

Praktische Informatik 3  
Einführung in die Funktionale Programmierung  
Vorlesung vom 19.11.2008:  
Funktionen höherer Ordnung

Christoph Lüth

WS 08/09



## Fahrplan

- Teil I: Grundlagen
  - Rekursion als Berechnungsmodell
  - Rekursive Datentypen, rekursive Funktionen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung
  - Funktionaler Entwurf, Standarddatentypen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke

## Inhalt

- Funktionen höherer Ordnung
  - Funktionen als gleichberechtigte Objekte
  - Funktionen als Argumente
  - Spezielle Funktionen: map, filter, fold und Freunde
- Formen der Rekursion:
  - Einfache und allgemeine Rekursion
- Typklassen

## Funktionen als Werte

- Rekursive Definitionen, z.B. über Listen:

```
concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ concat xs
```
- Argumente können auch Funktionen sein.
- Beispiel: Funktion zweimal anwenden

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```
- Auswertung wie vorher: `twice (twice inc) 3`  $\rightsquigarrow$  7

## Funktionen Höherer Ordnung

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

## Funktionen als Argumente: Funktionskomposition

- Funktionskomposition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```
- Vordefiniert
- Lies: f nach g
- Funktionskomposition vorwärts:

```
(>.) :: (a -> b) -> (b -> c) -> a -> c
(f >.) g x = g (f x)
```
- Nicht vordefiniert!

## Funktionen als Argumente: map

- Funktion auf alle Elemente anwenden: map
- Signatur:

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

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

```
lowercase :: String -> String
lowercase str = map toLower str
```

## Funktionen als Argumente: filter

- Elemente filtern: filter
- Signatur:

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

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

```
qsort :: [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (\y -> y < x) xs) ++
               filter (\y -> y == x) (x:xs) ++
               qsort (filter (\y -> y > x) xs)
```

## Beispiel: Primzahlen

- **Sieb des Erathostenes**
    - Für jede gefundene Primzahl  $p$  alle Vielfachen heraussieben
    - Dazu: **filtern** mit  $\lambda n \rightarrow \text{mod } n \ p \neq 0$
- ```
sieve :: [Integer]-> [Integer]
sieve [] = []
sieve (p:ps) =
  p : sieve (filter (\n-> mod n p /= 0) ps)
```
- Primzahlen im Intervall  $[1.. n]$ :

```
primes :: Integer-> [Integer]
primes n = sieve [2..n]
```
  - NB: Mit 2 anfangen!
  - Listengenerator  $[n.. m]$



## Partielle Applikation

- Funktionskonstruktor **rechtsassoziativ**:
$$a \rightarrow b \rightarrow c \equiv a \rightarrow (b \rightarrow c)$$
- **Inbesondere**:
$$(a \rightarrow b) \rightarrow c \neq a \rightarrow (b \rightarrow c)$$
- **Partielle Anwendung von Funktionen**:
  - Für  $f :: a \rightarrow b \rightarrow c$ ,  $x :: a$  ist  $f \ x :: b \rightarrow c$
- Beispiele:
  - `map toLower :: String-> String`
  - `3 == :: Int-> Bool`
  - `concat . map (replicate 2) :: String-> String`



## Die Kürzungsregel

Bei Anwendung der Funktion

$$f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$$

auf  $k$  Argumente mit  $k \leq n$

$$e_1 :: t_1, e_2 :: t_2, \dots, e_k :: t_k$$

werden die Typen der Argumente **gekürzt**:

$$f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_k \rightarrow t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t$$
$$f \ e_1 \ \dots \ e_k :: t_{k+1} \rightarrow \dots \rightarrow t_n \rightarrow t$$

- Beweis: Regel **App** (letzte VL)



## Einfache Rekursion

- **Einfache 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:
$$\begin{aligned} \text{sum } [4,7,3] &\rightsquigarrow 4 + 7 + 3 + 0 \\ \text{concat } [A, B, C] &\rightsquigarrow A ++ B ++ C ++ [] \end{aligned}$$



