nhc - Nearly a Haskell Compiler

Niklas Röjemo

Abstract

This paper describes nhc - a compiler for a subset of Haskell. Nhc is the result of the author's wish to have a Haskell compiler for his Acorn A5000 to play with. Previous the only Haskell system available for A5000 was Mark Jones' gofer. The disadvantage of gofer is that it is written in C so the play part isn't as fun as it could be.

Nhc is written in Haskell with a small run-time system in C. It's possible to re-compile nhc with itself in 3MB of memory, using a lot of time. But as long as the interfaces between modules don't change then the turn around time is quite tolerable.

Nhc can profile the use of memory to make it easier to detect space leaks in programs. The profiling includes: What created the nodes in the heap? What does the heap consist of? How long time do the nodes in the heap live? Who is retaining the nodes?

1 Introduction

Nhc compiles a subset of Haskell 1.2 (it has no extensions except a pretty loose treatments of some illegal Haskell code) and generates byte code which is interpreted by a small run-time system written in C.

The omissions are:

- Renaming and hiding in modules.
- Arrays.
- Overloaded numeric constants.
- Defaults for unresolved overloading.
- Derived instances of standard classes.
- Contexts in datatype definitions.
- Full range of numeric types and classes.

The goal for nhc was to re-compile itself on a 4MB Acorn A5000 and generate faster code than gofer. Both of these goals are now fulfilled but not with significant margins. Re-compiling nhc on an A5000 takes over 7 hours and the speed advantage compared to gofer is not very large (the all deciding `nfib 20` takes 5s with gofer and 4s with nhc on an A5000). But if it's only changes local to a module that is needed then nhc is quite usable.

The main part of nhc (NhcComp) is written in Haskell and translates Haskell sources into byte code. The output from NhcComp can be moved between different ports of nhc which makes it easy to cross compile nhc. The differences between target machines are resolved by feeding the byte code through C's preprocessor before the assembler creates the object files. The use of an assembler to create object files is an overkill as the only thing needed is to translate an ascii representation of a byte array, with some labels defined, into an object file. Code from nhc also includes an interpreter, written in ansi C, for the byte code.

Even people with a lot of memory might find a use for nhc's profiling capabilities. Retainer information looks promising for informing the programmer which parts of the program that need some extra tuning. Lifetime profiles contains information that is useful for deciding if generational garbage collection could be advantageous.

2 Implementation

Nhc generates code that uses a modified G-machine to evaluate Haskell programs. Most of the implementation is taken from [SPJ87].

It was clear from the beginning that space would be a problem for the compiler when it re-compile itself on a A5000 (with 4MB of memory of which 1MB is used by the screen and the operating system). The usage of space can be subdivided into three groups:

- Code and static data
- Heap
- Stacks

The code size of lmlcomp, the main part of hbc, is 3MB for a sun sparcstation, there is no reason to believe that it would be any smaller for an A5000.One reason for the large binary is the size of the source code, nearly 20000 lines of LML-code and another 20000 lines of C and M code for the run-time system. Note that the Haskell parser isn't included in the above numbers. The size of lmlcomp can be explained partly by the history of the program. From the beginning lmlcomp was intended to compile LML code, but have later been changed by L. Augustsson to compile Haskell. A lot of extensions have also been added as time went by so that hbc now compiles an extended version of Haskell. Nothing is however removed so the code grows over time.

The space growth is acceptable as size isn't a major concern for hbc, which can be seen in the choice of a copying garbage collector as the default garbage collector. Optimizing for speed is however of great importance with specialisation of functions, strictness analysis etc., all of them need memory both for the code and in the heap for data. This adds up to the mentioned 20000 lines of LML-code and the need for a heap size of 12MB for compiling some of the source files of the compiler.

Hbc also uses two unbounded stacks for evaluation. Unbounded memory usage isn't a serious problem if we have virtual memory, we just allocate more memory for the stacks than is needed. Unused virtual memory only uses disc space which is pretty cheap. In a small machine we normally don't have virtual memory so we must partition our memory with great care not to allocate to much to one stack so the another stack, or the heap, overflows.

Nhc tries to reduce all memory use as much as possible, in most cases ignoring the extra time it might take.

2.1 Code and static data

There are two ways to decrease the size of nhc's code

- Generate smaller code for a given source file.
- Write a smaller compiler, i.e., less than 20000 lines of source code.

