A Trip Down Memory Lane in Haskell

Neil C. C. Brown
University of Kent, UK
neil@twistedsquare.com

Adam T. Sampson
University of Kent, UK
ats@offog.org

Abstract

While writing a compiler in Haskell, we had a problem with a large memory usage. We show how, by changing only a few small aspects of our design, we were able to reduce the live memory use of our program from 650 megabytes down to a mere 10 megabytes. We examine the aspects of our program that caused this problem: the writer monad and generic programming. We conclude that Haskell is a powerful, useful and fun programming language, but that programmers must remain careful as to which techniques they choose to use, and must keep a close eye on their memory profile throughout the development process.

Categories and Subject Descriptors D1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Experience Report, Memory Usage.

Keywords Haskell, Monad Transformers, Generics, Memory Usage.

1. Introduction

Tock is a compiler for concurrent imperative languages such as occam-π (Welch and Barnes 2005) and Rain (Brown 2006) that generates C/C++ code which is subsequently linked against concurrent run-time libraries. Tock is currently around 14,000 (non-blank, non-comment) lines of Haskell, with an additional 6,000 lines of tests. It is based on the nanopass methodology (Sarkar et al. 2004) wherein an abstract syntax tree (AST) is transformed by a series of separate passes.

Our experience of programming in Haskell has been generally positive. However, at one point during development we found that Tock used a very large amount of memory. We knew that our AST contained over a million data items on a specific problematic test-case and we could not say exactly how much memory use would be reasonable for this in a lazy functional language. When our program overflowed the limits of our physical RAM, we began investigating the cause.

The GHC Haskell compiler supplies tools for profiling and visualising heap usage – all the figures in this paper are generated directly by GHC tools. Each band in these graphs represents the memory allocated by a particular function in the program (typically a compiler pass). The bands cumulatively form the total live memory use of the program over time (the vertical axis). Live memory is that which is still in use, and therefore cannot yet be garbage collected. We ignore here the additional overhead of dead memory that has yet to be garbage collected. The horizontal axis is computation time, and does not include time spent in garbage collection.

We will explain briefly some of the design of our program, and show how a few small changes to our choice of tools managed to reduce the memory usage by around a factor of 60, before reflecting on our experience.

2. Monads

All the nanopasses in our compiler are monadic. We use monads for several purposes, including:

- Carrying around compiler state, such as the symbol table;
- Providing an error mechanism for fatal compiler errors;
- Reading in and writing out files using the IO monad.

Monad transformers (Jones 1995) allow easy composition of common monads:

- StateT – persistent, modifiable state;
- ReaderT – persistent read-only state;
- WriterT – persistent append-only state;
- ErrorT – allows errors to short-circuit the computation.

The Haskell monad transformer library provides both strict and lazy versions of each of these. We found monad transformers very useful. Our original core monad stack, using the lazy implementations, was:

```
type PassM = ErrorT ErrorReport
             (StateT CompilerState
             (WriterT [Warning]
              IO))
```

We eventually traced two of the most severe memory problems back to the WriterT monad transformer.

2.1 Writing a Large Amount

Our final pass in Tock is a code generator, producing plain text C/C++ code that is written to a file upon completion (ready to be compiled by the C/C++ compiler). We took what seemed to be the obvious approach, and we added an extra WriterT [String] to our monad stack for this final pass. Here is the changed monad stack in full:

```
type CodeWriterM = WriterT [String]
                   (ErrorT ErrorReport
                   (StateT CompilerState
                   (WriterT [Warning]
                    IO)))

-- or: type CodeWriterM = WriterT [String] PassM
```

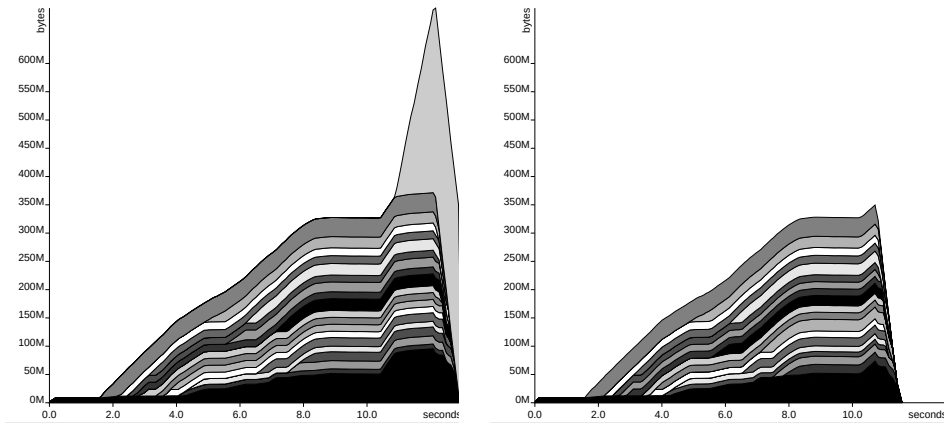


Figure 1. Memory usage of using a writer monad for the code generated by the final pass (a) and writing directly to disk (b).