## Einfache Rekursion

- **Allgemeines Muster**:
$$\begin{aligned} f [] &= A \\ f (x:xs) &= x \otimes f \ xs \end{aligned}$$
- **Parameter der Definition**:
  - Startwert (für die leere Liste)  $A :: b$
  - Rekursionsfunktion  $\otimes :: a \rightarrow b \rightarrow b$
- **Auswertung**:
$$f [x_1, \dots, x_n] = x_1 \otimes x_2 \otimes \dots \otimes x_n \otimes A$$
- **Terminiert immer**
- Entspricht einfacher **Iteration** (`while`-Schleife)



## Einfach Rekursion durch foldr

- **Einfache Rekursion**
  - **Basisfall**: leere Liste
  - **Rekursionsfall**: Kombination aus Listenkopf und Rekursionswert
- **Signatur**
$$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$
- **Definition**
$$\begin{aligned} \text{foldr } f \ e \ [] &= e \\ \text{foldr } f \ e \ (x:xs) &= f \ x \ (\text{foldr } f \ e \ xs) \end{aligned}$$



## Beispiel: foldr

- Beispiel: **Summieren** von Listenelementen.

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

- Beispiel: **Flachklopfen** von Listen.

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



## Noch ein Beispiel: rev

- Listen **umdrehen**:

```
rev :: [a]-> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]
```
- Mit **fold**:
$$\text{rev } xs = \text{foldr } \text{snoc} \ [] \ xs$$
- **Unbefriedigend**: doppelte Rekursion



## Einfache Rekursion durch foldl

- `foldr` faltet von rechts:  

$$\text{foldr } \otimes [x_1, \dots, x_n] A = x_1 \otimes (x_2 \otimes (\dots (x_n \otimes A)))$$
- Warum nicht andersherum?  

$$\text{foldl } \otimes [x_1, \dots, x_n] A = (((A \otimes x_1) \otimes x_2) \dots) \otimes x_n$$
- Definition von `foldl`:  

$$\begin{aligned} \text{foldl} &:: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\ \text{foldl } f \ a \ [] &= a \\ \text{foldl } f \ a \ (x:xs) &= \text{foldl } f \ (f \ a \ x) \ xs \end{aligned}$$



## foldr vs. foldl

- $f = \text{foldr } \otimes A$  entspricht  

$$\begin{aligned} f [] &= A \\ f (x:xs) &= x \otimes f \ xs \end{aligned}$$
- Kann nicht-strikt in `xs` sein
- $f = \text{foldl } \otimes A$  entspricht  

$$\begin{aligned} f \ xs &= g \ A \ xs \\ g \ a \ [] &= a \\ g \ a \ (x:xs) &= g \ (a \otimes x) \ xs \end{aligned}$$
- Endrekursiv (effizient), aber strikt in `xs`



## Noch ein Beispiel: rev revisited

- Listenumkehr ist falten von links:  

$$\begin{aligned} \text{rev}' \ xs &= \text{foldl } \text{cons} \ [] \ xs \\ \text{cons} \ :: [a] \rightarrow a \rightarrow [a] \\ \text{cons} \ xs \ x &= x : xs \end{aligned}$$

- Nur noch eine Rekursion



## foldl = foldr

- Def:  $(\otimes, A)$  ist ein Monoid wenn  

$$\begin{aligned} A \otimes x &= x && \text{(Neutrales Element links)} \\ x \otimes A &= x && \text{(Neutrales Element rechts)} \\ (x \otimes y) \otimes z &= x \otimes (y \otimes z) && \text{(Assoziativität)} \end{aligned}$$
- Satz: Wenn  $(\otimes, A)$  Monoid, dann  

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



## Funktionen Höherer Ordnung: Java

- Java: keine direkte Syntax für Funktionen höherer Ordnung
- Folgendes ist nicht möglich:  

```
interface Collection {
    Object fold(Object f(Object a, Collection c),
               Object a) }
```
- Aber folgendes:  

```
interface Foldable {
    Object f (Object a); }
```

```
interface Collection {
    Object fold(Foldable f, Object a); }
```
- Vergleiche Iterator aus Collections Framework (Java SE 6):  

```
public interface Iterator<E>
    boolean hasNext();
    E next(); }
```