Nhc tries to do both and currently produce a binary for NhcComp which is only 25% of the 2MB binary that hbc creates from the same source. We should note that hbc (using 12MB of heap) needed 15 minutes to compile NhcComp, nhc (using 4MB of heap) needed the same if NhcComp was compiled with hbc but a terrifying one and a half hour if NhcComp was compiled with nhc. These numbers are for a Sparc 10.

2.1.1 Generate smaller code

It's possible to generate a smaller binary by eliminating common subexpressions and inserting clever abstractions which makes it possible to combine code from different parts of the source. Unfortunately these transformations use heap space and was therefore abandoned quite early in the development of nhc.

A 'cheap' way to reduce the size of the generated code is to use byte code instead of machine code. This means that nhc must include a small interpreter in all generated programs but this is not be a big part of the binary for the compiler itself (only 18%). For small programs this might be unproportional large but small programs don't have any problem with code space by definition.

The saving in space must however be paid in speed. A byte code interpreter makes a lot of jumps whose destinations are decided very late, and this is bad behaviour for modern processors which use heavy pipelining and branch predictions. The processor in A5000 doesn't have any branch prediction at all and only a three stage pipeline so dynamic branch destinations aren't a total failure for this machine, even if it's slower than machine code.

The gain in space is however quite substantial as most byte code instructions fits in 2 or less bytes which is much better than the 4 bytes instructions the RISC processors in A5000 uses. The gain can in many cases be more than the 50-75% gain from smaller instructions as the byte code instruction set can be optimized for graph reduction so that fewer byte code instructions are needed than machine code instructions for a given Haskell function. The optimization of the byte code instruction set is however not done yet, only a few specialised instructions are defined.

2.1.2 Less source code

Nhc has nearly no optimizations, it only consists of the necessary parts in a Haskell compiler for the G-machine. The total source code for Nhc is 9000 lines of Haskell and 6000 lines of C for the run-time system. A reduction of nearly 1000 lines of Haskell could be done if the lexical analyser and parser where a separate program as is done in hbc.

The Haskell source for nhc can be divided into the following parts:

- Lexical - handwritten tokenizer.
  All identifiers are hashed when read so that equal identifiers can share the same string in the heap. Part of the cost of hashing all identifiers are paid back in the renaming phase as all comparisons of identifiers only need to compare integers instead of strings.
- Parser - handwritten parser.
  The parser is written with a continuation monad that allows backtracking but has an efficient commit combinator that get rids of all the data that is only needed for backtracking. The commit combinator is essential when parsing with backtracking, as we otherwise introduce a space leak starting with the first position of the source code where backtracking is possible.
- Renaming - gives all identifier unique names.
  All different identifiers are replaced with unique integers. The integers is then used as indexes into a balanced tree. Another possibility is to use sharing so that all instances of an identifier points to the same shared data. In the later case all information about an identifier must be known when renaming. This can be solved by lazy evaluation and some circular data structures. Unfortunately lazy evaluation can sometimes hold on to a lot of heap space which can be difficult to get rid of, especially if we have a circular dependency. A decision to use as few circular data structures as possible was therefore taken for nhc.
- Scc - find strongly connected groups. (Needed for typechecking.)
- Type - type checking and insertion of dictionaries.
  Here circular data structures are used to insert dictionaries as no other method was known that didn't either have a nasty complexity (exponential in the number of nested let-bindings) or holds on to a lot of information before it could insert all dictionaries with an extra pass over the syntax tree.
- Export - creates the interface file.
- Case - simplifies pattern bindings and pattern matching.
  The removal of pattern is done without code duplication. This means that some programs might test the same pattern more than once, but it saves space.
- Lift - lambda lifting.
  This is a simple lambda lifter and doesn't use fully lazy lambda lifting. It only does what is necessary for the G machine which can't evaluate code with free variables.
- Code - intermediate code for some small optimizations.
  All applications with known functions are marked as vector applications or constant expressions if the arity is greater than the number of available arguments. A vector applications use less space than a chain of binary apply nodes, the later use at least 2 pointers for every argument compared to one pointer/argument for vector applications.
- G-code - generates (and does a little peep-hole optimizing of) the byte code.

