

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

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Rev. 1152

1 [23]

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

2 [23]

Inhalt

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

3 [23]

Der Allgemeine Fall: Algebraische Datentypen

Definition eines algebraischen Datentypen T:

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

- ▶ Konstruktoren C_1, \dots, C_n sind disjunkt:
 $C_i x_1 \dots x_n = C_j y_1 \dots y_m \implies i = j$
 - ▶ Konstruktoren sind injektiv:
 $C x_1 \dots x_n = C y_1 \dots y_n \implies x_i = y_i$
 - ▶ Konstruktoren erzeugen den Datentyp:
 $\forall x \in T. x = C_i y_1 \dots y_m$
- Diese Eigenschaften machen Fallunterscheidung möglich.

4 [23]

Rekursive Datentypen

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

5 [23]

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: $\phi(0), \phi(x) \implies \phi(S x)$, dann $\forall n \in \mathbb{N}. \phi(n)$
- ▶ Induktionsprinzip erlaubt Definition rekursiver Funktionen:

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

6 [23]

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)
```

7 [23]

Beispiel: Zeichenketten selbstgemacht

- ▶ Eine Zeichenkette ist
 - ▶ entweder leer (das leere Wort ϵ)
 - ▶ oder ein Zeichen c und eine weitere Zeichenkette xs

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

- ▶ Lineare Rekursion
 - ▶ Genau ein rekursiver Aufruf

8 [23]

Rekursive Definition

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

9 [23]

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)
```

10 [23]

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

11 [23]

Wechselseitige Rekursion: Variadische Bäume

- ▶ Variable Anzahl Kinderknoten

```
data VTree = MtVTree
           | VNode String VNodes

data VNodes = MtVNodes
            | VMore VTree VNodes
```

- ▶ VNodes: Liste von Kinderbäumen

12 [23]

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
```

13 [23]

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

14 [23]

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;
}
```

15 [23]

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))) == NULL) {
    printf("Out of memory\n"); exit(-1);
  }
  l->elem = hd; l->next = tl;
  return l;
}
```

16 [23]

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

17 [23]

Beweis durch vollständige Induktion

Zu zeigen:

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

Beweis:

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

18 [23]

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 $P(x\ xs)$

19 [23]

Beweis durch strukturelle Induktion (Allgemein)

Zu zeigen:

Für alle x in T gilt $P(x)$

Beweis:

- Für jeden Konstruktor C_j :
 - Voraussetzung: für alle $t_{i,j}$ gilt $P(t_{i,j})$
 - zu zeigen $P(C_j\ t_{i,1}\ \dots\ t_{i,k})$

20 [23]

Beispielbeweise

$$\text{len } s \geq 0 \quad (1)$$

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

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

21 [23]

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:

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

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

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

22 [23]

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)

23 [23]