## Funktionen Höherer Ordnung: C

- Implizit vorhanden:  

```
struct listel {
    void *hd;
    struct listel *tl;
};

typedef struct listel *list;

list filter(int f(void *x), list l);
```
- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Funktionen = Zeiger auf Funktionen
- Benutzung: `signal` (C-Standard 7.14.1)  

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```



## Funktionen Höherer Ordnung: C

Implementierung von `filter`:

```
list filter(int f(void *x), list l)
{ if (l == NULL) {
    return NULL;
  }
  else {
    list r;
    r = filter(f, l->tl);
    if (f(l->hd)) {
      l->tl = r;
      return l;
    }
    else {
      free(l);
      return r;
    }
  }
}
```



## Übersicht: vordefinierte Funktionen auf Listen II

|                        |                                                                                   |                             |
|------------------------|-----------------------------------------------------------------------------------|-----------------------------|
| <code>map</code>       | $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$                               | Auf alle anwenden           |
| <code>filter</code>    | $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$                     | Elemente filtern            |
| <code>foldr</code>     | $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$     | Falten von rechts           |
| <code>foldl</code>     | $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$     | Falten von links            |
| <code>takeWhile</code> | $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$                     | Längster Prefix s.t. p gilt |
| <code>dropWhile</code> | $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$                     | Rest davon                  |
| <code>any</code>       | $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$             | any p = or . map p          |
| <code>all</code>       | $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$             | all p = and . map p         |
| <code>elem</code>      | $\text{Eq } a \rightarrow a \rightarrow [a] \rightarrow \text{Bool}$              | elem x = any (x ==)         |
| <code>zipWith</code>   | $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$ | Verallgemeinertes zip       |



## Typklassen

- Allgemeiner Typ für elem:

```
elem :: a -> [a] -> Bool
```

zu allgemein wegen c ==

- (==) kann nicht für alle Typen definiert werden:
- Gleichheit auf Funktionen nicht entscheidbar.
  - z.B. (==) :: (Int -> Int) -> (Int -> Int) -> Bool
  - Extensionale vs. intensionale Gleichheit



## Typklassen

- Lösung: Typklassen

```
elem :: Eq a => a -> [a] -> Bool
```

```
elem c = any (c ==)
```

- Für a kann jeder Typ eingesetzt werden, für den (==) definiert ist.
- Typklassen erlauben systematisches Überladen (ad-hoc Polymorphie)
  - Polymorphie: auf allen Typen gleich definiert
  - ad-hoc Polymorphie: unterschiedliche Definition für jeden Typ möglich



## Standard-Typklassen

- Eq a für == :: a -> a -> Bool (Gleichheit)
- Ord a für <= :: a -> a -> Bool (Ordnung)
  - Alle Basisdatentypen
  - Listen, Tupel
  - Nicht für Funktionen
  - Damit auch Typ für qsort oben:  
qsort :: Ord a => [a] -> [a]
- Show a für show :: a -> String
  - Alle Basisdatentypen
  - Listen, Tupel
  - Nicht für Funktionen
- Read a für read :: String -> a
  - Siehe Show



## Allgemeine Rekursion

- Einfache 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: 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.



## Beispiel für allgemeine Rekursion: Mergesort

- Hauptfunktion:

```
msort :: [Int] -> [Int]
msort xs
  | length xs <= 1 = xs
  | otherwise = merge (msort front) (msort back) where
    (front, back) = splitAt ((length xs) `div` 2) xs
```
- splitAt :: Int -> [a] -> ([a], [a]) spaltet Liste auf
- Hilfsfunktion: ordnungserhaltendes Zusammenfügen

```
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)
```



## Zusammenfassung

- Funktionen höherer Ordnung
  - Funktionen als gleichberechtigte Objekte und Argumente
  - Spezielle Funktionen höherer Ordnung: map, filter, fold und Freunde
  - Partielle Applikation, Kürzungsregel
- Formen der Rekursion:
  - Einfache und allgemeine Rekursion
  - Einfache Rekursion entspricht fold
- Typklassen
  - Überladen von Bezeichnern

