Praktische Informatik 3: Funktionale Programmierung
Vorlesung 8 vom 04.12.2012: Abstrakte Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Fahrplan

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

Inhalt

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

Refakturierung im Einkaufsparadies

Printed by Christoph LÃ¼th

Nov 13, 12 14:02 **Shoppe3.hs** Page 1/3

```
module Shoppe3 where

import Data.Maybe

-- Modellierung der Artikel.

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

apreis :: Apfel -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

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

kpreis :: Kaese -> Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
         deriving (Eq, Show)

data Artikel =
   Apfel Apfel  | Eier
 | Kaese Kaese  | Schinken
 | Salami       | Milch Bio
 deriving (Eq, Show)

data Menge = Stueck Int | Gramm  Int | Liter  Double
           deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel -> Menge-> Preis
preis (Apfel a) (Stueck n) = Just (n* apreis a)
preis Eier (Stueck n)    = Just (n* 20)
preis (Kaese k)(Gramm g) = Just (round(fromIntegral g* 1000* kpreis k))
preis Schinken (Gramm g) = Just (div (g* 199) 100)
preis Salami (Gramm g)   = Just (div (g* 159) 100)
preis (Milch bio) (Liter l) =
   Just (round (l* case bio of Bio -> 119; Konv -> 69))
preis _ _ = Nothing

cent :: Artikel-> Menge-> Int
cent a m = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler!


-- Addition von Mengen
addiere :: Menge-> Menge-> Menge
addiere (Stueck i) (Stueck j)= Stueck (i+ j)
addiere (Gramm g) (Gramm h)  = Gramm (g+ h)
addiere (Liter l) (Liter m)  = Liter (l+ m)
addiere m n = error ("addiere: "++ show m++ " und "++ show n)

-- Lagerhaltung:
type Lager = [(Artikel, Menge)]

suche :: Artikel-> Lager-> Maybe Menge
suche art ((lart,  m): l)
  | art == lart = Just m
  | otherwise   = suche art l
suche art []    = Nothing

einlagern :: Artikel-> Menge-> Lager-> Lager
einlagern a m l =
  let hinein a m [] = [(a, m)]
      hinein a m ((al, ml):l)
```

Nov 13, 12 14:02 **Shoppe3.hs** Page 2/3

```
        | a == al   = (a, addiere m ml): l
        | otherwise = (al,ml): hinein a m l
  in case preis a m of
       Nothing -> l
       _ -> hinein a m l

type Einkaufswagen = [(Artikel, Menge)]

einkauf :: Artikel-> Menge-> Einkaufswagen-> Einkaufswagen
einkauf a m e
  | isJust (preis a m) = (a, m): e
  | otherwise          = e

kasse :: Einkaufswagen-> Int
kasse [] = 0
kasse ((a, m): e) = cent a m+ kasse e

kassenbon :: Einkaufswagen-> String
kassenbon ew =
  "Bob's Aulde Grocery Shoppe\n\n"++
  "Artikel        Menge     Preis\n"++
  "-----------------------------------\n"++
  artikel ew ++
  "===================================\n"++
  "Summe:"++ formatR 31 (showEuro (kasse ew))

artikel :: Einkaufswagen-> String
artikel [] = ""
artikel ((a, m): e) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent a m)) ++ "\n"++
      artikel e

menge :: Menge-> String
menge (Stueck n) = show n++ " St"
menge (Gramm g)  = show g++ " g."
menge (Liter l)  = show l++ " l."

formatL :: Int-> String-> String
formatL n str = take n (str++ replicate n ' ')

formatR :: Int-> String-> String
formatR n str =
  take n (replicate (n- length str) ' '++ str)

showEuro :: Int-> String
showEuro i =
  show (div i 100) ++ "."++
  show (mod (div i 10) 10) ++
  show (mod i 10)++ " EU"

inventur :: Lager-> Int
inventur [] = 0
inventur ((a, m): l) = cent a m+ inventur l

invFold :: Lager-> Int
invFold = foldr (\(a, m) r-> cent a m+ r) 0

{- Examples: -}

w1= einkauf (Apfel Boskoop) (Stueck 3) []
w2= einkauf Schinken (Gramm 50) w1
w3= einkauf (Milch Bio) (Liter 1) w2
w4= einkauf Schinken (Gramm 50) w3

l1= einlagern (Apfel Boskoop) (Stueck 1) []
l2= einlagern Schinken (Gramm 50) l1
l3= einlagern (Milch Bio) (Liter 6) l2
```

Warum Modularisierung?

