Praktische Informatik 3: Funktionale Programmierung
Vorlesung 7 vom 29.11.2016: Abstrakte Datentypen

Christoph Lüth

Universität Bremen

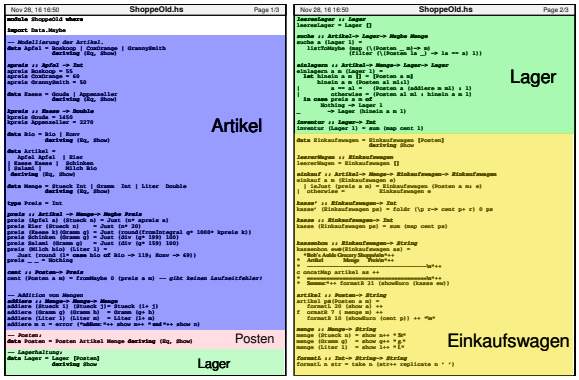
Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
  - Spezifikation und Beweis
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

Refakturierung im Einkaufsparadies



Warum Modularisierung?

- Übersichtlichkeit der Module

  Lesbarkeit
- Getrennte Übersetzung

  technische Handhabbarkeit
- Verkapselung

  konzeptionelle Handhabbarkeit

Abstrakte Datentypen

Definition (Abstrakter Datentyp)

Ein abstrakter Datentyp (ADT) besteht aus einem (oder mehreren) Typen und Operationen darauf, mit folgenden Eigenschaften:

- Werte des Typen können nur über die bereitgestellten Operationen erzeugt werden:
- Eigenschaften von Werten des Typen werden nur über die bereitgestellten Operationen beobachtet;
- Einhaltung von Invarianten über dem Typ kann garantiert werden.

Implementation von ADTs in einer Programmiersprache:

- benötigt Möglichkeit der Kapselung (Einschränkung der Sichtbarkeit)
- bspw. durch Module oder Objekte

ADTs vs. algebraische Datentypen

- Algebraische Datentypen
  - Frei erzeugt
  - Keine Einschränkungen
  - Insbesondere keine Gleichheiten ($[] \neq x:xs$, $x:ls \neq y:ls$ etc.)
- ADTs:
  - Einschränkungen und Invarianten möglich
  - Gleichheiten möglich

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
- Gleichzeitig: Modul $\triangleq$ Übersetzungseinheit (getrennte Übersetzung)

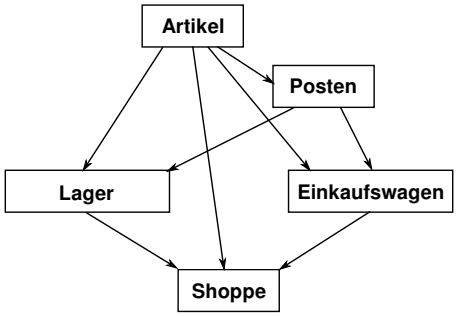
Module: Syntax

- Syntax:

```
module Name(Bezeichner) where Rumpf
```

- Bezeichner können leer sein (dann wird alles exportiert)
- Bezeichner sind:
  - Typen: T, T(c1, ..., cn), T(..)
  - Klassen: C, C(f1, ..., fn), C(..)
  - Andere Bezeichner: Werte, Felder, Klassenmethoden
  - Importierte Module: **module** M
- Typsynonyme und Klasseninstanzen bleiben sichtbar
- Module können rekursiv sein *(don't try at home)*

Refakturierung im Einkaufsparadies: Modularchitektur



Refakturierung im Einkaufsparadies I: Artikel

- Es wird alles exportiert
- Reine Datenmodellierung

```
module Artikel where

import Data.Maybe

data Apfel = Boskoop | CoxOrange | GrannySmith
           deriving (Eq, Ord, Show)

apreis :: Apfel → Int

data Kaese = Gouda | Appenzeller
           deriving (Eq, Ord, Show)

kpreis :: Kaese → Double
```


Refakturierung im Einkaufsparadies II: Posten

```
module Posten(
  Posten,
  artikel,
  menge,
  posten,
  cent,
  hinzu) where
```

- Implementiert ADT Posten:

```
data Posten = Posten Artikel Menge
```

  - Konstruktor wird nicht exportiert
- Garantierte Invariante:
  - Posten hat immer die korrekte Menge zu Artikel

```
posten a m =
  case preis a m of
    Just _ → Just (Posten a m)
    Nothing → Nothing
```


