Heap profiling for space efficiency

Colin Runciman and Niklas Röjemo

Department of Computer Science, University of York,
Heslington, York, YO1 5DD, UK
(e-mail: {colin,rojemo}@cs.york.ac.uk)

1 Introduction

Excessive requirements for memory space have in the past hindered or even prevented otherwise attractive applications of functional programming. Although this could be blamed in part on space-hungry implementation methods, in most cases it would have been possible to cut memory requirements very significantly by making a few changes to the source program. But there were no tools to help programmers make appropriate changes. Usage of heap memory was reported only as a total volume of allocations; there was no way to investigate how different parts of a program made demands on heap memory — something which may be far from apparent in the source of a sizable program making use of lazy evaluation and higher-order functions. Finding the appropriate place to make a space-saving change could be very difficult.

In the last few years, there has been a renewed effort to provide appropriate profiling tools for functional programmers. Because the first aspect of performance that many software developers are concerned with is speed, many profiling tools are designed mainly to account for the use of execution time by different program components. But for current functional programming systems, memory-use is often more critical than processor-use. Memory-saving improvements typically save time too, whereas the reverse is less often the case. At any rate, our concern here is with profiling the use of heap memory.

A brief review of the development of memory-profiling systems for functional languages will be given towards the end of these notes. Apart from that, we concentrate throughout on explaining the concepts and use of the latest version of our own heap-profiling tools, as implemented in the `nhc` Haskell compiler. Perhaps this seems a little narrow-minded! By way of explanation: first, so far as we know the `nhc` profiler is the most advanced memory-profiling system currently available for a lazy functional language such as Haskell[1]; secondly, it suits our aim to give a practical tutorial with a series of worked examples and exercises. (We hope that all readers, like the participants at the Summer School, will have access to a computer system with `nhc` installed. Everything necessary can be obtained by FTP from `ftp.cs.york.ac.uk` under the directory `nhc`.)

[1] *Pace* Glasgow! Their `ghc` compiler can profile both space and time, with *cost centres* to help localise faults, but has little support for classifying heap contents according to the dynamic characteristics of memory cells.

2 How to obtain `nhc` heap profiles

To profile the heap usage of a Haskell program `prog.hs` using the `nhc` compiler, there are three steps.

1. *Compile the program.* Compilers that support heap-profiling typically require additional compile-time flags to request an executable with the potential to collect heap-profiling data at run-time. Making profiling an optional extra is usually appropriate because it does slow down the program. However, the version of `nhc` to be used in conjunction with these notes compiles all programs for heap profiling by default.

2. *Run the program.* Heap-profiling data of various kinds can be obtained by selecting an appropriate combination of run-time flags. At regular intervals, a *census* of the heap is taken, and profile data is written to a file `prog.hp`. By default, a census is taken every time a multiple of the heap size has been allocated. To request a different interval we use either `-i`*size*`b` where *size* is the number of bytes allocated between censuses, or `-i`*time*`s` where *time* is the number of seconds between censuses.

3. *Post-process the profiling data.* The file `prog.hp` now contains the information we want, but not in a form that is easy to understand. The program `hp2graph` transforms `.hp` files into readable graphical charts (in PostScript by default). A command such as `hp2graph prog.hp` actually creates *two* new files, `prog.ps` and `prog.aux`. The former is a single-page PostScript file in which a graph of the live heap over time is automatically scaled to fill the page. The `.aux` file is useful if we wish to produce more than one graph on the same scale, for comparison purposes: `hp2graph -p`*old*`.aux` *new*`.hp`.

Steps 2 and 3 may be repeated several times, to obtain a variety of profiles. This may lead to a revision of the program, and restarting from step 1.

2.1 An example: profiling the `xref` program

Take as an example the `xref` program shown in Figure 1. The program reads text from the standard input, and writes on the standard output an ordered index to aid cross-reference; it lists all words in the input, and for each word the numbers of lines on which it occurs. The program is compiled by the command:

```
nhc -o xref xref.hs
```

To obtain profiles of the program running, we need to some test input. We shall use a 215-line file `fplang3` containing the first three messages sent to the 'FPlang' mailing-list (the start of discussions that led to the design of Haskell).

```
main = readChan stdin abort $ \input ->
       appendChan stdout (xref input) abort done

data Index = Empty | Branch Index String [Int] Index

xref cs = disp (inx 1 cs Empty) ""

inx :: Int -> String -> Index -> Index
inx n [] t        = t
inx n ('\n':cs) t = inx (n+1) cs t
inx n (c:cs) t
    | isAlpha c = case span isAlpha cs of
                  (alphas, etc) -> inx n etc (enter (c:alphas) n t)
    | otherwise = inx n cs t

enter w n Empty = Branch Empty w [n] Empty
enter w n (Branch l k ns r)
    | w < k     = Branch (enter w n l) k ns r
    | w > k     = Branch l k ns (enter w n r)
    | otherwise = Branch l k (n:ns) r

disp :: Index -> String -> String
disp Empty             = id
disp (Branch l k ns r) =
    disp l .
    (k++) . (": "++) . dispNos ns . ('\n':) .
    disp r

dispNos :: [Int] -> String -> String
dispNos []     = id
dispNos (n:ns) = dispNos ns . (' ':) . shows n
```

Fig. 1. The `xref` program.

Producer profile

Various run-time options request different forms of heap-profile. One of the most straightforward is a *producer profile*: it characterises cells in the heap by identifying which function produced them — ie. directly caused them to be allocated. A producer profile is requested by the `-p` option:

```
xref -p < input
```

We can now apply `hp2graph` to create the PostScript file `xref.ps` (and scaling information in `xref.aux`).

```
hp2graph xref.hp
```

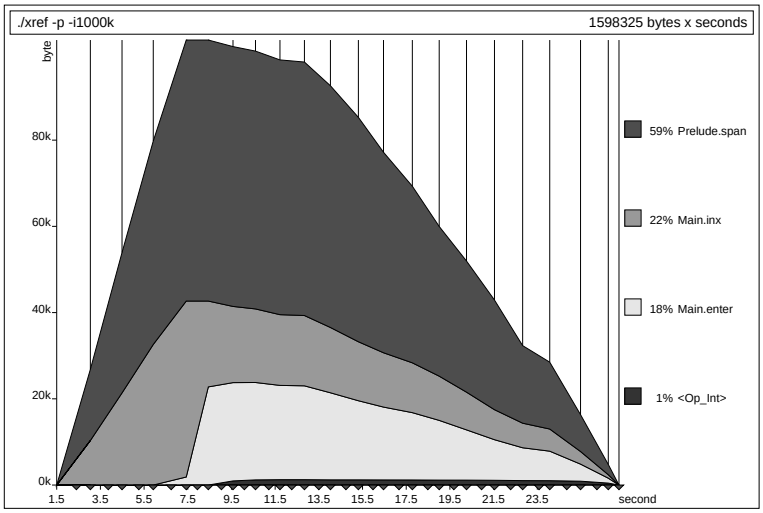

Fig. 2. A producer profile for `xref`.

