HIGHER-ORDER FUNCTIONS UND AUSWERTUNGSSTRATEGIEN IN HASKELL

EIKE ERDMANN 334612

FREDERIC KEHREIN 343928

Literatur

[1] Richard Bird. *Thinking functionally with Haskell*. Cambridge University Press, 2014.
[2] Jürgen Giesel. *Funktionale Programmierung*. 2016.
[3] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

Inhaltsverzeichnis

Date: 24. April 2017.

1. Einleitung

Higher-order functions sind ein wesentlicher Bestandteil der funktionalen Programmierung in Haskell und bilden die zentrale Achse, die Funktionen und Datentypen in Haskell aufspannt.
Das volle Potential dieser Konstrukte wird erst durch Haskells lazy-evaluation Auswertungsstrategie vollständig ausgeschöpft, welche das effiziente Arbeiten sowohl auf endlichen wie auch auf unendlichen Datenobjekten ermöglicht. Im Folgenden werden wir uns mit ihrer Syntax, Eigenschaften, Vorteilen und Nutzung beschäftigen.

2. Funktionen in Haskell

2.1. **Funktionen allgemein.** Bevor wir uns den higher-order functions widmen, wollen wir zunächst einmal grundlegend Funktionen und deren Funktionsweise in Haskell wiederholen.

Haskell ist syntaktisch sehr nah an der formalen mathematischen Notation. So wird aus der Funktion

$$dbl : \mathbb{Z} \to \mathbb{Z}$$
$$dbl(x) = x + x$$

in Haskell die Funktion

```
dbl :: Int -> Int
dbl x = x + x
```

Die erste Zeile stellt dabei die Typen der Funktion dar, welche den Mengen in der Mathematik entsprechen. **Int** bezeichnet die Menge der Integer, den ganzen Zahlen. Mit der zweiten Zeile wird die Abbildungsvorschrift angegeben. Bei einer Eingabe einer ganzen Zahl x ist das Resultat die Summe aus $x + x$.

Funktionen, die mehr als ein Argument übergeben bekommen, sind in der Mathematik über das Kartesische Produkt verschiedener Mengen realisiert. Das Argument ist dann ein Tupel der einzelnen Werte. Auch dieses Konzept übernimmt Haskell:

```
plus :: (Int, Int) -> Int
plus (x, y) = x + y

plus (2,3)
5
```

Hierbei bezeichnet x das erste Element aus dem Tupel und y das Zweite.

Da man mit dieser Notation wegen der Klammern relativ schnell den Überblick verlieren kann, bietet Haskell eine weitere Möglichkeit mehrstellige Funktionen zu realisieren. Eine andere plus-Funktion sieht so aus:

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

plus 2 3
5
```

Dieses Verfahren, die Aufteilung von Tupeln in Folgen von Argumenten, ist für jede Funktion, die Tupel als Eingabe nimmt, möglich und bezeichnet man als *currying*.

2.2. **Higher-order functions.** In Haskell sind Funktionen, genauso wie Tupel oder Integer, Datenobjekte, welche auch als Argument oder Funktionswert einer Funktion verwendet werden können. Eine Funktion, deren Argument oder Funktionswert eine andere Funktion ist, bezeichnet man als *higher-order function*.
Technisch können Funktionen in Haskell nur ein- oder nullstellig sein[3]. Damit currying dennoch möglich ist, werden higher-order functions verwendet.

Unser Beispiel, die plus-Funktion, hat die Typdefintion

```
plus :: Int -> Int -> Int
```

Der $->$ Operator assoziiert nach rechts, d.h. unsere Definition entspricht geklammert:

```
plus :: Int -> (Int -> Int)
```

So lässt sich schnell erkennen, dass unsere Funktion eigentlich einen **Int** übergeben bekommt und eine Funktion (**Int** -> **Int**) zurückgibt.

Für das Aufrufen von plus sind also folgende Schreibweisen äquivalent:

```
--1.
plus 2 3
--2.
(plus 2) 3
```

Dass wir durch Eingabe nur eines Parameters eine Funktion zurück erhalten, können wir nutzen, um neue Funktionen zu definieren. Durch die Eingabe des Parameters 1 erhalten wir eine neue Funktion, die mathematische Nachfolgefunktion succf.

