Lag, drag, void and use – heap profiling and space-efficient compilation revisited

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

Abstract

The context for this paper is functional computation by graph reduction. Our overall aim is more efficient use of memory. The specific topic is the detection of dormant cells in the live graph — those retained in heap memory though not actually playing a useful role in computation. We describe a profiler that can identify heap consumption by such 'useless' cells. Unlike heap profilers based on traversals of the live heap, this profiler works by examining cells *post-mortem*. The new profiler has revealed a surprisingly large proportion of 'useless' cells, even in some programs that previously seemed space-efficient such as the boot-strapping Haskell compiler nhc.

1 Introduction

A typical computation by graph reduction involves a large and changing population of heap-memory cells. Taking a *census* of this population at regular intervals can be very instructive, both for functional programmers and for functional-language implementors. A *heap profiler* [RW93] records population counts for different classes of cells at each census. A *producer profile* classifies cells by the program components that created them; a *constructor profile* classifies cells according to the kinds of values they represent. A post-processor generates a graphical summary of heap contents throughout the computation. Census information can be extended to include the active components that retain access to cells (*retainer profile*) and the cells' eventual lifetimes (*lifetime profile*) [RR95].

Such heap profiling is a surprisingly effective tool for discovering space faults. We say 'surprisingly' because there is nothing in a who-produces-what profile (nor even in the deeper structural information of something like a retainer profile) that directly points to some part of heap usage as a likely fault. It is up to the programmer to assess features such as sudden spikes of growth or steadily widening bands. Do these represent behaviour only to be expected of their program? Or do they represent the anomalous behaviour of a space fault?

In other lines of work on memory management, researchers are questioning the conventional acceptance of safe over-estimates of garbage in memory management — see for example [MFH95]. There is an important difference in principle between (a) the time for which a cell is retained simply because it is attached to a 'live graph', and (b) the interval over which the cell is actually needed because it plays a useful role in the computation.

These observations together motivate our goal to identify 'live but useless' memory cells directly in some form of heap profile. We want to discover how great the difference is in practice between the traditional over-estimate of needed cells (i.e. as used for garbage collection) and a true measure of need (i.e. as defined by full knowledge of how and when cells are used).

The main application so far is to halve the memory demands for nhc – a boot strapping Haskell compiler. This is not the first time a heap profiler has been used to improve a compiler for a lazy functional language [RW92]. So what's new?

The compiler in the '92 paper was not designed with space efficiency in mind. The initial version needed 2Mb of live heap data at peak for a 280-line module. It was not too surprising that by constructing a profiling tool we were able to uncover space faults and make significant gains. Overall space×time cost was halved, and peak heap-memory demand was reduced by about 30%.

The compiler examined in this paper, however, was designed for space-efficiency right from the start. Indeed, the profiling tools from '92 were available to help in this. Despite a more complex language (Haskell 1.2 [Hud92] rather than Lazy ML [AJ93, Aug84, AJ89]) compilation of a comparably sized source file (280 lines + 1400 lines of interface files) at the *start* of our profiling exercise in this paper uses only 800kb.[1] So one might not expect substantial reductions in heap space. Yet not only do we again improve the overall space×time cost by nearly a factor of two; the peak memory demand is also reduced by up to 50%. We achieve the 'factor of two' result in a tougher context because we have developed more powerful tools for heap-profiling. By slicing the heap in more ways than just the two-dimensional 'who produces what', many more questions can be answered about the way heap memory is used. Our latest profiler specifically directs attention to apparently wasted heap space.

The remainder of the paper is as follows. §2 makes more

[1] The actual module we shall use as a running example is smaller – 80 lines + 634 lines of interface files – requiring only 400kb at the outset.

precise what we mean by the distinction between live cells and useful ones, concluding with the definitions of terms such as *heap lag* and *heap drag*. §3 explains both in principle and in practice how the techniques of heap profiling can be adapted to obtain a profiler for measuring lag and drag. §4 describes how the space-efficiency of nhc was improved with the aid of the new profiler. §5 discuss the meaning of 'useful heap'. §6 concludes with a discussion of current limitations and future potential.

