Programmieren für Fortgeschrittene - eine Einführung in Haskell

Tag 4 — ein bisschen noch

Stephan Mielke, 12.01.2015

Technische Universität Braunschweig, IPS

Überblick

- **Lazy**
- Funktionen höherer Ordnung (HOF)
- Lambda Ausdrücke
- Beispiele für Currying, Lambda und HOF
- Listen API

Lazy

Betrachte folgende Funktion

```
rechne :: Double -> Double -> Double
rechne a b = if a > 10
             then a + b
             else a
```

Was erwartet ihr beim Aufruf von

```
rechne 12 6
```

Lazy

Betrachte folgende Funktion

```
rechne :: Double -> Double -> Double
rechne a b = if a > 10
             then a + b
             else a
```

und bei

```
rechne 9 (10 / 0)
```

Lazy

Wir betrachten

```
prims :: [Integer]-> Int -> [Integer]
prims _ 0  = []
prims (p:xs) i  = (:) p $prims
                  [x|x<- xs, mod x p > 0] $i + 1
```

Terminiert die Funktion?

Lazy

Wir betrachten

```
prims :: [Integer]-> Int -> [Integer]
prims _ 0  = []
prims (p:xs) i  = (:) p $prims
                  [x|x<- xs, mod x p > 0] $i + 1
```

Terminiert die Funktion?

Terminiert die Funktion auch bei der Eingabe von

```
primes [2..] 1
```

Lazy

- Haskell verwendet die Lazy-Evaluation für Ausdrücke
- Lazy $\equiv$ Call-by-Need
- Dadurch sind Funktionen nicht strikt

Lazy — unendliche Listen

- Es werden vom Start an eine bestimmte Anzahl an Elemente erstellt
- Wenn weitere Elemente benötigt werden, werden diese neu erstellt
- Wird immer nur ein Abschnitt benötigt, wird der Start wieder gelöscht stellt es euch als „Ringpuffer“ vor
- Wenn jedoch alle Elemente benötigt werden $\rightarrow$ bis Speicher voll
- z.B.: für die Liste `[x**x | x <- [1..]]` sind wir bei Element 42, die Elemente 32–52 sind bereits berechnet

```
drop 32 $ take 52 [x**x | x <- [1..]]
[1.2911004008776103e50,1.1756638905368616e52,1.1025074993541487e54,1.0638735892371651e56
      ,1.0555134955777783e58,1.0759111801979994e60,1.1259514746207l2e62,1.2089258196146292e64
      ,1.3308776306327l2e66,1.5013093754529656e68,1.7343773367030268e70,2.05077382356061e72
      ,2.4806364445134117e74,3.068034630079427e76,3.877924263464449e78,5.007020782634593e80
      ,6.6009724686219 54e82,8.881784197001252e84,1.2192113050946485e87,1.7067655527413216e89]
```

Lazy — Parameter und Ausdrücke

- Haskell verwendet Call-by-Need
- Call-by-Need ist eine Form des Call-by-Name

Lazy — Call-by-Name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- `max (4 + 6)(10 / 0)`

Lazy — Call-by-Name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- `max (4 + 6)(10 / 0)`
- $\Rightarrow$ `if (4 + 6)> (10 / 0)then (4 + 6)else (10 / 0)`

Lazy — Call-by-Name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- `max (4 + 6)(10 / 0)`
- $\Rightarrow$ `if (4 + 6)> (10 / 0)then (4 + 6)else (10 / 0)`
- $\Rightarrow$ `(4 + 6)> (10 / 0)`

Lazy — Call-by-Name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- `max (4 + 6)(10 / 0)`
- ⇒ `if (4 + 6)> (10 / 0)then (4 + 6)else (10 / 0)`
- ⇒ `(4 + 6)> (10 / 0)`
- ⇒ `10 > (10 / 0)`

Lazy — Call-by-Name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- `max (4 + 6)(10 / 0)`
- ⇒ `if (4 + 6)> (10 / 0)then (4 + 6)else (10 / 0)`
- ⇒ `(4 + 6)> (10 / 0)`
- ⇒ `10 > (10 / 0)`
- ⇒ ↯