- Übersichtlichkeit der Module

  Lesbarkeit (*human consumption*)

- Getrennte Übersetzung

  technische Handhabbarkeit

- Verkapselung

  konzeptionelle Handhabung, Invarianten

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 werden nur über die bereitgestellten Operationen beobachtet
- Einhaltung von Invarianten über dem Typ kann garantiert werden

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

- Module
- Objekte

ADTs vs. algebraische Datentypen

- Alg. Datentypen
  - Frei erzeugt
  - Vordefinierte Invarianten
  - Insbes. keine Gleichheiten
- 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 $\hat{=}$ Übersetzungseinheit (getrennte Übersetzung)

Module: Syntax

- Syntax:

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

- Bezeichner können leer sein (dann wird alles exportiert)
- Bezeichner sind:
  - Typen: T, T($c_1, \ldots, c_n$), T(..)
  - Klassen: C, C($f_1, \ldots, f_n$), C(..)
  - Andere Bezeichner: Werte, Felder, Klassenmethoden
  - Importierte Module: **module** M
- Typsynonyme und Klasseninstanzen bleiben sichtbar
- Module können rekursiv sein *(don't try at home)*

Beispiel: Das Lager

- Export als abstrakter Datentyp:

```
module Lager(Lager, leer, suche, einlagern) where
```

  - Typ Lager extern sichtbar
  - Konstruktoren versteckt

- Export als konkreter Datentyp:

```
module Lager(Lager(..), leer, suche, einlagern) wher
```

  - Konstruktoren von Lager extern sichtbar
  - Pattern Matching ist möglich
  - Erzeugung von inkonsistentem Lager möglich

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 A(x,y) where ...
```

Import(e)	Bekannte Bezeichner
import A	x, y, A.x, A.y
import A()	*(nothing)*
import A(x)	x, A.x
import qualified A	A.x, A.y
import qualified A()	*(nothing)*
import qualified A(x)	A.x
import A hiding ()	x, y, A.x, A.y
import A hiding (x)	y, A.y
import qualified A hiding ()	A.x, A.y
import qualified A hiding (x)	A.y
import A as B	x, y, B.x, B.y
import A as B(x)	x, B.x
import qualified A as B	B.x, B.y

Quelle: Haskell98-Report, Sect. 5.3.4

Schnittstelle vs. Implementation

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

Endliche Abbildungen

- Eine Sichtweise: Ersatz für Hashtables in imperativen Sprachen. Sehr nützlich!
- Abstrakter Datentyp für endliche Abbildungen:
  - Datentyp

```
data Map α β
```

  - Leere Abbildung:

```
empty :: Map α β
```

  - Abbildung auslesen:

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

  - Abbildung ändern:

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

  - Abbildung löschen:

```
delete :: Eq 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

Endliche Abbildungen: Anwendungsbeispiel

- Artikel im Lager:

```
newtype Lager = Lager (M.Map Artikel Menge)
```

- Artikel

```
suche art (Lager m) = M.lookup art m
```

- Ins Lager hinzufügen:

```
einlagern a m (Lager l) =
  case preis a m of
    Nothing → Lager l
    _ → let m' = maybe m (addiere m) (M.lookup a l)
        in  Lager (M.insert a m' l)
```

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

Bewertung

- Map als Assoziativliste bietet
  - Instanzen von Eq und Show
  - Iteration (fold)
  - ... 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:

```
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
```

