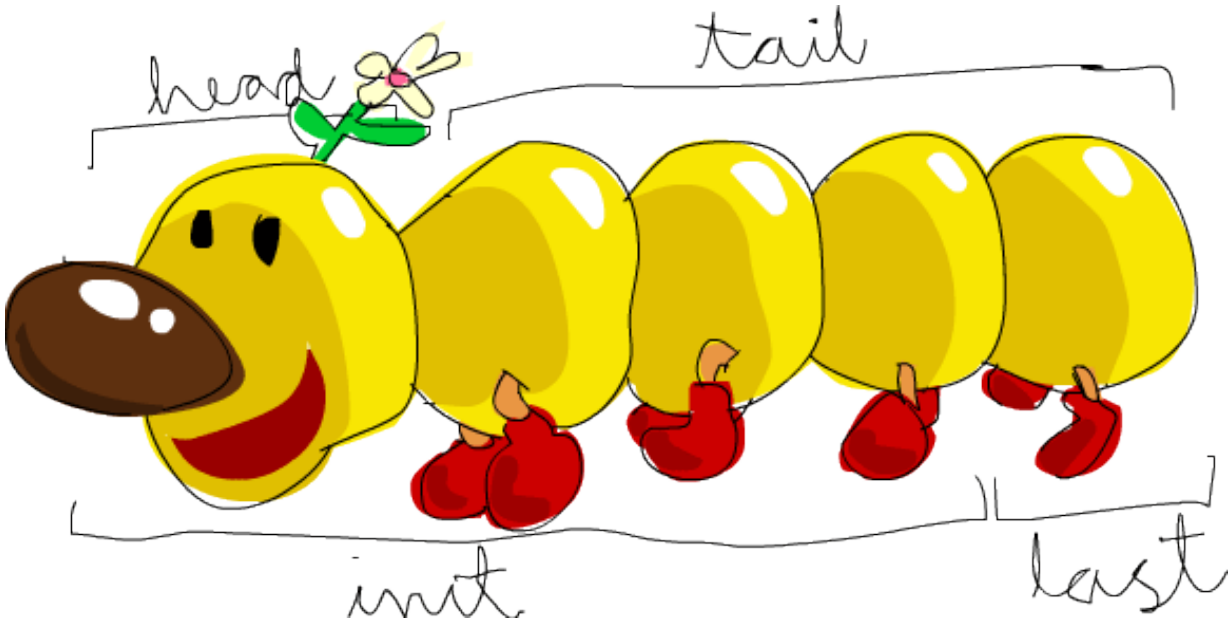
Rein funktionale Warteschlangen

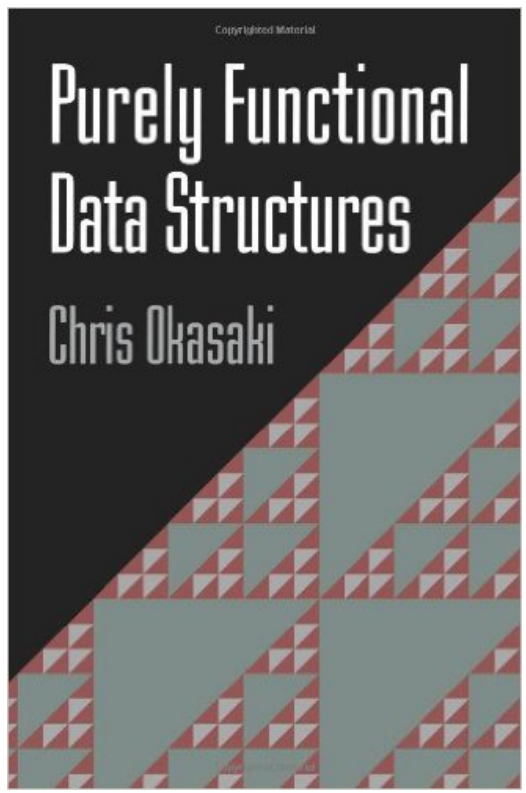


Tim Baumann

Curry Club Augsburg
28. Januar 2016

Literatur

- Standardwerk: Chris Okasaki, *Purely Functional Data Structures*
- Artikel von Edsko de Ries auf dem Well-Typed Blog:
  Efficient Amortised and Real-Time Queues in Haskell



