Praktische Informatik 3: Funktionale Programmierung
Vorlesung 8 vom 21.12.2020: Abstrakte Datentypen

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Universität Bremen

Wintersemester 2020/21

Organisatorisches

- Abgabe des 7. Übungsblattes in Gruppen zu **drei** Studenten.
  - Bitte **jetzt** eine Gruppe suchen!
- Klausurtermine:
  - Klausur: 03.02.2020, 10:00/11:30/15:00
  - Wiederholungstermin: 21.04.2020, 10:00/11:30/15:00

Fahrplan

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

Inhalt

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

I. Modularisierung und Abstrakte Datentypen

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:

1. Werte des Typen können nur über die Operationen **erzeugt** werden
2. Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**
3. 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** durch **Konstruktoren**
  - Keine Einschränkungen
  - Insbesondere keine Gleichheiten der Konstruktoren (`[] ≠ x:xs`, `x:ls ≠ y:ls` etc.)
- ADTs:
  - Keine ausgezeichneten Konstruktoren
  - Einschränkungen und Invarianten möglich
  - Gleichheiten möglich

ADTs vs. Objekte

- ADTs (z.B. 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
  - Vererbungsstruktur auf Objekten (Verfeinerung für ADTs)
  - Java: `interface` eigenes Sprachkonstrukt
  - Java: `packages` für Sichtbarkeit

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 ≙ Übersetzungseinheit (getrennte Übersetzung)

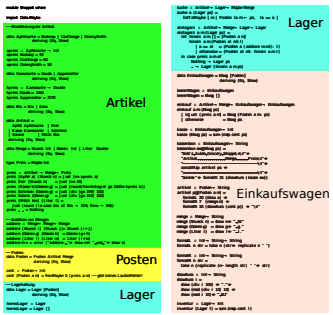
Module: Syntax

- Syntax:

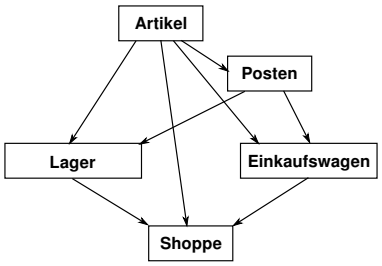
```
module Name(Bezeichner) where Rumpf
```

- 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



Refakturierung im Einkaufsparadies: Modularchitektur



Refakturierung im Einkaufsparadies I: Artikel

- Es wird **alles** exportiert
- Reine Datenmodellierung

```
module Artikel where

data Apfelsorte = Boskoop | CoxOrange | GrannySmith
apreis :: Apfelsorte → Int

data Kaesesorte = Gouda | Appenzeller
kpreis :: Kaesesorte → Double

data Menge = Stueck Int | Gramm  Int | Liter  Double
addiere :: Menge→ Menge→ Menge
```


Refakturierung im Einkaufsparadies II: Posten

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

- Implementiert ADT `Posten`:

```
data Posten = Posten Artikel Menge
              deriving (Eq, Show)

artikel :: Posten → Artikel
artikel (Posten a _) = a
```

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

```
posten :: Artikel→ Menge→ Maybe Posten
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,
  liste,
  inventur
  ) where
import Artikel
import Posten
```

- Implementiert ADT `Lager`

```
data Lager
```

- Signatur der exportierten Funktionen:

```
leeresLager :: Lager

einlagern :: Artikel→ Menge→ Lager→ Lager

suche a (Lager l) =  M.lookup a l

liste (Lager m) = M.toList m

inventur = sum ∘ map (fromJust ∘ uncurry preis) ∘ liste
```

- **Invariante**: `Lager` enthält keine doppelten Artikel

Refakturierung im Einkaufsparadies IV: Einkaufswagen

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

- ADT durch **Verkapselung**:

```
data Einkaufswagen = Ekwg [Posten]
                   deriving (Eq, Show)
```

  - Ein Typsynmonym würde exportiert
- **Invariante**: Korrekte Menge zu Artikel im Einkaufswagen

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

  - Nutzt dazu ADT `Posten`

Refakturierung im Einkaufsparadies V: Hauptmodul

```
module Shoppe where

import Artikel
import Lager
import Einkaufswagen
```

