Vorlesung vom 29.11.2004: Algebraische Datentypen

Inhalt

- Letzte VL: Funktionsabtraktion durch Funktionen höherer Ordnung.
- Heute: Datenabstraktion durch algebraische Datentypen.
- Einfache Datentypen: Aufzählungen und Produkte
- Der allgemeine Fall
- Bekannte Datentypen: `Maybe`, Bäume
- Geordnete Bäume
- Abgeleitete Klasseninstanzen

Was ist Datenabstraktion?

- Typsynonyme sind keine Datenabstraktion
  - `type Dayname = Int` wird textuell expandiert.
  - Keinerlei Typsicherheit:
    - `Freitag + 15` ist kein Wochentag;
    - `Freitag * Montag` ist kein Typfehler.
  - Kodierung `0 = Montag` ist willkürlich und nicht eindeutig.
- Deshalb:
  - Wochentage sind nur Montag, Dienstag, . . . , Sonntag.
  - Alle Wochentage sind unterschiedlich.
- $\Longrightarrow$ Einfachster algebraischer Datentyp: Aufzählungstyp.

Aufzählungen

- Beispiel: Wochentage

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- Konstruktoren sind Konstanten:

```
Mon :: Weekday, Tue :: Weekday, Wed :: Weekday, ...
```

  - Konstruktoren werden nicht deklariert.

- Funktionsdefinition durch pattern matching:

```
isWeekend :: Weekday -> Bool
isWeekend Sat = True
isWeekend Sun = True
isWeekend _   = False
```

Produkte

- Beispiel: Datum besteht aus Tag, Monat, Jahr:

```
data Date  = Date Day Month Year
type Day   = Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
type Year  = Int
```

- Beispielwerte:

```
today     = Date 29 Nov 2004
bloomsday = Date 16 Jun 1904
fstday    = Date 1 Jan 1
```

- Konstruktor:

```
Date :: Day-> Month-> Year-> Date
```

Funktionsdefinition für Produkte

- Über pattern matching Zugriff auf Argumente der Konstruktoren:

```
day  :: Date-> Day
year :: Date-> Year
day  (Date d m y) = d
year (Date d m y) = y
```

- `day`, `year` sind Selektoren:

```
day  today     = 29
year bloomsday = 1904
```

Alternativen

Beispiel: Eine geometrische Figur ist

- ein Kreis, gegeben durch Mittelpunkt und Durchmesser,
- oder ein Rechteck, gegeben durch zwei Eckpunkte,
- oder ein Polygon, gegeben durch Liste von Eckpunkten.

```
type Point = (Double, Double)
data Shape = Circ Point Int
           | Rect Point Point
           | Poly [Point]
```

- Ein Konstruktor pro Variante.

```
Circ :: Point-> Int-> Shape
Rect :: Point-> Point-> Shape
Poly :: [Point] -> Shape
```

Funktionen auf Geometrischen Figuren

- Funktionsdefinition durch pattern matching.
  - Eine Gleichung pro Konstruktor
- Beispiel: Anzahl Eckpunkte

```
corners :: Shape-> Int
corners (Circ _ _)        = 0
corners (Rect _ _)        = 4
corners (Poly ps)         = length ps
```

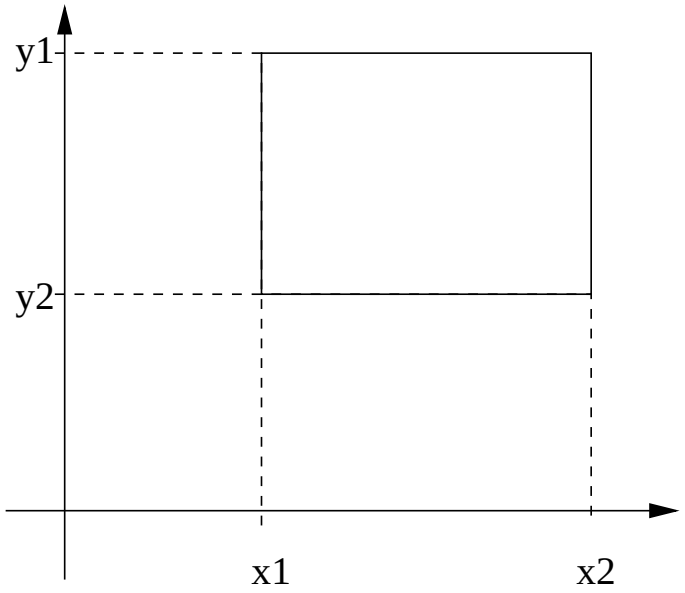
- Beispiel: Translation um einen Punkt (Vektor):

```
move :: Shape-> Point-> Shape
move (Circ m d) p   = Circ (add p m) d
move (Rect c1 c2) p = Rect (add p c1) (add p c2)
move (Poly ps) p    = Poly (map (add p) ps)
```