Was sind Warteschlangen?

```haskell
class Queue q where
  empty  :: q a
  -- hinten anhängen:
  snoc   :: q a -> a -> q a
  -- vorne wegnehmen:
  uncons :: q a -> Maybe (a, q a)
  head :: q a -> Maybe a
  head = fst <$> uncons
  tail :: q a -> Maybe (q a)
  tail = snd <$> uncons
```

Was sind Warteschlangen?

```haskell
class Queue q where
  empty  :: q a
  -- hinten anhängen:
  snoc   :: q a -> a -> q a
  -- vorne wegnehmen:
  uncons :: q a -> Maybe (a, q a)
  head :: q a -> Maybe a
  head = fst <$> uncons
  tail :: q a -> Maybe (q a)
  tail = snd <$> uncons
```

Zum Beispiel:

```haskell
instance Queue [] where
  empty = []
  snoc xs x = xs ++ [x]
  uncons []
    = Nothing
  uncons (x:xs)
    = Just (x, xs)
```

Was sind Warteschlangen?

```haskell
class Queue q where
  empty  :: q a
  -- hinten anhängen:
  snoc   :: q a -> a -> q a
  -- vorne wegnehmen:
  uncons :: q a -> Maybe (a, q a)
  head :: q a -> Maybe a
  head = fst <$> uncons
  tail :: q a -> Maybe (q a)
  tail = snd <$> uncons
```

Zum Beispiel:

```haskell
instance Queue [] where
  empty = []
  snoc xs x = xs ++ [x]
  uncons []
    = Nothing
  uncons (x:xs)
    = Just (x, xs)
```

- *Problem*: `head (snoc (... (snoc empty x1) ...) xn)` benötigt $\mathcal{O}(n)$ Zeit in Haskell! In einer strikten Programmiersprache braucht der Aufruf snoc(xs, x) Zeit $\mathcal{O}(n)$.

Was sind Warteschlangen?

```haskell
class Queue q where
  empty  :: q a
  -- hinten anhängen:
  snoc   :: q a -> a -> q a
  -- vorne wegnehmen:
  uncons :: q a -> Maybe (a, q a)
  head :: q a -> Maybe a
  head = fst <$> uncons
  tail :: q a -> Maybe (q a)
  tail = snd <$> uncons
```

Zum Beispiel:

```haskell
instance Queue [] where
  empty = []
  snoc xs x = xs ++ [x]
  uncons []
    = Nothing
  uncons (x:xs)
    = Just (x, xs)
```

- *Problem*: `head (snoc (... (snoc empty x1) ...) xn)` benötigt $\mathcal{O}(n)$ Zeit in Haskell! In einer strikten Programmiersprache braucht der Aufruf snoc(xs, x) Zeit $\mathcal{O}(n)$.
- *Ziel*: Eine effiziente Warteschlangen-Datenstruktur für das funktionale Setting entwerfen. Die Operationen `snoc` und `uncons` sollen in konstanter Zeit laufen.

Amortisierte Laufzeitanalyse

Bei der amortisierten Laufzeitanalyse dürfen einzelne Operationen gelegentlich den für sie vorgeschriebenen Zeitrahmen sprengen, solange im Mittel die vorgegebene Zeitschranke eingehalten wird.

Amortisierte Laufzeitanalyse

Bei der amortisierten Laufzeitanalyse dürfen einzelne Operationen gelegentlich den für sie vorgeschriebenen Zeitrahmen sprengen, solange im Mittel die vorgegebene Zeitschranke eingehalten wird.

- Bankiers-Sichtweise: Es gibt ein (nichtnegatives!) Guthabenkonto. Jede Operation kann Geld abheben ($g < 0$) oder einzahlen ($g > 0$). Die Gesamtkosten einer Operation ist dann $g + t$, wobei t die Laufzeit die Operation ist. Der Kontostand muss zu jedem Zeitpunkt nichtnegativ sein.

Amortisierte Laufzeitanalyse

Bei der amortisierten Laufzeitanalyse dürfen einzelne Operationen gelegentlich den für sie vorgeschriebenen Zeitrahmen sprengen, solange im Mittel die vorgegebene Zeitschranke eingehalten wird.

