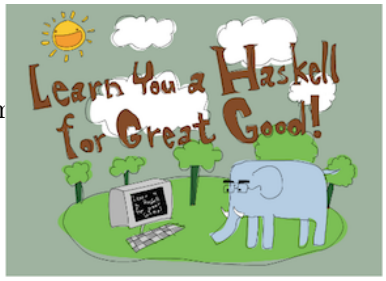
11. Dezember 2017

Funktionale Programmierung mit Haskell

Wichtige Quellen:

- https://www.haskell.org (Was sonst?)
- http://learnyouahaskell.com/learnyouahaskell.pdf
- Simon Thompson: **Haskell: The Craft of Functional Programming**. Addison Wesley, 3rd edition, 2011.
- R. Bird; P. Wadler: **Einführung in die funktionale Programmierung**. München, Hanser 1992.
- Marco Block; Adrian Neumann: **Haskell, Intensivkurs**. Springer Heidelberg Dordrecht London New York, ISBN 978-3-642-04717-6.
- Ernst-Erich-Doberkat: **Haskell - Eine Einführung für jektorientierte**. 2012 Oldenbourg Wissenschaftsverlag Gn ISBN 978-3-486-71417-3.



Einige Konzepte habt ihr in Haskell schon verstanden.

Aufgabe: Nenne mind. 10 Merkmale der funktionalen Programmierung im Allgemeinen und von Haskell im Speziellen. Erkläre!

Aber Haskell kann noch mehr. Es gibt noch weitere, höhere Konzepte, die wir nun betrachten wollen. Grundlage aller funktionaler Programmierung sind Funktionen. Eine Funktion $f :: A \to B$ liefert Resultate vom Typ B in Abhängigkeit der Argumente vom Typ A. f(x) beschreibt das Resultat der Anwendung der Funktion f auf das Argument x. In der funktionalen Programmierung wird strenger zwischen der Funktion f und ihrer Anwendung auf ein Argument f(x) unterschieden als in den meisten mathematischen Ausarbeitungen. Wer funktionale Programmierung betreibt, begreift Funktionen als Werte, die sich durch nichts von anderen Werten unterscheiden. Sie können zum Beispiel als Argumente an andere Funktionen übergeben werden oder selbst Funktionsresultate sein.

1 Currying

Schon bei der Notation der Anwendung einer Funktion f auf ihr Argument x lassen sich durch Verwendung der Schreibweise f x statt f(x) eine Menge eigentlich überflüssiger Klammern im

Code sparen. Dasselbe Ziel wird mit dem Currying (nach dem amerikanischen Logiker Haskell B. Curry) verfolgt. Ansatzpunkte für die Klammerersparnis sind hier strukturierte Funktionsargumente. So besteht zum Beispiel das Argument für die Funktion `plus` aus einem Integer-Paar:

Listing 1: Funktion plus mit Tupeln

```
1 plus :: (Integer, Integer) -> Integer
2 plus (x,y) = x + y
```

In der curried-Version nimmt die Funktion `plusc` die beiden Integer-Werte nicht mehr als Paar, sondern einzeln nacheinander entgegen:

Listing 2: Curry-Version von plus

```
1 plusc :: Integer -> (Integer -> Integer)
2 plusc x y = x + y
```

Genauer gesagt erwartet die Funktion `plusc` einen Integer-Wert `x` als Argument und liefert als Resultat eine Funktion `plusc x`, die wiederum einen Integer-Wert y als Argument erwartet und einen Integer-Wert (die Summe von `x` und y) als Resultat zurückgibt. Dabei tut die Funktion `plusc x` nichts anderes, als `x` zum übergebenen Integer-Argument zu addieren. Man sagt, die Funktion wertet die Argumentliste von links beginnend aus (Linksasoziativität der Auswertung). Das Funktionen Funktionen als Argumente verarbeiten können und selbst Funktionen zurückliefern, ist eine Entwicklung, die heute jede ordentliche Sprache hat, aber in den funktionalen Konzepten von Anfang an vorhanden war.

Neben der Reduzierung der Klammern im Code bietet das Currying einen zweiten Vorteil: Die curried-Version einer Funktion erwartet stets nur ein einfaches Argument und liefert als Resultat eine Funktion, die auch für sich genommen nützlich sein kann. So lässt sich zum Beispiel eine Inkrement-Funktion wie folgt formulieren:

Listing 3: inc

```
1 inc = plusc 1
```

Zu jeder Funktion mit einem strukturierten Argument kann eine curried-Version erstellt werden. Entweder tut das der Programmierer selbst, indem er `plusc x y = x + y` definiert. Es existiert aber auch eine vordefinierte Funktion `curry`, mit deren Hilfe Funktionen mit einem strukturierten Argument in ihre curried-Version konvertiert werden können.
Für die umgedrehte Richtung gibt es auch die Funktion `uncurry` (siehe Referenz).

