Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 21.11.2006: Rekursion als allgemeines Berechnungsmuster

Christoph Lüth

Inhalt

- Formen der Rekursion:
  - Listenkomprehension
  - Primitive Rekursion
  - Allgemeine Rekursion
- Rekursion als Berechnungsmuster
- Funktionen höherer Ordnung
  - Funktionen als gleichberechtigte Objekte
  - Funktionen als Argumente
  - `map`, `filter`, `fold` und Freunde

Primitive Rekursion

- Primitive Rekursion: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste
- Beispiel:

```
sum :: [Int]-> Int
sum []     = 0
sum (x:xs) = x+ sum xs
```

- Weitere Beispiele: `length`, `concat`, `(++)`, ...
- Auswertung:

```
sum [4,7,3]        ⇝  4 + 7 + 3 + 0
concat [A, B, C]   ⇝  A ++ B ++ C++ []
```

Primitive Rekursion

- Allgemeines Muster:

$$\begin{aligned} \texttt{f []} &= \texttt{A} \\ \texttt{f (x:xs)} &= \texttt{x} \otimes \texttt{f xs} \end{aligned}$$

- Parameter:
  - Startwert (für die leere Liste) `A :: b`
  - Rekursionsfunktion $\otimes$ `:: a -> b-> b`
- Auswertung:

$$\texttt{f}[\texttt{x1}, \ldots, \texttt{xn}] = \texttt{x1} \otimes \texttt{x2} \otimes \ldots \otimes \texttt{xn} \otimes \texttt{A}$$

Primitive Rekursion

- Allgemeines Muster:

$$\begin{aligned} \texttt{f []} &= \texttt{A} \\ \texttt{f (x:xs)} &= \texttt{x} \otimes \texttt{f xs} \end{aligned}$$

- Parameter:
  - Startwert (für die leere Liste) `A :: b`
  - Rekursionsfunktion $\otimes$ `:: a -> b-> b`
- Auswertung:

$$\texttt{f}[\texttt{x1}, \ldots, \texttt{xn}] = \texttt{x1} \otimes \texttt{x2} \otimes \ldots \otimes \texttt{xn} \otimes \texttt{A}$$

- Terminiert immer
- Entspricht einfacher Iteration (`while`-Schleife)
- Vergleiche `Iterator` und `Enumeration` in Java

Allgemeine Rekursion

- Primitive Rekursion ist Spezialfall der allgemeinen Rekursion
- Allgemeine Rekursion:
  - Rekursion über mehrere Argumente
  - Rekursion über andere Datenstruktur
  - Andere Zerlegung als Kopf und Rest

Beispiele für allgemeine Rekursion

- Rekursion über mehrere Argumente:

```
zip :: [a]-> [b]-> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

- Rekursion über ganzen Zahlen:

```
take :: Int-> [a]-> [a]
take 0 _  = []
take _ [] = []
take n (x:xs)
  | n > 0     = x: take (n-1) xs
  | otherwise = error "take: negative Argument"
```

Beispiele für allgemeine Rekursion: Sortieren

- Quicksort:
  - zerlege Liste in Elemente kleiner, gleich und größer dem ersten,
  - sortiere Teilstücke, konkateniere Ergebnisse
- Mergesort:
  - teile Liste in der Hälfte,
  - sortiere Teilstücke, füge ordnungserhaltend zusammen.

Beispiele für allgemeine Rekursion: Mergesort

```
msort :: [Int]-> [Int]
msort xs
  | length xs <= 1 = xs
  | otherwise = merge (msort front) (msort back) where
    (front, back) = splitAt ((length xs) `div` 2) xs
    merge :: [Int]-> [Int]-> [Int]
    merge [] x = x
    merge y [] = y
    merge (x:xs) (y:ys)
      | x<= y     = x:(merge xs (y:ys))
      | otherwise = y:(merge (x:xs) ys)
```

Beispiel: das n-Königinnen-Problem

- Problem: n Königinnen auf $n \times n$-Schachbrett sicher plazieren
- Spezifikation:
  - Position der Königinnen

    ```
    type Pos = (Int, Int)
    ```

  - Eingabe: Anzahl Königinnen, Rückgabe: Positionen

    ```
    queens :: Int-> [[Pos]]
    ```

Beispiel: das n-Königinnen-Problem

- Rekursive Formulierung:
  - Keine Königin— kein Problem.
  - Lösung für n Königinnen: Lösung für $n-1$ Königinnen, n-te Königin so stellen, dass sie keine andere bedroht.
- Vereinfachung: n-te Königin muß in n-ter Spalte plaziert werden.
  - Limitiert kombinatorielle Explosion

Beispiel: das n-Königinnen-Problem

- Hauptfunktion:

```
queens num = putqueens num where
  putqueens :: Int-> [[Pos]]
  putqueens n | n == 0  = [[]]
              | otherwise =
     [ p++ [(n, m)] | p <- putqueens (n-1),
                      m <- [1.. num],
                      safe p (n, m)]
```

  - `[n..m]`: Liste der Zahlen von `n` bis `m`
  - Mehrere Generatoren in Listenkomprehension.
  - Rekursion über Anzahl der Königinnen.

Beispiel: das n-Königinnen-Problem

- Sichere neue Position: durch keine andere bedroht

```
safe :: [Pos]-> Pos-> Bool
safe others nu =
   and [ not (threatens other nu)
         | other <- others ]
```

  - Verallgemeinerte Konjunktion `and :: [Bool]-> Bool`
- Gegenseitige Bedrohung:
  - Bedrohung wenn in gleicher Zeile, Spalte, oder Diagonale.

```
threatens :: Pos-> Pos-> Bool
threatens (i, j) (m, n) =
   (j== n) || (i+j == m+n) || (i-j == m-n)
```

  - Test auf gleicher Spalte `i==m` unnötig. Testen.

Funktionen Höherer Ordnung

- Grundprinzip der funktionalen Programmierung
- Funktionen als Argumente.
- Funktionen sind gleichberechtigt: Werte wie alle anderen
- Vorzüge:
  - Modellierung allgemeiner Berechungsmuster
  - Höhere Wiederverwendbarkeit
  - Größere Abstraktion

Funktionen als Argumente: `map`

- Funktion auf alle Elemente anwenden: `map`
- Signatur:

```
map :: (a-> b)-> [a]-> [b]
```

- Definition

```
map f xs = [ f x | x <- xs ]
```

- oder -

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Funktionen als Argumente: `filter`

- Elemente filtern: `filter`
- Signatur:

```
filter :: (a-> Bool)-> [a]-> [a]
```

- Definition

```
filter p xs = [ x | x <- xs, p x ]
```

- oder -

```
filter p []    = []
filter p (x:xs)
  | p x        = x:(filter p xs)
  | otherwise = filter p xs
```

Funktionen als Argumente: `foldr`

- Primitive Rekursion
  - Basisfall: leere Liste
  - Rekursionsfall: Kombination aus Listenkopf und Rekursionswert
- Signatur

```
foldr :: (a-> b-> b)-> b-> [a]-> b
```

- Definition

```
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Beispiele: `foldr`

- Beispiel: Summieren von Listenelementen.

```
sum :: [Int]-> Int
sum xs = foldr (+) 0 xs

sum [3,12] = 3 + sum [12]
           = 3+ 12+ sum []
           = 3+ 12+ 0= 15
```

Beispiele: `foldr`

- Beispiel: Summieren von Listenelementen.

```
sum :: [Int]-> Int
sum xs = foldr (+) 0 xs

sum [3,12] = 3 + sum [12]
           = 3+ 12+ sum []
           = 3+ 12+ 0= 15
```

- Beispiel: Flachklopfen von Listen.

```
concat :: [[a]]-> [a]
concat xs = foldr (++) [] xs

concat [l1,l2,l3,l4] = l1++ l2++ l3++ l4++ []
```

Listen zerlegen

- `take, drop`: n Elemente vom Anfang
- Verallgemeinerung: Längster Präfix für den Prädikat gilt

