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 Berechungsmuster
  - 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-> x< y) xs)
```

Beispiel: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl `p` alle Vielfachen heraussieben
  - Dazu: filtern mit `\n-> mod n p /= 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:

$$\texttt{a -> b-> c} \equiv \texttt{a-> (b-> c)}$$

- Inbesondere:

$$\texttt{(a -> b)-> c} \neq \texttt{a-> (b-> c)}$$

- Partielle Anwendung von Funktionen:
  - Für `f :: a-> b-> c`, `x :: a` ist `f x :: b-> c`
- Beispiele:
  - `map toLower :: String-> String`
  - `3 == :: Int-> Bool`
  - `concat . map (replicate 2) :: String-> String`

Die Kürzungsregel

Bei Anwendung der Funktion

$$\texttt{f :: t}_1\texttt{ -> t}_2\texttt{ -> ... -> t}_n\texttt{-> t}$$

auf k Argumente mit $k \leq n$

$$\texttt{e}_1\texttt{ :: t}_1\texttt{, e}_2\texttt{ :: t}_2\texttt{, ..., e}_k\texttt{ :: t}_k$$

werden die Typen der Argumente gekürzt:

$$\texttt{f :: }\not{\texttt{t}}_1\texttt{ -> }\not{\texttt{t}}_2\texttt{ -> ... -> }\not{\texttt{t}}_k\texttt{ -> t}_{k+1}\texttt{ -> ...-> t}_n\texttt{ -> t}$$
$$\texttt{f e}_1\texttt{ ...e}_k\texttt{ :: t}_{k+1}\texttt{ -> ...-> t}_n\texttt{ -> t}$$

- 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:

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

Einfache Rekursion

- Allgemeines Muster:

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

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

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

- Terminiert immer
- Entspricht einfacher Iteration (`while`-Schleife)

Einfach Rekursion durch `foldr`

- Einfache 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
```

- 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`:

```
rev xs = foldr snoc [] xs

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

- Unbefriedigend: doppelte Rekursion

Einfache Rekursion durch `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
```

foldr vs. foldl

- `f` $=$ `foldr` $\otimes$ A entspricht

```
f []     = A
f (x:xs) = x ⊗ f xs
```

  - Kann nicht-strikt in `xs` sein

- `f` $=$ `foldl` $\otimes$ A entspricht

```
f xs       = g A xs
g a []     = a
g a (x:xs) = g (a ⊗ x) xs
```

  - Endrekursiv (effizient), aber strikt in `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 eine Rekursion

foldl = foldr

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

$$A \otimes \mathtt{x} = \mathtt{x} \qquad \text{(Neutrales Element links)}$$
$$\mathtt{x} \otimes A = \mathtt{x} \qquad \text{(Neutrales Element rechts)}$$
$$(\mathtt{x} \otimes \mathtt{y}) \otimes \mathtt{z} = \mathtt{x} \otimes (\mathtt{y} \otimes \mathtt{z}) \qquad \text{(Assoziativät)}$$

- Satz: Wenn $(\otimes, A)$ Monoid, dann

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

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

`map`	`(a-> b)-> [a]-> [b]`	Auf alle anwenden
`filter`	`(a-> Bool)-> [a]-> [a]`	Elemente filtern
`foldr`	`(a -> b -> b) -> b -> [a] -> b`	Falten von rechts
`foldl`	`(a -> b -> a) -> a -> [b] -> a`	Falten von links
`takeWhile`	`(a -> Bool) -> [a] -> [a]`	Längster Prefix s.t. `p` gilt
`dropWhile`	`(a -> Bool) -> [a] -> [a]`	Rest davon
`any`	`(a-> Bool)-> [a]-> Bool`	`any p = or . map p`
`all`	`(a-> Bool)-> [a]-> Bool`	`all p = and . map p`
`elem`	`Eq a=> a-> [a]-> Bool`	`elem x = any (x ==)`
`zipWith`	`(a -> b -> c) -> [a] -> [b] -> [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