Fortgeschrittene Funktionale Programmierung

2. und 3. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2015/16

Laziness, Kompositionalität und Re-Use

Schön kompositionell:

$$\texttt{any} :: (a \to \mathsf{Bool}) \to [a] \to \mathsf{Bool}$$
$$\texttt{any}\ p = \texttt{or} \circ (\texttt{map}\ p)$$

Mit:

$$\texttt{or} :: [\mathsf{Bool}] \to \mathsf{Bool}$$
$$\texttt{or} = \texttt{foldr}\ (||)\ \mathsf{False}$$

wobei:

$$\texttt{foldr} :: (a \to b \to b) \to b \to [a] \to b$$
$$\texttt{foldr}\ f\ k\ []\qquad = k$$
$$\texttt{foldr}\ f\ k\ (x : xs) = f\ x\ (\texttt{foldr}\ f\ k\ xs)$$

Auswertung:

$$\texttt{any}\ \texttt{even}\ [1\,..\,10\,\hat{}\,6] = \ldots$$

Laziness, Kompositionalität und Re-Use

Ist Laziness hier wirklich essentiell?

Nun ja, mit strikter Auswertung würde

$\texttt{any} :: (a \to \mathsf{Bool}) \to [a] \to \mathsf{Bool}$
$\texttt{any}\ p = \texttt{or} \circ (\texttt{map}\ p)$

in $\texttt{any}\ \texttt{even}\ [1\,..\,10\hat{\ }6]$ zunächst die vollständige Liste

$$\texttt{map}\ \texttt{even}\ [1\,..\,10\hat{\ }6]$$

berechnen. Gegensteuern mit `foldr`-Fusion?

Nicht sehr erfolgreich, denn bei strikter Auswertung würde selbst

$\texttt{any} :: (a \to \mathsf{Bool}) \to [a] \to \mathsf{Bool}$
$\texttt{any}\ p = \texttt{foldr}\ (\lambda x\ r \to p\ x \,||\, r)\ \mathsf{False}$

die rekursiven (`foldr`-) Aufrufe zuerst ausführen.

Laziness, Kompositionalität und Re-Use

Nicht sehr erfolgreich, denn bei strikter Auswertung würde selbst

$$\texttt{any} :: (a \to \mathsf{Bool}) \to [a] \to \mathsf{Bool}$$
$$\texttt{any}\ p = \texttt{foldr}\ (\lambda x\ r \to p\ x \,||\, r)\ \mathsf{False}$$

die rekursiven (`foldr`-) Aufrufe zuerst ausführen.

Wie könnte man dem Problem denn noch begegnen?

Eine mögliche (aber auf ad-hoc Verhalten basierende) Lösung wäre:

$$\begin{aligned} &\texttt{any} :: (a \to \mathsf{Bool}) \to [a] \to \mathsf{Bool} \\ &\texttt{any}\ p\ [\,] \qquad\quad = \mathsf{False} \\ &\texttt{any}\ p\ (x : xs) = p\ x \,||\, \texttt{any}\ p\ xs \end{aligned}$$

Aber was ist dann aus unserer schönen Kompositionalität geworden?

Und wieso sollten wir allgemein auf wiederverwendbare Rekursionskombinatoren verzichten?

Laziness for DSLs

Zur Erinnerung, Parserkombinatoren:

$\textbf{type}$ Parser a

$\texttt{parse} :: \text{Parser } a \to \text{String} \to a$

$\texttt{char} :: \text{Char} \to \text{Parser } ()$

$\texttt{yield} :: a \to \text{Parser } a$

$(|||) :: \text{Parser } a \to \text{Parser } a \to \text{Parser } a$

$(+\!\!+\!\!>) :: \text{Parser } a \to (a \to \text{Parser } b) \to \text{Parser } b$

$(+\!\!+\!\!+) :: \text{Parser } a \to \text{Parser } b \to \text{Parser } b$

$\ldots$

Zum Beispiel für Verwendung wie folgt:

$\texttt{expr} :: \text{Parser } ()$

$\texttt{expr} = (\texttt{term} +\!\!+\!\!+ \texttt{char}\ \texttt{'+'} +\!\!+\!\!+ \texttt{expr}) \mid\mid\mid \texttt{term}$

$\ldots$

Würde ohne Laziness so nicht funktionieren!

