New dimensions in heap profiling

Colin Runciman
Dept. of Computer Science
University of York
York YO1 5DD
England
colin@minster.york.ac.uk

Niklas Röjemo
Dept. of Computer Science
Chalmers University
S-412 96 Gothenburg
Sweden
rojemo@cs.chalmers.se

July 1994 (revised February 1995)

Abstract

First-generation heap profilers for lazy functional languages have proved to be effective tools for locating some kinds of space faults, but in other cases they cannot provide sufficient information to solve the problem. This paper describes the design, implementation and use of a new profiler that goes beyond the two-dimensional 'who produces what' view of heap cells to provide information about their more dynamic and structural attributes. Specifically, the new profiler can distinguish between cells according to their *eventual lifetime*, or on the basis of the *closure retainers* by virtue of which they remain part of the live heap. A bootstrapping Haskell compiler (`nhc`) hosts the implementation: among examples of the profiler's use we include self-application to `nhc`. Another example is the original heap-profiling case study `clausify`, which now consumes even less memory and is much faster.

1 Introduction

Purely functional languages are attractive for their concise uncluttered style — a style made possible by freeing the programmer from many responsibilities such as memory management and evaluation strategy. There is a drawback however. Lazy functional programs that look concise and elegant as collections of equations can make surprisingly large demands on machine resources. The programmer sees only recursion equations, but the machine performs normal order graph reduction using those equations as rules. So it may be easy to reason about the results obtained, yet hard to reason about the costs of obtaining them.

1.1 Heap profiling

In science, *observation* plays a central role in the advancing of knowledge. Careful study of accurate observations often precedes the formulation of valuable new ideas. But observation usually requires *instruments*, and the availability or lack of suitable instruments in some field of enquiry often proves critical. A *heap profiler* [RW93b, RW93a] is an instrument for observing lazy functional computation. The observations made possible by heap-profiling help programmers to analyse and hence reduce the *memory use* of lazy functional programs evaluated by graph reduction.

Here is how the original implementation of heap-profiling works. A modified compiler generates additional code to *label* every graph node. The labels of a node indicate its *construction* (what it represents, in terms of symbols occurring in the source program) and its *producer* (which part of the program caused it to be introduced). At specified regular intervals, a *census* of the live heap is taken: the run-time system traverses the entire live graph structure collecting and storing label data. Once the program has finished running, the collected data is processed to generate charts such as those in Figure 13 of §5.1. Shaded bands, in the area beneath a 'curve' plotting live-heap size against time, represent partitions of the heap population. There are two sorts of profile: in one, each band represents a different producer; in the other, bands represent constructions. To focus on specific aspects of memory consumption, the user can narrow the scope of profiling to a specified set of constructions or producers.

Although the basic idea of heap-profiling is simple, it has often proved effective. In many successful applications, iterative profiling and improvement has achieved substantial gains in efficiency. The original case-study was a propositional simplifier [RW93b] for which the gains were better than two orders of magnitude. (This example application is revisited in §5.1.) With constructions aggregated by type, and producers by module or source file, heap-profiling has also been successfully applied to large and complex programs [RW93a, RE95].

1.2 New dimensions

A two-dimensional characterisation of a heap by such simple attributes of its cells — producer and construction — clearly has its limitations, notably of two kinds:

- The heap cell attributes involved are *static*. Labels attached to a cell *remain fixed* throughout the time it is part of the live graph. This has the advantage that labels can be fully determined at compile-time for each static point in the program at which a cell is allocated, but it denies the programmer any information about dynamic attributes of cells.

- *No structural information* about the access relation of the graph is recorded. For each heap census, graph traversal serves only to reach the full population of live cells. Given such 'flat' profile data for a particular program, it may be possible for a programmer to *infer* some structural information: indeed this may be essential before space-saving modifications can be made. But the information directly presented simply partitions a set of otherwise unrelated cells.

These limitations in principle are reflected in practice. Heap-profiling of the 'who produces what' kind cannot answer all the questions programmers want to ask. Here are two examples:

- Suppose a profile shows a 'plateau' of sustained high demand for heap space at some stage in a computation. Does this represent 'dragging' of an *unchanging* subgraph (perhaps remote from the sites of reduction), or is it a phase of substantial graph *replacement* in which there happens to be a fairly constant turn-over of cells? The distinction could be important, as 'dragging' is a common space-fault that can often be corrected. But to make this distinction requires information derived from a run-time attribute of cells — their time of creation.

- Suppose a profile includes a wide band representing 'cons' cells produced by the `append` function. Such a band *might* be surprising to a programmer, and give new insights into the workings of a program. But it might be a familiar sight to a programmer working on text-based applications — who is never quite sure whether this band could somehow be made smaller! What the programmer would *really* like to know is *why* these cells are retained in the heap? Which part of the program is 'holding onto them'? This is a *structural* question to do with access paths in the graph.

This paper therefore describes an extension of heap-profiling for functional programs evaluated by graph reduction. The extension provides information about dynamic and structural aspects of a graph being reduced: *lifetime* profiling is described in §2; and *retainer profiling* in §3. A profiler with these 'new dimensions' has been implemented, and details of the implementation are given in §4. Example applications, illustrating what can be done with the new profiling information, are presented in §5. Some related work is discussed in §6, and §7 concludes.

2 Lifetime profiling

An easy first step towards supplying dynamic information about heap cells is to label each heap cell with the time at which it is created. But is *creation-time* the most helpful dynamic attribute to present in a profile? Knowing the time of each heap census, it is easy to infer the *age* of cells labelled with creation times. With

```
reverse xs = rev [] xs
  where
  rev r []     = r
  rev r (x:xs) = rev (x:r) xs

traverse xs = trav xs xs
  where
  trav t []     = t
  trav t (x:xs) = trav t xs
```

Figure 1: Two similar functions?

```
legion = [1..12000]

repeat f 0 x = x
repeat f n x = repeat f (n-1) (f x)

work f = repeat f 100 legion
```

Figure 2: A definition of `work`.

only a little more ingenuity we could use a post-processor to work out eventual cell *lifetimes*. Which of these is best, or doesn't it matter?

To answer this question, consider the functions `reverse` and `traverse` defined in Figure 1. Superficially, the two functions are very similar. Each recurses over a list argument and yields as its result a list of the same length. But the result of reverse is a *distinct new* list structure, whereas the result of `traverse` is the *same old* list. We shall see how this difference would be reflected in three different profiles of the heap cells representing the list structures; the computations assumed in each case are `work reverse` and `work traverse`, where the higher-order driving function `work` is defined as in Figure 2.

Creation-time profiling?

The *creation-time* profiles in Figure 3 band the heap according to the creation-times of cells. Creation-times are dynamic in the sense that they are known only at run-time rather than compile time, and vary across cell allocations from the same static program point. However, creation-time is a fixed attribute throughout the lifetime of a cell so each cell is consistently shaded across the profile. All the list cells in the `traverse` computation are created early, hence the uniform picture. But the `reverse` profile illustrates two problems with using creation-times.

- There is an in-built skew in the distribution of shadings, because early in the computation there cannot be any cells with a late time of creation. So

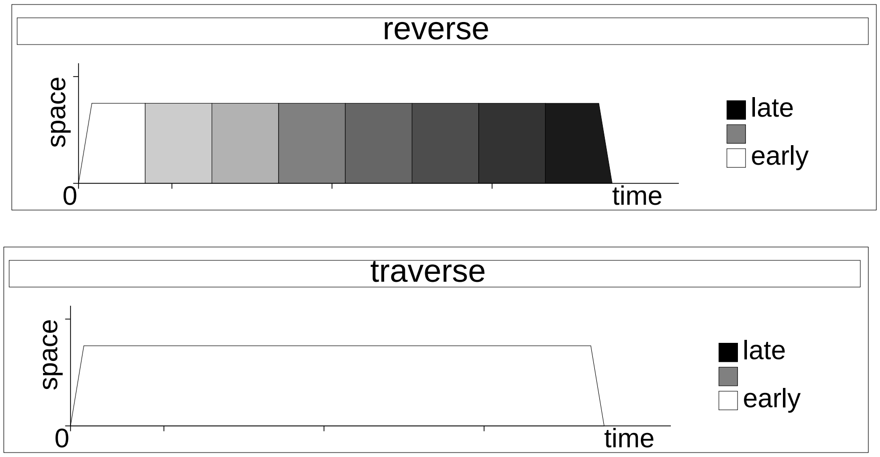

Figure 3: Profiling by cell creation-time.

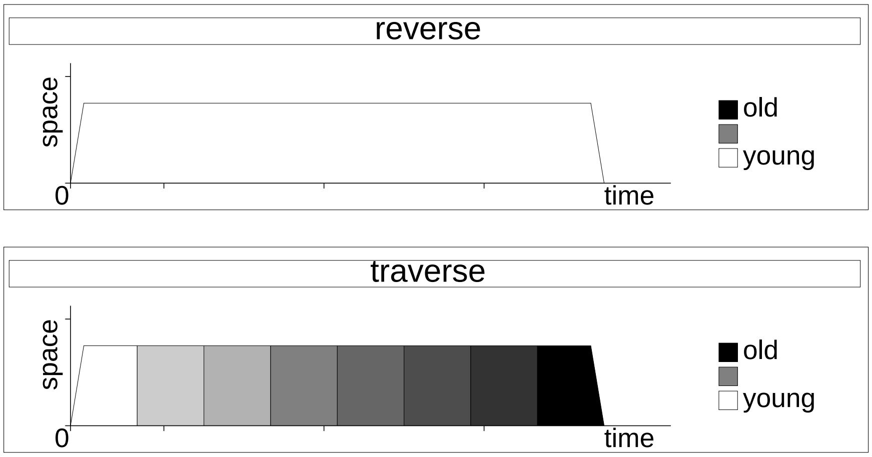

Figure 4: Profiling by cell age.

some profiles are impossible, and the information capacity of the shaded charts is under-used.

