# Praktische Informatik 3: Einführung in die Funktionale Programmierung
# Vorlesung vom 10.11.2010: Rekursive Datentypen

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

- Rekursive Datentypen
  - Formen der Rekursion
  - Rekursive Definition
  - Rekursive Datentypen in anderen Sprachen
- Induktiver Beweis
  - Schluss vom kleineren aufs größere

# Der Allgemeine Fall: Algebraische Datentypen

Definition eines algebraischen Datentypen T:

$$\begin{aligned} \text{data } T = \quad & C_1\ t_{1,1} \ldots t_{1,k_1} \\ & \ldots \\ \mid \quad & C_n\ t_{n,1} \ldots t_{n,k_n} \end{aligned}$$

- Konstruktoren $C_1, \ldots, C_n$ sind disjunkt:
$$C_i\ x_1 \ldots x_n = C_j\ y_1 \ldots y_m \longrightarrow i = j$$
- Konstruktoren sind injektiv:
$$C\ x_1 \ldots x_n = C\ y_1 \ldots y_n \longrightarrow x_i = y_i$$
- Konstruktoren erzeugen den Datentyp:
$$\forall x \in T.\, x = C_i\ y_1 \ldots y_m$$

Diese Eigenschaften machen Fallunterscheidung möglich.

# Rekursive Datentypen

- Der definierte Typ T kann rechts benutzt werden.
- Rekursive Datentypen sind unendlich
- Entspricht induktiver Definition

# Induktive Definitionen

- Beispiel natürliche Zahlen: Peano-Axiome
  - $0 \in \mathbb{N}$
  - wenn $n \in \mathbb{N}$, dann $S\ n \in \mathbb{N}$
  - $S$ injektiv und $S\ n \neq 0$
  - Induktionsprinzip: $\phi(0)$, $\phi(x) \longrightarrow \phi(S\ x)$, dann $\forall n \in \mathbb{N}.\phi(n)$
- Induktionsprinzip erlaubt Definition rekursiver Funktionen:

$$\begin{aligned} n + 0 &= n \\ n + S\ m &= S(n + m) \end{aligned}$$

# Natürliche Zahlen in Haskell

- Direkte Übersetzung der Peano-Axiome
- Der Datentyp:

```
data Nat = Zero
         | S Nat
```

- Rekursive Funktionsdefinition:

```
add :: Nat → Nat → Nat
add n Zero  = n
add n (S m) = S (add n m)
```

# Beispiel: Zeichenketten selbstgemacht

- Eine Zeichenkette ist
  - entweder leer (das leere Wort $\epsilon$)
  - oder ein Zeichen *c* und eine weitere Zeichenkette *xs*

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

- Lineare Rekursion
  - Genau ein rekursiver Aufruf

# Rekursive Definition

- Typisches Muster: Fallunterscheidung
  - Ein Fall pro Konstruktor
- Hier:
  - Leere Zeichenkette
  - Nichtleere Zeichenkette

# Funktionen auf Zeichenketten

- Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

# Funktionen auf Zeichenketten

- Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

- Verkettung:

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

# Funktionen auf Zeichenketten

- Länge:

```
len :: MyString → Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

- Verkettung:

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

- Umkehrung:

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

# Baumartige Rekursion: Binäre Bäume

- Datentyp:

```
data BTree = MtBTree
           | BNode Int BTree BTree
```

- Funktion, bsp. Höhe:

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

- Baumartige Rekursion
  - Doppelter rekursiver Aufruf

# Wechselseitige Rekursion: Variadische Bäume

- Variable Anzahl Kinderknoten

```
data VTree = MtVTree
           | VNode String VNodes

data VNodes = MtVNodes
            | VMore VTree VNodes
```

- VNodes: Liste von Kinderbäumen

# Wechselseitige Rekursion: Variadische Bäume

- Hauptfunktion:

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

- Hilfsfunktion:

```
count_nodes :: VNodes → Int
count_nodes MtVNodes = 0
count_nodes (VMore t ns)= count t+ count_nodes ns
```

# Rekursive Typen in anderen Sprachen

- Standard ML: gleich
- Lisp: keine Typen, aber alles ist eine S-Expression

```
data SExpr = Quote Atom | Cons SExpr SExpr
```

- Python, Ruby:
  - Listen (Sequenzen) vordefiniert
  - Keine anderen Typen

# Rekursive Typen in Java

- Nachbildung durch Klassen, z.B. für Listen:

```
class List {
  public List(Object el, List tl) {
    this.elem= el;
    this.next= tl;
  }
  public Object elem;
  public List   next;
```

- Länge (iterativ):