Lazy — Call-by-Need

- Call-by-Need erweitert Call-by-Name um Sharing
- Sharing: gleiche Ausdrücke werden nur einmal ausgewertet

```
(product [1..50000]) - (product [1..49999])
...

take 9 [div 100 (10 - x) | x <- [1..]]
[11,12,14,16,20,25,33,50,100]
```

Überblick

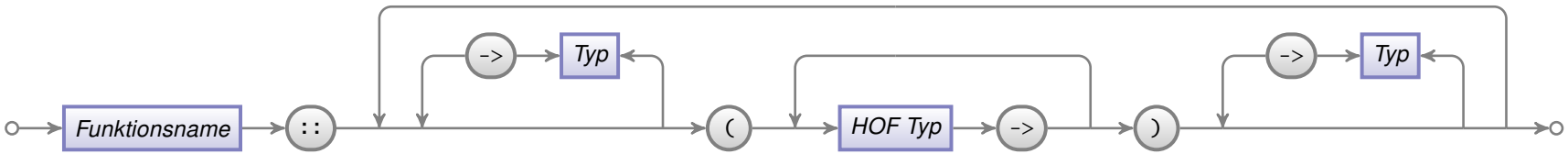
- Lazy
- **Funktionen höherer Ordnung (HOF)**
- Lambda Ausdrücke
- Beispiele für Currying, Lambda und HOF
- Listen API

Funktionen höherer Ordnung (HOF)- Allgemeines zu HOF

- **Funktionen höherer Ordnung (HOF)**
  - Allgemeines zu HOF
  - Funktionskomposition

Funktionen höherer Ordnung (HOF)

- Funktionen können als Parameter nicht nur Ausdrücke sondern auch Funktionen erhalten
- Dieses wird im Funktionskopf angegeben



Funktionen höherer Ordnung (HOF)

```
filter :: (Int -> Bool) -> [Int] -> [Int]
filter do []     = []
filter do (x:xs) | do x       = x : filter do xs
                 | otherwise = filter do xs
```

Funktionen höherer Ordnung (HOF)- Funktions-komposition

- **Funktionen höherer Ordnung (HOF)**
  - Allgemeines zu HOF
  - Funktionskomposition

Funktionskomposition

Wie vermeiden wir am besten „Klammerungswirrwarr“

```
f (f (f (f (f (f (f (f (f x)))))))))
```

Funktionskomposition

Wie vermeiden wir am besten „Klammerungswirrwarr“

```
f (f (f (f (f (f (f (f (f x)))))))))
```

Mit dem „Punkt“-Operator können wir Funktionen verbinden

```
(f.f.f.f.f.f.f.f.f)x
```

Funktionskomposition

Wie vermeiden wir am besten „Klammerungswirrwarr“

```
f (f (f (f (f (f (f (f (f x)))))))))
```

Mit dem „Punkt“-Operator können wir Funktionen verbinden

```
(f.f.f.f.f.f.f.f.f)x
```

Oder dem `$`-Operator die Auswertungsreihenfolge verändern

```
f $ f $ f $ f $ f $ f $ f $ f $ f x
```

Funktionskomposition

Der „Punkt“-Operator ist definiert mit

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) outerFunc innerFunc x = outerFunc (innerFunc x)
```

Das Resultat der inneren Funktion wird auf die äußere angewandt

Funktionskomposition

Der `$`-Operator ist definiert mit