Beispiel: Flächenberechnung

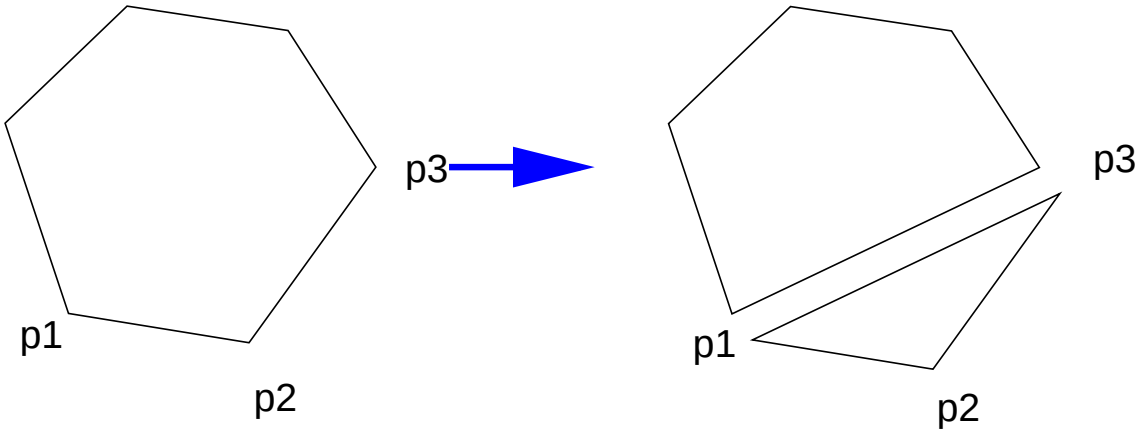
- Einfach für Kreis und Rechteck.

```
area :: Shape-> Double
area (Circ _ d) = pi* (fromIntegral d)
area (Rect (x1, y1) (x2, y2)) =
       abs ((x2- x1)* (y2- y1))
```



Beispiel: Flächenberechnung

- Fläche für Polygone:
  - Vereinfachende Annahme: Polygone konvex
  - Reduktion auf einfacheres Problem und Rekursion:




```
area (Poly ps) | length ps < 3  = 0
area (Poly (p1:p2:p3:ps)) =
           triArea p1 p2 p3 +
           area (Poly (p1:p3:ps))
```

Beispiel: Flächenberechnung

- Fläche für Dreieck mit Seitenlängen a, b, c (Heron):

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{mit} \quad s = \frac{1}{2}(a+b+c)$$

```
triArea :: Point-> Point-> Point-> Double
triArea p1 p2 p3 =
  let s= 0.5*(a+ b+ c)
      a= dist p1 p2
      b= dist p2 p3
      c= dist p3 p1
  in  sqrt (s*(s- a)*(s- b)*(s- c))
```

- Distanz zwischen zwei Punkten (Pythagoras):

```
dist :: Point-> Point-> Double
dist (x1, y1) (x2, y2) = sqrt((x1-x2)^2+ (y2- y1)^2)
```

Der allgemeine Fall

Definition von T:

$$\begin{aligned}\texttt{data T} = \;& \texttt{C}_1\ \texttt{t}_{1,1} \ldots \texttt{t}_{1,k_1} \\ & \ldots \\ \mid \;& \texttt{C}_n\ \texttt{t}_{n,1} \ldots \texttt{t}_{n,k_n}\end{aligned}$$

- Konstruktoren (und Typen) werden groß geschrieben.
- Konstruktoren $\texttt{C}_1, \ldots, \texttt{C}_n$ sind disjunkt:

$$\texttt{C}_i\ \texttt{x}_1 \ldots \texttt{x}_n = \texttt{C}_j\ \texttt{y}_1 \ldots \texttt{y}_m \Rightarrow i = j$$