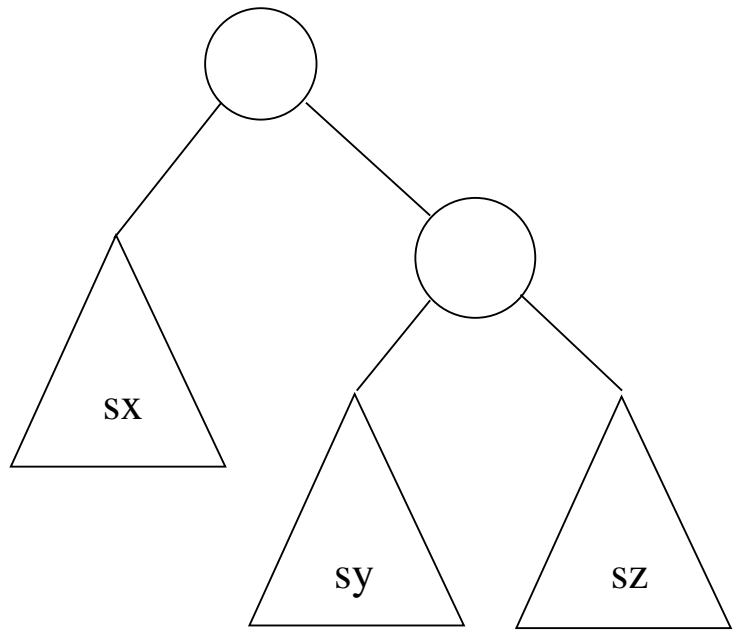
Balance sicherstellen

- Problem:

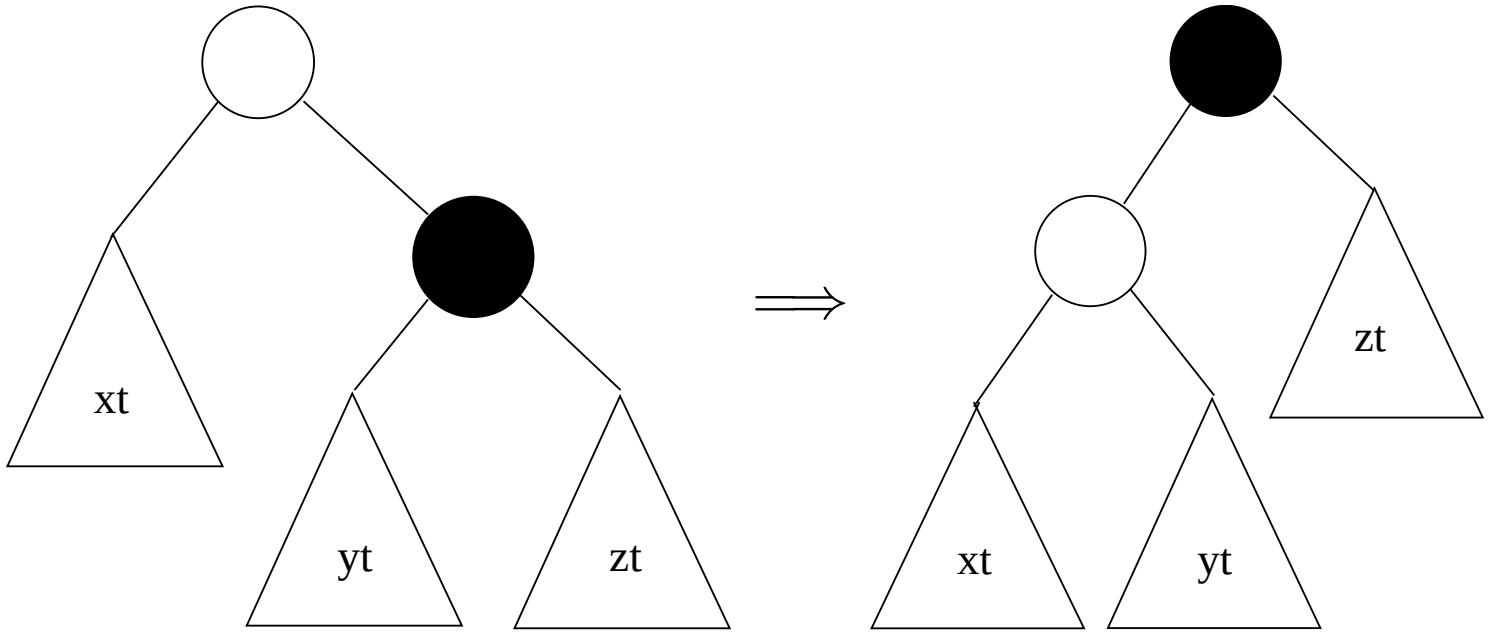
  Nach Löschen oder Einfügen zu großes Ungewicht