- Bankiers-Sichtweise: Es gibt ein (nichtnegatives!) Guthabenkonto. Jede Operation kann Geld abheben ($g < 0$) oder einzahlen ($g > 0$). Die Gesamtkosten einer Operation ist dann $g + t$, wobei t die Laufzeit die Operation ist. Der Kontostand muss zu jedem Zeitpunkt nichtnegativ sein.
- Physiker-Sichtweise: Es gibt eine Potenzialfunktion $\phi : D \to \mathbb{N}$ (wobei D die Menge aller möglichen Zustände einer Datenstruktur ist). Die Kosten einer Operation, die $v \in D$ liest und $w \in D$ produziert, sind $\phi(w) - \phi(v) + t$.

Erinnerung: Umdrehen von Listen

Man kann eine Liste der Länge n in Zeit $\mathcal{O}(n)$ umdrehen:

```
reverse :: [a] -> [a]
reverse = go []
  where go as [] = as
        go as (x:bs) = go (x:as) bs
```

Banker's Queues

Idee: Speichere den vorderen und den hinteren Teil der Liste separat, den hinteren in umgekehrter Reihenfolge (damit man effizient anhängen kann). Dabei soll der vordere Teil immer größer als der hintere Teil sein. Falls der hintere Teil die gleiche Länge erreicht, so drehe ihn um und hänge ihn an den vorderen Teil an.

Banker's Queues

```
{-# LANGUAGE Strict #-}
```

Idee: Speichere den vorderen und den hinteren Teil der Liste separat, den hinteren in umgekehrter Reihenfolge (damit man effizient anhängen kann). Dabei soll der vordere Teil immer größer als der hintere Teil sein. Falls der hintere Teil die gleiche Länge erreicht, so drehe ihn um und hänge ihn an den vorderen Teil an.

```
data BQueue a = BQueue { front :: [a], frontLen :: Int
                       , rear  :: [a], rearLen  :: Int }

empty :: BQueue a
empty = BQueue [] 0 [] 0

mkBQueue :: [a] -> Int -> [a] -> Int
mkBQueue f fl r rl = if
  | fl > rl   = BQueue f fr r rl
  | otherwise = BQueue (f ++ reverse r) (fl + rl) [] 0
```

Banker's Queues

```
{-# LANGUAGE Strict #-}
```

Idee: Speichere den vorderen und den hinteren Teil der Liste separat, den hinteren in umgekehrter Reihenfolge (damit man effizient anhängen kann). Dabei soll der vordere Teil immer größer als der hintere Teil sein. Falls der hintere Teil die gleiche Länge erreicht, so drehe ihn um und hänge ihn an den vorderen Teil an.

```
data BQueue a = BQueue { front :: [a], frontLen :: Int
                       , rear  :: [a], rearLen  :: Int }
-- mkBQueue :: [a] -> Int -> [a] -> Int

uncons :: BQueue a -> Maybe (a, BQueue a)
uncons (BQueue [] _ _ _) = Nothing
uncons (BQueue (x:xs) fl r rl) =
  Just (x, mkBQueue xs (fl-1) r rl)

snoc :: BQueue a -> a -> BQueue a
snoc (BQueue f fl r rl) x = mkBQueue f fl (x:r) (rl+1)
```

Banker's Queues

```
{-# LANGUAGE Strict #-}
```

Idee: Speichere den vorderen und den hinteren Teil der Liste separat, den hinteren in umgekehrter Reihenfolge (damit man effizient anhängen kann). Dabei soll der vordere Teil immer größer als der hintere Teil sein. Falls der hintere Teil die gleiche Länge erreicht, so drehe ihn um und hänge ihn an den vorderen Teil an.

Laufzeitanalyse: Umdrehen der hinteren Liste und Anhängen an die vordere Liste dauert $\mathcal{O}(n)$ Zeit. Falls kein Umdrehen der Liste erforderlich ist, so benötigen `uncons` und `snoc` konstante Zeit. Bevor das nächste Mal umgedreht werden muss, müssen weitere n Operationen ausgeführt werden. Wenn wir für jede dieser Operationen ein oder zwei Geldstücke in das Guthabenkonto legen, können wir für das nächste Mal Umdrehen bezahlen.

