Haskell Bytes

A guided tour through the heap of a Haskell program

Joachim Breitner*

13 June 2014
Galois tech talks

Haskell is a great programming language; I assume this is nothing new to people attending a Galois tech talk. But sometimes, one is let down by Haskell. Lets take the following code

```
main = do
   input <- getContents
   putStrLn $ "I read " ++ show (length input) ++ " bytes."
   putStrLn $ "The last line is:"
   putStrLn $ last (lines input)
                                                        Haskell
```

and run it on a 100MB text file. Using the parameter `+RTS -t` we can see some statistics and we learn that the program used 2521MB of memory – that's 24× too much!

Another classical example for unexpected runtime behaviour is the following code, which calculates the length of a list:

```
count :: Int -> [a] -> Maybe Int
count n (x:xs) = count (n+1) xs
count n [] = Just n
                                                        Haskell
```

In the Haskell interpreter we see that this code uses huge amount of memory and finally aborts with a stack overflow:

```
*Count> let x = count 0 [0..100000000]
*Count> x
Just *** Exception: stack overflow
                                                        GHCi
```

On the other hand, using Haskell sometimes leads to pleasant surprises. Turning to the first program again, if we delete the third line and run

```
main = do
   input <- getContents
   putStrLn $ "The last line is:"
   putStrLn $ last (lines input)
                                                        Haskell
```

we measure a memory footprint of 2MB – that is 50× better than expected! Not to speak of Haskell's ability to work with infinite lists…

*breitner@kit.edu, http://www.joachim-breitner.de/. The author was supported by the Deutsche Telekom Stiftung.

1 The protagonists

Before diving into the heap, let's first think about what we expect to find there: What does a Haskell program need to store in memory to do its work? Firstly, the actual *data*, i.e. constructors like True or Just or (,), which can contain more values. Haskell is a functional language, and the distinguishing feature of that is that it can treat functions as values. So we need to be able to store *functions*. Furthermore, Haskell is lazy, so there must be a way to store values that have not yet been fully computed. These are called *thunks*. These three are the most important; the general term for them is *closure*.

For these various closures, what bits of information will find there?

- Its type: This is actually not needed! The static type system ensures that all code always and only sees the values of the expected type, so it can blindly rely on that. That is very different from, say, Python!
- Which constructor: Yes, this needs to be stored, at least for types with multiple constructors like **data** Maybe a = Just a | Nothing.
- The parameters of the constructor: Of course!
- For functions: The code of the function.
- For functions and thunks: Don't forget their free variables!

1.1 Constructors

Let's get started and see what we find by playing around with GHCi a bit. We begin with a simple number:

```
*Utils> let one = 1 :: Int
*Utils> viewClosure one
0x00007f9c7a3337f8: 0x0000000040502608 0x0000000000000001
```
GHCi

We see that in order to store an integer, we need to words (which on my machine are 8 bytes long). The second obviously stores the actual number.

What about characters?

```
*Utils> let zett = 'z'
*Utils> viewClosure zett
0x00007f9c7a0e8238: 0x0000000040502548 0x000000000000007a
```
GHCi

Again we need two words, so 16 instead of one byte, which someone with a C background might expect!

Next we look at an algebraic data type, and put our value one in Just:

```
*Utils> let jone = Just one
*Utils> viewClosure jone
0x00007f9c7b082710: 0x00000000420DC920 0x00007f9c7a3337f8
```
GHCi

Note that the value Just one does not store a copy of one, but rather a reference to it.

With this information we can attempt to understand why our first example used so much memory. For that we recall that the String type in Haskell is just an alias for [Char], a list of characters.

