Praktische Informatik 3: Funktionale Programmierung
Vorlesung 5 vom 13.11.2012: Funktionen Höherer Ordnung I

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
  - Typinferenz
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- Funktionen höherer Ordnung
- Funktionen als gleichberechtigte Objekte
- Funktionen als Argumente
- Spezielle Funktionen: map, filter, fold und Freunde

Ähnliche Funktionen der letzten Vorlesung

- Pfade:

```
cat :: Path → Path → Path
cat Mt p = p
cat (Cons p ps) qs = Cons p (cat ps qs)
```

```
rev :: Path → Path
rev Mt = Mt
rev (Cons p ps) = cat (rev ps) (Cons p Mt)
```

- Zeichenl

Gelöst durch Polymorphie

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

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


Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen → Int
kasse LeererWagen = 0
kasse (Einkauf a m e) = cent a m + kasse e
```

```
inventur :: Lager → Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m + inventur l
```

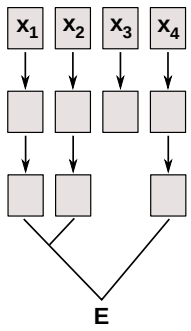
```
len :: MyString → Int
len Empty = 0
len (Cons c str) = 1 + len str
```

Gemeinsamkeiten:

- ein Fall pro Konstruktor
- linearer rekursiver Aufruf
- durch Polymorphie nicht gelöst (Instanz einer Definition)

Muster der primitiven Rekursion

- Anwenden einer Funktion auf jedes Element der Liste
- möglicherweise Filtern bestimmter Elemente
- Kombination der Ergebnisse zu einem Gesamtergebnis E



Ein einheitlicher Rahmen

- Beispiele:

```
toL :: String → String
toL []     = []
toL (c:cs) =
    toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) =
    toUpper c : toL cs
```

- Warum nicht ...

```
map f []     = []
map f (c:cs) = f c : map f cs

toL cs = map toLower cs
toU cs = map toUpper cs
```

- Funktion f als Argument
- Was hätte map für einen Typ?

Funktionen Höherer Ordnung

Slogan

"Functions are first-class citizens."

- Funktionen sind gleichberechtigt: Ausdrücke wie alle anderen
- Grundprinzip der funktionalen Programmierung
- Modellierung allgemeiner Berechungsmuster
- Kontrollabstraktion

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
```

- Definition wie oben

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

- Auswertung (wie vorher):

toL "ABC" ⇝ map toLower ('A':'B':'C':[])
⇝ toLower 'A' : map toLower ('B':'C':[])
⇝ ... ⇝ 'a':'b':'c' : map toLower []
⇝ 'a':'b':'c': [] = "abc"

  - Funktionsausdrücke reduzieren durch Applikation

Funktionen als Argumente: filter

- Elemente filtern: filter
- Signatur:

```
filter :: (α→ Bool)→ [α]→ [α]
```

- Definition

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

- Beispiel:

```
letters :: String→ String
letters = filter isAlpha
```


Beispiel filter: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl *p* alle Vielfachen heraussieben
  - Dazu: filtern mit \n→ mod n p /=0!
  - Namenlose (anonyme) Funktion

```
sieve :: [Integer]→ [Integer]
sieve []     = []
sieve (p:ps) =
  p: sieve (filter (\n→ mod n p /= 0) ps)
```

- Primzahlen im Intervall [1.. n]:

```
primesTo :: Integer→ [Integer]
primesTo n = sieve [2..n]
```

- Die ersten *n* Primzahlen:

```
primes :: Int→ [Integer]
primes n = take n (sieve [2..])
```


Funktionen als Argumente: Funktionskomposition

- Funktionskomposition (mathematisch)

```
(∘) :: (β→ γ) → (α→ β)→ α→ γ
(f ∘ g) x = f (g x)
```

  - Vordefiniert
  - Lies: f nach g

- Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(f >.> g) x = g (f x)
```

  - Nicht vordefiniert!

η-Kontraktion

- Vertauschen der Argumente (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

- Da fehlt doch was?! Nein:

(>.>) = flip (∘) ≡ (>.>) f g a = flip (∘) f g a

- η-Kontraktion (η-Äquivalenz)
  - Bedingung: E :: $\alpha\to\beta$, x :: α, E darf x nicht enthalten

$\lambda x\to E\ x \equiv E$

  - Syntaktischer Spezialfall Funktionsdefinition (punktfreie Notation)

f x = E x ≡ f = E

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$a \to b \to c \equiv a \to (b \to c)$$

  - Inbesondere: $(a \to b) \to c \neq a \to (b \to c)$

- Funktionsanwendung ist linksassoziativ:

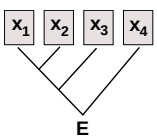
$$f\ a\ b \equiv (f\ a)\ b$$

  - Inbesondere: $f\ (a\ b) \neq (f\ a)\ b$

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

Einfache Rekursion

- Einfache Rekursion: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste (mit einem rekursiven Aufruf)



- Beispiel: kasse, inventur, sum, concat, length, (++), ...
- Auswertung:

sum [4,7,3] ⇝ 4 + 7 + 3 + 0
concat [A, B, C] ⇝ A ++ B ++ C++ []
length [4, 5, 6] ⇝ 1+ 1+ 1+ 0

Einfache Rekursion

- Allgemeines Muster:

f [] = A
f (x:xs) = x ⊗ f xs

- Parameter der Definition:
  - Startwert (für die leere Liste) A :: b
  - Rekursionsfunktion ⊗ :: a -> b-> b
- Auswertung:

f [x1,..., xn] = x1 ⊗ x2 ⊗ ... ⊗ xn ⊗ A

- Terminiert immer (wenn Liste endlich und ⊗, *A* terminieren)
- Entspricht einfacher Iteration (while-Schleife)

Einfach Rekursion durch foldr

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