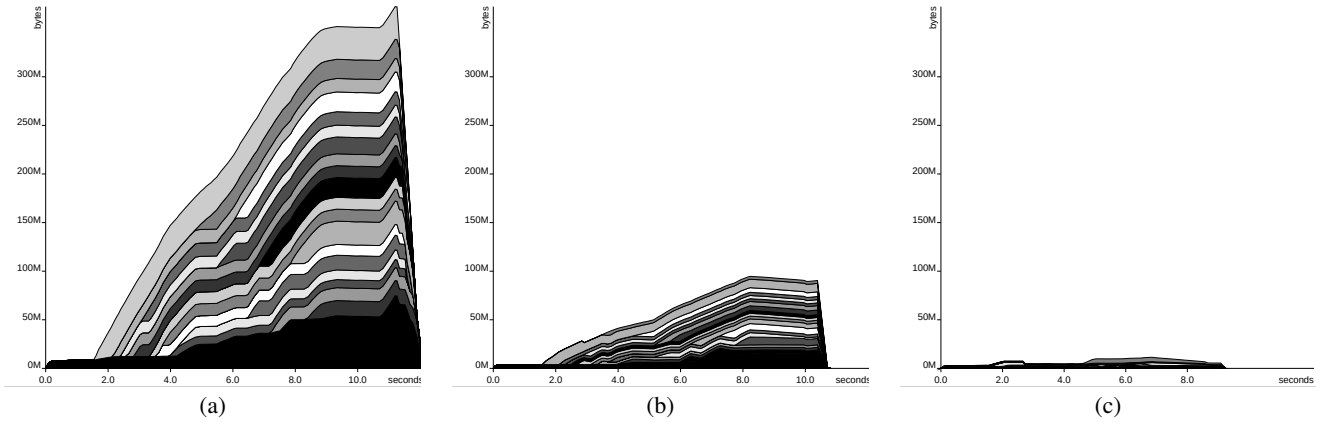


Figure 2. Memory usage of using different monads for keeping track of warnings: lazy writer (a), strict writer (b), state monad (c).

All the C/C++ code generation was done inside this top-most writer monad. Each small string generated was added to a list of strings that were then written to a file (without first being concatenated). We expected some overhead for generating the file in memory before writing it out, but the true extent was surprising.

The graph in figure 1a shows the memory usage of the program with this writer monad. It is clear that the code generation stage results in a huge spike at the end: over 300 megabytes. Using the strict and lazy implementations of WriterT gives the same result here. We removed the writer monad and instead wrote the strings directly to the file (recall that our monad stack has IO at its base). This almost completely removed the memory spike, as can be seen in figure 1b.

The generated file was around 240 kilobytes in size – the memory spike was over 1,000 times larger than the extra data being stored. This overhead cannot be explained solely by an increase in data storage due to having lists of small strings. Instead it must be due to the building up of unevaluated thunks that simply adds the empty list on to the list of strings in each lifted action that does not add to the output. Regardless of the exact reasons behind the overhead, it is surprisingly large.

2.2 Writing a Small Amount

The memory use in Tock remained generally high, as shown in the graph in figure 2a. No single pass could be picked out as the cause for this; each pass took up a similarly-sized band of memory. Biographical heap profiling revealed nothing of interest – only five megabytes of data (out of over 300) was retained longer than necessary or never used.

Given our previous trouble with writer monads, we tried switching the lazy implementation of the writer monad for our warnings to the strict implementation. The effect can be seen by comparing the graph in figure 2a (lazy) to figure 2b (strict) – overall memory use was reduced by roughly a factor of three. At this stage we also tried unrolling our monad stack into a single monad with equivalent code, but this made no difference.

During the test depicted in the graph, only seven warnings are generated, in the very first pass. Suspecting that the high memory usage was related to the repeated concatenation of this list of seven warnings with the empty list, we tried switching from a writer to a state monad for warnings: a five line change. The effect can be seen by comparing the strict writer implementation in figure 2b to using the (lazy) state monad transformer in figure 2c. The latter graph can also be seen in figure 4c.

The difference this time was roughly a factor of 10 improvement over the strict writer monad, giving a total improvement of a factor of 30 over the original lazy writer monad. There is also a dramatic qualitative change, which can be seen more clearly by comparing figures 3 (writer) and 4 (state).

