Aufgaben für die 1. Sitzung

13. August 2017

Umgang mit Funktionen

(1) a. Analog zu **id** :: a → a in den Folien: Was machen die Funktionen
   - f :: a → b → a
   - g :: (a,b) → a und
   - h :: ((a,b) → c) → a → b → c

   zwangsläufig?

   b. Suche in Hoogle nach den Typen und finde heraus, wie f, g und h wirklich heißen.

   c. Implementiere die drei Funktionen. Kannst du f durch eine Kombination von g und h ausdrücken?

(2) Implementiere **flip** :: (a → b → c) → b → a → c. Wofür ist diese Funktion gut?

(3) Implementiere (.) :: (b → c) → (a → b) → a → c. Wofür ist diese Funktion gut?

(4) Zeige im λ-Kalkül, dass **flip** . **flip** === **id**.

Umgang mit Listen

(5) Implementiere die Funktion **length** :: [a] → **Int**.

Welche Laufzeit hat **length**?

(6) Implementiere die Funktion (!!) :: [a] → **Int** → a so, dass 'xs !! i' das i-te Element aus xs liefert (beginnend bei 0).

Welche Laufzeit hat (!!) ? Was bedeutet das für die Übertragung Array-basierter Algorithmen auf Listen?

(7) Implementiere **map** :: (a → b) → [a] → [b]. Hier soll die gegebene Funktion auf jedes einzelne Listenelement angewandt werden.

Welche Typensignatur hat **map flip**? Welche hat **flip map**?

(8) Implementiere **filter** :: (a → **Bool**) →[a] → [a]. Hier sollen alle Elemente entfernt werden, die das gegebene Prädikat nicht erfüllen.

(9) Implementiere **zipWith** :: (a →b → c) → [a] → [b] → [c].

Dabei gilt '**zipWith** f [1,2,3, ...] [a,b,c, ...] === [f 1 a, f 2 b, f 3 c, ...] '.

Implementiere auch den Spezialfall **zip** :: [a] → [b] → [(a,b)], indem du ihn auf **zipWith** zurückführst.

Welche Typensignatur hat **zipWith** (.)? Welche hat **zipWith id**?

(10) Implementiere **reverse** :: [a] → [a]. Hier soll die Liste umgedreht werden.

(11) Implementiere **splitAt** :: **Int** → [a] → ([a],[a]) . Hier soll die Liste nach den ersten n Elementen durchgeschnitten und die beiden Teillisten zurückgegeben werden.

Implementiere darauf aufbauend **take** :: **Int** → [a] → [a], welches von einer Liste nur die ersten n Elemente behält, und **drop** :: **Int** → [a] → [a], welches die ersten n Elemente einer Liste wegwirft.

(12) Implementiere **cycle** :: [a] → [a]. Hier soll die gegebene Liste unendlich oft wiederholt werden. Denke daran, beim Testen **take** zu Verwenden.

(13) Implementiere **iterate** :: (a → a) → a → [a], so dass das i-te Element des Ergebnisses, die i-fache Anwendung der gegebenen Funktion auf den Startwert enthält.

Fibonacci-Zahlen

Die Reihe der Fibonacci-Zahlen beginnt mit 0 und 1, und ist weiter definiert durch

$$a_n := a_{n-2} + a_{n-1}$$

Die ersten Elemente sind also: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

(14) Implementiere analog zur Definition die Funktion fib :: **Int** → **Int**, die auf naive Weise die n-te Fibonacci-Zahl liefert. Welche Laufzeit bekommst du?

(15) Verbessere die Laufzeit auf $\mathcal{O}(n)$ (Tipp[1])

(16) Implementiere die unendliche Liste fibs :: [**Int**] mithilfe von **iterate**.

(17) Implementiere die unendliche Liste fibs ' :: [**Int**] mithilfe von **zipWith**.

(18) Warum sind diese beiden Lösungen besser als die folgende?

[1] Tupel als Rückgabewert, das die n-te *und* die n-1-te Zahl enthält

```
fibs'' :: [Int]
fibs'' = map fib [0..]
```

Fizz Buzz

Fizz Buzz ist ein Spiel, bei dem die Teilnehmer*innen reihum von 1 hochzählen, wobei:

- Zahlen, die durch 3 teilbar sind, durch „fizz“,
- Zahlen, die durch 5 teilbar sind, durch „buzz“ und
- Zahlen, die durch 3 *und* 5 teilbar sind, durch „fizz buzz“ ersetzt werden.

(19) Implementiere die Funktion isFizzBuzz :: **Int** → **String**, die für eine gegebene Zahl herausfindet, was gesagt werden muss. (Tipp: **show** :: **Show** a ⇒a →**String**).

(20) Implementiere die unendliche Liste fizzBuzz :: [**String**], die den gesamten Spielverlauf enthält.

Leseverständnis

„Einfach in ghci eingeben“ ist hier Schummeln ;)

(21) Wie sieht die folgende Liste aus? Wie kann man sie leserlicher implementieren?

```
foo :: [Int]
foo = 1 : map (+1) foo
```

(22) Was macht die folgende Funktion? Wie kann man sie leserlicher implementieren?

```
fuu :: Int → Int
fuu = (!!) $ iterate (*2) 1
```

Pointfree-Notation

Im sogenannten „Pointfree“-Style[2] werden Funktionen als Komposition aus anderen Funktionen geschrieben, möglichst ohne das Argument („point“) explizit zu nennen. Einfaches Beispiel:

```
concatMap :: (a → [b]) → [a] → [b]
-- pointful
concatMap f xs = concat (map f xs)
-- medium
concatMap f = concat . map f
-- pointfree
concatMap = (concat .) . map
```

[2] auch regelmäßig spaßeshalber „Pointless“-Notation genannt

(23) Schreibe die folgende Funktion in Point*ful*-Style um:

```
foo = (sum .) . zipWith (*)
```

Welchen Typ hat foo überhaupt?

(24) Schreibe die folgenden Funktionen in Point*free*-Style um:

```
bar x y z = f (g x y) z
baz (x,y) z = f z (g x y)
buz x y = y
```

Denk an η-Reduktion und folgende Hilfsfunktionen: (.), id, const, flip, curry, uncurry. Weitere wirst du in zukünftigen Sitzungen kennenlernen.