- Similar computations repeated at different stages may not be easily recognised, since their appearance may vary in the profile.

Age profiling?

The *age* profiles in Figure 4 band the heap according to the age of cells at each census. This also suffers from an inherently skewed distribution of shadings: early in the computation there cannot be any old cells. More seriously, when profiling by age:

- long-lived cells may change bands several times.

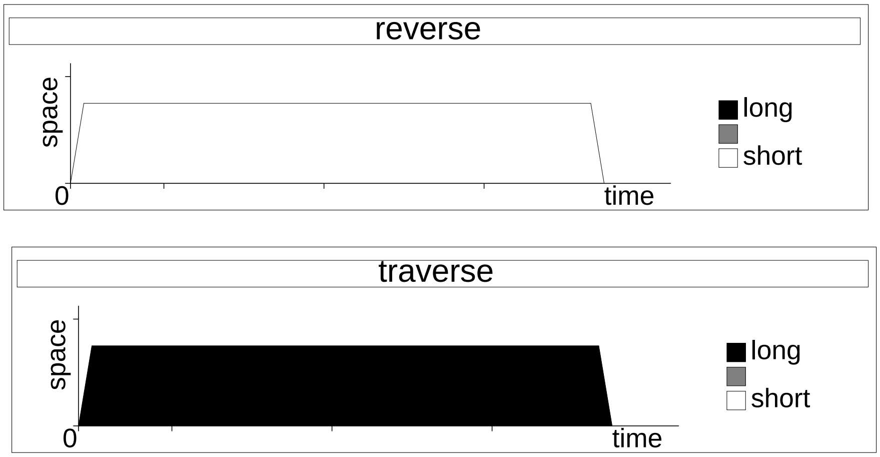

Figure 5: Profiling by cell lifetime.

This makes it tricky for the programmer to maintain a correct intuition when reasoning about a particular class of cells.

Lifetime profiling?

Lastly, consider the *lifetime* profiles in Figure 5. Here heap cells are banded according to their eventual full lifetime as part of the live graph. The distinction between the `reverse` and `traverse` computations is shown with black-and-white clarity. Whereas the data for creation-time and age profiles can be generated as a computation proceeds, lifetime profiling requires retrospective calculation to determine cell counts at each stage in the computation. However, lifetime profiling has significant advantages: there is no inherent skew in the distribution of shadings; cells are assigned a consistent shading throughout a computation; and any repeated similar computations are similarly shaded.

So it seems that lifetime profiles are the most attractive for the programmer, though also the most demanding for the implementor. Identifying any sizable population of *either* very long-lived *or* very short-lived cells may help to improve programs. For example, short-lived cells may be intermediate structures to which *deforestation* [Wad90] is applicable; long-lived cells might represent large unevaluated closures that can be reduced to small results before they are actually needed.

2.1 Post-processing for lifetimes

Let time in a profiled computation be measured as the number of heap censuses taken so far. We shall refer to all cells with the same time of creation as belonging to the same *generation*.

At each heap census, population counts for each generation can be recorded in a log-file. A post-processor is needed to derive lifetime information from this

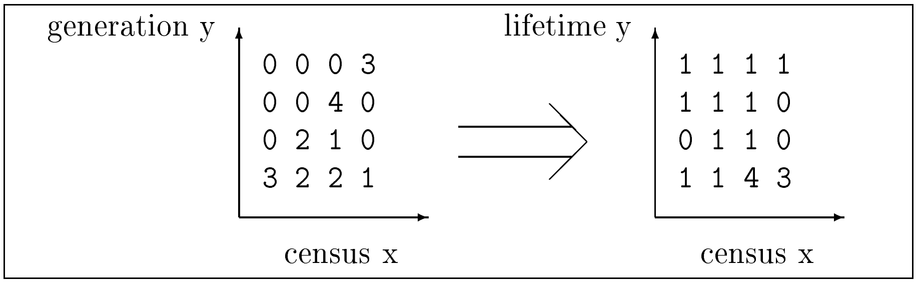

Figure 6: An example of lifetime data derived from creation-time data.

file.

Transforming creation-times to lifetimes

The problem is to take an *input matrix*, containing population counts for each generation at each census, and to compute from it an *output matrix*, containing counts for each *lifetime* at each census. Figure 6 shows an example of such input and output. In this example, just one cell created as part of the first generation survives to the final census: it is represented by the bottom-right element 1 in the creation-time census, and by the top row of 1's in the derived matrix of lifetime data.

Consider first a basic algorithm that uses two separate data structures to store the input and output matrices. The output matrix is initially filled with zeroes. The basis of calculation is that a cell created as part of generation g, not surviving to a census beyond $g+l$, has lifetime l. The algorithm works backwards through the census data in the input matrix. No cells survive to a census beyond the *last* one, recorded at time c_{last}. Each cell recorded at c_{last} belongs to some generation g, and has lifetime c_{last}-g. Population counts in the *output* matrix can be increased accordingly for censuses g upto c_{last}. This done, we subtract the contribution of the last surviving group of cells from the population counts of earlier censuses in the *input* matrix. We can then repeat the whole process for the penultimate census (since all cells that contribute to the remaining population counts for this census do not survive to the next). And so the calculation continues, working back through the census data, until only zeros are left in the input matrix. The final output matrix is just as if eventual cell lifetimes had been recorded in it at each census.

The same transformation can be done *in place* and *in linear time* (in the number of array elements). Suppose census information is read into a two dimensional array A:[0..N,0..N], so that initially the value of A[x,y] contains the number of yth generation cells recorded in the heap census at time x. Figures 7–9 give a three-phase algorithm for transforming A so that afterwards the value at A[x,y] is the number of cells alive at time y with lifetime x. (That is, the algorithm computes the *transpose* of lifetime matrices such as in Figure 6.) Figures 7–9 also show the intermediate values of `A` after each phase, assuming the initial census data of Figure 6.

2.2 Lifetimes as selectors

Experience with the orginal heap-profiler has shown the value of being able to examine particular *sections* of the heap, specified by cell characteristics. This principle should extend also to lifetimes so that, for example, the programmer can look at the constructions of long-lived cells.

When lifetime information is used to specify a selected population of cells, recorded census data must includes cell counts *for each generation for each separate construction* (or producer, etc). Similarly, the postprocessor must build and transform an array for each separate construction. As this may involve a large amount of data, the efforts to devise an efficient transformation algorithm are well spent.

3 Retainer profiling

An earlier paper [RW91] argued in principle that programmers may need to know not only about the *producers* of heap cells, but also about *consumers*. If we think of producers as *cell writers* then a cell's consumers are its *readers* — all functions that examine the cell as part of their computational work. Alternatively, if we regard producers as *cell allocators* then a cell's (unique) consumer is its *disposer* — the function whose computational work causes the cell to become detached from the live graph. Information about either one of these kinds of consumer could be useful to a programmer, but both are expensive to identify with accuracy.

There is a third view of production and consumption: producers build cells to represent new pieces of graph; consumers hold edges or paths with these cells as targets. Such consumers we also call *retainers*. Retainer profiling should answer the questions 'Why are these cells still in the heap? What's holding onto them?'. But to form the basis of useful profiling information the specification of retainers needs refining. For example, what retains a live 'cons' cell? At one extreme, the answer could be a preceding 'cons' in the spine of a list, if that is where the sole reference to the cell is held. Not very informative! Going to the other extreme is no better: we do not wish to identify the set of retainers with the root set of the live graph. There is a useful intermediate position:

1. Do not consider as a candidate retainer any cell representing a weak-head-normal form; rather, consider only function closures.

2. But consider these closures as candidates whether they are in the heap or (currently being evaluated) on the stack.

```
                                       generation  y
for y in 0..N-1                                      0 0 0 3
    for x in y..N-1                                  0 0 4 0
       A[x,y] := A[x,y] - A[x+1,y]                   0 1 1 0
                                                     1 0 1 1

                                                   x = lifetime + y
```

Figure 7: Lifetime transformation: Phase I. Afterwards A[x,y] is the number of yth generation cells with lifetime x-y.

```
                                       generation  y
for y in 1..N                                        3 0 0 0
    for x in y..N                                    4 0 0 0
        A[x-y,y] := A[x,y]                           1 1 0 0
    for x in (N-y)+1..N                              1 0 1 1
        A[x,y] := 0
                                                   x = lifetime
```

Figure 8: Lifetime transformation: Phase II. A simple shear transforms A so that afterwards A[x,y] represents the number of yth generation cells with lifetime x.

```
v:[0..N]
V0(i) = if i<0 then 0 else v[i]

                                           census  y
for x in 1..N                                        3 0 0 1
    s := v[0] := A[x,0]                              4 1 1 1
    for y in 1..N                                    1 1 1 1
        v[y] := A[x,y]                               1 0 1 1
        s := s + v[y] - V0(y-(x+1))
        A[x,y] := s                                x = lifetime
```

Figure 9: Lifetime transformation: Phase III. Afterwards A[x,y] is the number of cells alive at the yth census whose eventual lifetime was x. The assignment to s in the inner loop updates a running total of the last x entries in the vector.