The result of interpreting the PostScript in `xref.ps` is shown in Figure 2. Ignore the banding for the moment. Concentrate instead on the overall shape of the graph. The graph illustrates how the amount of *live heap memory* (vertical axis) varies over time (horizontal axis). In this example the live heap grows rapidly to over 100 kb before it starts to decrease. Although the collection of profiling data increases a program's memory requirements, this is *not* reflected in a heap profile — so the profile provides an accurate indication of how much heap memory is needed by the unprofiled program. The total *execution* time was slightly more than 25 seconds. Neither garbage-collection time nor heap-profiling overheads are included in this figure, so the wall-clock time for the profiling run may be much longer than the time shown in the heap profile. Also, execution times shown in heap profiles *cannot* be taken as accurate for unprofiled programs — though when comparing two programs, the *ratio* of their execution times is reasonably accurate.

The thin vertical lines mark the times when a census of the live heap was taken. Linear interpolation is used between censuses, which can be misleading for some programs. In doubtful cases the program can always be run again with more frequent censuses; there cannot be a hidden spike of more than 20 kb between censuses if the interval is `-i20kb`, for example. The triangles below the time axis mark garbage collections: every heap census forces a garbage collection, but there may also be collections at other times.

The title line contains the command line used when running the program (unfortunately without any redirections), and the total area of the graph. The latter can be viewed as a measure of the *overall cost* of the program in byte×seconds.

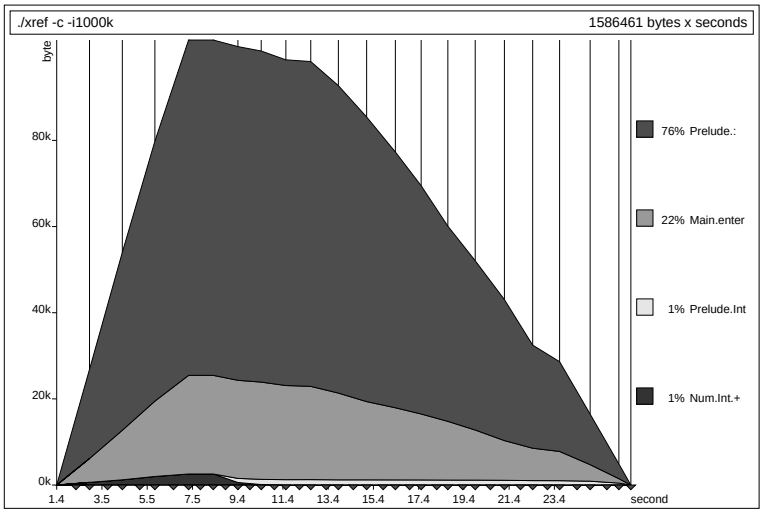

Fig. 3. A constructor/closure profile for `xref`.

To the right is a list of keys, describing what each shaded band in the graph stands for. The example in Figure 2 is a *producer profile* so there is a shaded band for each different program component that allocates memory. In our example, the function `span` in the module `Prelude` was the main allocator of heap memory. The percentages indicate how much of the overall cost is attributed to each key. Percentages are rounded to the nearest whole number, so they do not always sum to 100. The key `<Op_Int>` marks memory allocated by primitive functions working on machine words. If there are more distinct keys in a heap profile than can fit in the graph, then the keys with smallest area are collapsed into an OTHER key.

It is possible to obtain a coarser form of producer profile, in which producers are whole *modules* by using the command `xref -m < input`. This can be very useful for large programs, but would in our example only join `Main.inx` and `Main.enter` into one key, `Main`.

Constructor/closure profile

A useful complement to a producer profile (and in older heap profiling systems the only other possibility) is a *constructor/closure profile*. After the run:

```
xref -c < input
```

`hp2graph` produces the chart shown in Figure 3. The overall shape and cost are *almost* the same as before — they differ very slightly because of small variations in timing. Now each key represents a different kind of cell: a constructor profile

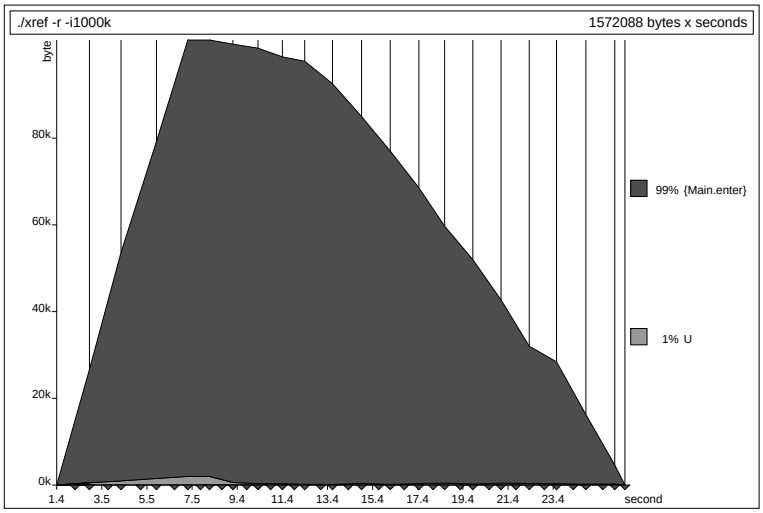

Fig. 4. A retainer profile for `xref`.

characterises a heap cell by asking 'What is it?'. In our example 76% of the overall cost is accounted for by 'cons' cells (`Prelude.:`). Perhaps surprisingly, there are no cells representing the constructors of the `Index` tree: we shall return to this observation in §4.5.

Names of functions other than constructors can appear among the keys, indicating that memory is used for unevaluated closures. For example, Figure 3 includes a band for closures of the function `Main.enter`. Closures of functions that are defined as part of a type-class instance are indicated by a three-part name of the form *class*`.`*instance*`.`*method*. For example, 1% is allocated to closures of the `+` method for the `Int` instance of the class `Num`.

Retainer profile

A *retainer profile* characterises each heap cell according to the program components that have immediate access to it, perhaps as part of a larger data structure. After the command:

```
xref -r < input
```

`hp2graph` generates the retainer profile shown in Figure 4 showing that closures of `enter` retain almost the entire heap. Notice that the keys are now sets. A heap cell can be retained by closures of more than one function due to sharing. By default all sets with more than one member are collapsed into one set. By abuse of terminology this is called the *universal* set `U`. To split the `U`-band, give an integer after the `r` flag: for example, `xref -r3 < input` distinguishes

retainer sets up to size 3. Sets larger than the given limit are still combined into a universal set.

