Funktionen höherer Ordnung

Übungsblatt 3

Eric Kunze — 24. April 2021

Keine Garantie auf Vollständigkeit und/oder Korrektheit!

Funktionen höherer Ordnung sind Funktionen, die selbst wieder Funktionen als Argumente oder Ergebnisse besitzen. Steckt man ein wenig tiefer in der Mathematik, dann erkennt man Funktionale und Operatoren als "Funktionen höherer Ordnung". Ein relativ einfaches Beispiel, das man auch als Normalsterblicher verstehen kann ist der Differentialoperator

$$D: \quad C^1(\mathbb{R}) \to C^0(\mathbb{R}), \quad f \mapsto f'$$

der eine Funktion auf ihre Ableitungsfunktion abbildet. Das Argument ist eine Funktion und als Resultat erhält man wieder eine Funktion: D ist eine Funktion höherer Ordnung.

Genug der Mathematik, schauen wir uns drei Funktionen höherer Ordnung an:

(1) Wir wollen eine Funktion auf alle Elemente einer Liste anwenden. Dies geschieht durch die Funktion `map`, die wie folgt definiert ist:

```
1  map :: (Int -> Int) -> [Int] -> [Int]
2  map f []     = []
3  map f (x:xs) = f x : map f xs
```

Schauen wir uns einmal den Typ der Funktion an:

$$\texttt{map :: } \underbrace{\texttt{(Int -> Int)}}_{\text{Funktion f}} \texttt{ -> } \underbrace{\texttt{[Int]}}_{\text{Liste (x:xs)}} \texttt{ -> } \underbrace{\texttt{[Int]}}_{\text{Ergebnis}}$$

`f` ist also selbst eine Funktion `Int -> Int`, die ein Listenelement und dieses verändert.

(2) Wir wollen aus einer Liste nur bestimmte Elemente behalten. Dazu definieren wir uns ein sogenanntes Prädikat `p :: Int -> Bool`, welches entscheidet, ob ein Element behalten werden soll (`True`) oder nicht (`False`). Dieses Prädikat (das selbst ja eine Funktion ist) übergeben wir an die Funktion `filter` mitsamt der zu bearbeitenden Liste. `filter p xs` ist dann also die Liste aller Elemente, die `p` erfüllen. Schauen wir uns an, wie `filter` definiert werden kann:

```
1  filter :: (Int -> Bool) -> [Int] -> [Int]
2  filter _ []     = []
3  filter p (x:xs)
4    | p x       = x : filter p xs
5    | otherwise =     filter p xs
```

Erneut betrachten wir kurz den Datentyp der Funktion:

```
filter :: (Int -> Bool) ->  [Int]  -> [Int]
          Prädikat p        Liste (x:xs)  Ergebnis
```

(3) Die dritte Funktion im Bunde soll uns eine Liste falten, d.h. immer zwei Elemente miteinander verknüpfen. Wir betrachten hier die Faltung von rechts mit `foldr`. Es gibt auch die Faltung von links mit `foldl`; die betrachten wir unten in Aufgabe 3 näher. Die Faltung beginnen wir mit einem Startwert `a` und anschließend verknüpfen wir von rechts beginnend immer ein Element der Liste mit dem vorherigen Ergebnis unter Nutzung einer Verknüpfungsfunktion `f`. In Haskell sieht das wie folgt aus:

```
1  foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int
2  foldr _ a []     = a
3  foldr f a (x:xs) = f a (foldr f a xs)
```

Betrachtung des Datentyps ergibt:

```
foldr :: (Int -> Int -> Int) ->  Int  ->  [Int]  -> Int
         Verknüpfungsfunktion f  Startwert a  Liste (x:xs)  Ergebnis
```

Wir erhalten in der Auswertung also zunächst einen Rekursionsbaum, der nach rechts wächst. Anschließend werten wir dann die Funktionsaufrufe von hinten beginnen aus.

Aufgabe 2

Aufgabe. In der Vorlesung wurden die Higher-Order-Funktionen

```
map    :: (Int -> Int)  -> [Int] -> [Int]
filter :: (Int -> Bool) -> [Int] -> [Int]
foldr  :: (Int -> Int -> Int) -> Int -> [Int] -> Int
```

vorgestellt. Implementieren Sie eine Funktion `f :: [Int] -> Int` mithilfe von `map`, `filter` und `foldr`, die das Produkt der Quadrate der geraden Zahlen in der Eingabeliste berechnet.
Hinweis: In Haskell sind die Funktionen `map`, `filter` und `foldr` bereits implementiert, Sie können sie ohne eigene Definition verwenden.

Wir überlegen uns wie wir die einzelnen Bestandteile der Aufgabe den Higher-order-functions) zuordnen:

(1) Wir brauchen alle geraden Zahlen. Um auf Geradzahligkeit zu testen gibt es die vorimplementierte `even`-Funktion. Alle geraden Elemente einer Liste `xs` zu extrahieren deutet auf die `filter`-Funktion hin mit dem Prädikat `even`. Also erhalten wir die Liste gerader Zahlen als

```
filter even xs
```

(2) Wir wollen diese (geraden) Elemente quadrieren. Dafür nehmen wir den Operator `^2` und mache daraus eine Funktion, indem wir Klammern drumherum setzen. Alternativ können wir auch einen anonyme (namenslose) Funktion notieren mit `\x -> x*x`. Eine Funktion auf alle Elemente einer Liste anwenden schreit nach `map`. Dementsprechend wenden wir `map` mit der Funktion `(^2)` auf die erhaltene Liste aus dem ersten Schritt an, also auf `filter even xs` und erhalten als Quadrate aller gerade Elemente die Liste

```
map (^2) (filter even xs)
```

oder

```
map (\x -> x*x) (filter even xs)
```

(3) Nun wollen wir noch das Produkt dieser Elemente bilden. Dafür eignet sich `foldr`, weil wir alle Listenelemente zusammenfalten wollen. Dafür brauchen wir eine Verknüpfungsfunktion, wie wir die einzelnen Elemente verknüpfen wollen, das ist hier die Multiplikation und die Funktion dafür notieren wir mit Klammern drumherum als `(*)`. Zusätzlich brauchen wir einen Startwert, mit dem wir die Operation beginnen und weil wir nichts vorausgesetzt haben nehmen wir das neutrale Element, also die `1`. Und nun fehlt noch die zu faltende Liste: die erhalten wir aus Schritt 2 als `map (^2)(filter even xs)`. Somit können wir das Produkt der Quadrate aller geraden Zahlen einer Liste `xs` schreiben als

```
foldr (*) 1 (map (^2) (filter even xs))
```

und weil das schon unsere Funktion `f` machen soll, gilt also

```
1 f :: [Int] -> Int
2 f xs = foldr (*) 1 (map (^2) (filter even xs))
```

Aufgabe 3

Aufgabe. Geben Sie eine Funktion `foldleft :: (Int -> Int -> Int) -> Int -> [Int] -> Int` an, so dass für jedes `f :: Int -> Int -> Int` und a_0`:: Int`, b_1`, ..., `b_k `:: Int`, `k` $\in \mathbb{N}$ gilt, dass

$$\texttt{foldleft f } a_0 \texttt{ [}b_1\texttt{ , ..., }b_k\texttt{] = f (f ...(f }a_0\ b_1\texttt{) ...}b_{k-1}\texttt{) }b_k$$

also z.B.

```
foldleft (+) 5 [1, 4, 3] = (+) ((+) ((+) 5 1) 4) 3 = ((5 + 1) + 4) + 3
```

Insbesondere soll `foldleft f a [] = a` gelten.

Hier sollen wir die Funktion `foldleft` uns ansehen, die eine Liste nun von links beginnend faltet (im Gegensatz zu foldr, wo von rechts gefaltet wird). *Hinweis:* `foldleft` ist als `foldl` schon vorimplementiert, wir sollen uns hier also nochmal überlegen wie das ganze funktioniert.

Dazu folgende Ausgangssituation: `foldleft` nimmt wieder eine (Verknüpfungs-)Funktion `f`, einen Startwert `a` und eine Liste als Argumente entgegen. Also erhalten wir den Funktionstyp

$$\texttt{foldleft :: } \underbrace{\texttt{(Int -> Int -> Int)}}_{\text{Funktion}} \texttt{ -> } \underbrace{\texttt{Int}}_{\text{Start}} \texttt{ -> } \underbrace{\texttt{[Int]}}_{\text{Liste}} \texttt{ -> } \underbrace{\texttt{Int}}_{\text{Ergebnis}}$$

Nehmen wir an, dass die Liste mindestens ein Element habe und daher die Struktur `(x:xs)`. Nun beginnen wir die Faltung von links und verknüpfen den Startwert mit dem ersten Element unserer Liste, also dem `x`. Wir erhalten als Ergebnis `f a x`. Dann müssen wir dieses Ergebnis weiter mit dem Rest `xs` verfalten. Dazu können wir doch einfach das gerade erhaltene Ergebnis `f a x` als neuen Startwert verwenden und rufen `foldleft` rekursiv auf der Restliste auf. Das sieht dann so aus:

$$\texttt{foldleft f a (x:xs) = foldleft } \underbrace{\texttt{f}}_{\text{Funktion}} \underbrace{\texttt{(f a x)}}_{\text{Start}} \underbrace{\texttt{xs}}_{\text{Restliste}}$$

Der zugehörige Basisfall ist nicht so einfach zu sehen und daher explizit angegeben mit

```
foldleft f a [] = a
```

Das heißt also wir machen nichts und geben nur unseren Startwert zurück. Damit erhalten wir also die ganze Funktion als:

```
1 foldleft :: (Int -> Int -> Int) -> Int -> [Int] -> Int
2 foldleft f a []     = a
3 foldleft f a (x:xs) = foldleft f (f a x) xs
```