

Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 08.12.2010: Abstrakte Datentypen

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Rev. 1258

1 [31]

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
 - ▶ Aktionen und Zustände
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

2 [31]

Inhalt

- ▶ Abstrakte Datentypen
 - ▶ Allgemeine Einführung
 - ▶ Realisierung in Haskell
 - ▶ Beispiele

3 [31]

Einfache Bäume

- ▶ Schon bekannt: Bäume

```
data Tree α = Null
            | Node (Tree α) α (Tree α)
```

- ▶ Dazu Test auf Enthaltensein:

```
member' :: Eq α => α -> Tree α -> Bool
member' _ Null = False
member' b (Node l a r) =
  a == b || member' b l || member' b r
```

- ▶ Problem: Suche aufwändig (Backtracking)
- ▶ Besser: Baum geordnet
 - ▶ Noch besser: Baum balanciert

4 [31]

Geordnete Bäume

- ▶ Voraussetzung:
 - ▶ Ordnung auf a ($\text{Ord } a$)
 - ▶ Es soll für alle Bäume gelten:
$$\forall x \ t. \ t = \text{Node } l \ a \ r \longrightarrow (\text{member } x \ l \longrightarrow x < a) \wedge (\text{member } x \ r \longrightarrow a < x)$$
 - ▶ Beispiel für eine Datentyp-Invariante
- ▶ Test auf Enthaltensein vereinfacht:

```
member :: Ord α => α -> Tree α -> Bool
member _ Null = False
member b (Node l a r)
  | b < a = member b l
  | a == b = True
  | b > a = member b r
```

5 [31]

Geordnete Bäume

- ▶ Ordnungserhaltendes Einfügen

```
insert :: Ord α => α -> Tree α -> Tree α
insert a Null = Node Null a Null
insert b (Node l a r)
  | b < a = Node (insert b l) a r
  | b == a = Node l a r
  | b > a = Node l a (insert b r)
```

- ▶ Problem: Erzeugung ungeordneter Bäume möglich.
- ▶ Lösung: Verstecken der Konstrukturen.
- ▶ Warum? E.g. Implementation von geordneten Mengen

6 [31]

Geordnete Bäume als abstrakter Datentyp

- ▶ Es gibt einen Typ $\text{Tree } a$
- ▶ Es gibt Operationen
 - ▶ $\text{empty} :: \text{Ord } \alpha \Rightarrow \text{Tree } \alpha$
 - ▶ Nicht $\text{Null} :: \text{Tree } a$, sonst Konstruktor sichtbar
 - ▶ $\text{insert} :: \text{Ord } \alpha \Rightarrow \alpha \rightarrow \text{Tree } \alpha \rightarrow \text{Tree } \alpha$
 - ▶ $\text{member} :: \text{Ord } \alpha \Rightarrow \alpha \rightarrow \text{Tree } \alpha \rightarrow \text{Bool}$
 - ▶ ... und keine weiteren!
- ▶ Beispiel für einen abstrakten Datentypen
- ▶ Datentyp-Invarianten können außerhalb des definierenden Moduls nicht verletzt werden

7 [31]

Abstrakte Datentypen

Definition (ADT)

Ein abstrakter Datentyp (ADT) besteht aus einem (oder mehreren) Typen und Operationen auf diesem.

- ▶ Werte des Typen können nur über die bereitgestellten Operationen erzeugt werden.
- ▶ Eigenschaften von Werten des Typen (insb. ihre innere Struktur) können nur über die bereitgestellten Operationen beobachtet werden.

Zur Implementation von ADTs in einer Programmiersprache: Möglichkeit der Kapselung durch

- ▶ Module
- ▶ Objekte

8 [31]

ADTs in Haskell: Module

- ▶ Einschränkung der Sichtbarkeit durch **Verkapselung**
- ▶ **Modul**: Kleinste verkapselbare **Einheit**
- ▶ Ein **Modul** umfaßt:
 - ▶ Definitionen von Typen, Funktionen, Klassen
 - ▶ Deklaration der nach außen **sichtbaren** Definitionen
- ▶ Syntax:
`module Name (sichtbare Bezeichner) where Rumpf`
 - ▶ *sichtbare Bezeichner* können leer sein
- ▶ Gleichzeitig: Modul $\hat{=}$ Übersetzungseinheit (getrennte Übersetzung)

9 [31]

