Eine nicht ganz faule Show

Zusammenfassung

In diesem Vortrag wird an einem konkreten Beispiel gezeigt, dass man in manchen Situationen weder reine Bedarfsauswertung (Lazy Evaluation) noch vollständige Striktheit möchte. Dabei werden wir uns Gedanken über die Repräsentation im Speicher und den Garbage Collector, und beobachten unsere Verbesserungen mit Benchmarks.

1 Das Problem: Collatz

Das Problem, das wir behandeln wollen, ist eine Analyse der Collatz-Folge: Ausgehend von einer natürlichen Zahl k, wiederhole folgende Vorschrift: Auf n folgt $\frac{n}{2}$, wenn n gerade ist, sonst $3 \cdot n + 1$. Es wird vermutet dass man für jeden Startwert k irgendwann bei 1 ankommt. Für die Zahlen, die wir anschauen werden, weiß man es, weil sie kleiner als $20 \cdot 2^{58}$ sein werden. Wir wollen hier jetzt in dieser Folge die monotonen Teilfolgen bestimmen, also Folgen, die nur ab- oder aufsteigend sind, und deren Längen statistisch untersuchen – warum auch immer.

Also beginnen wir erst einmal, diese Folgen in einer Datei Collatz.hs zu implementieren. Die Berechnung des Nachfolgers ist noch einfach, und mit Haskell-Bordmitteln bekommt man daraus auch schnell die Collatz-Folge. Dass die letzte 1 da nicht mehr mit drin ist soll uns nicht stören.

```
collatz :: Integer → Integer
collatz n | even n    = n `div` 2
          | otherwise = 3*n +1

collatzSeries :: Integer → [Integer]
collatzSeries = takeWhile (/= 1) ∘ iterate collatz
```

Etwas kniffliger ist es, aus einer Folge von Zahlen die monotonen Teilfolgen herauszufinden. Hier ist ein Ansatz, der immerhin versucht noch eine generisch verwendbare Funktion dabei herauszuschlagen. Übrigens, Data.List.groupBy kann man hier nicht verwenden, da die Werte immer mit dem ersten Wert der Teilliste verglichen werden.

```
streaks :: [Integer] → [[Integer]]
streaks [] = []
streaks xs = let (this,rest) = oneStreak xs
             in this:streaks rest

oneStreak :: [Integer] → ([Integer], [Integer])
oneStreak [x] = ([x],[])
oneStreak l@(x:y:_) = splitWhile2 (λa b → a `compare` b == x `compare` y) l
```

```
splitWhile2 :: (Integer → Integer → Bool) → [Integer] → ([Integer], [Integer])
splitWhile2 p [x] = ([x],[])
splitWhile2 p (x:y:xs) | p x y = let (s,r) = splitWhile2 p (y:xs) in (x:s,r)
                       | otherwise = ([x],y:xs)
```

Nun wollen wir darauf Statistik betreiben. Wenn man sich auf Hackage umschaut, findet man das Paket vector-statistics, das schon viel mitbringt. Um das zu verwenden müssen wir die gewünschten Zahlen allerdings in einen Vektor packen:

```
rawData :: Integer → V.Vector Double
rawData param =
    V.fromList $
    map fromIntegral $
    map length $
    concatMap streaks $
    map collatzSeries $
    [1.. param]

quantile :: V.Vector Double → Double
quantile = weightedAvg 9 10
```

In eine weitere Datei Main.hs schreiben wir eine kleine Main-Funktion:

```
main = do
    putStrLn "Collatz quantile calculation"
    let vec = rawData 100000
    putStrLn $ "Number of streaks: " ++ show (V.length vec)
    putStrLn $ "Quantile: " ++ show (quantile vec)
```

Diese können wir nun testen und sehen, dass die Längen von 3664891 Teilfolgen verwurschtelt wurden.

