Haskell – Seminar
Abstrakte Datentypen

Nils Bardenhagen
ms2725

Gliederung

- Konzept
- Queue
- Module
- Sets
- Bags
- Flexible Arrays
- Fazit

Abstrakte Datentypen (ADT)

Definition:

„Eine Zusammenfassung von Operationen, die auf einer Menge von Objekten durchgeführt werden, wird als *abstrakter Datentyp* bezeichnet.“

Alternative Bezeichnung: *Klasse*, *Modul*

ADT: Eigenschaften

- **Universalität**: Verwendung in verschiedenen Programmen
- **Präzise Beschreibung**: Interface muss eindeutig und vollständig sein
- **Kapselung**: Der Anwender soll wissen was der ADT tut, aber nicht wie
- **Schutz**: Der Anwender kann nicht in die interne Datenstruktur eingreifen.
- **Modularität**: Einfacher Austausch, Fehlersuche, Verbesserung

=> Objektorientierung

ADT: Beispiele

- Float
- Tree
- List
- Queue
- Stack

Queue

- Operationen:

empty	:: Queue α
join	:: $\alpha \longrightarrow$ Queue $\alpha \longrightarrow$ Queue α
front	:: Queue $\alpha \longrightarrow \alpha$
back	:: Queue $\alpha \longrightarrow$ Queue α
isEmpty	:: Queue $\alpha \longrightarrow$ Bool

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))

isEmpty (join x bottom) = isEmpty $\perp$ = $\perp$

Queue: Implementierung 1

```
joinc              :: α ⟶ [α] ⟶ [α]
joinc x xs         = xs ++ [x]

emptyc             :: [α]
emptyc             = []

isEmptyc           :: [α] ⟶ Bool
isEmptyc xs        = null xs

frontc             :: [α] ⟶ α
frontc (x:xs)      = x

backc              :: [α] ⟶ [α]
backc (x:xs)       = xs


abstr              :: [α] ⟶ Queue α
abstr              = foldr join empty . Reverse
reprn              :: Queue α ⟶ [α]
reprn empty        = []
reprn (join x xq)  = reprn xq ++ [x]
```

Queue: Implementierung 2

```
valid            :: ([α],[α]) ⟶ Bool
valid (xs,ys)    = not (null xs) v null ys

abstr            :: ([α],[α]) ⟶ Queue α
abstr   (xs,ys)  = (foldr join empty . reverse) (xs ++ reverse ys)
```

Queue: Implementierung 2

emptyc	= ([],[])
isEmptyc (xs,ys)	= null xs
joinc x (xs,ys)	= mkValid (ys, x:zs)
frontc (x:xs, ys)	= x
backc (x:xs,ys)	= mkValid (xs,ys)

mkValid	:: $([\alpha],[\alpha]) \longrightarrow ([\alpha],[\alpha])$
mkValid (xs, ys)	= **if** null xs **then** (reverse ys,[]) **else** (xs,ys)

Module

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

isEmpty	:: Queue $\alpha \longrightarrow$ Bool
isEmpty (MkQ (xs:ys))	= null xs
empty	:: Queue α
empty	= MkQ([],[])
join	:: $\alpha \longrightarrow$ Queue $\alpha \longrightarrow$ Queue α
join x (MkQ (ys,xs))	= mkValid(ys,x:xs)
front	:: Queue $\alpha \longrightarrow \alpha$
front (MkQ (x:xs, ys))	= x
back	:: [α] $\longrightarrow$ [α]
back (MkQ(x:xs, ys))	= mkValid(xs,ys)
mkValid	:: ([α],[α]) $\longrightarrow$ Queue α
mkValid (xs, ys)	= **if** null xs **then** MkQ (reverse ys, []) **else** mkQ (xs, ys)

