Heap Profiling of Lazy Functional Programs

Colin Runciman and David Wakeling
University of York*

Abstract

We describe the design, implementation and use of a new kind of profiling tool that yields valuable information about the memory use of lazy functional programs. The tool has two parts: a modified functional language implementation which generates profiling information during the execution of programs, and a separate program which converts this information to graphical form. With the aid of profile graphs, one can make alterations to a functional program which dramatically reduce its space consumption. We demonstrate this in the case of a genuine example — the first to which the tool was applied — for which the results are strikingly successful.

1 Introduction

There was a time when it seemed that almost every paper on functional programming began with the standard arguments about the semantic elegance and simplicity of functional programming languages. We shall spare the reader another rendition of them here. Suffice it to say that the *extensional properties* of a functional program (what it computes as its result) are usually far easier to understand than those of the corresponding imperative one. However, the *intensional properties* of a functional program (how it computes its result) can often be much harder to understand than those of an imperative one, especially in the presence of higher order functions and lazy evaluation.

Several authors have observed this problem. In his thesis, Meira [13] uses a number of examples to show that it is hard to write *efficient* functional programs because of the need to understand the underlying order of expression evaluation. Wray [26] also notes that lazy evaluation leads to uncertainty about time and space behaviour, as does Stoye [21], who laments the fact that so much research is directed towards improving implementation performance, rather than towards providing profiling facilities. All of this work is tidily summarised in a chapter of Peyton Jones' book [14].

The problem of reasoning about the time and space complexity of functional programs can be addressed in two different ways. These might be called the *theoretical* and the *practical* approaches.

*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`

The theoretical approach attempts to develop a framework that allows the programmer to reason about the intensional properties of the program using similar algebraic methods to those used to reason about the extensional ones. There has been some work in this area, notably by Wadler [25], Bjerner and Holmström [4] and Sands [19] concerning time behaviour. However, the problem is a hard one and progress has so far been modest.

The practical approach involves the construction of profiling tools which gather information when the program is executed. Such execution profiles assist the programmer by revealing the underlying intensional properties of the program. However, current functional language implementations provide only the most rudimentary profiling facilities. A measure such as the number of reductions performed does *not* correlate reliably with actual execution time (in seconds), because some kinds of reduction are much slower than others — an extreme example of a slow reduction is one that involves a garbage collection. Neither does a measure such as the number of heap cells allocated correlate well with actual memory demand (in bytes), because it ignores the pattern of allocations and the lifetimes of cells — on account of these the peak memory demand of one program may vastly exceed that of another which allocates the same number of cells.

This paper describes the implementation and use of a new tool for profiling the space consumption of lazy functional programs. It is organised as follows. Section 2 describes *heap profiling*, a simple way for a functional language implementation to furnish the programmer with information about how memory is being used. Heap profiles are best understood when they are drawn as graphs, and Section 3 is concerned with some graphical design issues. Aspects of an LML implementation of heap profiling are described in Section 4. Section 5 presents an extended example, demonstrating that the use of heap profiling can dramatically improve the space behaviour of a lazy functional program. Section 6 discusses some issues arising from the example, and Section 7 describes how the tool was subsequently developed. Section 8 reviews some closely related work, Section 9 makes some suggestions for future work and Section 10 concludes.

2 Heap Profiling

Before we can execute a functional program we must decide how it is to be represented in the computer's memory. A popular choice is as a *graph*, and in this case the program is executed by reducing the graph to *normal form*, printing the result as it becomes available. During reduction new graph nodes are produced and attached to the graph; existing nodes are consumed and detached from it. In other words, the execution of a functional program can be regarded as nothing more than the production and consumption of pieces of graph.

Graph reduction is implemented by storing nodes in cells allocated from a large area of memory called the *heap*. As new graph nodes are attached to the graph they are stored in new cells. When the supply of new cells runs out, the implementation may suspend reduction in order to determine the nodes that have become detached from the graph, and to recover their cells for re-use. Other approaches, such as reference counting, interleave this *garbage collection* process with graph reduction.

In current implementations, it is quite common to have several hundred kilobytes or even a few megabytes of heap memory. When the evaluation of a particular functional program needs such a large amount of memory, the programmer might reasonably ask a few questions. For example, what sorts of nodes in the graph occupy the most space for the longest time? Which functions were the (immediate) cause of those nodes being introduced into the graph? Our heap profiling tool is designed to supply precise answers to exactly such questions.

The first component of the tool is a modified compiler which attaches two *tags* to every cell in the heap. These tags, which are intended only for use by the profiler, identify:

- the function that produced the graph node;
- the construction that the graph node represents.

All nodes have an identifiable producer, even if it is only the SYSTEM which is assumed to produce the original graph and a few special nodes representing such things as command-line arguments and open files. However, not all nodes have an immediately identifiable construction because they represent *closures* which have yet to be reduced to normal form. For closures we take the construction to be the name of the function component, or UNKNOWN if we cannot easily determine what the function is. We shall say more about cell tags in Section 4.

When the programmer requests a heap profile, execution is suspended at specified regular intervals and the profiler traverses the 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.

3 Data Graphics

In the introduction to his excellent book on the design of statistical graphics, Tufte remarks:

> At their best, graphics are instruments for reasoning about quantitative information. Often, the most effective way to describe, explore, and summarise a set of numbers — even a very large set — is to look at pictures of those numbers. Furthermore, of all methods for analyzing and communicating statistical information, well-designed data graphics are usually the simplest and at the same time the most powerful [22].

The second component of our heap profiling tool is a program that generates a graph from a heap profile. This program produces PostScript*, and so the graph may be

*PostScript is a trademark of Adobe Systems Incorporated either displayed on a graphics workstation or printed on a laser-printer. An example graph is shown in Figure 1. It shows how the amount of heap storage occupied by the program (measured in bytes) varies over the time that the program takes to run (measured in seconds). Shaded bands are used to show how much of the total storage is taken up by cells tagged with each of the identifiers mentioned in the key. Since it is only possible to distinguish between a few shades of grey on the screen or the printed page, the shaded bands and the entries in the key cycle through only a small number of different tones. Any ambiguities are resolved by reading both in the same order.

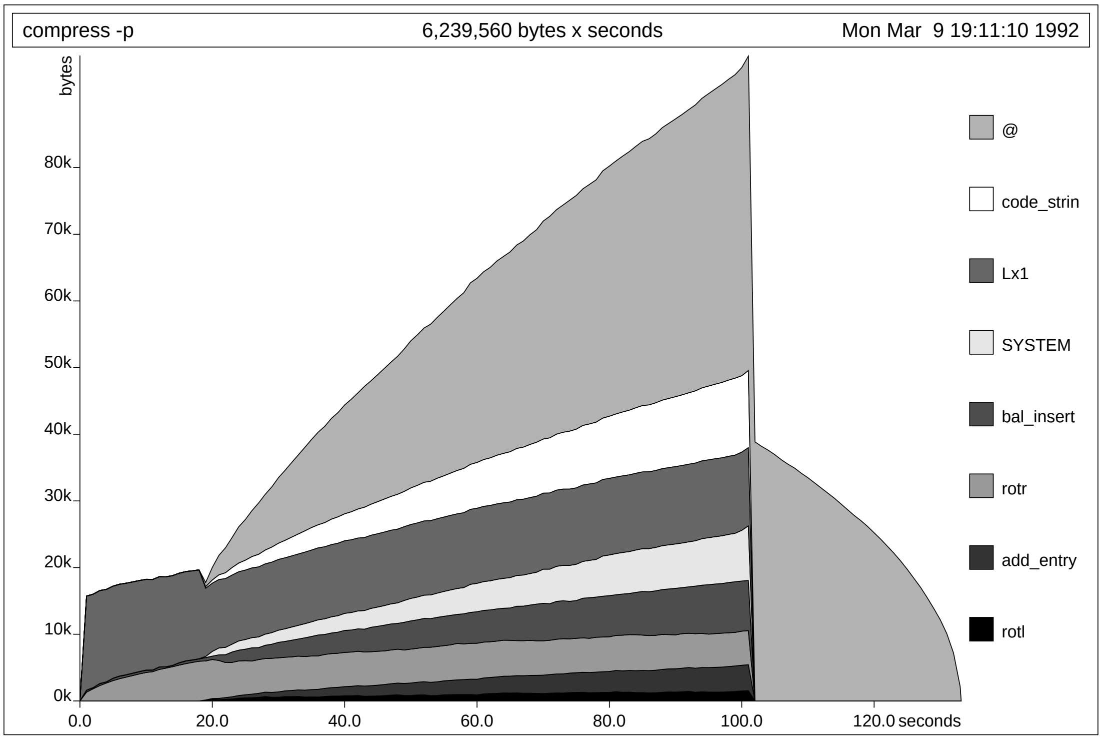

Figure 1: An example graph

The title of the graph comes in three parts: the name of the program together with the profiling options used, a figure representing the cost of the program as a product of bytes and seconds, and the date on which the program was run. The program in Figure 1 was called "compress"; the option selected was "-p", requesting a profile of the nodes according to which function produced them. The cost of running this program (calculated as the total area below the graph) is approximately 6.2 Mbs (megabyte-seconds).

Graphical design is a complicated business, and we have made every effort to follow Tufte's guidelines in order to avoid producing "chartjunk". Thus, our graphs attempt to maximize the *data-ink ratio*, which is defined to be the proportion of the graph's ink devoted to the non-redundant display of data-information. In some places though, our graphs still fall short of the ideal. One of Tufte's guidelines suggests that varying shades of grey should be used instead of cross-hatching. This prevents moiré effects (in which the design of the graph interacts with the physiological tremor of the eye) from producing a distracting appearance of vibration and movement. However, the same guideline also suggests that specific areas of the graph should be labelled with words, rather than being encoded with shades of grey via a key.

In addition to Tufte's guidelines, our experience has led us to develop three rules of our own. The first rule avoids cluttering up the key with identifiers that account for only a small fraction of the total storage allocated. All those identifiers which, when taken together, account for less than one percent of the total storage allocated by the computation are ignored when drawing the graph. In practice this "one percent rule" is remarkably effective. It removes *trace elements* from the graph, which then focuses clearly on those identifiers that occupy large amounts of space. The second rule is that every graph should occupy just one page. Originally, we tried graphs flowing over several pages, but these proved to be rather unwieldy and unhelpful. Our final rule is a consequence of the way in which the bands for each identifier are stacked on top of each other. It turns out that the graphs are much easier to understand when the smoothest bands (those representing series of values with the smallest standard deviations) are at the bottom, with the rougher bands stacked on top. This ordering also naturally focuses attention on the "troublemakers" at the top of the graph.