Laziness, noch ein nettes Beispiel

$$\textbf{data}\ \mathsf{Tree} = \mathsf{Leaf}\ \mathsf{Int} \mid \mathsf{Node}\ \mathsf{Tree}\ \mathsf{Tree}\ \textbf{deriving}\ \mathsf{Show}$$

$$\begin{aligned}
&\texttt{minleaf} :: \mathsf{Tree} \to \mathsf{Int}\\
&\texttt{minleaf}\ (\mathsf{Leaf}\ n) \quad\ = n\\
&\texttt{minleaf}\ (\mathsf{Node}\ s\ t)\ = \texttt{min}\ (\texttt{minleaf}\ s)\ (\texttt{minleaf}\ t)
\end{aligned}$$

$$\begin{aligned}
&\texttt{replace} :: \mathsf{Tree} \to \mathsf{Int} \to \mathsf{Tree}\\
&\texttt{replace}\ (\mathsf{Leaf}\ n) \quad\ m = \mathsf{Leaf}\ m\\
&\texttt{replace}\ (\mathsf{Node}\ s\ t)\ m = \mathsf{Node}\ (\texttt{replace}\ s\ m)\ (\texttt{replace}\ t\ m)
\end{aligned}$$

$$\texttt{run}\ t = \texttt{replace}\ t\ (\texttt{minleaf}\ t)$$

$$\Downarrow$$

$$\begin{aligned}
&\texttt{repmin} :: \mathsf{Tree} \to \mathsf{Int} \to (\mathsf{Tree}, \mathsf{Int})\\
&\texttt{repmin}\ (\mathsf{Leaf}\ n) \quad\ m = (\mathsf{Leaf}\ m, n)\\
&\texttt{repmin}\ (\mathsf{Node}\ s\ t)\ m = (\mathsf{Node}\ s'\ t', \texttt{min}\ m_1\ m_2)\\
&\qquad\qquad\qquad\qquad\quad \textbf{where}\ (s', m_1) = \texttt{repmin}\ s\ m\\
&\qquad\qquad\qquad\qquad\qquad\qquad (t', m_2) = \texttt{repmin}\ t\ m
\end{aligned}$$

$$\texttt{run}\ t = \textbf{let}\ (t', m) = \texttt{repmin}\ t\ m\ \textbf{in}\ t'$$

Laziness, zirkuläre Programme

$$
\begin{array}{l}
\texttt{repmin} :: \mathrm{Tree} \to \mathrm{Int} \to (\mathrm{Tree}, \mathrm{Int}) \\
\texttt{repmin}\ (\mathrm{Leaf}\ n) \quad\ \ m = (\mathrm{Leaf}\ m, n) \\
\texttt{repmin}\ (\mathrm{Node}\ s\ t)\ m = (\mathrm{Node}\ s'\ t', \texttt{min}\ m_1\ m_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{where}\ (s', m_1) = \texttt{repmin}\ s\ m \\
\qquad\qquad\qquad\qquad\qquad\qquad\ (t', m_2) = \texttt{repmin}\ t\ m
\end{array}
$$

$$\texttt{run}\ t = \textbf{let}\ (t', m) = \texttt{repmin}\ t\ m\ \textbf{in}\ t'$$

Zum Nachdenken:

1. Was wäre denn eine geeignete Darstellung, um die Auswertungsreihenfolge bzw. „Funktionsweise“ von `repmin` deutlich zu machen?
2. Wie sieht für

   $$\textbf{let}\ s = \texttt{foldr}\ (+)\ 0\ xs\ \textbf{in}\ \texttt{foldr}\ (\lambda\_\ ys \to s : ys)\ []\ xs$$

   ein äquivalentes zirkuläres Programm aus, welches den doppelten Durchlauf durch xs vermeidet?