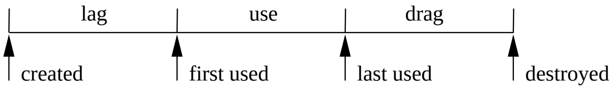
2 Definitions of terms

2.1 The individual cell

The *biography* of a typical cell in heap memory includes four important events:

1. the cell is *created* — it is 'born' as one small piece of the live graph;
2. the cell is *used for the first time* — it is 'employed' in the computation;
3. the cell is *used for the last time* — it goes into 'retirement';
4. the cell is *destroyed* — it 'dies', ceasing to be part of the live graph.

The intervals between these successive events correspond to three phases for the cell: *lag*, *use* and *drag*.



In this paper, we are concerned with whether and when cells are used at all, not with the details of how frequent the uses are. So we choose to ignore the fact that the first and last uses of one cell may be its *only* uses, while those of another cell may be the first and last of many. We make the simplifying assumption that the first and last uses mark the beginning and end of an undivided phase during which the cell is useful.

We are most interested in the lag and drag phases. A cell is in the lag phase from its creation to its first use, and in the drag phase from its last use to its destruction. For the most effective use of memory, both lag and drag phases should be as short as possible. That is, most cells should be useful for most of their lifetime.

Often one or more of the three phases of lag, use and drag will in fact be so short as to be virtually instantaneous. Under a call-by-need regime, we should not be surprised to find that many cells are used as soon as they are created — zero lag. For any cell that is used exactly once, the events of first and last use coincide. And many cells are destroyed as an immediate consequence of their last use — zero drag — even if the memory they occupy is not recycled until the next garbage collection.

What if a cell is *never* used? Then we refer to the interval between its creation and its destruction as *void*.

2.2 Cell populations

These biographical terms applied to individual cells can be extended to collections of cells. In particular, we can apply them to the complete population of cells maintained in heap memory. At *any given moment* during a computation we can divide cells into four classes: lag, use, drag or void. Hence we can define the *instantaneous heap lag* as the amount of heap memory occupied by cells in their lag phase; and similarly for the other cell phases.

The same sort of classification can be applied across the entire period of a computation. Since we can label all of heap memory at any given instant with one of the four phases, we can integrate across all such instants to obtain total space×time cost (e.g. expressed in byte×seconds) for each of the phases. This gives a measure of the overall proportions of *heap use* in comparison with any *heap lag*, *heap drag* or *heap void*.

3 Profiling method

Earlier heap profilers [RW93, San94, SP95, RR95] collected their information by traversing the *live* heap. This is not possible when collecting biographical data. There is no way to determine the phase of all cells on-the-fly in the middle of the computation, without knowledge of how they will be used in the future. The alternative is to record the timing of important events in the cell, and retrieve this information when the cell is eventually removed from the live graph.

3.1 Practical implementation

We now outline an implementation of our biographical profiler, using nhc [Röj95a, Röj95b] as the host compiler.

Every heap cell is enlarged to accommodate three additional pieces of information: (1) creation time; (2) time of first use, if any so far; and (3) time of most recent use, if any. As in some previous heap profilers, time is reckoned by the number of heap censuses that have occurred. So the time of the final event for a cell, its destruction, need not be stored in the (now dead) cell. It is implicit at the next census, when the information is required.

Setting the first time-stamp is easy. To maintain the other two, we must agree what constitutes a 'use' of a cell. In the unoptimised G-machine [Joh84, PJL92] setting of nhc the broad rule is that *use means evaluation*. This includes: (1) case analysis of a data construction; (2) application of a function closure; (3) evaluation as argument in a primitive operation.

In addition, a functional closure is also deemed to be used when it is updated with its result. This choice is open to debate. It was not the rule we implemented at first, but we found it unhelpful to have function closures awaiting update labelled as heap drag.

The previous section explained the need to examine cells *post mortem*. At each census, a summary of all cells destroyed since the last census is written to a log file. The summary comprises a time-stamp for the census, followed by a series of population counts for each distinct triple of time-stamps occurring in the dead cells.