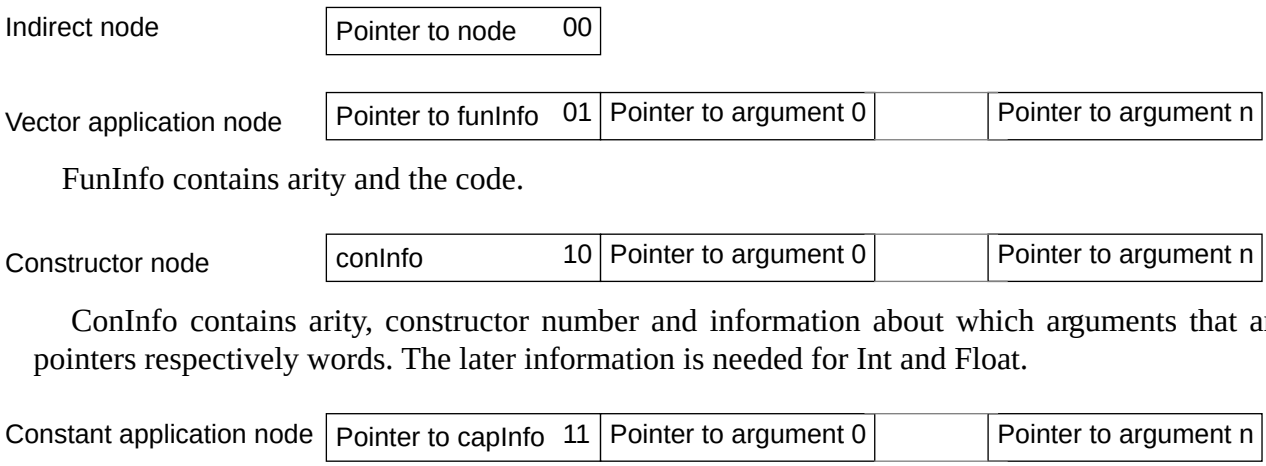

Figure 1: Node layout for nhc. Application nodes are represented with a vector application node of the inbuilt function _apply.

2.1.3 Heap

The possibility to use a small heap depends mostly on the amount of data that is live at a given moment, but if we want the program to finish in reasonable time, then also the rate that free heap space is used is of interests.

Nhc uses the following implementation choices to reduce the amount of live data:

- compact graph representation
  When designing the node layout, size was the main consideration (see Figure 1). The lack of a dedicated application means the high order functions cost more (3 words per argument) than if a two-word application node where defined. Profiling has shown that nhc uses a lot of vector applications with the apply function, over 20% of the heap at some times, so something should be done here.
- pat bind update [Spa93]
  When a variable in a left hand pattern is used then all variables in the pattern are set to point to their parts of the expression. This means that the node representing the expression that we matched against can be released as soon as the first variable in the pattern is used instead of waiting for the last one.
- indirection
  No updates are done by copying, instead the reduced node is overwritten with a pointer to the result. If a function creates the result node then overwriting is used if the result is smaller or equal to the redex. Copying does not cause any extra evaluation if the result node is evaluated before being copied, but we might lose sharing of the answer. This was one of many reasons for space leak in [RW93].
- tail calls
  If the last expression of a function is a call to a function then some compilers (e.g., hbc) rearrange the stack and jump to the new function. This might lead to space leaks as the node representing the old application hasn't been overwritten yet and therefore might hold on to unnecessary data. In hbc this can be solved by using the option to zap redex, which overwrite all application nodes with a dummy node at the start of their evaluation. Nhc instead tries to build the new application on top of the old application, rearranging the stack and then jump to the function in the new application. If the new application is larger than the old application then an indirection node is used to overwrite the old application before jumping to the new function.