Module (2)

import Queue

```
toQ        :: [α] ⟶ Queue α
toQ        = foldr join empty . Reverse

fromQ      :: Queue α ⟶ [α]
fromQ q  = if isEmpty q then [] front q:fromQ (back q)
```

```
? join 1 (join 2 empty)
  ([2],[1])

? join 1 (join 2 empty) == join 2 (join 1 empty)
  False
```

Sets

- Ausgewählte Operationen:

empty	:: Set α
isEmpty	:: Set $\alpha \longrightarrow$ Bool
member	:: Set $\alpha \longrightarrow \alpha \longrightarrow$ Bool
insert	:: $\alpha \longrightarrow$ Set $\alpha \longrightarrow$ Set α
delete	:: $\alpha \longrightarrow$ Set $\alpha \longrightarrow$ Set α
union	:: Set $\alpha \longrightarrow$ Set $\alpha \longrightarrow$ Set α
meet	:: Set $\alpha \longrightarrow$ Set $\alpha \longrightarrow$ Set α
minus	:: Set $\alpha \longrightarrow$ Set $\alpha \longrightarrow$ Set α

Sets: Axiome

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

isEmpty empty               = True
isEmpty (insert x xs)       = False

member empty y              = False
member (insert x xs) y      = (x=y) v 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)

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

meet xs empty               = empty
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
```

Sets: Implementierung als Liste

abstr	:: $[\alpha] \longrightarrow$ Set a
abstr	= foldr insert empty
valid xs	= True
valid xs	= nonduplicated xs
member xs x	= some (==x)
insert x xs	= x:xs
delete x xs	= filter (≠x) xs
union xs ys	= xs ++ ys
minus xs ys	= filter (not . member ys) xs
some	:: $(\alpha \longrightarrow$ Bool) $\longrightarrow [\alpha] \longrightarrow$ Bool
some p	= or . map p

Sets: Implementierung als Liste

```
insert x xs        = x:filter (≠ x) xs
union xs ys        = xs ++ filter (not . Member xs) ys


member xs x        = if null ys then False else (x == head ys)
                     where ys = dropWhile (<x) xs


union [] ys        = ys
union (x:xs) []    = x:xs
union (x:xs)(y:ys)
     | (x < y)                = x:union xs (y:ys)
     | (x==y)                 = x:union xs ys
     | (x > y)                = y:union (x:xs) ys
```

Sets: Implementierung als Baum

Data Stree a = Null | Fork (Stree α) α (Stree α)

```
empty                    :: Set α
empty                    = Null

isEmpty                  :: Set α ⟶ Bool
isEmpty Null             = True
isEmpty (Fork xt y yt)   = False

member                   :: (Ord α) => Stree α ⟶ α ⟶ Bool
member Null x            = False
member (Fork xt y yt) x
      | (x < y)          = member xt x
      | (x == y)         = True
      | (x > y)          = member zt x

insert                   :: (Ord α) => α ⟶ Stree α ⟶ Stree α
insert x Null            = Fork Null x Null
insert x (Fork xt y zt)
      | (x < y)          = Fork (insert x xt) y zt
      | (x == y)         = Fork xt y zt
      | (x > y)          = Fork xt y (insert x zt)
```

Sets: Implementierung als Baum

```
delete                        ::(Ord α) => α ⟶ Stree α ⟶ Stree α
delete x Null                 = Null
delete x (Fork xt y zt)
    | (x < y)          = Fork (delete x xt) y zt
    | (x == y)         = join xt zt
    | (x > y)          = Fork xt y (delete x zt)

join                          :: Stree α ⟶ Stree α ⟶ Stree α
join xt yt                    = if isEmpty yt then xt else Fork xt y zt
                                where (y,zt) = splitTree xt

splitTree                     :: Stree α ⟶ (α, Stree α)
splitTree (Fork xt y zt)      = if isEmpty xt then (y,zt) else (u, Fork vt y zt)
                                where (u,vt) = splitTree xt
```

Bags / Multisets

- {[1,2,2,3]} ={[3,2,1,2]} aber {[1,2,2]} != {[1,2]}

- Operationen

  mkBag :: $[\alpha] \longrightarrow$ Bag α
  isEmpty :: Bag $\alpha \longrightarrow$ Bool
  union :: Bag $\alpha \longrightarrow$ Bag $\alpha \longrightarrow$ Bag α
  minBag :: Bag $\alpha \longrightarrow \alpha$
  delMin :: Bag $\alpha \longrightarrow$ Bag α

Bags: Axiome

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

Bags: Implementierung (Heap)