Beispiel: Exportliste für Bäume

- ▶ Export als **abstrakter Datentyp**
`module OrdTree (Tree, insert, member, empty) where`
 - ▶ Typ `Tree` extern sichtbar
 - ▶ Konstruktoren versteckt
- ▶ Export als **konkreter Datentyp**
`module OrdTree (Tree(..), insert, member, empty) where`
 - ▶ Konstruktoren von `Tree` sind extern sichtbar
 - ▶ Pattern Matching ist möglich
 - ▶ Erzeugung auch von ungeordneten Bäumen möglich

10 [31]

Benutzung von ADTs

- ▶ **Operationen** und **Typen** müssen bekannt gemacht werden (Import)
- ▶ Möglichkeiten des Imports:
 - ▶ **Alles** importieren
 - ▶ Nur bestimmte **Operationen** und **Typen** importieren
 - ▶ Bestimmte **Typen** und **Operationen** nicht importieren

11 [31]

Importe in Haskell

- ▶ **Schlüsselwort**: `import Name [hiding] (Bezeichner)`
- ▶ **Bezeichner** geben an, **was** importiert werden soll:
 - ▶ Ohne Bezeichner wird **alles** importiert
 - ▶ Mit `hiding` werden Bezeichner **nicht** importiert
 - ▶ Alle Importe stehen immer am **Anfang** des Moduls
- ▶ Qualifizierter Import zur Vermeidung von Namenskollisionen
 - ▶ `import qualified Name as OtherName`
 - ▶ Z. B. `import qualified Data.Map as M`

12 [31]

Beispiel: Importe von Bäumen

Import(e)	Bekannte Bezeichner
<code>import OrdTree</code>	<code>Tree, insert, member, empty</code>
<code>import OrdTree(Tree, empty)</code>	<code>Tree, empty</code>
<code>import OrdTree(insert)</code>	<code>insert</code>
<code>import OrdTree hiding (member)</code>	<code>Tree, empty, insert</code>
<code>import OrdTree(empty)</code>	<code>empty,</code>
<code>import OrdTree hiding (empty)</code>	<code>Tree, insert, member</code>

13 [31]

Baumtraversion als Funktion höherer Ordnung

- ▶ Nützlich: Traversion als generisches fold
- ▶ Dadurch Iteration über den Baum möglich, ohne Struktur offenzulegen

```
foldT :: (α → β → β → β) → β → Tree α → β
foldT f e Null = e
foldT f e (Node l a r) =
  f a (foldT f e l) (foldT f e r)
```

- ▶ Damit externe Definition von Aufzählung möglich:

```
enum :: Ord α ⇒ Tree α → [α]
enum = foldT (λx l1 l2 → l1 ++ x : l2) []
```

14 [31]

Schnittstelle vs. Implementation

- ▶ Gleiche **Schnittstelle** kann unterschiedliche **Implementationen** haben
- ▶ Beispiel: (endliche) Mengen

15 [31]

Endliche Mengen: Typsignaturen (1)

- ▶ Abstrakter Datentyp für endliche Mengen (polymorph über Elementtyp)

```
type Set a
```

- ▶ Leere Menge:

```
empty :: Set a
```

- ▶ Einfügen in eine Menge:

```
insert :: Ord a ⇒ a → Set a → Set a
```

- ▶ Test auf Enthaltensein

```
member :: Ord a ⇒ a → Set a → Bool
```

16 [31]

Endliche Mengen: Typsignaturen (2)

- ▶ Test auf leere Menge

```
null :: Set a → Bool
```

- ▶ Vereinigung

```
union :: Ord a ⇒ Set a → Set a → Set a
```

- ▶ Schnittmenge

```
intersection :: Ord a ⇒ Set a → Set a → Set a
```

- ▶ Umwandlung zu Listen

```
toList :: Set a → [a]  
fromList :: Ord a ⇒ [a] → Set a
```

- ▶ Mappen und Falten:

```
map :: (Ord a, Ord b) ⇒ (a → b) → Set a → Set b  
fold :: (a → b → b) → b → Set a → b
```

17 [31]

Endliche Mengen: Eigenschaften

- ▶ Die leere Menge

```
empty null  
not (empty (insert x s))
```

- ▶ Extensionalität

```
s1 == s2 ⇔ (∀x. member x s1 ⇔ member x s2)
```

- ▶ Einfügen und Enthaltensein

```
insert x (insert y s) == insert y (insert x s)  
member x (insert x s)
```

