

Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 05.12.2006: Algebraische Datentypen

Christoph Lüth

WS 06/07



Inhalt

- Letzte VL: Funktionsabstraktion 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**.
- 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 5 Dec 2006
```

```
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      = 5
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 **Dreieck**, gegeben durch **drei Eckpunkte**,
- oder ein **Polygon**, gegeben durch **Liste von Eckpunkten**.

Alternativen

Beispiel: Eine **geometrische Figur** ist

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

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

- Ein **Konstruktor** pro **Variante**.

Funktionen auf geometrischen Figuren

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

```
corners :: Shape -> Int
corners (Circ _ _)      = 0
corners (Tri _ _ _)     = 3
corners (Rect _ _)      = 4
corners (Poly ps)       = length ps
```

Funktionen auf geometrischen Figuren

- Funktionsdefinition durch **pattern matching**.
 - Eine **Gleichung** pro **Konstruktor**
- 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 (Tri p1 p2 p3) p  = Tri (add p p1) (add p p2)
                        (add p p3)
move (Poly ps) p       = Poly (map (add p) ps)
```

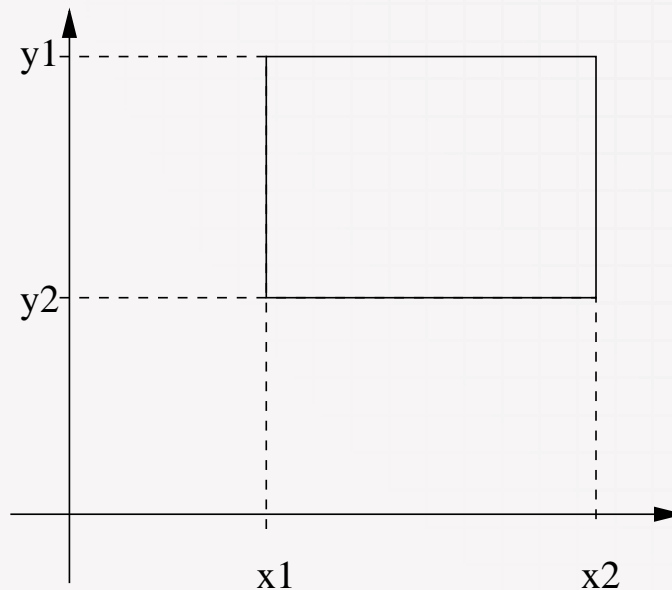
Geometrische Figuren: Flächenberechnung

- Einfach für Kreis und Rechteck.

```
area :: Shape -> Double
```

```
area (Circ _ d) = pi * (d/2)^2
```

```
area (Rect (x1, y1) (x2, y2)) =  
    abs ((x2 - x1) * (y2 - y1))
```



Geometrische Figuren: Flächenberechnung

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

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

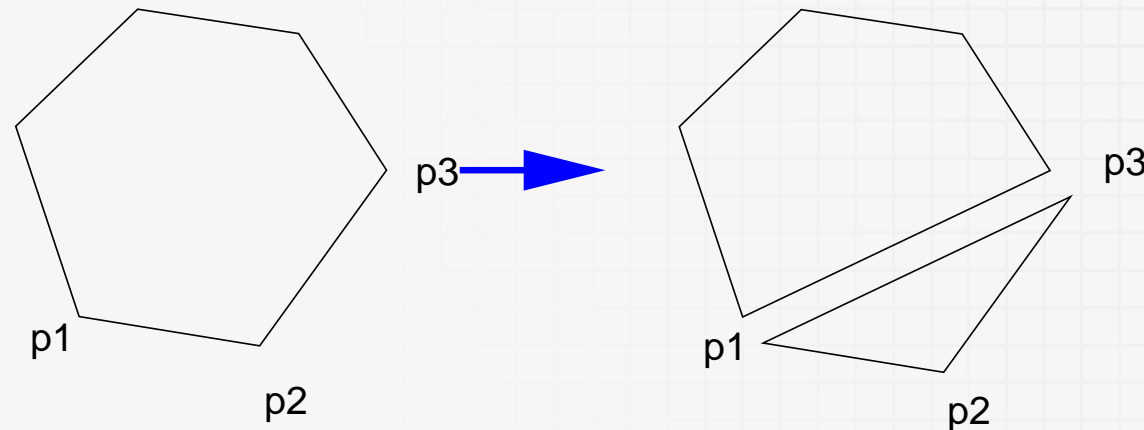
```
area (Tri p1 p2 p3) =  
  let s= 0.5*(a+ b+ c)  
      a= dist p1 p2  
      b= dist p2 p3  
      c= dist p3 p1  
  in  sqrt (s*(s- a)*(s- b)*(s- c))
```

- **Distanz** zwischen **zwei Punkten** (Pythagoras):

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

Geometrische Figuren: 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)) =  
    area (Tri p1 p2 p3) +  
    area (Poly (p1:p3:ps))
```

Der allgemeine Fall

Definition von T :

$$\begin{array}{l} \text{data } T = C_1 t_{1,1} \dots t_{1,k_1} \\ \quad \dots \\ \quad | C_n t_{n,1} \dots t_{n,k_n} \end{array}$$

- **Konstruktoren** (und Typen) werden groß geschrieben.
- **Konstruktoren** C_1, \dots, C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \Rightarrow i = j$$

- **Konstruktoren** sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_n \Rightarrow x_i = y_i$$

- **Konstruktoren erzeugen** den Datentyp:

$$\forall x \in T. x = C_i y_1 \dots y_m$$

Rekursive Datentypen

Der **definierte Typ** `T` kann **rechts** benutzt werden.

- **Beispiel:** einfache arithmetische **Ausdrücke** sind
 - Zahlen (Literele), 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**.

Funktionen auf Ausdrücken

- 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
```

- Testen.

Funktionen auf Ausdrücken

- 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**

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)
```

Primitive Rekursion auf Ausdrücken

- Damit Auswertung und Ausgabe:

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

- Testen.

Polymorphe Datentypen

- Algebraische Datentypen parametrisiert über **Typen**:

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

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

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:

```
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)
```

- **Höhe**:

```
height :: Tree a -> Int
height Null = 0
height (Node l a r) = max (height l) (height r) + 1
```

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)
```

Funktionen auf Bäumen

- Damit **Elementtest**:

```
member' :: Eq a => Tree a -> a -> Bool
member' t x =
    foldT (\e b1 b2 -> e == x || b1 || b2) False t
```

- **Höhe**:

```
height' :: Tree a -> Int
height' = foldT (\ _ h1 h2 -> 1 + max h1 h2) 0
```

- Testen.

Funktionen auf Bäumen

- Traversal: **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.

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
```

```
Tri  p1 q1 r1 == Tri p2 q2 r2 = p1 == p2 && q1 == q2  
                                     && r1 == r2
```

```
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.