Listing 4: curry

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
```

Die Funktion `plusc` kann demnach auch durch definiert werden.

Listing 5: plusc in der curried-Version

```
plusc = curry plus
```

Aufgaben:
1. Erkläre wie die Funktion f x y z ausgewertet wird. Notiere die Signatur!

2 Funktionsverkettung

Zu einem guten Programmierstil gehört auch, komplexe Funktionen als Komposition von einfacheren Teilfunktionen aufzubauen. Die Notation für die Komposition zweier Funktionen f und g in Haskell lautet: `(f.g)(x)) = f(g(x)).`
Das erste Klammerpaar ist notwendig, da sonst versucht würde, f mit g(x) zusammenzusetzen. So würde folgende Verkettung inkrementiere.dekrementiere für zwei definierte Funktionen inkrementiere (+1) und dekrementiere (-1) zu folgendem Fehler führen:

Listing 6: Fehler bei der Komposition

```
Prelude> inkrementiere.dekrementiere 5
ERROR - Type error in application
*** Expression     : inkrementiere.dekrementiere 5
*** Term           : dekrementiere 5
*** Type           : Int
*** Does not match : a -> b
```

Aus der Nacheinanderausführung folgt, dass nicht jedes Funktionspaar auf diese weise zusammengefügt werden kann. Der Resultattyp der zuerst angewandten Funktion muss mit dem Argumenttyp der danach angewandten Funktion übereinstimmen. Demzufolge ist die Definition des Operators (.) wie folgt:

Listing 7: Operator (.)

```
1 (.) :: (b -> c) -> (a -> b) -> a -> c
2 f g x = f(g x)
```

Als Abschluss noch den von Theo benutzten *syntaktische Zucker*, bei der Verwendung von Funktionskompositionen mithilfe des Operators ($). Dieser Operator macht bei einfachen Funktionen nichts weiter, d. h. f $ x = f x. Schön wird dies erst bei Verkettungen mehrerer Funktionen:

Listing 8: $ - Operator

```
1 Prelude> f(g(x))
2 ....
3 Prelude> (f.g) x
4 ....
5 Prelude> f.g $ x
6 ....
7 Prelude> f $ g $ x
8 ....
```

Alle Ausdrücke liefern die selben Ergebnisse. Nur eben ohne die Klammern. Somit ist der letzte Ausdruck eben viel schöner. ;-))

3 Höhere Funktionen

Funktionen können nicht nur einfache Datentypen oder Listen als Argumente verarbeiten, sondern auch Funktionen. Dies ist nützlich, wenn immer wiederkehrende Rechenschritte oder Muster im Quelltext auftauchen.

3.1 Mapping (map-Funktion)

Beispiel: Es soll jedes Element einer Liste quadriert und mit 42 multipliziert werden.
Dies kann elegant mit der Funktion `map` gelöst werden.

Listing 9: Definiton von `map`

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

Listing 10: Anwendung von `map`

```
verdoppeln liste = map (2*) liste
-- verdoppeln [2,3,4] = map (2*) [2,3,4] ⇝ (2*3):map (2*)
    [3,4] ⇝ (2*3):(2*3):map (2*) [4] ⇝ (2*3):(2*3):(2*4):map
    (2*) [] ⇝ [4,6,8]
```

Somit kann man nun leicht das oben beschriebene Problem lösen.

Listing 11: `map` mit Verkettung

```
mapListe = map ((*2).quad)
 where quad x = x*x
-- Aufgabe: Schreibe mapListe mit $
```

3.2 filter-Funktion

Listing 12: `filter`-Funktion

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

ToDo

3.3 Faltungen

Listing 13: Faltungen: Signatur und Beispiele

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

sum = foldr (+) 0

and = foldr (&&) True

length = foldr (\a b -> 1 + b) 0
```

4 λ-Kalkül

Bisher haben wir Funktionen definiert, benutzt, verkettet, als Argumente und Rückgabewerte erhalten. Man nennt alle diese Funktionen *benannte* Funktionen. Man kann aber auch *unbenannte oder anonyme* Funktionen durch Abstraktion gewinnen.
Beispiel: 5 * x +1 ist ein Ausdruck. Diesen kann man durch eine λ-Notation zu einer Funktion machen: λ x -> 5 * x +1. Um diese Funktion zu benutzen muss man Argumente auf ihr anwenden. Das geht so: (λ x -> 5 * x +1) 3 ⇝ 16.
Dieser Mechanismus ist z. B. sehr elegant, wenn man mehrstellige Funktionen in höheren Funktionen verwenden möchte.