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []           =  []
takeWhile p (x:xs)
            | p x        =  x : takeWhile p xs
            | otherwise =  []
```

- Restliste des längsten Präfix

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

- Es gilt: `takeWhile p xs ++ dropWhile p xs == xs`

Listen zerlegen

- Kombination von `takeWhile` und `dropWhile`

```
span       :: (a -> Bool) -> [a] -> ([a],[a])
span p xs = (takeWhile p xs, dropWhile p xs)
```

- Ordnungserhaltendes Einfügen:

```
ins :: Int-> [Int]-> [Int]
ins x xs = lessx ++ (x: grteqx) where
  (lessx, grteqx) = span less xs
  less z = z < x
```

- Damit sortieren durch Einfügen:

```
isort :: [Int]-> [Int]
isort xs = foldr ins [] xs
```

Beliebiges Sortieren

- Wieso eigentlich immer aufsteigend?

Beliebiges Sortieren

- Wieso eigentlich immer aufsteigend?
- Ordnung als Argument `ord`
  - Totale Ordnung: transitiv, antisymmetrisch, reflexiv, total
  - Insbesondere: x `ord` $y \wedge y$ `ord` $x \Rightarrow x = y$

```
qsortBy :: (a-> a-> Bool)-> [a]-> [a]
qsortBy ord [] = []
qsortBy ord (x:xs) =
      qsortBy ord [y| y<-xs, ord y x] ++ [x] ++
      qsortBy ord [y| y<-xs, not (ord y x)]
```

- Resultat: Liste $[x_1, x_2, \ldots, x_n]$ mit `ord` x_i x_{i+1}

Noch ein Beispiel: `rev`

- Listen umdrehen:

```
rev :: [a]-> [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

- Mit `fold`:

```
rev xs = foldr snoc [] xs

snoc :: a-> [a]-> [a]
snoc x xs = xs ++ [x]
```

- Unbefriedigend: doppelte Rekursion

Funktionen als Argumente: `foldl`

- `foldr` faltet von rechts:

$$\texttt{foldr} \otimes [\texttt{x}_1, ..., \texttt{x}_n]\ A = \texttt{x}_1 \otimes (\texttt{x}_2 \otimes (\ldots (\texttt{x}_n \otimes \texttt{A})))$$

- Warum nicht andersherum?

$$\texttt{foldl} \otimes [\texttt{x}_1, ..., \texttt{x}_n]\ A = (((A \otimes \texttt{x}_1) \otimes \texttt{x}_2) \ldots) \otimes \texttt{x}_n$$

Funktionen als Argumente: `foldl`

- `foldr` faltet von rechts:

$$\texttt{foldr} \otimes [\texttt{x}_1, ..., \texttt{x}_n]\ A = \texttt{x}_1 \otimes (\texttt{x}_2 \otimes (\ldots (\texttt{x}_n \otimes \texttt{A})))$$

- Warum nicht andersherum?

$$\texttt{foldl} \otimes [\texttt{x}_1, ..., \texttt{x}_n]\ A = (((A \otimes \texttt{x}_1) \otimes \texttt{x}_2) \ldots) \otimes \texttt{x}_n$$

- Definition von `foldl`:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

Noch ein Beispiel: `rev` revisited

- Listenumkehr ist falten von links:

```
rev' xs = foldl cons [] xs
cons :: [a]-> a-> [a]
cons xs x = x: xs
```

- Nur noch einfache Rekursion
- Satz: Wenn $(\otimes, A)$ Monoid, dann

$$\texttt{foldl} \otimes A\ \texttt{xs} = \texttt{foldr} \otimes A\ \texttt{xs}$$

Zusammenfassung

- Verallgemeinerte Berechnungsmuster
  - Listenkomprehension
  - Primitive Rekursion
  - Allgemeine Rekursion
- Funktionen höherer Ordnung:
  - Funktionen als gleichberechtigte Werte
  - Erlaubt Verallgemeinerungen
  - Erhöht Flexibilität und Wiederverwendbarkeit
  - Beispiele: `map`, `filter`, `foldr`, `foldl`
  - E.g. Sortieren nach beliebiger Ordnung