2 QuickCheck

Brave Programmierer testen ihren Code, also wollen wir das auch machen. Die Collatz-Funktionen lassen sich schwer testen, aber streak ist eine nicht-triviale Funktion, deren Eigenschaften man nachprüfen kann. Insbesondere sollte die Verkettung der Teillisten wieder die ganze Liste geben, und jede Liste für sich sollte monoton sein:

```
propStreaksConcat xs = concat (streaks xs) == xs

propStreaksMonotone xs = all monotone (streaks xs)
  where monotone xs = ascending xs || descending xs
        ascending (x:y:ys) = x <= y && ascending (y:ys)
        ascending _ = True
        descending (x:y:ys) = x >= y && descending (y:ys)
        descending _ = True
```

Im Haskell Interpreter kann man die Tests schon benutzen, aber für ein vernünftiges Testprogramm brauchen wir noch eine Main-Funktion, die die Tests laufen lässt. Statt hier etwas eigenes zu basteln nehmen wir lieber das Paket test-framework her:

```
main = defaultMain
    [ testGroup "streaks"
        [ testProperty "concat" propStreaksConcat
        , testProperty "monotone" propStreaksMonotone
        ]
    ]
```

Das so erstellte Programm bietet ein paar nützliche Kommandozeilenparameter und eine schöne farbige Darstellung.

3 Benchmarks

Da wir gleich anfangen werden, an der Programmperformance zu schrauben, sollten wir jetzt einen Benchmark erstellen. Der letzte Schrei in der Haskell-Welt dabei ist die Bibliothek criterion, die Code systematisch testet, also genügend Durchlaufe macht, um statistisch aussagekräftig zu sein, die Zeit rausrechnet, die es dauert, die Uhrzeit zu messen und statistische Ausreißer ausweist.

Da wir mehrere Programmversionen vergleichen wollen, greifen am besten gleich auf die Bibliothek progression zurück, die mehrere Criterion-Benchmarks ausführt und mit früheren Ausführungen graphisch vergleicht. Das Ergebnis nach allen folgenden Code-Transformationen ist in Abbildung 4 zu sehen; als Code genügt dazu:

```
import Progression.Main
import Criterion
import Collatz

main = defaultMain $ bgroup "collatzQuant"
    [ bench "50" (whnf (quantile ∘ rawData) 50)
    , bench "500" (whnf (quantile ∘ rawData) 500)
    , bench "5000" (whnf (quantile ∘ rawData) 5000)
    ]
```

4 Speicherverbrauch

Wir haben also ein Programm das macht was es soll. Aber es braucht gefühlt deutlich zu viel Speicher. Stimmt das Gefühl? Wenn man das Programm mit −rtsopts compiliert und mit +RTS −s −RTS ausführt, bekommt man Informationen zum Speicherverbrauch:

```
./Main +RTS -s
Collatz quantile calculation
Number of streaks: 3664891
Quantile: 5.0
   4,561,344,624 bytes allocated in the heap
```

```
 1,954,742,784 bytes copied during GC
   417,433,456 bytes maximum residency (11 sample(s))
     3,473,352 bytes maximum slop
           796 MB total memory in use (0 MB lost due to fragmentation)

Generation 0:  8492 collections,     0 parallel,  2.76s,  2.76s elapsed
Generation 1:    11 collections,     0 parallel,  1.53s,  1.54s elapsed

INIT  time    0.00s  (  0.00s elapsed)
MUT   time    4.10s  (  4.11s elapsed)
GC    time    4.29s  (  4.30s elapsed)
EXIT  time    0.00s  (  0.03s elapsed)
Total time    8.39s  (  8.41s elapsed)

%GC time      51.1%  (51.1% elapsed)

Alloc rate    1,112,063,335 bytes per MUT second

Productivity  48.9% of total user, 48.8% of total elapsed
```

Interessant sind für uns erst mal die Werte „maximum residency“ und „total memory in use“. Maximum residency ist die maximale Größe des Heaps. Dass total memory da meist ungefähr das doppelte beträgt liegt daran, dass der GHC einen kopierenden Garbage Collector einsetzt, der alle Objekte, auf die nicht inzwischen verzichtet werden kann, in einen komplett neuen Bereich kopiert. Dass das hier viel gemacht wurde sieht man sowohl an dem Eintrag „bytes copied during GC“ als auch daran dass mehr als die Hälfte der Zeit mit dem Garbage Collector verbracht wurde. Da ist noch Verbesserungspotential.

Wieviel Speicherverbrauch hätten wir denn erwartet? Der Vektor, den wir berechnen, enthält 3664891 Zeiger auf ebenso viele Double-Werte. Auf meiner 64-Bit-Maschine ergibt das 58638256 bytes. Da liegen wir mit einem Faktor 7 drüber! Irgendwas geht also schief.

