Praktische Informatik 3: Funktionale Programmierung

Vorlesung 7 (29.11.2020): Funktionen Höherer Ordnung II

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

Organisatorisches

- Die Vorlesung **nächste Woche** findet im **NW2 A0242** statt.

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Rekursive und zyklische Datenstrukturen
  - 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

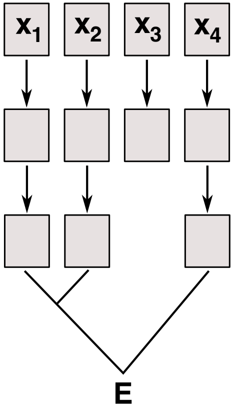
Lernziel

Wir verstehen, warum `map` und `fold` besonders sind, wie sie für andere Datentypen aussehen, und wann wir sie benutzen können.

I. 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 ]
```

Listenkomprehension mit mehreren Generatoren

- Anderes Beispiel: Primzahlzwillinge

```
twin_primes :: [(Integer, Integer)]
twin_primes = [(x, y) | (x, y) ← zip primes (tail primes), x+2 == y]
```

- 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 effizienter, aber wesentlich länger

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 ]
```

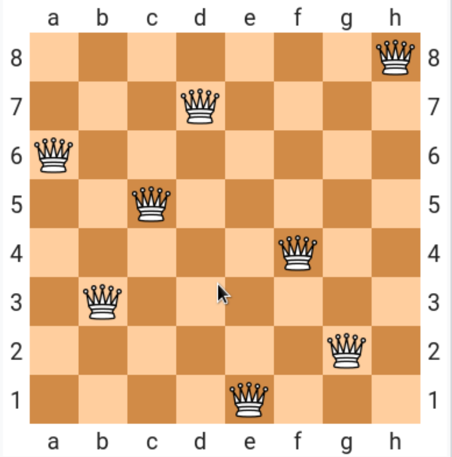
- Erstaunlich effizient

DEMO

- Einfache Rekursion mit 3-Weg-Split 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

- Position der Königinnen:

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

- Rekursiv: Lösung für $n - 1$ Königinnen, n-te sicher dazu positionieren

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

- Invariante: n-te Königin in n-ter Spalte

Beispiel II: n-Damen-Problem

- Wann ist eine Königin sicher?

```
safe :: Board→ Pos→ Bool
safe others nu = and [ not (threatens other nu) | other ← others ]
```

- Bedrohung: gleiche Zeile oder Diagonale

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

  - Diagonalen charakterisiert durch $y = a + x$ bzw. $y = a - x$ für konstantes a
  - Gleiche Spalte (`i == m`) durch Konstruktion ausgeschlossen

☞ Siehe Übung 7.1

II. Map und Fold: Jenseits der Listen

`map` als strukturerhaltende Abbildung

`map` ist die kanonische **strukturerhaltende Abbildung**

- Für `map` gelten folgende Aussagen:

$$\texttt{map id} = \texttt{id}$$

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

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

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeinern?

`map` als strukturerhaltende Abbildung

`map` ist die kanonische **strukturerhaltende Abbildung**

- Für `map` gelten folgende Aussagen:

$$\texttt{map id} = \texttt{id}$$

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

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

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeinern?

$\longrightarrow$ Typklassen?

`map` als strukturerhaltende Abbildung

`map` ist die kanonische **strukturerhaltende Abbildung**

- Für `map` gelten folgende Aussagen:

$$\texttt{map id} = \texttt{id}$$

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

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

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeinern?

$\longrightarrow$ 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):

$$\texttt{fmap id} = \texttt{id}$$

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

- Infix-Synomym <$> für `fmap`

Instanzen von `Functor`

- Listen sind eine Instanz von `Functor`, aber es gibt `map` und `fmap`

Instanzen von `Functor`

- Listen sind eine Instanz von `Functor`, aber es gibt `map` und `fmap`
- `Maybe` ist eine Instanz von `Functor`:

```haskell
instance Functor Maybe where
   fmap f (Just a) = Just (f a)
   fmap f Nothing  = Nothing
```

  - Propagiert `Nothing` — oft sehr nützlich

Instanzen von `Functor`

- Listen sind eine Instanz von `Functor`, aber es gibt `map` und `fmap`
- `Maybe` ist eine Instanz von `Functor`:

```haskell
instance Functor Maybe where
   fmap f (Just a) = Just (f a)
   fmap f Nothing  = Nothing
```

  - Propagiert `Nothing` — oft sehr nützlich
- Tupel sind Instanzen von `Functor` im **zweiten** Argument, bspw:

```haskell
instance Functor (a, ) where
  fmap f (a, b) = (a, f b)
```

`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` ☞ Siehe Übung 7.3
- Es gilt: `foldr (:) [] = id`
- Jeder algebraischer Datentyp hat ein `foldr`
- Nicht als Konstrukturklasse darstellbar (wie `Functor` und `fmap`)
  - 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 $f_{\mathtt{C}}$
  - Freie Typvariable β für `T`
- Kanonische Definition:
  - Pro Konstruktor `C` eine Gleichung
  - Gleichung wendet $f_{\mathtt{C}}$ auf Argumente an (und `fold` rekursiv auf Argumente vom Typ `T`)

`fold` für andere Datentypen

- Beispiel:

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

- Das Fold dazu:

`fold` für andere Datentypen

- Beispiel:

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

- Das Fold dazu:

```
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
```

- Was ist das?
  - Eine Art Listen von `Int` mit Fehlern („Ausnahmen“)
  - Das zweite Argument von `foldIL` fängt aufgetretene Ausnahmen

`fold` für bekannte Datentypen

- `Bool`:

```
data Bool = False | True

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

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:

```
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
```

`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:

```
data (α, β) = (α, β)
```

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

fold für bekannte Datentypen

- Tupel: die uncurry-Funktion

```
data (α, β) = (α, β)
```

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

- Dazu gehört die Funktion curry (beide vordefiniert):

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

- Die beiden sind **invers**:

$$\texttt{uncurry} \circ \texttt{curry} = \texttt{id} \quad \texttt{curry} \circ \texttt{uncurry} = \texttt{id}$$

fold für bekannte Datentypen

- Natürliche Zahlen:

```
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 bekannte Datentypen

- 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)
```

- Wendet Funktion `f` `n`-mal auf Startwert `e` an:

$$\texttt{foldNat}\ e\ f\ n = f^n(e)$$

- Konversion nach `Int`:

```
natToInt :: Nat→ Int
natToInt = foldNat 0 (1+)
```

☞ Siehe Übung 7.2

fold für binäre Bäume

- Binäre Bäume:

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

  - Label **nur** in den Knoten

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

- Instanz von `Functor`, kein (offensichtliches) Filter

```
instance Functor Tree where
  fmap f Mt = Mt
  fmap f (Node a l r)=  Node (f a) (fmap f l) (fmap f r)
```

Funktionen mit `foldT`

- Höhe des Baumes berechnen:

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

- Inorder-Traversion der Knoten:

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

- Enthält der Baum dieses Element?

```
isElem :: Eq α⇒ α→ Tree α→ Bool
isElem a = foldT False (λb l r→ a == b || l || r)
```

  - Nich-Striktheit von `||` begrenzt Traversion

Kanonische Eigenschaften von `foldT` und `fmap`

- Auch hier gilt:

$$\texttt{foldT Mt Node} = \texttt{id}$$

$$\texttt{fmap id} = \texttt{id}$$

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

- Gilt für **alle** Datentypen. Insbesondere gilt:

$$\texttt{fold}\ C_1\ C_2 \ldots C_n = \texttt{id}$$

Falten mit den Konstruktoren ergibt die Identität.

Variadische Bäume

- Das Labyrinth ist ein variadischer Baum:

```
data VTree α = NT α [VTree α]
```

- Auch hierfür `fold` und `map`:

Variadische Bäume

- Das Labyrinth ist ein variadischer Baum:

```
data VTree α = NT α [VTree α]
```

- Auch hierfür `fold` und `map`:

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

```
instance Functor VTree where
  fmap f (NT a ns) = NT (f a) (map (fmap f) ns)
```

Suche im Labyrinth

- Tiefensuche via `foldT`

```
dfs1 :: VTree α → [Path α]
dfs1 = foldT add where
  add a [] = [[a]]
  add a ps = [ a:p | p ← concat ps]
```

- Problem:

  - `foldT` terminiert **nicht** für **zyklische** Strukturen

Suche im Labyrinth

- Tiefensuche via `foldT`

```
dfs2 :: Eq α⇒ VTree α→ [Path α]
dfs2 = foldT add where
  add a [] = [[a]]
  add a ps = [a:p | p ← concat ps, not (a ‘elem‘ p) ]
```

- 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`

- `foldr` traversiert die gesamte Struktur, konstruiert Ergebnis von nicht-rekursiven Konstruktoren her
- Nicht-Striktheit erlaubt zyklische Strukturen, wenn **lokal** Abbruch der Rekursion möglich
  - Beispiel: `all = foldr (&&) True`
  - Gegenbeispiel: Tiefensuche in zyklischen Strukturen, Breitensuche
- `foldl` ist **nicht** generalisierbar
  - Warum?

Grenzen von `foldr`

- `foldr` traversiert die gesamte Struktur, konstruiert Ergebnis von nicht-rekursiven Konstruktoren her
- Nicht-Striktheit erlaubt zyklische Strukturen, wenn **lokal** Abbruch der Rekursion möglich
  - Beispiel: `all = foldr (&&) True`
  - Gegenbeispiel: Tiefensuche in zyklischen Strukturen, Breitensuche
- `foldl` ist **nicht** generalisierbar
  - Warum? Nur für **linear rekursive** Typen

Andere Arten der Rekursion

- Andere rekursive Struktur über Listen
  - Quicksort: **baumartige** Rekursion
- Rekursion nicht (nur) über Listenstruktur:
  - `take`: Begrenzung der Rekursion

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

  - Version mit `fold` divergiert für nicht-endliche Listen

☞ Siehe Übung 7.4

III. Funktionen höher Ordnung in anderen Programmiersprachen

Funktionen höherer Ordnung in 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

- Einge Funktionen höherer Ordnung sind speziell:
  - `map` ist die strukturerhaltende Funktion
  - `fold` ist die strukturelle Rekursion über dem Typen
- Jeder Datentyp hat `map` und `fold`
- Konstruktorklassen sind Klassen für Typkonstruktoren
  - Beispiel `Functor`
- Listenkomprehension ist ein nützlicher, leichtgewichtiger syntaktischer Zucker für `map` und `filter`

Zusammenfassung

- Einge Funktionen höherer Ordnung sind speziell:
  - `map` ist die strukturerhaltende Funktion
  - `fold` ist die strukturelle Rekursion über dem Typen
- Jeder Datentyp hat `map` und `fold`
- Konstruktorklassen sind Klassen für Typkonstruktoren
  - Beispiel `Functor`
- Listenkomprehension ist ein nützlicher, leichtgewichtiger syntaktischer Zucker für `map` und `filter`
- Die Vorlesung **nächste Woche** findet im **NW2 A0242** statt.