There are two ways that cells can be destroyed: they can be overwritten or they can be disconnected from the graph. Information about overwritten cells is collected just before the update is done; the data is later merged into the next census. Disconnected cells are processed during the census

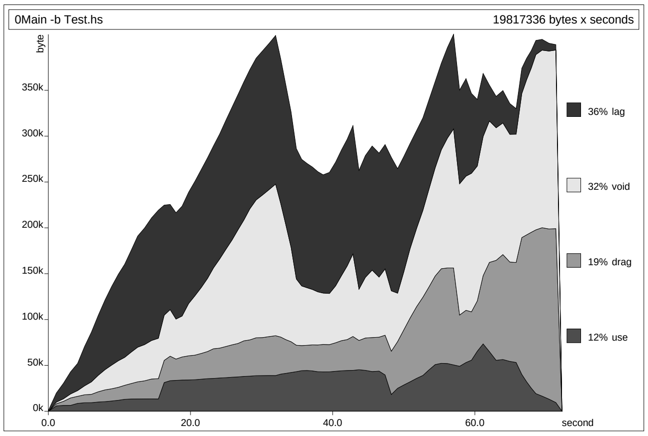


Figure 1. A profile showing the lag, drag, void and use components of heap memory when the original version of nhc compiles a small part of itself — an 80-line module.

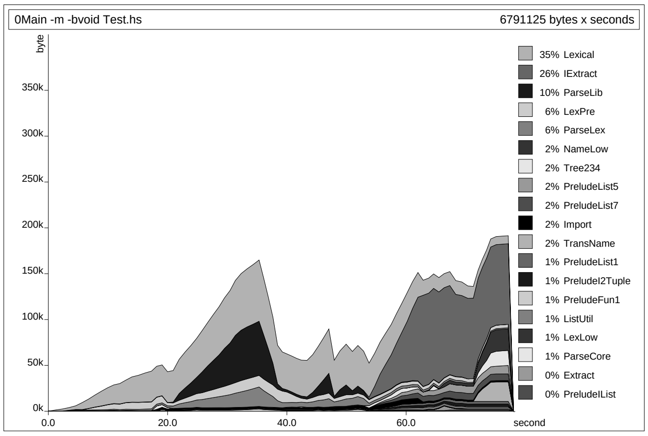


Figure 2. A producer profile showing which modules produce the void cells in Figure 1.

itself. First all live cells are marked (using the mark phase of the garbage collector) and then the heap is scanned. All unmarked cells have been disconnected and are therefore included in this census. Afterwards the heap is compacted to prevent the unmarked cells from being counted at the next census.

The use of variable-sized cells in nhc is a problem when scanning for disconnected cells. For an ordinary garbage collection only marked cells are of interest, and they can be found by searching for set mark-bits. But during a census we must locate the unmarked cells. Our solution is to guarantee that there is no empty space between cells: the next cell always starts immediately after its predecessor. This is true when cells are allocated, but ceases to be the case when a cell is updated with a smaller cell. If this happens a filler cell of the appropriate size must be inserted in the vacant space. The census then finds all dead cells by scanning from the start of the heap, ignoring filler cells and cells with their mark-bits set.

The time needed for censuses is *not* included in the heap profile, but the overhead when overwriting cells is. This is not a serious problem as our main concern is space. Times shown in different heap profiles can still be *compared* to see whether a space optimisation slows down the program or speeds it up.

After the computation is over, a post-processor derives population counts for cells in different phases at each census point. The derived census data is plotted graphically in the time-honoured way. To increase the accuracy of the approximation one simply increases the frequency with which censuses are taken.

3.2 Profiling restricted parts of the heap

The heap profiler must also be able to answer questions like 'What is producing the heap drag?', using biographical information to restrict the scope of a producer profile.

Our implementation of a biographical profiler can only be combined with *invariant* profiles, i.e. those where cells never change their attributes. The current implementation can *not* combine retainers and biographical profiles. There is not enough space to record in each cell the different retainer sets it has during its lifetime, and as we have already observed there is in general no way to obtain the biographical state of a cell in mid-computation. Future work may remove some of these restrictions: see §6.

4 Example application – nhc

The bootstrapping Haskell compiler nhc was written with space-efficiency as the main objective [Röj95a, Röj95b]. Indeed, it can compile itself in well under 3 Mb which compares favourably with other compilers. Yet, as Figure 1 shows, approximately *88%* of nhc's heap memory is lag, drag or void!

4.1 Symbol tables in nhc

As we shall see some of this wasted memory is due to the symbol table. A short description of how identifiers are treated in nhc is therefore necessary.

After a source module is parsed, but before scope is determined, all interface files are read and checked for consistency. Identifiers are entered into a symbol table sorted by their *real names* (the names used when the identifiers were defined). Each identifier is given a *unique number* when it is inserted in this table. After the consistency check, the symbol table is *re-ordered* by *visible* names (after any renaming done during import, see [Hud92] section 5.2.2), and all identifiers in the parse-tree are replaced by their unique numbers. The symbol table is then re-ordered once more, this time by the unique numbers, after which all needed information about identifiers (e.g. arity, type, name) is fetched from the symbol table using the unique number as key.