Program components that can occur as retainers are of two kinds. First there are functions whose applications, until fully-evaluated, may retain heap cells in their arguments. Secondly, there are so-called *constant applicative forms* (or CAFs): named data structures defined at the top-level of the program.

Biographical profile

The final kind of heap profile provided by `nhc` is a *biographical profile*, which characterises heap cells according to their past, present and future usefulness in the computation. After running the xref program by:

```
xref -b < input
```

`hp2graph` gives the profile shown in Figure 5.

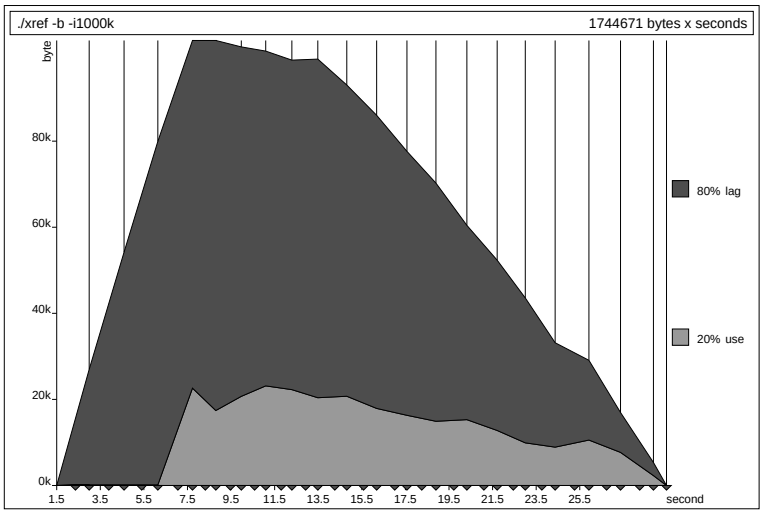

Fig. 5. A biographical profile for `xref`.

In general, a biographical profile splits the heap into four bands (although only two appear in our example). Cells that are never used are *void*. If a cell is used then it passes through three phases: it is *lag* between creation and first usage, *use* between the first and the last usage, and finally *drag* between the last use and its destruction.

Cell used in the past?	*yes*		*no*	
Cell used in the future?	*yes*	*no*	*yes*	*no*
Cell phase:	use	drag	lag	void

A cell is considered *used* when it is scrutinised in a case, or taken part in a primitive operation, or updated with its result (only possible for function closures).

One big difference between the biographical profile and the others is the lack of references to the source code. Other profiles have keys that identify parts of the source program, but the keys in a biographical profile are just cell phases. However, the unique advantage of a biographical profile is that it may directly point to an apparent waste of heap memory in the *drag* and *void* bands, and perhaps in the *lag* band too — the `xref` program, for example, seems to create a lot of heap cells long before they are needed (see §4.5).

2.2 Choosing the right profile

Often one resorts to a heap-profiler with a question already in mind, and that determines the kind of profile needed. For example, the question 'Which is taking up more heap-space, the lists or the trees?' would be answered by a constructor/closure profile.

But what about the outset of a general examination of space-efficiency in a program? Which kind of profile should be looked at first?

One tactic is to obtain all four, and then proceed by investigating the most puzzling of the large bands or spikes. However, we have found that a biographical profile often gives the most useful overall picture of heap-use, without identifying specific program components. A very large percentage of drag, void or lag is often the symptom of a space-fault, and the offending category of cells should be studied further by one of the profiling methods that identifies program components. Such investigations of *parts* of the heap only are made by specifying *restrictions* among the run-time profiling options. Examples will be seen in §4, and summary tables of available options are given in an appendix.

There is an important *caveat* to the tactic of beginning with a biographical profile: even if it shows that *all* cells in the live heap are in their use phase, this is *not* a proof of an efficient program. The program might just use a poor algorithm which re-evaluates everything several times!

3 Causes of space-inefficiency

Before we look any further into the space-efficiency of particular programs, we next give a brief statement in general terms of the typical sources of space-inefficiency in functional programs. We can divide these into three categories.

1. *Degree of evaluation.* An often-made criticism of lazy evaluation is that without very careful programming it can lead to a large number of unnecessarily suspended computations taking up a lot of heap space. The criticism is valid, and one use of a heap-profiler is to check whether such a fault is present so that, if so, it can be remedied. However, it is equally the case that eager evaluation can lead to unnecessary computations whose *results* fill the heap. Even in a 'lazy' language one can define functions in a way that makes them unnecessarily strict.

2. *Degree of sharing.* Sharing is another two-edged sword. One might think that shared reference to a single structure would always give better space-efficiency than reference to multiple versions of the same value. But the 'same value' when shared must also be evaluated to the same extent: once evaluated because of the demands of one reference, a large shared structure must remain in heap pending its access by other references. By contrast, the pattern of demand in the evaluation of an unshared structure may allow it to be traversed in constant space (unevaluated 'in front' of current references, and garbage-collected 'behind' them).

3. *Representation and algorithmic choices.* These choices are often closely connected with the two previous issues, but can also stand alone as sources of space-inefficiency. Even in the stylised world of sum-of-products data types, there are more-compact and less-compact alternatives. The effect of an algorithm choice on space-efficiency often has to do with the *order* in which parts of the computation are done: advancing or delaying the use of a component function may be one way to avoid long-lived large structures, for example.

There is no fixed association between each source of space-inefficiency and a single kind of profile by which it is detected. A degree-of-evaluation problem, for example, might be apparent either in a high proportion of closures in a constructor/closure profile or in a large lag component in a biographical profile. However, only retainer profiling yields direct information about sharing.

It is beyond the scope of these notes to discuss *implementation* aspects of space-inefficiency in any detail, though the space-efficiency of some functional programs is certainly compiler-dependent. *Beware!*

4 A series of examples

In this the major section of the notes we shall examine various aspects of heap profiling and space efficiency as they arise in a series of example programs. Like `xref`, all the programs are small utilities that take text files as input; some are loosely based on UNIX tools. Profiles throughout are of runs with the 215-line file `fplang3` as input.

4.1 The `cat` program

We begin with something very simple that already makes efficient use of heap space. The `cat` program of Figure 6 prints its input as its output. Figure 7 shows a heap profile of `cat` running with the 'FPlang' sample as input. We note with pleasure that `cat` runs using a small and almost constant amount of heap-space — about 200 bytes. What exactly do those 200 bytes contain? The constructor/closure profile of Figure 7 gives the answer: about half the space is occupied by cons (`:`) cells forming the spine of the lazy list of characters representing the text. As the input is lazily demanded for consumption as output, each

```
main = readChan stdin abort $ \input ->
       appendChan stdout (cat input) abort done

cat = id
```

Fig. 6. The `cat` program.