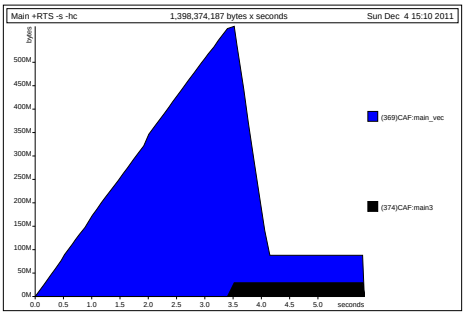
Um herauszufinden, was, schmeißen wir den Profiler an. Dazu kompilieren wir das Programm mit `-prof -caf-all` und geben bei den RTS-Optionen noch `-hc` an. Mit `hp2ps -c Main.hp` bekommen wir dazu ein Bild (Abbildung 1a), allerdings ein wenig hilfreiches: Aller Speicher wird beim Berechnen der Variable vec gebraucht.

Hilfreicher ist es zu wissen, welche Datentypen denn im Speicher so rumliegen, das erfahren wir mit der Option `-hy`. Jetzt sehen wir (Abbildung 1b), dass erst eine sehr lange Liste von Integern erstellt wird, bevor das Programm überhaupt erst anfängt, sie in Doubles umzuwandeln.

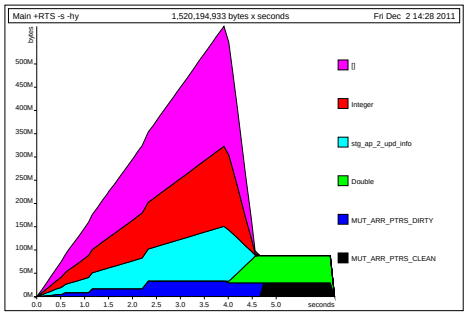
Das wäre auch nicht weiter verwunderlich, wenn wir eine Programmiersprache mit strikter Auswertung verwenden würden: Erst wird die Liste mit allen Collatz-Sequenzen erstellt (das ist die ganz arg lange), dann unterteilt (das ist zusammen ebenso lang), dann die Längen genommen und diese als Double in einen Vektor gesteckt. Aber wir haben ja eine Sprache mit Lazy Evaluation und könnten doch hoffen, dass er die große Integer-Liste nur soweit berechnet, wie er es im Moment braucht, also nur bis er weiß, wie lang die aktuelle monotone Teilfolge ist.

Mit dem GHCI-Debugger können wir versuchen einen Einblick zu gewinnen, in was da passiert:

```
*Collatz> let vec = rawData 10
*Collatz> :print vec
```



(a) Nach “Kostenstelle”

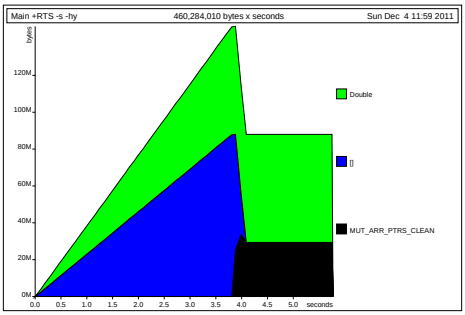


(b) Nach Typ

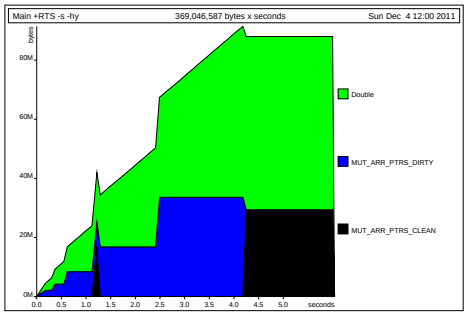
Abbildung 1: Profiler-Ausgabe des ersten Versuchs

```
vec = (_t53::V.Vector Double)   -- Hier ist der Vektor noch völlig unevaluiert
*Collatz> V.length vec
26                              -- Jetzt ist zumindest die Länge bekannt.
*Collatz> :print vec            -- Leider kann ghci nicht in den enthaltenen Array blicken.
vec = Data.Vector.Vector 0 26 <array>
*Collatz> let list = V.toList vec -- Deswegen erstellen wir nochmal eine Liste davon
*Collatz> :print list -- Die ist erstmal wieder unevaluiert.
list = (_t54::[Double])
*Collatz> length list -- Aber das kann man ändern
26
*Collatz> :print list -- Wir sehen, dass jeder Eintrag der Liste noch unevaluiert ist.
list = [(_t55::Double),(_t56::Double),(_t57::Double),
        (_t58::Double),(_t59::Double),(_t60::Double),(_t61::Double),
        (_t62::Double),(_t63::Double),(_t64::Double),(_t65::Double),
        (_t66::Double),(_t67::Double),(_t68::Double),(_t69::Double),
        (_t70::Double),(_t71::Double),(_t72::Double),(_t73::Double),
        (_t74::Double),(_t75::Double),(_t76::Double),(_t77::Double),
        (_t78::Double),(_t79::Double),(_t80::Double)]
*Collatz> vec V.! 1 -- Wir können jetzt die Einträge des Vektors auswerten
2.0
*Collatz> vec V.! 5
2.0
*Collatz> :print list -- Und sehen dass die sich einzeln evaluieren lassen.
list = [(_t81::Double),2.0,(_t82::Double),(_t83::Double),
        (_t84::Double),2.0,(_t85::Double),(_t86::Double),(_t87::Double),
        (_t88::Double),(_t89::Double),(_t90::Double),(_t91::Double),
        (_t92::Double),(_t93::Double),(_t94::Double),(_t95::Double),
        (_t96::Double),(_t97::Double),(_t98::Double),(_t99::Double),
        (_t100::Double),(_t101::Double),(_t102::Double),(_t103::Double),
        (_t104::Double)]
```

