Praktische Informatik 3: Funktionale Programmierung

Vorlesung 7 vom 27.11.2018: Funktionen Höherer Ordnung II: Jenseits der Liste

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Diese Woche **zwei** Übungsblätter.
- Ein Bonusübungsblatt für diese Woche.
- Das erste Gruppenübungsblatt — ab **nächste** Woche zwei Wochen.
- Nächste Woche **Tag der Lehre** — Mittwochstutorien fallen aus.

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Zyklische Datenstrukturen
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

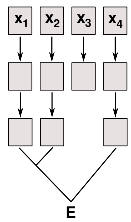
Heute

- Mehr über map und fold
- map und fold sind nicht nur für Listen
- Funktionen höherer Ordnung in anderen Programmiersprachen

Berechnungsmuster

map und filter als Berechnungsmuster

- map, filter, fold als Berechnungsmuster:
  1. Anwenden einer Funktion auf **jedes** Element der Liste
  2. möglicherweise **Filtern** bestimmter Elemente
  3. **Kombination** der Ergebnisse zu Endergebnis E
- Gut parallelisierbar, skalierbar
- Berechnungsmuster für große Datenmengen
  - Map/Reduce (Google), Hadoop



Listenkomprehension

- Besondere Notation: Listenkomprehension
  [f x | x← as, g x] ≡ map f (filter g as)
- Beispiel:
  - Remember this?

```
suche :: Artikel→ Lager→ Maybe Menge
suche a (Lager ps) =
   listToMaybe (map (λ(Posten _ m)→ m)
               (filter (λ(Posten la _) → la == a) ps))
```

  - Sieht so besser aus:

```
suche :: Artikel→ Lager→ Maybe Menge
suche a (Lager ps) =
   listToMaybe [ m | Posten la m ← ps,  la == a ]
```

  - Anderes Beispiel:

```
digits str = [ord x −ord '0'| x← str, isDigit x]
```


Listenkomprehension mit mehreren Generatoren

- Mit mehreren Generatoren werden **alle Kombinationen** generiert:

```
idx :: [String]
idx = [ a: show i| a← ['a'.. 'z'], i← [0.. 9]]
```

Beispiel I: Quicksort

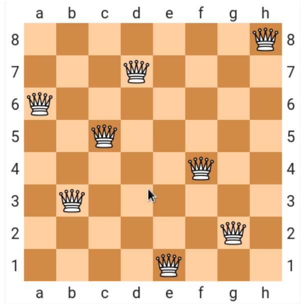
- Quicksort per Listenkomprehension:

```
qsort1 :: Ord α⇒ [α]→ [α]
qsort1 [] = []
qsort1 xs@(x:_) = qsort1 [y | y← xs, y< x ]++
                         [x0| x0← xs, x0 == x ]++
                  qsort1 [z | z← xs,  z> x ]
```

- Erstaunlich effizient
- Einfache Rekursion mit 3-Weg-Split nicht wesentlich effizienter, aber wesentlich länger
- Grund: Sortierte Liste wird nicht im ganzen aufgebaut

Beispiel II: 8-Damen-Problem

- Problem: plaziere 8 Damen sicher auf einem Schachbrett



Source: wikipedia

Beispiel II: n-Damen-Problem

- Spezifikation: Position der Königinnen, Hauptfunktion:

```
type Pos = (Int, Int)
type Board = [Pos]
```

- Rekursive Lösung:
  - Lösung für $n-1$ Königinnen, n-te sicher dazu positionieren
  - Invariante: n-te Königin in n-ter Spalte

```
queens :: Int→ [Board]
queens n = qu n where
  qu :: Int→ [Board]
  qu i | i == 0  = [[]]
       | otherwise =
    [ p++ [(i, j)] | p ← qu (i−1), j ← [1.. n],
                     safe p (i, j)]
```

```
safe :: Board→ Pos→ Bool
```


Map und Fold: Jenseits der Listen

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- Für map gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{length} \circ \text{map } f = \text{length}$$

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeineren?
  $\longrightarrow$ Typklassen?Konstruktorklassen!

Funktoren

- **Konstruktorklassen** sind Typklassen für Typkonstruktoren.
- Die Konstruktorklasse Functor für alle Typen mit einer stukturerhaltenden Abbildung:

```
class Functor f where
  fmap :: (α→ β) → f α → f β
```

- Es sollte gelten (kann nicht geprüft werden):

$$\text{fmap id} = \text{id}$$

$$\text{fmap } f \circ \text{fmap } g = \text{fmap } (f \circ g)$$

- Infix-Synomym <$> für fmap

foldr ist kanonisch

foldr ist die **kanonische strukturell rekursive** Funktion.

- Alle strukturell rekursiven Funktionen sind als Instanz von foldr darstellbar
- Insbesondere auch map und filter
- Es gilt: foldr (:) [] = id
- Jeder algebraischer Datentyp hat ein foldr
- Anmerkung: Typklasse Foldable schränkt Signatur von foldr ein

fold für andere Datentypen

fold ist universell

Jeder algebraische Datentyp T hat genau ein foldr.