Inspection of the graphs indicates that memory use with the writer monad only increases, with thick persistent bands, whereas with the state monad the memory use increases and decreases in spikes. This suggests that the garbage collector is unable to function effectively when using the writer monad, perhaps because one huge list of unevaluated concatenations (with the empty list) is being built up and evaluated at the very end of the program. In contrast, the state will be carried through the program unaltered except where warnings are issued.

3. Generics

One of the most attractive aspects of Haskell to us is its support for generic programming. There are several generics frameworks available in Haskell; we use the Scrap Your Boilerplate framework (Lämmel and Peyton Jones 2003).

This framework provides operations such as everywhereM that apply a monadic transformation function (type: `a -> m a`) to every instance of type `a` in a tree structure. This makes programming our passes very easy – we write small functions for specific AST types (such as expressions), and use the generics framework to apply our functions over the whole tree automatically.

We discovered that the everywhereM transformation was inefficient in terms of memory usage. Tock has a pass for transforming one occam-π operator, `AFTER` into another (`MINUS`). The pass transforms the part of the expression using the operator, and leaves all other AST elements untouched. This operator is used only seven times in a particular compiler test containing over 2,000 lines of code.

The graphs in figures 3a and 4a show Tock's memory use, with the writer and state monads respectively for warnings (as discussed in the previous section). These graphs use everywhereM for traversing the tree.

At this point, using the state monad, we were not as concerned with memory use (down to around 30MB, figure 4a) as we were with execution time. We were worried that users would not appreciate waiting for well over a minute for compilation. We deduced that the execution time was taken up by the blanket everywhereM traversal descending into each character of each string in the AST. Our AST contains a large amount of strings; for example, each node is annotated with a source position for giving back error messages.

We implemented a custom adaptation of SYB that explicitly did not descend into Strings (we have no passes that operate explicitly on strings or characters). The execution time was greatly reduced (by roughly a factor of ten), but there was also a reduction in the memory use (roughly a factor of four). Figures 3c and 4c show the memory use with all passes switched across to this custom traversal. For interest, we also provide graphs of memory use where one pass (figures 3b and 4n) still uses everywhereM and all the others use the custom traversal – it is easy to see which is the pass that still uses the old traversal.

It seems that the everywhereM traversal is copying all the strings when it descends into them. This difference is cumulative when using the writer monad (which was foiling the garbage collector), so that each pass retains the increased amount of memory across the execution of the program. This copying cannot be entirely avoided by lazy evaluation because the traversal is monadic, so each node must be traversed in case it has an effect on the monadic state (or causes an error, etc). Changing to not descend into strings allows the pointer to the whole string to be re-used.

After experiencing this time and memory problem with the SYB library, we began investigating improving our generics techniques, focusing particularly on execution time. However, this falls outside the scope of this paper.

4. Reflections on Memory

Now that we have eliminated the causes of our high memory usage, they are clear and to some extent reasonable. However, we do not think that we took an obviously wrong approach in our design. A WriterT [String] monad seemed like a sensible approach for our code generation step and a WriterT [Warning] similarly reasonable for warnings. Given that everywhereM does not modify the strings in the tree, we might have anticipated a little slow-down in the program for processing them, but not an increase in memory use.

We also did not know what our memory usage could or should be. The long-standing myth is that high-level languages are inefficient. When our program used several hundred megabytes, it was clearly inefficient. But when the peak dropped to around 80 megabytes, we did not think it could drop much lower. Ultimately, Haskell's memory usage is very good – for an AST with a million data items[1], a peak memory use of ten megabytes is highly commendable, especially on a 64-bit machine where each pointer will be eight bytes. This efficiency must be due to lazy evaluation, since that the full number of data items in the AST could not fit into that amount of memory all at once.

Similarly, we did not know what shape we should expect our heap profile to be. Given our initial knowledge of monads – that they sequence operations into one huge expression – an ascending wedge shape seemed a reasonable fit. In our minds, the program was building up one huge expression and then evaluating it at the end. We have now seen that this can be optimised out.