Once entered in the symbol table an identifier and all its associated information is kept until the compiler terminates.

4.2 Laziness may keep redundant closures

The heap void is a large part of both the peaks in nhc's heap profile (see Figure 1). A profile of *only* the heap void (Figure 2) shows that most of the first peak is produced in

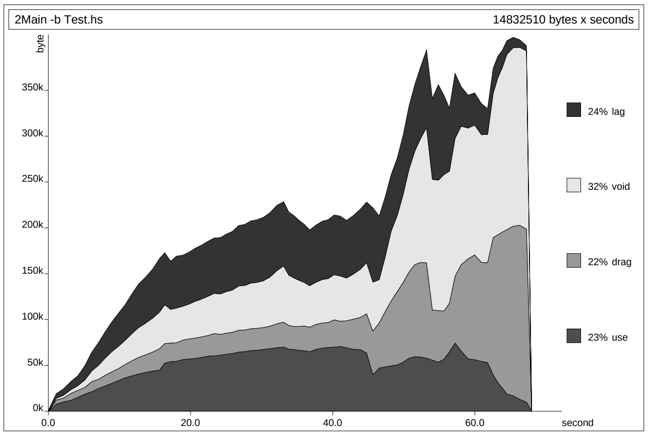


Figure 3. The heap profile after strictifying the parser combinators and promoting the filtering of hidden identifiers.

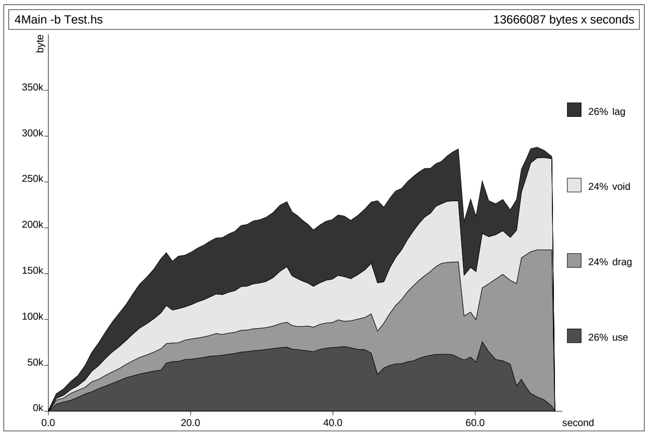


Figure 4. Memoization removes a large part of the peak at the end of evaluation.

the module ParseLib, and most of the second in the module IExtract. Let us start by trying to remove the first peak.

A constructor profile restricted to the heap void produced by ParseLib shows that *all* these cells are binary application nodes! It is the parser combinators in ParseLib that create these binary application nodes. These combinators are written in a continuation-based style with two continuations: one (`good`) is used if the parser succeeds, and the other (`bad`) in case of failure. Using these combinators it is possible to build larger parsers by combining simpler ones (a longer description is available in [Röj95a] chapter 4). Take for example the `ap` combinator. This combinator is defined as:

```
ap :: Parser (a->b) i c ->
          Parser a i c -> Parser b i c
ap x y = \good bad ->
          x (\u -> y (\v -> good (u v)) bad)
            bad
```

In short `ap` takes two parsers (`x` and `y`), applies the first (`x`), and if it succeeds, then applies the second (`y`) with the rest of the input. The result of the combined parser is the result from the first parser (`u`) applied to the result of the second parser (`v`). But this application (`u v`) is *not evaluated* at this point. It will not be evaluated until some function needs the return value of the whole parser later in the compilation. That would account for some *lag* of application cells, but the problem is *void*. We conclude that not all parts of syntax trees that represent interface files are actually needed.

The reason can be found in the source module being compiled. The programmer used *selective import* from the Prelude, i.e. all identifiers defined in the Prelude that are used in the code are mentioned in the import declaration. The compiler can therefore remove all other definitions from the syntax tree of the interface file. The necessary comparisons make use of removed identifiers, but not of their associated type and arity information. It is this information that amounts for much of the heap void. Since the compiler knows which identifiers will be needed *before* it starts reading the interface files, this filtering can be done earlier.

Moving the filter function into the parser solves this problem *if only* the parser combinators evaluate the applications in their return values.