4 Implementation

Our implementation is based on Augustsson and Johnsson's LML compiler [9, 2]. In what follows we shall assume some familiarity with this compiler and the underlying idea of programmed graph reduction. For those without such familiarity, Augustsson and Johnsson's paper [3] provides a good overview, and an excellent tutorial description can be found Peyton Jones' book [14].

Our LML compiler has a modified lambda lifting pass and a new run-time system. These are described in turn below.

Lambda Lifting

In order to avoid the overhead of run-time environment management, the LML compiler converts all functions into supercombinators prior to code generation. This conversion involves binding the function's free variables as extra formal parameters and then passing values for these variables as extra actual parameters at every point where the function is called. Johnsson [8] christened this conversion *lambda lifting*, although the term *supercombinator* was invented by Hughes [6] who independently discovered a similar conversion.

As Peyton Jones notes [14], an unfortunate consequence of lambda lifting is that the bodies of some functions get broken up into many small fragments which are then turned into individual combinators. From the profiling perspective this causes problems because names generated for these fragments may bear no relation to the original function name and are therefore meaningless to the programmer. In our modified compiler the original function name appears as part of every new function name formed by the compiler during lambda lifting. The profiling tool can then recover and use the original function name.

The Run-time System

In our implementation we distinguish between *static* and *dynamic* cell tags. Static cell tags carry information determined at compile-time and dynamic cell tags carry information determined at run-time. Extra fields are added to every cell to accommodate these tags. 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 2 shows how a list cell is tagged. The static tags identify the function that produced the

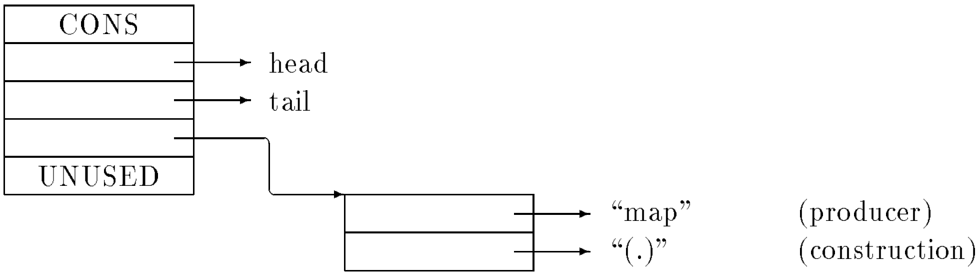

Figure 2: A tagged list cell

graph node, and the construction that it represents. Space is also reserved for one dynamic tag, but at the moment this is unused.

The garbage collector and other components of the run-time system must be modified in order to cope with the larger tagged cells. Although straightforward, these modifications are numerous and tedious to describe, and so we shall move on to consider the additions that we have made to the run-time system to allow the graph to be sampled at regular intervals.

Our first thought was to sample the graph after each garbage collection. This is easy to implement because garbage collection provides a natural break in program execution. In practice though, the resulting sample grain is often either too coarse (for very short computations) or too fine (for very long ones). A much better idea is to sample the graph every t seconds, where t is specified when the program is run. Of course, the graph had better be in good shape before it is traversed — there must be no partially-updated nodes, for example — and so sampling is performed as follows.

To the standard LML run-time system we add a clock which ticks at 50Hz. Each tick generates a signal which is fielded by a signal handler. When the signal handler detects that another sample interval has elapsed, a flag is set. Code to test this flag is planted by the compiler at the beginning of the code for each function. If the flag is set then graph reduction is suspended, the graph is sampled, the flag is reset, and reduction continues. Testing the flag at the start of each basic block ensures that when the graph is sampled there are no partially-updated nodes, and that all of its roots are on the pointer stack rather than in registers. The samples themselves are recorded in an ordinary text file.

The space occupied by the cell tags is ignored during sampling, as are any clock ticks that may occur. As a result, the profile graphs do not depict the additional space and time that is required by the implementation to store and gather heap profiling information.

One consequence of interval-based sampling is that no two runs of a program yield identical sample files. Virtual system time is derived from real system time, and variations can occur as a result. In our SUN-3 implementation, for example, the real time clock interrupts at 50Hz (100Hz divided by 2 in software), and the operating system charges whichever process happens to be running then with the full 50th of a second. Thus, context switching, interrupts and the phase relationship between a process' start time and the 100Hz signal all combine to produce variations between runs which are reflected in the sample file. In practice, they are not a problem: invariably, one is interested in the overall behaviour of the program and minor differences between runs do not matter.

5 An Example

In this section we shall give an account of the very first series of experiments we conducted with our heap profiling tool. As the subject of these experiments, we deliberately chose a small (130 lines), but by no means trivial, program that:

(a) had been written long before we started work on profiling;

(b) had been used quite satisfactorily by classes of students;

(c) had already been "neatened up" for presentation to functional programmers, and so could be assumed to have no glaringly obvious defect.

This program, which we shall call *clausify*, takes as input a series of propositional formulae, and yields as output their clausal form equivalents. The required transformation of each proposition to a set of clauses can be specified by the following rules.

The *elim* rule, which eliminates equivalences and implications:

$$\begin{array}{rcl} p = q & \longrightarrow & (p \Rightarrow q) \wedge (q \Rightarrow p) \\ p \Rightarrow q & \longrightarrow & \neg p \vee q \end{array}$$

The *negin* rule, which makes negations the innermost connectives:

$$\begin{array}{rcl} \neg\neg p & \longrightarrow & p \\ \neg(p \vee q) & \longrightarrow & \neg p \wedge \neg q \\ \neg(p \wedge q) & \longrightarrow & \neg p \vee \neg q \end{array}$$

The *disin* rule, which pushes disjunctions within conjunctions:

$$\begin{array}{rcl} p \vee (q \wedge r) & \longrightarrow & (p \vee q) \wedge (p \vee r) \\ (p \wedge q) \vee r & \longrightarrow & (p \vee r) \wedge (q \vee r) \end{array}$$

The *split* rule, which splits up conjuncts:

$$\begin{array}{lcl} p \wedge q & \longrightarrow & p \\ & & q \end{array}$$

The *unicl* rule, which forms a set of unique non-tautologous clauses:

$$q_1 \vee \ldots \vee q_n \vee \neg p_1 \vee \ldots \vee \neg p_m \quad \longrightarrow \quad (\{q_1, \ldots, q_n\}, \{p_1, \ldots p_m\})$$

A clause (qs, ps) is tautologous if $(qs \cap ps) \neq \emptyset$.

The implementation of the above transformation rules in the clausify program uses the following type definition for the abstract syntax of propositional formulae — note the constructors `Sym`, `Not`, etc which will shortly figure prominently in our discussion.

```
type Prop  = Sym Char
           + Not Prop
           + Con Prop Prop
           + Dis Prop Prop
           + Imp Prop Prop
           + Eqv Prop Prop
```

The heart of the program is a "pipeline" composition of several functions, each corresponding to one of the rules given above.

```
... unicl o
      split o
         disin o
            negin o
               elim o
                  parse ...
```

Appendix A contains a full source listing of the clausify program, including definitions of all these functions.

For the purposes of the profiling experiments we also needed to select a fairly demanding proposition as a benchmark problem to be used as input for the program. From inspection of the transformation rules, any proposition involving several equivalences can be expected to generate a substantial amount of work. Hence we chose to use the following proposition as input.

$$(a = a = a) = (a = a = a) = (a = a = a)$$

Applying the transformation rules to this proposition eventually reduces it to the single clause

$$(\{a\}, \emptyset)$$

but the intermediate formulae involved are indeed extensive.

Version 0

The first heap profiles that we produced for the original clausify program were a real surprise. Both the producer profile (Figure 3) and the construction profile (Figure 4) show the amount of heap space in use steadily increasing, with a peak of around 1.3Mb in use just before the program terminates. In a strict call-by-value language, we would

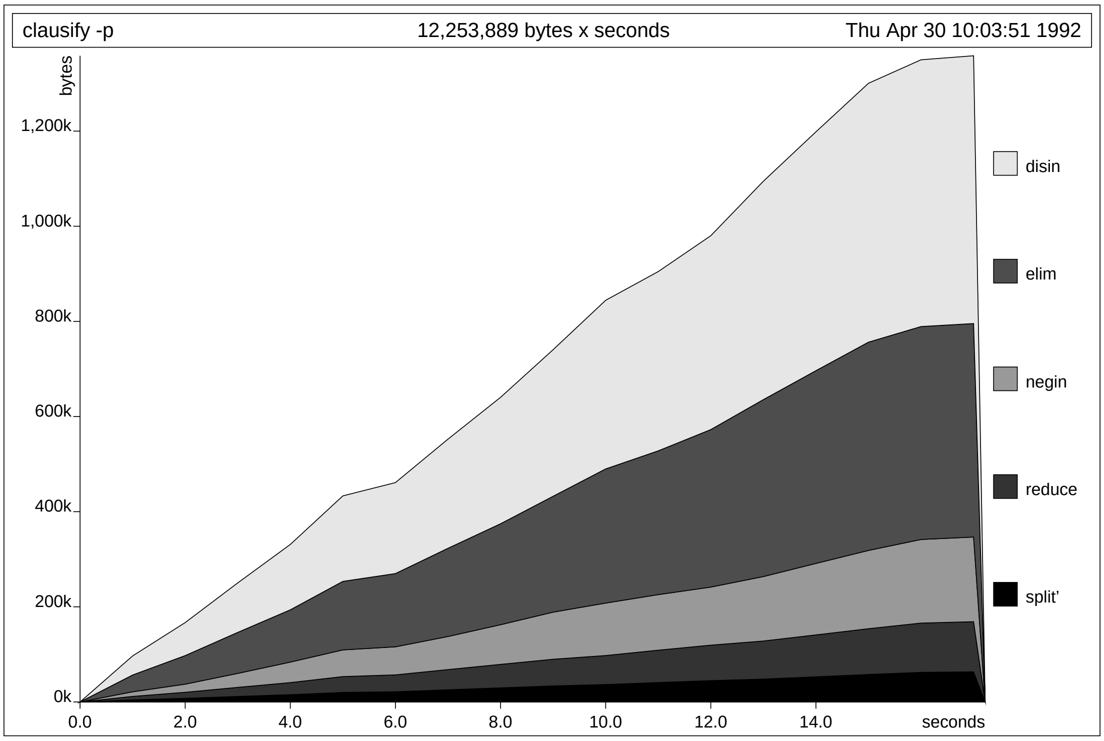

Figure 3: A producer profile for version 0

expect to see the graph grow and shrink as each successive intermediate form of the proposition in turn is computed "eagerly" in its entirety, and replaces its predecessor — the peak memory demand occurring when the largest intermediate form is reached. But in a non-strict call-by-need language, each intermediate form is constructed "lazily", and so we would expect to see demand-driven pipe-lining.