Refakturierung im Einkaufsparadies III: Lager

```
module Lager(
  Lager,
  leeresLager,
  einlagern,
  suche,
  inventur
  ) where

import Artikel
import Posten
```

- Implementiert ADT Lager
- Signatur der exportierten Funktionen:

```
leeresLager :: Lager

einlagern :: Artikel→ Menge→ Lager→ Lager

suche :: Artikel→ Lager→ Maybe Menge

inventur :: Lager→ Int
```

- Garantierte Invariante:
  - Lager enthält keine doppelten Artikel

Refakturierung im Einkaufsparadies IV: Einkaufswagen

```
module Einkaufswagen(
  Einkaufswagen,
  leererWagen,
  einkauf,
  kasse,
  kassenbon
  ) where
```

- Implementiert ADT Einkaufswagen

```
data Einkaufswagen =
                Einkaufswagen [Posten]
```

- Garantierte Invariante:
  - Korrekte Menge zu Artikel im Einkaufswagen

```
einkauf :: Artikel→ Menge
                  → Einkaufswagen
                  → Einkaufswagen
einkauf a m (Einkaufswagen e) =
  case posten a m of
    Just p → Einkaufswagen (p: e)
    Nothing→ Einkaufswagen e
```

- Nutzt dazu ADT Posten

Benutzung von ADTs

- Operationen und Typen müssen importiert werden
- Möglichkeiten des Imports:
  - Alles importieren
  - Nur bestimmte Operationen und Typen importieren
  - Bestimmte Typen und Operationen nicht importieren

Importe in Haskell

- Syntax:

```
import [qualified] M [as N] [hiding][(Bezeichner)]
```

- *Bezeichner* geben an, was importiert werden soll:
  - Ohne Bezeichner wird alles importiert
  - Mit **hiding** werden Bezeichner nicht importiert
- Für jeden exportierten Bezeichner f aus M wird importiert
  - f und qualifizierter Bezeichner M.f
  - **qualified**: nur qualifizierter Bezeichner M.f
  - Umbenennung bei Import mit as (dann N.f)
  - Klasseninstanzen und Typsynonyme werden immer importiert
- Alle Importe stehen immer am Anfang des Moduls

Beispiel

```
module M(a,b) where...
```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	*(nothing)*
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	*(nothing)*
import qualified M(a)	M.a
import M **hiding** ()	a, b, M.a, M.b
import M **hiding** (a)	b, M.b
import qualified M **hiding** ()	M.a, M.b
import qualified M **hiding** (a)	M.b
import M **as** B	a, b, B.a, B.b
import M **as** B(a)	a, B.a
import qualified M **as** B	B.a, B.b

Quelle: Haskell98-Report, Sect. 5.3.4

Ein typisches Beispiel

- Modul implementiert Funktion, die auch importiert wird
- Umbenennung nicht immer praktisch
- Qualifizierter Import führt zu langen Bezeichnern
- Einkaufswagen implementiert Funktionen `artikel` und `menge`, die auch aus Posten importiert werden:

```
import Posten hiding (artikel, menge)
import qualified Posten as P(artikel, menge)
```

```
artikel :: Posten→ String
artikel p =
  formatL 20 (show (P.artikel p)) ++
  formatR 7  (menge (P.menge p)) ++
  formatR 10 (showEuro (cent p)) ++ "\n"
```


Schnittstelle vs. Implementation

- Gleiche Schnittstelle kann unterschiedliche Implementationen haben
- Beispiel: (endliche) Abbildungen

Endliche Abbildungen

- Viel gebraucht, oft in Abwandlungen (Hashtables, Sets, Arrays)
- Abstrakter Datentyp für endliche Abbildungen:
  - Datentyp

    ```
    data Map α β
    ```

  - Leere Abbildung:

    ```
    empty :: Map α β
    ```

  - Abbildung auslesen:

    ```
    lookup :: Ord α⇒ α→ Map α β→ Maybe β
    ```

  - Abbildung ändern:

    ```
    insert :: Ord α⇒ α→ β→ Map α β→ Map α β
    ```

  - Abbildung löschen:

    ```
    delete :: Ord α⇒ α→ Map α β→ Map α β
    ```


Eine naheliegende Implementation

- Modellierung als Haskell-Funktion:

```
data Map α β = Map (α→ Maybe β)
```

- Damit einfaches `lookup`, `insert`, `delete`:

```
empty = Map (λx→ Nothing)
```

```
lookup a (Map s) = s a
```

```
insert a b (Map s) =
  Map (λx→ if x == a then Just b else s x)
```

```
delete a (Map s) =
  Map (λx→ if x == a then Nothing else s x)
```

- Instanzen von Eq, Show nicht möglich
- Speicherleck: überschriebene Zellen werden nicht freigegeben

Endliche Abbildungen: Anwendungsbeispiel

- Lager als endliche Abbildung:

```
data Lager = Lager (M.Map Artikel Posten)
```

- Artikel suchen:

```
suche :: Artikel→ Lager→ Maybe Menge
suche a (Lager l) = fmap menge (M.lookup a l)
```

- Ins Lager hinzufügen:

```
einlagern :: Artikel→ Menge→ Lager→ Lager
einlagern a m (Lager l) =
 case posten a m of
   Just p  → case M.lookup a l of
     Just q  → Lager (M.insert a (fromJust (hinzu q p)) l)
     Nothing → Lager (M.insert a p l)
   Nothing → Lager l
```

- Für Inventur fehlt Möglichkeit zur Iteration
- Daher: Map als Assoziativliste

Map als Assoziativliste

```
newtype Map α β = Map [(α, β)]
```

- Zusatzfunktionalität:
  - Iteration (`foldr`)

    ```
    fold :: Ord α⇒ ((α, β)→ γ→ γ)→ γ→ Map α β→ γ
    fold f e (Map ms) = foldr f e ms
    ```

  - Instanzen von Eq und Show

    ```
    instance (Eq α, Eq β)⇒ Eq (Map α β) where
      Map s1 == Map s2 =
        null (s1 \\ s2) && null (s1 \\ s2)
    ```

  - ... ist aber ineffizient (Zugriff/Löschen in $\mathcal{O}(n)$)
- Deshalb: balancierte Bäume

AVL-Bäume und Balancierte Bäume

AVL-Bäume

Ein Baum ist ausgeglichen, wenn

- alle Unterbäume ausgeglichen sind, und
- der Höhenunterschied zwischen zwei Unterbäumen höchstens eins beträgt.

Balancierte Bäume

Ein Baum ist balanciert, wenn

- alle Unterbäume balanciert sind, und
- für den linken und rechten Unterbaum l, r gilt:

$$size(l) \leq w \cdot size(r) \quad (1)$$
$$size(r) \leq w \cdot size(l) \quad (2)$$

w — Gewichtung (Parameter des Algorithmus)

Implementation von balancierten Bäumen

- Der Datentyp

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

- Gewichtung (Parameter des Algorithmus):

```
type Weight = Int
```

```
weight :: Weight
```

- Hilfskonstruktor, setzt Größe (l, r balanciert)

```
node :: Tree α→ α→ Tree α→ Tree α
node l n r = Node h l n r where
                h = 1+ size l+ size r
```

- Hilfskonstruktor, balanciert ggf. neu aus:

```
mkNode :: Tree α→ α→ Tree α→ Tree α
```

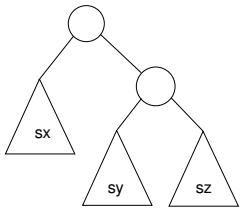

Balance sicherstellen

- Problem:

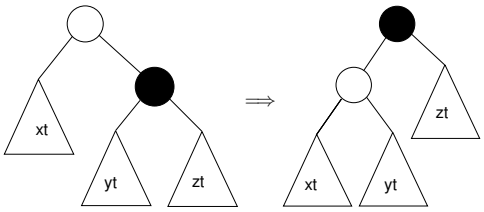
  Nach Löschen oder Einfügen zu großes Ungewicht

- Lösung:

  Rotieren der Unterbäume

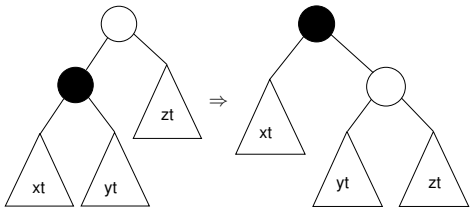


Linksrotation




```
rotl :: Tree α→ Tree α
rotl (Node _ xt y (Node _ yt x zt)) =
  node (node xt y yt) x zt
```