By forcing evaluation of the application (`u v`) in `ap` and promoting the filter into the parser, we flatten the first peak (see Figure 3).

To make this change straightforwardly it is essential that *functional* values can be forced, as the parser combinator has no way of knowing the result type of the application (`u v`).[2]

4.3 Sharing can reduce memory usage

The natural next step is to obtain a constructor profile of cells created in IExtract and never used. However this does not give any useful information: memory is more or less evenly divided among a dozen types. But a producer profile with the same restrictions reveals that 61% of the memory in the profile is produced by the function `uniqueType`. This function checks that all type constructors and type classes used in a type are imported.[3] If so then `uniqueType` returns the type with all identifiers replaced by their unique numbers. A biographical profile of cells produced by `uniqueType` reveals that no less than 81% of the heap occupied by these cells are heap void, and only *1%* is heap use! The large heap void is because many of these types belong to class methods defined in PreludeCore. Haskell 1.2 does not allow identifiers in PreludeCore to be hidden, so the programmer has no way to import from it selectively.

Making `uniqueType` more lazy does not help: too much is already evaluated due to the check for undefined type constructors or classes. Introducing a memo table as an auxiliary argument to `uniqueType` is far more effective as it increases sharing. The void percentage becomes even higher than before (91%), but the total amount of memory used by `uniqueType` is now only one tenth of the amount needed before. The one drawback of this modification is that it slows down the compiler by 4%.

[2] If it is not possible to force evaluation of functional values then two different types of `ap` are needed. It is then up to the programmer to choose the correct one depending on the type of the returned value.

[3] This check is required by the Haskell 1.2 standard.

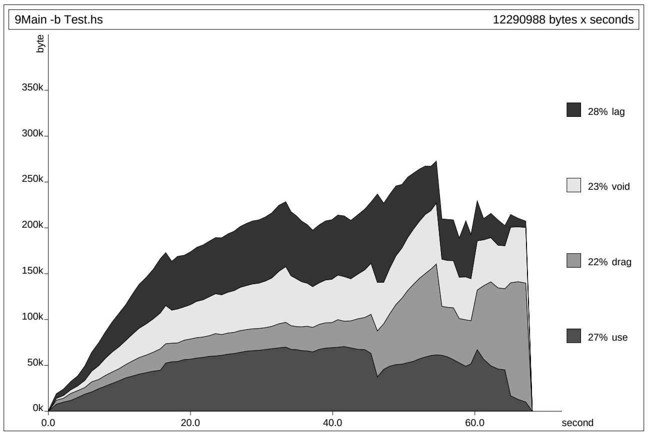


Figure 5. Not creating symbol-table entries for instance definitions of class methods removes the narrow plateau left at the end of evaluation in Figure 4. Most of the bands are even and no obvious point of attack is available. (The bump in the heap drag is due to circular dependencies and not easily removed).

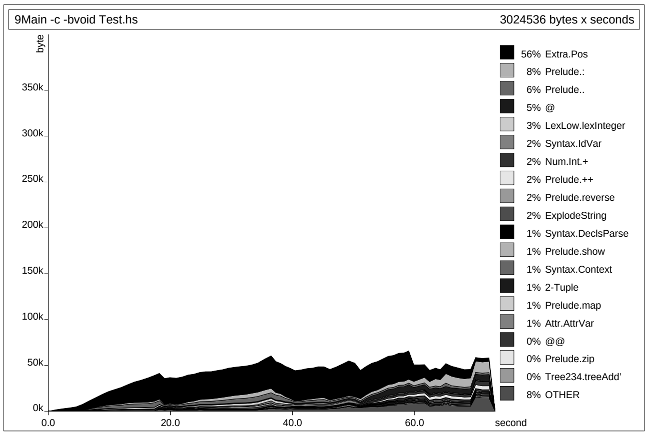


Figure 6. A constructor profile of the void band from Figure 5.

4.4 Sometimes sharing *increases* memory usage

After two attacks on heap void, it is time to tackle the heap drag. Part of the *second* of the two peaks of heap drag in Figure 4 is a consequence of using a symbol table. The symbol table turns into heap drag as each entry is used for its last time.