Was heißt das für uns? Wir erstellen einen Vektor, von dem wir zwar die Größe wissen, aber noch nicht den Inhalt. Aber bei unserem Problem ist es ja so, dass um die Größe des Vektors zu kennen,



(a) Mit deepseq



(b) Mit valueStrictList

Abbildung 2: Profiler-Ausgabe der zweiten Versuche

alle Collatz-Folgen durchgegangen werden müssen, um erstmal alle Streaks zu berechnen. Was noch nicht passiert ist, dank Lazyness, die Ermittlung der Länge der Streaks. Damit das später noch möglich ist, hält das Programm alle Streaks als Listen von Integern weiterhin vor, und zusammen ergibt das den großen Berg an Speicherverbrauch, den wir beobachten konnten.

5 Strikter?

Versuchen wir doch ob weniger Lazyness das Programm beschleunigt. Dazu fügen wir in der Funktion rawData ein deepseq ein:

```
rawData :: Integer → V.Vector Double
rawData param =
    V.fromList $
    (λl → l `deepseq` l) $
    map fromIntegral $
    map length $
    concatMap streaks $
    map collatzSeries $
    [1..param]
```

Und tatsächlich, das Programm braucht deutlich weniger Speicher, lief fast doppelt so schnell und der Anteil der Garbage-Collection an der Gesamtzeit hat sich halbiert. Auch unser Benchmark-Programm bestätigt dies, für große Listen, und die Ausgabe des Profilers (Abbildung 2a) zeigt das Bild das wir erwarten: Erst wird die Liste von Doubles aufgebaut, diese dann (schnell) in einen Vektor umgewandelt, der dann verarbeitet wird.

```
$ ./Main +RTS -s -RTS
./Main +RTS -s
Collatz quantile calculation
Number of streaks: 3664891
Quantile: 5.0
```

```
   4,406,349,376 bytes allocated in the heap
     433,641,208 bytes copied during GC
     142,680,584 bytes maximum residency (9 sample(s))
      27,016,248 bytes maximum slop
             322 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:  8206 collections,     0 parallel,  0.56s,  0.56s elapsed
  Generation 1:     9 collections,     0 parallel,  0.47s,  0.47s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    3.25s  (  3.25s elapsed)
  GC    time    1.03s  (  1.03s elapsed)
  EXIT  time    0.00s  (  0.03s elapsed)
  Total time    4.28s  (  4.28s elapsed)

  %GC time      24.0%  (24.0% elapsed)

  Alloc rate    1,354,357,105 bytes per MUT second

  Productivity  76.0% of total user, 75.9% of total elapsed
```

6 Doch weniger Strikt?

Aber wir sind immer noch recht weit weg vom erwarten Speicherverbrauch von knapp 60MB. Das ist nicht weiter verwunderlich, schließlich erzwingen wir ja dass eine komplette Liste mit unseren Daten im Speicher liegt, und Listen haben nicht gerade das effizienteste Speicherverhalten. Was muss passieren dass wir nie lange Listen im Speicher behalten? Wir müssen dafür sorgen dass die Werte, die nachher im Vektor gespeichert werden sollen, ausgewertet werden, ohne die gesamte Liste am Stück auszuwerten. Wir wollen also, dass die Liste strikt in den Werten, aber lazy in der Rest-Liste ist. Dazu kann man folgende Funktion basteln, und statt dem deepseq in rawData einfügen:

```
valueStrictList :: [a] → [a]
valueStrictList [] = []
valueStrictList (x:xs) = x `seq` (x:valueStrictList xs)
```