Aber: Laziness kann auch wehtun

Bekannter Verwandter von `foldr`:

$$\texttt{foldl} :: (b \to a \to b) \to b \to [a] \to b$$
$$\texttt{foldl}\ f\ k\ [\,] \quad = k$$
$$\texttt{foldl}\ f\ k\ (x : xs) = \texttt{foldl}\ f\ (f\ k\ x)\ xs$$

Auswertung:

$$\texttt{foldl}\ (+)\ 0\ [1 \mathinner{..} 10\hat{}8] = \ldots$$

Problem: riesige Zwischenausdrücke!

„Lösung“:

$$\texttt{foldl}'\ :: (b \to a \to b) \to b \to [a] \to b$$
$$\texttt{foldl}'\ f\ k\ [\,] \quad = k$$
$$\texttt{foldl}'\ f\ k\ (x : xs) = \textbf{let}\ y = f\ k\ x\ \textbf{in}\ \texttt{seq}\ y\ (\texttt{foldl}'\ f\ y\ xs)$$

Nochmal zum Nachdenken:

1. Sei `sort`:
   a) eine Haskell-Implementierung von Merge Sort
   b) eine Haskell-Implementierung von Insertion Sort

   Machen Sie jeweils Aussagen zur Effizienz (Laufzeit und Speicher) von $\texttt{smallest} = \texttt{head} \circ \texttt{sort}$.

2. Gegeben die Definition von `fiblist` mittels list comprehension und `zip`, analysieren Sie den Speicherverbrauch während der Berechnung von Ausdrücken $\texttt{drop}\ n\ (\texttt{take}\ n\ \texttt{fiblist})$.

3. Wie sieht es aus bei folgender Variante?

   $$\begin{aligned} &\texttt{fiblist} = \texttt{go}\ 1\ 1 \\ &\qquad\qquad \textbf{where}\ \texttt{go}\ a\ b = a : \texttt{go}\ b\ (a + b) \end{aligned}$$

4. Sind bei 2. und 3. Verbesserungen möglich?

Haskell: Typklassen

Eine Typklasse als Interface

$$
\begin{array}{l}
\textbf{class}\ \text{Queue}\ q\ \textbf{where} \\
\quad \texttt{empty} \quad :: q\ a \\
\quad \texttt{isEmpty} :: q\ a \to \text{Bool} \\
\quad \texttt{head} \quad :: q\ a \to a \\
\quad \texttt{tail} \quad :: q\ a \to q\ a \\
\quad \texttt{snoc} \quad :: q\ a \to a \to q\ a
\end{array}
$$

Gesetze:

- $\texttt{isEmpty}\ \texttt{empty} = \text{True}$
- $\texttt{isEmpty}\ (\texttt{snoc}\ q\ x) = \text{False}$
- $\texttt{head}\ (\texttt{snoc}\ \texttt{empty}\ x) = x$
- $\texttt{isEmpty}\ q = \text{False} \Rightarrow \texttt{head}\ (\texttt{snoc}\ q\ x) = \texttt{head}\ q$
- $\texttt{tail}\ (\texttt{snoc}\ \texttt{empty}\ x) = \texttt{empty}$
- $\texttt{isEmpty}\ q = \text{False} \Rightarrow \texttt{tail}\ (\texttt{snoc}\ q\ x) = \texttt{snoc}\ (\texttt{tail}\ q)\ x$

Beispiel-Implementierung und Verwendung

```
instance Queue [] where
  empty     = []
  isEmpty   = List.null
  head      = List.head
  tail      = List.tail
  snoc xs x = xs ++ [x]

fun :: Queue q => Int -> q Int
fun n = let q1 = snoc empty 1
            q2 = snoc q1 n
            q3 = snoc q2 3
        in tail q3

main = do print (List.length (fun 2 :: [Int]))
          n <- readLn
          print (head (fun n :: [Int]))
```

Abhängigkeiten zwischen Typklassen

```
class Queue q ⇒ Deque q where
  cons :: a → q a → q a
  last :: q a → a
  init :: q a → q a

fun :: Deque q ⇒ Int → q Int
fun n = let q1 = snoc empty 1
            q2 = snoc q1 n
            q3 = snoc q2 3
        in init q3

instance Deque [] where
  cons = (:)
  last = List.last
  init = List.init
```

Etwas Algorithmik

Batched Queues

Die angegebene Implementierung **instance** Queue [] ist ineffizient, wegen des linearen Aufwands in `snoc` $xs\ x = xs ++ [x]$.