But the *first* peak has no obvious explanation. By further profiling it is easy to find out that the module Import creates most of the cells that are dragged, with the function `processInterface1` as its main producer. However, looking at the code for `processInterface1` does not explain the large amount of memory produced, or why it is dragged. The function processes the header part of interface files, extracting *renaming* and *fixity* information, and returning mappings from visible names to fixity and real name. We should like to see *why* these mappings are retained, but we cannot obtain a retainer profile restricted by biographical information as explained in §3.2.

Since the definition of `processInterface1` itself looks innocent enough, we suspect a problem in the code that uses its return value. We manually track the values that `processInterface1` returns until the problem is finally found in IExtract. This module re-orders the symbol table from using real names as keys to using visible names instead. It does so by flattening the original symbol table into a list and then rebuilding it with the new sorting order. However, the work done by `uniqueType` happens at the same time, and `uniqueType` needs the table sorted by real names to decide which type constructors and type classes are available, and to find their unique numbers. This sharing means that the original symbol table (sorted by real name) is kept until the new one is completely built! The parts of the symbol table created by the functions that `processInterface1` returns are not used by `uniqueType`, and hence show up as drag in the heap profile. By building a small table with only the mapping needed for `uniqueType`, at the same time as the new symbol table is built, the peak memory usage can be reduced by approximately 30kb. This solution depends on *laziness* as `uniqueType` needs the mapping to build the new symbol table – a circular dependency that makes it difficult to remove the rest of the first drag peak.

4.5 Eagerness may build redundant structures

There is just one unexplained peak left. This is the narrow plateau of heap void created shortly before the end of the execution (again see Figure 4). Our usual tactic of doing a module profile restricted to the biographical data of interest draws our attention to two modules, NameLow and TransName. Profiling producers in these two modules, we find the function `translate` accounts for 40% of the memory. This function does a fix-and-clean job on the symbol table as part of the final preparations before code generation. One of these fixes is to create entries for instance definitions of class methods (e.g. the equality function in the `Int` instance of the class `Eq`). These definitions have not had their own entries before. Their unique numbers have been kept in a list in the entry for the class they belong to. Entries are now needed as the type checker may insert their unique numbers in the syntax tree, and these numbers must be translated into real names before code generation. But the compiled module might only use a few, if any, of these methods. The others are void. Knowledge about which methods are actually used is not available at this point in the compiler. Our remedy is to store both the number of the class and the number of the class method in the syntax tree, instead of the number of the instance method. Then no entries are needed in the symbol table for instance methods.[4] We have now reached the heap profile shown in Figure 5.

4.6 The final(?) squeeze – trading words for bits

Most of the peaks are gone, and those remaining can be explained. However, it would be nice to remove the void band completely.

The constructor profile restricted to heap void in Figure 6 clearly illustrates the problem: 56% of the heap void

[4] For other reasons, instance definitions in the compiled module itself must have symbol-table entries.

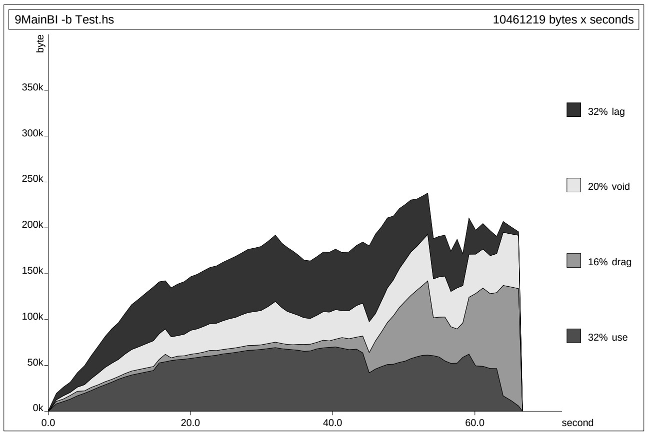


Figure 7. The full profile when position information is encoded in an `Int`.