Diese Funktion lässt sich einfach per Quickcheck testen:

```
    , testGroup "valueStrictList"
        [ testProperty "id" (λxs → valueStrictList xs == (xs :: [Int]))
        ]
```

Und in der Tat, der maximale Speicherverbrauch geht nochmal um die Hälfte runter:

```
$ ./Main +RTS -s -RTS
./Main +RTS -s
Collatz quantile calculation
Number of streaks: 3664891
Quantile: 5.0
   4,737,472,992 bytes allocated in the heap
     188,661,504 bytes copied during GC
```

```
    67,187,664 bytes maximum residency (9 sample(s))
     2,637,016 bytes maximum slop
           145 MB total memory in use (24 MB lost due to fragmentation)

  Generation 0:  8830 collections,     0 parallel,  0.93s,  0.93s elapsed
  Generation 1:     9 collections,     0 parallel,  0.15s,  0.15s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    3.03s  (  3.04s elapsed)
  GC    time    1.08s  (  1.08s elapsed)
  EXIT  time    0.00s  (  0.03s elapsed)
  Total time    4.11s  (  4.12s elapsed)

  %GC time      26.2%  (26.2% elapsed)

  Alloc rate    1,560,407,803 bytes per MUT second

  Productivity  73.7% of total user, 73.7% of total elapsed
```

Es werden zwar immer noch gleich viele Listen-Elemente erzeugt, aber sie können alle schnell wieder vergessen werden, deswegen muss der der Garbage-Collector deutlich weniger kopieren. Das Laufzeitverhalten hat sich aber nicht deutlich verändert – man sieht dass der Garbage-Collector sehr effizient implementiert ist. Der Benchmark verrät, dass wir bei kurzen Eingaben etwas mehr Laufzeitverbesserung herausgeschlagen haben. Und im Profiler (Abbildung 2b) sehen wir gar keine Listen-Elemente mehr, dafür erkennt man gut die exponentielle Strategie der Vektor-Bibliothek, die den Vektor dynamisch vergrößert bis das Ende der Liste erreicht ist.

7 List Fusion

Den maximalen Speicherverbrauch werden wir nicht weiter optimieren können. Aber vielleicht können wir ja dafür sorgen, dass weniger Zwischenlisten erzeugt werden. Der Haskell-Compiler GHC bringt da eine gewisse Infrastruktur mit, die auf dem foldr/build-Prinzip basiert (Code aus GHC.Base):

```
{-# RULES "fold/build" forall k z (g :: forall b. (a → b → b) → b → b) .
      foldr k z (build g) = g k z
```

Die Funktion foldr kennt ihr ja sicherlich. Die Funktion GHC.Exts.build ist eigentlich eine Trivialität:

```
build :: forall a. (forall b. (a → b → b) → b → b) → [a]
build g = g (:) []
```

Aber damit eine Regel wie oben funktioniert, muss der Compiler ein build als solches erkennen.

Was müssen wir nun machen, damit wir von der List-Fusion profitieren? Möglichst viele unserer Funktionen müssen dafür ausgelegt sein. Funktionen wie map und [1.. param] sind das von Haus aus, aber zum Beispiel unser valueStrictList sicherlich nicht. Hier können wir nachhelfen. Erst schreiben wie sie mit foldr:

```
valueStrictList' = foldr (λx xs → x `seq` (x : xs)) []
```

und dann bauen wir die Liste nicht mit den Konstruktoren (:) und [], sondern parametrisieren den Code darüber, mittels build (was wir am besten gleich mit Quickcheck testen):

```
valueStrictList' xs = build $ λc n → foldr (λx xs → x `seq` (x `c` xs)) n xs
{-# INLINE valueStrictList' #-}
```

Die fold/build-Regel funktioniert nur, wenn foldr und build wirklich nebeneinander stehen; zur Sicherheit weisen wir den Compiler an, unsere Funktion zu inlinen.