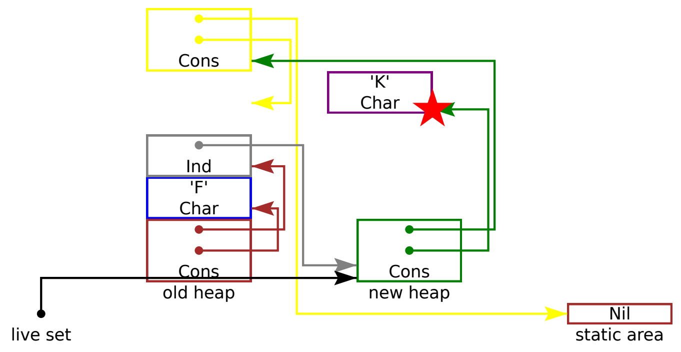
```
*Utils> let hello = "hello"
*Utils> viewListClosures hello
0x00007f9c7ba21fa0: 0x0000000040502668 0x00007f9c7ba21f91 0x00007f9c7ba21f70
0x00007f9c7ba21f90/1: 0x0000000040502548 0x0000000000000068
0x00007f9c7ba77b18: 0x0000000040502668 0x00007f9c7ba77b09 0x00007f9c7ba77ae8
0x00007f9c7a1a9768/1: 0x0000000040502548 0x0000000000000065
0x00007f9c7ba405f0: 0x0000000040502668 0x00007f9c7ba405e1 0x00007f9c7ba405c0
0x00007f9c7ba405e0/1: 0x0000000040502548 0x000000000000006c
0x00007f9c7ba83f38: 0x0000000040502668 0x00007f9c7ba83f29 0x00007f9c7ba83f08
0x00007f9c7ba83f28/1: 0x0000000040502548 0x000000000000006c
0x00007f9c7ba55a20: 0x0000000040502668 0x00007f9c7ba55a11 0x00007f9c7ba559f0
0x00007f9c7a01e840/1: 0x0000000040502548 0x000000000000006f
0x0000000040507e70: 0x0000000040502648                                   GHCi
```

We see that the list "hello" consists at first of one closure with three words. The second word is a pointer to a closure for the character 'h' and the third to the list "ello" and so on, until we reach the empty list []. Note the obviously different address: There is only one [], and there it can live in a static code area.

At this point I can demonstrate why it would be a bad idea to have direct access to the pointers in Haskell:

```
*Utils> System.Mem.performGC
*Utils> viewListClosures hello
0x00007f9c7a1510a8: 0x0000000040502668 0x00007f9c7a151389 0x00007f9c7a151372
0x00007f9c7a151388/1: 0x0000000040502548 0x0000000000000068
0x00007f9c7a151370: 0x0000000040502668 0x00007f9c7a1516f1 0x00007f9c7a1516da
0x00007f9c7a1516f0/1: 0x0000000040502548 0x0000000000000065
0x00007f9c7a1516d8: 0x0000000040502668 0x00007f9c7a151ac1 0x00007f9c7a151aaa
0x00007f9c7a151ac0/1: 0x0000000040502548 0x000000000000006c
0x00007f9c7a151aa8: 0x0000000040502668 0x00007f9c7a151ed9 0x00007f9c7a151ec2
0x00007f9c7a151ed8/1: 0x0000000040502548 0x000000000000006c
0x00007f9c7a151ec0: 0x0000000040502668 0x00007f9c7a150211 0x0000000040507e71
0x00007f9c7a150210/1: 0x0000000040502548 0x000000000000006f
0x0000000040507e70: 0x0000000040502648                                   GHCi
```

The same value, suddenly at different addresses! GHC uses by default a copying garbage collector: All still needed values are copied into a new memory region and the old one will be freed completely. This is faster and more precise than, for example, counting references, but this requires additional memory. Since a pictures says more than a thousand words, I visualized this (using the Haskell library diagrams) in a short video.