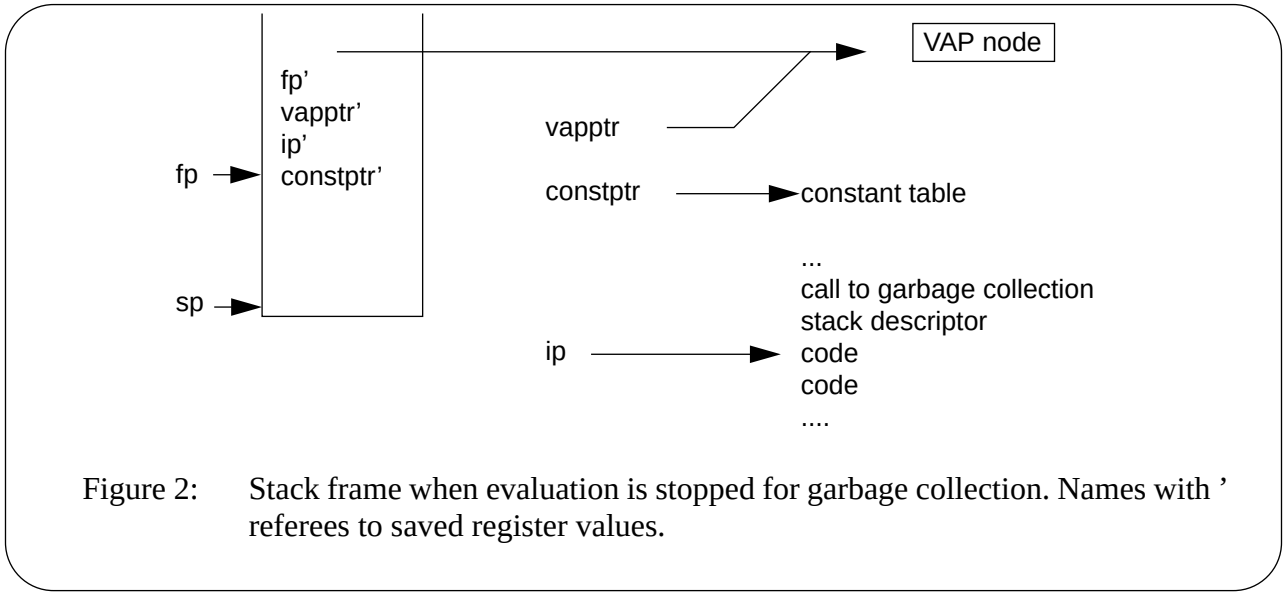

Figure 2: Stack frame when evaluation is stopped for garbage collection. Names with ' referees to saved register values.

- sharing of zero arity constructors
  Characters and all user defined constructors with zero arity are allocated once outside the heap, e.g, all uses of [] or a character uses the same node. It's not possible to do the same for Int and Float, but small integers have a table outside the heap.

2.1.4 Stacks

The only thing currently done for reducing the stack usage is that arguments to functions aren't copied to the stack before reduction. As all reductions are on vector application nodes (VAP nodes) it's quite cheap to fetch the argument from the VAP nodes whenever they are needed.

Another difference between nhc and the G machine in [SPJ87] is that nhc's G-machine only uses one stack (Figure 2). The stack is organised into frames according to a sketch on Lennart's whiteboard. It makes garbage collection slightly slower because of the need to keep track of which values on the stack that are pointers. A nice property is that it is easy to decide which functions that are responsible for which pointers on the stack, which is needed for some heap profiles (section 3).

It is possible to reduce the size of a frame if we use the following observations:

- Always true: vapptr points to the same node as the stack cell below fp'.
- Mostly true: It's possible to get the constptr from the VAP node pointed at by vapptr.
  The problem here is that pat-bind-update might overwrite the VAP node. This can be solved by emitting code so that the VAP nodes that can be overwritten by pat-bind-update always are of the same kind.

It's also possible to remove the stack descriptor from the code if we only allow pointers on the stack, this does however prevent the possibility to use un-boxed values.

3 Profiling

An unplanned diversion in the development was the inclusion of heap profiling. Heap profiling is done in hbc [CW93] but there are some extra possibilities when nhc is used. Hbc can answer the following questions about the heap:

1. groups: Which parts of the program produce the nodes in the heap?
2. module: Which modules produce the nodes in the heap?
3. producer: Which functions produce the nodes in the heap?
4. constructor: Which constructors do the nodes in the heap belong to?

5. type: Which types do the nodes in the heap belong to?

Nhc can answer 2 (Figure 3), 3 (Figure 4) and 4 (Figure 5) above. There is no significant difference between question 1-5, the answers to all questions can be found by adding one extra word to each node in the heap. The value of the extra word is a pointer to a table containing the answers for this node.

The extended profiling possible with nhc uses more extra words for every cell. Nhc can therefore also answer the following questions:

6. lifetime: How long time do the nodes in the heap live?(Figure 6)
   Every node has an extra word which records the creation time for the node. It's then possible to deduce lifetimes for all nodes by postprocessing the profile log.
7. retainer: What is retainig the nodes in the heap?(Figure 7 and Figure 8)
   The answer to this question is possible to find by looking at the structure of the graph in the heap. This can be a very costly operation which in the worst case needs *n+1* passes over the graph where *n* is the number of functions in the program. The implementation allows the user to limit the maximum number of functions in a retainer set. If the set of functions that hold on to a node is larger than the maximum retainer set then the profiler marks that node as belonging to the set of all functions. This limits the *n* to the size of the maximum retainer set.

6 and 7 was implemented after/during discussions with Colin Runciman.

4 Related work

Nhc is just one of the many Haskell compilers available. There now exist at least four other available (extension of / subset of / standard) Haskell systems. Only one of them is written in Haskell (ghc) and only one fits in 4MB of memory (gofer), unfortunately it isn't the same one.