Hat es was gebracht?

```
$ ./Main +RTS -s -RTS
./Main +RTS -s
Collatz quantile calculation
Number of streaks: 3664891
Quantile: 5.0
   4,405,830,208 bytes allocated in the heap
     188,100,872 bytes copied during GC
      67,189,176 bytes maximum residency (9 sample(s))
       2,415,832 bytes maximum slop
             145 MB total memory in use (24 MB lost due to fragmentation)

  Generation 0:  8197 collections,     0 parallel,  0.88s,  0.89s elapsed
  Generation 1:     9 collections,     0 parallel,  0.15s,  0.15s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    3.04s  (  3.05s elapsed)
  GC    time    1.04s  (  1.04s elapsed)
  EXIT  time    0.00s  (  0.03s elapsed)
  Total time    4.08s  (  4.09s elapsed)

  %GC time      25.4%  (25.4% elapsed)

  Alloc rate    1,445,493,259 bytes per MUT second

  Productivity  74.6% of total user, 74.5% of total elapsed
```

Nun, es wurden ein bisschen weniger Bytes alloziert, aber auf die Laufzeit hatte das keinen durchschlagenden Erfolg. Der Benchmark zeigt sogar an, dass die Laufzeit leicht gestiegen ist. Vielleicht ist valueStrictList gar nicht so kritisch. Die Funktion streaks dagegen verarbeitet ja doch recht viel an Listendaten – vielleicht haben wir dort mehr Erfolg?

Schreiben wir also streaks mit möglichst viel foldr und build und möglichst wenig eigener Rekursion und möglichst wenig Verwendung von (:) und []. Leider ist es so nicht möglich, die inneren Listen auch mit build zu erzeugen, aber immerhin klappt es für die äußeren Listen:

```
streaks' :: [Integer] → [[Integer]]
streaks' xs = build $ λc n →
    uncurry c $ foldr (streaksF c) ([],n) xs

streaksF :: ([Integer] → b → b) → Integer → ([Integer],b) → ([Integer],b)
```

```
streaksF c i  ([], ys) = ([i], ys)
streaksF c i  ([x], ys) = ([i,x], ys)
streaksF c i  ((x1:x2:xs),ys) = if i `compare` x1 == x1 `compare` x2
                                then (i:x1:x2:xs, ys)
                                else ([i],  (x1:x2:xs) `c` ys)
```

Das war ja eine nicht-triviale Änderung, wir sollten also per Quickcheck prüfen, ob das neue streaks wirklich das gleiche wie das alte macht. Wir fügen dazu also einen neuen Check in die Datei Check.hs ein, und schauen was passiert: testProperty *"streaks'"* (λxs → streaks xs == streaks' xs)

```
$ ./Check
streaks:
  concat: [OK, passed 100 tests]
  monotone: [OK, passed 100 tests]
[0,0,1]
  streaks': [Failed]
Falsifiable with seed 4212867887836746586, after 9 tests. Reason: Falsifiable
valueStrictList:
  id: [OK, passed 100 tests]

         Properties  Total
 Passed  3           3
 Failed  1           1
 Total   4           4
```

Und in der Tat, GHCi bestätigt uns: streaks [0,0,1] == [[0,0],[1]] während für die andere Funktion streaks' [0,0,1] == [[0],[0,1]] gilt. Aber das ist ja beides eine gültige Ausgabe im Sinne unserer (nicht sehr guten) Spezifikation. Also statt zu prüfen ob streaks und streaks' das gleiche tun, testen wir lieber streaks' selbst, und da ist dann alles gut.

Nachdem das geklärt ist wollen wir endlich sehen, ob sich was getan hat. Und tatsächlich, wir erreichen nicht nur, dass deutlich weniger Speicher alloziert wird, sondern auch eine merkliche Verbesserung des Laufzeitverhaltens, was auch wieder der Benchmark bestätigt:

```
$ ./Main +RTS -s -RTS
./Main +RTS -s
Collatz quantile calculation
Number of streaks: 3664891
Quantile: 5.0
   3,396,036,128 bytes allocated in the heap
     198,699,320 bytes copied during GC
      67,206,200 bytes maximum residency (9 sample(s))
       2,519,256 bytes maximum slop
             145 MB total memory in use (24 MB lost due to fragmentation)

  Generation 0:  6271 collections,     0 parallel,  0.79s,  0.79s elapsed
  Generation 1:     9 collections,     0 parallel,  0.15s,  0.15s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.61s  (  2.61s elapsed)
  GC    time    0.95s  (  0.95s elapsed)
  EXIT  time    0.00s  (  0.03s elapsed)
```

```
  Total time    3.56s  (  3.56s elapsed)

  %GC time      26.6%  (26.6% elapsed)

  Alloc rate    1,300,422,375 bytes per MUT second

  Productivity  73.3% of total user, 73.3% of total elapsed
```

Können wir noch mehr herauskitzeln? Laut Dokumentation funktioniert List-Fusion nicht mit concatMap, wohl aber mit List Comprehensions. Also schreiben wir unsere rawData Funktion um:

```
rawData param =
    V.fromList $
    valueStrictList' $
    [ fromIntegral (length j)
    | k ← [1.. param]
    , j ← streaks' (collatzSeries k)
    ]
```

Das bringt nur noch ein bisschen, 2,977,092,832 Bytes alloziert, 12 Sekunden gespart.

