Haskell Bytes

Eine geführte Tour durch den Hauptspeicher eines Haskell-Programms

Joachim Breitner*

3. Juli 2013
fun-fr, Freiburg

Haskell ist eine tolle Programmiersprache; ich schätze ihr das alle schon wisst, sonst wärt ihr wohl nicht in meinem Vortrag. Aber manchmal wird man von Haskell auch enttäuscht. Nehmen wir folgenden Code:

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

und füttern ihn mit einer 100MB großen Textdatei. Mit dem Parameter `-RTS -t` können wir uns Statistiken anzeigen lassen und erfahren, dass das Programm 2521MB Speicher braucht – über 24× zu viel!

Ein anderes, klassisches Beispiel für unerwartetes Laufzeitverhalten ist der folgende Code, der die Länge einer Liste zählt:

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

Im Interpreter sehen wir, dass der Code unnötig viel Speicher verbraucht und dann mit einem Stack Overflow abbricht:

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

Manchmal aber überrascht uns Haskell auch positiv. Wenn wir, wieder beim ersten Programm, die dritte Zeile löschen und

*mail@joachim-breitner.de, http://www.joachim-breitner.de/. Diese Arbeit wurde durch ein Promotionsstipendium der Deutschen Telekom Stiftung gefördert.

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

laufen lassen haben wir plötzlich einen Speicherverbrauch von 2MB – das ist 50× besser als erwartet! Und dann gibt es ja auch noch die unendlichen Listen in Haskell…

1 Die Akteure

Fragen wir uns also, was ein Haskell-Programm während der Ausführung alles im Speicher vorhalten muss. Zuerst wären da natürlich die eigentlichen *Daten*, also Konstruktoren wie True oder Just oder (,), die wiederum andere Werte enthalten können. Weiter ist Haskell ja eine funktionale Sprache die sich dadurch auszeichnet, dass man Funktionen selbst wie Daten behandeln kann. Also müssen wir auch *Funktionen* speichern können. Und zuletzt ist Haskell lazy, das heißt es gibt Werte die noch nicht ausgewertet sind. Diese heißen *Thunks*. Das sind schon mal die wichtigsten; allgemein spricht man hierbei von *Closures*.

Bevor wir nun in den Speicher eines Haskell-Programms reinschauen überlegen wir noch, was denn jeweils zu so einer Funktion gespeichert werden soll.

- Der Typ eines Wertes: Der ist tatsächlich nicht nötig! Das Typsystem stellt sicher dass jeglicher Code stets Werte von dem Typ vorfindet, den er erwartet, er kann also blind drauf vertrauen. Das ist ganz anders als z.B. in Python!
- Welcher Konstruktor: Ja, das muss man speichern, zumindest für Typen mit mehreren, etwa bei **data** Maybe a = Just a | Nothing.
- Die Parameter des Konstruktors: Natürlich!
- Bei Funktionen: Der Code der Funktion.
- Nicht vergessen bei Funktionen und Thunks: Freie Variablen!

1.1 Konstruktoren

Schauen wir mal an was wir vorfinden, wenn wir mit GHCi etwas rumspielen, und fangen mit einer einfachen Zahl an:

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

Wir sehen also dass wir für einen Integer-Wert zwei Worte, die auf meiner Maschine jeweils 8 Bytes groß sind, benötigen. Das zweite speichert offensichtlich die eigentliche Zahl.

Wie sieht es mit Zeichen, also Charakters aus?

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

Auch diese benötigen zwei Wörter, also 16 statt einem Byte!

Kommen wir nun zu algebraischen Datentypen, und packen eins in Just:

```
*Utils> let jeins = Just eins
*Utils> viewClosure jeins
0x00007f9c7b082710: 0x00000000420DC920 0x00007f9c7a3337f8                GHCi
```

Beachte dass im Wert Just eins keine Kopie von eins gespeichert ist, sondern eine Referenz drauf.

Nun wollen wir verstehen, warum unser erstes Beispielprogramm so viel Speicher brauchte. Dazu erinnern wir uns dass der Typ String in Haskell nur ein Alias für [Char], also Listen von Zeichen, ist.