4.1 hbc

Hbc is written in LML and compiles extended Haskell to native machine code. An interpreter (hbi) also exists. Goal is to generate as fast code as possible. Hbc runs on most UNIX workstations with enough memory which means 12MB (6MB with an optional garbage collector) for the heap, 3MB for the code and less than 0.5MB for the stacks to re-compile the compiler.

4.2 ghc

Ghc is written in Haskell and compiles extended Haskell to C code. Goal to generate as fast code as possible and be a system which is easy to extend and use for research. The recommended machine has at least 16MB memory, GNU C, and pearl. It's however highly unlikely that ghc can re-compile itself in 16MB.

4.3 yale

A Haskell interpreter written in Lisp.

4.4 gofer

An interpreter which Mark Jones wrote for his personal use as part of his research into "qualified types". This system is the one which is nearly everything nhc tries to be, the main differences being that gofer is written in C and that it is an interpreter. Gofer interprets an extended subset of Haskell with the following omissions:

- Modules.
- Arrays.
- Overloaded numeric constants.
- Defaults for unresolved overloading.

- Derived instances of standard classes.
- Contexts in datatype definitions.
- Full range of numeric types and classes.
- Some other minor differences (e.g., slightly different treatment of unary minus and the character '-' in operator symbols).

The extensions are in the typesystem but is mostly compatible with standard Haskell. It is possible to write Haskell programs that can run both under gofer and a standard Haskell system such as hbc with a some care (nhc is an example of such a program).

Mark Jones wanted to be able to use Gofer on a range of machines - in particular, on his PC and wrote it in a very portable C. Gofer is nowadays available for almost any machine with a C-compiler e.g., MSDOS, OS/2, NEXTSTEP, UNIX (sun hp dec ibm), Minix, Linux, Acorn, Amiga, etc.

5 Conclusion

The implementation of profiles for lifetime and retainers would probably have taken much longer time if we had tried to implement it in hbc, since nobody has a complete understanding what is happening in that code any more.

It can however be argued if the easy to implement the extra profiling methods compensates for the time to write nhc. The first lines of nhc was written sometime in the spring 93 and then work have been done on and off for nearly 9 months before nhc reached the current state including all heap profilers.

But besides easier implementation of the new profiling schemes we now have a Haskell compiler that can re-compile itself on an A5000 using 3MB of memory but it needs 7 and a half hour to do it. The code produced is slightly faster than gofer. A speed-up is probably possible with a more careful selection of the byte code instruction set.

6 Future works

- Increasing the subset of Haskell that nhc can compile, in particular:
  - Arrays, probably only small changes.
  - Renaming and hiding in modules, probably needs non-trivial changes in the symbol table.
- Increasing the speed of the generated code. A possible extension is to generate machine code for speed critical parts of the code.
- More heap profiling in particular:
  - usage: How often are the nodes used?
    An interesting variant is to ask for nodes that were unused for a long time before they died.

7 References

[AJ89] L. Augustsson and T. Johnsson
The Chalmers Lazy-ML Compiler
The Computer Journal, vol. 32, no. 2, 1989, p. 127--141

[RW93] Colin Runciman and David Wakeling
Heap profiling of lazy functional programs
J. Functional Programming, April 1993

[Spa93] J. Sparud
Fixing Some Space Leaks without a Garbage Collector
Proc. of FPCA 1993

[SPJ87] Simon L. Peyton Jones
The implementation of Functional Programming Languages
Prentice-Hall 1987

