ADT's

Jan Sören Ramm & Stefan Ahrens

Fachhochschule Wedel

03.12.2010

Inhaltsverzeichnis I

Inhaltsverzeichnis II

Einleitung

Grundansatz

- Implementierung und Schnittstelle trennen
- Datenkapselung
- Austauschbarkeit

Schnittstelle	sichtbare Teile von Werten und Operationen
Implementierung	unsichtbare Teile von Werten und Operationen

Vergleich zu einfachen Datentypen

- einfache Datentypen
  - Explizit / konkret über Werte oder Wertebereiche definiert
  - Feste Anzahl von Werten
  - Teilweise stark Programmiersprachen abhängig
  - Beispiele: Integer, ENUM, Boolean, ...
- abstrakte Datentypen
  - ADT werden über Operationen definiert
  - Export der Schnittstelle über Module
  - Vorteile von ADT
    - Sicherheit
    - Flexibilität

Spezifikation

- Modellierende Spezifikation
  - Signatur
  - Semantik (Spezifikation z.B. durch Beschreiben im Pseudocode)
- Algebraische Spezifikation
  - Signatur
  - Erzeuger
  - Axiome

Beispiel einer algebraischen Spezifikation

Signatur und Erzeuger

```
data Stack e =  E
              | S e (Stack e)
empty         :: E
isStackEmpty  :: Stack e -> Bool
push          :: e -> Stack e -> Stack e
pop           :: Stack e -> Stack e
top           :: Stack e -> e
```

Beispiel einer algebraischen Spezifikation

Axiome

```
isStackEmpty empty        = true
isStackEmpty (push s i)   = false
pop (push i s)            = s
top (push i s)            = i
```

Angestrebte Eigenschaften eines ADTs

- Universalität
  - Einmal entworfen und implementiert
- Präzise Beschreibung
  - Schnittstelle eindeutig und vollständig
- Einfachheit
  - Für Anwendung interessiert interne Realisierung nicht
- Kapselung
  - Benutzer weiss genau WAS der ADT macht, aber nicht WIE
- Geschütztheit
  - Anwender / Angreifer kann auf interne Struktur nicht zugreifen
- Modularität
  - Viele kleine Module vereinfachen das Gesamtprogramm

Modulkonzept in Haskell

Beschreibung

- Bilden eigenen Namensraum (namespace)
- Pro Datei nur ein Modul
- Module beginnen laut Konvention mit einem Großbuchstaben
- Keine zyklischen Abhängigkeiten erlaubt
- Untermodule werden durch Punktnotation gebildet

Standardverhalten

Default: Alles wird exportiert

```
module MyModule1 where
  funct1 = ...
  funct2 = ...
```

Im-/Export von Funktionen I

Importieren von funct1 und 2

```
module MyModule2 where
import MyModule1
  funct3 = ...
```

Exportieren von einzelnen Funktionen

```
module MyModule3 (funct1) where
  funct1 = ...
  funct2 = ...
```

Im-/Export von Funktionen II

Exportieren einzelner Funktionen und reexport eines Moduls

```
module MyModule4 (funct1 , module Foo) where
import Foo
  funct1 = ...
  funct2 = ...
```

Importiert nur funct1 aus dem Modul

```
...
import MyModule1 (funct1)
...
```

Modifizierter Import

Importiert alles ausser funct1 aus dem Modul

```
...
import MyModule1 hiding (funct1)
...
```

Umbennenen des importierten Moduls

```
...
import Foo as Bar
...
```

Set

Einleitung

- Mengen sind für viele Anwendung nützlich
- Beinhalten Elemente nur einmal
- Reihenfolge der beinhalteten Elemente irrelevant
- Beispielimplementation für ADT
- Grundlegende Funktionen implementiert

Einfache Funktionen

Definition

```
empty     :: (Eq a) => Set a
isEmpty   :: (Eq a) => Set a -> Bool
member    :: (Eq a) => Set a -> a -> Bool
```