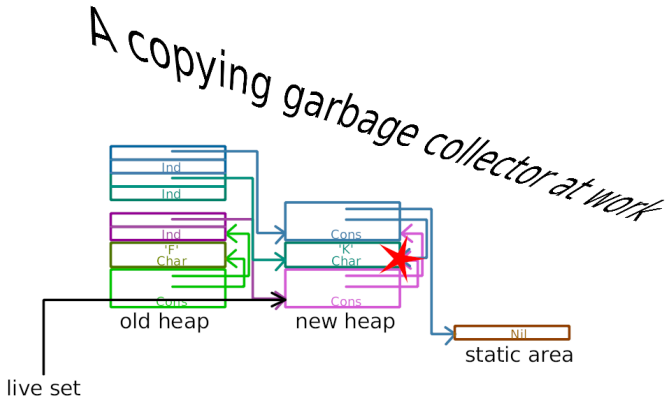
```
*Utils> let hallo = "hallo"
*Utils> viewListClosures hallo
0x00007f9c7ba21fa0: 0x0000000040502668 0x00007f9c7ba21f91 0x00007f9c7ba21f70
0x00007f9c7ba21f90/1: 0x0000000040502548 0x0000000000000068
0x00007f9c7ba77b18: 0x0000000040502668 0x00007f9c7ba77b09 0x00007f9c7ba77ae8
0x00007f9c7a1a9768/1: 0x0000000040502548 0x0000000000000061
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
```

Wir sehen nun dass die Liste *"hallo"* erst einmal aus einem Closure mit drei Wörtern besteht. Das zweite ist ein Pointer auf ein Closure für 'h' und das dritte ein Pointer auf *"allo"*, und so weiter, bis wir bei der leeren Liste [] angekommen sind. Man beachte die deutlich andere Adresse: [] gibt es global nur einmal und lebt im statischen Codebereich.

An der Stelle will ich demonstrieren warum man in GHC keinen Zugriff auf die eigentlichen Pointer haben sollte:

```
*Utils> System.Mem.performGC
*Utils> viewListClosures hallo
0x00007f9c7a1510a8: 0x0000000040502668 0x00007f9c7a151389 0x00007f9c7a151372
0x00007f9c7a151388/1: 0x0000000040502548 0x0000000000000068
0x00007f9c7a151370: 0x0000000040502668 0x00007f9c7a1516f1 0x00007f9c7a1516da
0x00007f9c7a1516f0/1: 0x0000000040502548 0x0000000000000061
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
```

Der gleiche Wert, plötzlich woanders! GHC verwendet standardmäßig einen kopierenden Garbage-Collector – alle noch benötigten Werte werden in einen komplett neuen Speicherbereich kopiert und der alte am Stück freigegeben. Das ist schneller und genauer als z.B. Referenzen zu zählen, aber dafür braucht man auch doppelt so viel physischen Speicher. Und für die denen ein Bild mehr als tausend Worte sagt habe ich das noch als Video visualisiert.