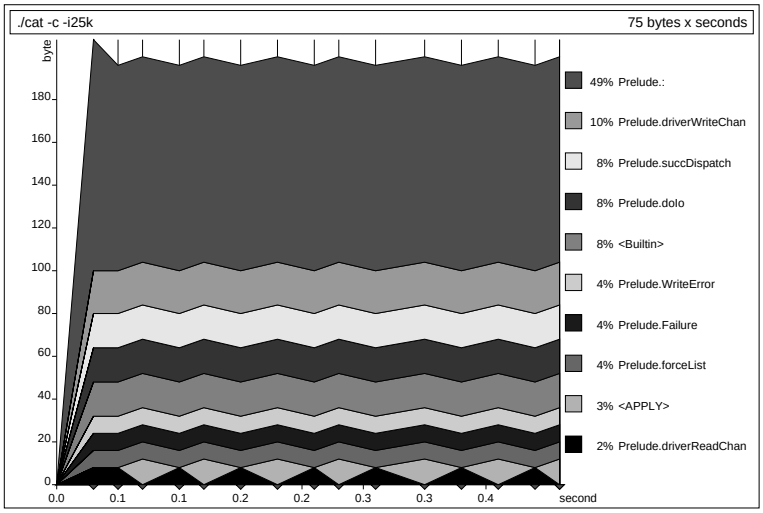

Fig. 7. A constructor/closure profile of `cat`.

cell is short-lived and only a tiny fraction of the spine for the whole text is present in the heap at any time. The rest of the live heap is accounted for by primitive auxiliaries such as `Prelude.driverReadChan` and `Prelude.driverWriteChan` that deal with input and output. Individual characters do not appear: there is exactly one copy of each distinct character constant, stored outside the heap.

Exercise The function `lines :: String -> [String]` breaks text into a list of lines; the inverse function is `unlines`. So one should be able to replace `id` in the `cat` program by `unlines . lines`. Both functions are defined in the Haskell prelude, but try defining your own equivalents. Check that their composition can be used in `cat` without upsetting its space behaviour (apart from a modest constant overhead).

4.2 The `trail` program

Our next example can be viewed as a refinement of the `cat` program. Instead of the *whole* of the input, the output of `trail` is only the trailing `n` lines of input,

where `n` is an argument to the program[2].

One popular way to approach such a problem is to decompose it into a functional pipeline. The auxiliaries `lines` and `unlines` let us transfer between lists of lines and lists of characters. The asymmetric cons-list representation makes it awkward to extract `n` lines from the *back* of a list. But to extract `n` items from the *front* is straightforward using the auxiliary `take`; so let's reduce our problem to this case by applying `reverse` to both input and output. Hence we obtain the program in Figure 8.

```
main = readChan stdin abort $ \input ->
       getArgs abort $ \[n]->
       appendChan stdout (trail (read n) input) abort done

trail :: Int -> String -> String
trail n = unlines . reverse . take n . reverse . lines
```

Fig. 8. A prototype `trail` program.

But is it space-efficient? Applying `trail -b 50` with `fplang3` as input gives the biographical profile in Figure 9. The live heap grows to about 100 kb. The bulk of it is occupied by void cells, with a significant minority in the lag phase. Seeking an explanation in terms of program components, we apply retainer profiling to each of the void and lag components. The profiles show that void cells are retained by `reverse` (and subsequently `take`) on the input side (Figure 10); and similarly that the lag cells are retained by `reverse` (and subsequently `unlines`) on the output side.

The use of a recursively-strict function over a data structure in a compositional pipeline is a standard pitfall. It negates the space-efficiency of lazy evaluation by forcing an unbounded amount of data structure into live heap. The `reverse` function is a classic example. There are various techniques for avoiding this sort of problem. A change of representation is not an option for the `trail` program. But a revised algorithm can make simultaneous access to a whole structure and some part(s) of it, as yet unevaluated, in order to combine structure-creating computations with structure-discarding ones. In conventional terms, we maintain *two* pointers into the file. See Figure 11 for the new program. Figure 12 shows a profile with the same input and scale as before. It might seem that the program still has unduly large void and lag components, but this is not the case. The program maintains a bounded buffer containing the required number of lines of text. The buffer contents are void until the final section of the input is reached; this final section builds up as lag in the buffer until it is printed.

[2] So it is a version of the UNIX utility `tail`, but renamed to avoid a clash with Haskell's prelude function `tail` that extracts the tail of list.

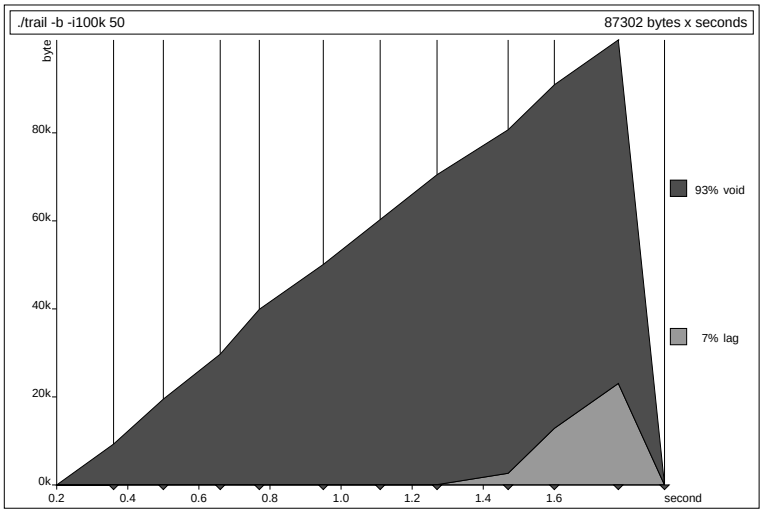

Fig. 9. Biographical profile of the `trail` prototype.

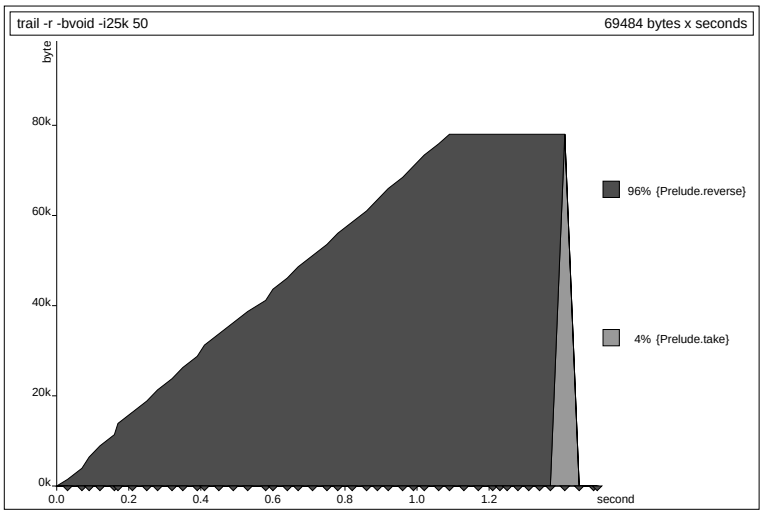

Fig. 10. A retainer profile for the heap void of the `trail` prototype.

```
main = readChan stdin abort $ \input ->
       getArgs abort $ \[n] ->
       appendChan stdout (trail (read n) input) abort done

trail :: Int -> String -> String
trail n = unlines . trail' n . lines

trail' n xs = tandem xs (drop n xs)

tandem xs [] = xs
tandem (x:xs) (y:ys) = tandem xs ys
```

Fig. 11. An improved version of `trail`.