Einfache Funktionen

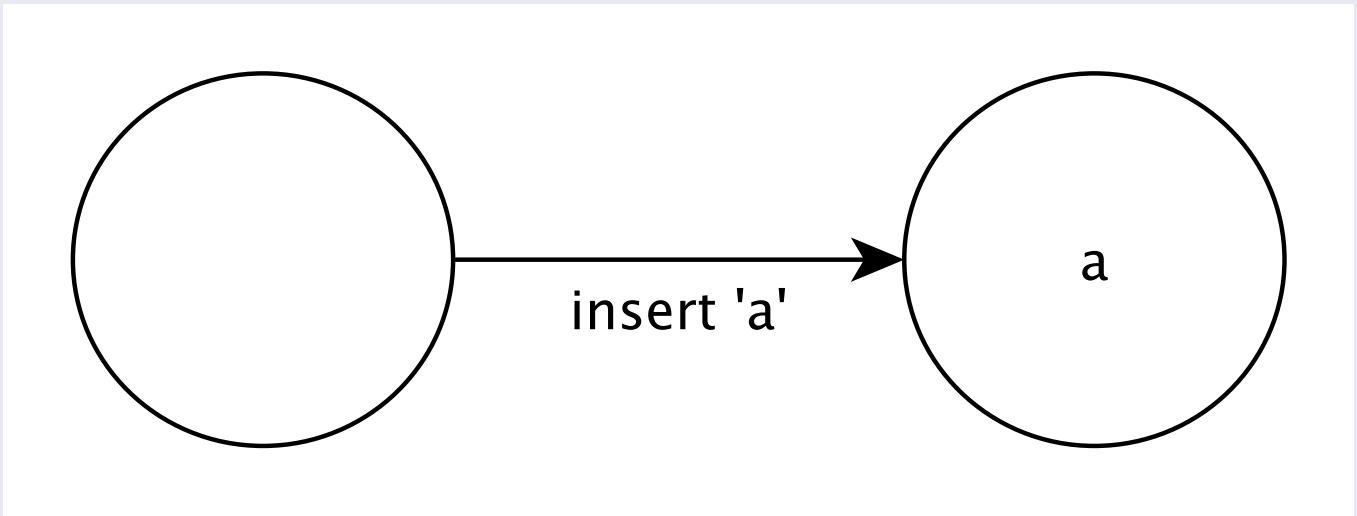
Axiome

```
isEmpty  empty              = true
isEmpty  (insert x  xs)     = false
member   empty y            = false
member   (insert x xs) y    = (x = y)
                              || member xs y
```