Wagen wir uns also an die letzte verwendete Funktion, die laut Dokumentation nicht gefused wird, takeWhile, und implementieren sie entsprechend:

```
takeWhile' :: (a → Bool) → [a] → [a]
takeWhile' p xs = build $ λc n → foldr (takeWhileF p c n) n xs
{−# INLINE takeWhile' #−}

takeWhileF p c n x xs = if p x then x `c` xs else n
```

Und ja, das hat nochmal einen deutlichen Schub gebracht:

```
$ ./Main +RTS -s -RTS
./Main +RTS -s
Collatz quantile calculation
Number of streaks: 3664892
Quantile: 5.0
   2,456,066,296 bytes allocated in the heap
     194,709,008 bytes copied during GC
      67,214,720 bytes maximum residency (9 sample(s))
       2,449,600 bytes maximum slop
             145 MB total memory in use (24 MB lost due to fragmentation)

  Generation 0:  4478 collections,     0 parallel,  0.68s,  0.68s elapsed
  Generation 1:     9 collections,     0 parallel,  0.15s,  0.15s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    2.34s  (  2.34s elapsed)
  GC    time    0.83s  (  0.83s elapsed)
  EXIT  time    0.00s  (  0.03s elapsed)
  Total time    3.18s  (  3.18s elapsed)
```

```
  %GC time      26.2%  (26.2% elapsed)

  Alloc rate    1,047,144,666 bytes per MUT second

  Productivity  73.7% of total user, 73.7% of total elapsed
```

8 Thinking outside the box

Wir hätten uns viel Arbeit sparen (aber auch hier weniger lernen) können wenn wir gleich eine bessere Datenstruktur für unseren Vektor gewählt hätten. Die vector-Bibliothek bietet nämlich nicht nur den generischen Vektor, der beliebige Werte speichern kann (gerne auch partiell angewandte Funktionen oder so), sondern auch sogenannte „unboxed vectors“. Bei diesen werden nicht Pointer konsekutiv im Speicher arrangiert, sondern direkt die eigentlichen Werte. Das geht nur für bestimmte primitive Datentypen, aber Double gehört dazu.

Um diesen platzsparenderen Vektor-Typ zu verwenden genügt es in Main.hs und Collatz.hs die Zeile import qualified Data.Vector as V zu ändern in import qualified Data.Vector.Unboxed as V – das wars. Und siehe da, die Performance steigt noch einmal gewaltg, während sich der Speicherverbrauch (wie erwartet) halbiert:

```
$ ./Main +RTS -s -RTS
./Main +RTS -s
Collatz quantile calculation
Number of streaks: 3664892
Quantile: 5.0
   2,503,194,088 bytes allocated in the heap
      15,606,256 bytes copied during GC
      33,602,176 bytes maximum residency (8 sample(s))
       2,085,432 bytes maximum slop
              72 MB total memory in use (10 MB lost due to fragmentation)

  Generation 0:  4571 collections,     0 parallel,  0.05s,  0.05s elapsed
  Generation 1:     8 collections,     0 parallel,  0.00s,  0.00s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    1.69s  (  1.69s elapsed)
  GC    time    0.05s  (  0.05s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    1.74s  (  1.74s elapsed)

  %GC time       2.8%  (2.8% elapsed)

  Alloc rate    1,481,399,952 bytes per MUT second

  Productivity  97.0% of total user, 97.1% of total elapsed
```

Hier können wir sogar auf valueStrictList verzichten, da ein unboxed Vektor stets strikt in den Werten ist, was den gleichen Effekt hat. Zum Vergleich und zum besseren Verständnis schauen wir uns auch das Speicher-Profil in Abbildung 3 dazu an: Lediglich der Array belegt Speicher.