Eine populäre Alternative:

data BQ a = BQ [a] [a]

instance Eq $a \Rightarrow$ Eq (BQ a) **where**
  $q_1 == q_2 =$ `toList` $q_1 ==$ `toList` q_2
    where `toList` (BQ f r) =
      f ++ List.`reverse` r

Erzwinge Invariante dass nie nur f leer:

`bq` [] r = BQ (List.`reverse` r) []
`bq` f r = BQ f r

Batched Queues

```
data BQ a = BQ [a] [a]
```

```
instance Eq a => Eq (BQ a) where
  q1 == q2 = toList q1 == toList q2
    where toList (BQ f r) =
            f ++ List.reverse r
```

Erzwinge Invariante dass nie nur f leer:

```
bq [] r = BQ (List.reverse r) []
bq f  r = BQ f r
```

```
instance Queue BQ where
  empty              = BQ [] []
  isEmpty (BQ f _)   = List.null f
  head    (BQ f _)   = List.head f
  tail    (BQ f r)   = bq (List.tail f) r
  snoc    (BQ f r) x = bq f (x : r)
```

Übung: Testen Sie mit QuickCheck, ob diese Implementierung die Queue-Gesetze erfüllt.

Effizienz: Nun ist snoc $\mathcal{O}(1)$, aber tail nicht mehr. Jedoch, ...

Amortisierende Laufzeitanalyse

Man betrachte, statt Kosten nur einzelner Operationen, die Kosten über die Lebenszeit einer Datenstruktur hinweg.

(siehe Tafel)

Formalisierung:

- per Operation, Zuordnung von amortisierten Kosten via:
  $a_i = t_i + c_i - \bar{c}_i$, wobei t_i: tatsächliche Kosten
  c_i: Ansparung von Credits
  $\bar{c}_i$: Verausgabung von Credits
- Nur zuvor angesparte Credits dürfen ausgegeben werden.
- Dann gilt zu jedem Zeitpunkt (m):

$$\sum_{i=1}^{m} t_i \leqslant \sum_{i=1}^{m} a_i$$

Amortisierende Laufzeitanalyse

Konkret für die BQ-Implementierung:

- `empty`, `isEmpty`, `head`: jeweils $t_i = 1$, $c_i = \bar{c}_i = 0$, also $a_i = 1$
- `tail` (auf BQ f r):
  - falls $|f| = 1$: $t_i = 1 + |r|$, $c_i = 0$, $\bar{c}_i = |r|$, also $a_i = 1$
  - falls $|f| > 1$: $t_i = 1$, $c_i = \bar{c}_i = 0$, also $a_i = 1$
- `snoc` (auf BQ f r):
  - falls $|f| = 0$: $t_i = 1$, $c_i = \bar{c}_i = 0$, also $a_i = 1$
  - falls $|f| > 0$: $t_i = 1$, $c_i = 1$, $\bar{c}_i = 0$, also $a_i = 2$
- Bedingung dass nur zuvor angesparte Credits ausgegeben werden dürfen, wird erfüllt durch Credits-Invariante dass zu Datenstruktur BQ f r stets genau $|r|$ Credits angespart.

Also sind alle Operationen amortisiert $\mathcal{O}(1)$.

Erweiterung zu Deque (double-ended queue)?

Nahe liegender Versuch:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Deque}\ \mathsf{BQ}\ \textbf{where}\\
\quad \texttt{cons}\ x\ (\mathsf{BQ}\ f\ r) = \texttt{bq}\ (x : f)\ r\\
\quad \texttt{last}\ \ (\mathsf{BQ}\ f\ [\,]) = \mathsf{List}.\texttt{last}\ f\\
\quad \texttt{last}\ \ (\mathsf{BQ}\ f\ r) = \mathsf{List}.\texttt{head}\ r\\
\quad \texttt{init}\ \ (\mathsf{BQ}\ f\ [\,]) = \texttt{bq}\ (\mathsf{List}.\texttt{init}\ f)\ [\,]\\
\quad \texttt{init}\ \ (\mathsf{BQ}\ f\ r) = \texttt{bq}\ f\ (\mathsf{List}.\texttt{tail}\ r)
\end{array}
$$

Aber dann ist zum Beispiel `last` nicht $\mathcal{O}(1)$, nicht mal amortisiert!