```
($) :: (a -> b) -> a -> b
($) func x = func x
```

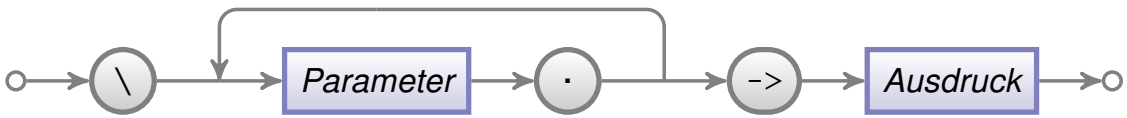
Die Funktion wird auf das Resultat von dem Ausdruck der „rechts“ vom Operator steht angewandt

Überblick

- Lazy
- Funktionen höherer Ordnung (HOF)
- **Lambda Ausdrücke**
- Beispiele für Currying, Lambda und HOF
- Listen API

Anonyme Funktionen

- Haskell unterstützt anonyme Funktionen in Form von λ-Ausdrücken
- Das λ-Symbol wird durch „\“ repräsentiert
  $\lambda x \rightsquigarrow \mathtt{x}$
- Aufbau:
- Durch das Currying können λ-Ausdrücke mehrere Argumente besitzen
- Es gelten alle bekannten Regeln für die λ-Notation



Beispiele - Lambda-Ausdrücke

```
plus = \x -> \y -> x + y

istKleiner = \x -> \y ->  x < y

g = \x -> \y -> (\y -> \x -> (y,x)) y x
```

Beispiele — Lambda-Ausdrücke

```haskell
-- y "Operator"
y f = f (y f)

fac = y (\f n -> if n > 0 then n * f (n - 1) else 1)
```

Beispiele — Lambda-Ausdrücke

```haskell
-- y "Operator"
y f = f (y f)

fac = y (\f n -> if n > 0 then n * f (n - 1) else 1)
```

Aufruf

```haskell
fac 5
```

Beispiele — Lambda-Ausdrücke

```haskell
-- y "Operator"
y f = f (y f)

fac = y (\f n -> if n > 0 then n * f (n - 1) else 1)
```

Aufruf

```
fac 5
```

Ausgabe

```
120
```

Überblick

- Lazy
- Funktionen höherer Ordnung (HOF)
- Lambda Ausdrücke
- **Beispiele für Currying, Lambda und HOF**
- Listen API

Menge I

```
module MySet(myPrint, myElem, mySingleSet, myBoolSet,
    myListSet, myAdd, myUnion, myIntersect, myDiff, myFilter
    , myForAll, myExists, mySubset, myEquals) where

type Set = Int -> Bool

myStart = -10000
myEnd   = 10000

myPrint :: Set -> [Int]
myPrint set = myPrint' set myStart myEnd
  where
    myPrint' s min max | min > max     = []
      | myElem s min = min : (myPrint' s (min + 1) max)
      | otherwise    = myPrint' s (min + 1) max
```

Menge II

```haskell
myElem :: Set -> Int -> Bool
myElem s e = s e

mySingleSet :: Int -> Set
mySingleSet e = (==) e

myBoolSet :: (Int -> Bool) -> Set
myBoolSet p = p

myListSet :: [Int] -> Set
myListSet list = \x -> myFind list x
  where
    myFind :: [Int] -> Int -> Bool
    myFind []     _ = False
    myFind (y:ys) x = x == y || myFind ys x
```

Menge III

```
myAdd :: Set -> Int -> Set
myAdd s i = (mySingleSet i) `myUnion` s

myUnion :: Set -> Set -> Set
myUnion a b = \x -> ((myElem a x) || (myElem b x))

myIntersect :: Set -> Set -> Set
myIntersect a b = \x -> ((myElem a x) && (myElem b x))

myDiff :: Set -> Set -> Set
myDiff a b = \x -> ((myElem a x) && not (myElem b x))

myFilter :: Set -> (Int -> Bool) -> Set
myFilter s f = \x -> (f x && s x)
```

Menge IV

```
myForAll :: Set -> (Int -> Bool) -> Int -> Int -> Bool
myForAll s p min max
      | min > max     = True
      | myElem s min = p min && myForAll s p (min + 1) max
      | otherwise     = myForAll s p (min + 1) max

myExists :: Set -> (Int -> Bool) -> Int -> Int -> Bool
myExists s p min max = not (myForAll s (not.p) min max)

mySubset :: Set -> Set -> Bool
mySubset a b = myForAll a (\x -> myExists b (\y -> x == y)
    myStart myEnd) myStart myEnd