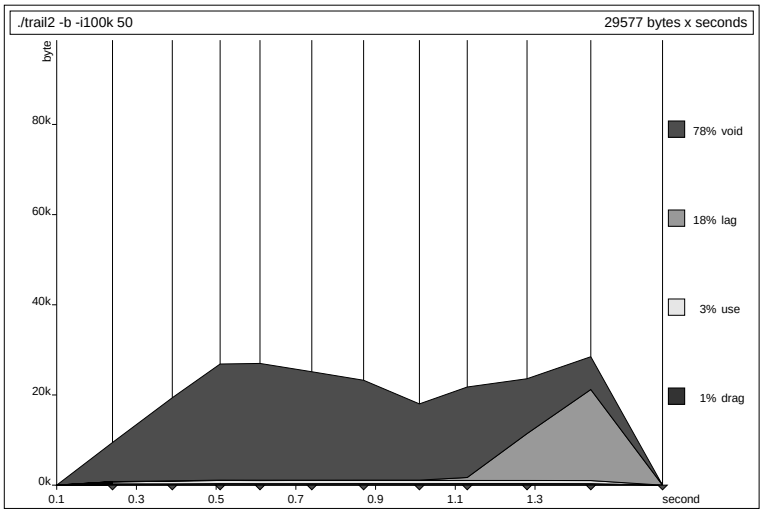

Fig. 12. An improved `trail` profile (cf. Figure 9).

One last puzzle: why does live-heap size dip in the middle of the computation? Because the author of the second contribution to 'FPlang' wrote in short lines!

4.3 The `maxw` program.

The `maxw` program lists the longest words it can find in its input, in alphabetical order.

Again we can tackle the problem as one of composing appropriate elements in a pipeline, keeping in mind that because of lazy evaluation the computations of pipeline elements do not necessarily occur in strict sequence. A prototype pro-

```
main = readChan stdin abort $ \input ->
       appendChan stdout (maxw input) abort done

maxw = unlines . reverse . maxw' [] 0 .
       sortUniq . filter (all isAlpha) . words

maxw' mws _ []              = mws
maxw' mws m (w:ws) | n >  m = maxw' [w]     n ws
                   | n == m = maxw' (w:mws) m ws
                   | n <  m = maxw' mws     m ws
    where n = length w

sortUniq = foldr insertUniq []

insertUniq x [] = [x]
insertUniq x (y:ys) | x <  y    = x : y : ys
                    | x == y    = y : ys
                    | otherwise = y : insertUniq x ys
```

Fig. 13. The prototype `maxw` program.

gram is shown in Figure 13. From the input we extract words: using the standard `words` auxiliary we actually obtain more than that — it gives all maximal sequences of non-space characters — so the next element is a `filter` selecting the alphabetic strings only. A `sortUniq` at this stage also enables duplicates to be discovered and discarded. A list of the longest words is extracted and `unlines` prints them one-per-line.

Exercise The prototype `maxw` program is far from space-efficient: can you predict the shape of its heap profile? There is a remedy without any major rewriting of the program; only a couple of minor changes are needed. With the aid of the heap profiler, obtain and verify much-improved space-efficiency (and speed) in a revised version of the program. Is the revised program better for *all* inputs?

4.4 The `wc` program

The UNIX `wc` (for word-count) program does slightly more than its name suggests. It actually counts the numbers of lines, words and characters in its input, which may be specified as named files. Our version will do the same, but for the standard input only.

Figure 14 shows a naive prototype, in which the three required counts are computed by three separate expressions over the input. This is not a space-efficient solution: because of the shared references to the list `cs` of input characters, as the first expression is evaluated the entire spine of `cs` is forced into heap memory. It cannot be discarded as the first computation advances through it,

```
main = readChan stdin abort $ \input ->
       appendChan stdout (wc input) abort done

wc cs = show (length (lines cs)) ++" "++
        show (length (words cs)) ++" "++
        show (length cs)         ++"\n"
```

Fig. 14. A naive `wc` program.

because it will also be needed by the other two. A retainer profile for cons-cells (Figure 15) shows the effect. The auxiliary functions `break` and `dropWhile` are used in the `Prelude` definitions of `lines` and `words`. The two dark spikes occur because in a small number of censuses, `words` had just passed the remainder of the file contents to its auxiliary `dropWhile`: a profile with fine enough intervals would show many more such spikes. *Note*: since the retained spine is used again

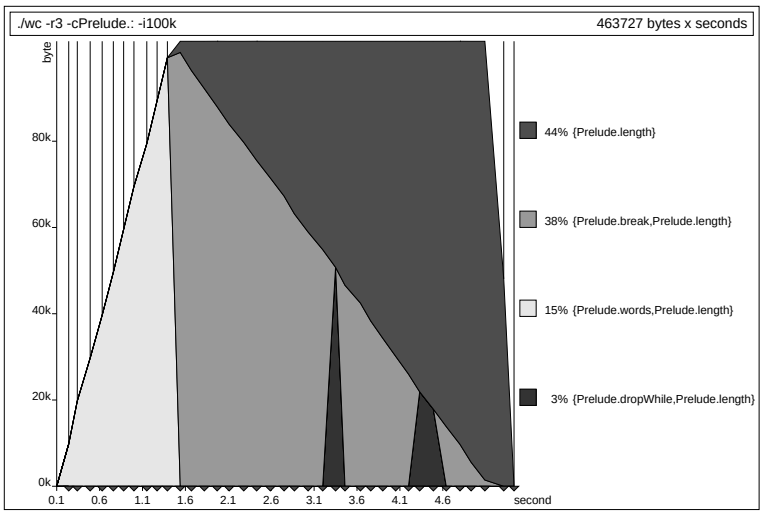

Fig. 15. The cons-cell retainers in naive `wc`.

later, a biographical profile would *not* in this case point to wasted memory.

Aiming for a program that runs in a small constant space for all inputs, we rewrite `wc` as in Figure 16. This program makes a single traversal of the input, checking for both word and line boundaries. The first argument of the tail-recursive auxiliary `wc'` is a triple accumulator for the line, word and character counts.

```
main = readChan stdin abort $ \input ->
       appendChan stdout (wc input) abort done

data LWC = LWC Int Int Int

wc = wc' (LWC 0 0 0) False

wc' (LWC l w c) _ []  =
    show l ++" "++ show w ++" "++ show c ++ "\n"
wc' (LWC l w c) inAWord (x:xs)
    | isSpace x = let l' = if x=='\n' then l+1 else l in
                  wc' (LWC l' w (c+1)) False xs
    | otherwise = let w' = if inAWord then w else w+1 in
                  wc' (LWC l w' (c+1)) True xs
```

Fig. 16. A single-pass `wc` program.