Es gibt noch einen weiteren Effekt den man hier jetzt gesehen hätte, würden wir das Programm wirklich ausführen (und nicht im Interpreter laufen lassen). Dann wäre die Ausgabe nämlich:

```
$ ./HelloGC
0x00007f16c0d04270/2: 0x0000000000049b1c8 0x00007f16c0d04261 0x00007f16c0d04240
0x00007f16c0d04260/1: 0x0000000000049b128 0x0000000000000068
0x00007f16c0d162b0/2: 0x0000000000049b1c8 0x00007f16c0d162a1 0x00007f16c0d16280
0x00007f16c0d162a0/1: 0x0000000000049b128 0x0000000000000061
0x00007f16c0d262b0/2: 0x0000000000049b1c8 0x00007f16c0d262a1 0x00007f16c0d26280
0x00007f16c0d262a0/1: 0x0000000000049b128 0x000000000000006c
0x00007f16c0d362b0/2: 0x0000000000049b1c8 0x00007f16c0d362a1 0x00007f16c0d36280
0x00007f16c0d362a0/1: 0x0000000000049b128 0x000000000000006c
0x00007f16c0d462b0/2: 0x0000000000049b1c8 0x00007f16c0d462a1 0x00007f16c0d46280
0x00007f16c0d462a0/1: 0x0000000000049b128 0x000000000000006f
0x00000000006fb188/1: 0x0000000000049b1a8
0x00007f16c0dfd4b8/2: 0x0000000000049b1c8 0x00000000006fbad1 0x00007f16c0dfd51a
0x00000000006fbad0/1: 0x0000000000049b148 0x0000000000000068
0x00007f16c0dfd518/2: 0x0000000000049b1c8 0x00000000006fba61 0x00007f16c0dfd552
0x00000000006fba60/1: 0x0000000000049b148 0x0000000000000061
0x00007f16c0dfd550/2: 0x0000000000049b1c8 0x00000000006fbb11 0x00007f16c0dfd56a
0x00000000006fbb10/1: 0x0000000000049b148 0x000000000000006c
0x00007f16c0dfd568/2: 0x0000000000049b1c8 0x00000000006fbb11 0x00007f16c0dfd582
0x00000000006fbb10/1: 0x0000000000049b148 0x000000000000006c
0x00007f16c0dfd580/2: 0x0000000000049b1c8 0x00000000006fbb41 0x00000000006fb189
0x00000000006fbb40/1: 0x0000000000049b148 0x000000000000006f
0x00000000006fb188/1: 0x0000000000049b1a8
```

Shell

und wir sehen dass nach dem Garbage Collector die Closures für das `'|'` identisch sind (beide zeigen nun auf 0x00000000006fbb11) und im statischen Codebereich liegen. Das ist eine Optimierung speziell für Chars im ASCII-Bereich und für Ints mit Betrag bis zu 16.

Nun müssten wir den Speicherverbrauch von string abschätzen können. Wir haben 100000000 Bytes, die jeweils in einem Char abgespeichert werden. Da die aber alle aus dem ASCII-Bereich sind, kosten sie nichts. Die Liste selbst jedoch braucht für jede Zelle drei Wörter á 8 Bytes, das sind fast die beobachteten 2521MB Speicherverbrauch. (Warum nicht das Dopplete? Weil der Garbage Collector nicht immer den gesamten Speicher kopiert sondern ihn in Generationen aufteilt – was wir hier nicht weiter vertiefen wollen.)

An dieser Stelle sollte klar sein dass sich der eingebaute String-Datentyp *nicht* für schnellen und speichereffizienten Code eignet. Es gibt Alternativen, allen voran ByteString für rohe Bytes und Text für Unicode-Text.

1.1.1 Boxed und Unboxed types

An dieser Stelle ein kleiner Exkurs zum Thema boxed und unboxed, was man nicht kennen muss um Haskell zu programmieren, aber gut zu wissen ist wenn man gute Performance braucht. Wir haben bereits gesehen, dass Integers nicht nur als dem eigentlich Wert gespeichert werden, sondern in eine Box mit einem Pointer davor (dem Info-Pointer) gepackt werden. Wenn man lazy evaluation haben will geht das nicht anders.

Manchmal will man aber gar keine lazy evaluation. Dann ist das unnötiger Overhead. Wir erinnern uns an die Variable jeins vom Typ Just Int – da wurde im Just-Konstruktor nicht etwa der Integer-Wert gespeichert, sondern nur ein Pointer auf einen Int-Konstruktor.

Mit GHC geht das auch anders, wie folgender Code demonstriert:

```haskell
module UnpackedFields where

data Pair1 = Pair1 !Int !Int
    deriving Show
data Pair2 = Pair2 {-# UNPACK #-} !Int {-# UNPACK #-} !Int
    deriving Show
data Pair3 = Pair3 {-# UNPACK #-} !Int !Int
    deriving Show
data Pair4 = Pair4 {-# UNPACK #-} !(Int,Int)
    deriving Show
data Pair5 = Pair5 {-# UNPACK #-} !Pair3 {-# UNPACK #-} !Pair3
    deriving Show
```
Haskell