Reparatur durch symmetrische Invariante und „Balancing“:

$$
\begin{array}{l}
\texttt{bq}\ [x]\ [\,] = \mathsf{BQ}\ [x]\ [\,]\\
\texttt{bq}\ [\,]\ [y] = \mathsf{BQ}\ [y]\ [\,]\\
\texttt{bq}\ [\,]\ r = \textbf{let}\ (xs, ys) = \texttt{splitHalf}\ r\ \textbf{in}\ \mathsf{BQ}\ (\mathsf{List}.\texttt{reverse}\ ys)\ xs\\
\texttt{bq}\ f\ [\,] = \textbf{let}\ (xs, ys) = \texttt{splitHalf}\ f\ \textbf{in}\ \mathsf{BQ}\ xs\ (\mathsf{List}.\texttt{reverse}\ ys)\\
\texttt{bq}\ f\ r = \mathsf{BQ}\ f\ r
\end{array}
$$

Amortisierende Laufzeitanalyse jetzt

Credits-Invariante diesmal: zu Datenstruktur BQ f r mindestens so viele Credits angespart wie sich $|f|$ und $|r|$ unterscheiden.

- `empty`, `isEmpty`, `head`, `last`: jeweils $t_i = 1$, $c_i = \bar{c}_i = 0$, also $a_i = 1$
- `tail` (auf BQ f r):
  - falls $|f| = 1$, $|r| > 1$: $t_i = 1 + |r|$, $c_i = 1$, $\bar{c}_i = |r| - 1$, also $a_i = 3$
  - ansonsten: $t_i = 1$, $c_i = 1$, $\bar{c}_i = 0$, also $a_i = 2$
- `init` (auf BQ f r):
  - falls $|r| = 1$, $|f| > 1$: $t_i = 1 + |f|$, $c_i = 1$, $\bar{c}_i = |f| - 1$, also $a_i = 3$
  - ansonsten: $t_i = 1$, $c_i = 1$, $\bar{c}_i = 0$, also $a_i = 2$
- `snoc`, `cons`: jeweils $t_i = 1$, $c_i = 1$, $\bar{c}_i = 0$, also $a_i = 2$

Also sind wieder alle Operationen amortisiert $\mathcal{O}(1)$.

Ein Problem

Erinnern wir uns an die einfachere Variante (ohne double-endedness):

data BQ a = BQ $[a]$ $[a]$

$\texttt{bq}$ [ ] r = BQ (List.$\texttt{reverse}$ r) [ ]
$\texttt{bq}$ f r = BQ f r

instance Queue BQ **where**

```
empty                = BQ [] []
isEmpty (BQ f _)     = List.null f
head    (BQ f _)     = List.head f
tail    (BQ f r)     = bq (List.tail f) r
snoc    (BQ f r) x   = bq f (x : r)
```

Dafür haben wir amortisiert $\mathcal{O}(1)$ für alle Operationen bewiesen.

Aber das gilt nicht unter Berücksichtigung von Persistenz.

Persistenz – Beispiel

$$
\begin{aligned}
\texttt{main} = \textbf{let}\ & q_0 = \texttt{empty} :: \mathsf{BQ\ Int} \\
& q_1 = \texttt{snoc}\ q_0\ 1 \\
& q_2 = \texttt{snoc}\ q_1\ 2 \\
& q_3 = \texttt{snoc}\ q_2\ 3 \\
& q_4 = \texttt{snoc}\ q_3\ 4 \\
\textbf{in}\ & \texttt{print}\ (\texttt{head}\ (\texttt{tail}\ q_3) + \texttt{head}\ (\texttt{tail}\ q_4))
\end{aligned}
$$

… Diskussion

A long walk in the snow

Chris Okasaki (`http://goo.gl/8EeO9s`):

The day before the meeting, I realized that my approach was completely broken. Afraid of looking like an idiot, I went into a frenzy, playing with different variations, and a few hours later I came up with an approach again based on lazy evaluation. I was sure that this approach worked, but I had no idea how to prove it. (Again, this was a case where the implementation was simple, but the analysis was a bear.) My wife had our car somewhere else that day, so I ended up walking home. The walk usually took about 45 minutes, but it was snowing pretty hard, so it took about twice that long. The whole way home I thought about nothing but how to analyze my data structure. I knew it had something to do with amortization, but the usual techniques of amortization weren't working. About halfway home, I came up with the idea of using debits instead of credits, and by the time I got home the entire framework of how to analyze data structures involving lazy evaluation had crystallized in my head.

Debits statt Credits

- Mit Credits wird für zukünftigen Aufwand angespart.
- Dessen Höhe ist nicht im Voraus beschränkt.
- Durch Persistenz können unabhängige Zukünfte für eine konkrete Ausprägung (Wert) der Datenstruktur entstehen – mit jeweils unabhängig anfallendem zukünftigen Aufwand.
- Einmal angesparte Credits einfach in beiden „Zukunftszweigen“ auszugeben, wäre jedoch illegitim.

<u>stattdessen:</u>

- Mit Laziness werden beschränkte Aufwandspakete (Thunks) in die Zukunft verzögert.
- Proportional zu ihrem Aufwand werden Schulden aufgenommen.
- Erst wenn diese Schulden/Debits für einen Thunk abgebaut sind, darf er ausgewertet werden.
- Ein Schuldenpaket in mehrere Zukünfte mitzugeben, ist zwar pessimistisch, aber kein „Schönrechnen“ zu eigenen Gunsten.
- Laziness garantiert, dass ein gegebenes Aufwandspaket über alle Zukünfte hinweg höchstens einmal realisiert wird.

(Lazy, Amortized) Banker's Queue

Zusätzliche Speicherung der Listenlängen:

data BQ a = BQ Int [a] Int [a]

Erzwinge Invariante dass f nie kürzer als r:

$$
\begin{aligned}
\texttt{bq}\ n\ f\ m\ r \mid n == m - 1 &= \mathsf{BQ}\ (n + m)\ (f +\!\!+ \mathsf{List.}\texttt{reverse}\ r)\ 0\ [\,] \\
\mid n \geqslant m &= \mathsf{BQ}\ n\ f\ m\ r
\end{aligned}
$$

Implementierung der Operationen frei von Überraschungen:

instance Queue BQ **where**

$$
\begin{aligned}
&\texttt{empty} && = \mathsf{BQ}\ 0\ [\,]\ 0\ [\,] \\
&\texttt{isEmpty}\ (\mathsf{BQ}\ n\ \_\ \_\ \_) && = n == 0 \\
&\texttt{head}\ (\mathsf{BQ}\ \_\ f\ \_\ \_) && = \mathsf{List.}\texttt{head}\ f \\
&\texttt{tail}\ (\mathsf{BQ}\ n\ f\ m\ r) \mid n > 0 && = \texttt{bq}\ (n - 1)\ (\mathsf{List.}\texttt{tail}\ f)\ m\ r \\
&\texttt{snoc}\ (\mathsf{BQ}\ n\ f\ m\ r)\ x && = \texttt{bq}\ n\ f\ (m + 1)\ (x : r)
\end{aligned}
$$

(Lazy, Amortized) Banker's Queue – Analyse

Formaler Ansatz:

- per Operation, Zuordnung von amortisierten Kosten via:
  $a_i = u_i + d_i$, wobei u_i: ungeteilte Kosten (eingehende Thunks als vollständig ausgewertet vorliegend angenommen, selbst angelegte Thunks ignoriert)
  d_i: Abzahlung von Debits
- Außerdem kann eine Operation noch geteilte Kosten s_i haben (potentielle zukünftige Kosten selbst angelegter, nicht direkt ausgewerteter Thunks), diese werden als Debits in der Datenstruktur akkumuliert.
- Dabei ist jetzt die Lokalität von Debits innerhalb der Datenstruktur relevant, denn erst wenn die zu einem bestimmten Thunk gehörenden Debits abgezahlt sind, darf genau dieser ausgewertet werden. (Inkrementelle Funktionen erlauben die Verteilung von Debits auf mehrere Positionen.)