- Konstruktoren sind injektiv:

$$\texttt{C}\ \texttt{x}_1 \ldots \texttt{x}_n = \texttt{C}\ \texttt{y}_1 \ldots \texttt{y}_n \Rightarrow \texttt{x}_i = \texttt{y}_i$$

- Konstruktoren erzeugen den Datentyp:

$$\forall x \in T.\, x = \texttt{C}_i\ \texttt{y}_1 \ldots \texttt{y}_m$$

Rekursive Datentypen

Der definierte Typ `T` kann rechts benutzt werden.

- Beispiel: einfache arithmetische Ausdrücke sind
  - Zahlen (Literale), oder
  - Variablen (Zeichenketten), oder
  - Addition zweier Ausdrücke, oder
  - Multiplikation zweier Ausdrücke.

```
data Expr = Lit Int
          | Var String
          | Add Expr Expr
          | Mult Expr Expr
```

- Funktion darauf meist auch rekursiv.

- Ausdruck auswerten (gegeben Auswertung der Variablen):

```
eval :: (String-> Int)-> Expr-> Int
eval f (Lit n) = n
eval f (Var x) = f x
eval f (Add e1 e2)  = eval f e1+ eval f e2
eval f (Mult e1 e2) = eval f e1* eval f e2
```

- Ausdruck ausgeben:

```
print :: Expr-> String
print (Lit n) = show n
print (Var x) = x
print (Add e1 e2)  = "("++ print e1++ "+"++ print e2++ ")"
print (Mult e1 e2) = "("++ print e1++ "*"++ print e2++ ")"
```

- Testen.

- Primitive Rekursion auf Ausdrücken
  - Eine Funktion für Rekursionsverankerung
  - Eine Funktion für Variablen
  - Eine binäre Funktion für Addition
  - Eine binäre Funktion für Multiplikation

```
foldE :: (Int-> a)-> (String-> a)
         -> (a-> a-> a)-> (a-> a-> a)-> Expr-> a
foldE b v a m (Lit n)      = b n
foldE b v a m (Var x)      = v x
foldE b v a m (Add e1 e2) = a (foldE b v a m e1)
                              (foldE b v a m e2)
foldE b v a m (Mult e1 e2) = m (foldE b v a m e1)
                               (foldE b v a m e2)
```

- Damit Auswertung und Ausgabe:

```
eval' f  = foldE id f (+) (*)
print'   =
  foldE show id
        (\s1 s2-> "("++ s1++ "+"++ s2++ ")")
        (\s1 s2-> "("++ s1++ "*"++ s2++ ")")
```

Polymorphe Datentypen

- Algebraische Datentypen parametrisiert über Typen:

```
data Pair a = Pair a a
```

- Paar: zwei beliebige Elemente gleichen Typs. Zeigen.
- Elemente vertauschen:

```
twist :: Pair a-> Pair a
twist (Pair a b) = Pair b a
```

- Map für Paare:

```
mapP :: (a-> b)-> Pair a-> Pair b
mapP f (Pair a b)= Pair (f a) (f b)
```

Polymorph und rekursiv: Listen selbstgemacht

Eine Liste von `a` ist

- entweder leer
- oder ein Kopf `a` und eine Restliste `List a`

  ```
  data List a = Mt | Cons a (List a)
  ```

- Syntaktischer Zucker der vordefinierten Listen:

  ```
  data [a] = [] | a : [a]
  ```

  - Nicht benutzerdefinierbar: Typ `[a]`, Konstruktor `[]`
  - Operatoren als binäre Konstruktoren möglich.

Funktionen auf Listen

- Funktionsdefinition:

```
fold :: (a-> b-> b)-> b-> List a-> b
fold f e Mt           = e
fold f e (Cons a as) = f a (fold f e as)
```

- Mit `fold` alle primitiv rekursiven Funktionen, wie:

```
map' f    = fold (Cons . f) Mt
length'   = fold ((+).(const 1)) 0
filter' p = fold (\x-> if p x then Cons x else id) Mt
```

Modellierung von Fehlern: `Maybe a`

- Typ a plus Fehlerelement
  - Im Prelude vordefiniert.

```
data Maybe a = Just a | Nothing
```

- `Nothing` modelliert Fehlerfall zurückgegeben. Bsp:

```
find :: (a-> Bool)-> [a]-> Maybe a
find p []     = Nothing
find p (x:xs) = if p x then Just x
                else find p xs
```

Funktionen auf `Maybe a`

- Anwendung von Funktion mit Default-Wert für Fehler (vordefiniert):

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe d f Nothing  = d
maybe d f (Just x) = f x
```

- Liften von Funktionen ohne Fehlerbehandlung:
  - Fehler bleiben erhalten.

