Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 17.11.2010: Typvariablen und Polymorphie

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung
  - Typinferenz
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- Letzte Vorlesung: rekursive Datentypen
- Diese Vorlesung:
  - Abstraktion über Typen: Typvariablen und Polymorphie

Zeichenketten und Listen von Zahlen

- Letzte VL: Eine Zeichenkette ist
  - entweder leer (das leere Wort ϵ)
  - oder ein Zeichen und eine weitere Zeichenkette

```
data MyString = Empty
              | Cons Char MyString
```

Zeichenketten und Listen von Zahlen

- Letzte VL: Eine Zeichenkette ist
  - entweder leer (das leere Wort ϵ)
  - oder ein Zeichen und eine weitere Zeichenkette

```
data MyString = Empty
              | Cons Char MyString
```

- Eine Liste von Zahlen ist
  - entweder leer
  - oder eine Zahl und eine weitere Liste

```
data IntList = Empty
             | Cons Int IntList
```

Zeichenketten und Listen von Zahlen

- Letzte VL: Eine Zeichenkette ist
  - entweder leer (das leere Wort ϵ)
  - oder ein Zeichen und eine weitere Zeichenkette

```
data MyString = Empty
              | Cons Char MyString
```

- Eine Liste von Zahlen ist
  - entweder leer
  - oder eine Zahl und eine weitere Liste

```
data IntList = Empty
             | Cons Int IntList
```

- Strukturell gleiche Definition

$\rightsquigarrow$ Zwei Instanzen einer allgemeineren Definition.

Typvariablen

- Typvariablen abstrahieren über Typen

```
data List α = Empty
            | Cons α (List α)
```

- α ist eine Typvariable
- α kann mit Char oder Int instantiiert werden
- List α ist ein polymorpher Datentyp
- Typvariable α wird bei Anwendung instantiiert
- Signatur der Konstruktoren

```
Empty :: List α
Cons  :: α → List α → List α
```

Polymorphe Datentypen

- Typkorrekte Terme: Typ

  Empty

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
Empty	List α

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int

Polymorphe Datentypen

- Typkorrekte Terme: Typ

Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	List Char

Polymorphe Datentypen

- Typkorrekte Terme: | Typ

Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	List Char
Cons True Empty	

Polymorphe Datentypen

- Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	List Char
Cons True Empty	List Bool

- Nicht typ-korrekt:

  Cons 'a' (Cons 0 Empty)

  Cons True (Cons 'x' Empty)

  wegen Signatur des Konstruktors:

```
Cons :: α → List α → List α
```

Polymorphe Funktionen

- Verkettung von MyString:

```
cat :: MyString → MyString → MyString
cat Empty t        = t
cat (Cons c s) t = Cons c (cat s t)
```

Polymorphe Funktionen

- Verkettung von MyString:

```
cat :: MyString → MyString → MyString
cat Empty t        = t
cat (Cons c s) t = Cons c (cat s t)
```

- Verkettung von IntList:

```
cat :: IntList → IntList → IntList
cat Empty t        = t
cat (Cons c s) t = Cons c (cat s t)
```

Polymorphe Funktionen

- Verkettung von MyString:

```
cat :: MyString → MyString → MyString
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

- Verkettung von IntList:

```
cat :: IntList → IntList → IntList
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

- Gleiche Definition, unterschiedlicher Typ

  $\rightsquigarrow$ Zwei Instanzen einer allgemeineren Definition.

Polymorphe Funktionen

- Polymorphie auch für Funktionen:

```
cat :: List α → List α → List α
cat Empty ys          = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

- Typvariable α wird bei Anwendung instantiiert:

```
cat (Cons 3 Empty) (Cons 5 (Cons 57 Empty))
cat (Cons 'p' (Cons 'i' Empty)) (Cons '3' Empty)
```

aber nicht

```
cat (Cons True Empty) (Cons 'a' (Cons 0 Empty))
```

- Typvariable: vergleichbar mit Funktionsparameter

Polymorphe Datentypen: Bäume

Datentyp:

```
data BTree α = MtBTree
             | BNode α (BTree α) (BTree α)
```

Höhe des Baumes:

```
height :: BTree α→ Int
height MtBTree = 0
height (BNode j l r) = max (height l) (height r)+ 1
```

Traversion — erzeugt Liste aus Baum:

```
inorder :: BTree α→ List α
inorder MtBTree = Empty
inorder (BNode j l r) =
  cat (inorder l) (Cons j (inorder r))
```

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α → β → Pair α β
```

- Beispielterm Typ

  Pair 4 'x'