- ▶ Schnittmenge

```
member x (intersection s1 s2) ==  
member x s1 && member x s2
```

- ▶ Vereinigung

```
member x (union s1 s2) ==  
member x s1 || member x s2
```

18 [31]

Endliche Mengen: Implementierung

- ▶ Für den Anwender von `Data.Set` **irrelevant!**
- ▶ Wichtig aus Implementierungssicht: **Effizienz**
- ▶ Verschiedene Möglichkeiten der Repräsentation

- ▶ Sortierte Listen: **type** `Set a = [a]`

- ▶ Funktionen: **type** `Set a = a → Bool`

- ▶ In der Tat verwendet: Balancierte Bäume

```
data Set a = Tip | Bin Int a (Set a) (Set a)  
(Int gibt die Größe des Baumes an.)
```

19 [31]

Endliche Abbildungen

- ▶ Eine Sichtweise: Ersatz für Hashtables in imperativen Sprachen. **Sehr nützlich!**
- ▶ Abstrakter Datentyp für endliche Abbildungen (polymorph über Schlüssel- und Werttyp)

```
type Map a b
```

- ▶ Leere Abbildung:

```
empty :: Map a b
```

- ▶ Hinzufügen eines Schlüssel/Wert-Paars

```
insert :: Ord a ⇒ a → b → Map a b → Map a b
```

- ▶ Test auf Enthaltensein

```
member :: Ord a ⇒ a → Map a b → Bool
```

20 [31]

Weitere Funktionen

- ▶ Test auf leere Abbildung

```
null :: Map a b → Bool
```

- ▶ Nachschlagen eines Werts

```
lookup :: Ord a ⇒ a → Map a b → Maybe b  
(!) :: Ord a ⇒ Map a b → a → b
```

- ▶ Löschen

```
delete :: Ord a ⇒ a → Map a b → Map a b
```

- ▶ Einfügen und Duplikatkonflikte lösen

```
insertWith :: Ord a ⇒  
(b → b → b) → a → b → Map a b → Map a b
```

- ▶ Mappen und Falten:

```
map :: (b → c) → Map a b → Map a c  
fold :: (b → c → c) → c → Map a b → c
```

21 [31]

Endl. Abbildungen: Anwendungsbeispiele

- ▶ Anzahl von Artikeln im Warenhaus

```
type Warehouse = Data.Map String Int  
  
nLeft :: Warehouse → String → Int → Bool  
nLeft w art n =  
  case s `Data.Map.lookup` w of  
    Nothing → False  
    Just m → m ≥ n  
  
addArticle :: String → Int → Warehouse →  
  Warehouse  
addArticle art n w =  
  Data.Map.insertWith (+) art w
```

22 [31]

Weiterer Datentyp: Prioritätswarteschlangen

- ▶ Signatur von Prioritätswarteschlangen ähnlich Stacks und FIFO Queues:

```
type PriorityQueue k a
```

k steht für Priorität, a ist eigentlicher Wert

- ▶ Operationen:

- ▶ `empty :: PriorityQueue k a`
- ▶ `null :: Ord k ⇒ PriorityQueue k a → Bool`
- ▶ `insert :: Ord k ⇒ k → a → PriorityQueue k a → PriorityQueue k a`
- ▶ `minKeyValue :: Ord k ⇒ PriorityQueue k a → (k, a)`
- ▶ `deleteMin :: Ord k ⇒ PriorityQueue k a → PriorityQueue k a`

23 [31]

Implementierung mittels Heaps

- ▶ Ein **Heap** ist eine baumartige Datenstruktur mit der **Heap-Eigenschaft**:

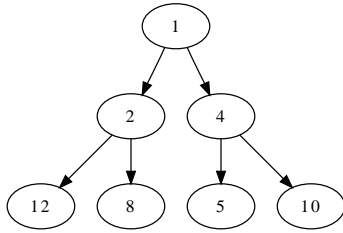
```
data Heap k a = Nil | Branch k a (Heap k a) (Heap k a)  
  
heapProp :: Ord k ⇒ Heap k a → Bool  
heapProp Nil = True  
heapProp (Branch k a l r) =  
  k ≤ min (minH k l) (minH k r)  
  && heapProp l && heapProp r  
  where minH k Nil = k  
        minH _ (Branch k a l r) =  
          min k (min (minH k l) (minH k r))
```