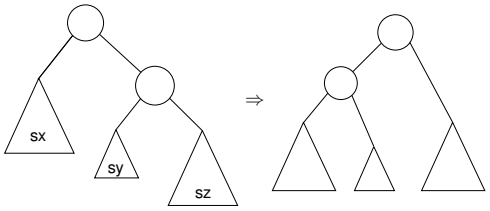

Rechtsrotation




```
rotr :: Tree α→ Tree α
rotr (Node _ (Node _ ut y vt) x rt) =
  node ut y (node vt x rt)
```

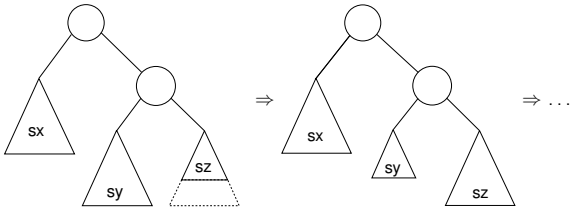

Balanciertheit sicherstellen

- Fall 1: Äußerer Unterbaum zu groß
- Lösung: Linksrotation



Balanciertheit sicherstellen

- Fall 2: Innerer Unterbaum zu groß oder gleich groß
- Reduktion auf vorherigen Fall durch Rechtsrotation des Unterbaumes



Balance sicherstellen

- Hilfsfunktion: Balance eines Baumes

```
bias :: Tree α→ Ordering
bias Null = EQ
bias (Node _ lt _ rt) = compare (size lt) (size rt)
```

- Zu implementieren: mkNode lt y rt
  - Voraussetzung: lt, rt balanciert
  - Konstruiert neuen balancierten Baum mit Knoten y
- Fallunterscheidung:
  - rt zu groß, zwei Unterfälle:
    - Linker Unterbaum von rt kleiner (Fall 1): bias rt == LT
    - Linker Unterbaum von rt größer/gleich groß (Fall 2): bias rt == EQ, bias rt == GT
  - lt zu groß, zwei Unterfälle (symmetrisch).

Konstruktion eines ausgeglichenen Baumes

- Voraussetzung: lt, rt balanciert

```
mkNode lt x rt
  | ls+ rs < 2  = node lt x rt
  | weight∗ ls < rs  =
      if bias rt == LT then rotl (node lt x rt)
      else rotl (node lt x (rotr rt))
  | ls > weight∗ rs =
      if bias lt == GT then rotr (node lt x rt)
      else rotr (node (rotl lt) x rt)
  | otherwise   = node lt x rt where
    ls = size lt; rs= size rt
```

Balancierte Bäume als Maps

- Endliche Abbildung: Bäume mit (key, value) Paaren

```
type Map α β = Tree (α, β)
```

- insert fügt neues Element ein:

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
insert k v Null = node Null (k, v) Null
insert k v (Node n l a@(kn, _) r)
  | k < kn  = mkNode (insert k v l) a r
  | k == kn = Node n l (k, v) r
  | k > kn  = mkNode l a (insert k v r)
```

- lookup liest Element aus
- remove löscht ein Element
  - Benötigt Hilfsfunktion join :: Tree α→ Tree α→ Tree α

Zusammenfassung Balancierte Bäume

- Auslesen, einfügen und löschen: logarithmischer Aufwand ($\mathcal{O}(\log n)$)
- Fold: linearer Aufwand ($\mathcal{O}(n)$)
- Guten durchschnittlichen Aufwand
- Auch in der Haskell-Bücherei: Data.Map (mit vielen weiteren Funktionen)

Defizite von Haskells Modulsystem

- Signatur ist nur implizit
  - Exportliste enthält nur Bezeichner
  - Wünschenswert: Signatur an der Exportliste annotierbar, oder Signaturen in separater Datei
  - In Java: Interfaces
- Klasseninstanzen werden immer exportiert.
- Kein Paket-System

ADTs vs. Objekte

- ADTs (Haskell): Typ plus Operationen
- Objekte (z.B. Java): Interface, Methoden.
- Gemeinsamkeiten:
  - Verkapselung (information hiding) der Implementation
- Unterschiede:
  - Objekte haben internen Zustand, ADTs sind referentiell transparent;
  - Objekte haben Konstruktoren, ADTs nicht (Konstruktoren nicht unterscheidbar)
  - Vererbungsstruktur auf Objekten (Verfeinerung für ADTs)
  - Java: `interface` eigenes Sprachkonstrukt
  - Java: `packages` für Sichtbarkeit

Zusammenfassung

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