Is *this* program space-efficient? No! A biographical profile shows a heap that grows to a peak size almost twice that of the 'naive' `wc`, only diminishing in the last 10% of execution time. To the nearest 1% the heap is 100% lag! This suggests an accumulation of unevaluated closures, confirmed by a constructor/closure profile for the heap lag (Figure 17). Unevaluated additions and other computations from the body of `wc'`[3] fill up the heap. No further profile is needed to determine *where* they accumulate; it can only be in the `LWC` counters.

This is the kind of space-fault that can discourage one from using a lazy functional language, even though the use of laziness can also be elegant and effective. Of course, it can save space to delay the evaluation of a recursively-structured value; the structure may be larger than the closure that computes it. But it never saves space to delay the evaluation of a basic value such as an integer — though it may save time if the integer turns out not to be needed. In the `wc` program, all three integers of the `LWC` accumulator are needed, so they are better evaluated strictly. In Haskell, strict evaluation can be specified by adding `!`-annotations to the `Int` components in the definition of `LWC`.

```
data LWC = LWC !Int !Int !Int
```

Now if we re-run the program and print a profile to the same scale, the live heap almost vanishes, barely exceeding 350 bytes.

4.5 The `xref` program (resumed).

Reviewing the heap profiles of the prototype `xref` program in §2, we make the following observations.

[3] These 'other computations' are actually the conditional increments.

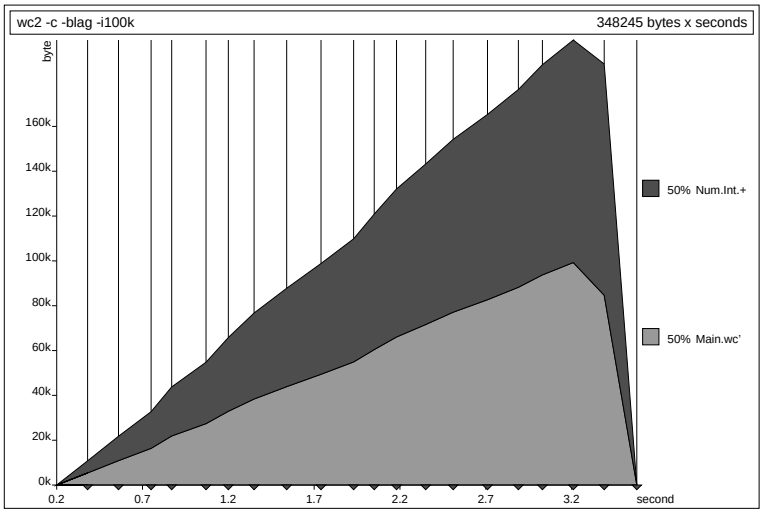

Fig. 17. Lazy accumulation; closures lagging in `wc`.

1. In the constructor/closure profile (Figure 3), as expected there are many cons-cells forming list spines. But there is no sign of constructors for the cells representing the constructors of the `Index` tree; and almost a fifth of the space is occupied by `enter` closures.

2. In the retainer profile (Figure 4) `enter` closures account for the retention of almost the entire heap.

3. In the biographical profile (Figure 5) most of the heap is lag.

It seems that we may be losing space-efficiency because computation is delayed. The function `enter` inserts occurrences of a word into the `Index` tree. But because evaluation is lazy, successive applications of `enter` simply extend a chain of closures. The `Index` tree is only built when the time comes to print it — it does not show up in the heap profile because each piece is discarded as soon as it has been printed.

Our preferred order for the main computational events is clear: as each word is reached in the input text it should be transferred to an entry in the index tree, so that only one copy of each word need be retained. *Continuation-passing* is a standard technique for expressing sequence in functional programs. To ensure that application of a function `f` occurs *after* an `enter` computation is complete, we make `f` an additional argument to `enter`, redefining it like this:

```
enter w n Empty f  = f (Branch Empty w [n] Empty)
enter w n (Branch l k ns r) f
    | w < k      = enter w n l $ \l' -> f (Branch l' k ns r)
    | w > k      = enter w n r $ \r' -> f (Branch l  k ns r')
    | otherwise = f (Branch l k (n:ns) r)
```

Correspondingly, the call to `enter` from `inx` becomes:

```
enter (c:alphas) n t $ inx n etc
```

These changes ensure that `enter` records each word in the `Index` tree before `inx` reads the next. Figure 18 shows a constructor profile of the new program, on the same scale as Figure 3. Overall cost is reduced by about 25%. The `enter` closures are gone, replaced by constructors of the index tree; and there is less list structure representing words of the text. A biographical profile would show

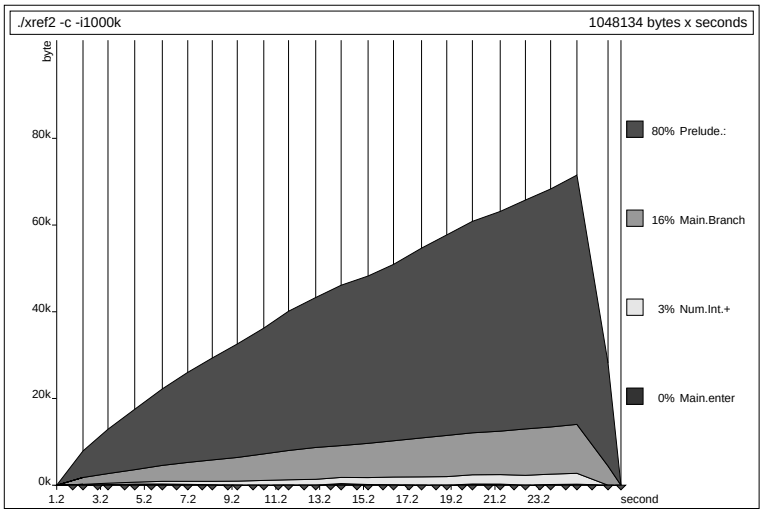

Fig. 18. A constructor/closure profile of improved `xref`.

that there still is quite a lot of lag: much of the `Index` tree, including all the recorded line numbers, will only be used in the final stage of printing.

Exercise As seen in the `wc` example, strictness annotations can also be used to bring computation forward. What is the effect of introducing strictness annotations in the definition of the `Index` type in the original `xref` program? Can this be the basis of a similar improvement in space efficiency?