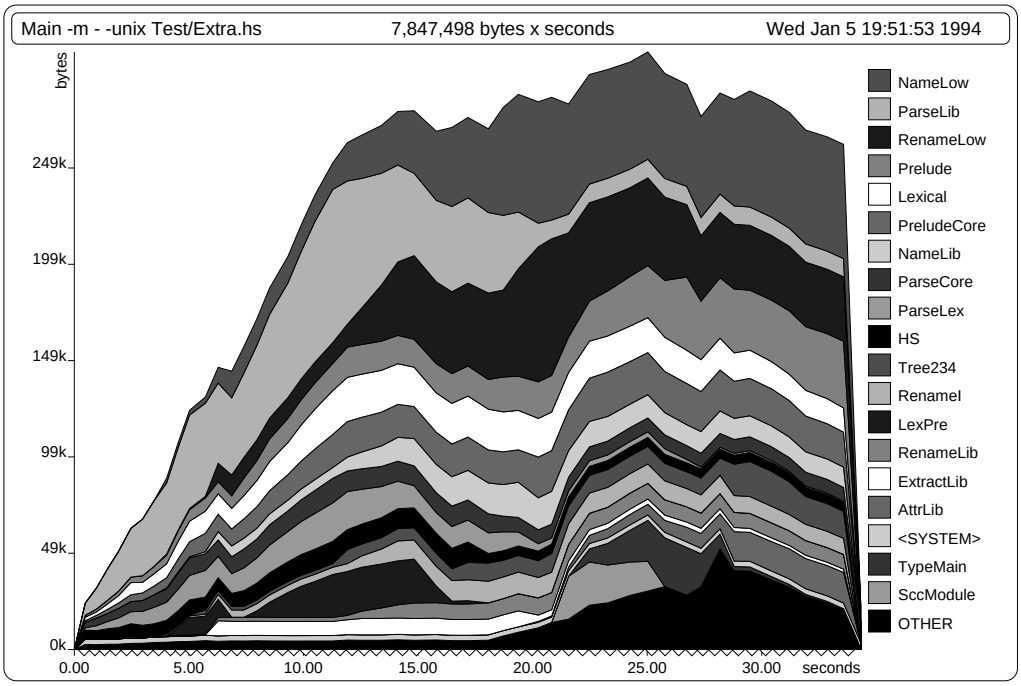

Figure 3:
Which modules produce the nodes in the heap?

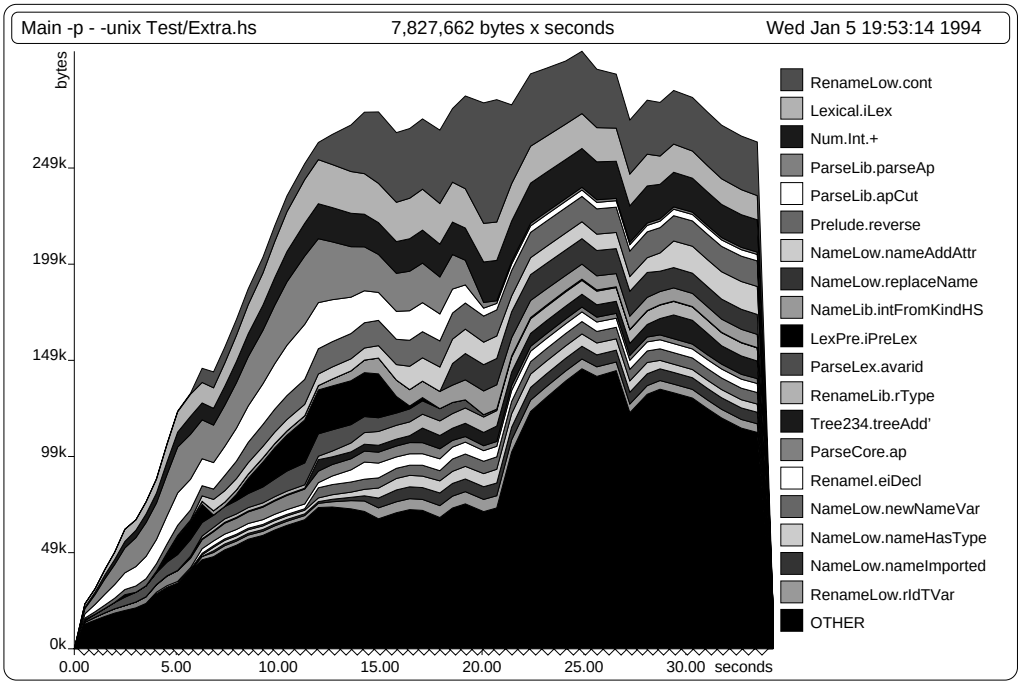

Figure 4:
Which functions produce the nodes in the heap?