Our benchmark problem is a single proposition, whereas in general the program is used to normalise a series of propositions supplied by the user. The profiles show a single tooth of what would be an irregular saw-blade pattern for a complete session involving a variety of propositions.

As with the famous Sherlock Holmes case involving the significant dog in the night, the significance of which was that it did not bark, so with version 0 of clausify: Figure 3 shows the functions `elim`, `negin`, `disin` and `split'` (an auxiliary of `split`) producing nodes which are steadily being added to the graph. The next stage in the pipeline, `unicl`, should be consuming this structure and producing its clausal representation — to be printed as soon as it becomes available. In fact, however, the program yields no clausal output at all until the computation is almost over. From this observation, and the information in the heap profiles, it seems that `unicl` accumulates its input

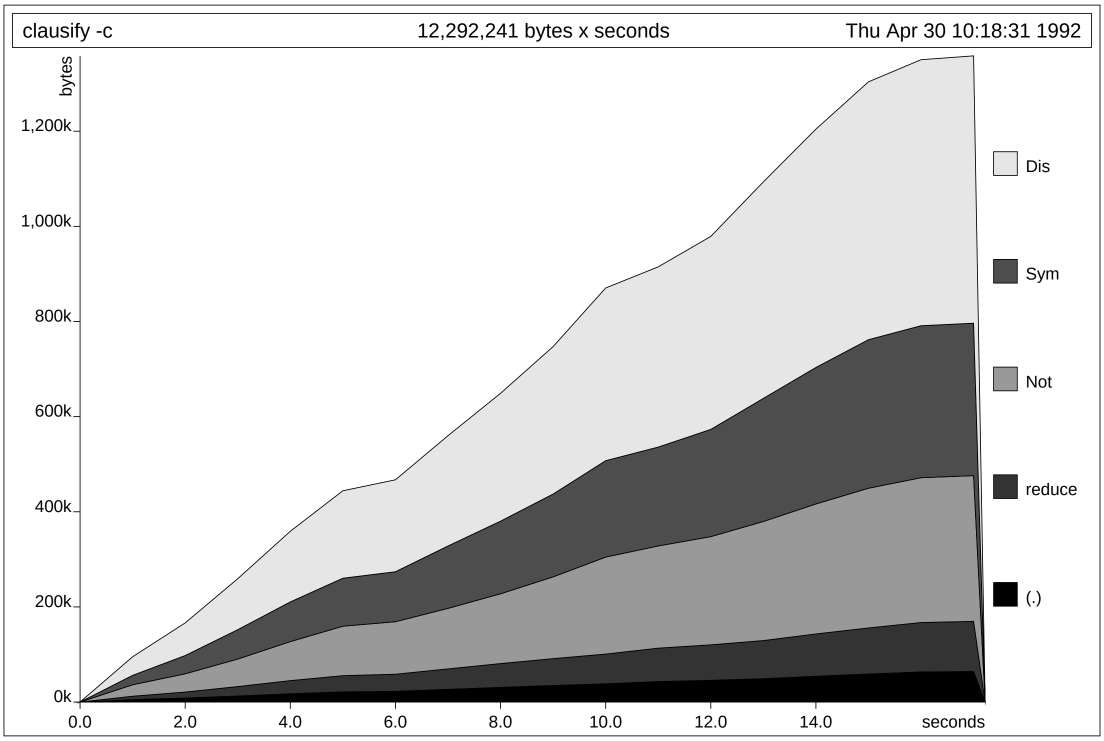

Figure 4: A construction profile for version 0

in memory until it has all been received, blocking the clausify "pipeline". Only when the entire input is available does `unicl` finally consume it in the production of clausal output. Examining the definition of `unicl`, the source of the blockage is soon apparent:

```
unicl a =
      let unicl' p x =
            (if tautclause cp then x else insert cp x
            where cp = clause p)
      in
            foldr unicl' [] a
```

This formulation of `unicl` is *tail-strict*: it demands to see all of its input — representing all of the conjuncts in a conjunctive proposition — before giving any output.

Version 1

It is not hard to write a version of `unicl` that is not tail-strict:

```
unicl = filterset (not o tautclause) o map clause

filterset = filterset' []

filterset' s p [] = []
filterset' s p (x.xs) =
      if not (mem x s) & p x then
            x .  filterset' (x.s) p xs
      else
            filterset' s p xs
```

The new version of `unicl` brings about an impressive factor of seven decrease in the cost of running the benchmark problem — compare the 12.1Mbs of version 0 in Figure 4 with the 1.7Mbs of version 1 in Figure 5, for which the same scale has been requested.

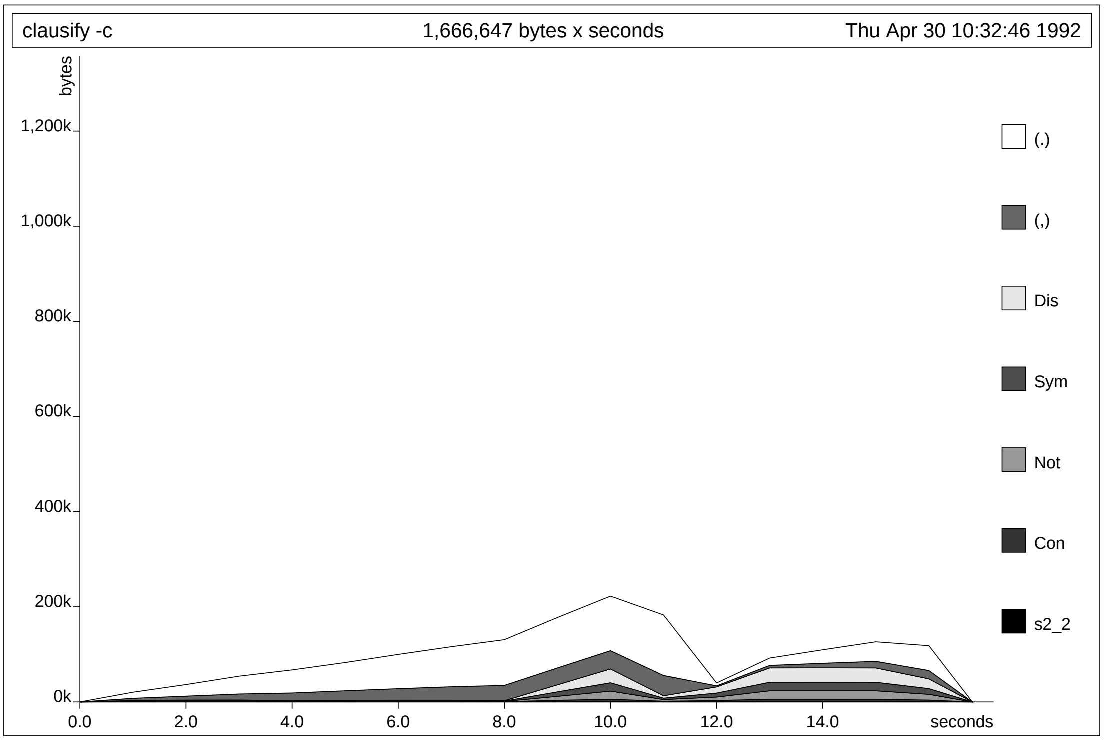

Figure 5: A construction profile for version 1 to same scale as Version 0

This though, is really just the beginning. By default, the new heap profile is scaled to fill the entire page, and this brings another problem to light (see Figure 6). List constructions (represented by (.) in LML) now dominate the computation. Once again, this is a real surprise because the pipeline nature of clausify would lead one to expect that list constructions would be very short-lived. In order to fix this new problem we first need to identify the producers of list constructions. Our heap profiling tool makes this quite easy: we simply re-run the program specifying that the only producers of interest are those of list constructions (see Figure 7).