Insert

Definition

```
insert    :: (Eq a) => a -> Set a -> Set a
```



Insert

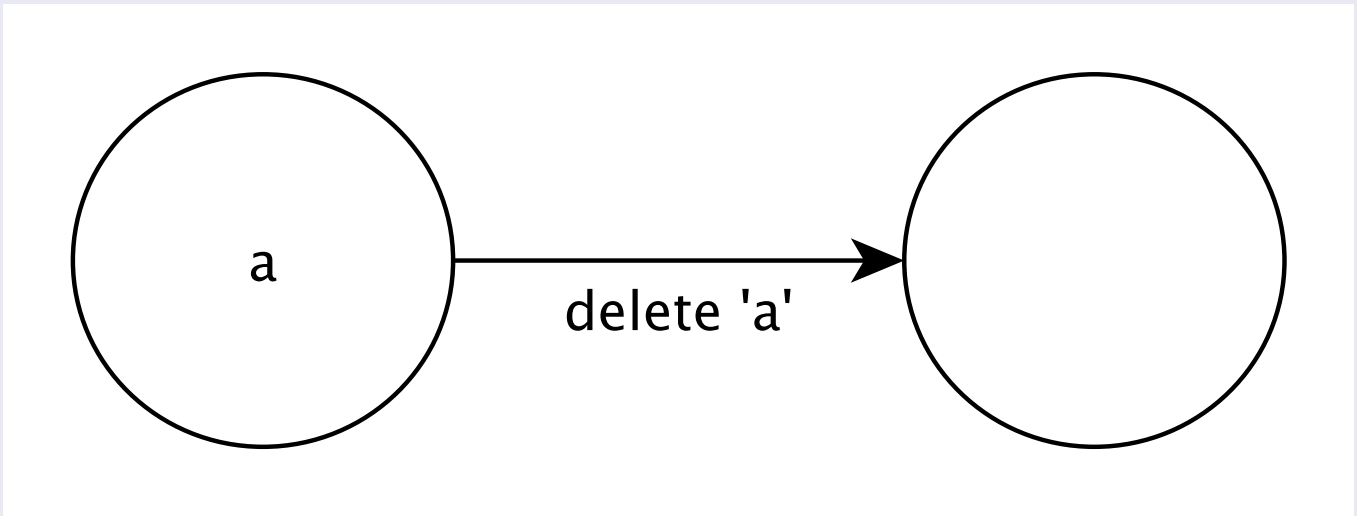
Axiome

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

Delete

Definition

```
delete    :: (Eq a) => a -> Set a -> Set a
```



Delete

Axiome

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

Implementierungsmöglichkeiten

- Listen
- einfache Bäume
- Binäre Suchbäume
- und weitere

Implementierung mit Hilfe von Listen

Bedingungen - zur Verdeutlichung

- Einmaliges Vorkommen eines Elements in einer Liste
- Die Liste ist sortiert

Implementierung mit Hilfe von Listen

Modul und Generator

```
module LSet (LSet, empty, isEmpty, insert,
             member, delete) where

newtype LSet a = LSt ([a])
```

Implementierung mit Hilfe von Listen

Grundfunktionen

```
empty :: LSet a
empty =  LSt []

isEmpty           :: LSet a -> Bool
isEmpty (LSt x) =  null x
```

Implementierung mit Hilfe von Listen

member auf Basis einer sortierten Liste

```
member                  :: (Ord a) =>
                           LSet a -> a -> Bool

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

Implementierung mit Hilfe von Listen

insert für sortierte eindeutige Listen

```
insert                  :: (Ord a) =>
                           a -> LSet a -> LSet a

insert x (LSt xs) =
  LSt (ys ++ [x] ++ (filter (/=x) zs))
  where
    (ys, zs) = span (<x) xs
```

Implementierung mit Hilfe von Listen

delete für sortierte eindeutige Listen

```
delete                  :: (Eq a)  =>
                           a -> LSet a -> LSet a
delete x (LSt xs) =  LSt(filter (/=x) xs)
```

Implementierung mit Hilfe von Bäumen

Modul und Generator

```
module SetByTree (TSet, empty, isEmpty,
                  member, insert, delete)
where
  data TSet a = Null
              | Fork (TSet a) a (TSet a)
```

Implementierung mit Hilfe von Bäumen

Grundfunktionen

```
empty :: TSet a
empty =  Null

isEmpty                    :: TSet a -> Bool
isEmpty Null               =  True
isEmpty (Fork xt y zt)     =  False
```

Implementierung mit Hilfe von Bäumen

member auf einen Baum

```
member            :: (Ord a) =>
                     TSet a -> a -> Bool

member Null x =  False
member (Fork xt y zt) x
  |(x < y)      = member xt x
  |(x == y)     = True
  |(x > y)      = member zt x
```

Implementierung mit Hilfe von Bäumen

insert

```
insert :: (Ord a) =>
          a -> TSet a -> TSet a

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)
```

Implementierung mit Hilfe von Bäumen

delete

```
delete              :: (Ord a) =>
                       a -> TSet a -> TSet a