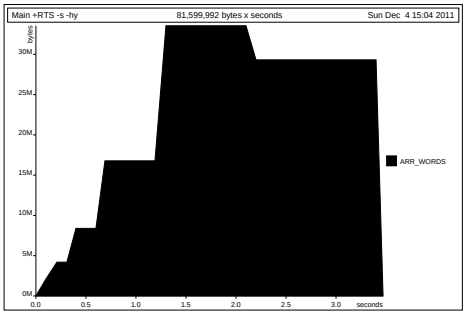


Abbildung 3: Profiler-Ausgabe mit einem unboxed Array

9 Verwendete Software

Wir haben folgende Programme und Pakete verwendet, sie sich jeweils auf http://hackage.haskell.org/package/<*name*> finden lassen:

- vector für effiziente und komfortable Array in Haskell.
- statistics für statistische Algorithmen auf Vektoren.
- QuickCheck zum automatischen Testen von Programeigenschaften.
- test-framework um die Tests schön laufen zu lassen.
- criterion um die Performance einzelner Funkionen zu messen.
- progression um die diese Ergebnisse zwischen verschiedenen Programmversionen zu vergleichen.

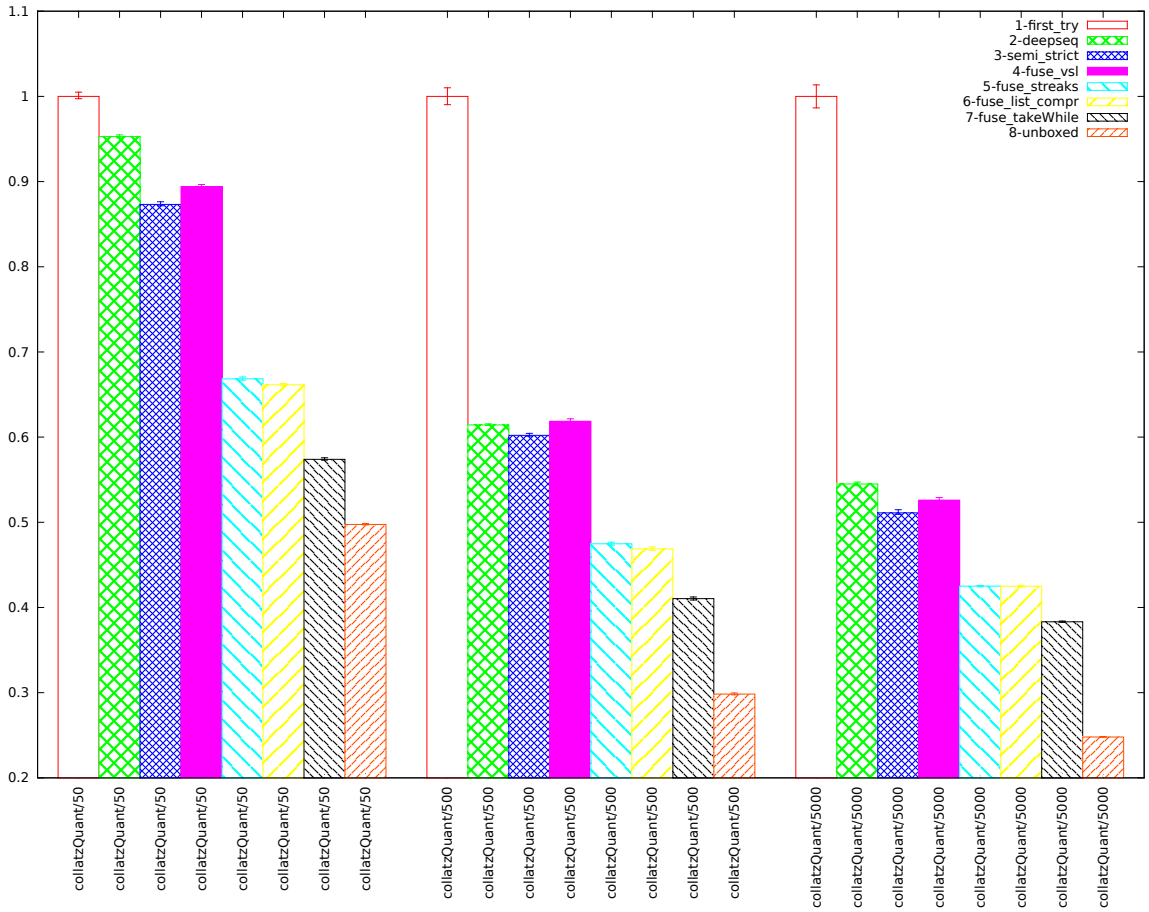


Abbildung 4: Die Ergebnisse des Benchmarks