Diesen muss man, damit das überhaupt was bringt, mit −O kompilieren. Im Interpreter erzeugen wir Werte der verschiedenen Varianten und stellen sicher, dass sie komplett ausgewertet sind:

```
Prelude Utils UnpackedFields> :m UnpackedFields Utils
Prelude Utils UnpackedFields> let eins = 1 :: Int
Prelude Utils UnpackedFields> let zwei = 2 :: Int
Prelude Utils UnpackedFields> let p1 = Pair1 eins zwei
Prelude Utils UnpackedFields> let p2 = Pair2 eins zwei
Prelude Utils UnpackedFields> let p3 = Pair3 eins zwei
Prelude Utils UnpackedFields> let p4 = Pair4 (eins, zwei)
Prelude Utils UnpackedFields> let p5 = Pair5 p3 p3
Prelude Utils UnpackedFields> (p1,p2,p3,p4,p5)
(Pair1 1 2,Pair1 1 2,Pair3 1 2,Pair4 (1,2),Pair5 (Pair3 1 2) (Pair3 1 2))
Prelude Utils UnpackedFields> System.Mem.performGC
```
GHCi

Die erste Variante ist ein ganz normaler Datentyp, der einfach Pointer auf die enthaltenen Werte speichert:

```
Prelude Utils UnpackedFields> viewClosure eins
0x00007ffafd58f4e0: 0x0000000041926608 0x0000000000000001
Prelude Utils UnpackedFields> viewClosure zwei
0x00007ffafd58f4f0: 0x0000000041926608 0x0000000000000002
Prelude Utils UnpackedFields> viewClosure p1
0x00007ffafd5909d8/1: 0x0000000047ad8750 0x00007ffafd58f4e1 0x00007ffafd58f4f1
```
GHCi

Beim zweiten haben wir dagegen den gewünschten Effekt, dass die Werte direkt im Pair2-Konstruktor gespeichert werden. Wir sparen also zwei Wörter und die Daten liegen näher beieinander:

```
Prelude Utils UnpackedFields> viewClosure p2
0x00007ffafd590840/1: 0x0000000047ad8710 0x0000000000000001 0x0000000000000002
```
GHCi

Man kann das auch mischen, bei Pair3 haben wir einen Wert direkt im Konstruktor und einen über einen Pointer. Man beachte dass die Reihenfolge im Speicher nicht notwendigerweise mit der Reihenfolge im Quellcode übereinstimmt. Warum das so ist sehen wir später.

```
Prelude Utils UnpackedFields> viewClosure p3
0x00007ffafd590a68/1: 0x0000000047ad86d0 0x00007ffafd58f4f1 0x0000000000000001
```
GHCi

Das Ganze geht auch mit anderen Datentypen als Int – genauer gesagt mit solchen Datentypen, die genau einen Konstruktor haben. Wie etwa die Standard-Tupel:

```
Prelude Utils UnpackedFields> viewClosure p4
0x00007ffafd590a80/1: 0x0000000047ad8690 0x00007ffafd58f4e0 0x00007ffafd58f4f0
```
GHCi

Und das es geht sogar verschachtelt: Pair5 entpackt die Parameter von zwei Pair3Konstruktoren, also insgesamt zwei Pointer und zwei Werte:

```
Prelude Utils UnpackedFields> viewClosure p5
0x00007ffafd590a98/1: 0x0000000047ad8650 0x00007ffafd58f4f1 0x00007ffafd58f4f1
0x0000000000000001 0x0000000000000001
```
GHCi