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)
```

Vergleich der Implementierungen

Funktion	Liste	ausbalancierter Baum
empty	$\mathcal{O}(1)$	$\mathcal{O}(1)$
isEmpty	$\mathcal{O}(1)$	$\mathcal{O}(1)$
member	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$
insert	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$
delete	$\mathcal{O}(n)$	$\mathcal{O}(log(n))$

Flexible Arrays

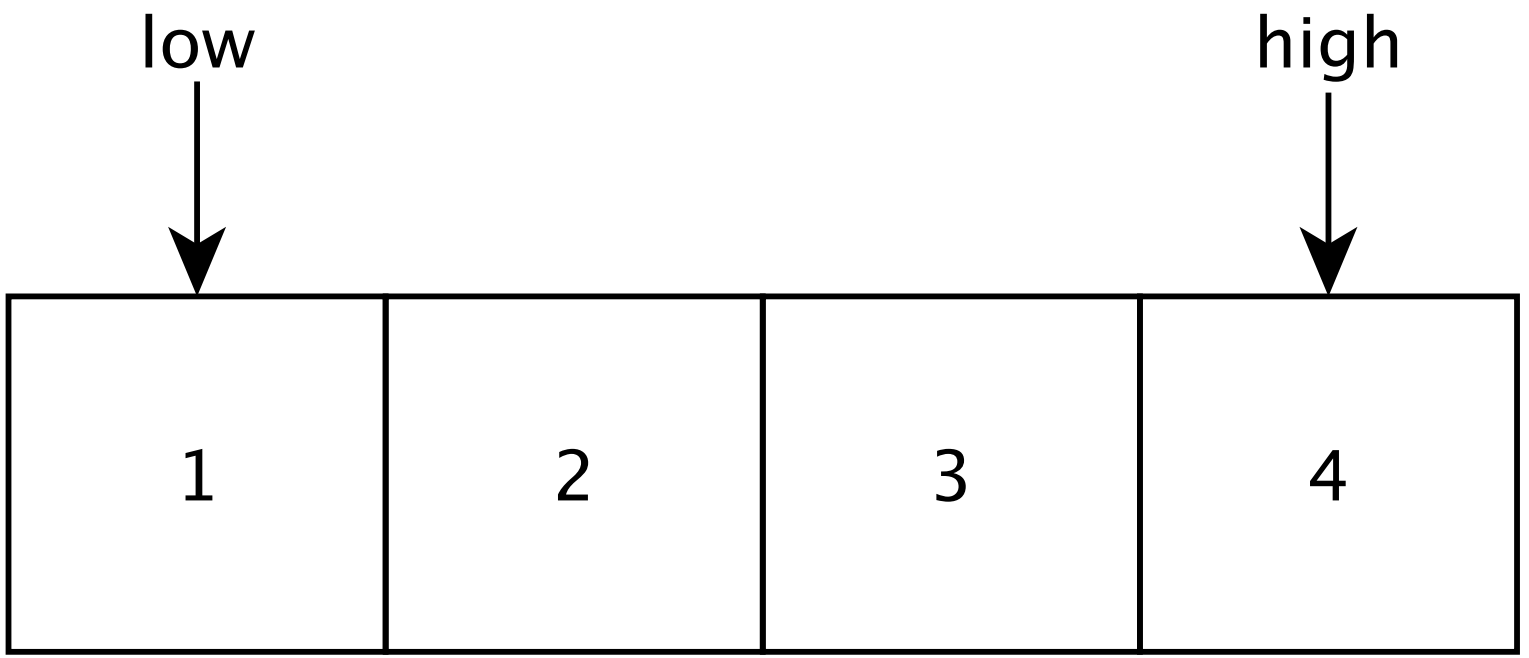
Einleitung

Beschreibung

- Datentyp für endliche Listen mit indiziertem Zugriff
- Keine festen Grenzen
  - Anfügen am Anfang sowie am Ende
- Geeignete Repräsentation durch Bäume
  - Knoten enthalten Anzahl der Elemente (Binary Search Tree)
  - Informationen befindet sich in den Blättern

Einleitung

Arraydarstellung



Definition

Definition

```
empty   :: Flex a
isEmpty :: Flex a -> Bool
access  :: Flex a -> Int -> a
update  :: Flex a -> Int -> a -> Flex a
hiext   :: a -> Flex a -> Flex a
loext   :: a -> Flex a -> Flex a
hirem   :: Flex a -> Flex a
lorem   :: Flex a -> Flex a
```

Definition

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))
```

Implementierungsmöglichkeit

access

```
access :: Flex a -> Int -> a

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
```

Implementierungsmöglichkeit

update

```
update :: Flex a -> Int -> a -> Flex a

update (Leaf y) 0 val           = Leaf val
update (Fork n xt yt) pos val
  |(pos < m) = Fork n (update xt pos val) yt
  |otherwise = Fork n xt
               (update yt (pos - m) val)
  where m     = size xt
```