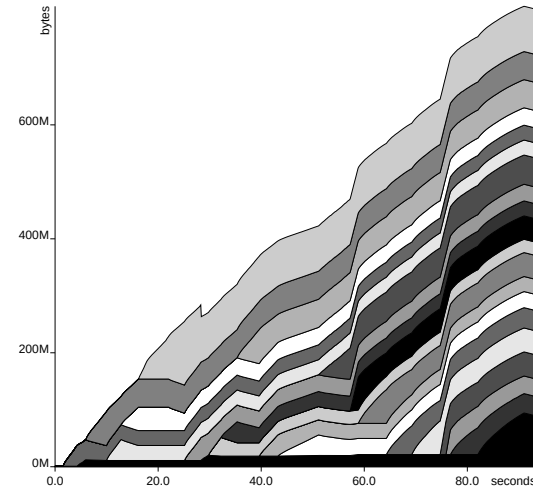
We wonder whether this is the side-effect of moving to a high-level functional language. Although we have a good mental model of Haskell's semantics, we do not have the same for its implementation – especially compared to lower level languages such as C. This opinion confirms a similar belief by the authors of the original Haskell profiler (Sansom and Jones 1997).

Trying out other optimisations such as changing strings to packed strings, or adding strictness annotations to data-types, always worsened rather than improved our memory performance. This adds to our experience that knowing how to optimise lazy functional programs is difficult.

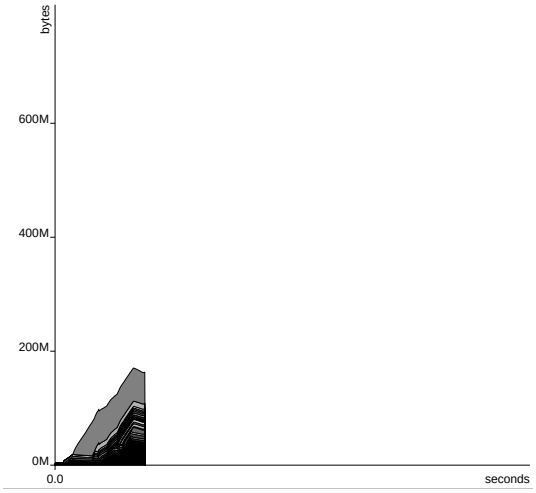
We also had problems with profiling the memory use. By the time the live usage had reached nearly 700 megabytes, the actual use would be double or triple that (including dead memory yet to be garbage collected). With profiling overhead on top, we needed a machine with three gigabytes of memory to perform the profiling! This is why all our benchmarks were run on a 64-bit machine; it was the only machine we had with a big enough address space to perform the the profiling.

We are thankful that GHC provided the tools with which to track down the memory problems in our program (Sansom and Jones 1997). When a particular function takes up a lot of memory (graphically: one band is especially thick), it is clear where to start. However, when all our passes required a lot of memory, it was not at all obvious to us that we should alter the generic traversal. Similarly, when our graph was filled with persistent thick bands, swapping the writer monad for a state monad was not an obvious fix. We can envisage other programmers new to Haskell having similar difficulty tracking down their memory problems.

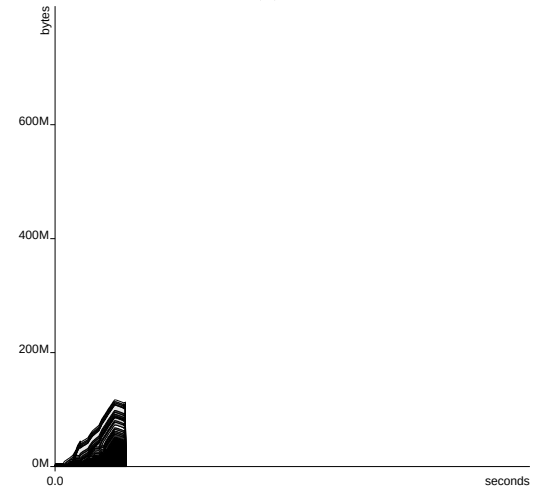
[1] By data item, we refer to the number of nodes that would be processed by a generic traversal. This can be measured using the SYB gsize function.



(a)

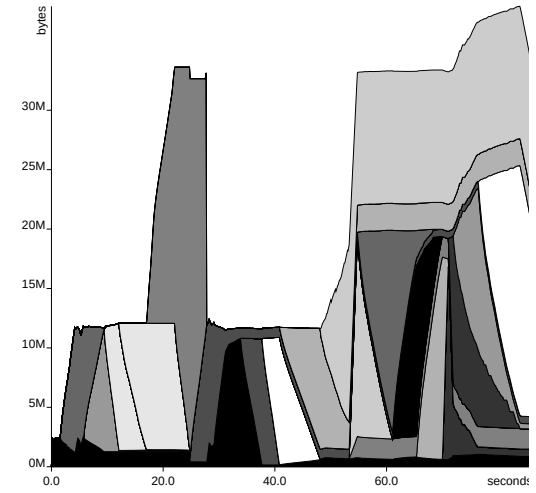


(b)