- Lösung:

  Rotieren der Unterbäume

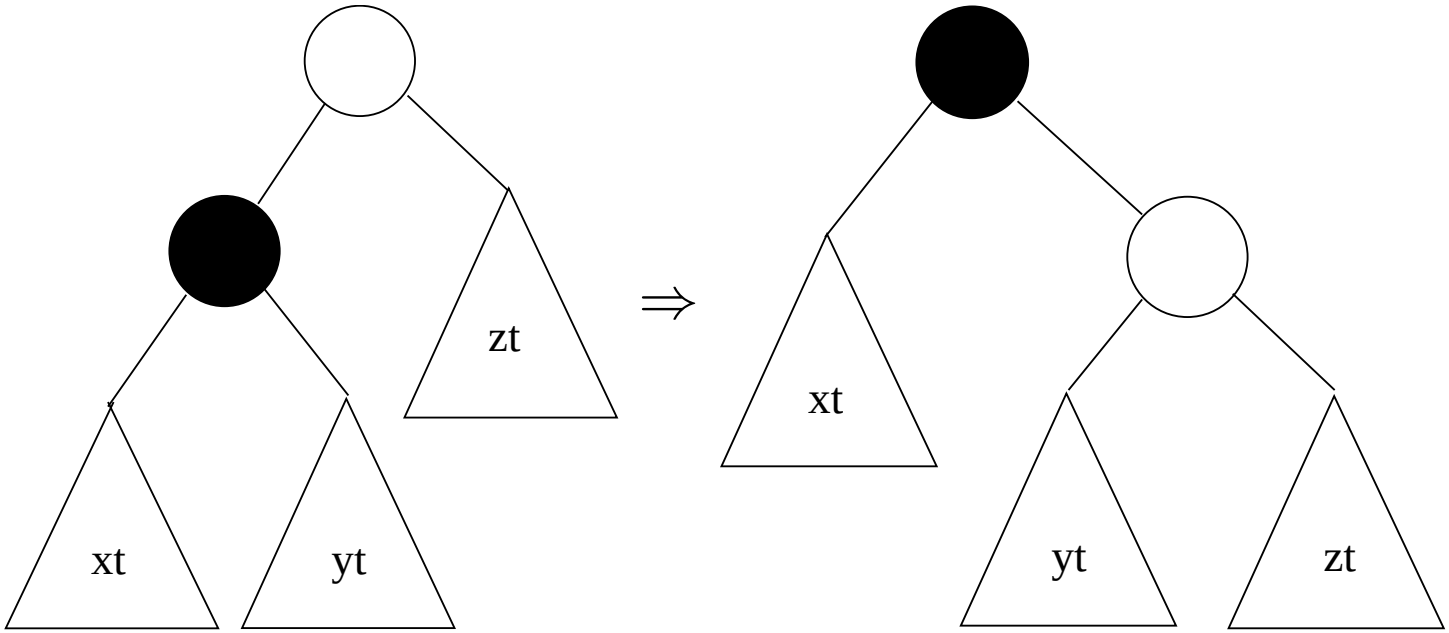


Linksrotation




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

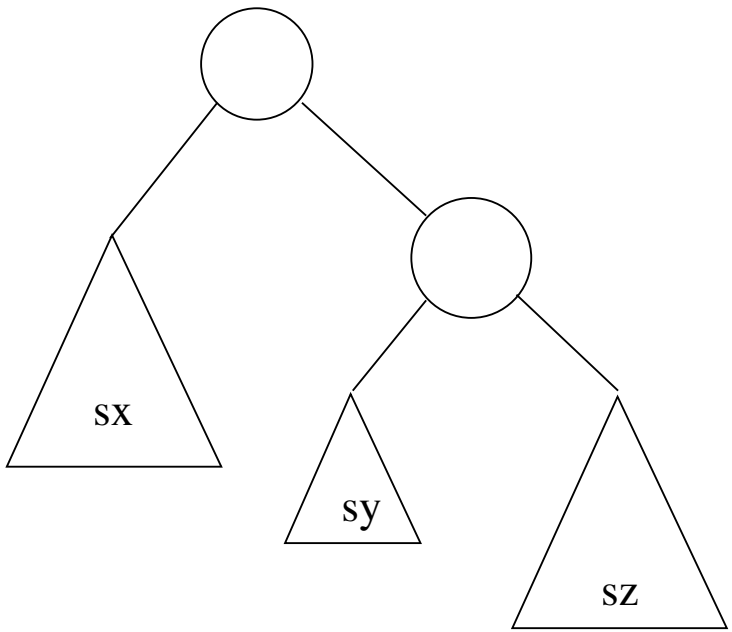
Rechtsrotation




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