```
foldr :: (α→ β→ β)→ β→ [α]→ β
```

- Definition

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


Beispiele: foldr

- Summieren von Listenelementen.

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

- Flachklopfen von Listen.

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

- Länge einer Liste

```
length :: [a]→ Int
length xs = foldr (λx n→ n+ 1) 0 xs
```


Beispiele: foldr

- Kasse:

```
type Einkaufswagen = [(Artikel, Menge)]

kasse :: Einkaufswagen→ Int
kasse = foldr (λ(a, m) r→ cent a m+ r) 0
```

- Inventur

```
type Lager = [(Artikel, Menge)]

inventur :: Lager→ Int
inventur = foldr (λ(a, m) r→ cent a m+ r) 0
```


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:

$$\text{foldr} \otimes [x_1, \ldots, x_n]\ A = x_1 \otimes (x_2 \otimes (\ldots (x_n \otimes A)))$$

- Warum nicht andersherum?

$$\text{foldl} \otimes [x_1, \ldots, x_n]\ A = (((A \otimes x_1) \otimes x_2) \ldots) \otimes x_n$$

- Definition von foldl:

```
foldl :: (α → β → α) → α → [β] → α
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```


Beispiel: rev revisited

- Listenumkehr ist falten von links:

```
rev' xs = foldl (flip (:)) [] xs
```

- Nur noch eine Rekursion

foldr vs. foldl

- f = foldr $\otimes$ A entspricht

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

  - Kann nicht-strikt in xs sein, z.B. and, or
- f = foldl $\otimes$ A entspricht

$$\begin{array}{lcl} f\ xs & = & g\ A\ xs \\ g\ a\ [] & = & a \\ g\ a\ (x{:}xs) & = & g\ (a \otimes x)\ xs \end{array}$$

  - Endrekursiv (effizient), aber strikt in xs

foldl = foldr

Definition (Monoid)

$(\otimes, A)$ ist ein Monoid wenn

$$\begin{array}{rr} A \otimes x = x & \text{(Neutrales Element links)} \\ x \otimes A = x & \text{(Neutrales Element rechts)} \\ (x \otimes y) \otimes z = x \otimes (y \otimes z) & \text{(Assoziativität)} \end{array}$$

Theorem

Wenn $(\otimes, A)$ *Monoid, dann für alle A, xs*

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

- Beispiele: length, concat, sum
- Gegenbeispiel: rev

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: Funktionen = Zeiger auf Funktionen

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

```
list filter(int f(void *x), list l);
```

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- 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→ next);
    if (f(l→ elem)) {
       l→next= r;
       return l;
       }
    else {
       free(l);
       return r;
} }    }
```


Übersicht: vordefinierte Funktionen auf Listen II

```
map       :: (α→ β)→ [α]→ [β]      -- Auf alle anwenden
filter    :: (α→ Bool)→ [α]→ [α]   -- Elemente filtern
foldr     :: (α→ β→ β)→ β→ [α]→ β  -- Falten v. rechts
foldl     :: (β→ α→ β)→ β→ [α]→ β  -- Falten v. links
takeWhile :: (α→ Bool)→ [α]→ [α]
dropWhile :: (α→ Bool)→ [α]→ [α]
      -- takeWhile ist längster Prefix so dass p gilt, dropWhile der Rest
any   :: (α → Bool) → [α] → Bool   -- p gilt mind. einmal
all   :: (α → Bool) → [α] → Bool   -- p gilt für alle
elem  :: (Eq α) ⇒ α → [α] → Bool   -- Ist enthalten?
zipWith :: (α → β → γ) → [α] → [β] → [γ]
                                   -- verallgemeinertes zip
```


Zusammenfassung

- Funktionen höherer Ordnung
  - Funktionen als gleichberechtigte Objekte und Argumente
  - Partielle Applikation, η-Kontraktion, namenlose Funktionen
  - Spezielle Funktionen höherer Ordnung: map, filter, fold und Freunde
- Formen der Rekursion:
  - Einfache Rekursion entspricht foldr