```
fmap :: (a-> b)-> Maybe a-> Maybe b
fmap f Nothing = Nothing
fmap f (Just x)= Just (f x)
```

Binäre Bäume

Ein binärer Baum ist

- Entweder leer,
- oder ein Knoten mit genau zwei Unterbäumen.

Knoten tragen eine Markierung.

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

- Andere Möglichkeit: Markierungen für Knoten und Blätter:

```
data Tree' a b = Null'
               | Leaf' b
               | Node' (Tree' a b) a (Tree' a b)
```

Funktionen auf Bäumen

- Test auf Enthaltensein:

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

Funktionen auf Bäumen

Primitive Rekursion auf Bäumen:

- Rekursionsanfang
- Rekursionsschritt:
  - Label des Knotens,
  - Zwei Rückgabewerte für linken und rechten Unterbaum.

```
foldT :: (a-> b-> b-> b)-> b-> Tree a-> b
foldT f e Null = e
foldT f e (Node l a r) = f a (foldT f e l) (foldT f e r)
```

- Damit Elementtest:

```
member' t x =
   foldT (\e b1 b2-> e == x || b1 || b2) False t
```

- Testen.

Funktionen auf Bäumen

- Traversion: preorder, inorder, postorder

```
preorder  :: Tree a-> [a]
inorder   :: Tree a-> [a]
postorder :: Tree a-> [a]
preorder   = foldT (\x t1 t2-> [x]++ t1++ t2) []
inorder    = foldT (\x t1 t2-> t1++ [x]++ t2) []
postorder  = foldT (\x t1 t2-> t1++ t2++ [x]) []
```

  - Äquivalente Definition ohne `foldT`:

```
preorder' Null           = []
preorder' (Node l a r)   = [a] ++preorder' l ++preorder' r
```

- Testen.

Geordnete Bäume

- Voraussetzung:
  - Ordnung auf `a` (`Ord a`)
  - Es soll für alle Bäume gelten:

$$\forall \texttt{Tree a l r}.\ \texttt{member x l} \Rightarrow \texttt{x} < \texttt{a} \wedge \texttt{member x r} \Rightarrow \texttt{a} < \texttt{x}$$

- Test auf Enthaltensein vereinfacht:

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

Geordnete Bäume

- Ordnungserhaltendes Einfügen

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

- Problem: Erzeugung ungeordneter Bäume möglich.
  - Lösung erfordert Verstecken der Konstrukturen — nächste VL.

Geordnete Bäume

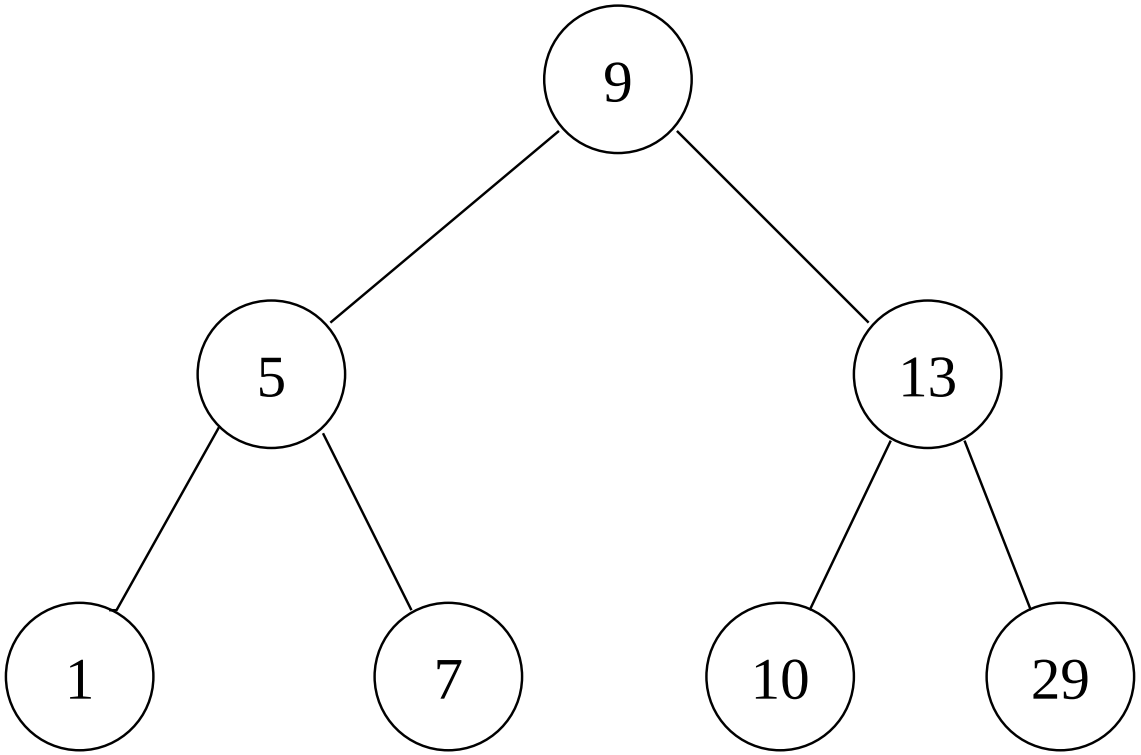
- Löschen:

```
delete :: Ord a=> Tree a-> a-> Tree a
delete Null _ = Null
delete (Node l y r) x
   | x < y  = Node (delete l x) y r
   | x == y = join l r
   | x > y  = Node l y (delete r x)
```

- `join` fügt zwei Bäume ordnungserhaltend zusammen.

Geordnete Bäume

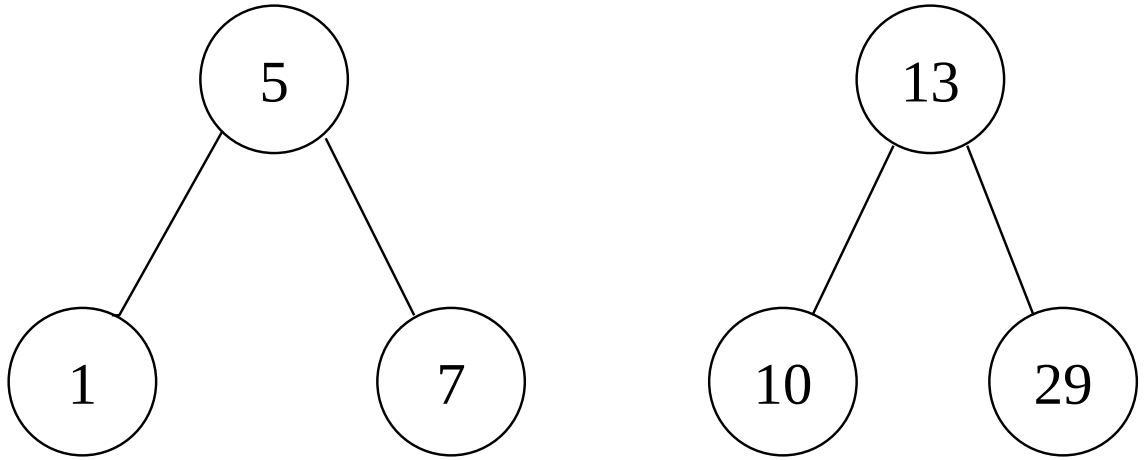
- Beispiel: Gegeben folgender Baum, dann `delete t 9`



  - Wurzel wird gelöscht.

Geordnete Bäume

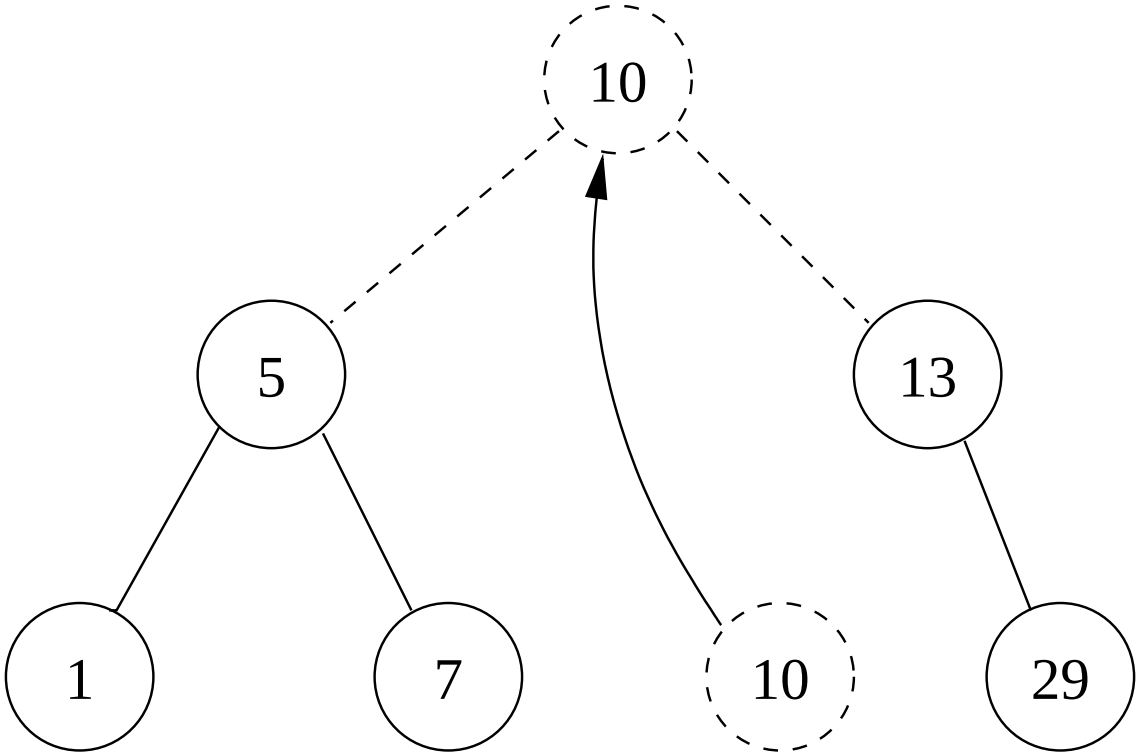
- Wurzel wurde gelöscht:



  - `join` fügt Bäume ordnungserhaltend zusammen.

Geordnete Bäume

- `join` fügt zwei Bäume ordnungserhaltend zusammen.



  - Wurzel des neuen Baums: Knoten links unten im rechten Teilbaum (oder Knoten rechts unten im linken Teilbaum)

Geordnete Bäume

- Implementation:
  - `splitTree` spaltet Baum in Knoten links unten und Rest.
  - Testen.