- ▶ Wurzelement jedes Teilbaums ist minimales Element des Teilbaums

24 [31]

Beispiel: Heap-Eigenschaft

- ▶ Ein vollständiger binärer Baum mit Heap-Eigenschaft
- ▶ **Kein** geordneter Baum



25 [31]

Binäre Heaps

- ▶ **Vollständigkeit** zusätzlich zur Heap-Eigenschaft

```
depth :: Ord k => Heap k a -> Int
depth Nil = 0
depth (Branch _ _ l r) =
  1 + max (depth l) (depth r)

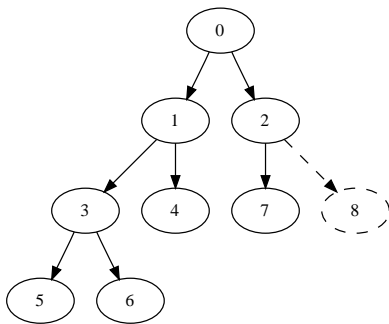
completeAt n Nil = False
completeAt n (Branch _ _ l r) =
  if n > 0 then completeAt (n - 1) l &&
    completeAt (n - 1) r
  else not (null l) && not (null r)

complete t = d < 3 || completeAt (d - 3) t
  where d = depth t
```

26 [31]

Beispiel: Vollständigkeit

- ▶ Mit Knoten 8: vollständig
- ▶ Ohne 8: höhenbalanciert, aber nicht vollständig



27 [31]

Operationen

- ▶ Exportierte Operationen:

```
singleton :: Ord k => k -> a -> Heap k a
singleton k a = Branch k a Nil Nil

empty :: Heap k a
empty = Nil

insert :: Ord k => k -> a -> Heap k a -> Heap k a
insert k a Nil = singleton k a
insert k a (Branch k' a' l r) =
  | k < k' = Branch k a (insert k' a' r) l
  | otherwise = Branch k' a' (insert k a r) l

minKeyValue :: Ord k => Heap k a -> (k, a)
minKeyValue (Branch k a _ _) = (k, a)
```

28 [31]

Entfernen des minimalen Elements

- ▶ Zum Entfernen: "Hochziehen" des jeweils kleineren Kindelements

```
deleteMin :: Ord k => Heap k a -> Heap k a
deleteMin t =
  case t of
    Branch _ _ Nil r -> r
    Branch _ _ l Nil -> l
    Branch k a (l@(Branch lk la _ _))
      (r@(Branch rk ra _ _)) ->
      if lk < rk then Branch lk la (deleteMin l) r
      else Branch rk ra l (deleteMin r)
```

- ▶ Beispiele siehe ../uebung/ueb06/trees.pdf

29 [31]

Effizienz

- ▶ Laufzeitverhalten: $\mathcal{O}(\log(n))$ für insert und deleteMin, $\mathcal{O}(1)$ für minKeyValue, singleton und empty.
- ▶ **Pairing Heaps** als Alternative zu Binären Heaps
 - ▶ Worst-case Laufzeit für deleteMin in $\mathcal{O}(n)$
 - ▶ Aber: amortisierte Kosten in $\mathcal{O}(\log(n))$ und in der Praxis erstaunlich schnell
- ▶ Bsp.: 10^6 zufällig priorisierte Elemente einfügen und entfernen
 - ▶ Haskell: Laufzeit $\approx 7s$ (im Vergleich: $\approx 12.7s$ für Binärheap)
 - ▶ OCaml: $\approx 3.3s$
 - ▶ Java (Binärer Heap als Array): $\approx 3.6s$
- ▶ Tests auf 2GHz Intel Dual Core

30 [31]

Zusammenfassung

- ▶ **Abstrakte Datentypen (ADTs)**:
 - ▶ Besteht aus **Typen** und **Operationen** darauf
- ▶ Realisierung in Haskell durch **Module**
- ▶ Beispieldatentypen: endliche Mengen und Abbildungen, Prioritätswarteschlangen
- ▶ Nächste Vorlesung: ADTs durch **Eigenschaften** spezifizieren

Vorlesung nächste Woche (15.12.2010) **entfällt** wegen Tag der Lehre!

Der Übungsbetrieb findet **normal** statt.

31 [31]