From Figure 7 we can see that only the `insert` and `map` functions produce significant numbers of list constructions. The appearance of `insert` in this profile is as expected,

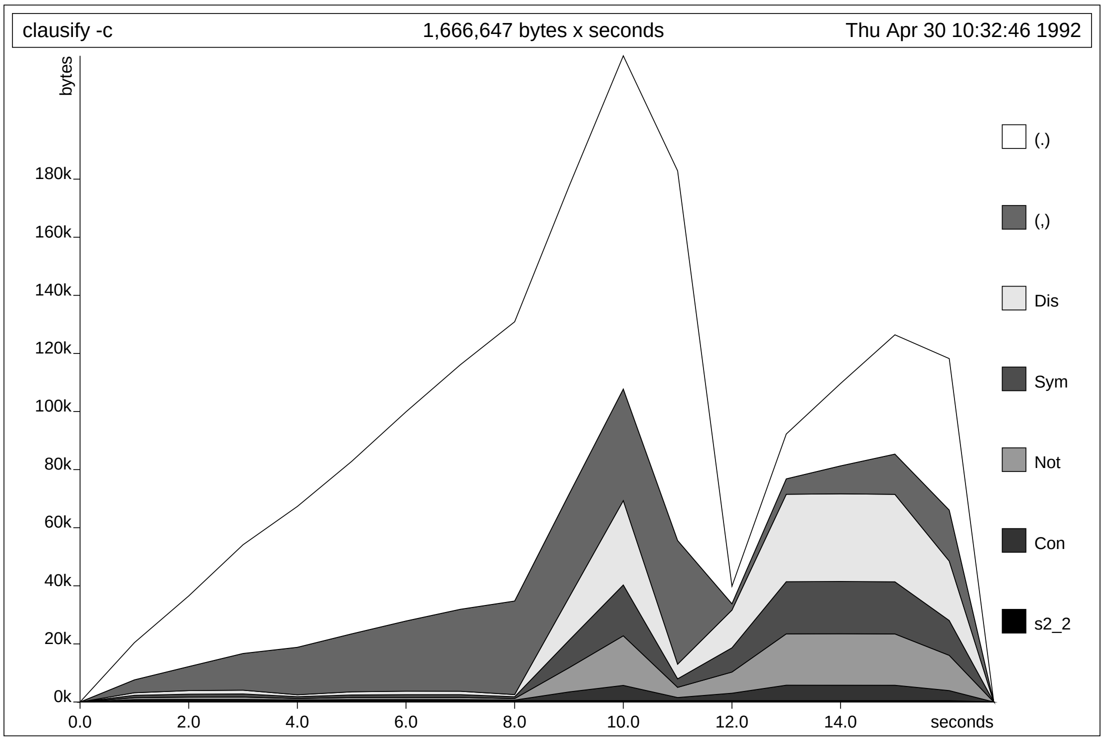

Figure 6: A page-scaled construction profile for version 1

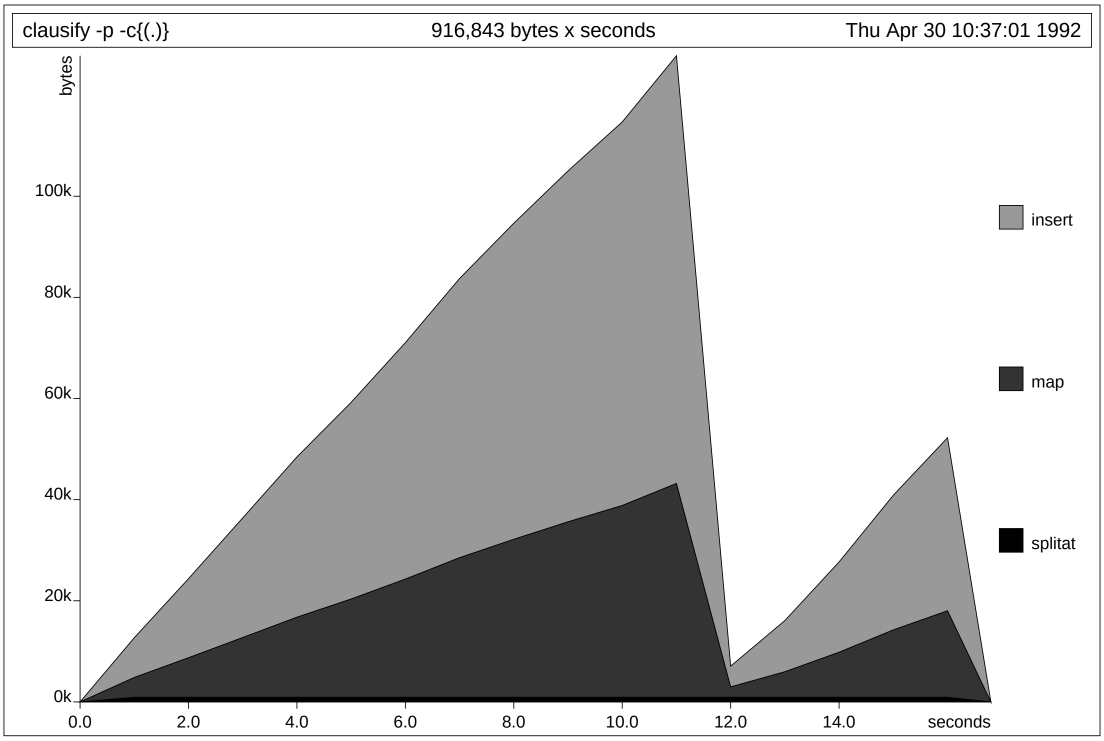

Figure 7: A producer profile for version 1 (producers of (.) nodes only)

but the appearance of `map` is completely unexpected. Studying the program, we can determine that the `map` in question must be the one used in the new definition of `unicl` because the only other uses of `map` generate but one list node per input proposition or per output clause. The curious thing is that the suspected instance of `map` generates a list for exclusive and immediate consumption by `filterset`.

After checking our profiling tool very carefully, we were forced to conclude that the standard LML compiler was at fault, and indeed this turned out to be the case. As Jones [10] explains, LML compilers of the vintage that we were working with are subject to a space leak which causes cells to be unnecessarily preserved by the garbage collector. The problem arises when tail-recursive functions, such as `filterset'`, are compiled into G-machine code. Figure 8(a) shows how the stack is arranged on entry to `filterset'`, and Figure 8(b) shows how it is rearranged prior to a tail call using the mysterious $\mathcal{S}$ rules hinted at by Johnsson in [9]. Shuffling the arguments to the original call

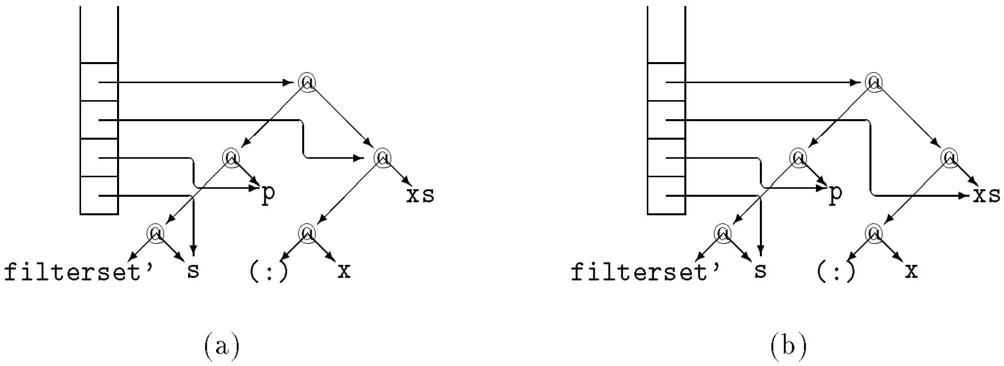

Figure 8: Stack rearrangement prior to a tail call of `filterset'`

of `filterset'` like this means that the stack frame is re-used and that recursion is effectively replaced by iteration. However, the root cell of the original redex is not overwritten and so everything accessible from it is preserved by the garbage collector. In the case of `filterset'`, list structure is unnecessarily preserved, and this accounts for the large number of list constructions that appear so unexpectedly in Figure 6.

Version 2

The way to avoid space leaks when performing tail recursion is to "blackhole" the root of the original redex by overwriting it with a HOLE node. We added a new BLACKHOLE instruction — a hybrid of ALLOC and UPDATE — to our G-machine, and modified our compiler to emit this instruction at the site of each tail call. The result was a further factor of four reduction in the cost of running the benchmark problem — from 1.7Mbs in Figure 6 to 0.4Mbs in Figure 9.

This new profile, with its striking twin peaks, prompts us to look for further improvements. The peaks represent large numbers of `Dis`, `Sym`, `Not` and `Con` constructions — not too surprising, perhaps, since the entire program is about reformulation of propo-

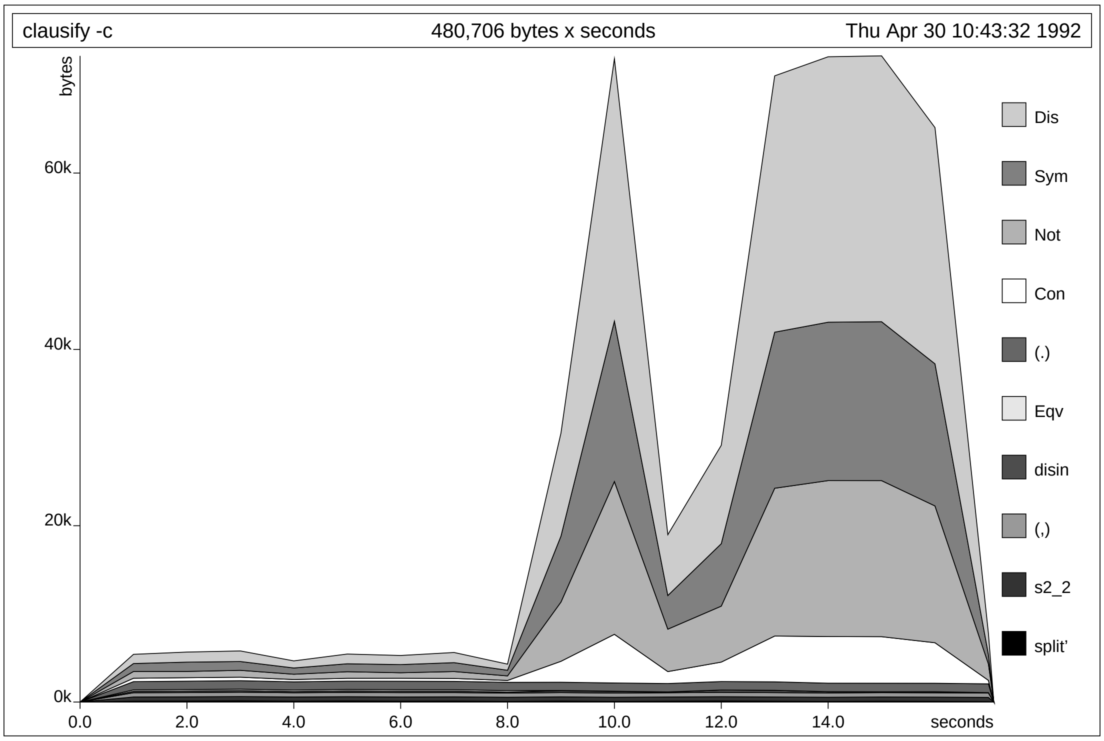

Figure 9: A construction profile for version 2

sitions. But wait a minute: why are there 15kb of `Sym` constructions representing basic proposition symbols? After all, the benchmark problem contains just nine "a" symbols, so its representation contains just nine `Sym` constructions. Since the transformation into clausal form does not involve introducing any new symbols, and since any replication of existing symbols ought to be achieved by sharing, we would certainly not expect the number of symbols to increase. Yet it does, as the profile in Figure 9 makes clear.