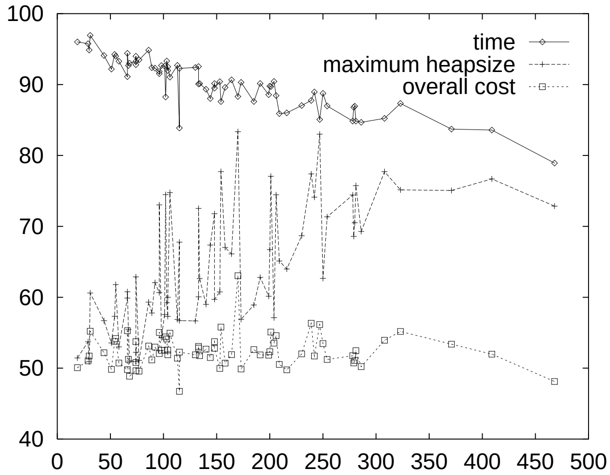


Figure 8. The graph shows what percentages of the maximum heap size, execution time and overall space×time cost of the original compiler are needed by the improved one. The x-axis shows the number of source lines in the compiled modules.

is used by `Pos` constructions, representing lexical position information! This is very depressing as the information is only used if an error message is generated. It is fairly easy to change the coding of position information from the naive `data Pos = Pos PackedString Int Int`[5] to a more compact `type Pos = Int`. File names are encoded as small numbers which are combined with line and column numbers into a 32 bit integer. This limits the number of files, and the maximum line and column numbers, but the pay-off is very good – a 15% reduction in overall cost (see Figure 7).

There is also a reduction in heap drag because many integers were previously used only once, to calculate the next column or line position.

4.7 Have we specialised for one test file?

The previous sections have all used the same test file. However, we obtain a similar overall improvement for *every* source file in the compiler as shown in Figure 8. The curves are very rough but at least two trends are clear. The improvement in time is larger for larger source files, but the fall in maximum heap size is smaller.

One explanation of the smaller percentage reduction in maximum heap size for larger files is that our improvements mostly affect the processing of interface files. When the source module is small the processing of the (very large) interface file for the standard prelude accounts for a large part of the maximum peak in heap space. But for larger source files, type checking creates most of the maximum peak in heap size, and no improvements have been made to the type checker as a result of the profiling described in this paper.

The reason for the greater improvements in time for larger modules is less clear. A reduction in live heap does mean fewer and cheaper garbage collections, especially if the live heap originally used most of the available memory. But we suspect that this is only a partial explanation.

The two curves for time and maximum heap size more or less cancel each other when costs in byte×seconds are considered. The overall improvement in cost is about 50% for nearly all modules. This is the factor of two improvement promised at the start of the paper.

A 115-line module dealing with flag decoding shows the greatest improvement. The main reason is an abnormally large speed-up in the compilation. The code in this module is not typical: one large function (50 lines) that builds a tuple, and 45 selector functions for this tuple.[6] This is very easy code to compile: almost no type checking and no complicated mutually recursive structures. An unusually large part of the time is spent processing interface files. And since our improvements mostly affect the parts of the compiler dealing with interfaces we get a larger than usual speed-up.

The module with the smallest improvement in cost is PPSyntax. This 170-line module is a pretty-printer for the syntax tree. The module consist of many mutually recursive functions with high order arguments. Over 2/3 of the time and more than half of the maximum heap usage is due to type checking. In our running example type checking was never a problem, and hence no improvements were tried. But when nhc compiles PPSyntax, a biographical profile shows that nearly half of the memory is drag (49%!) even after our space-saving modifications.

One immediate conclusion is that our choice of source file did affect our improvements. If we had chosen PPSyntax instead then it is likely that something would have been done about the type checker. Nevertheless, the modifications prompted by profiling the compilation of a single module did result in improvements for *all* other compilations. It is only that some are less improved than others.

[5] The shared packed string is used to store the file name for the position.

[6] If records had existed in Haskell 1.2 then this module would not have been necessary.

5 Discussion

It was an (unpleasant!) surprise to discover that only 12% of the live heap structure in nhc – supposed to be a 'space-efficient' compiler – represented cells currently *in use* for computational purposes. Even in the final compiler with improved space-efficiency the figure only rises to 32%. Of course this depends on our definition of *use*. Perhaps this definition is too harsh or restrictive? Yes and no!

One the one hand, classifying lag cells as 'useless' is indeed rather harsh. They are not useless in the same sense that drag and void cells are. Drag and void cells are truly redundant, though not yet identifiable as garbage: they will never (again) be used in the computation, so they could in principle be reallocated for other purposes. But lag cells are *not* redundant: they will be needed eventually. Lag cells are identified as possible indicators of memory inefficiency only because it might be possible to save memory by delaying their creation. So a more generous assessment of the useful part of the heap might include both use and lag cells: 48% for the initial version of nhc compiling our running example, and 64% for the final version.