- Nutzt andere Module

```
w0= leererWagen
w1= einkauf (Apfel Boskoop) (Stueck 3) w0
w2= einkauf Schinken (Gramm 50) w1
w3= einkauf (Milch Bio) (Liter 1) w2
w4= einkauf Schinken (Gramm 50) w3
```


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"
```


Was zum Nachdenken

Übung 8.1: Import

Warum schreibt man

```
import Prelude hiding (repeat)
```

und was bewirkt das? (Hinweis: `Prelude` ist das Modul der vordefinierten Funktionen.)

Lösung: Die Import-Anweisung import alle vordefinierten Funktionen **bis auf** `repeat`. Dadurch können wir `repeat` selber (anders) definieren.

II. Schnittstelle vs. Implementation

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 Menge)
  ```
- Artikel suchen:
  ```
  suche a (Lager l) =  M.lookup a l
  ```
- Ins Lager hinzufügen:
  ```
  einlagern :: Artikel→ Menge→ Lager→ Lager
  einlagern a m (Lager l) = case posten a m of
     Just _  → case M.lookup a l of
       Just q  → Lager (M.insert a (addiere m q) l)
       Nothing → Lager (M.insert a m l)
     Nothing → Lager l
  ```
- Für Inventur fehlt Möglichkeit zur **Iteration**
- Daher: Map als **Assoziativliste**

Mitmachfolie

Übung 8.2: Die Map als Assoziativliste

```
data Map α β = Map [(α, β)]
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
insert a b m = (a,b):m
```

Was ist der Nachteil dieser einfachen Implementation?

Lösung: Erzeugt ein Speicherleck — überschriebene Elemente bleiben in der Liste. Besser: beim Einfügen alte Elemente entfernen

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
insert a b xs = (a, b): filter ((a ≠ ).fst) xs
```

Nicht sehr effizient. Besser: `Map` als **sortierte** Liste.

Map als sortierte Assoziativliste

```
data Map α β = Map { toList :: [(α, β)] }
```

- Invariante: Liste ist in der ersten Komponente aufsteigend sortiert
- `lookup` ist vordefiniert; beim einfügen auch überschreiben;

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
insert a v (Map s) = Map (insert' s) where
  insert' []          = [(a, v)]
  insert' s0@((b, w):s) | a >  b = (b, w): insert' s
                        | a == b = (a, v): s
                        | a <  b = (a, v): s0
```

- ... 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:

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

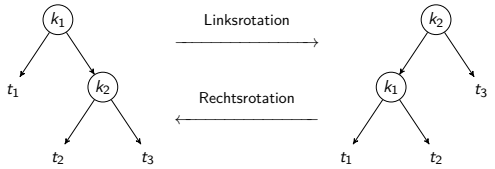
w — **Gewichtung** (Parameter des Algorithmus)

Implementation

- Balanciertheit ist **Invariante**
- Nach Einfügen oder Löschen: Balanciertheit wiederherstellen
- Dabei drei Fälle:
  1. Linker Unterbaum größer $size(l) > w \cdot size(r)$
  2. Rechter Unterbaum größer $size(r) > w \cdots size(l)$
  3. Keiner größer — Baum balanciert

Balanciertheit durch Einfache Rotation

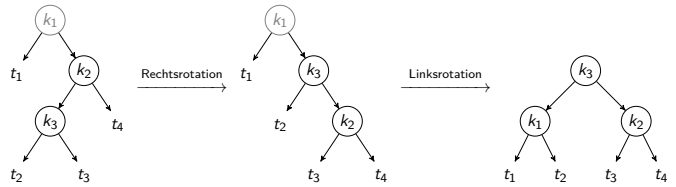
- Sei der rechte Unterbaum größer
- Zwei Unterfälle:
  1. Linkes Enkelkind t_2 größer
  2. Rechtes Enkelkind t_3 größer
- Einfache **Linksrotation** heilt (2)
- Ansonsten: **Doppelrotation** reduziert (1) zu (2)



Balanciertheit durch Doppelrotation

Falls linkes Enkelkind um Faktor α größer als rechtes:

- Nach einer einfachen Rechtsrotation des Unterbaumes ist rechtes Enkelkind größer
- Danach Linksrotation des gesamten Baumes