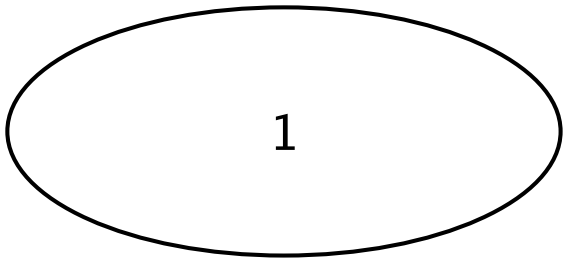
Implementierungsmöglichkeit

hiext

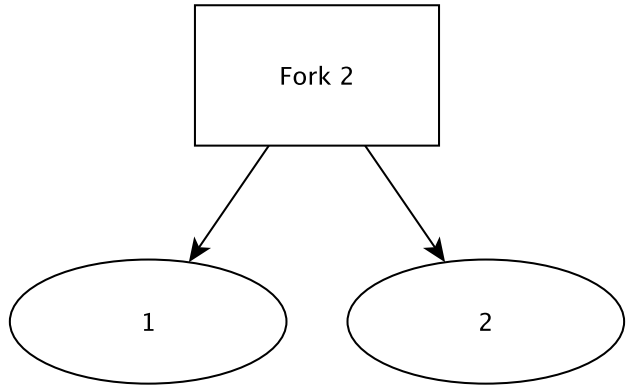
```
hiext :: a -> Flex a -> Flex a

hiext x Null                = Leaf x
hiext x (Leaf y)            =
  Fork 2 (Leaf y) (Leaf x)
hiext x (Fork n xt yt) =
  Fork (n + 1) (Fork n xt yt) (Leaf x)
```

Implementierungsmöglichkeit

hiext 1

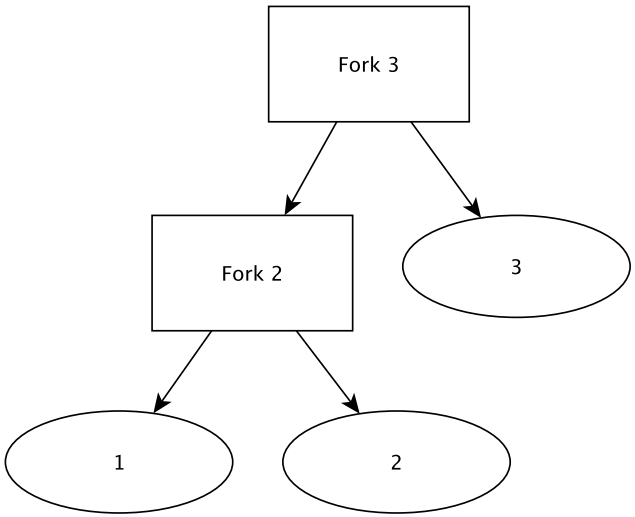


hiext 2 . hiext 1

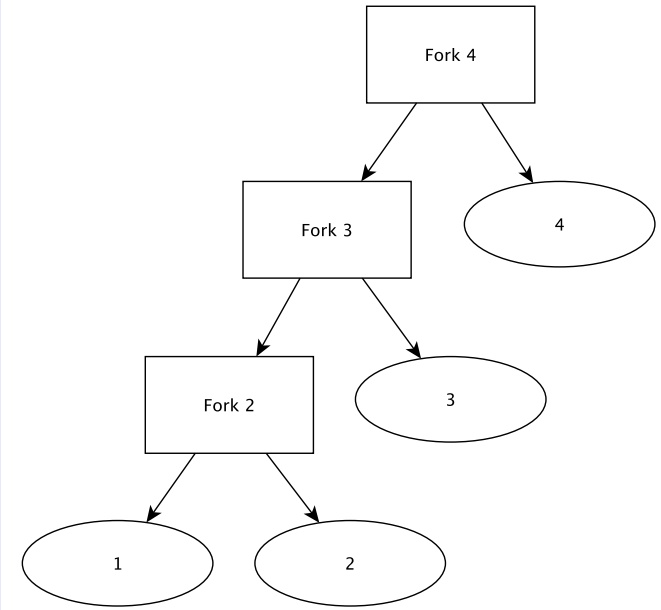


Implementierungsmöglichkeit

hiext 3 . hiext 2 . hiext 1



hiext 4 . hiext 3 . hiext 2 . hiext 1



Implementierungsmöglichkeit

hirem

```
hirem :: Flex a -> Flex a

hirem (Leaf x)                = Null
hirem (Fork (n + 1) xt yt) =
  fork n xt (hirem yt)
```

Vordefinierte ADTs

Vordefinierte ADTs

- Data.List
- Data.Tuple
- Data.Map
- Data.HashTable
- Data.Array
- Data.Set
- Data.Tree
- ...

Quellen

- Richard Bird - Introduction to Functional Programming using Haskell second edition
- `http://de.wikipedia.org/wiki/Abstrakter_Datentyp`
- `http://www.fh-wedel.de/~si/vorlesungen/java/OOP/Adts.html`
- Eduard Wiebe - Konzepte der Programmiersprachen (ss 2007)
- D. Rösner - Algorithmen und Datenstrukturen 1 (22. Nov 2009)
- Alte Vorträge