The producer of these extra symbols can be discovered by re-running the program to get a profile just for the producers of `Sym` constructions (see Figure 10). Now clearly there must be something wrong with the `elim` function, and a glance at its definition pinpoints the problem:

```
elim (Sym s) = Sym s
elim (Not p) = Not (elim p)
elim (Dis p q) = Dis (elim p) (elim q)
elim (Con p q) = Con (elim p) (elim q)
elim (Imp p q) = Dis (Not (elim p)) (elim q)
elim (Eqv p q) = Con (elim (Imp p q)) (elim (Imp q p))
```

The first clause of this definition creates a new `Sym` construction, even though the argument presents an existing identical construction which could be returned instead.

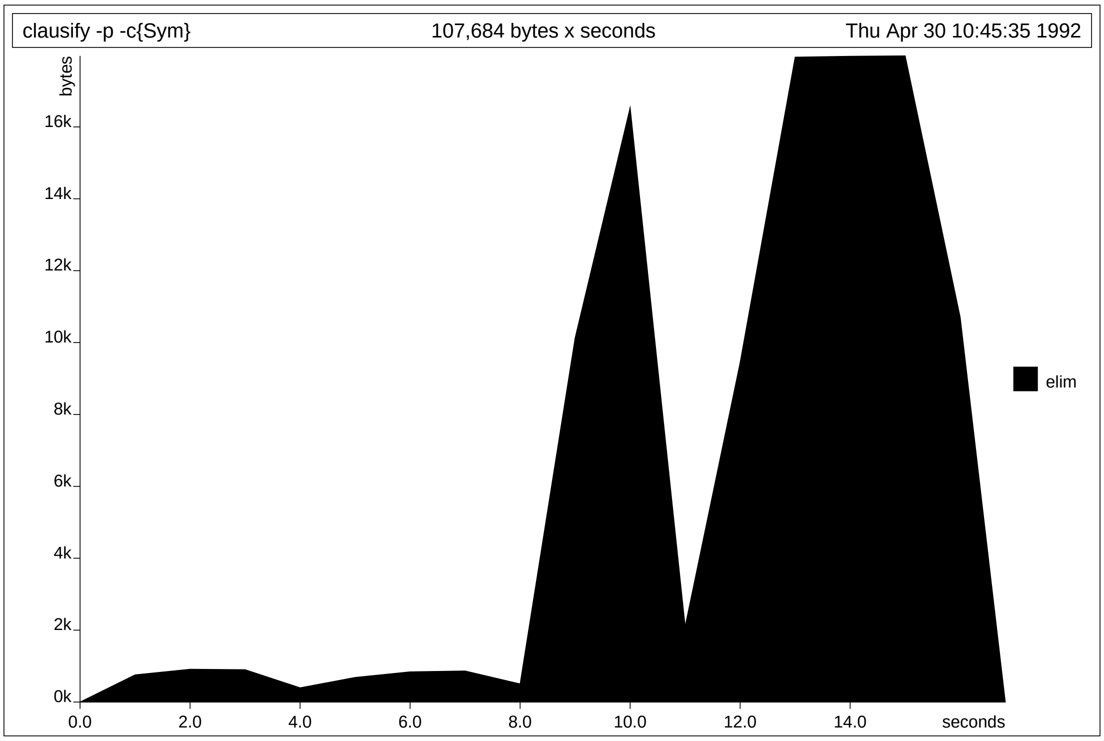

Figure 10: A producer profile for version 2 (producers of `Sym` constructions only)

Version 3

The desired effect can be expressed by rewriting the `elim` function, replacing the initial `Sym` clause with a final default clause:

```
elim (Not p) = Not (elim p)
elim (Dis p q) = Dis (elim p) (elim q)
elim (Con p q) = Con (elim p) (elim q)
elim (Imp p q) = Dis (Not (elim p)) (elim q)
elim (Eqv p q) = Con (elim (Imp p q)) (elim (Imp q p))
elim p = p
```

The bad news is that this change in the definition of `elim` makes no difference whatsoever! The good news is that this failed experiment points out the real problem and leads us to a solution.

There are two choices that can be made when updating the root cell of a redex with a cell that was not constructed during the reduction. The first is to update the root of the redex with a *copy* of the root of the result, and the second is to update with an *indirection* to the result. The advantages of each choice are set out in Peyton Jones' book [14]. Since our improved definition of `elim` makes no difference, we can conclude that our compiler's G-machine must update by copying. Altering the update mechanism to use indirections instead gives the heap profile shown in Figure 11. Although the overall profile looks very similar to that of the previous version, note the change of

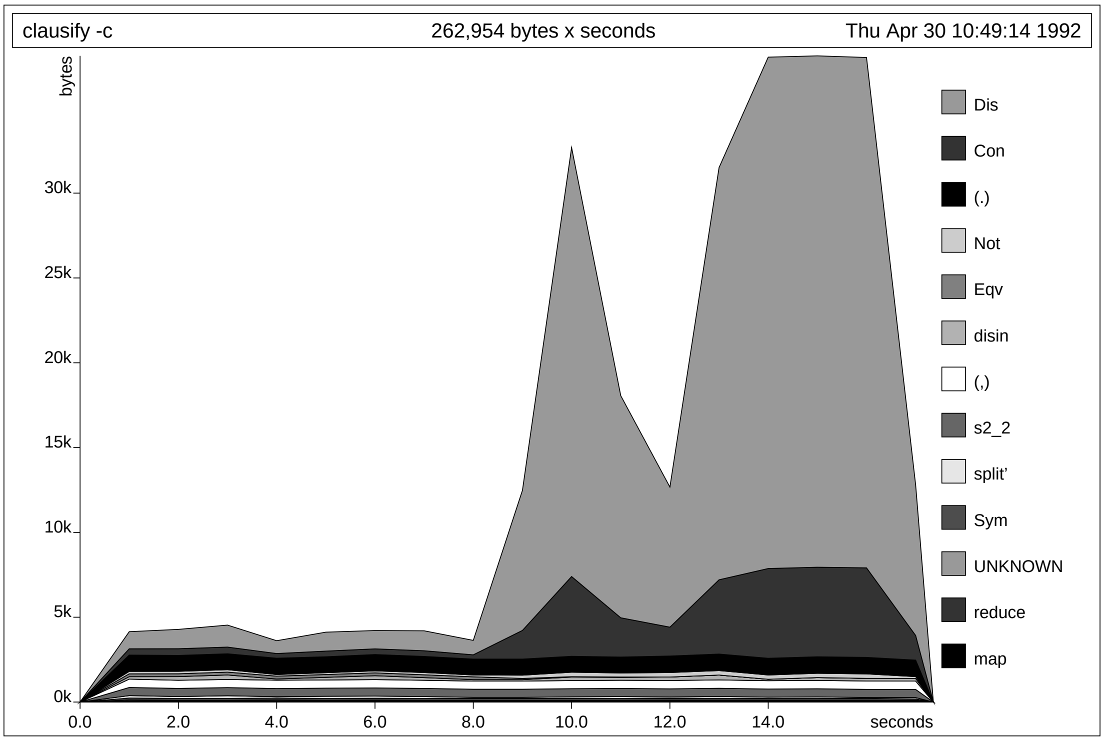

Figure 11: A construction profile for version 3

scale. The number of `Sym` constructions has been greatly reduced, as has the number of `Dis`, `Not` and `Con` constructions. This is because the `disin` and `negin` functions benefit from the use of indirections in exactly the same way that the `elim` function does — their original definitions *already had* final clauses of the form `disin p = p` and `negin p = p`. Notice that indirection nodes do *not* appear in a heap profile; they are, after all, only placeholders that will be elided at the next garbage collection.

The problem now is clearly the large number of `Dis` constructions in the graph. Once again, the producer of these constructions can be discovered by requesting a profile for just the producers of `Dis` constructions (see Figure 12). Recall that `disin` is responsible for distributing disjunction over conjunction, assuming that equivalences and implications have already been eliminated and that any negations apply to elementary proposition symbols only. Here is the way it is defined in version 3.

```
disin (Dis p (Con q r)) = Con (disin (Dis p q)) (disin (Dis p r))
disin (Dis (Con p q) r) = Con (disin (Dis p r)) (disin (Dis q r))
disin (Dis p q) =
      let dp = disin p in
      let dq = disin q in
      if conjunct dp | conjunct dq then
            disin (Dis dp dq)
      else
            (Dis dp dq)
disin (Con p q) = Con (disin p) (disin q)
disin p = p
```

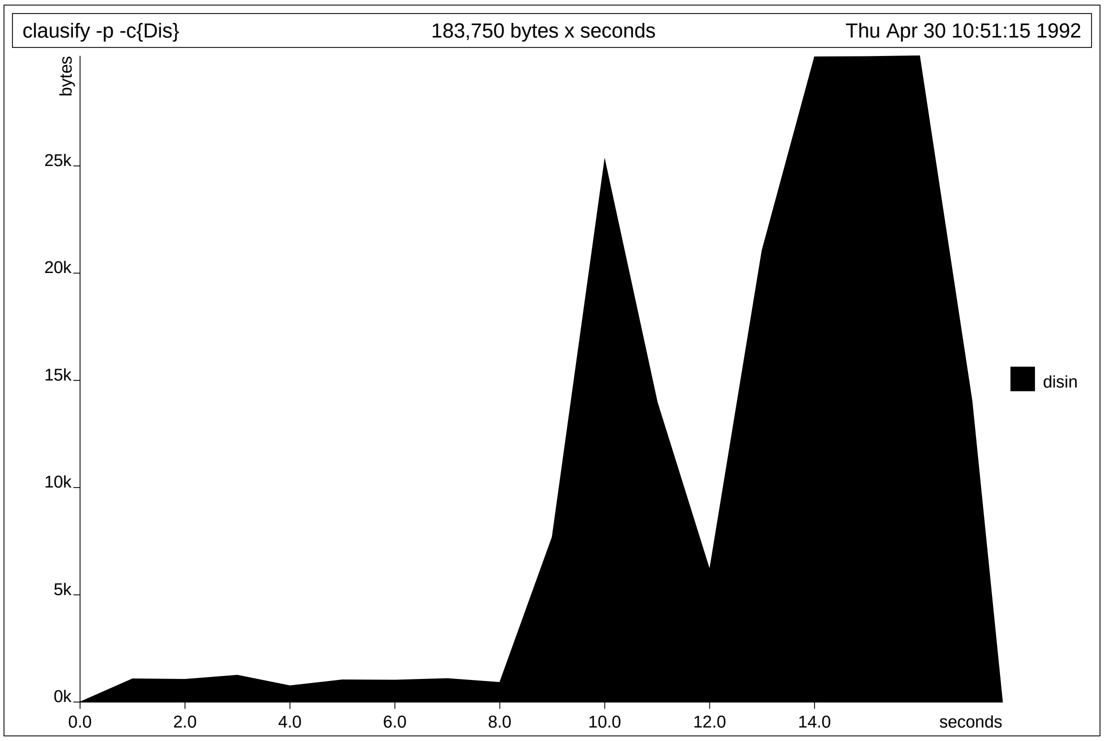

Figure 12: A producer profile for version 3 (producers of `Dis` constructions only)

The potential for exponential growth is evident in the first two clauses each of which "doubles" a disjunction. Looking at the right hand sides in the full sequence of defining clauses, there are no less than six occurrences of `Dis` constructions in all. An opportunity for improvement presents itself in a notable feature common to all but the last of these: they appear as arguments to recursive `disin` calls.