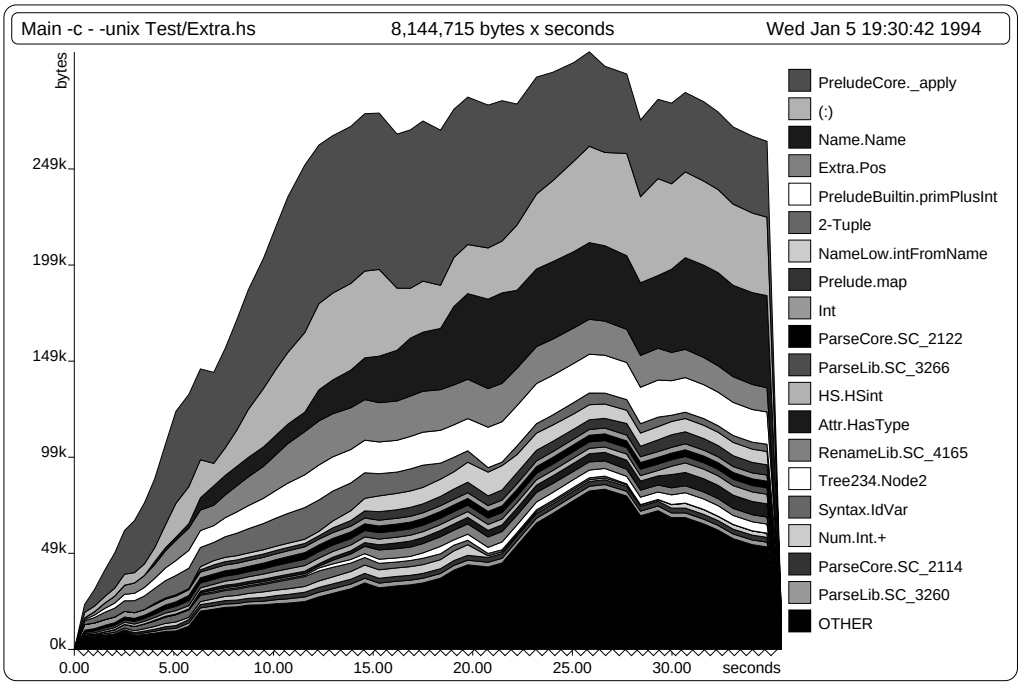

Figure 5:
Which constructors do the nodes in the heap belong to?
Function names, e.g., Num.Int.+, are used to name closures.

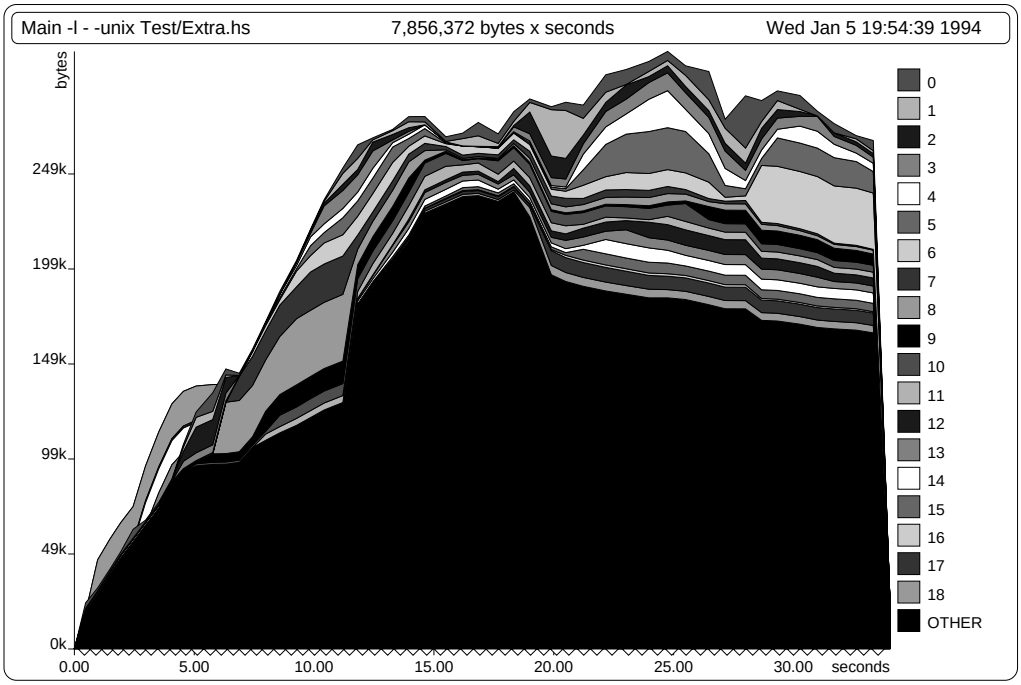

Figure 6:
How long time do the nodes in the heap live?
Lifetime is currently measured in the number of samples a node survived. The scale should be logarithmic, and in seconds, but that doesn't work yet.