```
  int length() {
    int i= 0;
    for (List cur= this; cur ≠ null; cur= cur.next)
      i++;
    return i;
  }
```

# Rekursive Typen in C

- C: Produkte, Aufzählungen, keine rekursiven Typen
- Rekursion durch Zeiger

```
typedef struct list_t {
      void           *elem;
      struct list_t  *next;
      } *list;
```

- Konstruktoren nutzerimplementiert

```
list cons(void *hd, list tl)
{ list l;
  if ((l= (list)malloc(sizeof(struct list_t)))== NUL
    printf("Out␣of␣memory\n"); exit(-1);
  }
  l→ elem= hd; l→ next= tl;
  return l;
}
```

# Rekursive Definition, induktiver Beweis

- Definition durch Rekursion
  - Basisfall (leere Zeichenkette)
  - Rekursion (nicht-leere Zeichenkette)

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

  - Reduktion der Eingabe (vom größeren aufs kleinere)
- Beweis durch Induktion
  - Schluß vom kleineren aufs größere

# Beweis durch vollständige Induktion

Zu zeigen:

Für alle natürlichen Zahlen $x$ gilt $P(x)$.

Beweis:

- Induktionsbasis: $P(0)$
- Induktionsschritt:
  - Induktionsvoraussetzung $P(x)$
  - zu zeigen $P(x+1)$

# Beweis durch strukturelle Induktion (Zeichenketten)

Zu zeigen:

Für alle (endlichen) Zeichenketten $xs$ gilt $P(xs)$

Beweis:

- Induktionsbasis: $P(\epsilon)$
- Induktionsschritt:
  - Induktionsvoraussetzung $P(xs)$
  - zu zeigen $P(x\ xs)$

# Beweis durch strukturelle Induktion (Allgemein)

Zu zeigen:

$$\text{Für alle } x \text{ in } T \text{ gilt } P(x)$$

Beweis:

- Für jeden Konstruktor $C_i$:
  - Voraussetzung: für alle $t_{i,j}$ gilt $P(t_{i,j})$
  - zu zeigen $P(C_i\ t_{i,1}\ \ldots\ t_{i,k_i})$

# Beispielbeweise

$$\text{len } s \geq 0 \tag{1}$$

$$\text{len } (\text{cat } s\ t) = \text{len } s + \text{len } t \tag{2}$$

$$\text{len } (\text{rev } s) = \text{len } s \tag{3}$$

# Spezifikation und Korrektheit

- Die ersten $n$ Zeichen einer Zeichenkette ($n \geq 0$)

```
takeN :: Int → MyString → MyString
takeN n Empty = Empty
takeN n (Cons c s) = if n == 0 then Empty
                     else Cons c (takeN (n−1) s)
```

# Spezifikation und Korrektheit

- Die ersten $n$ Zeichen einer Zeichenkette ($n \geq 0$)

```
takeN :: Int → MyString → MyString
takeN n Empty = Empty
takeN n (Cons c s) = if n == 0 then Empty
                     else Cons c (takeN (n-1) s)
```

- Zeichenkette ohne die ersten $n$ Zeichen ($n \geq 0$)

```
dropN :: Int → MyString → MyString
dropN n Empty       = Empty
dropN n (Cons c s) = if n == 0 then Cons c s
                     else dropN (n- 1) s
```

# Spezifikation und Korrektheit

- Die ersten $n$ Zeichen einer Zeichenkette ($n \geq 0$)

```
takeN :: Int → MyString → MyString
takeN n Empty = Empty
takeN n (Cons c s) = if n == 0 then Empty
                     else Cons c (takeN (n−1) s)
```

- Zeichenkette ohne die ersten $n$ Zeichen ($n \geq 0$)

```
dropN :: Int → MyString → MyString
dropN n Empty       = Empty
dropN n (Cons c s) = if n == 0 then Cons c s
                     else dropN (n− 1) s
```

- Eigenschaften:

$$\text{len}\ (\text{takeN}\ n\ s) \leq n \tag{4}$$

$$\text{len}\ (\text{dropN}\ n\ s) \geq \text{len}\ s - n \tag{5}$$

$$\text{cat}\ (\text{takeN}\ n\ s)\ (\text{dropN}\ n\ s) = s \tag{6}$$

# Zusammenfassung

- Datentypen können rekursiv sein
- Rekursive Datentypen sind unendlich (induktiv)
- Funktionen werden rekursiv definiert
  - Formen der Rekursion: linear, baumartig, wechselseitig
- Rekursive Definition ermöglicht induktiven Beweis
- Nächste Woche: Abstraktion über Typen (Polymorphie)