Funktionales Programmieren

Teil 2

Carl Philipp Reh

Universität Siegen

24. November 2023

Lambda-Funktionen

Betrachten wir folgende Funktion `filter`, wobei `filter f l` genau die Elemente `x` von `l` behält, für die `f x == True` gilt:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Man kann dann die geraden Zahlen filtern über

```
filterEven :: [Int] -> [Int]
filterEven = filter (\x -> mod x 2 == 0)
```

Hier ist `\x -> mod x 2 == 0` eine *Lambda-Funktion*, was eine Funktion ohne Namen ist. Diese ist gleichbedeutend mit der benannten Funktion

```
even x = mod x 2 == 0
```

Lambda-Funktionen können Code übersichtlicher machen.

Let-Ausdrücke

Die Syntax `let x = y in z` bindet den Wert `y` an den Namen `x`, welcher in `z` benutzt werden kann. Betrachten wir eine Implementierung von **`filter`**:

```
filter f l = case l of
  [] -> []
  (x : xs) -> if f x
              then (x : filter f xs)
              else filter f xs
```

Wir sehen, dass der Teilausdruck `filter f xs` mehrmals vorkommt. Dies können wir vermeiden, indem wir schreiben

```
  (x : xs) -> let rest = filter f xs
              in if f x then (x : rest)
                 else rest
```

Data-Deklarationen

Data-Deklarationen werden benutzt, um „eigene“ Typen zu definieren. Einer der einfachsten Typen ist `Bool`:

```
data Bool = True | False
```

Dies definiert:

- Dass `Bool` ein Typ ist. Dies nennt man auch einen *Typkonstruktor*.
- Dass `True` und `False` Werte vom Typ `Bool` sind. Diese nennt man auch *Wertkonstruktoren*.

Die `and`-Funktion kann man definieren über

```
and :: Bool -> Bool -> Bool
and x y = case x of
            False -> False
            True -> y
```

Rekursive Data-Deklarationen

Rekursive Data-Deklarationen sind solche, die in ihrer Definition auf sich selbst verweisen. Zum Beispiel kann man die natürlichen Zahlen wie folgt definieren:

```
data Nat = Zero | Succ Nat
```

Dies definiert:

- Den Typkonstruktor `Nat`.
- Den Wertkonstruktor `Zero` vom Typ `Nat`.
- Den Wertkonstruktor `Succ` vom Typ `Nat -> Nat`.

Zum Beispiel steht `Zero` für `0` und `Succ (Succ Zero)` für `2`. Die Vorgängerfunktion kann man definieren über

```
pred :: Nat -> Nat
pred x = case x of
  Zero -> Zero
  Succ y -> y
```

Data-Deklarationen mit Parametern

Data-Deklarationen können auch Typparameter erhalten. Beispiel: Um einen Wert „optional“ zu machen, bietet Haskell den `Maybe`-Typ:

```
data Maybe a = Nothing | Just a
```

Dies definiert:

- Den Typkonstruktor `Maybe`. Für jeden Typ `t` ist `Maybe t` auch ein Typ.
- Den Wert-Konstruktor `Nothing` vom Typ `Maybe a`.
- Den Wert-Konstruktor `Just` vom Typ `a -> Maybe a`.

Beispiel:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv x y = if y == 0 then Nothing
              else Just (div x y)
```

Eigene Definition von Listen

Listen sind in Haskell vordefiniert und verwenden spezielle Syntax. Wir können aber auch unsere eigenen Listen definieren:

```
data List a = Nil | Cons a (List a)
```

Dies definiert:

- Den Typkonstruktor `List`. Für jeden Typ `t` ist `List t` auch ein Typ.
- Den Wertkonstruktor `Nil` vom Typ `List a`.
- Den Wertkonstruktor `Cons` vom Typ `a -> List a -> List a`.

Listen in Haskell verwenden `[a]` statt `List a` sowie `x : y` statt `Cons x y` und `[]` statt `Nil`.

Kinds

Typen erhalten ebenfalls „Typen“, welche *Kinds* heißen.
Alle „normalen“ Typen wie `Int`, `Int -> Int`, usw. bekommen Kind `*`, was wir als `Int :: *` und `Int -> Int :: *` schreiben.
Typkonstruktoren, die noch einen Typ erwarten, bekommen Kind `* -> *`. Zum Beispiel gilt `List :: * -> *`. Ein Typkonstruktor, der zwei Typen erwartet, hat Kind `* -> * -> *`, usw.
Es gibt auch „Kinds höherer Ordnung“, zum Beispiel `Fix :: (* -> *) -> *`, das wir möglicherweise später kennenlernen werden.

Typklassen

Eine Typklasse legt eine Menge von Funktionen fest, die von einem Typ implementiert werden können. Zum Beispiel wird die Typklasse `Show` benutzt, um einen Typ „auszudrucken“:

```
class Show a where
  show :: a -> String
```

Um die Typklasse `Show` für den Typ `t` verwenden zu können, muss man `Show t =>` an den Funktionstyp schreiben, zum Beispiel:

```
print :: Show a => a -> String
print x = "The␣value␣is␣" ++ show x
```

`++` ist hier Stringkonkatenation.
Eine mögliche Implementierung für `Bool` sähe wie folgt aus:

```
instance Show Bool where
  show x = if x then "True" else "False"
```

Typklassen

Man kann sich die Implementierung von

```
print :: Show a => a -> String
print x = "The␣value␣is␣" ++ show x
```

folgendermaßen vorstellen:

```
print :: (a -> String) -> a -> String
print show x = "The␣value␣is␣" ++ show x
```

Wenn `print` mit `a = Bool` aufgerufen wird, wird das `show` von der `Bool`-Instanz an `print` übergeben.

Vermeiden expliziter Rekursion

Funktionen auf rekursiven Daten wie Listen sind oft selbst rekursiv implementiert. Die Definition von diesen kann man meist durch Hilfsfunktionen, die Rekursion kapseln, vereinfachen. Ein Beispiel hierfür ist `foldr` für Listen:

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

Beispielsweise kann man alle Werte einer Liste aufsummieren über

```
sum :: [Int] -> Int
sum l = foldr (+) 0 l
```

Hier muss man den Operator + in Klammern schreiben, um die Funktion `(+) :: Int -> Int -> Int` zu erhalten.

Vermeiden expliziter Rekursion

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

sum :: [Int] -> Int
sum l = foldr (+) 0 l
```

Als Beispiel erhalten wir

```
sum (3 : (5 : []))
  = foldr (+) 0 (3 : (5 : []))
  = (+) 3 (foldr (+) 0 (5 : []))
  = (+) 3 ((+) 5 (foldr 0 []))
  = (+) 3 ((+) 5 0)
  = 8
```