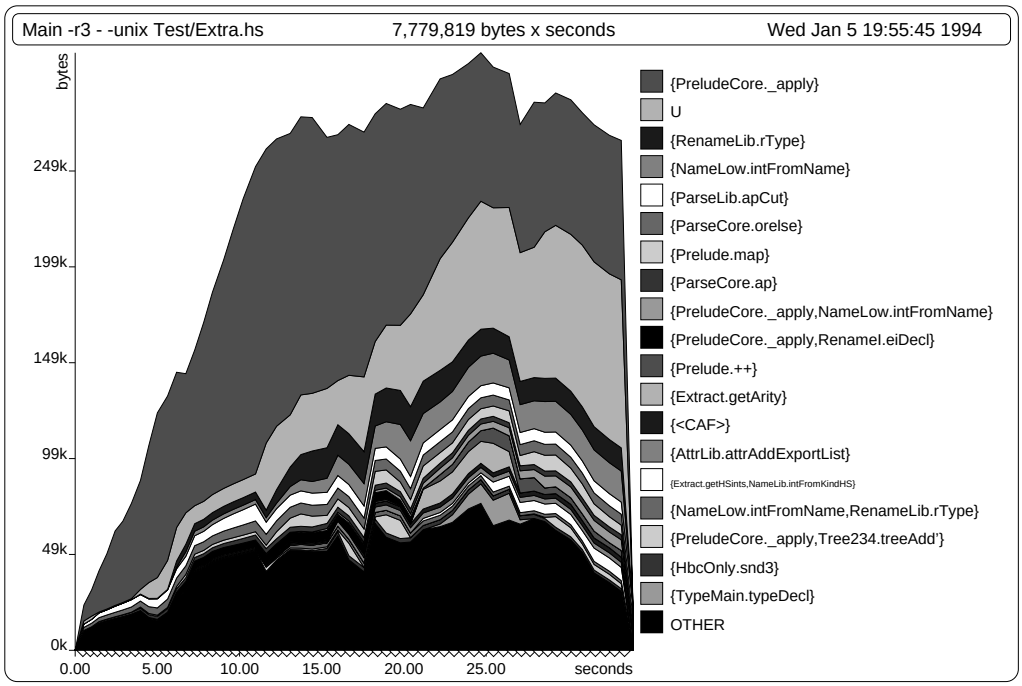

Figure 7:
What is retainig the nodes in the heap?
The '-r3' in the title means that we are only interested in sets with three or fewer member. Larger sets are collected under U.

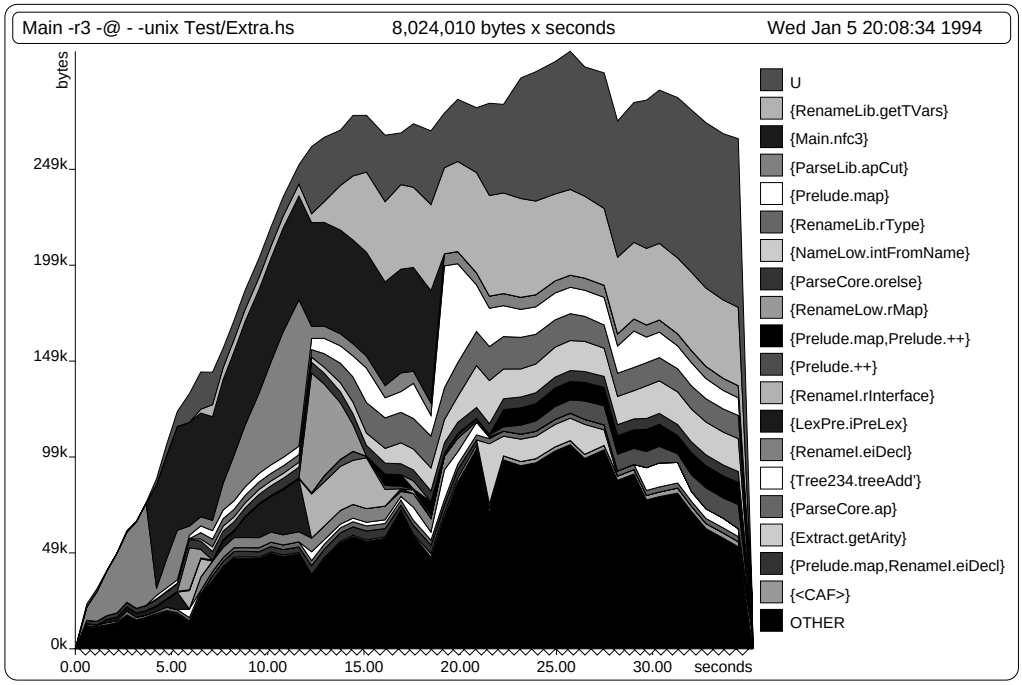

Figure 8:
As above, but we don't count `_apply` as a function of it's own. The memory used by `_apply` nodes, and whatever they hold on to, are counted as retained by the retainer of the `_apply` nodes.