```
join ::  Tree a-> Tree a-> Tree a
join xt Null = xt
join xt yt   = Node xt u nu where
  (u, nu) = splitTree yt
  splitTree :: Tree a-> (a, Tree a)
  splitTree (Node Null a t) = (a, t)
  splitTree (Node lt a rt) =
    (u, Node nu a rt) where
       (u, nu) = splitTree lt
```

Abgeleitete Klasseninstanzen

- Wie würde man Gleichheit auf `Shape` definieren?

```
Circ p1 i1 == Circ p2 i2 = p1 == p2 && i1 == i2
Rect p1 q1 == Rect p2 q2 = p1 == p2 && q1 == q2
Poly ps    == Poly qs    = ps == qs
_          == _          = False
```

- Schematisierbar:
  - Gleiche Konstruktoren mit gleichen Argumente gleich,
  - alles andere ungleich.
- Automatisch generiert durch `deriving Eq`
- Ähnlich `deriving (Ord, Show, Read)`

Zusammenfassung

- Algebraische Datentypen erlauben Datenabstraktion durch
  - Trennung zwischen Repräsentation und Semantik und
  - Typsicherheit.
- Algebraischen Datentypen sind frei erzeugt.
- Bekannte algebraische Datentypen:
  - Aufzählungen, Produkte, Varianten;
  - `Maybe a`, Listen, Bäume
- Für geordnete Bäume:
  Verstecken von Konstruktoren nötig — nächste Vorlesung.