3. Also consider CAF's as candidate retainers.

4. Now define the retainers of a cell, at any given point in the computation, to be those candidate retainers from which there is a path to the cell *not passing through any other candidate.*

Prohibiting constructors as retainers removes the problem with 'cons' cells being retained by other 'cons' cells. Instead all the 'cons' cells in a list spine inherit the retainers of the initial 'cons'. Take the expression:

```
filter (/="hello") ["world"]
```

as a simple example. Here `filter` retains both the list `["world"]` and the closure for the partial application of `/=`. But the string `"hello"` is retained by `/=`, as the only path to it from the `filter` closure passes through the `/=` closure.

In general a cell has a *set* of retainers, not just one. Since the powerset of all named definitions in a program may be huge, it might seem that profile charts banded by retainer sets pose nasty display problems — fragmentation into many bands, each with a rather large label! In practice, however, this powerset is sparsely populated because many combinations of retainers are impossible. Also, for initial profiling of large programs, modules rather than individual functions could be treated as retainers — an idea we know works well for producers. *Approximation* is another way to avoid costly and complex retainer charts, as we shall see shortly.

3.1 Census traversal for retainer profiling

All the work involved in collecting information about retainers can be confined to the census traversals. The only overhead required during evaluation is an extra word in each heap cell to represent a set of functions found to retain the cell during a census traversal.

If every live cell had a unique retainer, all the (single element) retainer sets could be computed in a single traversal of the whole graph. It would suffice to keep a note of which was the last candidate retainer from which pointers were followed to reach the present cell. In practice, however, graph reduction is only interesting because there are *shared references*, and shared references mean that multi-element retainer sets are possible. (There may be sharing *without* multiple retainers – for example, a closure with the same structure in two different argument positions, or many closures for the same function with an argument common to each.)

Ignoring representation details for the moment, assume there is some representation of the empty set. Assume also operations to test for set membership and to add an element to a set. Figure 10 sketches a simple algorithm for computing retainer sets. Prior to every recursive call of `retain` an element is added to the retainer set of some cell `c`. No element is ever removed. It follows that the

```
assert: c.retainers is empty for every cell c

for every root cell r
  retain(r,r)

retain(c,r) =
  if r ∈ c.retainers then return;
  c.retainers := c.retainers ∪ {r};
  if isretainer(c) then r' := c else r' := r;
  for every successor c' of c
    retain(c',r')
```

Figure 10: a simple method for computing retainer sets.

algorithm terminates correctly, because there are only finitely many cells, and finitely many possible elements of retainer sets.

However, the algorithm of Figure 10 is unsatisfactory on at least two counts.

1. It is *space-consuming*, implicitly assuming a (large) stack to maintain the contexts of recursive calls.

2. It is *time-consuming*, possibly requiring many traversals of some parts of the graph.

Sharing traversal with garbage collection

The garbage collector also has to traverse the heap, so a natural way to improve efficiency is to combine garbage collection with census taking [RW93b]. If the implementor is concerned about space-efficiency the garbage collector may use *pointer reversal* to avoid the need for a large stack of ancestral cells [SW67]. But if pointers are reversed, it is not possible to re-traverse parts of the graph! The solution is to introduce a comparatively *small* stack of *traversal requests*. Instead of revisiting a shared subgraph as soon as it is encountered, a traversal request for it is pushed onto this stack. Each request is just a pair `(c,r)` where `c` is the root cell of a subgraph and `r` is an additional retainer. At the end of the main traversal, the stack of subgraph traversal requests must also be fulfilled. The revised algorithm is sketched in Figure 11 — pointer reversal is not shown explicitly, but assumed at the points of recursive call and return.

Though the algorithm of Figure 11 avoids the space-consumption of a large stack, it is just as time-consuming as the previous algorithm of Figure 10. Profiling retainers for a program with many functions and a lot of sharing could still be very expensive. So we next consider two ways of reducing the cost of multiple traversals.

```
assert: c.retainers is empty for every cell c

for every root cell r
  s := emptystack;
  retain(r,r);
  while nonempty(s)
    sharetain(pop(s))

retain(c,r) =
  if r ∈ c.retainers then return;
  if c.retainers = {} then
    c.retainers := {r};
    if isretainer(c) then r’ := c else r’ := r;
    for every successor c’ of c
      retain(c’,r’)
  else
    push(s,(c,r))

sharetain(c,r) =
  if r ∈ c.retainers then return;
  c.retainers := c.retainers ∪ {r};
  if isretainer(c) then return;
  for every successor c’ of c
    sharetain(c’,r)
```

Figure 11: computing retainer sets using a stack of requests.

Approximating retainer sets

Suppose we aim to compute only those retainer sets with fewer than `N` elements, for some small `N` specified by the user, *approximating* all larger retainer sets by a single value meaning 'many retainers'. Then no cell in the graph need be visited more than `N+1` times.

The programmer is often most interested in knowing when closures of one particular function (or group of functions in one module) *uniquely* retain a large number of cells, since this identifies a particular part of the program to be looked at. For this purpose, even the coarsest approximation, with `N=1`, is good enough. Moreover, having distinct bands in a profile chart for multi-element retainer sets with all but one element in common could be unhelpful, giving an unduly fragmented view of the heap. For this reason, there may be little call for the precision of retainer profiles with `N>3`, say. (However, see the remarks about the `parser` example in §5.3.)

Combining traversal requests

Even in cases where an `N`th approximation is good enough, we should prefer, if possible, to avoiding visiting cells `N+1` times. Also, there may still be occasions when exact retainer sets, or approximations including larger sets, are needed.

In the algorithms of Figure 10 and Figure 11 retainer sets are extended one element at a time. If a cell is going to end up with a set of several retainers, another way to improve efficiency is to insert more than one of these retainers in a single visit to the cell.

For this purpose we now generalise traversal requests to specify a *set* of additional retainers rather than just one. To make use of this capacity, as each new request `(c,{r})`, is created on the stack, *a pointer to it* is placed in cell `c`. When a `retain` traversal revisits a cell, if it is already the subject of a traversal request, no further request is needed: any new retainer is added to the set in the existing request. Similarly, `sharetain` traversals need not extend beyond any cells found to have further requests pending: the retainer set in the pending request is replaced by its union with `sharetain`'s set argument.

Figure 12 sketches the revised algorithm. The operations to set and clear pointers from cells to requests are subsumed under `push` and `pop` operations respectively. This algorithm clearly performs far less traversal than that of Figure 11 for some graphs. However, the set operations required are more complex: subset testing rather than membership testing, and set union rather than addition of a single element, so the net gain is uncertain.

How many passes?

An ideal algorithm would need only a small fixed number of passes over the heap to compute exact retainer sets (or any desired approximation to them). As yet

```
assert: c.request is null and
        c.retainers is empty for every cell c

for every root cell r
  s := emptystack;
  retain(r,r);
  while nonempty(s)
    sharetain(pop(s))

retain(c,r) =
  if r ∈ c.retainers then return;
  if c.retainers = {} then
    c.retainers := {r};
    if isretainer(c) then r’ := c else r’ := r;
    for every successor c’ of c
      retain(c’,r’)
  else if c.request = null then
    push(s,(c,{r}))
  else
    c.request.set := c.request.set ∪ {r}

sharetain(c,rs) =
  if rs ⊆ c.retainers then return;
  if c.request = null then
    c.retainers := c.retainers ∪ rs;
    if isretainer(c) then return;
    for every successor c’ of c
      sharetain(c’,rs)
  else if rs ⊈ c.request.set then
    c.request.set := c.request.set ∪ rs
```

Figure 12: computing retainers using request combination.

we have neither devised such an algorithm nor convinced ourselves that it cannot be done!

Whether the request-combining algorithm of Figure 12 completes in no more than two passes depends on the order in which traversal requests are executed. The critical cells are those reachable from the root of two or more separate requests along a retainer-free path. If all paths to such cells from the root of the earlier request also pass through the root of the later request, the algorithm works perfectly: `sharetain` visits no part of the graph more than once. This advantageous property does not hold in general. There may be suitable heuristics for re-ordering the stack to make it more likely, but we have not investigated any.

3.2 Retainers as selectors

Each kind of cell classification should be available to the programmer not only as the basis of banding itself, but also as a way to specify a restricted population of the heap to be banded in some other way. Extending this principle to retainers, the profiler should be able to chart the producers, constructions or lifetimes of all cells held by a specified set of retainers. Profiling only the cells whose retainer set `r` *exactly equals* a specified set `s` might seldom be useful: alternative more relaxed conditions include $\mathtt{r}\subseteq\mathtt{s}$, or $\mathtt{s}\subseteq\mathtt{r}$, or $\mathtt{r}\cap\mathtt{s}\neq\emptyset$.

4 Implementation in `nhc`

Heap profiling extended to lifetimes and retainers has been implemented as part of `nhc`, a Haskell compiler[1] written in Haskell for a machine with small memory [Röj94].

4.1 A Haskell compiler for small machines

The main goal in `nhc` is to use as little memory as possible. Fancy algorithms that increase the memory requirements of the compiler itself are ruled out; `nhc` provides only what is essential in a G-machine implementation [Pey87]. The source code for `nhc` is less than 11,000 lines of Haskell.

To reduce run-time code size, `nhc` generates byte code instead of machine code. The gain in space is substantial as most byte code instructions fit in 2 bytes or less whereas modern RISC processors use 4 byte instructions. Further, the byte code instruction set can be optimized for graph reduction so that fewer byte code instructions are needed than machine code instructions for a given Haskell function.