```
succf :: Int -> Int
succf = plus 1
```

Nun können higher-order functions nicht nur Funktionen ausgeben, sondern auch als Argument verwenden. So kann man zum Beispiel eine Funktion konstruieren, welche eine Liste und eine Funktion entgegen nimmt und die Funktion auf jedes Listenelement anwendet

```
mapf:: (a -> b) -> [a] -> [b]
mapf _ [] = []
mapf f (x:xs) = f x : mapf f xs
```

Falls eine leere Liste vorliegt geben wir eine leere Liste zurück. Falls eine nichtleere Liste vorliegt wenden wir die übergebene Funktion auf den Anfang der Liste an und rufen mapf rekursiv auf dem Rest der Liste auf.

Damit können wir, wie bereits bekannt, durch Einsetzen von Argumenten neue Funktionen bilden. So z.B. die Funktion succlist

```
succlist :: [Int] -> [Int]
succlist xs = mapf succf xs
```

Analog konstruieren wir eine Funktion filterf, die mit einer Funktion des Typs **a** -> **Bool** entscheidet, ob ein Element in die Ergebnisliste kommt oder nicht.

```
filterf::(a -> Bool) -> [a] -> [a]
filterf _ [] = []
filterf f (x:xs) = | f x = x : filterf f xs
                   | otherwise = filterf f xs
```

Diese können wir z.B. nutzen, um alle Elemente einer Liste zu extrahieren, die kleiner 5 sind:

```
filterf (< 5) [1..10]
[1, 2, 3, 4]
```

2.3. **Vordefinierte Funktionen.**

2.3.1. *map und filter.* Wir haben oben die Funktionen mapf und filterf als Beispiele für higher-order functions eingeführt. Tatsächlich sind diese, aufgrund ihrer generischen Eigenschaften und den daraus resultierenden breitgefächerten Einsatzgebieten, bereits im Haskellstandard enthalten als

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

Neben diesen gibt es im Haskellstandard auch noch viele weitere grundlegende higher-order functions, einige davon wollen wir nun vorstellen.

2.3.2. *fold und scan.* Diese Funktionen führen kaskadierende Berechnungen auf den Elementen einer Liste aus. Verwendet wird dabei ein Akkumulator und eine Funktion, welche auf jedes Element der Liste zusammen mit dem Akkumulator angewandt wird, um den neuen Akkumulator für das nächste Element zu berechnen. Hierbei unterscheidet man zwischen den Funktionen fold und scan. fold berechnet nur das Endergebnis. scan liefert eine Liste mit jedem Element als Zwischenergebnis nach einem Schritt. Außerdem wird unterschieden, ob man von Rechts oder von Links berechnet, da natürlich nicht jede Funktion kommutativ ist. Die links- und rechtsseitige Funktion fold sieht wie folgt aus

```
-- Rechtsseitig
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr _ y [] = y
foldr (f) (y) (x:xs) = f x foldr (f) y xs

--Linksseitig
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ y [] = y
foldl (f) (y) (x:xs) = foldl (f) (f y x) xs
```

Die scan Funktionen funktionieren entsprechend ähnlich. Die Signatur lautet

```
-- Linksseitig
scanl :: (a -> b -> a) -> a -> [b] -> [a]

--Rechtsseitig
scanr :: (a -> b -> a) -> a -> [b] -> [a]
```

Als Vereinfachung gibt es auch noch die Funktionen foldl1 und foldr1. Diese verwenden einfach das erste Element der Liste als Akkumulator und rufen dann fold mit diesem auf dem Rest der Liste auf:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f (x:xs) = foldr f x xs
```

Da diese Funktionen allerdings ein Element in der Liste erwarten, führen sie auf leeren Listen zu einer runtime-exception.

2.3.3. *(.) - Der period Operator.* Wie bereits bekannt orientiert sich Haskell stark an der Mathematik und ihrer Notation. So gibt es für die aus der Mathematik bekannte Verknüpfung von Funktionen

$$(f \circ g)(x) = f(g(x))$$

in Haskell einen vordefinierten (.) Operator

```
infixr 9
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f(g x)
```

2.3.4. *Listenkomprehension.* Haskells Listenobjekte orientieren sich an der mathematischen Mengennotation. Neben typischen Listenoperatoren wie die Vereinigung oder Schnitt, welche bereits aus der Programmiervorlesung bekannt sind, gibt es auch die so genannte Listenkomprehension. Hierbei wird sich der beschreibenden Definition einer Menge aus der Mathematik bedient, um Mengen bedingt zu erzeugen.

In der Mathematik bezeichnet der Ausdruck

$$f : \mathbb{U} \to \mathbb{V} : x \mapsto f(x) \tag{1}$$

$$\{\, f(x) \mid x \in \mathbb{U},\ \mathscr{A} \,\} \tag{2}$$

die Menge aller Funktionswerte f(x) von allen Zahlen x aus der Menge $\mathbb{U}$, welche die Bedingung $\mathscr{A}$ erfüllt.

Haskell stellt hierfür die folgenden Notation bereit:

```
[ f x | x <- u, a]
```

Ein konkretes Beispiel wäre damit

```
sixEvens :: [Int]
sixEvens = [ 2*x | x <- [1..10], x <= 6]

sixEvens
[2, 4, 6, 8, 10, 12]
```

Diese Mengennotationen sind sehr gebräuchlich in Haskell. Allerdings sind die darin enthaltenen Möglichkeiten auch durch Kombinationen mit map und filter repräsentierbar. Die linke Seite ist eine Ausführung von map, die rechten Seiten multiple Anwendungen der filter Funktion.

So ist es möglich, das obere Beispiel umzuschreiben zu

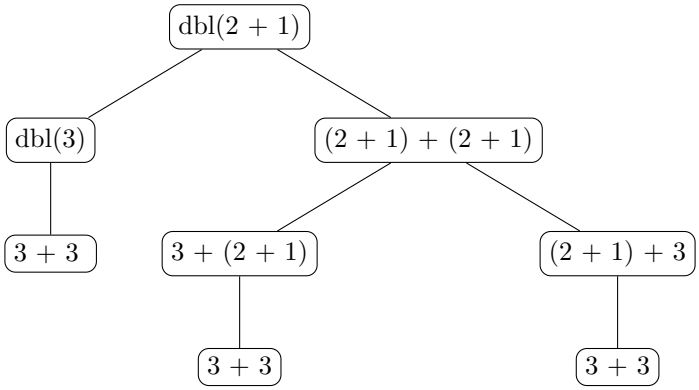
```
map (*2) (filter (<= 6) [1..10])
```

Die Listenkomprehension ist allerdings deutlich lesbarer und bei der Verwendung von mehreren Filtern auch deutlich kürzer.

3. Lazy-evaluation und Unendliche Objekte

3.1. **Lazy-evaluation.** Um zu verstehen, wie Funktionen in Haskell ausgewertet werden, muss man die Auswertungsstrategie von Haskell betrachten.

Zunächst betrachten wir die Auswertungsmöglichkeiten eines einfachen Ausdruckes. Der Ausdruck dbl(2+3) kann auf verschiedenen Wegen ausgewertet werden, welche durch den folgenden Baum illustriert werden:



Grundsätzlich unterscheidet man die strikte Auswertung (der linke Pfad) und die nicht-strikte Auswertung (die beiden rechten Pfade). Haskell verwendet eine nicht strikte "leftmost-outermost" Strategie. Dies bedeutet, dass Ausdrücke, welche außen stehen, zuerst ausgewertet werden und im Falle einer Wahl zwischen der Vereinfachung eines linken und eines rechten Ausdrucks stets der linke ausgewählt wird. Die Auswertung terminiert, sobald ein Ergebnis gefunden wurde, selbst wenn noch nicht jeder Teilausdruck ausgewertet wurde.
Um die Performance zu verbessern speichert Hakskell Ergebnisse von bereits berechneten Ausdrücken. So kann in unserem Beispiel der Ausdruck $3 + (2 + 1)$ ohne Rechenaufwand zu $3 + 3$ aufgelöst werden, da $2 + 1$ bereits ausgewertet wurde.
Diese leftmost-outermost Stragie mit vorzeitigem Terminieren bezeichnet man als lazy-evaluation.

