III.6 Funktionale Programmiertechniken

Mittwoch, 19. Dezember 2018 09:00

Jetzt: Zeige 2 Programmiertechniken, die typisch für Funktionale Programmierung sind.

1. Funktionen höherer Ordnung

Funktion höherer Ordnung:
Funktion, die als Argument oder Ergebnis wieder eine Funktion hat.

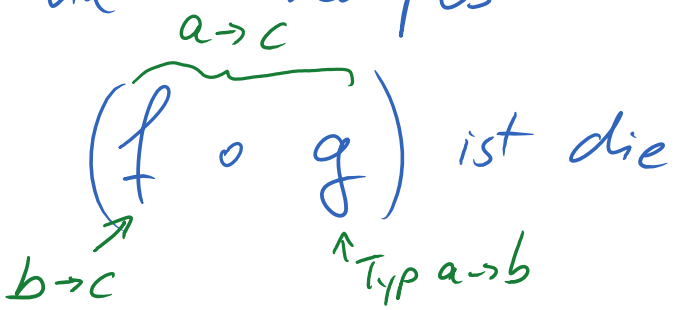
```
square :: Int -> Int
```
ist Funktion erster Ordnung

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

ist Funktion höherer Ordnung, denn das Resultat von plus 1 ist eine Funktion.

Weiteres Bsp:

Funktionskomposition



ist die Funktion, die erst g und dann f ausführt.

comp soll o berechnen.

Ist in Haskell vordefiniert als Infix-Operator.

Wenn square und half schon definiert wurden, dann berechnet

comp half square

die Funktion, die Zahlen erst quadriert und dann halbiert.

(comp half square) $4 = \frac{4^2}{2} = 8$

(half . square) $4 = 8$

Fkt. höherer Ordnung als Programmiertechnik:

Verwende feste Menge von Fkt. höherer Ordnung, die bestimmte Rekursionsmuster implementieren.

⇒ einfach wiederverwendbar

⇒ bessere Lesbarkeit,

Schnellere Prog-Entwicklung

Bsp: map

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

```
suclist [7,3,5] = [8,4,6]
sqrtlist [25,9] = [5,3]
```

Erkenntnis:

suclist und sqrtlist arbeiten nach dem gleichen Prinzip: Durchlaufe eine Liste und wende eine Fkt. auf jedes Listenelement an.

⇒ Abstrahiere von den Unterschieden zwischen suclist und sqrtlist, um einen Algorithmus für dieses

generelle Rekursionsmuster zu finden.

1. Abstrahiere vom Typ der Listenelemente
   ⇒ benötigt Prog.-Sprache mit param. Polymorphie (d.h. mit Typvariablen)
2. Abstrahiere von der Funktion, die auf die Listenelemente angewendet wird
   ⇒ benötigt Prog-Sprache mit Fkt. höherer Ordnung

In der entstehenden Fkt. f muss g ein weiteres Eingabeargument sein. (g ↖ Typ: a→b)

⇒ f entspricht map g

Die Fkt. map ist in

Haskell vordefiniert. Sie implementiert das Rekursionsprinzip: Durchlaufe eine Liste und wende eine Funktion auf jedes Listenelement an.
Nun lassen sich sucList und sqrtList viel einfacher implementieren:

```
sucList :: [Int] -> [Int]
sucList xs = map suc xs

sqrtList :: [Float] -> [Float]
sqrtList xs = map sqrt xs
```

Funktionen wie map lassen sich auch für andere rekursive Datenstrukturen wie Bäume, Graphen, ... definieren.

Bsp: filter

dropEven löscht alle geraden Zahlen in einer Liste

dropEven [1,2,3,4] = [1,3]

dropUpper löscht alle Groß-buchstaben eines Strings

dropUpper "GmbH" = "mb"

Die vordef. Fkt isLower ist nicht in der Standard-Prog-Bibliothek (Prelude), sondern im Modul

Data.Char

⇒ import Data.Char
(importiert alle Funktionen des Moduls)

oder import Data.Char(isLower)

Abstrahiere von den Unterschieden zwischen dropEven und dropUpper.
⇒ Fkt f

Typa->Bool

Die Filter-Funktion g muss ein weiteres Eingabeargument sein.
⇒ "filter" ist in Haskell vordefiniert
⇒ Damit lassen sich dropEven und dropUpper sehr leicht implementieren.

2. Unendliche Datenobjekte

Aufgrund der Outermost Auswertungsstrategie kann man mit unendl.

Datenobjekten rechnen

Bsp:

from 5 = [5, 6, 7, ...]

terminiert nicht

take n [$x_1, x_2, \ldots, x_n, \ldots$]

liefert [$x_1, \ldots, x_n$].

Dies funktioniert auch bei unendlichen Listen.

⇒ Man kann in Haskell mit unendlichen Datenobjekten rechnen, falls zur Berechnung des Ergebnisses nur ein endlicher Teil des unendlichen Datenobjekts benötigt wird.

Bsp: Berechne die (un-endliche) Liste aller Primzahlen:

"Sieb des Eratosthenes"

Idee: Erzeuge unendl. Datenobjekt, das die gesuchte Lösung approximiert. Dann filtere die Lösung darin heraus.

[2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ...]

Eindruck v. Funktionaler Programmierung

mehr in
⇒ Vorlesung "Funktionale Programmierung"