partial implementation of the idea should be straightforward. Graph nodes can be *tagged* on creation with a production time (*i.e.* reduction number) and the identity of the producer. Lifetimes can be determined approximately by the garbage collector, since the disposal of “fresh” garbage must have occurred at some time since the last collection, and the frequency of collections can always be increased to obtain greater accuracy. Recording node uses is tricky: it could require a lot of space for some nodes, but for applications a single bit may suffice. Determining the identity of disposers could be very expensive: doesn't it require garbage collection in the redex after every reduction, and then in the wake of any processing by the output driver?

Example

Assuming the definitions

$$\text{listmin } (x : y : zs) = \text{listmin } (\text{min } x \ y \ : \ zs) \quad (1)$$

$$\text{listmin } [x] = x \quad (2)$$

$$\text{min } x \ y = \text{if } (x < y) \ x \ y \quad (3)$$

$$\text{if True } x \ y = x \quad (4)$$

$$\text{if False } x \ y = y \quad (5)$$

a tagged reduction of `listmin [M,I,N]` where $I < M < N$ is shown in Figure 2, and the producer-consumer matrix for `(:)` nodes during this computation is shown in Table 1. (Note that in this example, users and disposers coincide.)

| (:) nodes | | Production | | |
|-----------|----------|------------|------|----------|
| | | (0) | (1) | Σ |
| Use & | (1) | 3 | 2 | 5 |
| disposal | (2) | 0 | 1 | 1 |
| | Σ | 3 | 3 | 6 |
| average | | 1.00 | 1.33 | 1.20 |
| lifetime | | | | |

Table 1: Aggregate production and consumption of `(:)` nodes for `listmin [M,I,N]`

Application to Reducing Graph Space?

A programmer may wish to minimise either the *average* size of the graph, or its *maximum* size. In either case, it should be possible to identify a *target group* of graph nodes. If the aim is to reduce the average then the target group should contain the *longest lasting high volume* nodes, and if the aim is to reduce the *maximum*, the target group should contain the *highest volume nodes in the largest graphs* occurring during the computation.

The first aim is to modify the program to *avoid creating* the nodes in this target group — for example, by fusing together the functions that produce them and the functions that use and discard them. An alternative (or additional) aim is to *shorten the time* for which they are retained in the graph. This may be achieved by forcing earlier evaluation of expressions involving them to yield a small result such as a basic value. Another way to shorten the lifetime of graph nodes is to avoid sharing large results involving them when the uses of these results are a long time apart.

2.4 An Accounting Procedure for Garbage Collection?

The cost of garbage collection in a lazy functional programming system has been estimated at *up to a quarter* of total computing time [9], so it should not be ignored. But what is an appropriate *accounting procedure* for charging collection costs to specific reduction rules of a program?