4.6 The `diff` program.

Our final example is `diff`, a file-comparison program. The problem of listing the differences between files is often tackled using simple rules of thumb — fast to compute but not guaranteed to find the least expression of differences. This version of `diff` gives as output a *minimal* sequence of editing steps that if applied to the first file would make it identical to the second.

The program in Figure 19 is based on one given by Allison [1] to compute distances between strings. Definitions of `main` and `format` have been omitted to fit the program on the page: `main` simply reads two files whose names are arguments to the program, and prints the result of applying `diff` to the file contents; `format` is of type `[Edit]->String`, and constructs the output representation of a sequence of edits.

The central idea is to compute a *matrix* of correction sequences by dynamic programming. For an application `diff file1 file2`, with `M` lines in `file1` and `N` in `file2`, the matrix has rows `0..M` and columns `0..N`. The element at row `m` and column `n` is the minimal correction sequence between lines `1..m` of `file1` and lines `1..n` of `file2`. In particular, the desired result is the sequence at `(M,N)`. The elements of the matrix are computed recursively. As the base case, the correction sequence at `(0,0)` is empty; otherwise at `(0,n)` it adds at the start of `file1` the first `n` lines of `file2`, and at `(m,0)` it deletes the first `m` lines of `file1`. For positive `(m,n)`, if the files differ at those lines, the correction sequence is obtained by appropriately extending a shortest sequence from those at `(m-1,n)`, `(m-1,n-1)` and `(m,n-1)`; if the lines do not differ the correction sequence is just that at `(m-1,n-1)`.

Because a line-by-line simultaneous advance through each file corresponds to a diagonal path in the matrix, the matrix is most conveniently represented as a collection of diagonals. We define the principal diagonal `prince`, and two further lists of diagonals (`uppers` and `lowers`) each ordered by increasing distance from the principal.

Figure 20 shows the biographical heap profile of the program in Figure 19 applied to our usual 215-line trio of mail messages and a corrupted version of them with one change (in line 138). The space consumption is monstrous! A step-by-step investigation by successive heap profiles would be too extensive to include here. But in this case Allison has already made the critical observation in [1]: the matrix should only be computed *by need*. If the two files are in fact identical we need no more than the principal diagonal. More generally, we should be careful not to demand the evaluation of diagonals to any greater extent than is strictly necessary. For just this reason `head` and `tail` are used in the body of `diagTails` rather than deeper pattern-matching on the left-hand side. To improve space-efficiency we must do the same in `diag`, revising the final equation in its definition as follows:

```
diag u ((i,x):xs) ((j,y):ys) w nw n =
    me : diag u xs ys (tail w) me (tail n)
    where
    me = if x == y then nw else min3 (head w) nw (head n)
```

```
data Edit =
    Del Int String Int | Add Int String Int | Cha Int String Int String

diff f1 f2 =
    format (last (diagFor (length xs - length ys)))
    where
    xs,ys :: [(Int,String)]
    xs = zip [1..] (lines f1)
    ys = zip [1..] (lines f2)
    diagFor :: Int -> [(Int,[Edit])]
    diagFor 0         = prince
    diagFor d | d > 0 = lowers !! ( d-1)
              | d < 0 = uppers !! (-d-1)
    prince = (0,[]) :
             diag True xs ys (head lowers) (head prince) (head uppers)
    uppers = zipWith (:) (top ys [])
                         (diagTails True  xs ys (prince : uppers))
    lowers = zipWith (:) (lhs xs [])
                         (diagTails False ys xs (prince : lowers))
    top [] _ = []
    top ((j,y):ys) e = (j,e') : top ys e' where e' = Add 0 y j : e
    lhs [] _ = []
    lhs ((i,x):xs) e = (i,e') : lhs xs e' where e' = Del i x 0 : e

diag _ _           []         _ _  _ = []
diag _ []          _          _ _  _ = []
diag u ((i,x):xs) ((j,y):ys) (w:ws) nw (n:ns) =
    me : diag u xs ys ws me ns
    where
    me = if x == y then nw else min3 w nw n
    min3 (a,ae) (c,ce) (d,de)
      | a < c     = (1+a, (if u then Add i y j   else Del j y i  ): ae)
      | c < d     = (1+c, (if u then Cha i x j y else Cha j y i x): ce)
      | otherwise = (1+d, (if u then Del i x j   else Add j y i  ): de)

diagTails _ _  []     _            = []
diagTails u xs (y:ys) (last:diags) =
    diag u xs ys (tail last) (head this) next :
    diagTails u xs ys diags
    where
    this = head diags
    next = head (tail diags)
```

Fig. 19. A `diff` program.

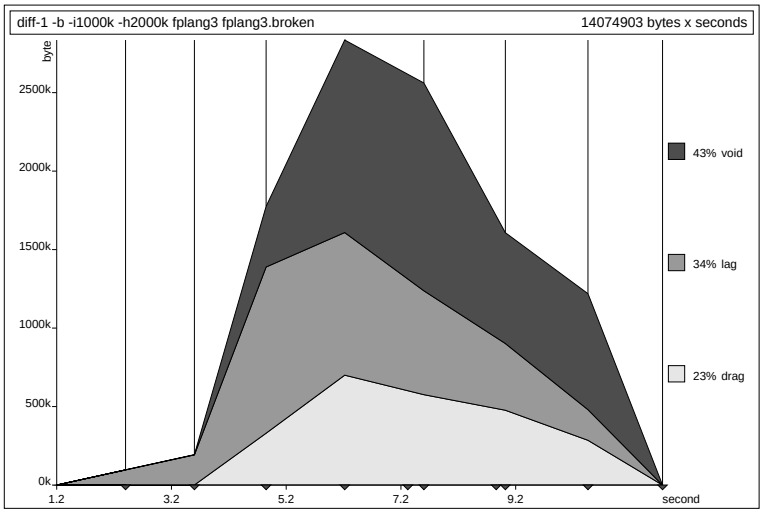

Fig. 20. A monster profile: `diff` at work.

Now if we repeat the previous `diff` application, we obtain the profile of Figure 21 (plotted to the same scale). Making the evaluation lazier has saved a lot of space.

Exercises Though only a fraction of the size of the more eager `diff`'s live heap, the live heap even in the improved version remains large. In part this is an inevitable consequence of insisting on minimal edit sequences, but perhaps there is still scope for improvement.

1. Early in the computation the `length` of each file is computed, forcing the full contents of both files into heap memory, where they remain as they will be needed again. Investigate the effect on heap-use of re-reading the files instead.

2. Use heap-profiling with restrictions to characterise each of the `void`, `drag` and `lag` components of the heap. Can you further reduce the volume of cells in any of these phases?

