FP/Haskell: Abstrakte Datentypen

Definition

- Typen normalerweise explizit/konkret über die Werte definiert
  - data Foobar = Foo | Bar
- Abstrakte Datentypen sind über die **Operationen** definiert (Interface)
  - „Queue kann front() und join() …“
- Operationen sind (idealerweise) über Axiome definiert
  - back (join 1 empty) == empty
- Ziel: Implementierungs- / Repräsentationsunabhängigkeit
  (verschiedene Optimierungen, Effizienzfokus auf unterschiedlichen Funktionen)

Schnittstelle

- Export der öffentlichen Schnittstelle über *module*

- *newtype*-Deklaration statt *type*-Synonym
  - „type“ übernimmt Klasseninstanzen (Eq, Ord, etc)
    (simple Substitution des Typnamens)
  - „newtype“ übernimmt keine Klasseninstanzen (aber erlaubt eigene Instanzen)

- ```
  module Queue (Queue, empty, isEmpty, join, front, back) where
        newtype Queue a = MkQ([a], [a])

        empty :: Queue a
        empty = MkQ([], [])

        isEmpty …
        join …
        front …
        back …

        internalHelper …
  ```
  ```

Queue

- Queue – Warteschlange – First In / First Out
- Operationen
  - *empty* Konstruktor
  - isEmpty Bool
  - *join* anhängen
  - *front* erstes Element
  - *back* Restelemente (Queue ohne front)
- Anforderungen
  - *empty* O(1)
  - isEmpty O(1)
  - *join* O(1)
  - *front* O(1)
  - *back* O(?)

Queue – Axiome

- `isEmpty empty = True`
- `isEmpty (join x xq) = False`

- `front (join x empty) = x`
- `front (join x (join y xq)) = front (join y xq)`

- `back (join x empty) = empty`
- `back (join x (join y xq)) = join x (back (join y xq))`

Queue – 1.) Liste

- `Type Queue a = [a]`

  vs.

- `newtype Queue a = MkQ [a]`

Queue – 2.) Tupel von 2 Listen

- `newtype Queue a = MkQ ([a],[a])`

Set – Axiome 1

- `insert x (insert x xs) = insert x xs`
- `insert x (insert y xs) = insert y (insert x xs)`

- `isEmpty empty = True`
- `isEmtpy (insert x xs) = False`

- `member empty y = False`
- `member (insert x xs) y = (x == y) || member xs y`

- `delete x empty = empty`
- `delete x (insert y xs) =`

```
    if x == y
    then delete x xs
    else insert y (delete x xs)
```

Set – Axiome 2

- `union xs empty = xs`
- `union xs (insert y ys) = insert y (union xs ys)`

- `meet xs empty = emtpy`
- ```
  meet xs (insert y ys)  =
       if member xs y
       then insert y (meet xs ys)
       else meet xs ys
  ```

- `minus xs empty = xs`
- `minus xs (insert y ys) = minus(delete y xs) ys`
  ```

Bag – Axiome 1

- `isEmpty (mkBag xs) = null xs`
- `union (mkBag xs) (mkBag ys)= mkBag(xs ++ ys)`
- `minBag (mkBag xs) = minlist xs`
- `delMin (mkBag xs) = mkBag (deleteMin xs)`

Bag – Axiome 2

- `insert x (insert y xb) = insert y (insert x xb)`

- `mkBag = foldr insert empty`

- `union xb empty = xb`
- `union xb (insert y yb) = insert y (union xb yb)`

- `minBag (insert x empty) = x`
- `minBag (insert x (insert y xb)) = x `min` minBag (insert y xb)`

- `delMin (insert x empty) = empty`
- `insert (minBag xb) (delMin xb) = xb`

FlexArray

- Flexibler Array, endliche Liste
- Indizierter Zugriff (0 – n-1) access, update
- Hinzufügen am Anfang (0) : loext
  am Ende (n-1): hiext
- Entfernen am Anfang (0): lorem
  am Ende (n-1): hirem

FlexArray – Axiome

- `hiext x . loext x = loext y . hiext x`

- `hirem empty = error`

- `hirem (hiext x xf)    = xf`
- `hirem (loext x empty) = empty`

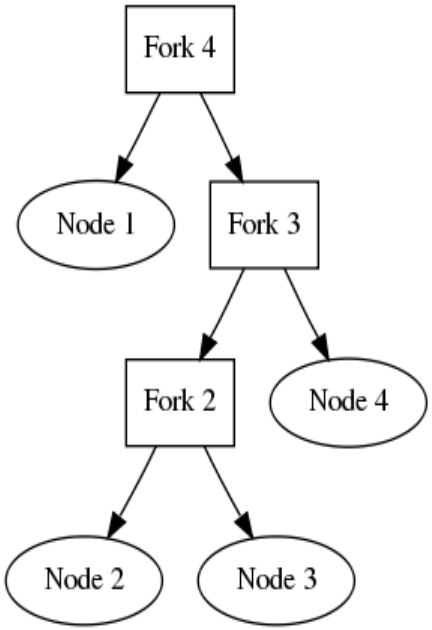
- `hirem (loext x (hiext y xf)) = loext x xf`
- `hirem (loext x (loext y xf)) = loext x (hirem (loext y xf))`

FlexArray – Code …

FlexArray – hiext vs. loext

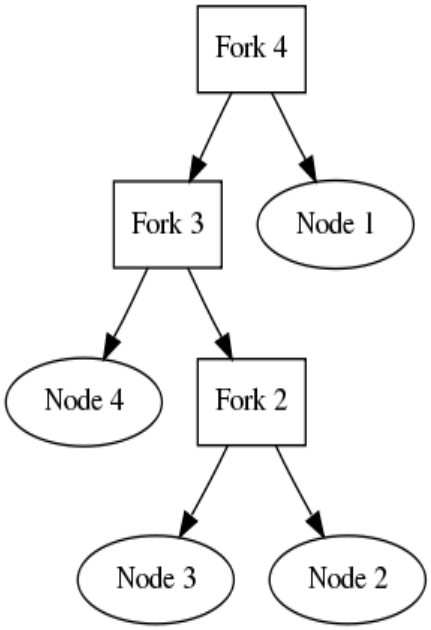
hiext 4 . hiext 3 . hiext 2 . hiext 1

Right subtrees: leftist + left-perfect



loext 4 . loext 3 . loext 2 . loext 1

Left Subtrees: rightist + right-perfect



Flexibler Array – hiet