III.6 Funktionale Programmiertechniken

Mittwoch, 14. Januar 2015 08:30

Funktion höherer Ordnung:

Funktion, deren Argument oder Resultat wieder eine Fkt ist.

square :: Int → Int
Fkt 1. Ordnung

plus :: Int → (Int → Int)
Fkt höherer Ordnung, denn Resultat von plus 2 ist Fkt.

comp: Funktionskomposition

Wenn $g :: a \to b$

$f :: b \to c$

dann $f \circ g :: a \to c$

Fkt, die erst g u. dann f ausführt

∘ hat also den Typ

$(b \to c) \to (a \to b) \to (a \to c)$

f g Resultat

Ist in Haskell unter dem Namen . vordefiniert.

(half . square) 4 =

half (square 4) = 8

Bsp curry:

Wenn plus :: (Int, Int) → Int,

dann

curry plus 2 3 = 5

Fkt. höherer Ordnung lassen sich zur übersichtlichen Strukturierung v. Programmen einsetzen. Programmiere häufig vorkommende Rekursionsmuster als Fkt. höherer Ordnung u. verwende diese dann immer wieder in konkreten Algorithmen. Manche dieser typischen Rekursionsmuster sind in Haskell schon vordefiniert (z.B. map, filter).

Hierbei: suc :: Int → Int
suc = plus 1

suclist $[x_1, \ldots, x_n]$ = $[suc\ x_1, \ldots, suc\ x_n]$

suclist [1,4,9] = [2,5,10]
sqrtlist [1,4,9] = [1,2,3]

Erkenntnis: Beide Funktionen verwenden das gleiche Rekursionsmuster (durchlaufe eine Liste u. wende eine Fkt auf jedes Element an).

Lösung: Abstrahiere v. d. Unterschieden zwischen suclist u. sqrtlist und definiere eine allgemeine Fkt für dieses Rekursionsmuster.

Geht nur in Progspr. mit
- polymorphen Typen (ersetze Int bzw. Float

durch Typvariable)

- Fkt höherer Ordnung
(ersetze suc bzw. sqrt durch Variable g, die für eine Funktion steht)
g hat Typ a -> b

Fkt f benötigt g als weiteres Eingabeargument.
=> Fkt map
(ist in Haskell vordefiniert)

Suclist u. sqrtlist lassen sich nun einfacher mit map programmieren:

```
suclist :: [Int] -> [Int]
suclist l = map suc l

sqrtlist :: [Float] -> [Float]
sqrtlist l = map sqrt l
```

Weiteres Bsp für Rekursionsmuster: filter

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

dropUpper "GmbH" = "mb"

Fkt isLower ist vordefiniert, steht im Modul Data.Char.
(ähnlich wie packages in Java)

Beide Fkt verwenden das selbe Rekursionsmuster: Durchlaufe Liste u. filtere dabei Elemente. Alle Elemente, die das Filterprädikat nicht erfüllen, werden gelöscht.
=> Abstrahiere v. d. Unterschie-

den u. formuliere allgem. Fkt für dieses Rekursionsmuster.

- Ersetze Int bzw Char durch Typvar. a.
- Ersetze odd bzw isLower durch Fkt-Var g

  ↗

  Typ a → Bool

Fkt filter ist in Haskell Vordef.

Einfachere Def v. dropEven u. dropUpper:

```
dropEven :: [Int] -> [Int]
dropEven l = filter odd l

dropUpper :: [Char] -> [Char]
dropUpper l = filter isLower l
```

Haskells outermost Auswertungsstrategie erlaubt Programme mit unendl. Datenobjekten, die trotzdem terminieren.

```
from x = [x, x+1, x+2, ...]
```

Kann als

```
[x..]
```

geschrieben werden.

⇒ `[2..]` =

```
[2, 3, 4, ...]
```

`take n l` liefert die ersten n Elemente der Liste l

`take 2 [1,2,3,4]` =

```
[1,2]
```

take 1 (from 5) = [5]

⇒ Man kann Prog. mit ∞ Datenobjekten schreiben. