Praktische Informatik 3: Funktionale Programmierung

Vorlesung 4 / 14. November 2023

Funktionen höherer Ordnung

Thomas Barkowsky

Wintersemester 2023/24

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- Testen und Qualitätssicherung
- I/O, Aktionen und Zustände
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

heute in dieser Vorlesung...

- Funktionen höherer Ordnung
  - Funktionen als Argumente von Funktionen
  - *Currying*
  - Funktionen für Listen: `map`, `filter`, etc.
  - Faltung von Listen: `foldr` und `foldl`
  - Lambda-Ausdrücke
  - Funktionskomposition

Funktionen höherer Ordnung

Funktionen höherer Ordnung

- = Funktionen, die Funktionen als Argumente haben oder eine Funktion als Resultat liefern
- Funktionen in Haskell sind “first-class citizens” *(Christopher Strachey)*
  - Grundprinzip funktionaler Programmierung: Funktionen sind Ausdrücke
- Exkurs: *Curried Functions* (benannt nach Haskell B. Curry)
- Beispiel: Was tut diese Funktion?

```
f1 :: Int -> (Int -> Int)
f1 a b = a + b
```

Beispiel: *Curried Functions*

```
f1 :: Int -> (Int -> Int)
f1 a b = a + b
```

```
*Main> f1 4 5
9
*Main> :type f1 4
f1 4 :: Int -> Int
*Main> (f1 4) 5
9
```

- → Äquivalente Definition:

```
f1 :: Int -> (Int -> Int)
(f1 a) b = a + b
```

- Und der Unterschied zu:

```
f2 :: Int -> Int -> Int
f2 a b = a + b
```

?

Beispiel: *Curried Functions*

- Es gibt keinen!
  (Konvention zur Vermeidung von Klammern)

```
*Main> (f2 4) 5
9
*Main> :type f1
f1 :: Int -> Int -> Int
*Main> :type f2
f2 :: Int -> Int -> Int
```

- Merke:
  - “ `->` ” ist rechtsassoziativ:
    `Int -> Int -> Int` bedeutet `Int -> (Int -> Int)`
  - Funktionsanwendung ist linksassoziativ:
    `f a b bedeutet (f a) b`

Funktionen als Argumente

- Beispiel:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

```
*Main> twice (*2) 10
40
*Main> twice reverse "hallo"
"hallo"
```

- auch hier: *Currying* durch partielle Anwendung von Funktionen:

```
*Main> :t twice (*2)
twice (*2) :: Num a => a -> a
```

Funktionen höherer Ordnung für Listen

- `map`:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

```
*Main> map (+1) [2,4,6,8]
[3,5,7,9]
*Main> map even [1..5]
[False,True,False,True,False]
*Main> map reverse ["abc","def","ghi"]
["cba","fed","ihg"]
```

Funktionen höherer Ordnung für Listen

- `filter`:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

```
Prelude> filter even [1..10]
[2,4,6,8,10]
Prelude> filter (>5) [1..10]
[6,7,8,9,10]
Prelude> filter (/=' ') "ab cd ef"
"abcdef"
```

Funktionen höherer Ordnung für Listen

- `all`:

```
Prelude> all even [2,4,8]
True
```

- `any`:

```
Prelude> any odd [2,4,9]
True
```

- `takeWhile`:

```
Prelude> takeWhile even [2,4,7,8]
[2,4]
```

- `dropWhile`:

```
Prelude> dropWhile even [2,4,7,8]
[7,8]
```

Strukturelle Rekursion: Faltungen

Strukturelle Rekursion: `foldr`

- Bekanntes Muster für Rekursion über Listen:

```
f []     = v
f (x:xs) = x # f xs
```

  - leere Liste wird auf einen festen Wert `v` abgebildet
  - bei nicht-leerer Liste wird ein Operator `#` auf das erste Element und den Rest der Liste angewendet

- Beispiele:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

```
product []     = 1
product (x:xs) = x * product xs
```

```
or []     = False
or (x:xs) = x || or xs
```

Kapselung des Musters durch `foldr`

- Funktion höherer Ordnung in *standard prelude*
  - Abkürzung für *fold right*
  - 2 Parameter: Basiswert `v` und Operator `#`

```
sum []     = 0
sum (x:xs) = x + sum xs
```

```
product []     = 1
product (x:xs) = x * product xs
```

```
or []     = False
or (x:xs) = x || or xs
```

```
sum :: Num a => [a] -> a
sum = foldr (+) 0
```

```
product :: Num a => [a] -> a
product = foldr (*) 1
```

```
or :: [Bool] -> Bool
or = foldr (||) False
```

  - Operatoren, die als Argumente benutzt werden, müssen geklammert werden

Definition von `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- Aufruf mit leerer Liste liefert Wert `v`
- bei nicht-leerer Liste wird Funktion `f` angewendet auf Kopf und rekursiven Aufruf mit Rest der Liste
- Anschauliche, nicht-rekursive Interpretation von `foldr`…

`foldr` anschaulich...

- cons Operatoren der übergebenen Liste werden durch Funktion `f` ersetzt, leere Liste am Ende durch Wert `v`
- Beispiel:
  `foldr (+) 0` angewendet auf `1 : (2 : (3 : []))`
  (= Liste `[1,2,3]`)
  liefert: `1 + (2 + (3 + 0))`
- `foldr` ist ein sehr mächtiges Werkzeug!
- weiteres Beispiel...

Was tut diese Funktion?

```
l :: [a] -> Int
l = foldr (\ _ n -> 1 + n) 0
```

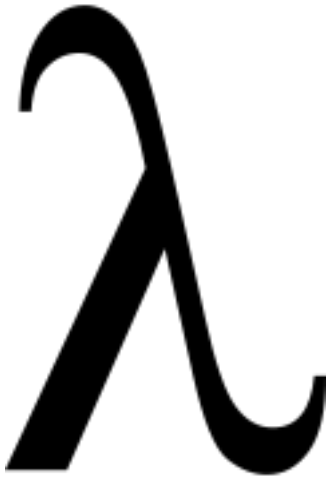
... und was ist das???

Exkurs: **Lambda-Ausdrücke**

\ = λ (der griechische Buchstabe Lambda)

vgl: Lambda-Kalkül *(Alonzo Church, Stephen C. Kleene)*

formales Instrument in der Theorie zu Funktionen



Lambda-Ausdrücke in Haskell

- Alternative zur Funktionsdefinition durch Gleichungen
- Wie gehabt:
  - *Pattern* für formale Argumente
  - Rumpf mit Funktionsdefinition
- Aber: Kein Funktionsbezeichner!
  - Lamda-Ausdrücke sind **namenlose** Funktionen
- Beispiel: `\x -> x + x`
- Verwendung wie gewöhnliche Funktionen:

```
Prelude> (\x -> x + x) 13
26
```

Warum Lambda-Ausdrücke?

- Formalisierung von Funktionsdefinitionen (*Currying*!):
  - Beispiel:

```
add :: Int -> Int -> Int
add x y = x + y
```

```
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

  - Signatur und Funktionsdefinition syntaktisch gleich: `? -> (? ->?)`
- Funktionen, die Funktionen zurückliefern (eigentlich auch *Currying*...)
  - Beispiel:

```
const :: a -> b -> a
const x _ = x
```

```
const :: a -> (b -> a)
const x = \_ -> x
```

  - `const` erzeugt eine Funktion!

Aber vor allem:

- Wir wollen Funktionen oft gar nicht benennen!
  - insbesondere dann nicht, wenn sie nur lokal und/oder einmalig benötigt werden...
  - Beispiel: erste n ungeraden Zahlen

```
odds :: Int -> [Int]
odds n = map f [0..n-1]
         where f x = x*2 + 1
```

```
odds :: Int -> [Int]
odds n = map (\x -> x*2 + 1) [0..n-1]
```

Also: was tut diese Funktion?

```
l :: [a] -> Int
l = foldr (\ _ n -> 1 + n) 0
```

- Beispiel: Liste `[1,2,3] = 1 : (2 : (3 : []))`
  - `cons` Operatoren werden sukzessive ersetzt durch 1 + <zweites Argument>
  - leere Liste wird ersetzt durch 0
  - also:
    `1 : (2 : (3 : [])) → 1 + (1 + (1 + 0)) = 3`

Fazit: `foldr`

- `foldr` steht für *fold right* ,
- d.h.: Operator ist **rechtsassoziativ**
- z.B. `foldr (+) 0 [1,2,3] = 1 + (2 + (3 + 0))`
- oder allgemein:

- `foldr (#) v [x0,x1,...,xn] =`

```
x0 # (x1 # (... # (xn # v)))
```

`foldr` anders herum: `foldl`

- `foldr (#) v [x0,x1,...,xn]  =  x0 # (x1 # (... # (xn # v)))`
- vs.
- `foldl (#) v [x0,x1,...,xn]  =  (((v # x0) # x1) # ...) # xn`
- `foldl` arbeitet **linksassoziativ**
- v fungiert als **Akkumulator**

`foldr` anders herum: `foldl`

- Beispiel `sum`, linksassoziativ:

```
sum = sum' 0 where
      sum' v []     = v
      sum' v (x:xs) = sum' (v+x) xs
```

- Rekursives Muster:
  - leere Liste wird auf einen Akkumulator `v` abgebildet
  - bei nicht-leerer Liste wird rekursiv der Rest der Liste bearbeitet, mit aktualisiertem Akkumulator durch Anwendung von `#` auf das erste Element
  - damit ist `foldl` ein **Iterator** mit Anfangszustand `v` und Iterationsfunktion `#`

```
f v []     = v
f v (x:xs) = f (v # x) xs
```

`foldl` vs. `foldr`

- Damit, analog zu obigen Definitionen:
  - (`foldr` ersetzt durch `foldl`)

```
sum :: Num a => [a] -> a
sum = foldl (+) 0
```

```
product :: Num a => [a] -> a
product = foldl (*) 1
```

```
or :: [Bool] -> Bool
or = foldl (||) False
```

- anders hier:

```
l :: [a] -> Int
l = foldl (\n _ -> n + 1) 0
```

vs.

```
l :: [a] -> Int
l = foldr (\_ n -> 1 + n) 0
```

foldl vs. foldr: Definitionen

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
```

- Wesentliche Unterschiede?

`foldl` vs. `foldr`: Unterschiede

- `foldr` ist
  - nicht-strikt bzgl. des Rests der Liste (Beispiel: `and`, `or`), daher
  - auch für unendliche Listen verwendbar
  - (es wird u.U. nicht die gesamte Liste verarbeitet)
- `foldl` ist
  - strikt bzgl. der Liste (es wird immer die ganze Liste verarbeitet), daher
  - nicht terminierend für unendliche Listen
  - aber: effizient, da endrekursiv!

Funktionskomposition

- Anwendung einer Funktion auf den Ausgabewert einer anderen Funktion
- in Haskell: `f (g x)`
- Operator: `.`
- Signatur:
```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```
- Definition:
```
f . g = \x -> f (g x)
```
- äquivalent zu:
```
(f . g) x = f (g x)
```
- Vorteil?
  - vereinfachte Schreibweise für geschachtelte Funktionen
  - weniger Klammern
  - erstes Argument implizit

Funktionskomposition: Beispiele

```
odd n = not (even n)
```

```
twice f x = f (f x)
```

```
odd = not . even
```

```
twice f = f . f
```

- Assoziativität:

```
sumsqreven ns = sum (map (^2) (filter even ns))
```

```
sumsqreven = sum . map (^2) . filter even
```

- keine Klammerung, da `f.(g.h) = (f.g).h` für alle Funktionen `f, g, h`

Zusammenfassung

- Funktionen sind *first-class citizens*
- Partielle Anwendung von Funktionen: *Currying*
- Funktionen höherer Ordnung für Listen: `map`, `filter`, `all`, …
- Strukturelle Rekursion mit `foldr`
- Lambda-Ausdrücke zur anonymen Funktionsdefinition
- Iteration mit `foldl`
- Geschachtelte Funktionen durch Funktionskomposition

nächstes Mal...

- Algebraische Datentypen
  - Definition neuer Typen mit `type` und `data`
  - Aufzählungstypen
  - Parametrisierte Datentypen
  - Datentypen zur Fehlerbehandlung
  - Rekursive Datentypen (Listen, Bäume & Co.)