Wie bereits erwähnt kann Haskell die Auswertung vorzeitig abbrechen. Dies ist besonders für das *pattern matching* relevant, da Haskell den Ausdruck nur soweit auswerten muss, bis das erste passende pattern gefunden wurde, danach bricht Haskell die Auswertung ab und berechnet das Ergebnis für das gefundene pattern.

Wir definieren nonTerm als nicht terminierende Funktion und idX als zweistellige Funktion, die ihr erstes Argument zurückgibt.

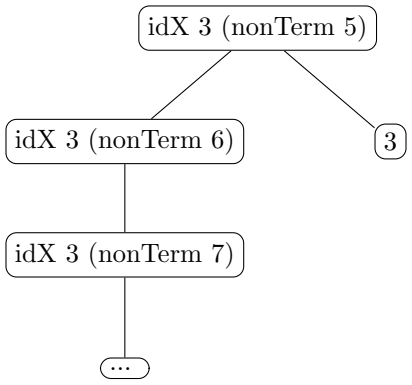
```
nonTerm :: Int -> Int
nonTerm x = nonTerm (x + 1)

idX :: Int -> Int -> Int
idX x y = x
```

Das folgende Beispiel illustriert die Vorteile von Haskells Auswertungsstrategie:

```
idX 3 (nonTerm 5)
3
```

Der Auswertungsbaum dieses Ausdrucks ist



Die strikte Auswertungsstrategie auf dem linken Zweig terminiert nicht, da zunächst der Ausdruck (nonTerm 5) aufgelöst werden müsste, welcher ebenfalls nicht terminiert.
Durch die lazy-evaluation Strategie erkennt Haskell allerdings bereits nach der ersten Auflösung von idX das pattern und für dessen Berechnung ist die Auswertung von nonTerm nicht notwendig.

3.2. **Unendliche Objekte.** Haskell unterstützt nicht terminierende Rekursion, die durch Haskells Auswertungsstrategie nicht immer vollständig aufgelöst werden muss. Damit lassen sich unendliche Datenobjekte bilden.
Die bekannteste Form eines unendlichen Datenobjektes sind unendliche Listen. Andere Datenobjekte wollen wir hier nicht betrachten.

3.2.1. *Unendliche Listen.* Listen sind in Haskell rekursiv definiert durch 2 Konstruktoren:

[ ]	Konstruiert eine leere Liste.
(:) a $[a]$	Konstruiert eine Liste, an die ein Element vorne angehängt wird.

Man kann z.B ein Abbild der natürlichen Zahlen $\mathbb{N}$ in Haskell als unendliche Liste konstruieren.

```
infList :: Int -> [Int]
infList n = n:infList (n+1)

naturals :: [Int]
naturals = infList 1
```

Die Funktion infList generiert beim Aufruf eine unendlich Liste von aufeinander folgenden ganzen Zahlen beginnend mit dem übergebenen Argument. Diese nutzen wir dann, um die natürlichen Zahlen $\mathbb{N}$ durch die nullstellige Funktion `naturals` zu konstruieren.

Wir wollen außerdem die Menge der Fibonacci-Zahlen konstruieren.

```
fibGen :: Int -> Int -> [Int]
fibGen a b = (a+b):(fibGen b (a+b))

fib :: [Int]
fib = fibGen 1 0
```

Ein Aufruf von fib würde nicht terminieren.
Man kann allerdings mithilfe von "Extraktionsfunktionen" endliche Teile der Liste gesondert betrachten.
take und takeWhile. Wollen wir eine feste Anzahl Elemente ab Anfang der Liste extrahieren so benutzen wir **take**.
take hat die Signatur

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take y (x:xs) = x : (take (y-1) xs)
```

Die ersten 10 Fibonacci-Zahlen können wir dann so herausfinden:

```
take 10 (fib)
[1,1,2,3,5,8,13,21,34,55]
```

Wollen wir Elemente aus der Liste extrahieren, solange eine Bedingung erfüllt ist, so benutzen wir **takeWhile**.
takeWhile hat die Signatur

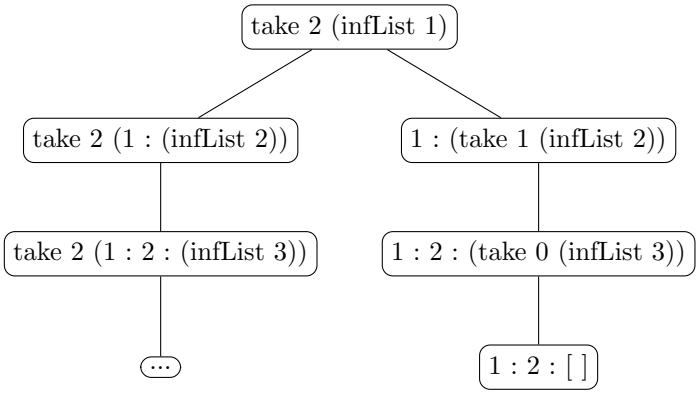
```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile f [] = []
takeWhile f (x:xs) | f x = x : (takeWhile f xs)
                   | otherwise = []
```

Alle Fibonacci-Zahlen < 100 können wir dann so herausfinden:

```
takeWhile (<100) (fib)
[1,1,2,3,5,8,13,21,34,55,89]
```

3.2.2. *Lazy-evaluation auf unendlichen Datenobjekten.* Mit der lazy-evaluation Strategie lässte sich auch erklären, weshalb Haskell unendliche Datenobjekte auswerten kann.

Wir betrachten das Beispiel **take 2 (infList 1)**.
Dies wird illustriert durch den folgenden Baum. Der linke Pfad entspricht der strikten Auswertung und der rechte Haskells Strategie:



Die strikte Auswertungsstrategie links würde nicht terminieren, weil der unendliche Ausdruck infList ausgewertet werden müsste.
Die nicht strikte Strategie rechts kann allerdings terminieren, indem es take zuerst auswertet und nach der dritten Auflösung das *take* 0 _ pattern erkennt.

3.2.3. *Listenkomprehensionen.* Auch Listenkomprehensionen können unendliche Listen verwenden. Allerdings terminiert keine solche Listenkomprehension auf unendlichen Listen.

Wenn wir das Beispiel von oben mit der unendlichen Liste naturals definieren erhalten wir trotz endlicher Ergebnisliste einen nicht terminierenden Ausdruck.

```
nonTermList :: [Int]
nonTermList = [2*x | x <- naturals, x <= 6]

sixEvens == (take 6 nonTermList)
True
```

Folgender Aufruf terminiert **NICHT**

```
take 7 nonTermList
[1,2,3,4,5,6Interrupted.
--This call was manually interrupted.
```

Will man dies umgehen, so kann man auf geordneten Listen **takeWhile** benutzen:

```
sixEvensNew =
        let xs = takeWhile (<= 6) naturals
        in [ x*2 | x <- xs]
```

4. Zusammenfassung

Higher-order functions und lazy-evaluation Strategie sind ein essentieller Bestandteil des Hakell-Programmierens.
Nach dem Lesen dieses Dokuments sollten sie wissen was higher-order functions sind, wie sie funktionieren, wieso sie hilfreich sind und wie man sie anwendet.

Sie sollten außerdem wissen wie Haskell Ausdrücke auswertet und wie man mit unendlichen Datenobjekten arbeiten kann.