Listing 14: Beispiele für anonyme Funktionen

```
1 map (\x -> x^2 + 1)  [1..10]
2 filter (\x -> x `mod` 5 == 0)  [1..100]
3 map (\(x,y) -> x+y) (zip [1..10]  [1..10])
```

Aufgaben: 1. x ++ y
2. concat

Und nun noch Theos Beispielprogramm vom Rucksack, welches sehr schön alle diese Konzepte verpackt.

```
import Data.List
import Data.Function (on)

type Index = Int
type Value = Int
type Weight = Int
type Item = (Weight, Value)
type ItemGroup = [Index]
type Solution = (ItemGroup, Value)

example1 :: [Item]
example1 = [(2,6),(3,11),(4,15),(5,23),(6,25),(7,27),(8,35),(9,40),(10,45)]

knapsack :: [Item] -> Weight -> Solution
knapsack objects size = head $ partials objects size

partials :: [Item] -> Weight -> [Solution]
```

```
partials _ 0 = [([], 0)]
partials objects size = bestsolution:previous
    where bestsolution = maximumBy (compare `on` solutionValue) [s1, s2]
          s1 = bestwithsize objects size
          s2 = bestcombination previous size
          previous = partials objects (size-1)

bestwithsize :: [Item] -> Weight -> Solution
bestwithsize objects size
    | null fittingitems = ([], 0)
    | otherwise = ([fst bestitem], itemValue $ snd bestitem)
    where bestitem = maximumBy (compare `on` (itemValue . snd)) fittingitems
          fittingitems = filter ((==size) . itemWeight . snd) $ zip [0..] object

bestcombination :: [Solution] -> Weight -> Solution
bestcombination _ 1 = ([], 0)
bestcombination solutions size = maximumBy (compare `on` solutionValue) $ [soluti

solutionConcat :: (Solution, Solution) -> Solution
solutionConcat (s1, s2) = (solutionItems s1 ++ solutionItems s2, solutionValue s


solutionItems :: Solution -> ItemGroup
solutionItems = fst

solutionValue :: Solution -> Value
solutionValue = snd

itemWeight :: Item -> Weight
itemWeight = fst

itemValue :: Item -> Value
itemValue = snd
```

Aufgaben

Löse die Aufgaben auf verschiedenen Weisen, wenn möglich (mit/ohne Muster, Bibliotheksfunktionen, mit höheren Funktionen, Listengenerator, λ-Ausdruck)

1. Bestimme die Reste bzgl. der Division durch eine Zahl
   `divrest` 5 [1,2,3,4,5,6] $\rightsquigarrow$ [1,2,3,4,0,1]

2. Liste aller Teiler einer Zahl
   `teilerliste` 60 $\rightsquigarrow$ [1,2,3,4,5,6,10,12,15,20,30,60]

3. Liste aller Primzahlen bis zu einer Zahl
   `primliste` 25 $\rightsquigarrow$ [1,2,3,5,7,11,13,17,19,23]

4. Liste der pythagoräischen Tripel, gesucht sind Tripel (a,b,c), a,b,c $\in \mathbb{R}$ mit $a^2 + b^2 = c^2$ bis zu einer Zahl n
   `pythliste` 10 $\rightsquigarrow$ [(3,4,5),(6,8,10)]

5. Was macht folgende Funktion `foo`
   `foo = foldr (\x y -> (x+y)/2) 54 [12,4,10,6]`

6. Bestimme die Anzahl eines bestimmten Elementes in einer Liste
   `wieviele 'b' „Schlabberbacke“` $\rightsquigarrow$ `3`

7. Zu einem Wort ist die Liste aller Teillisten zu erzeugen.
   `potenzliste „Peng“` $\rightsquigarrow$ `[[], „P“, „Pe“, „Pen“, „Peng“]`

8. Implementiere mit `foldr` folgende Funktionen
   - maxList :: [Int] -> Int soll das Maximum einer Liste ermitteln,
   - productList :: [Int] -> Int das Produkt der Listenelemente.
   - orList :: [Bool] -> Bool

9. Sieb des Erathosthenes liefert effizient die Liste aller Primzahlen
   `primliste` = [2,3,5,7,11,13,17,19, ...]

10. Mirp-Zahlen sind jene Primzahlen, die rückwärts gelesen ebenfalls eine Primzahl darstellen. Erzeuge eine unendliche Liste aller Mirp-Zahlen.

11. Die kleinste Zahl, die sich durch die Zahlen 1 bis 10 ohne Rest teilen lässt, ist 2520. Finde die kleinste Zahl, die ohne Rest durch die Zahlen 1 bis 20 teilbar ist.

12. Türme von Hanoi
    Wir haben dieses interessante Problem hinsichtlich ihrer Komplexität untersucht. Eine Lösung lässt sich sehr schnell rekursiv formulieren (und in Haskell programmieren, dank Rekursion). Seien n Scheiben gegeben und die drei Stangen nenne ich a,b,c. Eine Lösung in Pseudocode kann dann wie folgt formuliert werden:

$$hanaoi(i,a,b,c) = \begin{cases} fertig & \text{für } i = 0 \\ hanoi(i-1,a,c,b) & \text{für } i < 0 \end{cases}$$

Führe eine ausführliche Analyse des Pseudocodes durch und entwickle ein Programm.

Weitere Aufgaben folgen!