```
data Htree α = Null | Fork Int α (Htree α) (Htree α)

fork                     :: α ⟶ Htree α ⟶ Htree α
fork x yt zt             = if m < n then Fork p x zt yt else Fork p x yt zt
                             where m = size yt
                                   n = size zt
                                   p = m + n + 1

size                     :: Htree α ⟶ Int
size Null                = 0
size (Fork n x yt zt)    = n

isEmpty                  :: Htree α ⟶ Bool
isEmpty Null             = True
isEmpty (Fork n x yt zt) = False

minBag                   :: Htree α ⟶ α
minBag (Fork n x yt zt)  = x

delMin                   :: Htree α ⟶ Htree α
delMin (Fork n x yt zt)  = union yt zt
```

Bags: Implementierung (Heap)

```
union                              :: Htree α ⟶ Htree α ⟶ Htree α
union Null yt                      = yt
union (Fork m u vt wt) Null        = Fork m u vt wt

union (Fork m u vt wt) (Fork n x yt zt)
    | (u ≤ x)          = fork u vt (union wt (Fork n x yt zt))
    | (x < u)          = fork x yt (union (Fork m u vt wt) zt)

mkBag                              :: [α] ⟶ Htree α
mkBag xs                           = fst (mkTwo (length xs) xs)

mkTwo                              :: Int ⟶ [α] ⟶ (Htree α, [α])
mkTwo n xs
    | (n == 0)                     = (Null, xs)
    | (n == 1)                     = (fork (head xs) Null Null, tail xs)
    | otherwise                    = (union xt yt, zs)
      where (xt, ys)               = mkTwo m xs
            (yt,zs)                = mkTwo (n-m) ys
            m                      = n div 2
```

Flexible Arrays

- Operationen

empty	$:: \text{Flex}\ \alpha$
isEmpty	$:: \text{Flex}\ \alpha \longrightarrow \text{Bool}$
access	$:: \text{Flex}\ \alpha \longrightarrow \text{Int} \longrightarrow \alpha$
update	$:: \text{Flex}\ \alpha \longrightarrow \text{Int} \longrightarrow \alpha \longrightarrow \text{Flex}\ \alpha$
hiext	$:: \alpha \longrightarrow \text{Flex}\ \alpha \longrightarrow \text{Flex}\ \alpha$
hirem	$:: \text{Flex}\ \alpha \longrightarrow \text{Flex}\ \alpha$
loext	$:: \alpha \longrightarrow \text{Flex}\ \alpha \longrightarrow \text{Flex}\ \alpha$
lorem	$:: \text{Flex}\ \alpha \longrightarrow \text{Flex}\ \alpha$

Flexible arrays: Axiome

```
hiext x . loext y                    = 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))
access ampty k                       = error „out of range“
access (loext x xf) 0                = x
access (hiext x xf) (k + 1)          = access xf k

access (hiext x xf) k
    | (k < n)          = access xf k
    | (k == n)         = x
    | (k > n)          = error
    where n = length xf
```

Flexible Arrays: Implementierung

```
data Flex α = Null | Leaf α | Fork Int (Flex α) (Flex α)

…
access                      :: Flex α ⟶ Int α
access (Leaf x) 0           = x
access (Fork n xt yt) k     = if k < m then access xt k
                                       else access yt (k – m)
                               where m = size xt


size                        :: Flex α ⟶ Int
size Null                   = 0
size (Leaf x)               = 1
size (Fork n xt yt)         = n
…
```

Fazit