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α → β → Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α → β → Pair α β
```

- Beispielterm — Typ

  Pair 4 'x' — Pair Int Char

  Pair (Cons True Empty) 'a'

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char
Pair (3+ 4) (Cons 'a' Empty)	

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char
Pair (3+ 4) (Cons 'a' Empty)	Pair Int (List Char)

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char
Pair (3+ 4) (Cons 'a' Empty)	Pair Int (List Char)
Cons (Pair 7 'x') Empty	

Tupel

- Mehr als eine Typvariable möglich
- Beispiel: Tupel (kartesisches Produkt, Paare)

```
data Pair α b = Pair α b
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char
Pair (3+ 4) (Cons 'a' Empty)	Pair Int (List Char)
Cons (Pair 7 'x') Empty	List (Pair Int Char)

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter syntaktischer Zucker
- Tupel sind das kartesisches Produkt

```
data (α, β) = (α, β)
```

  - (a, b) = alle Kombinationen von Werten aus a und b
  - Auch n-Tupel: (a,b,c) etc.

- Listen

```
data [α] = [] | α : [α]
```

  - Weitere Abkürzungen: [x]= x:[], [x,y] = x:y:[] etc.

Übersicht: vordefinierte Funktionen auf Listen I

```
(++)      :: [α] → [α] → [α]        -- Verketten
(!!)      :: [α] → Int → α          -- n-tes Element selektieren
concat    :: [[α]] → [α]            -- "flachklopfen"
length    :: [α] → Int              -- Länge
head, last :: [α] → α               -- Erstes/letztes Element
tail, init :: [α] → [α]             -- Hinterer/vorderer Rest
replicate :: Int → α → [α]          -- Erzeuge n Kopien
take      :: Int → [α] → [α]        -- Erste n Elemente
drop      :: Int → [α] → [α]        -- Rest nach n Elementen
splitAt   :: Int → [α] → ([α], [α]) -- Spaltet an Index n
reverse   :: [α] → [α]              -- Dreht Liste um
zip       :: [α] → [β] → [(α, β)]   -- Erzeugt Liste v. Paaren
unzip     :: [(α, β)] → ([α], [β])  -- Spaltet Liste v. Paaren
and, or   :: [Bool] → Bool          -- Konjunktion/Disjunktion
sum       :: [Int] → Int            -- Summe (überladen)
product   :: [Int] → Int            -- Produkt (überladen)
```

Zeichenketten: String

- String sind Listen von Zeichen:

```
type String = [Char]
```

- Alle vordefinierten Funktionen auf Listen verfügbar.
- Syntaktischer Zucker zur Eingabe:

```
['y','o','h','o'] == "yoho"
```

- Beispiel:

```
cnt :: Char → String → Int
cnt c []     = 0
cnt c (x:xs) = if (c== x) then 1+ cnt c xs
               else cnt c xs
```

Variadische Bäume

- Variable Anzahl Kinderknoten: Liste von Kinderknoten

```
data VTree α = MtVTree
             | VNode α [VTree α]
```

- Anzahl Knoten zählen:

```
count :: VTree α→ Int
count MtVTree = 0
count (VNode _ ns) = 1+ count_nodes ns
```

```
count_nodes :: [VTree α] → Int
count_nodes []     = 0
count_nodes (t:ts) = count t+ count_nodes ts
```

Berechnungsmuster für Listen

- Primitiv rekursive Definitionen:
  - Eine Gleichung für leere Liste
  - Eine Gleichung für nicht-leere Liste, rekursiver Aufruf
- Komprehensionsschema:
  - Jedes Element der Eingabeliste
  - wird getestet
  - und gegebenfalls transformiert

Listenkomprehension

- Ein einfaches Beispiel: Zeichenkette in Kleinbuchstaben wandeln

```
toL :: String → String
toL s = [ toLower c | c ← s ]
```

- Buchstaben herausfiltern:

```
letters :: String → String
letters s = [ c | c ← s, isAlpha c ]
```

- Kombination: alle Buchstaben kanonisch kleingeschrieben

```
toLL :: String → String
toLL s = [ toLower c | c← s, isAlpha c ]
```

Listenkomprehension

- Allgemeine Form:

```
[ E c | c ← L, test c]
```

  - Ergebnis: E c für alle Werte c in L, so dass test c wahr ist
  - Typen: L :: $[\alpha]$, c :: α, test :: $\alpha \rightarrow$ Bool, E :: $\alpha \rightarrow \beta$, Ergebnis $[\beta]$

- Auch mehrere Generatoren und Tests möglich:

```
[E c1 ... cn | c1 ← L1, test1 c1,
               c2 ← L2 c1, test2 c1 c2, ...]
```