[1] or, to be strictly accurate, 'Nearly a Haskell Compiler'

Heap space in `nhc`

The minimum feasible heap size depends on the peak amount of live data and on the rate at which free heap space is used. Several design choices in `nhc` help to reduce the amount of live data. For example, all three of the space faults attributed to compilation in [RW93b] are avoided:

Variables in a LHS pattern are updated simultaneously. When a variable in a left-hand pattern is used, *all* variables in the pattern are set to point to their parts of the expression, using a compilation scheme due to Jan Sparud [Spa93]. This means that the node representing the right hand expression can be released as soon as the first variable in the pattern is used instead of waiting for the last one.

No updates are done by copying. If a function application reduces to the value of an existing piece of graph, the reduced application node is overwritten with a pointer to the root of this result. If a function creates its own result, then overwriting is used unless the result is larger than the redex, in which case an indirection is used.

Tail calls are treated as updates. If the last expression in a function body is a call to a function `nhc` tries to build the new application on top of the old one, rearranging the stack, and then jumps to the function in the new application. If the new application is larger than the old one then an indirection node is used to overwrite the old application before jumping to the new function.

The goal of space-efficiency

By cross-compiling `nhc` on a large machine using `hbc`, the original 2D heap-profiler was applied, but did not yield sufficient information to resolve all space-problems satisfactorily. A large part of the motivation for implementing a second generation heap-profiler in `nhc` was the prospect of self-application: it should enable the memory demands of the compiler itself to be further reduced. (Section 5.2 gives one illustration of how this has been realised in practice.)

4.2 Implementation of lifetime profiling

For lifetime profiling `nhc` uses an additional word in every cell to store the time at which the cell was allocated. Time is measured by the number of censuses that have occurred. The basic format of log files is simple: for example, the census lines

```
3 124
4 256
```

```
7 134
```

record 124 cells from the 3rd generation, 256 from the 4th and 134 from the 7th. When lifetime information is used to specify a selected population of cells, the format of the log file is different. For example, the line

```
Prelude.: 3 24 4 12 7 96
```

records the cons cells found during one census: 24 from the 3rd generation; 12 from the 4th and 96 from the 7th.

In §2, the transformation of the census data deals throughout with 'numbers of cells' of some kind. In practice the sizes of cells may vary, and the post-processor works with 'amounts of memory' in bytes. Also phases I and II of the transformation algorithm (Figures 7 and 8) are combined into a single pass.

4.3 Implementation of retainer profiling

Retainer profiling in `nhc` uses a variation of the algorithm sketched in Figure 11. Instead of interleaving calls to `retain` and `sharetain`, all calls to `retain` are made before any call to `sharetain`. The advantage is that the marking phase of the garbage collector can be used to implement `retain`. A clear mark-bit indicates an empty retainer set. This avoids the need to clear all retainer sets before taking each census. One disadvantage is a deeper request stack.

`Nhc` also maintains a count of how many cells are retained by each retainer set: there is only one structure representing each distinct set, and the 'sets' assigned to cells are in reality pointers to these shared structures. Each time a new retainer is added to a set, a new set structure is created if necessary; the cell count is incremented in the new set, and decremented in the old one.

After all the traversals corresponding to `retain` and `sharetain` have been completed, all the required census data is in the set structures. There is no need for any further traversal of the graph to collect this information.

4.4 User options and profile charts

The default for `nhc` is to compile without any profiling code at all. The compiler supports two levels of heap profiling:

1. *Level 1 profiling:* allows profiling of producers and constructions. It also gives access to *kind profiling* — a limited form of retainer profiling that divides cells into those reachable from the stack and those reachable *only* from code (CAFs). Kind profiling is useful because space faults due to CAFs can often be fixed by changing CAFs into functions.

2. *Level 2 profiling:* includes full retainer and lifetime profiling in addition to the heap profiles available at level 1.

The advantage of using level one profiling is faster execution and lower demands on memory. Any of the available heap profiles can also be used to restrict the observed set of nodes.

A program compiled for profiling does not by default produce a profile. The program must be told which profile the user wants. This is done by some combination of the run-time flags:

`-m`	module-level producer
`-p`	definition-level producer
`-c`	constructor
`-k`	kind
`-l`	lifetime
`-r`	retainer

It is the first occurrence of one of these flags that determines which kind of profile to produce. A `-r` flag can be followed by a number specifying a maximum size for exact sets of retainers (with `-r1` as default): larger sets are all treated as the same set of 'many retainers'.

Subsequent occurrences of `-m`,`-p`,`-c`,`-k`,`-l` and `-r` *restrict* profiling to part of the heap population. For example, `-p -cPrelude.:` profiles producers of 'cons' cells. A cell is only included in a profile if it satisfies *every* restriction. All but lifetime restrictions are specified by a list of names following the relevant flag. A retainer restriction is satisfied if one of the functions in the retainer set is mentioned in one of the retainer restrictions. Lifetime restrictions cannot be specified fully during execution: a `-l` restriction only prepares the way for a lifetime interval to be supplied to the post-processing program.

For all but lifetime profiles, bands are ordered in the way described in [RW93b], with smoother bands below rougher ones. In lifetime profiles, bands for long-lived objects are placed lower than bands for short-lived ones. The result is similar to the 'smoothness' ordering, as the populations of longer-lived cells tend to be more stable. Often there are more different cell lifetimes than can be distinguished in the bands of a profile; the solution is to group a range of lifetimes together, so that the Nth band represents cells with lifetimes between 2^N and $2^{N+1} - 1$.

4.5 Performance overheads of profiling

The final goal of heap profiling is to reduce the amount of space (and perhaps time) needed to run a program. But to find the space faults we need both extra space and extra time.

The extra space is for additional code and larger cells. As Table 1 shows, the code size of `nhc` increases by 50% if the compiler itself is being profiled, as a result of the need to store names for all constructors and producers, in addition to the code to insert this information in all created cells. If we are interested *only* in lifetimes, or kinds, then the extra code size could be negligible, as no name

level	code size	options available
0	893 kb	none
1	1344 kb	-k,-m,-p,-c
2	1374 kb	-k,-m,-p,-c,-l,-r

Table 1: Code size of `nhc` by level of profiling.

level	profile	`nhc` time	`parser` time	comment
0	none	125 s		
1	none	161 s		
	`-m`	209 s		153 censuses
2	none	260 s	41 s	
	`-l`	310 s		225 censuses for `nhc`
	`-m`	329 s		35 censuses for `parser`
	`-r1`	355 s	64 s	
	`-r3`	394 s	82 s	
	`-r5`	428 s	97 s	
	`-r7`	457 s	97 s	
	`-r9`	484 s	98 s	

Table 2: Execution times for `nhc` and for `parser`, by level of profiling.

information is needed. But `nhc` does not currently offer the option of compiling for these types of profiling alone.

Cells are larger when heap profiling is used, as we need to attach producer and constructor information to them. The average cell size *without* profiling information is between 5.7 and 6.5 words for most programs. Even the single extra word needed for 2D profiling results in an 18% increase of the amount of live heap. With the extension to dynamic and structural profiling, up to two further words must be allocated in each cell, to record generations and retainer sets, so the overall increase in live heap size rises to about 50%.

Execution time also increases when profiling as Table 2 shows. The figures for `nhc` relate to the compilation of a 200 line module that imports 39 interface files.

There are few widely shared data structures in `nhc`: most cells have only one or two retainers. One might suspect that this is why it is possible to increase the size of exact retainer sets without severely affecting execution times. But even for a program in which *most* cells have several retainers, such as the `parser` program discussed in section 5.3, detailed retainer profiling does not slow execution by much more than a factor of two.

Costs of log files and post-processing.

A typical log file is less than 100 kb, though when profiling nhc itself log files can run to 500 kb or more. Taking more frequent censuses results in larger files, but it is rarely useful to have more than 30 censuses of a single computation. For the moment log files are just ASCII texts; in a compact binary format they would be less than a quarter of their present size.

The post-processor, hp2graph, uses very little time in comparison with the previous recording of census information. One thing that can slow hp2graph down is the use of a lifetime interval to specify a restricted profile. The post-processor avoids this problem by 'thinning out' the data from the log file when the number of censuses exceeds a fixed threshold. (The user is informed, and has the option to force hp2graph to use all census data regardless of cost.) The maximum post-processing time needed for any example in this paper is 7 seconds; most examples need less than 2 seconds.

5 Example applications

5.1 Clausify revisited

The original paper on heap profiling[RW93b] includes as an extended example a program called clausify that puts logical formulae into clausal form. After several iterations of profiling and improvement, the 1.3 Mb required by Version 0 of the program for a benchmark computation is reduced to just 7 kb for Version 5. The nhc producer profile for Version 5, transcribed from Lazy ML to Haskell, is shown in the upper part of Figure 13. A similar profile is exhibited in the original paper with the remark that although it might be possible to reduce the space cost of disin', it is not obvious how. The construction profile in the lower part of Figure 13 reveals that the hump of cells produced by disin' is made up of Dis and Con constructions of the Formula type representing propositions:

```
data Formula =
  Sym Char |
  Not Formula |
  Dis Formula Formula |
  Con Formula Formula |
  Imp Formula Formula |
  Eqv Formula Formula
```

But given the defining equations for disin'

```
disin' :: Formula -> Formula
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
```

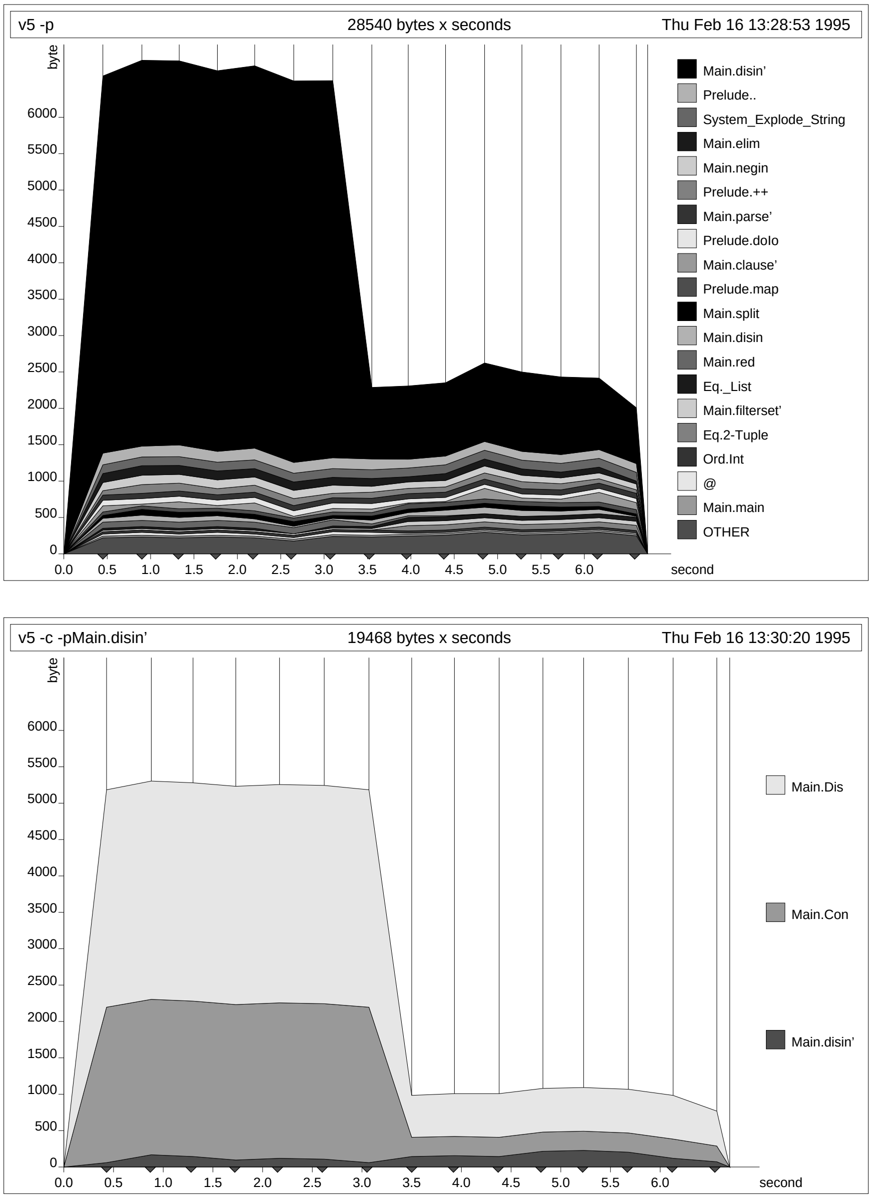

Figure 13: The original two dimensions of heap profiling, applied to Version 5 of `clausify`. Upper: `disin'` is the main *producer* of heap cells. Lower: `Dis` and `Con` are the main *constructions* (produced by `disin'`).

its production of numerous Dis and Con cells is not surprising, and without further information it is not clear how to reduce heap-size any further.

Using the lifetime and retainer profiling of nhc, however, we can now discover something more about the residual hump of cells in clausify computations. As Figure 14 shows, this population of Con and Dis cells produced by disin' *survives throughout the plateau of memory demand in the closures of calls to* disin' *itself.* Armed with this extra information, we seek first the explanation of it in terms of the operation of the program, and then a remedy for it in revised definitions of disin' and related functions.

The explanation

The overall structure of the clausify program is a Formula-processing pipeline

```
... unicl . split . disin . negin . elim ...
```

in which the elim function eliminates equivalences and implications by translating them to other connectives, negin pushes any negations to the innermost positions (enclosed by a mixture of conjuncts and disjuncts), and disin translates its argument to conjunctive normal form (CNF — with conjunctions outermost, then disjunctions, then literals). The focus of our attention, disin', is used as an auxiliary by disin.

```
disin :: Formula -> Formula
disin (Con p q) = Con (disin p) (disin q)
disin (Dis p q) = disin' (disin p) (disin q)
disin p = p
```

To complete the picture (to the extent we need here) split chops a CNF proposition up into separate conjuncts, and unicl rearranges each conjunct into clausal form discarding any tautologous or duplicate clauses.

So why *do* Formula constructions build up in disin' closures? Consider, for example, what happens if disin is applied to a *purely disjunctive* formula — one whose structure is a tree of Dis constructions with literals at the leaves.

- By the 2nd disin equation the result can be expressed as a similar tree but with disin' in place of Dis.

- By the 3rd disin' equation the result of this expression is once again the tree with Dis in place of disin'!

It might seem, then, that disin acts just like the identity function in this case. However, *it is not a lazy identity.* It is *path strict* in the disjunctive tree structure of its argument. Because the 3rd disin' equation can only be applied when the 1st and 2nd have been found *not* to match, *the full disjunctive tree must be evaluated and reconstructed* before the outermost Dis construction of the result can be made.

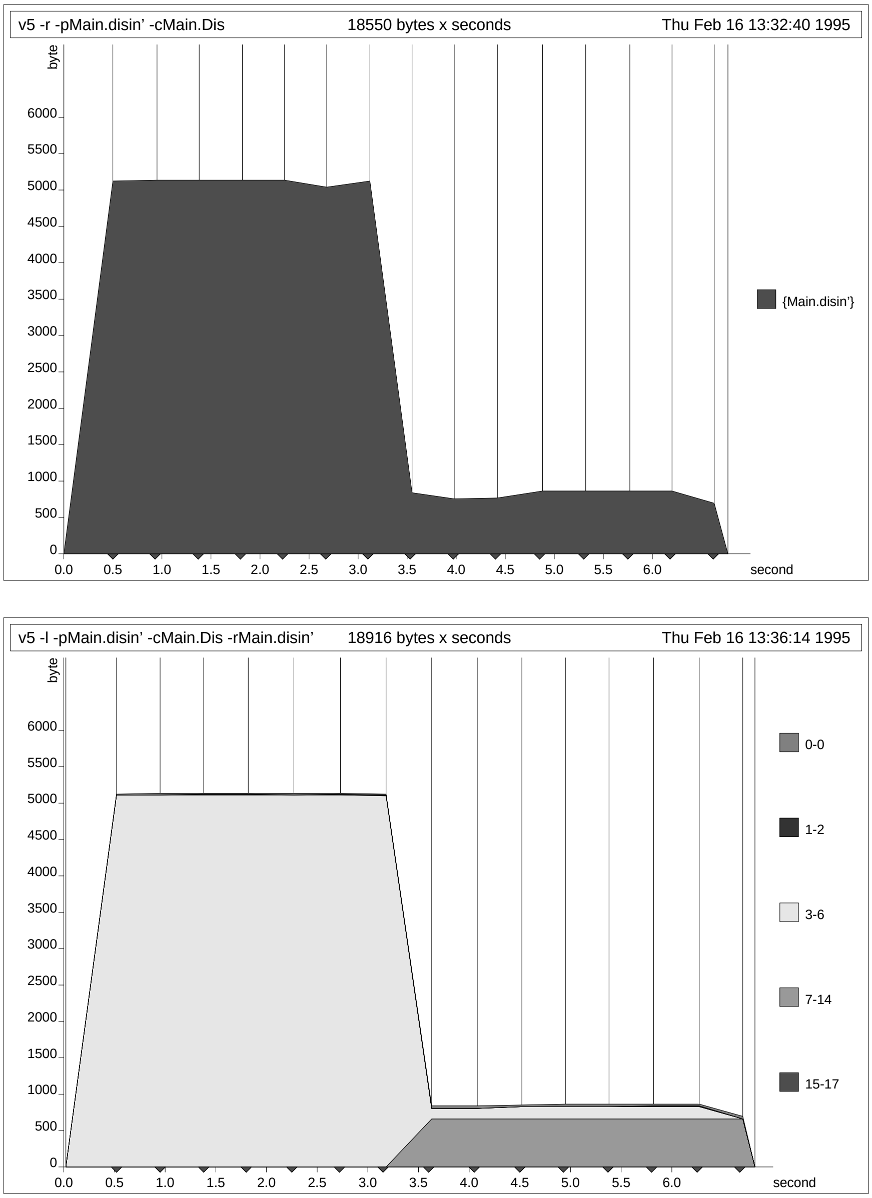

Figure 14: The new dimensions of heap profiling, applied to Version 5 of `clausify`. Upper: `disin'` is the *retainer* of the problem cells. Lower: their *lifetime* is nearly the whole period of peak memory demand.

The remedy

There is no avoiding the need to traverse disjunctions looking for conjunctions: that is in the nature of the CNF transformation. But for which subsequent process are the disjunctive trees required? It is `unicl`, which reduces them to a partitioned set of leaves — discarding duplicate leaves and discarding entire trees as tautologous if both positive and negative forms of the same literal are found. So rather than `disin` reconstructing disjunctive trees just as they are found in its argument, *normalising* them into ordered chains of literals enables us to bring forward some of the work from `unicl` so that heavy pruning can occur earlier in the pipeline. This sort of program transformation is sometimes termed *filter promotion*[Dar78].

To implement the idea we define a *normalising constructor* `dis` to be used in place of `Dis` in the final `disin'` equation. To realise the full benefits we also need an extra `Formula` constructor `Taut` to represent a tautology. Figure 15 shows how `dis` is defined in Version 6 of the `clausify` program. The auxiliary `cmplit` compares literals, with four possible results: two literals involving a common symbol are either the same or opposite (`Same` or `Oppo`); if the symbols differ, the first must either precede or follow the second in character order (`Prec` or `Foll`).

As Figure 16 shows, this reformulation is very successful for the benchmark problem of [RW93b]. The peak memory demand is reduced from 7 kb to 1 kb, and the time drops to a fraction of a second. Not only is the heap far smaller, the program runs *much* faster. The improvement is not specific to the benchmark proposition: for a much larger input formula (derived by combining a number of different exercises from a textbook[MW85]) peak memory demand falls from 15 kb to a little over 2 kb and again there is a dramatic speed-up.

In retrospect

So why didn't we think of this improvement before? This question is hard to answer accurately, but it seems appropriate to attempt some sort of answer in a practical assessment of the use and limitations of profiling. There were at least four reasons why the reformulation of `disin'` sketched above was not prompted by the earlier profiling exercise.

1. *The assumption of diminishing returns.* The peak live heap size had already dropped from 1.3 Mb to 7 kb as a result of successive revisions of the program. A maximum heap-size of 7 kb is very small by the usual standards of functional language programmers and implementors. The space occupied by `Dis` and `Con` cells in particular had been targetted at an earlier stage and substantially reduced. The last revision of the program had gained just 1 kb. So it seemed unlikely that there were further substantial gains to be made.

```
dis Taut q   =  Taut
dis p Taut   =  Taut
dis p@(Dis p1 p2) q@(Dis q1 q2) =
    case cmplit p1 q1 of
        Same -> dis’ p1 (dis p2 q2)
        Oppo -> Taut
        Prec -> dis’ p1 (dis p2 q)
        Foll -> dis’ q1 (dis p q2)
dis p@(Dis p1 p2) q =
    case cmplit p1 q of
        Same -> p
        Oppo -> Taut
        Prec -> dis’ p1 (dis p2 q)
        Foll -> Dis q p
dis p q@(Dis q1 q2) =
    case cmplit p q1 of
        Same -> q
        Oppo -> Taut
        Prec -> Dis p q
        Foll -> dis’ q1 (dis p q2)
dis p q =
    case cmplit p q of
        Same -> p
        Oppo -> Taut
        Prec -> Dis p q
        Foll -> Dis q p

dis’ p Taut  =  Taut
dis’ p  q    =  Dis p q
```

Figure 15: Normalising disjunctions in Version 6 of the `clausify` program.

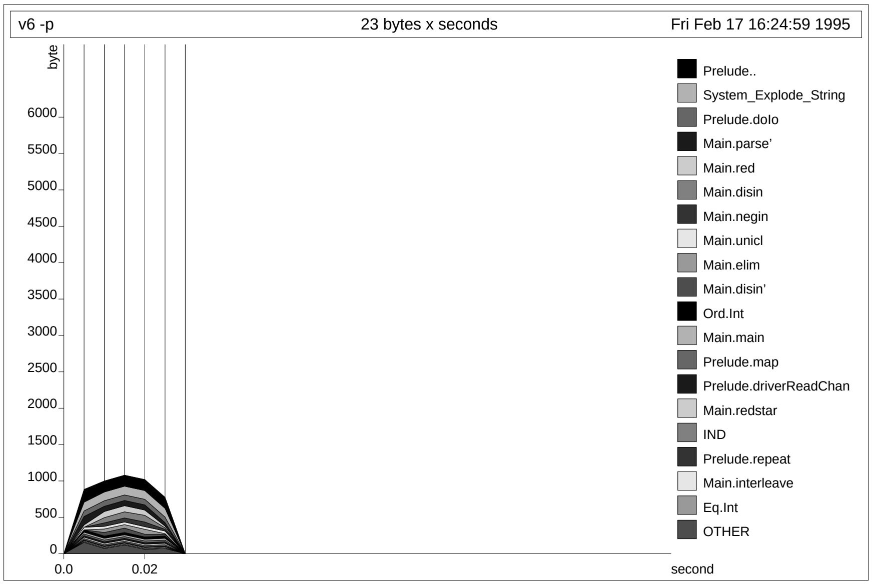

Figure 16: A producer profile of `clausify` Version 6 with `disin'` improved by the use of a normalising constructor function for disjunctions. The space-scale is the same as in Figures 13 and 14, but the time-scale has been altered to avoid showing a computation of negligible width!

2. *The nature of faults found in earlier versions of the program.* The major fault in Version 0 of `clausify` was a 'pipeline blockage' in which cells produced by one function in the main pipeline accumulated because of the over-strict workings of a *later* pipeline function. It was therefore natural, though mistaken, to look for a similar explanation for the accumulation of cells produced by `disin'`.

3. *The prior classification of* `disin'` *as a desirable remedy.* The overall problem being addressed was the inefficiency of the original `clausify` program. The `disin'` auxiliary was not part of this 'old inefficient' program. It was only introduced in Version 4 as a 'new, more efficient' component to flatten peaks of demand for heap space by the `disin` function in Version 3 — a modification that had proved very successful, reducing peak heap size by a factor of five.

4. *The limited kinds of information provided by 2D profiles.* The 'who produces what' profile pointed only to a hump of `Dis` and `Con` cells produce by `disin'`: in view of the preceding points, and in the absence of further information, it was a plausible assumption that the hump was either inevitable or else that reducing it required some subtle change in an unidentified consumer of the cells. It was only the evidence from lifetime and retainer

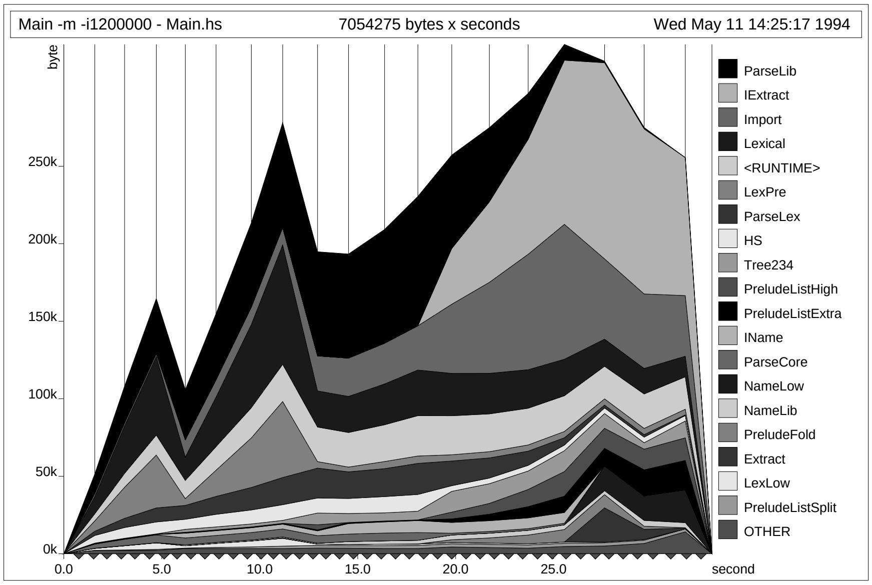

Figure 17: Heap production of `nhc` by module, when compiling a small program.

> profiling that forced attention back to the workings of `disin'` as the long-term retainer of the `Formula` constructions that `disin'` itself produces.

5.2 Self-application to `nhc`

An important part of the motivation for implementing an extended heap profiler in `nhc` was the potential application of the profiler to the compiler itself. Here is just one illustrative example of a space fault in the compiler, analysed using the new profiler, and subsequently fixed.

Figure 17 shows a heap profile of `nhc` compiling a small program (of about 20 lines), with cells banded according to the module that produced them. The two 'horns' are a prominent feature, and it seems natural to ask whether they could somehow be removed. The cores of these horns represent cells produced by the two lexical modules, `Lexical` and `LexPre`. A retainer profile restricted to these modules is given in Figure 18. Clearly the function `parseit` should be investigated. Examining its definition, the source of the space fault is soon apparent:

```
parseit :: Parser a i a -> i -> Either ParseError a
parseit p input =
    case (p initGood initBad input initError) of
    (Left err) -> Left err
    (Right (a,_,_)) -> Right a
```

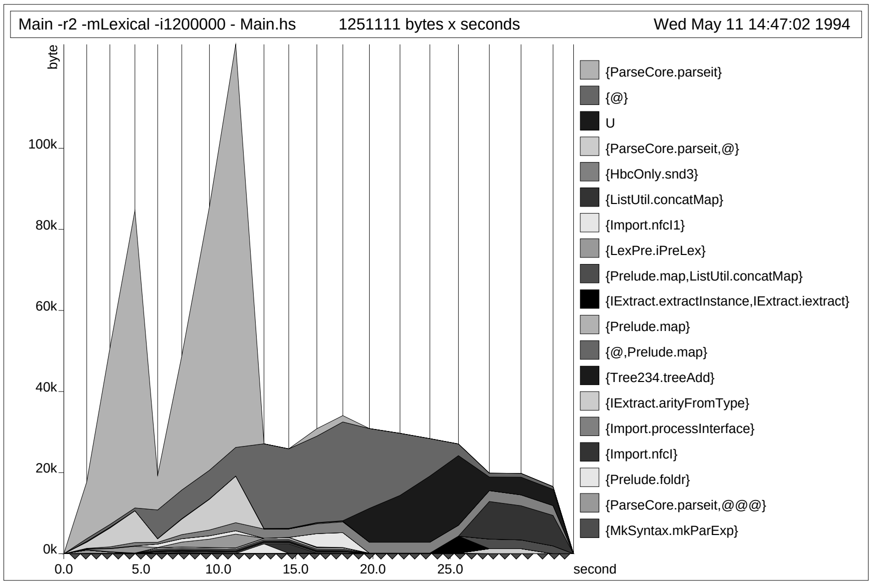

Figure 18: What retains the heap cells produced by the lexical modules?

The problem is that the abstract machine in `nhc` holds on to all function arguments until the function returns, or does a tail call, even if the arguments are not used any more. In this case `parseit` holds on to the input until it can return either an error message `Left err` or the syntax tree `Right a`. This means that no memory used for `input` can be garbage collected until the whole syntax tree is checked. The remedy is to replace the case expression by a function application, as follows.

```
parseit p input =
    parseit' (p initGood initBad input initError)
  where
    parseit' (Left err) = Left err
    parseit' (Right (a,_,_)) = Right a
```

The new `parseit` does a tail call before evaluating the application of `p`. The argument pointer to `input` is removed from the stack before the parsing begins. The garbage collector can therefore reclaim the memory used by `input` as soon as the parser has consumed it. Figure 19 shows a retainer profile for heap cells produced by the lexical modules of the modified compiler.

The two 'horns' were observed early in the development of `nhc`, but their cause was unknown. With only the view given by 2D profiling, it was easy to blame either the lexical analyser or the parser. After fruitless investigations of these parts the problem was considered unsolvable with the available tools. The retainer profile on the other hand blames the badly written wrapper function for

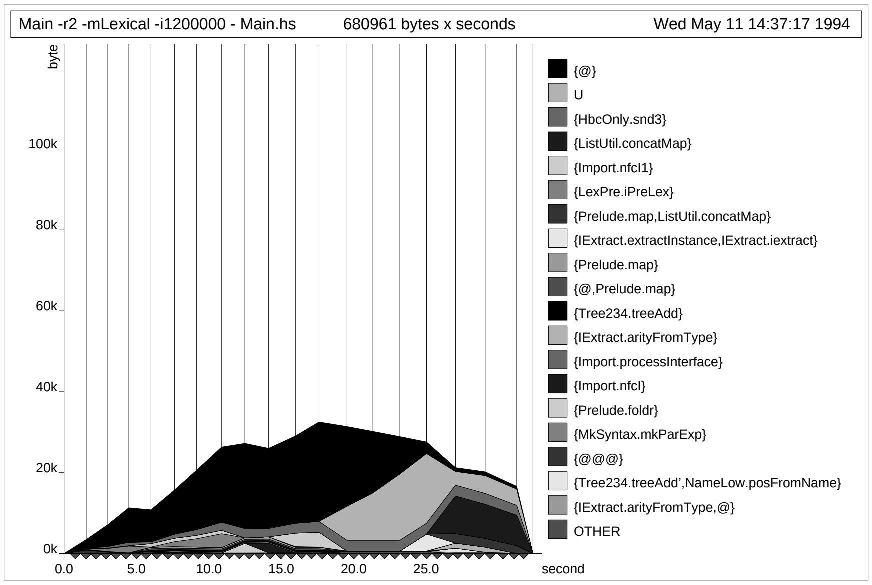

Figure 19: Compilation without lexical horns.

the parser, and this was where the real fault lay. Retainer profiling is very good for locating space faults due to functions holding on to unneeded data.

2D profiling has however been used to verify that changes in `nhc` really did decrease space usage. This is not always obvious. At one time, for example, `nhc` included a function to strip the symbol table of information no longer needed. But the stripped symbol table took up *more* space than the original due to lazy evaluation: large parts of the old table were not released when the new table was built.

5.3 Examples from the ‘nofib’ Benchmark Suite

By applying `nhc` to some programs from the ‘nofib’ Benchmark Suite[Par93], we hoped to gain knowledge about ‘normal’ lazy programs. Compilation succeeded for 42 of the 45 programs available[2] — not counting `clausify`. We were able to locate space faults in several of these programs. In some cases 2D profiling was sufficient, but we shall concentrate here on programs for which 2D profiling was inadequate but results from the new profiler led us to the problem.

`parser`***: a small parser built with parser combinators.*** The producer profile is ‘triangular’, representing a steadily growing heap; it does not suggest the location of an error as production is almost evenly distributed among six

[2] Recall that `nhc` stands for *Nearly* a Haskell Compiler! A number of the programs needed extra type signatures or minor changes to eliminate n+k patterns.

functions. Retainer profiling reveals an unusual amount of sharing: even using the 5th approximation the catch-all ‘many retainers’ band dominates the profile chart. However, the 6th approximation reveals that most retainer sets with substantial holdings include at least one parser combinator. On investigation, it turns out that all the parser combinators hold onto their input stream until the last component parser is called.

A *partial* solution, involving a compiler modification, prunes arguments from the stack after their last possible use. However, this does *not* help for the backtracking combinator defined in `parser` as it must hold on to the input until one parser has succeeded to be able to restart with another parser in the case of failure. It is possible to write a backtracking parser combinator without space leaks — indeed, `nhc` contains such a parser — but this would entail wholesale rewriting of the `parser` benchmark program.

`primetest`***: tests the Mersenne Prime*** $2^{607} - 1$ ***for primality.*** Again the profile is triangular with heap-size at a maximum just before termination. Retainer profiling pinpoints one function `doLine` that retains most of the memory. It is defined as follows:

```
 doLine cs cont rs
  = if t then "Probably prime": rest else "Composite": rest
    where n         = readInteger cs
          (t, rs’) = multiTest 100 rs n
          rest      = cont rs’
```

The problem is that references to the arguments are kept on the stack until `doLine` can return, and `rs` is a large list of integers. Although a revised compilation scheme might avoid the problem, it can in any case be solved by rewriting the definition of `doLine`:

```
 doLine cs cont rs
  = doLine’ t
    where n         = readInteger cs
          (t, rs’) = multiTest 100 rs n
          rest      = cont rs’

doLine’ True =  "Probably prime": rest
doLine’ False = "Composite": rest
```

The only thing done in `doLine` before all pointers to the arguments are removed from the stack is to build the necessary closures. The garbage collector can now reclaim the big part of the heap that was previously only retained by `doLine`. The overall cost was reduced to one third by the above change.

queens***: solves the 10×10 queens problem.*** The original version of queens has yet another triangular heap profile. There is more than one major producer, but the dominant *retainer* is foldl. The queens programs does not mention foldl explicitly, but nhc's prelude defines length just as in the Haskell report:

```
length = foldl (\n _->n+1) 0
```

The problem is a build-up of closures for the lambda term; these closures also retain the entire list argument to length. By redefining the length function the overall cost can be reduced from 3.87 Mbs to 115 kbs, and the maximum live memory from 70 kb to less than 2 kb.

reptile***: a graphical design program based on tiling.*** The producer profile shows that nearly all of the heap is produced by append – a low-level auxiliary that is heavily used in many parts of the program. However, kind profiling points to a large number of CAFs representing command-streams for the display of a complex screen. These CAFs are evaluated to strings and then kept in case they are needed again. The garbage collector cannot remove them as their future use depends on the input. Translating them into functions reduces the maximum heap size from 110 kb to 30 kb. Further savings by evaluating under constructors lowers this to 17 kb and also reduces execution time by 25%. (However, execution time would increase for a different input requiring a re-evaluation of the ex-CAFs.)

exp3_8***: raises 3 to the 8th power using Peanno arithmetic.*** This example is a little different. No space fault was found, but the original version of the program did seem needlessly *time*-consuming. A one-line modification[3] reduces a run-time of 43 s to a mere 0.2 s! The critical alteration was *not* prompted by information gleaned from heap profiles — heap profiling does not solve all problems.

Gains from improved compilation rules

The version of nhc first applied to the 'nofib' Benchmark Suite did not garbage collect CAFs. The reason was simple: CAFs in nhc itself occupy a negligible amount of memory! But this omission results in space leaks in at least four more programs from the 'nofib' suite.

Another problem with earlier versions of nhc was that the compiler tried to remove values from the evaluation stack as late as possible. By doing so, many separate removes could be combined with the removal of the stack frame at the end of the function. This was done to gain speed. It did however result in some

[3] The line to change is `x * S y = (x * y) + x`. Writing it `x * S y = x + (x * y)` is much more efficient. This is related to the difference in efficiency between `a++(b++c)` and `(a++b)++c`.

unexpected space faults and was therefore removed. One example of program with such a space fault is `mandel2`.

`mandel2`***: computes Mandelbrot sets.*** Under the original stack scheme, `mandel2` has a triangular heap profile with *two* main producers. Although there *might* be space leaks in both producers, it seems more likely that the culprit is a retainer holding on to unnecessary data. A retainer profile confirms this: `main`, defined as follows, is the major retainer.

```
main = if finite (...) then print "Success" else print "Failure"
```

The ... is a complicated expression, which the compiler cannot build in one step. The compiler therefore builds the necessary subparts first, and pushes pointers to them on the stack, before the final expression is built. The fault was that those 'temporary' pointers were not removed until `main` finished, and one of the subparts is lazily created as it is consumed by `finite`. Nothing in that subpart could be reclaimed during garbage collection due to the 'temporary' pointer on the stack. Redefining `main` as follows:

```
main = main' (finite (...))
main' True  = print "Success"
main' False = print "Failure"
```

reduces the overall cost by 80%! The reason is that all temporary pointers are removed from the stack together with the stack frame of `main` before `finite(..)` is evaluated. With an improved stack scheme, this kind of redefinition is no longer necessary.

Table 3 summarises the performance gains for the sample of programs from the 'nofib' suite.

6 Related work

This paper extends the original 2D heap-profiling scheme for lazy functional programs developed by Runciman and Wakeling [RW93b, RW93a] and now distributed as part of the `hbc` Haskell compiler. An outline of that work was given in §1.

Our attention has only recently been drawn to an implementation of memory profiling some years ago in a SNOBOL4 interpreter[RGH78]. This system labels allocated memory blocks with *creation histories* comprising a source line-number responsible for the allocation, an indication of the type of data stored (one of a small number of fixed categories) and a time of allocation. Information from all creation histories is output at each garbage collection, and a post-processor derives summary tables of various statistics, including *average* lifetimes. Despite primitives such as pattern-matching, the pragmatics of computation in SNOBOL4

program name	version	overall cost (bytes×seconds)	max space (bytes)	time (seconds)
`primetest`	before	5 420 k	90 k	99
	after	1 510 k	20 k	97
`queens`	before	3 870 k	70 k	103
	after	115 k	<2 k	99
`reptile`	before	1 830 k	110 k	8
	after	30 k	17 k	6
`exp3_8`	before	1 700 k	65 k	43
	after	4 k	65 k	<1
`mandel2`	before	1 530 k	90 k	31
	after	38 k	<2 k	31

Table 3: Some performance gains for 'nofib' programs.

are less subtle than in a functional language such as Haskell — no lazy evaluation and no higher-order functions, for example. Accordingly, the improvements reported for sample application programs are worthwhile, but comparatively modest: a 15% saving is regarded as 'dramatic'. Ripley *et. al.* stress rather the value of detailed information about alternative memory management schemes at the implementation level, especially to avoid wasting premature effort on clever tricks that may be hard to get right and actually save very little in practice. In their concluding remarks, they mention the potential value of a more sophisticated 'spectral analysis' of memory, both at run-time and in subsequent post-processing.

Hartel and Veen [HV88] used an instrumented SASL interpreter to study the characteristics of intermediate combinator graphs during reduction. Their method too was to 'analyse the graph at regular intervals'. This analysis involved measuring, among other things, the overall size of the graph (with separate totals for cells representing data constructions and function applications), the distribution of list lengths (lists being the sole data structure in typeless SASL), the distribution of cycle lengths, and the degree of sharing (as measured by reference counts). Although they focussed throughout on eight specific programs, their aim was *not* to improve these programs. Rather they wished to gain insights into the nature of combinator graph reduction that would be useful in the design of a special purpose machine. So, for example, although they derived information about the distribution of cell lifetimes, this was with a view to assessing the applicability of generational storage management, rather than with a view to identifying parts of a program that create or retain many long-lived cells. Simply accumulating all differences between creation times and GC times enabled Hartel and Veen to obtain lifetime information aggregated across all types of cell and all stages in the computation; they had no need for the kind of scheme described in this paper for lifetime profiling of specific cell types at distinct stages.

Sansom and Peyton Jones describe a profiling scheme implemented in `ghc`, the optimising Haskell compiler developed at Glasgow [SP95]. Their profiler attributes both space and time costs of a lazy functional program to 'cost centres' assigned either implicitly (eg. by identifying a cost centre with each module) or by explicit `scc` ('set cost centre') expression formers. The costs attributed to `c` in connection with an expression `scc c e` are the *entire* costs of evaluating the expression `e` as far as the context demands it, but excluding (a) the cost of evaluating any free variables in `e`, and (b) the costs of any inner `scc` expressions. Special rules for shared CAFs avoid unfairly attributing their costs to the first of several uses. The technique for profiling space essentially follows Runciman and Wakeling, with cost centres as producers. Experiments with profiling based on the age of cells were abandoned because the profiles were found awkward to interpret, but a straightforward creation-time scheme remains (cf. §2). In his thesis [San94], Sansom discusses the `clausify` example (cf. §5.1): a *time* profile reveals that `unicl` is doing the most work, sorting and sifting character literals; unboxing these characters yields a 25% speed-up. As we have seen in §5.1, retainer profiling points to the *source* of `unicl`'s heavy workload, and a revision there leads to a more dramatic improvement.

Clack, Clayman and Parrott advocate *lexical profiling* [CCP95], which they claim reflects the programmer's view of a program, as opposed to the evaluator's or implementor's view supported by other profilers. Costs are assigned to lexical units *as if* programs were executed using a 'call by value' rule, but the values reported faithfully reflect savings due to the actual use of 'call by need'. So the scheme shares with the Glasgow profiler the ideal of attributing costs to lexical units aggregated over all their subexpressions. CAFs do *not* need special treatment, but for the scheme to give accurate results all components with shared uses are forced to be regarded as independent units for profiling purposes. The host implementation for the prototype is an interpreter for a core functional language only, so to date the profiler has only been applied to modest examples.

7 Conclusions and Future work

To develop memory-efficient functional programs, some means of observing the use of heap memory seems essential. Two common kinds of space faults are *dragging* (something remains attached to the live graph beyond the point at which it is last needed) and *closure accumulation* (large graphs build up in closure chains that would be much smaller if evaluated). Sometimes such faults can be found by careful scrutiny of source code in the light of static heap profiles showing cell producers and constructions. But heap profiling by *retainer* so often points directly to an offending function that it seems well-worth the extra implementation effort required.

It may not be helpful to identify a low-level auxiliary used in many places as a major retainer. An extreme example: in our first implementation of retainer profiling 'apply' cells were acceptable retainers! Excluding them made the retainer profile far more informative. More generally, the option to exclude a specified group of functions as candidate retainers is an important enhancement of retainer profiling. It provides a form of aggregation, or inheritance, so that memory costs can be associated with high-level components of an application, rather than with the low-level auxiliaries they use. As with producer profiling, another useful form of aggregation in retainer profiling would be to treat each *module* as one unit, representing in the same band all the memory retained by functions defined in the same module.

Because the band labels in a *lifetime* profile are *not* names of components that can be located in the source program, the usefulness of this form of profiling is not so much to locate space faults as to provide auxiliary information about them. For example, lifetime profiling can reveal possible dragging (or confirm that no significant dragging remains). The lifetime values of major bands can then be used to specify further profiling of a restricted cell population. Lifetime profiles would be enhanced by the addition of *usage* information for long-lived cells: how are the points at which cells are *actually used* distributed across their lifetime? For example, one facet of this information might be *dragging time*, the time between the last use of a cell and its eventual removal from the graph, which could be determined by post-processing in much the same way as lifetime. Another dynamic cell attribute relating to usage is the longest interval between successive uses: where this is large, it might be better to recompute than to store a result.

Since the first heap profiler was introduced, there have been advances in compilation methods for lazy functional languages to remedy some of the space faults that heap profiling revealed. Eventually, improved methods of functional language implementors and programmers may mean that instruments for measuring memory costs 'after the fact' are rarely needed. For the time being, however, even expert functional programmers using the best available compilers are not imune to space faults. There is still much to learn about lazy functional computations; and more powerful heap-profilers are tools for discovery.

Acknowledgements

We are grateful to John Hughes, Thomas Johnsson and other users of the original 2D heap-profiler for complaints and wishful remarks that prompted us to explore the use of further dimensions. We are also indebted to Simon Peyton Jones and David Wakeling for their helpful comments and suggestions on an earlier draft of this paper.

References

[CCP95] C. Clack, S. Clayman, and D. Parrott. Lexical profiling: theory and practice. *Journal of Functional Programming (to appear)*, 5(3), 1995.

[Dar78] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[HV88] P.H. Hartel and A.H. Veen. Statistics on graph reduction of SASL programs. *Software — Practice and Experience*, 18(3):239–253, 1988.

[MW85] Z. Manna and R. Waldinger. *The logical basis for computer programming.* Addison Wesley, 1985.

[Par93] W. Partain. The **nofib** benchmark suite of haskell programs. In John Launchbury and Patrick Sansom, editors, *Proc. 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer–Verlag, 1993.

[Pey87] S.L. Peyton Jones. *The implementation of functional programming languages.* Prentice-Hall, 1987.

[RE95] C. Runciman and D. Wakeling (Eds.). *Applications of functional programming.* UCL Press, 1995.

[RGH78] G.D. Ripley, R.E. Griswold, and D.R. Hanson. Performance of storage management in an implementation of SNOBOL4. *IEEE Transactions on Software Engineering*, SE-4(2):130–137, 1978.

[Röj94] Niklas Röjemo. nhc: a space-efficient haskell compiler (DRAFT). In J. R. W. Glauert, editor, *Implementation of Functional Languages*, pages 30.1–30.29. School of Information Systems, Univ. of East Anglia, Norwich NR4 7TJ, UK, Sep 1994.

[RW91] C. Runciman and D. Wakeling. Problems and proposals for time and space profiling of functional programs. In S.L. Peyton Jones, G. Hutton, and C.K. Holst, editors, *Proc. 1990 Glasgow Workshop on Functional Programming*, pages 237–245. Springer–Verlag, 1991.

[RW93a] C. Runciman and D. Wakeling. Heap profiling of a lazy functional compiler. In John Launchbury and Patrick Sansom, editors, *Proc. 1992 Glasgow Workshop on Functional Programming*, pages 203–214. Springer–Verlag, 1993.

[RW93b] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[San94] P.M. Sansom. *Execution profiling for non-strict functional languages.* PhD thesis, Department of Computing Science, University of Glasgow, September 1994.

[SP95] P.M. Sansom and S.L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL'95)*, pages 355–366, January 1995.

[Spa93] Jan Sparud. Fixing some space leaks without a garbage collector. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 117–122. ACM Press, June 1993.

[SW67] H. Schorr and W. Waite. An efficient machine-independent procedure for garbage collection. *Communications of the ACM*, 10(8):501–506, August 1967.

[Wad90] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, pages 231–248, 1990.