Implementation in Haskell

- Der Datentyp

```
data Map α β = Empty
             | Node α β Int (Map α β) (Map α β)
             deriving Eq
```

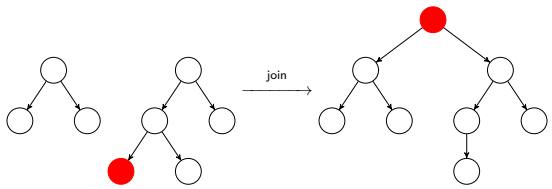
- Parameter:
  - `weight` Gewichtsfaktor w (für Einfachrotation)
  - `ratio` Gewichtsfaktor α (für Doppelrotation)
- Hilfskonstruktor `node`, setzt Größe (`l`, `r` balanciert)
- Selektor `size` für Größe des Baumes (0 für `Empty`)

Hauptfunktion

- `balance k x l r` konstruiert balancierten Baum
  - `l`, `r` sind balanciert und höchstens um einen Knoten unbalanciert
  - Vier Fälle:
    1. Beide Bäume zusammen höchstens einen Knoten $\longrightarrow$ keine Rotation
    2. $w \cdot size(l) < size(r)$: $\longrightarrow$ Linksrotation
    3. $size(l) > w \cdot size(r)$: $\longrightarrow$ Rechtsrotation
    4. Ansonsten: keine Rotation
- `balanceL k x l r` rotiert nach links. Sei r_l und r_r rechter und linker Unterbaum von `r`:
  1. $size(r_l) < \alpha \cdot size(r_r)$, dann einfache Linksrotation
  2. $size(r_l) \geq \alpha \cdot size(r_r)$ dann Doppelrotation (Rechtsrotation `r`, dann Linksrotation)

Hilfsfunktion `join` beim Löschen

- Zwei balancierte Bäume zusammenfügen (nachdem Wurzel gelöscht wurde)
- Linkester Knoten des rechten Unterbaumes wird neue Wurzel
- Mit `balance` wieder ausbalancieren



Was zum Selbermachen

Übung 8.3: Use the Source, Luke!

Ladet euch von der Webseite der Veranstaltung die Quellen für die 8. Vorlesung herunter, und öffnet die Datei `MapTree.hs`.
Vergleicht die Haskell-Implementation mit den Beschreibung der Folien.

Welche der Funktionen `lookup`, `insert`, `delete` könnte man als `fold` realisieren?

Lösung: `lookup` läßt sich falten:

```
lookup' k = fold (λak ax l r → if k == ak then Just ax
                               else maybe r Just l) Nothing
```

Ist aber nicht so effizient (linear statt logarithmisch), weil es immer erst links, dann rechts sucht.

Zusammenfassung Balancierte Bäume

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

Benchmarking: Setup

- Wie **schnell** sind die Implementationen **wirklich**?
- Benchmarking: nicht trivial
  - Verzögerte Auswertung und optimierender Compiler
  - Messen wir das **richtige**?
  - Benchmarking-Tool: Criterion
- Setup: `Map Int String` mit 50000 zufälligen Einträgen erzeugen
- Darin:
  - Einmal zufällig lesen (`lookup`), schreiben (`insert`), löschen (`delete`)
  - Sequenz aus fünfmal löschen und schreiben, zweihundertmal lesen (mixed)

Benchmarking: Resultate

	`create`	`lookup`	`insert`	`delete`	mixed
`MapFun`	333,3 ms	1,634 ms	11,27 ns	11,20 ns	1,659 ms
	13,58 ms	52,25 μs	130,8 ps	120,3 ps	79,22 μs
`MapList`	5,629 s	32,70 μs	96,12 μs	101,4 μs	6,182 ms
	168,7 ms	9,625 μs	1,294 μs	18,47 μs	2,059 μs
`MapTree`	383,9 ms	404,1 ns	119,4 μs	117,1 μs	2,803 ms
	19,62 ms	135,3 ns	13,18 μs	42,82 μs	521,5 μs
`Data.Map.Lazy`	473,0 ms	221,6 ns	104,7 μs	112,7 μs	2,396 ms
	44,97 ms	59,58 ns	49,66 μs	11,39 μs	278,8 μs

Einträge: durchschnittl. Ausführungszeit, Standardabweichung

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**

Zusammenfassung

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