- Kanonische Signatur für T:
  - Pro Konstruktor C ein Funktionsargument f_C
  - Freie Typvariable β für T
- Kanonische Definition:
  - Pro Konstruktor C eine Gleichung
  - Gleichung wendet Funktionsparameter f_C auf Argumente an
- Beispiel:

```
data IL = Cons Int IL | Err String | Mt

foldIL :: (Int→ β→ β)→ (String→ β)→ β→ IL→ β
foldIL f e a (Cons i il) = f i (foldIL f e a il)
foldIL f e a (Err str)   = e str
foldIL f e a Mt          = a
```

fold für bekannte Datentypen

- Bool: Fallunterscheidung:

```
data Bool = False | True

foldBool :: β→ β→ Bool→ β
foldBool a1 a2 False = a1
foldBool a1 a2 True  = a2
```

- Maybe α: Auswertung

```
data Maybe α = Nothing | Just α

foldMaybe :: β→ (α→ β)→ Maybe α→ β
foldMaybe b f Nothing  = b
foldMaybe b f (Just a) = f a
```

  - Als maybe vordefiniert

fold für bekannte Datentypen

- Tupel: die uncurry-Funktion

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

- Natürliche Zahlen: Iterator

```
data Nat = Zero | Succ Nat

foldNat :: β→ (β→ β)→ Nat→ β
foldNat e f Zero   = e
foldNat e f (Succ n) = f (foldNat e f n)
```


fold für binäre Bäume

- Binäre Bäume:

```
data Tree α = Mt | Node α (Tree α) (Tree α)
```

  - Label **nur** in den Knoten
- Instanz von fold:

```
foldT :: (α→ β→ β→ β)→ β→ Tree α→ β
foldT f e Mt = e
foldT f e (Node a l r) =  f a (foldT f e l) (foldT f e r)
```

- Instanz von Functor, kein (offensichtliches) Filter

```
instance Functor Tree where
  fmap :: (α→ β)→ Tree α→ Tree β
  fmap f Mt = Mt
  fmap f (Node a l r)=  Node (f a) (fmap f l) (fmap f r)
```


Funktionen mit foldT und mapT

- Höhe des Baumes berechnen:

```
height :: Tree α→ Int
height = foldT (λ_ l r→ 1+max l r) 0
```

- Inorder-Traversion der Knoten:

```
inorder :: Tree α→ [α]
inorder = foldT (λa l r→ l++ [a]++ r) []
```


Kanonische Eigenschaften von foldT und mapT

- Auch hier gilt:

```
foldT Node Mt = id

mapT id = id
mapT f ∘ mapT g = mapT (f ∘ g)
```


Das Labyrinth

- Das Labyrinth als variadischer Baum:

```
data VTree α = Node α [VTree α]
```

```
type Lab α = VTree α
```

- Auch hierfür foldT und mapT:

```
foldT :: (α→ [β]→ β)→ VTree α→ β
foldT f (Node a ns) = f a (map (foldT f) ns)
```

```
mapT :: (α→ β)→ VTree α→ VTree β
mapT f (Node a ns) = Node (f a) (map (mapT f) ns)
```


Suche im Labyrinth

- Tiefensuche via foldT

```
dfts' :: Lab α→ [Path α]
dfts' = foldT add where
  add a [] = [[a]]
  add a ps = concatMap (map (a :)) ps
```

- Problem:
  - foldT terminiert **nicht** für **zyklische** Strukturen
  - Auch nicht, wenn add prüft ob a schon enthalten ist
  - Pfade werden vom Ende konstruiert

Grenzen von foldr

- Andere rekursive Struktur über Listen
  - Quicksort: baumartige Rekursion
- Rekursion nicht über Listenstruktur:
  - take: Rekursion über take

```
take :: Int→ [α]→ [α]
take n _       | n ≤ 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n−1) xs
```

  - Version mit foldr divergiert für nicht-endliche Listen

Funktionen Höherer Ordnung in anderen Sprachen

C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

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

```
extern list map1(void *f(void *x), list l);
```

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Benutzung: qsort (C-Standard 7.20.5.2)

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```


C

- Implementierung von map
- Rekursiv, erzeugt neue Liste:

```
list map1(void *f(void *x), list l)
{
  return l== NULL ?
         NULL : cons(f(l-> elem), map1(f, l-> next));
}
```

- Iterativ, Liste wird in-place geändert (**Speicherleck**):

```
list map2(void *f(void *x), list l)
{
  list c;
  for (c= l; c!= NULL; c= c-> next) {
    c-> elem= f(c-> elem); }
  return l;
}
```


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(); }
```

- Seit Java SE 8 (März 2014): Anonyme Funktionen (Lambda-Ausdrücke)

Python

- Python kennt map, filter, fold:

```
letters = map(chr, range(97, 123))
```

  - Map auf Iteratoren definiert, nicht auf Listen

- Python kennt Listenkomprehension:

```
idx  = [ x+str(i) for x in letters for i in range(10) ]
```

- Python kennt Lambda-Ausdrücke:

```
num  = map (lambda x: 3*x+1, range (1,10))
```


Zusammenfassung

- map, filter, fold sind ein nützliches, skalierbares und allgemeines **Berechnungsmuster**.
- Listenkomprehensionen sind nützlicher syntaktischer Zucker.
- map und fold sind **kanonische Funktionen höherer Ordnung**, und für alle Datentypen definierbar.
- Nächste Woche: Funktionale Programmierung im Großen — Abstrakte Datentypen