Banker's Queues

```
{-# LANGUAGE Strict #-}
```

Idee: Speichere den vorderen und den hinteren Teil der Liste separat, den hinteren in umgekehrter Reihenfolge (damit man effizient anhängen kann). Dabei soll der vordere Teil immer größer als der hintere Teil sein. Falls der hintere Teil die gleiche Länge erreicht, so drehe ihn um und hänge ihn an den vorderen Teil an.

Laufzeitanalyse: Umdrehen der hinteren Liste und Anhängen an die vordere Liste dauert $\mathcal{O}(n)$ Zeit. Falls kein Umdrehen der Liste erforderlich ist, so benötigen `uncons` und `snoc` konstante Zeit. Bevor das nächste Mal umgedreht werden muss, müssen weitere n Operationen ausgeführt werden. Wenn wir für jede dieser Operationen ein oder zwei Geldstücke in das Guthabenkonto legen, können wir für das nächste Mal Umdrehen bezahlen.

Dieses Argument ist falsch!

Banker's Queues

`{-# LANGUAGE Strict #-}`

Idee: Speichere den vorderen und den hinteren Teil der Liste separat, den hinteren in umgekehrter Reihenfolge (damit man effizient anhängen kann). Dabei soll der vordere Teil immer größer als der hintere Teil sein. Falls der hintere Teil die gleiche Länge erreicht, so drehe ihn um und hänge ihn an den vorderen Teil an.

Laufzeitanalyse: Umdrehen der hinteren Liste und Anhängen an die vordere Liste dauert $\mathcal{O}(n)$ Zeit. Falls kein Umdrehen der Liste erforderlich ist, so benötigen `uncons` und `snoc` konstante Zeit. ~~Bevor das nächste Mal umgedreht werden muss, müssen weitere n Operationen ausgeführt werden.~~ Wenn wir für jede dieser Operationen ein oder zwei Geldstücke in das Guthabenkonto legen, können wir für das nächste Mal Umdrehen bezahlen.

Dieses Argument ist falsch! Dieser Satz stimmt nicht im funktionalen Setting: alte Zustände der Datenstruktur werden aufgehoben, es gibt mehrere Zukünfte!

Lazyness to the rescue!

Interpretieren wir den Code der Banker's Queue in gewöhnlichem, nicht-strikten Haskell, so haben `snoc` und `uncons` tatsächlich amortisiert konstante Laufzeit!
(Grund: `reverse rear` wird nicht sofort berechnet und das Ergebnis wird gespeichert. Beispiel: Amortized Queue)

Real-Time Queues

Wir wollen nun erreichen, dass `snoc` und `uncons` echt konstante Laufzeit besitzen (nicht nur amortisiert).
Dazu müssen wir das Umdrehen der hinteren Liste in Teilschritte konstanter Zeit zerlegen. Wir verbinden das Umdrehen mit dem Durchlaufen der vorderen Liste:

Real-Time Queues

Wir wollen nun erreichen, dass `snoc` und `uncons` echt konstante Laufzeit besitzen (nicht nur amortisiert).
Dazu müssen wir das Umdrehen der hinteren Liste in Teilschritte konstanter Zeit zerlegen. Wir verbinden das Umdrehen mit dem Durchlaufen der vorderen Liste:

```
-- ‘appendAndReverse xs ys zs = xs ++ reverse ys ++ zs‘
-- Vorbedingung: ‘length xs = length ys‘
appendAndReverse :: [a] -> [a] -> [a] -> [a]
appendAndReverse [] [] zs = zs
appendAndReverse (x:xs) (y:ys) zs =
  x : appendAndReverse xs ys (y:zs)
```