There is another effect which we could have observed here, had we run the compiled program (instead of running it in the interpreter). Then the output would be:

```
$ ./HelloGC
0x00007f16c0d04270/2: 0x000000000049b1c8 0x00007f16c0d04261 0x00007f16c0d04240
0x00007f16c0d04260/1: 0x000000000049b128 0x0000000000000068
0x00007f16c0d162b0/2: 0x000000000049b1c8 0x00007f16c0d162a1 0x00007f16c0d16280
0x00007f16c0d162a0/1: 0x000000000049b128 0x0000000000000065
0x00007f16c0d262b0/2: 0x000000000049b1c8 0x00007f16c0d262a1 0x00007f16c0d26280
0x00007f16c0d262a0/1: 0x000000000049b128 0x000000000000006c
0x00007f16c0d362b0/2: 0x000000000049b1c8 0x00007f16c0d362a1 0x00007f16c0d36280
0x00007f16c0d362a0/1: 0x000000000049b128 0x000000000000006c
0x00007f16c0d462b0/2: 0x000000000049b1c8 0x00007f16c0d462a1 0x00007f16c0d46280
0x00007f16c0d462a0/1: 0x000000000049b128 0x000000000000006f
0x00000000006fb188/1: 0x000000000049b1a8
0x00007f16c0dfd4b8/2: 0x000000000049b1c8 0x00000000006fbad1 0x00007f16c0dfd51a
0x00000000006fbad0/1: 0x000000000049b148 0x0000000000000068
0x00007f16c0dfd518/2: 0x000000000049b1c8 0x00000000006fba61 0x00007f16c0dfd552
0x00000000006fba60/1: 0x000000000049b148 0x0000000000000065
0x00007f16c0dfd550/2: 0x000000000049b1c8 0x00000000006fbb11 0x00007f16c0dfd56a
0x00000000006fbb10/1: 0x000000000049b148 0x000000000000006c
0x00007f16c0dfd568/2: 0x000000000049b1c8 0x00000000006fbb11 0x00007f16c0dfd582
0x00000000006fbb10/1: 0x000000000049b148 0x000000000000006c
0x00007f16c0dfd580/2: 0x000000000049b1c8 0x00000000006fbb41 0x00000000006fb189
0x00000000006fbb40/1: 0x000000000049b148 0x000000000000006f
0x00000000006fb188/1: 0x000000000049b1a8
```
Shell

and we see that after a run of the garbage collector, the closures for the 'l' are identical (both point to 0x00000000006fbb11), and that all character closures lie in the static code area. This is a speical optimization for ASCII Chars and Int with absolute value below 16, though.

Now we should be able to estimate the memory consumption of our string program. We have 100000000 bytes, each of which are stored in one Char. But as they are all from the ASCII-range, they do not increase the heap space usage. But the list itself needs three words á 8 bytes for every cell, which accounts nearly all of the observed 2521MB. (Why not twice that much? Because the garbage collector does not copy the whole memory every time, but separates it into generations, which we won't delve into now.)

At this point it should be clear that and why the standard String type is *not* suitable for writing fast and memory efficient code. There are good alternatives, in particular ByteString for raw sequences of bytes and Text for unicode text.

1.2 Functions

Now we turn to the second kind of closure, namely functions. These look quite differently in the interpreter, so we experiment using a program that we can compile:

```haskell
import System.Environment
import GHC.HeapView
import Utils

main = do
    let f = map
    viewClosure f
    let g toB = toB || not toB
    viewClosure g
    a <- getArgs
    let h = (++ a)
    print (asBox a)
    viewClosure h
```
Haskell

That program produces the following output:

```
$ ghc --make FunClosures.hs -O && ./FunClosures  1 2 3
0x00000000006f3090/2: 0x0000000000420dd8
0x00000000006ef7f0/1: 0x0000000000406408
0x00007f73f8d0e880/2
0x00007f73f8d10348/1: 0x0000000000406498 0x00007f73f8d0e882
```
Shell

Plain functions lie in the static code area and contain precisely one pointer to somewhere. The same holds for locally defined functions. A bit more interesting is the function h: It uses the value of a local variable (a). This means that the code main creates a new function closure on the heap, consisting of two words: One pointing to the static area with the function code, and one reference to the value of the variable a.

1.3 Thunks

So much about functions, let's look at thunks. In a way these are nothing but functions with parameters. As we now want to analyze more complex examples, we stop looking at the raw memory and start using ghc−vis, a tool written by my bachelor student Dennis Felsing. We load the following code from the module InfLists:

```haskell
infList f x = f x : infList f x
l = infList (+19) 23
```
Haskell

using

```
Prelude InfLists> :script /home/jojo/.cabal/share/ghc-vis-0.1/ghci
Prelude Inflists> :vis
Prelude Inflists> :switch
Prelude Inflists> :view l
```
GHCi