(Lazy, Amortized) Banker's Queue – Analyse

Ignorieren wir zunächst die Kosten des $+\!\!+$. Wie sollten wir Debits für `reverse`-Thunks verteilen?

Beispiel:

- aktuelle Struktur sei:

$$f = \boxed{0 \mid 1 \mid 2}\,,\ \ r = \boxed{5 \mid 4 \mid 3}$$

- nach `snoc`:

$$f' = f +\!\!+ \text{List.}\texttt{reverse}\,(x : r)\,,\ \ r' = [\,]$$

- also, konzeptionell nach Rotation:

$$f' = \boxed{0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid x}\,,\ \ r' = |$$

debits: $\sim|r|$

(Lazy, Amortized) Banker's Queue – Analyse

- also, konzeptionell nach Rotation:

$$f' = \boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{x}\,,\;\; r' = |$$

debits: $\sim|r|$

- Für die Abzahlung in Frage kommen `tail` und `snoc`.

- Alle Debits müssen abbezahlt sein, wenn die betroffene Position (aktuell in der Mitte von f') erreicht wird.

- Darüber hinaus ist es der Analyse zuträglich, wenn die Debits auch abbezahlt sind bevor die nächste Rotation ansteht.

- Mögliche Wahl: Anlegen von $2 * |r|$ Debits, jedes `tail` zahlt 2 Debits ab, jedes `snoc` zahlt 1 Debit ab.

(Lazy, Amortized) Banker's Queue – Analyse

Berücksichtigen wir nun auch die Kosten des $+\!\!+$.

Da $+\!\!+$ inkrementell arbeitet, müssen wir seine Debits nicht auf die erste Listenposition legen, sondern können sie verteilen.

Etwa für $n = 4$, direkt nach Rotation:

$$\begin{array}{l} f' = \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline \end{array},\ r' = \Big| \\ \text{debits:} \quad\ \ \begin{array}{ccccccccc} & 1 & 1 & 1 & 5 & & & & \end{array} \end{array}$$

Debits-Invariante (zu jeweils aktuellem f, r):

- Debits nur in f, dort bis einschließlich Position i höchstens $\min(2 * i, |f| - |r|)$
- … garantiert, dass nie Debits auf Position 0.
- … garantiert, dass vor Rotation immer alle Debits abbezahlt.
- … wird erhalten durch Abzahlung von 2 Debits durch `tail`, 1 Debit durch `snoc`. (Diskussion einzelner Fälle …)

Ergebnisse

- Eine Queue-Implementierung mit amortisiert $\mathcal{O}(1)$ für alle Operationen selbst unter Berücksichtigung von Persistenz
- Ohne Berücksichtigung von Persistenz bereits vorher Queue- und Deque-Implementierungen mit jeweils amortisiert $\mathcal{O}(1)$
- Okasaki zeigt außerdem, jeweils wieder mit Persistenz:
  - Hinzunahme von `cons`, `last`, `init` mit amortisiert $\mathcal{O}(1)$
  - Hinzunahme von auch noch $+\!\!+$ mit amortisiert $\mathcal{O}(1)$
  - Unter Ausklammerung von $+\!\!+$, Eliminierung der Abhängigkeit von Amortisierung, also dann sogar *worst-case* $\mathcal{O}(1)$ für alle Queue- und Deque-Operationen
  - Ähnliches für weitere Datenstrukturen

Möglich zur Übung

Aus Okasaki-Buch:

- Variieren Sie die Implementierung der (lazy, amortized, persistent) Banker's Queue durch Änderung der Invariante von $|f| \geqslant |r|$ zu $2 * |f| \geqslant |r|$.
- Vergleichen Sie die Effizienz der beiden Implementierungen, zum Beispiel auf einer Sequenz von 100×`snoc` gefolgt von 100×`tail`.
- Zusatz: Beweisen Sie, dass die geänderte Implementierung immer noch amortisiert $\mathcal{O}(1)$ für alle Operationen ist.