myEquals :: Set -> Set -> Bool
myEquals a b = (mySubset a b) && (mySubset b a)
```

Überblick

- Lazy
- Funktionen höherer Ordnung (HOF)
- Lambda Ausdrücke
- Beispiele für Currying, Lambda und HOF
- **Listen API**

Listen API

Basis Funktionen

- `(++) :: [a] -> [a] -> [a]`
  hängt die zweite an die erste Liste
- `last :: [a] -> a`
  letztes Element
- `init :: [a] -> [a]`
  ohne das letzte Element
- `null :: [a] -> Bool`
  testet, ob leer
- `length :: [a] -> Int`
  Länge der Liste

Listen API

transformations Funktionen

- `map :: (a -> b)-> [a] -> [b]`
  wendet die Funktion `a -> b` auf jedes Element an
- `reverse :: [a] -> [a]`
  dreht die Liste um
- `subsequences :: [a] -> [[a]]`
  berechnet die Potenzmenge
- `permutations :: [a] -> [[a]]`
  berechnet alle Permutationen

Listen API

```
map (^2) [1..10]
[1,4,9,16,25,36,49,64,81,100]

reverse [-x | x <- [1,,10]]
[-10,-9,-8,-7,-6,-5,-4,-3,-2,-1]

Data.List.subsequences "123"
["","1","2","12","3","13","23","123"]

Data.List.permutations "123"
["123","213","321","231","312","132"]
```

Listen API

Auflösungs-Funktionen

- `foldl :: (b -> a -> b)-> b -> [a] -> b`
  wendet die Funktion `(b -> a -> b)` mit dem übergebenen Element von links nach rechts an
- `foldr :: (a -> b -> b)-> b -> [a] -> b`
  wendet die Funktion `(a -> b -> b)` mit dem übergebenen Element von rechts nach links an

```
foldl (-) 10 [1..5]
10-1-2-3-4-5 = -15

foldr (-) 10 [1..5]
1-(2-(3-(4-(5-10)))) = -7
```

Listen API

Abschnitts-Funktionen

- `take :: Int -> [a] -> [a]`
  nimmt die ersten *n* Elemente
- `drop :: Int -> [a] -> [a]`
  löscht die ersten *n* Elemente
- `span :: (a -> Bool)-> [a] -> ([a], [a])`
  spaltet eine Liste in zwei Teile
- `group :: Eq a =>[a] -> [[a]]`
  gruppiert gleiche Elemente zu eigenen Listen in einer Liste
- `inits :: [a] -> [[a]]`
  gibt alle möglichen Restlisten zurück
- `tails :: [a] -> [[a]]`
  gibt alle möglichen Startlisten zurück

Listen API

Such-Funktionen

- `elem :: Eq a =>a -> [a] -> Bool`
  prüft, ob enthalten
- `lookup :: Eq a =>a -> [(a, b)] -> Maybe b`
  gibt den Value einer Key-Value-Liste
- `find :: (a -> Bool)-> [a] -> Maybe a`
  gibt das erste Element zurück, das die Bedingung erfüllt
- `filter :: (a -> Bool)-> [a] -> [a]`
  filtert die Liste (Positiv-Filter)
- `partition :: (a -> Bool)-> [a] -> ([a], [a])`
  teilt eine Liste in zwei Listen mit „erfüllt“ und „erfüllt nicht“

Listen API

ZIP-Funktionen

- `zip :: [a] -> [b] -> [(a, b)]`
  bildet Tupel aus den Elementen, Anzahl ist beschränkt nach der kürzesten
- `zipWith :: (a -> b -> c)-> [a] -> [b] -> [c]`
  statt Tupel zu bilden werden die Elemente an die Funktion übergeben
- `unzip :: [(a, b)] -> ([a], [b])`
  statt Tupel zu bilden werden die Elemente an die Funktion übergeben

Listen API

```
zip [42..] [5..10]
[(42,5),(43,6),(44,7),(45,8),(46,9),(47,10)]

zipWith (*)  [42..] [5..10]
[210,258,308,360,414,470]

unzip [(x, x^x) | x <- [3..5]]
([3,4,5],[27,256,3125])
```

Danke

Vielen Dank für die Aufmerksamkeit und das Interesse!