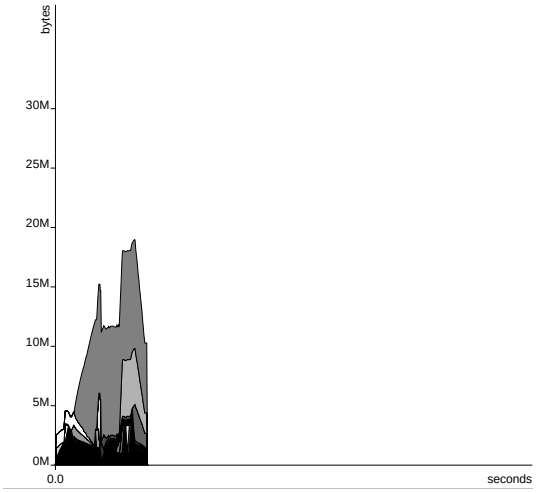


(c)

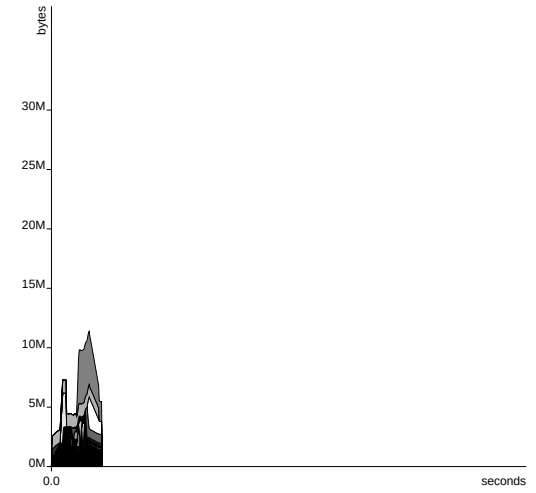
Figure 3. Memory usage with a strict writer monad for warnings, with all passes using everywhereM (a), with only one pass using everywhereM (b), and using our custom traversal for all passes (c).



(a)



(b)



(c)

Figure 4. Memory usage with a state monad for warnings, with all passes using everywhereM (a), with only one pass using everywhereM (b), and using our custom traversal for all passes (c).

Our advice to other Haskell programmers on a large project is to profile regularly as you develop. Just as tests should be run regularly, in hindsight we would have benefited from profiling regularly. By the time we began profiling, it was no longer clear what the cause of the excessive memory usage was. If we had been comparing one week's profile to the next, we could have more easily traced the cause for the ballooning memory profile to adding a writer monad for warnings.

5. Conclusions

In an ideal world, programmers use the techniques and algorithms that conceptually fit the problem, hoping to not have to care about implementation details. It is our experience that when programming in Haskell, the wrong choice of technique can prove costly – our program changed from using 650 megabytes down to 10 by changing around 30 lines in a program of 14,000.

When we did optimise our program, we were very impressed with total memory use – a peak of ten megabytes of live memory for our large test-case. On a 32-bit machine, the memory use was almost exactly half that of the 64-bit machine; a peak of just over five megabytes! This was better than the previous occam-π compiler, written in C, running on the same test case.

We also note that the performance of the garbage collector has a significant effect on the performance of our program. The percentage of the total program time spent garbage collecting varied across conditions – and is somewhat proportional to the total memory use of the program – but was always in the range 30–60%. Thus, any improvement to the efficiency of garbage collection, in terms of time or memory, will always be of great benefit to Tock.

Benchmark Information

All the benchmarks were carried out on an x86-64 machine, on GHC version 6.8.2, compiled with the `-O2` optimisation flag. Where possible, similar results were confirmed on an x86 machine and/or without optimisation.

The standard deviation in time on the smallest example, figure 4c, was 0.02 seconds over 10 samples. The garbage collection statistics are not completely deterministic. The standard deviation of total bytes allocated in the same example was 12,182 bytes (roughly 0.01 megabytes) over 10 runs. We expect the standard deviations for all other examples to be of similar magnitude.

Acknowledgments

Our work on Tock is supported by EPSRC grants EP/P50029X/1 and EP/E049419/1.

References

Neil C. C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In *Communicating Process Architectures 2006*, pages 237–251, September 2006. ISBN 1-58603-671-8.

Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, pages 97–136, London, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI*, pages 26–37, 2003.

Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Trans. Program. Lang. Syst.*, 19(2):334–385, 1997. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/244795.244802.

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *ICFP 2004*, pages 201–212. ACM Press, 2004. ISBN 1-58113-905-5. doi: http://doi.acm.org/10.1145/1016850.1016878.

Peter H. Welch and Fred R. M. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005. ISBN 3-540-25813-2.