Version 4

We therefore introduce an auxiliary `disin'` corresponding to this composition, replacing explicit `Dis` constructions with an implicit relationship of disjunction between `disin'` arguments. Further, we note that in one of the original occurrences, the arguments are in conjunctive normal form. In order that this can be exploited by `disin'` we make it a uniform assumption in the revised definition as follows.

```
disin (Con p q) = Con (disin p) (disin q)
disin (Dis p q) = disin' (disin p) (disin q)
disin p = p

disin' (Con p q) r = Con (disin' p r) (disin' q r)
disin' p (Con q r) = Con (disin' p q) (disin' p r)
disin' p q = Dis p q
```

The outcome of this reformulation is very satisfying. Figure 13 shows that the space needed for the maximum population of `Dis` constructions has been reduced to a mere 10% of its former size, from 30kb to 3kb. In addition, the new formulation of `disin`

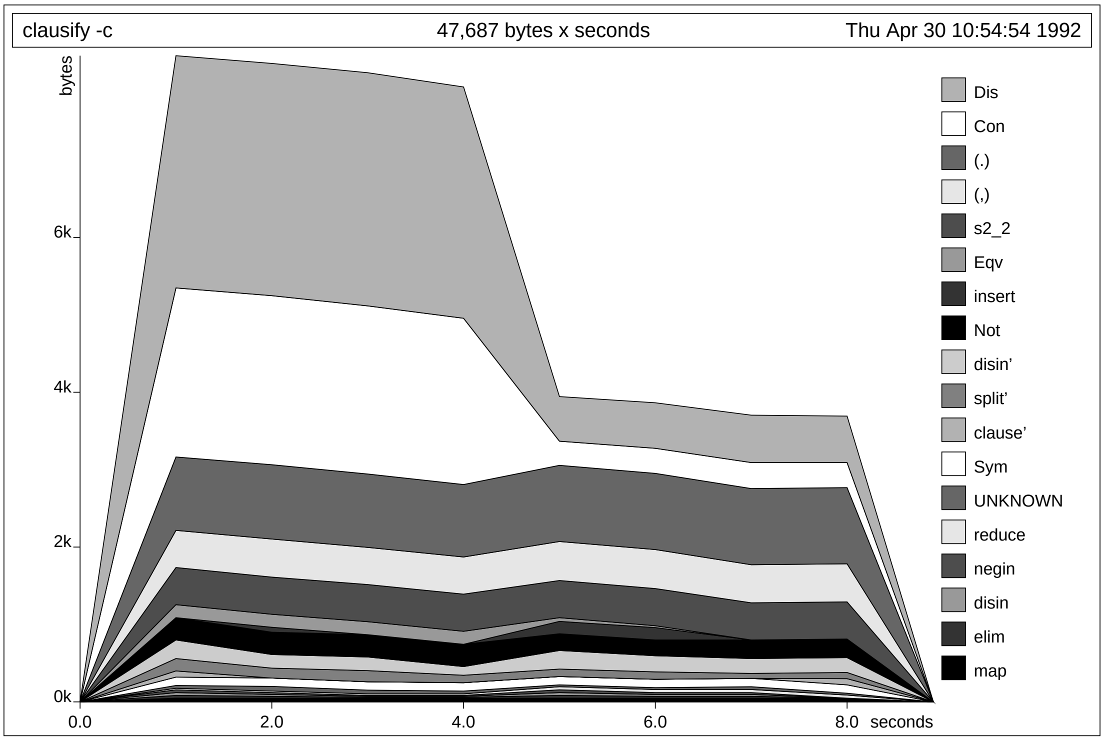

Figure 13: A construction profile for version 4

significantly decreases the overall run-time. Although the aim was only to reduce the volume of `Dis` constructions, the pursuit of this aim has also led to an improved algorithm.

However, the corresponding producer profile suggests that there is still further room for improvement (See Figure 14). The worrying thing here is that throughout the computation, the graph contains an almost constant number of nodes produced by the `splitat` function. In the clausify program, `splitat` is used only to split the list of characters representing a propositional formula into lines prior to parsing. Thus, we would not expect nodes produced by `splitat` to be retained in the graph once the benchmark problem has been parsed.

At first glance, the definition of `splitat` in the LML standard library seems harmless enough.

```
splitat c [] = ([], [])
splitat c (a.b) =
      if a = c then
            ([], b)
      else
            let (x, y) = splitat c b in (a.x, y)
```

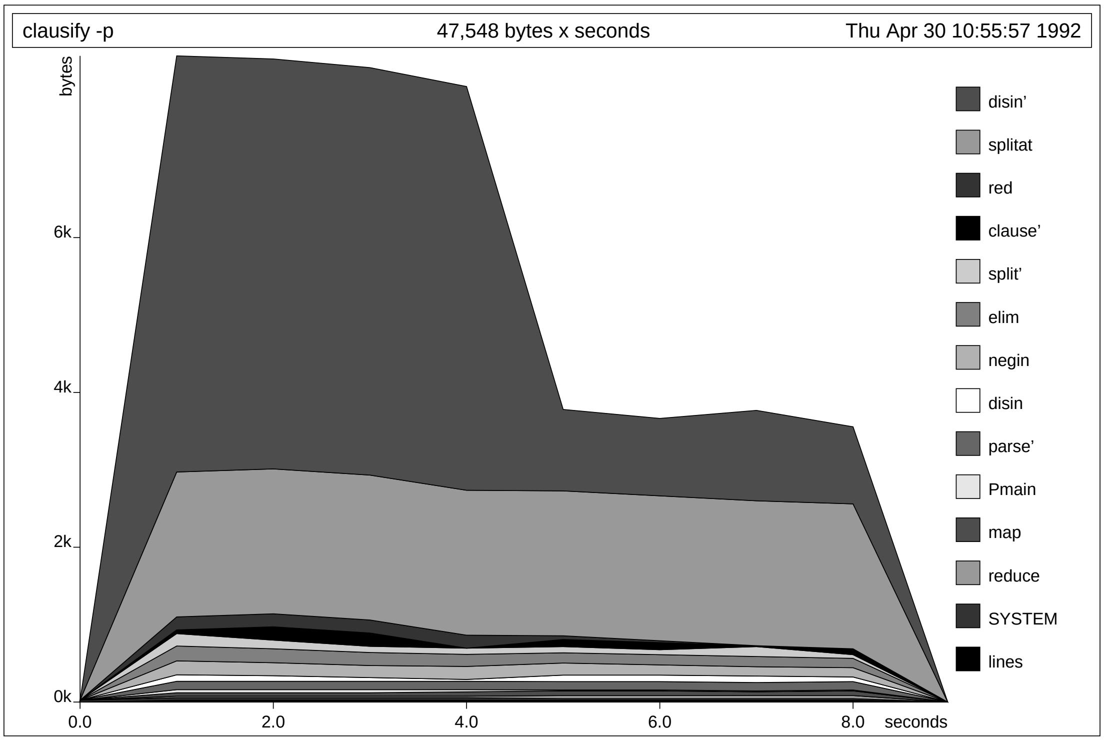

Figure 14: A producer profile for version 4

However, we know from Hughes' thesis [6] that all possible definitions of `splitat` are subject to a space leak if an ordinary sequential evaluator[†] is used. The problem is easily explained. In LML, and in most other lazy functional languages, a pattern on the left-hand side of a local definition is matched *lazily*. In other words,

$$\texttt{let } (p, q) \texttt{ = } r \texttt{ in } s\ p\ q$$

is treated as being equivalent to

$$\texttt{let } t \texttt{ = } r \texttt{ in } s \texttt{ (fst } t\texttt{) (snd } t\texttt{)}$$

where t is a new variable and `fst` and `snd` are the usual selection operations on pairs. One consequence of this is that the garbage collector will preserve the storage for p and q until *both* have been demanded by the computation. This is because whichever of p and q is discarded first, the other will cause its storage to remain accessible via a pending selection operation. In the clausify program, this space leak causes the entire input preceding the first newline to be unnecessarily preserved until the end of the computation.

[†]Hughes defines *a sequential evaluator* as one that, once it has begun to reduce an expression e, will only reduce e and other expressions that e demands until e has been completely reduced.

Version 5

Fortunately, the problem with `splitat` is easily solved. The idea, described by Wadler in [24], is to modify the garbage collector to perform reductions of the form

whenever it encounters an application of either a `fst` or a `snd` selector to a fully-evaluated argument. Modifying our G-machine's garbage collector in this way further reduces the cost of running the clausify program (see Figure 15).