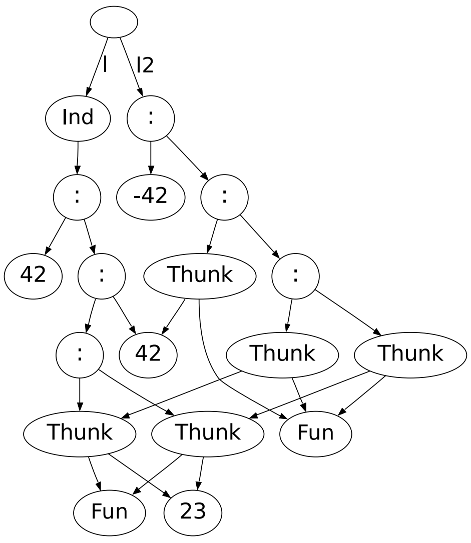
and click around on the then visible tree. Some observations:

- The list seems to really unwind infinitely.
- The number 42 is re-calculated every time, and new memory is allocated for each invocation.

It is also educating to look at

```
nl = map negate l
```
Haskell

and see how evaluation of the one list influences the other.

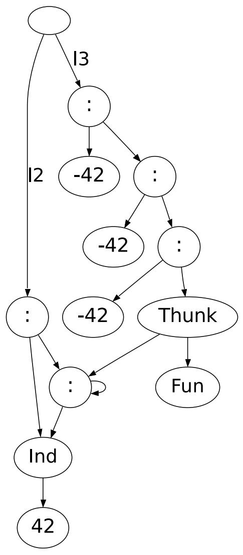


For comparison, consider this function which – semantically – does the same:

```
infList2 f x = let l = f x : l in l
```
Haskell

We look at l3 = infList2 (+19) (23::Int) using :view. This time, the cons cell and the number 42 are being calculated only once, and afterwards the tail of the list points to the list itself. This way, Haskell indeed manages to squeeze an infinite list into a finite amount of memory!

Unfortunately, such self-referential data structures are fragile when one tries to "change" them: A simple map negate l2 destroys the structure, as we can see here:



At this point I'd like to get back to the second example from the beginning. We can now use ghc−vis to find the cause of the Stack Overflow:

```
*Count> :script /home/jojo/.cabal/share/ghc−vis−0.1/ghci
*Count> :vis
*Count> :switch
*Count> let x = count 0 [1..5]
*Count> :view x
```
GHCi

We see a lot of thunks, which modify the 0 one after another. These are precisely the thunks representing the (+1) computation, and it makes sense: No computation should be performed until we actually need the result. Nevertheless the computation needs to be stored. If we click on one of these thunks, the whole stack below collapses in one go.

Better is this code:

```
count2 n (x:xs) = (count2 $! n+1) xs
count2 n [] = Just n
```
Haskell

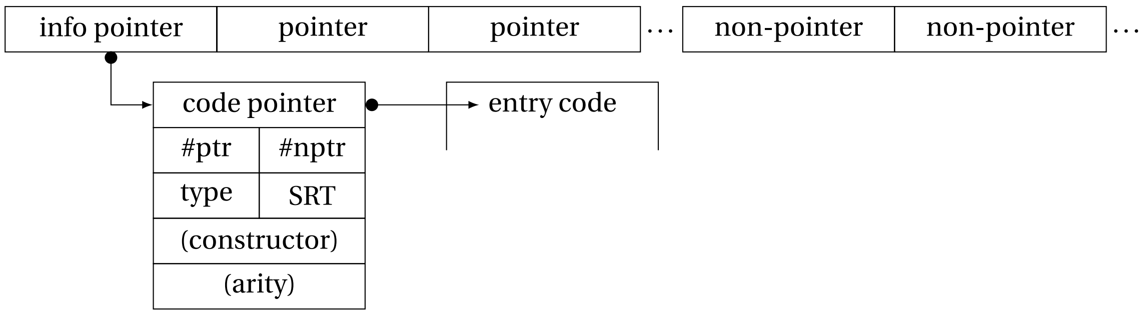
The operator $1 ensures that the argument (here n+1) is evaluated before it calls the function (here count2). This way we will always pass a number, and not a thunk. There is no stack overflow, the memory footprint is constant (and with ghc−vis we can't see anything).

```
*Count> let x = count2 0 [1..100000000]
*Count> x
100000000
```
GHCi

For the sake of honesty: For such a simple program the compiler would have done that change for use, had we not made the function artificially non-strict using Just.