On the other hand, one could argue that our figure for use is an *over*-estimate. This is because a cell is regarded as useful throughout the period between its first and last uses, no matter how infrequently it is used in between – or even whether there are any other uses at all.

A fundamental question is if all these problems are due to the use of a lazy language. Maybe we could get rid of our space problems just by using a strict language instead? We don't think so. Although the problem in §4.2 was solved by increasing strictness, all other problems were removed by other means. Increasing sharing (§4.3) or decreasing it (§4.4) could be just as useful in a strict language. By using heap profiles on a lazy language we find problems with lazy languages. Using it on a strict language we would find problems with strict languages too.

6 Conclusions and future work

To aim for negligible amounts of lag, drag and void in heap memory is quite an exacting requirement, especially for large and complex applications. But it could be another useful target as functional languages gradually migrate from large research machines to miniaturised systems. To work towards such a target it is essential to have the tools to measure and assist progress.

In §2 we noted the simplifying abstraction of an undivided phase of use for a cell, regardless of the actual frequency with which it is used. We'd like to remove this abstraction, to give a fuller picture of how heap cells are used. One method is to add bit strings to cells. Each bit represents one interval between censuses, and is set if the cell is used in that interval. If one can afford the space, such bit strings could provide a rich source of information. The main problem might be presenting this information in an accessible way.

Another thing on our to-do list is a proper integration of post-mortem and live-heap techniques. Programmers' questions are often qualified by restrictions specifying cells in terms of several different categories: e.g. 'What's the biographical profile of cells retained by this function?' or 'So what's retaining these cons cells being dragged for the latter part of the computation?'.

Bit strings make it possible to answer the first of these questions, using a bit to represent each census interval, set if the cell was retained by the function in question. We have not found any method (short of brute force enumeration) to implement the more specific restriction in the second question.[7] Unfortunately this is the question we would have liked to ask in §4.4!

Others concerned with efficient implementation of lazy functional languages have developed the use of *unboxed values* [PL91]. Eliminating the 'box' of a cell reference around basic values such as integers can result in marked savings of both space and time. In the case of lexical position information the total space×time cost of our example could be reduced by another 10%.[8] It is therefore important to note that nhc does *not* (as yet) make use of unboxed values, with the exception of literal strings that are held packed rather than as ordinary lists of characters. It is possible to program in the spirit of unboxed values: for example, the explicit change of representation for lexical position information can be viewed as a form of box-avoidance; in effect, three numeric indices are made to share the same box. But the introduction of true unboxed values can be expected to yield further reductions in the amount of heap space needed.

It is unsatisfactory that the programmer must make explicit use of selective imports to reduce memory consumption in the compiler. Selective imports are mainly intended as an aid to help a human reader of the program, not as a tool to tune memory consumption. We are therefore implementing *lazy loading* in nhc: the compiler will only load information about identifiers that are needed in the compiled module. Selective imports are still checked, as hiding a needed identifier is an error.

Acknowledgement

Niklas Röjemo is supported by a post-doctoral scholarship from the Swedish Research Council for Engineering Sciences. We also acknowledge the financial support of Canon Research Europe.

References

[AJ89] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[AJ93] L. Augustsson and T. Johnsson. *Lazy ML User's Manual*. Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1993. Distributed with the LML compiler.

[Aug84] L. Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, Texas, 1984. ACM Press.

[Hud92] Paul Hudak et al. Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language. *SIGPLAN Notices*, 27(5), May 1992.

[7] The less precise restrictions 'drag *or* use' or 'lag *or* void' are however easy to implement.

[8] Unboxed integers were simulated by the unit type to obtain this estimate.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[MFH95] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 66–77. ACM Press, June 1995.

[PJL92] S. L. Peyton Jones and D. Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.

[PL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proc. 5th Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 636–666. ACM Press, August 1991.

[RR95] Colin Runciman and Niklas Röjemo. New dimensions in heap profiling. Technical Report YCS 256, Department of Computer Science, University of York, 1995. (Revised version to appear in *Journal of Functional Programming*).

[RW92] Colin Runciman and David 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, 1992.

[RW93] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, 1993.

[Röj95a] Niklas Röjemo. *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden, May 1995.

[Röj95b] Niklas Röjemo. Highlights from nhc: a space-efficient Haskell compiler. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 282–292. ACM Press, June 1995.

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

[SP95] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *22nd ACM Symposium on Principles of Programming Languages*, pages 355–366, San Francisco, CA, January 1995. ACM Press.