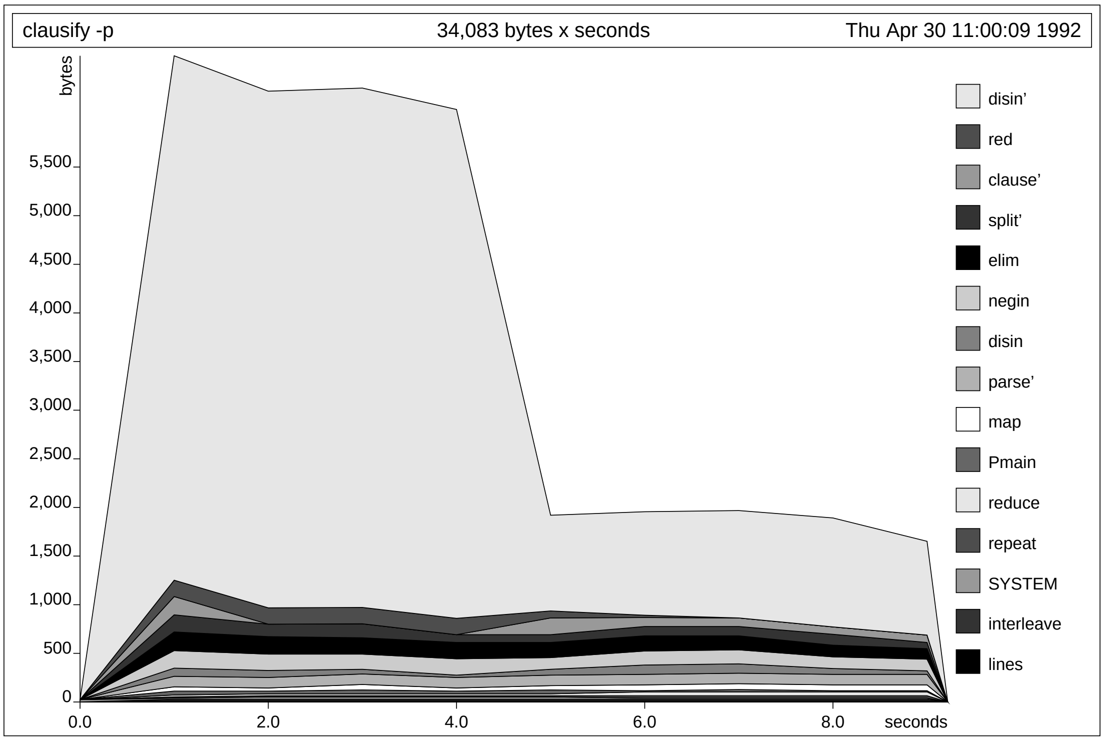

Figure 15: A producer profile for version 5

6 Discussion

Of course, the profile of version 5 suggests where to look for still further improvements. Perhaps the space cost of the `disin'` function could be reduced — although it is not immediately clear how this might be done. For now, let us be content with what we have achieved: we have *reduced the peak memory demand by more than two orders of* *magnitude*, from the original 1.3Mb to the present 7kb. Moreover, saving space has proved to be an extremely effective way of saving time, as version 5 is *almost twice as fast* as version 0. The ratio of their overall costs is no less than 350:1.

Summary and Assessment of the Method

In the course of working on the clausify program, we established a basic method of using the profiling tool, which can be summarised as repeated application of the following procedure.

1. Obtain a complete construction profile (-c option) and a complete producer profile (-p option).
2. Identify the major users of memory (constructions, producers or both) and obtain a product profile restricted to these.
3. Identify the producer/construction combinations demanding the largest share of memory.
4. Investigate the definitions of the relevant producers in this light. Try to reformulate at least one of these definitions in a way that reduces the demand for one or more of the critical constructions.

The iteration continues (or backtracks to consider alternatives) until a version of the program is obtained for which the memory peaks either fall within a target figure or else seem unavoidable given the extensional specification of the program.

Clausify was the first program to which we applied this method, but it has since been applied to many others. In our experience, steps 1, 2 and 3 above do successfully isolate critical definitions for investigation. One measure of the value of such information is the degree to which it is surprising: by this yardstick we have found producer-construction pairings delivered by our profiler extremely valuable on several occasions!

The real pay-off depends on step 4. The most striking thing is that typical gains from a successful reformulation are not just small percentages but factors of 2, 3 or even more — it does not take many such factors, multiplied together, to yield an overall factor in the hundreds. But how easy is it to determine the appropriate reformulation, even once the key definitions and constructions are known? Here our experience has varied. In some cases, such as the `Sym`-copying definition of `elim`, the problem is immediately apparent and a suitable reformulation is the work of a moment; but in others, such as the production of `Dis` constructions by `disin`, the appropriate reformulation is much less obvious and its effectiveness is less certain in advance of further tests.

Inevitably, it becomes harder and harder to find significant space-savings as the refinement process goes on. So, when should one stop? Currently we have no easy way to establish whether some observed part of the memory demands of a program is "unavoidable" or not — at the time of writing this is still an open question for the remaining plateau of version 5. However, if space consumption can be made "acceptably low" before the reformulation problem becomes intractable, the difficulty of distinguishing between avoidable and unavoidable space demands might not matter too much in practice: in comparison with an original figure of 1.3 megabytes, a difference of a few kilobytes is not great.

Implementation and/or Application Profiling

Profiling experiments of the kind we have described inevitably examine a particular *combination* of at least four elements:

1. an application program;
2. a compiler (and run-time system);
3. a set of input data;
4. a target machine.

Although space-efficient programming in a lazy functional language can present subtle difficulties whatever the implementation, several years' emphasis on compilation techniques to improve the *speed* of functional computation has resulted in a comparative disregard for even quite major space problems. So the emergence from the "clausify in LML" profiling exercise of both program *and* compiler problems was no surprise — though we never imagined at the outset quite how much mileage we would get from this particular combination of a well-tried program and a well-tried compiler.

We hope that profiling tools such as ours will serve to increase "space awareness" among compiler writers, so that space problems attributable to the implementation will decrease. This should correspondingly enhance the value of profiling as a tool for isolating and fixing space problems in programs.

So far as dependence on input data is concerned, it is true that our benchmark problem involving nested equivalence chains over a single propositional symbol makes extreme demands on the clausify program. The space savings are less dramatic when the input comprises a more typical mixture of propositions (for example, the exercises from chapter 1 of [12]), but they are still impressive — between one and two orders of magnitude rather than between two and three.

7 Subsequent Developments

Following our success with clausify, we extended the profiler to make it more suitable for dealing with large programs. The producers were extended to whole modules or groups of modules, and the constructions were extended to whole types. At the same time, we reduced the cost of profiling by performing a garbage collection while sampling the graph. This idea, suggested to us by Peyton Jones, makes heap profiling *intrusive* because the extra sampling garbage collections alter the program's normal pattern of garbage collection. However, we consider it to be a sensible compromise between virtue and efficiency. We have reported elsewhere [17] the successful application of this extended profiler to the LML compiler, including the discovery and correction of a long-standing space fault.

Our profiling system has subsequently been distributed to a large number of other users (and it is now available as part of the LML/HBC compiler distribution from Chalmers University). To give just two examples of various successful applications known to us, Hughes has located and fixed a space fault in his CDS loop-detecting interpreter [7], cutting the maximum heap size when interpretting a simple but highly recursive program from 1.8Mb to 30kb. He found that heap profiling was essential, although the job was not easy. Kozato has also used heap profiling to guide the reformulation of a lazy image processing system [11], so that the final version runs in constant space regardless of the size of the image.

8 Related Work

Not much work seems to have been done on providing profiling facilities for lazy functional languages.

We have already noted that other heap attributes, apart from statically determined properties of individual cells, may be of interest to functional programmers. Hartel and Veen [5], for example, studied the life-times of cells and the lengths of application chains. Indeed, one could profile all kinds of structural properties of the heap, but it is not obvious which properties to choose or what structural information would best serve the aim of reducing computational costs.

In a previous paper [16] we made a proposal for profiling the storage consumption of lazy functional programs. Essentially, our idea was to construct an interpretive profiler based on *source level graph reduction*. During the computation this interpreter would collect information about the production and consumption of every node as the program graph was reduced. This information could then be presented to the programmer in the form of a *producer-consumer matrix*. By studying the matrix, we hoped that the programmer could see how to modify the program so as to reduce either the maximum or the average size of the graph.

Sansom [20] suggests a scheme somewhat different from our own, the central idea being that the programmer should nominate *cost centres* to which the cost of evaluating selected expressions should be attributed. Cost centres are attached to expressions through applications of the primitive function `setCostCentre`. The only costs of evaluating an expression that are not attributed to its cost centre are the cost of evaluating any free variables, and the cost of evaluating any subexpressions for which another cost centre has been nominated. Sansom describes a possible implementation of his cost centre model based on Peyton Jones' Spineless Tagless G-machine [15]. The machine is modified by adding a new `CurrentCostCentre` register, and by arranging for every cell in the heap to be tagged with the value of this register when it is allocated. The `setCostCentre` primitive saves the value of the `CurrentCostCentre` register and sets it to a new value. When evaluation of the expression to which the new cost centre is attached is complete, the previous value of the `CurrentCostRegister` is restored. Heap profiling can then be accomplished by using the cost centre tag attached to each cell and a sampling scheme similar to the one that we have described. Sansom's work is still in the early stages of development, and it will be interesting to see how his approach compares in practice with our own.

9 Future Work

In the clausify example, to diagnose the first problem with `unicl` we had to use our knowledge of the pipeline structure of the program, and also an observation about its output behaviour. This gave us some information about graph *consumption* to put alongside the *production* profile. Our earlier view [16] that consumer profiling could also be of value is thus confirmed. Consumer profiling might also help with another problem we encountered: when `map` (say) produces an unexpectedly large number of nodes, how do we know which particular instance of `map` is responsible? It was somewhat fortuitous that when this problem arose, it was easy to rule out all but one. Currently, we have no way to distinguish between nodes that were produced by different instances of a given function: the best that we can do is to rename a suspect instance and generate an appropriately renamed copy of its code. Consumer profiling would also have been useful to Hughes in the analysis of his interpreter: once he had construction and producer information about an anomalously large heap component, his main problem was finding out *why* this component was being retained.

The basic sampling scheme could be further improved by replacing the test of the sample flag at the start of each function with a *sampling interrupt* similar to the timer interrupt used in the implementation of SML/NJ [1]. When a sample is due, the signal handler simply sets the heap limit register to zero. The next attempt to allocate space from the heap is then certain to call the garbage collector. The garbage collector examines the heap limit register and decides whether to perform an ordinary garbage collection or one with sampling.