5 A brief history of heap-profiling

Programming languages with heap-based implementations are hardly new. Yet, whereas many such implementations provide profilers for execution-*time*, we know of very little development and use of tools to examine the make-up of heap memory during a computation. We do not count profilers that measure

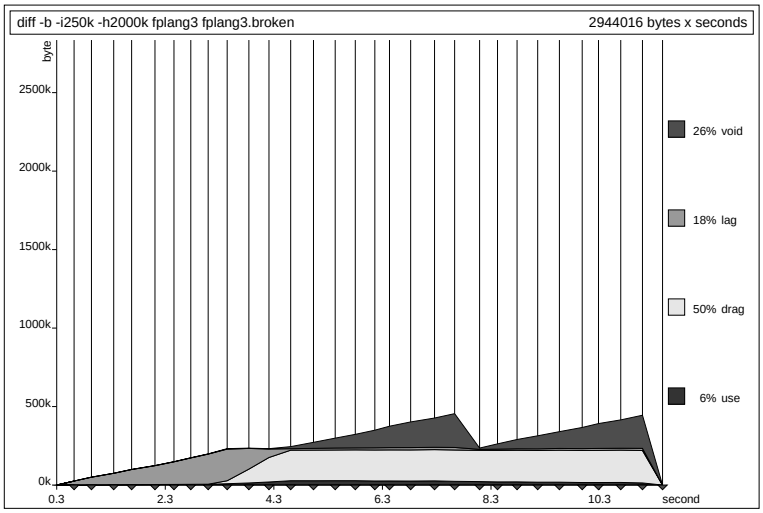

Fig. 21. Profile of a lazier `diff` (cf. Figure 20).

only memory *allocation*: for most applications the allocation count is just another run-time clock; it is no guide to the continuing size and content of the live heap.

The earliest 'true' memory-profiling tools were not for use by programmers wishing to improve their programs. Rather they were developed by researchers who wished to understand better the memory characteristics of *implementation* methods, with a view to improving them. For example, there was a published study of this kind in the late '70s for a SNOBOL4 system[4], and another a decade later for a fixed-combinator implementation of the lazy functional language SASL[3]. The emphasis in both cases was on summary statistics.

The first heap-profiler intended for functional programmers was developed in 1992; its design and use are described in [6]. The profiler recorded census data for static cell attributes only (constructor/closure or producer), and a separate program generated PostScript charts. Extended with whole-module producers and whole-type constructors, the profiler could be used to improve the space-efficiency of large programs, such as compilers [7]. Limitations of a 'who produces what' view of the heap prompted the subsequent development of the `nhc` profiler, with its extensions to dynamic characteristics of heap cells: see [5] for details of retainer profiling[4]; see [9] for more about biographical profiling.

There has also been a concerted effort to develop profiling tools as part of the `ghc` optimising compiler project at Glasgow. Their profiler can attribute both time and space costs to 'cost centres' assigned either implicitly (eg. each

[4] Also *lifetime profiling*, now largely superceded by biographical profiling, but occasionally still a useful source of auxiliary information.

function is a cost centre) or by explicit annotation of expressions — in which case the attributed costs are those expended in evaluating the *entire* expression (excluding free variables or separately annotated subexpressions). For profiling heap-memory cost-centres are treated as producers. See [11] or [10] for details of the `ghc` profiler.

Several other implementors have done work on their own heap-profiling tools, yet to be reported in the literature: for example, we know of such work by Appel (SML of New Jersey), Jones (HUGS/Gofer) and Tofte (ML with region-based memory management).

References

1. Allison, L.: Lazy dynamic-programming can be eager *Information Processing Letters* **43** (1992) 207–212
2. Clack, C., Clayman, S., Parrott, D.: Lexical profiling: theory and practice. *J. Functional Programming* **5** (1995) 225–277
3. Hartel, P.H., Veen, A.H.: Statistics on graph reduction of SASL programs. *Software — Practice and Experience* **18** (1988) 239–253
4. Ripley, G.D., Griswold, R.E., Hanson, D.R.: Performance of storage management in an implementation of SNOBOL4. *IEEE Transactions on Software Engineering* **SE-4** (1978) 130–137
5. Runciman, C., Röjemo, N.: New dimensions in heap profiling. *J. Functional Programming* **6** (1996) to appear
6. Runciman, C., Wakeling, D.: Heap profiling of lazy functional programs. *J. Functional Programming* **3** (1993) 217–245
7. Runciman, C., Wakeling, D.: Heap profiling of a lazy functional compiler. In *Functional Programming, Glasgow 1992*, Springer–Verlag Workshops in Computing (1993) 203–214
8. Röjemo, N.: *Garbage collection, and memory efficiency, in lazy functional languages.* Ph.D. thesis, Computer Sciences, Chalmers Univ. of Technology, Göteborg (1995)
9. Röjemo, N., Runciman, C.: Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proc. Intl. Conf. on Functional Programming (ICFP'96)*, ACM Press (1996) 34–41
10. Sansom, P.M.: *Execution profiling for non-strict functional languages.* Ph.D. thesis, Computing Science, Univ. of Glasgow (1994)
11. Sansom, P.M., Peyton Jones, S.L.: Time and space profiling for non-strict higher-order functional languages. *Proc. ACM Conf. on Principles of Programming Languages (POPL'95)*, ACM Press (1995) 355–366

Appendix: summary of profiling options

Run-time flags

Usage:
***program* profile-kind restrictions heap-size census-interval arguments**

profile-kind:	
`-p`	producer by function
`-m`	producer by module
`-c`	constructor
`-r`	retainer (can be given optional number for maximum retainer-set size)
`-b`	biographic
`-l`	lifetime
Zero or more **restrictions** can then be specified:	
`-p`*comma separated list*	restrict to functions in list
`-m`*comma separated list*	restrict to modules in list
`-c`*comma separated list*	restrict to constructors/closures in list
`-r`*comma separated list*	restrict to nodes for which at least one retainer is in the list
`-b`*comma separated list*	restrict to the phases in the list.
`-l`*min*`-`*max*	restrict to nodes with lifetimes of at least *min*, and at most *max*, censuses — one of the limits may be omitted
Set **heap size**. The prefix `M`(10^6) or `k`(10^3) can be used:	
`-h`*size*`b`	heap size in bytes
`-h`*size*`w`	heap size in words
Set **census interval**. The prefix `M`(10^6) or `k`(10^3) can be used for allocation intervals, `m`(10^{-3}) for timed intervals:	
`-i`*interval*`s`	census interval in seconds
`-i`*interval*`b`	census interval in allocated bytes
`-i`*interval*`w`	census interval in allocated words
Arguments to the Haskell program:	
`-`	stop decoding of run-time flags — only needed if first argument starts with `-`

Hp2graph

Usage: **hp2graph options** *file*.hp

Options:	
-t*percentage*	ignore trace bands — all those bands that together contribute less than the given percentage of overall cost
-p*filename*.aux	use same scale as in the given aux-file
-x	produce an exploded graph
-c	omit profiling comments
-y	omit census lines
-m	omit garbage-collection marks

This article was processed using the LaTeX macro package with LLNCS style