Das Unboxen von Parameter kann die Performance deutlich erhöhen, weil man damit viel Speicher sparen kann. Allerdings kann es auch nach hinten los gehen, insbesondere wenn man die Werte wieder aus den Konstruktoren herausnimmt und damit etwas machen will, was nur mit eingepackagen Werten geht, etwa in eine andere Datenstruktur packen oder einer nicht-strikten Funktion übergeben:

```
Prelude Utils UnpackedFields> let usePair3 (Pair3 x y) = (x,y,x+y)
Prelude Utils UnpackedFields> let (a,b) = (usePair3 p3, usePair3 p3)
Prelude Utils UnpackedFields> (a,b)
((1,2,3),(1,2,3))
Prelude Utils UnpackedFields> System.Mem.performGC
Prelude Utils UnpackedFields> viewClosure zwei
0x00007ffafc81d138: 0x0000000041926608 0x0000000000000002
Prelude Utils UnpackedFields> viewClosure a
0x00007ffafc81fa48: 0x0000000041925420 0x00007ffafc81d510 0x00007ffafc81d139
0x00007ffafc81d521
Prelude Utils UnpackedFields> viewClosure b
0x00007ffafc81fa68: 0x0000000041925420 0x00007ffafc81d530 0x00007ffafc81d139
0x00007ffafc81d541
```
GHCi

Jeder Aufruf von usePair3 muss beide Integers in ein Standard-Tupel (,,) packen. Dazu müssen diese allerdings eingepackt werden. Mit dem zweiten Wert ist das kein Problem, der liegt ja schon

eingepackt vor und kann bei jedem Aufruf verwenden werden. Um aber die eins einzupacken muss jedesmal ein *neuer* Konstruktor angelegt werden – wenn man das oft macht geht der Vorteil des einpackens kaputt.

1.2 Funktionen

Wenden wir uns nun der nächsten Art von Closures zu, nämlich Funktionen. Weil es hier interpretiert recht anders funktioniert, als kompiliert, lasse ich folgendes Programm laufen:

```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
```

was zu folgender Ausgabe führt:

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

Einfache Funktionen liegen im statischen Code-Bereich und enthalten genau einen Pointer irgendwo hin. Das gilt auch für lokal definierte Funktionen. Interessant ist die Funktion h. Diese verwendet einen Wert (a) aus einer lokalen Variable. Das heißt der Code für main legt eine neue Funktionen-Closure auf dem Heap an, die zwei Wörter groß ist: Ein Verweis auf den statischen Code und eine Referenz auf den Wert von a.

1.3 Thunks

Das wars erstmal von den Funktionen und wir wenden uns Thunks zu, die in gewisser weise nichts anderes sind als Funktionen ohne Parameter. Da es jetzt langsam kompliziert wird schauen wir uns nicht mehr den Speicher direkt an, sondern verwenden ghc–vis, ein Tool das Dennis Felsing in seiner Bachelorarbeit bei mir entwickelt hat. Wir probieren folgenden Code im Modul InfLists

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

mittels

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

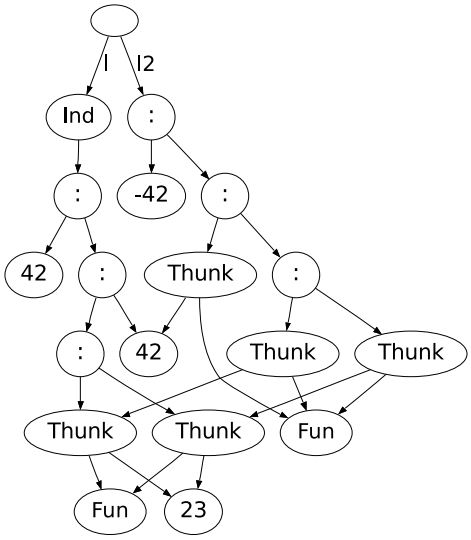
und klicken auf dem entstehenden Baum herum. Wichtige Beobachtungen:

- Die Liste scheint sich tatsächlich unendlich oft abzurollen.
- Der Wert 42 wird jedes mal neu berechnet und neu gespeichert.

Schön ist es jetzt auch noch

```
l2 = map negate l
```
Haskell

anzuschauen und zu beobachten, wie die Auswertung der einen Liste die andere beeinflusst.