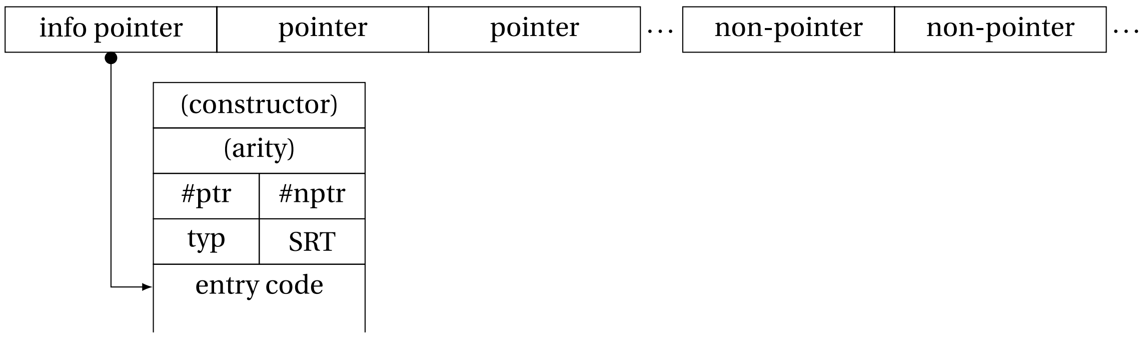
2 The info pointer

Finally I want to discuss this so far incomprehensible first word of every closure. You probably have observed that it always points into the static code area. And you surely noticed that the information about which constructor we happen to have, or where the executable code of a function is, must also be stored somewhere. All that, and a bit more, is hidden in the info pointer:



At this point I'd like to point out to you that the parameters of a constructor (resp. the free variables of a function) are always arranged so that first all pointers and then all other values are stored. This way the size and layout of a closure can be kept in two half-words. This is what the garbage collector looks at, and it thus does not special need code for every constructor.

The most common thing happening to a closure is that it is "entered". Therefore, reality looks yet a bit different: The info pointer in the closure points directly to the beginning of the function code, and the compiler puts the table right in front of that. That is a somewhat unorthodox mixing of data and code, but a compiler is allowed to do that.



3 The prime number sieve

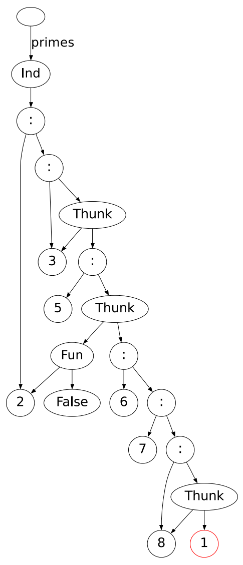
If time allows, I'd like to demonstrate another program's execution, namely this well-known, very elegant definition of a list of all prime numbers:

```
module Sieve where

primes :: [Integer]
primes = sieve [2..]
```

```
  where
    sieve (p:xs) = p : sieve [x|x <- xs, x `mod` p > 0]
                                                        Haskell
```

Here we can see very nicely the ever growing list of thunks, each of which removes, for one prime number, all multiple of its from the list:



4 Conclusion

This concludes my talk. We broke through a large number of abstractions layers of Haskell to have an unobstructed view on the memory. We saw that the values are stored somewhat clean and efficient after all. We understood why String is so expensive and how to store an infinite list in a few bytes. I hope this talk has helped you to understand better, why your Haskell programs run as they run, and to know how to make them run better.

5 References

- What I showed here is also known as the *Spineless, tagless G-machine*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.3729
- The GHC wiki describes the current implementation, in particular http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects.
- We have been using my library ghc−heap−view (http://hackage.haskell.org/package/ghc-heap-view) and Dennis Felsing's ghc−vis (http://felsin9.de/nnis/ghc-vis/).
- The video was generated from Haskell code using diagrams, based on an earlier version that I created using Synfig. (http://www.synfig.org/)