Real-Time Queues

Wir wollen nun erreichen, dass `snoc` und `uncons` echt konstante Laufzeit besitzen (nicht nur amortisiert).
Dazu müssen wir das Umdrehen der hinteren Liste in Teilschritte konstanter Zeit zerlegen. Wir verbinden das Umdrehen mit dem Durchlaufen der vorderen Liste:

```
-- ‘appendAndReverse xs ys zs = xs ++ reverse ys ++ zs‘
-- Vorbedingung: ‘length xs = length ys‘
appendAndReverse :: [a] -> [a] -> [a] -> [a]
appendAndReverse [] [] zs = zs
appendAndReverse (x:xs) (y:ys) zs =
  x : appendAndReverse xs ys (y:zs)
```

```
  appendAndReverse [1,2,3] [6,5,4] [7,8,9]
= 1 : appendAndReverse [2,3] [5,4] [6,7,8,9]
= 1 : 2 : appendAndReverse [3] [4] [5,6,7,8,9]
= 1 : 2 : 3 : appendAndReverse [] [] [4,5,6,7,8,9]
= [1,2,3,4,5,6,7,8,9]
```

Real-Time Queues

Wir wollen nun erreichen, dass `snoc` und `uncons` echt konstante Laufzeit besitzen (nicht nur amortisiert).
Dazu müssen wir das Umdrehen der hinteren Liste in Teilschritte konstanter Zeit zerlegen. Wir verbinden das Umdrehen mit dem Durchlaufen der vorderen Liste:

```
-- ‘appendAndReverse xs ys zs = xs ++ reverse ys ++ zs‘
-- Vorbedingung: ‘length xs = length ys‘
appendAndReverse :: [a] -> [a] -> [a] -> [a]
appendAndReverse [] [] zs = zs
appendAndReverse (x:xs) (y:ys) zs =
  x : appendAndReverse xs ys (y:zs)
```

Real-Time Queues

Wir wollen nun erreichen, dass `snoc` und `uncons` echt konstante Laufzeit besitzen (nicht nur amortisiert).
Dazu müssen wir das Umdrehen der hinteren Liste in Teilschritte konstanter Zeit zerlegen. Wir verbinden das Umdrehen mit dem Durchlaufen der vorderen Liste:

```
-- ‘appendAndReverse xs ys zs = xs ++ reverse ys ++ zs‘
-- Vorbedingung: ‘length xs = length ys‘
appendAndReverse :: [a] -> [a] -> [a] -> [a]
appendAndReverse [] [] zs = zs
appendAndReverse (x:xs) (y:ys) zs =
  x : appendAndReverse xs ys (y:zs)
```

Idee: Führe bei jeder `snoc` und `unsnoc`-Operation einen Teilschritt aus.
Wir forcieren Teilschritte, indem wir die Ergebnisliste von `appendAndReverse xs ys zs` durchlaufen.

Real-Time Queues

```
-- `appendAndReverse xs ys zs = xs ++ reverse ys ++ zs`
appendAndReverse :: [a] -> [a] -> [a] -> [a]
```

```
data RTQueue a =
  RTQueue { front :: [a], rear :: [a], schedule :: [a] }
-- Invariant: length front = length rear + length schedule
```

(Damit sind die vordere und hintere Liste genau dann gleich lang, wenn `schedule` leer ist. Wir müssen also die Längen von `front` und `rear` nicht explizit speichern.)

```
empty :: RTQueue a
empty = RTQueue [] [] []

mkRTQueue :: [a] -> [a] -> [a] -> RTQueue a
mkRTQueue f r [] = let f' = appendAndReverse f r []
                   in RTQueue f' [] f'
mkRTQueue f r (_:s') = RTQueue f r s'
```

Real-Time Queues

```haskell
data RTQueue a =
  RTQueue { front :: [a], rear :: [a], schedule :: [a] }
-- Invariant: length front = length rear + length schedule

mkRTQueue :: [a] -> [a] -> [a] -> RTQueue a
mkRTQueue f r [] = let f' = appendAndReverse f r []
                   in RTQueue f' [] f'
mkRTQueue f r (_:s') = RTQueue f r s'
```

```haskell
uncons :: RTQueue a -> Maybe (a, RTQueue a)
uncons (RTQueue [] _ _) = Nothing
uncons (RTQueue (x:f') r s) =
  Just (x, mkRTQueue f' r s)

snoc :: RTQueue a -> a -> RTQueue a
snoc (RTQueue f r s) x = mkRTQueue f (x:r) s
```

```
Prelude> uncons presentation
Nothing :: Maybe (Slide, RTQueue Slide)
```

Interaktive Demo: *timbaumann.info/pfds-visualizations*
Tausendfüßler-Bild: Miran Lipovača, *Learn You a Haskell for Great Good!*