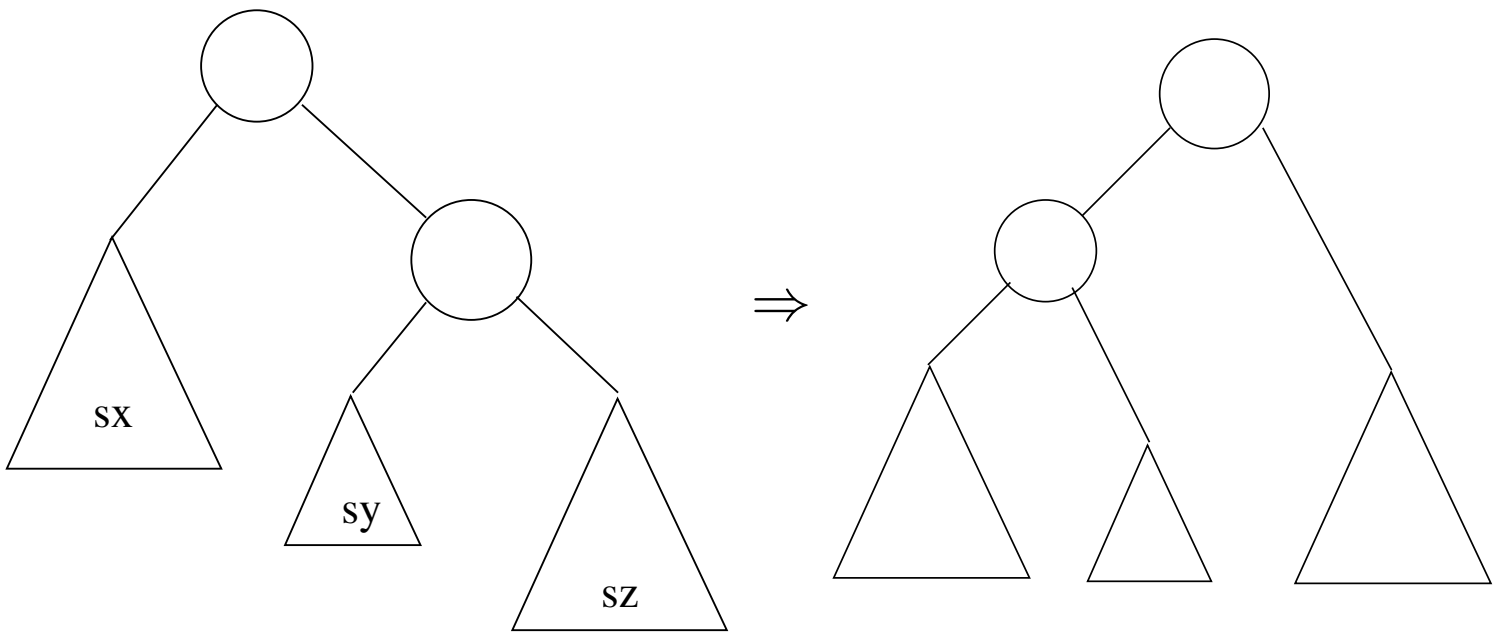
Balanciertheit sicherstellen

- Fall 1: Äußerer Unterbaum zu groß



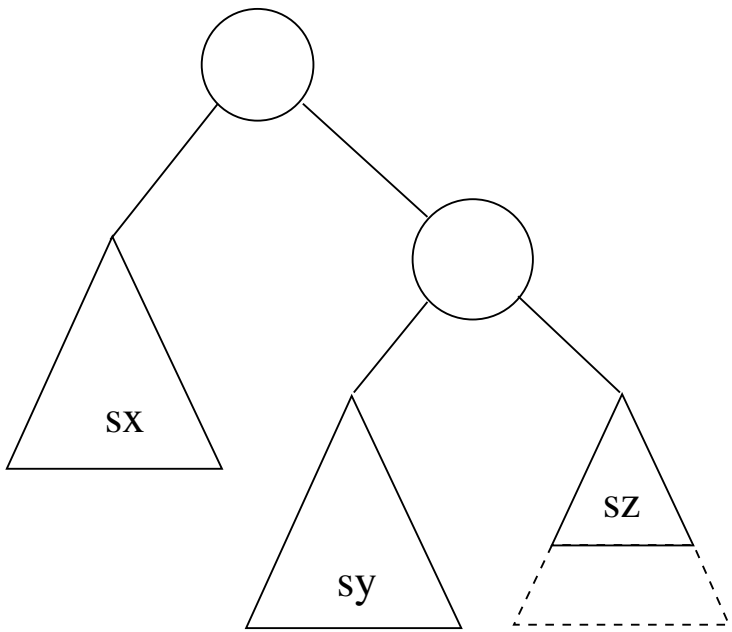
Balanciertheit sicherstellen

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



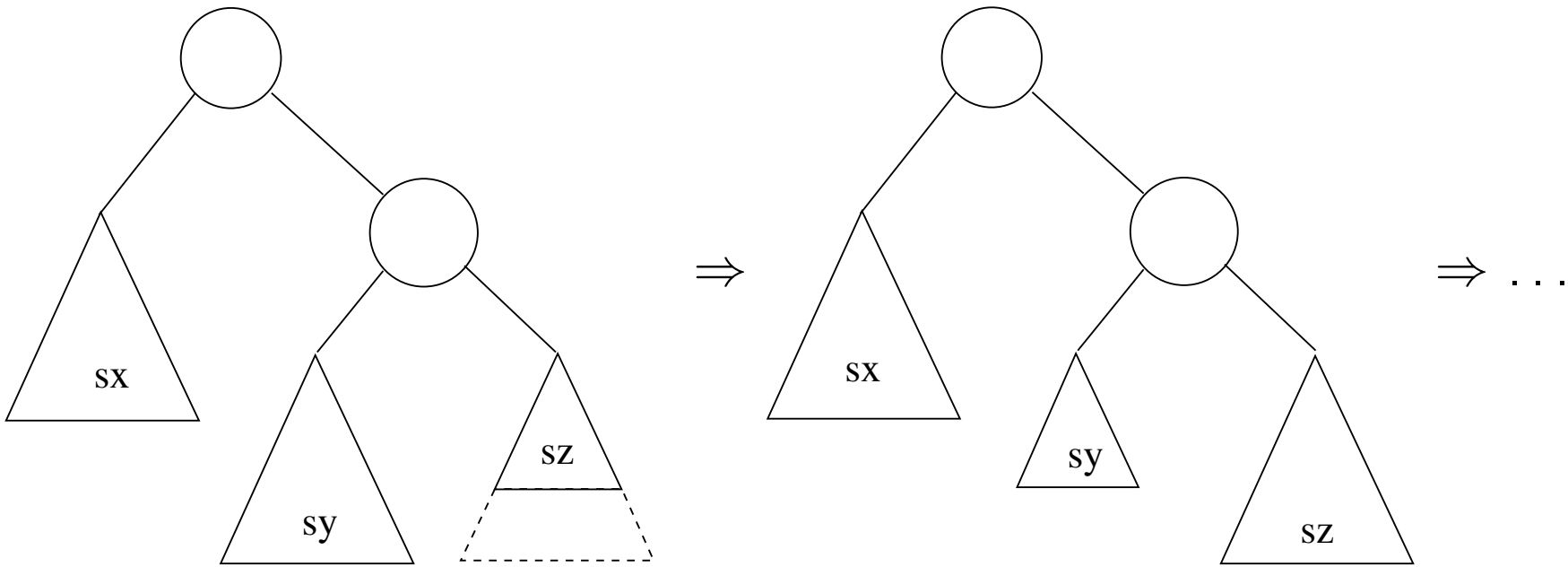
Balanciertheit sicherstellen

- Fall 2: Innerer Unterbaum zu groß oder gleich groß



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

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

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: `packagtes` 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