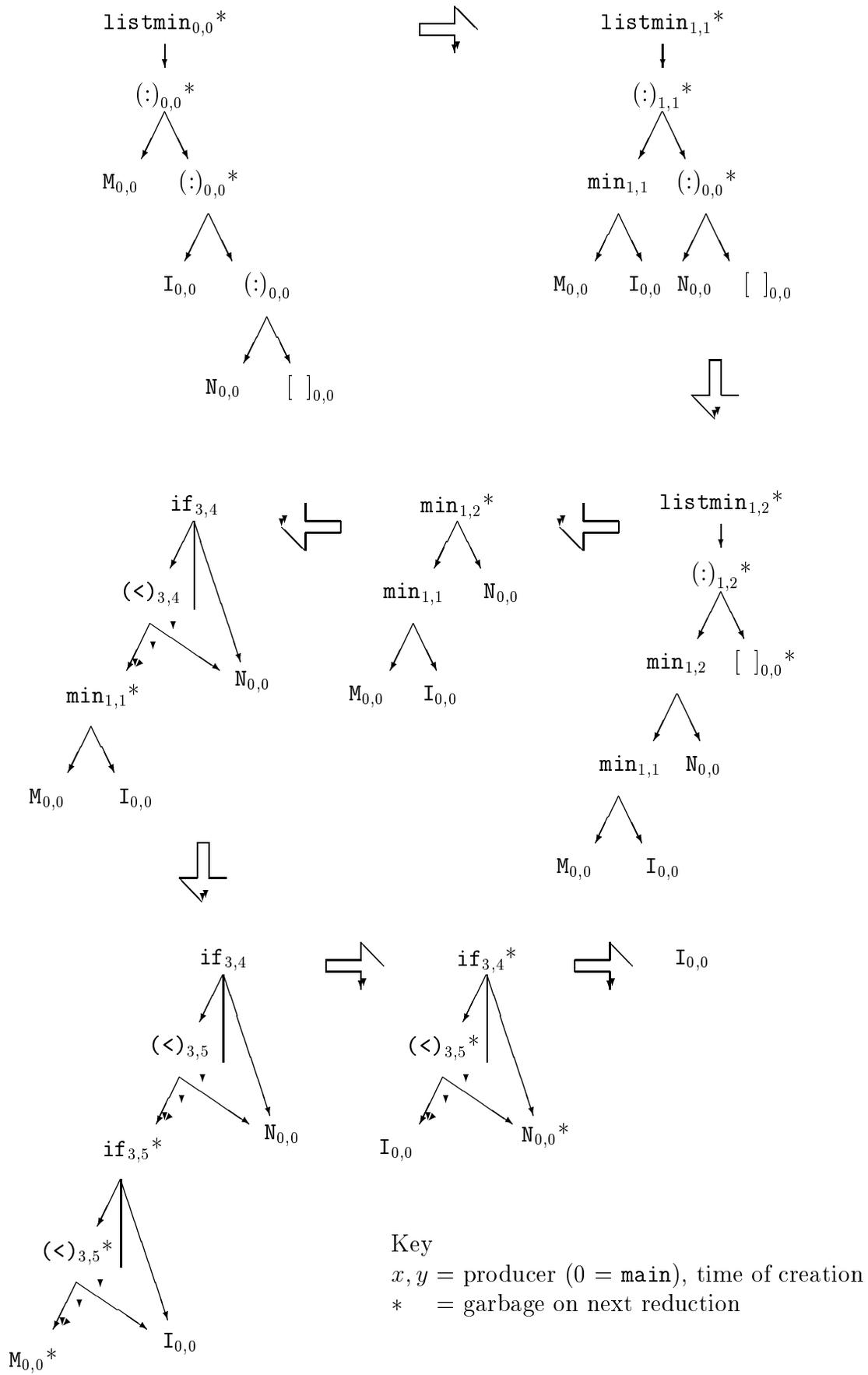


Figure 2: Tagged Reduction of listmin [M, I, N]

1. *Charge the disposer?* This is hardly appropriate. Would it be preferable for a rule to retain in the graph as many nodes as possible? Surely not. A disposer should be rewarded, so to speak, for rendering a valuable service!
2. *Charge the producer?* This seems a little more reasonable, until one considers that under lazy evaluation graph production is always in response to the needs of another program component or to the demands of the output driver.
3. *Charge the users?* This seems *far* more reasonable. Moreover, costs could be distributed so that the heaviest users pay the most, not forgetting the output driver.

However, some graph nodes may have *no* users. A producer may generate more graph than is actually needed, in which case it may after all be appropriate to charge it for the recycling of its excess production.

2.5 Serial Profiles?

Summary tables such as producer-consumer matrices provide aggregate information such as *totals* and *averages*, for an entire computation. An alternative or complementary approach is to provide a *serial* profile: that is, a summary of the most important characteristics of the graph at regular intervals (perhaps in conjunction with garbage collections).

Even extremely limited versions of such serial profiles displayed on monochrome ASCII devices can be quite informative. From the crude serial profile of the reduction of `listmin [M,I,N]` shown in Figure 3, one may verify at a glance, for example, that the graph never exceeds its original size. For larger computations, the scale of a profile is easily adjusted by using one character to represent x nodes and sampling after every y reductions.

| | |
|----------|-----------------------------|
| *****@ | |
| *****@@ | |
| *****@@@ | Key |
| ***@@ | * = irreducible constructor |
| ***@@@ | @ = reducible application |
| ***@@@@ | |
| **@@ | |
| * | |

Figure 3: Crude serial profile of `listmin [M,I,N]`

A more satisfying refinement of this idea might be achieved on a colour workstation. A complete profile could be formed as a vertically arranged series of horizontal bands, each band composed of several coloured segments where the colours, arranged in a fixed order, represent different classes of graph nodes. The “cons” nodes of list structures, for example, might be represented in blue — dark blue for those most

recently produced, and pale blue for the oldest. The extent of each colour and shade would vary between bands, being be proportional to the volume of the corresponding nodes found in the graph at successive stages in the computation. In this way, high volume node types, including those retained in the graph for a long time, or those predominating in graphs of peak sizes, would stand out at a glance. Ideally such a system would allow its users to specify the colour schemes — for example, depending on the program, colouring by producer may be more informative than colouring by the type of node.

3 Summary & Conclusion

There is no doubt about the outstanding need for profiling tools in lazy functional programming. The subtleties of lazy evaluation and the sophisticated transformations performed by modern functional compilers limit the effectiveness of conventional tools. This paper has put forward only a few tentative proposals: their usefulness must be tested by building and using experimental profiling systems.

4 Acknowledgements

Thanks to Sandra Foubister and the Referees.

References

- [1] Augustsson L and Johnsson T. The Chalmers Lazy-ML compiler. *Computer Journal*, 32(2):127–141, April 1989.
- [2] Peyton Jones SL. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- [3] Graham SL, Kessler PB, and McKusick MK. An execution profiler for modular programs. *Software — Practice and Experience*, 13:671–686, 1983.
- [4] Zorn B and Hilfinger P. A memory allocation profiler for C and LISP programs. *USENIX 88*, pages 223–237, 1988.
- [5] Turner DA. A new implementation technique for applicative languages. *SOFTWARE — Practice and Experience*, 9(1):31–50, January 1979.
- [6] Wadler P. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.
- [7] Hudak P and Wadler P (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical report, University of Glasgow, Department of Computer Science, April 1990.
- [8] Hartel PH and Veen AH. Statistics on graph reduction of SASL programs. *Software — Practice and Experience*, 18:239–253, 1988.

- [9] Augustsson L. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Chalmers University of Technology, S-412 96 Göteborg, November 1987.