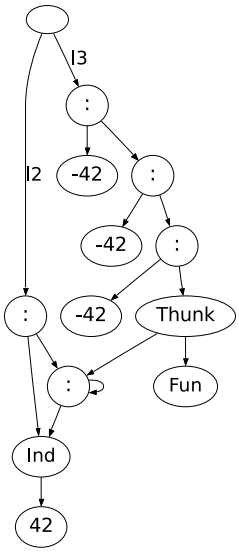


Zum Vergleich nehmen wir diese Funktion für unendliche Listen, die – semantisch – das gleiche macht:

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

und untersuchen `l3 = infList2 (+19) (23::Int)` mit `:view`. Tatsächlich wird hier die Cons-Zelle und die 42 nur jeweils einmal berechnet, danach zeigt der Tail der Liste wieder auf die Liste selbst. Auf diese Weise schafft es Haskell tatsächlich, eine unendliche Liste in endlich viel Speicher zu speichern!

Leider sind solche selbstbezüglichen Datenstrukturen fragil, wenn man sie „ändern“ will; schon ein einfaches `map negate l2` zerstört die Struktur, wie man hier gut sehen kann.



An dieser Stelle will ich wieder auf das zweite Beispiel vom Anfang zurückkommen. Jetzt können wir mit ghc−vis dem Stack Overflow auf den Grund gehen:

```
*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

Wir sehen ganze viele Thunks, die nacheinander die 0 modifizieren. Das sind eben die Thunks, die für (+1) stehen. Macht ja auch Sinn: Bevor wir die Zahl nicht benötigen, wird keine Berechnung ausgeführt. Trotzdem muss die Berechnung ja gespeichert werden. Klickt man auf einen der Thunks, kollabiert der Turm darunter in einem Rutsch.

Besser ist folgender Code:

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

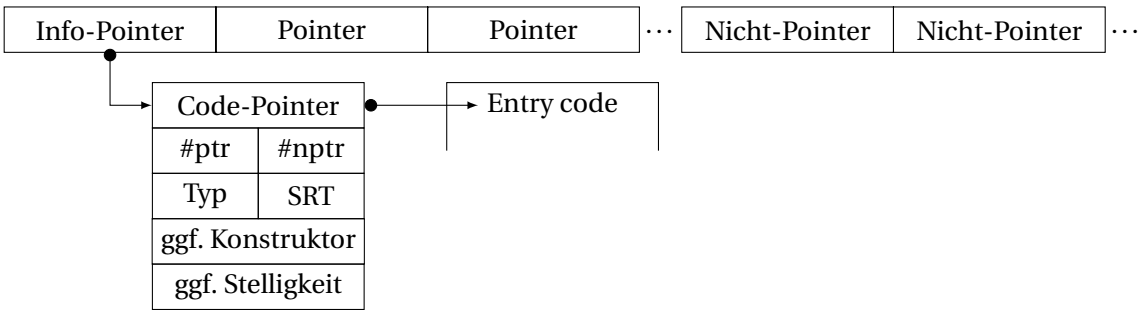
Der Operator $! sorgt dafür, dass das Argument (hier n+1) ausgewertet wird, bevor die Funktion (hier count2) aufgerufen wird. Dadurch wird stets nur eine Zahl, und kein Thunk, übergeben. Es tritt kein Stack Overflow auf, der Speicherverbrauch ist Konstant (und mit ghc−vis gibt es nichts zu sehen):

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

Nur der Ehrlichkeit halber: Bei diesem einfachen Programm hätte der Compiler das auch selbst gemacht, wenn ich die Funktion nicht durch das Just künstlich nicht-strikt im Argument gemacht hätte.

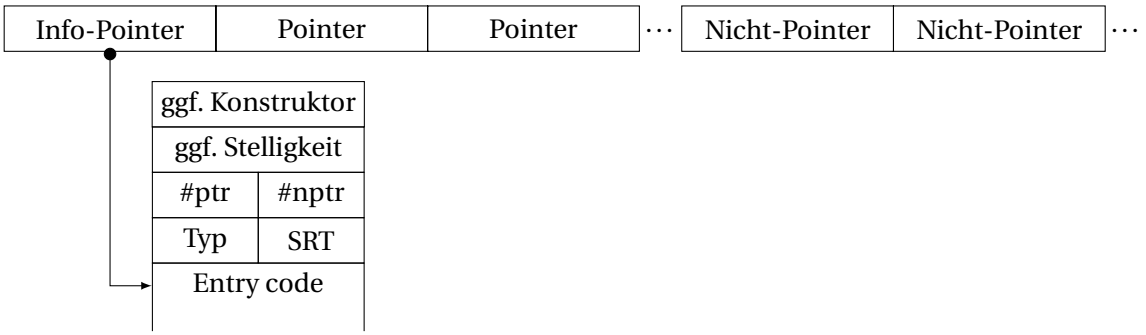
2 Der Info-Pointer

Zum Schluss will ich noch darauf eingehen, was das erste Wort jedes Closures ist. Ihr habt vielleicht schon beobachtet, dass es stets in den statischen Code-Bereich zeigt. Und euch ist sicher schon aufgefallen, dass die Information, welcher Konstruktor das denn gerade ist, oder wo der ausführbare Code einer Funktion steht, ja noch irgendwo stehen muss. All das, und noch viel mehr, versteckt sich hinter dem Info-Pointer:



Hier ist noch interessant dass die Parameter eines Konstruktors bzw. die freien Variablen einer Funktion stets so angeordnet sind, dass erst die Zeiger und dann die anderen Werte kommen. Damit kann die Größe und das Layout des Closures in zwei Halbworten gespeichert werden, die sich der Garbage Collector anschaut, ohne eigenen Code für jeden Konstruktor zu benötigen.

Was aber am häufigsten mit so einem Closure passiert ist, dass er ausgeführt wird. Daher sieht das ganze in Wirklichkeit nochmal anders aus. Der Zeiger im Closure zeigt direkt an den Anfang des Funktionscodes, und der Compiler legt die Tabelle direkt davor. Das ist zwar eine unübliche Mischung von Daten und Code, aber der Compiler darf sowas.



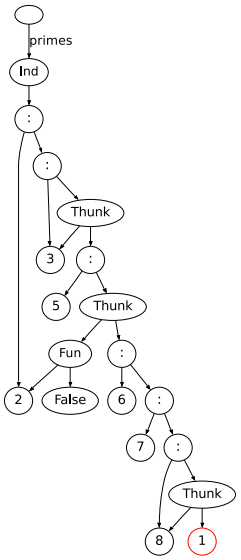
3 Das Primzahlensieb

Falls es die Zeit erlaubt will ich noch einem weiteren Programm bei der Ausführung zuschauen, nämlich dieser bekannten, sehr eleganten Definition der Primzahlen:

```haskell
module Sieve where

primes :: [Integer]
primes = sieve [2..]
  where
    sieve (p:xs) = p : sieve [x|x <- xs, x `mod` p > 0]
```
Haskell

Hier erkennt man schön die immer länger werdende Liste von Thunks, die jeweils für eine Primzahl deren Vielfache aus der Liste entfernt:



4 Fazit

Damit bin ich am Ende meines Vortrages. Wir haben uns die Mühe gemacht, all die vielen komfortablen Abstraktionsschichten, die uns Haskell bereitstellt, beiseite zu schieben und haben einen ungetrübten Blick auf den Speicher geworfen. Wir haben gesehen dass die Daten doch einigermaßen übersichtlich und effizient gespeichert werden. Wir haben auch gesehen, warum Strings so teuer sind und wie man eine unendlich lange Liste in wenigen Bytes speichert. Ich hoffe, dass euch dieser Vortrag hilft besser zu verstehen, warum eure Haskell-Programme so laufen wie sie laufen, und zu wissen, wie ihr sie besser laufen lassen könnt.

5 Referenzen

- Was wir hier gesehen haben ist in der Wissenschaft als *Spineless, tagless G-machine* bekannt. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.3729
- Das GHC-Wiki beschreibt die aktuelle Implementierung am pragmatischten, insbesondere http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects.

- Wir haben hier meine Bibliothek ghc-heap-view (http://hackage.haskell.org/package/ghc-heap-view) und Dennis Felsing's ghc−vis (http://felsin9.de/nnis/ghc-vis/) verwendet.
- Das Video habe ich mit Synfig erstellt. (http://www.synfig.org/)