One might imagine using more sophisticated graphical presentation techniques which take advantage of the rapidly falling cost of colour workstations and colour laser printers. However, we believe that the effort devoted to producing colour output would be largely wasted. In his book, Tufte [22] explains that the use of colour often results in "graphical puzzles" which are actually harder to understand than if shades of grey had been used instead. Van der Poel's motto puts it another way:

> What cannot be made clear by white chalk alone cannot be made clear by coloured chalk either [23].

So, although we shall continue to improve our graphs with reference to Tufte's guidelines, they will still be printed with shades of grey.

Our current methods of examining the graphical profiles, and interpreting them in the light of the way functions are defined, are entirely *ad hoc*. However, certain characteristic shapes recur in heap *strata*, and with experience we hope that it will be possible to give a systematic account of the most frequently occurring phenomena and their most likely causes, leading to improved techniques for reformulation. We also hope that the investigation of profile graphs will yield fresh insights of a more general nature, that might be useful in an eventual theory of lazy evaluation costs.

10 Summary and Conclusions

We have described the design and implementation of a tool for profiling the space consumption of lazy functional programs. Heap profiling gathers information about the live heap contents throughout a computation and presents it in a graphical, source-related form. It can be used to bring about dramatic reductions in the space consumption of functional computations, as illustrated by the clausify example in this paper, and subsequently confirmed in a wide variety of other applications.

Our experience suggests that space faults are not rare but common. Full remedies sometimes require not only modifications to the source program, but also changes to the implementation. But the information supplied by heap profiling is equally valuable in either case.

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. Charles Forsyth explained the vagaries of the UNIX interval timer to us, and Paul Sanders wrote the "compress" program that gave rise to the exotic graph shown in Figure 1. We are grateful to the referees for their comments.

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

An earlier version of this paper was issued as a technical report [18].

A The Clausify Program

This appendix contains the text of the original clausify program before modification.

```
let rec
--  abstract syntax for propositional formulae
    type Formula = Sym Char
                 + Not Formula
                 + Dis Formula Formula
                 + Con Formula Formula
                 + Imp Formula Formula
                 + Eqv Formula Formula
```

```
--  entries on stack used by proposition parser
and type StackFrame = Ast Formula + Lex Char


--  separate positive and negative literals, eliminating duplicates
and clause p =
    let rec clause' (Dis p q)       x   = clause' p (clause' q x)
         || clause' (Sym s)       (c,a) = (insert s c, a)
         || clause' (Not (Sym s)) (c,a) = (c, insert s a)
    in
         clause' p ([], [])


--  the pipeline from propositional formulae to printed clauses
and clauses = concat o
              map disp o
                   unicl o
                     split o
                       disin o
                         negin o
                            elim o
                              parse

--  the main program: simple line-based interaction
and clausify = concat (interleave (repeat "prop> ")
                      (map clauses (lines input)))


--  concatenation of a list of lists
and concat = foldr (@) []

--  test for conjunctive proposition
and conjunct (Con p q) = true
||  conjunct p = false


--  shift disjunction within conjunction
and disin (Dis p (Con q r)) = Con (disin (Dis p q)) (disin (Dis p r))
||  disin (Dis (Con p q) r) = Con (disin (Dis p r)) (disin (Dis q r))
||  disin (Dis p q) =
        let dp = disin p in
        let dq = disin q in
        if conjunct dp | conjunct dq then
            disin (Dis dp dq)
        else
            (Dis dp dq)
||  disin (Con p q) = Con (disin p) (disin q)
||  disin p = p
```

```
--  format pair of lists of propositional symbols as clausal axiom
and disp (l,r) = interleave l spaces @ "<=" @ interleave spaces r @ "\n"


--  eliminate connectives other than not, disjunction and conjunction
and elim (Sym s) = Sym s
||  elim (Not p) = Not (elim p)
||  elim (Dis p q) = Dis (elim p) (elim q)
||  elim (Con p q) = Con (elim p) (elim q)
||  elim (Imp p q) = Dis (Not (elim p)) (elim q)
||  elim (Eqv p q) = Con (elim (Imp p q)) (elim (Imp q p))


--  reduce familiarly renamed
and foldr = reduce

--  insertion of an item into an ordered list
and insert x [] = [x]
||  insert x (p as (y.ys)) =
        if x < y then x . p
        else if x > y then y . insert x ys
             else p


--  alternation of items from two lists until one is exhausted
and interleave (x.xs) ys = x . interleave ys xs
||  interleave []     _  = []


--  list of lines from given text
and lines [] = []
||  lines xs = let (l,r) = splitat '\n' xs in l . lines r


--  shift negation to innermost positions
and negin (Not (Not p)) = negin p
||  negin (Not (Con p q)) = Dis (negin (Not p)) (negin (Not q))
||  negin (Not (Dis p q)) = Con (negin (Not p)) (negin (Not q))
||  negin (Dis p q) = Dis (negin p) (negin q)
||  negin (Con p q) = Con (negin p) (negin q)
||  negin p = p


--  the priorities of symbols during parsing
and opri '(' = 0
||  opri '=' = 1
||  opri '>' = 2
||  opri '|' = 3
||  opri '&' = 4
||  opri '~' = 5
```

```
--  parsing a propositional formula
and parse t = let [Ast f] = parse' t [] in f

--  parsing auxiliary - extra argument is stack
and parse' [] s = redstar s
||  parse' (' '.t) s = parse' t s
||  parse' ('('.t) s = parse' t (Lex '(' . s)
||  parse' (')'.t) s =
        let (x . Lex '(' . s') = redstar s in
        parse' t (x.s')
||  parse' (c.t) s =
        if islower c then parse' t (Ast (Sym c) . s)
        else if spri s > opri c then parse' (c.t) (red s)
             else parse' t (Lex c . s)


--  reduction of the parse stack
and red (Ast p . Lex '=' . Ast q . s) = Ast (Eqv q p) . s
||  red (Ast p . Lex '>' . Ast q . s) = Ast (Imp q p) . s
||  red (Ast p . Lex '|' . Ast q . s) = Ast (Dis q p) . s
||  red (Ast p . Lex '&' . Ast q . s) = Ast (Con q p) . s
||  red (Ast p . Lex '~' . s) = Ast (Not p) . s


--  iterative reduction of the parse stack
and redstar = while ((~=) 0 o spri) red

--  infinite list of identical items
and repeat x = let rec xs = x . xs in xs

--  an infinite list of spaces
and spaces = repeat ' '


--  split conjunctive proposition into a list of conjuncts
and split p =
        let rec split' (Con p q) a = split' p (split' q a)
             || split' p a = p . a
        in
             split' p []


--  priority of the parse stack
and spri (Ast x . Lex c . s) = opri c
||  spri s = 0


--  does any symbol appear in both consequent and antecedent of clause
and tautclause (c,a) = intersect c a ~= []
```

```
--  form set of unique non-tautolous clauses given list of conjuncts
and unicl a =
        let unicl' p x =
            (if tautclause cp then x else insert cp x
            where cp = clause p)
        in
            foldr unicl' [] a

--  higher order functional counterpart to the 'while loop'
and while p f x = if p x then while p f (f x) else x


in
    clausify
```

References

[1] A. W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

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

[3] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *Computer Journal*, 32(2):127–141, April 1989.

[4] B. Bjerner and S. Holmstrom. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, September 1989.

[5] P. H. Hartel and A. H. Veen. Statistics on Graph Reduction of SASL Programs. *Software — Practice and Experience*, 18:239–253, 1988.

[6] R. J. M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Oxford University, September 1984.

[7] R. J. M. Hughes. A Loop-Detecting Interpreter for Lazy, Higher-Order Programs. In *Proceedings of the 5th Glasgow Workshop on Functional Programming*, July 1992. to be published by Springer.

[8] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, September 1985. LNCS 201.

[9] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, S-412 96 Göteborg, February 1987.

[10] R. Jones. Tail Recursion Without Space Leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[11] Y. Kozato and G. P. Otto. Geometric Transformations in a Lazy Functional Language. In *Proceedings of 11th International Conference on Pattern Recognition*, August 1992.

[12] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming (Volume 1: Deductive Reasoning)*. Adddison-Wesley, 1985.

[13] S. L. Meira. *On the Efficiency of Applicative Algorithms*. PhD thesis, University of Kent at Canterbury, March 1985.

[14] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[15] S. L. Peyton Jones. Implementation of Functional Languages on Stock Hardware: The Spineless Tagless G-machine. Technical report, Department of Computing Science, University of Glasgow, February 1992.

[16] C. Runciman and D. Wakeling. Problems and Proposals for Time and Space Profiling of Functional Programs. In *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, pages 237–245. Springer-Verlag, August 1990.

[17] C. Runciman and D. Wakeling. Heap Profiling of a Lazy Functional Compiler. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer-Verlag, August 1992. In press.

[18] C. Runciman and D. Wakeling. Heap Profiling of Lazy Functional Programs. Technical Report 172, Department of Computer Science, University of York, April 1992.

[19] D. Sands. Time Analysis, Cost Equivalence and Program Refinement. In *Proceedings of the Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 25–39. Springer-Verlag, December 1991. LNCS 560.

[20] P. Sansom. Profiling Lazy Functional Languages. Draft Memorandum, Department of Computing Science, University of Glasgow, January 1992.

[21] W. Stoye. *The Implementation of Functional Languages Using Custom Hardware*. PhD thesis, University of Cambridge Computer Laboratory, December 1986. Technical Report No. 81.

[22] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, 1983.

[23] W. M. van der Poel. The Mechanization of the Lambda Calculus. Undated typescript, University of Technology, Delft, The Netherlands.

[24] P. Wadler. Fixing Some Space Leaks with a Garbage Collector. *Software — Practice and Experience*, 17(9):595–608, September 1987.

[25] P. Wadler. Strictness Analysis Aids Time Analysis. In *Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 119–131, January 1988.

[26] S. C. Wray. *Implementation and Programming Techniques for Functional Languages*. PhD thesis, University of Cambridge Computer Laboratory, January 1986. Technical Report No. 92.