  - E vom Typ $\alpha_1 \rightarrow \alpha_2 \ldots \rightarrow \beta$

Variadische Bäume II

- Die Zähl-Funktion vereinfacht:

```
count' :: VTree α→ Int
count' MtVTree = 0
count' (VNode _ ts) =
  1+ sum [count' t | t← ts]
```

- Die Höhe:

```
height' :: VTree α→ Int
height' MtVTree = 0
height' (VNode _ ts) =
  1+ maximum (0: [height' t | t← ts])
```

Beispiel: Permutation von Listen

```
perms :: [α] → [[α]]
```

- Permutation der leeren Liste
- Permutation von x:xs
- x an allen Stellen in alle Permutationen von xs eingefügt.

```
perms [] = [[]] --- Wichtig!
perms (x:xs) = [ ps ++ [x] ++ qs
               | rs ← perms xs,
                 (ps, qs) ← splits rs ]
```

- Dabei splits: alle möglichen Aufspaltungen

```
splits :: [α] → [([α], [α])]
splits []     = [([],[])]
splits (y:ys) = ([], y:ys) :
                [(y:ps, qs) | (ps, qs) ← splits ys ]
```

Beispiel: Quicksort

- Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- sortiere Teilstücke,
- konkateniere Ergebnisse.

Beispiel: Quicksort

- Zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
- sortiere Teilstücke,
- konkateniere Ergebnisse.

```
qsort :: [α] → [α]
```

```
qsort [] = []
qsort (x:xs) =
  qsort smaller ++ x:equals ++ qsort larger where
    smaller = [ y | y ← xs, y < x ]
    equals  = [ y | y ← xs, y == x ]
    larger  = [ y | y ← xs, y > x ]
```

Überladung und Polymorphie

- Fehler: qsort nur für Datentypen mit Vergleichsfunktion
- Überladung: Funktion f :: a→ b existiert für einige, aber nicht für alle Typen
- Beispiel:
  - Gleichheit: (==) :: a→ a→ Bool
  - Vergleich: (<) :: a→ a→ Bool
  - Anzeige: show :: a→ String
- Lösung: Typklassen
  - Typklasse Eq für (==)
  - Typklasse Ord für (<) (und andere Vergleiche)
  - Typklasse Show für show
- Auch Ad-hoc Polymorphie (im Ggs. zur parametrischen Polymorpie)

Typklassen in polymorphen Funktionen

- qsort, korrekte Signatur:

```
qsort :: Ord α ⇒ [α] → [α]
```

- Element einer Liste (vordefiniert):

```
elem :: Eq α ⇒ α → [α] → Bool
elem e []     = False
elem e (x:xs) = e == x || elem e xs
```

- Liste ordnen und anzeigen:

```
showsorted :: (Eq α, Show α) ⇒ [α] → String
showsorted x = show (qsort x)
```

Polymorphie in anderen Programmiersprachen: Java

- Polymorphie in Java: Methode auf alle Subklassen anwendbar
  - Manuelle Typkonversion nötig, fehleranfällig
- Neu ab Java 1.5: Generics
  - Damit parametrische Polymorphie möglich

```
class AbsList<T> {
  public AbsList(T el, AbsList<T> tl) {
    this.elem= el;
    this.next= tl;
  }
  public T elem;
  public AbsList<T> next;
```

Polymorphie in anderen Programmiersprachen: Java

Typkorrekte Konkatenenation:

```
void concat(AbsList<T> o)
{
  AbsList<T> cur= this;
  while (cur.next ≠ null) cur= cur.next;
  cur.next= o;
}
```

Nachteil: Benutzung umständlich, weil keine Typherleitung

```
AbsList<Integer> l=
  new AbsList<Integer>(new Integer(1),
    new AbsList<Integer>(new Integer(2), null));
```

Polymorphie in anderen Programmiersprachen: C

- "Polymorphie" in C: void *

```
struct list {
    void        *head;
    struct list *tail;
    }
```

- Gegeben:

```
int x = 7;
struct list s = { &x, NULL };
```

- s.head hat Typ void *:

```
int y;
y= *(int *)s.head;
```

- Nicht möglich: head direkt als Skalar (e.g. int)
- C++: Templates

Zusammenfassung

- Typvariablen und (parametrische) Polymorphie: Abstraktion über Typen
- Vordefinierte Typen: Listen [a] und Tupel (a,b)
- Berechungsmuster über Listen: primitive Rekursion, Listenkomprehension
- Überladung durch Typklassen
- Nächste Woche: Funktionen höherer Ordnung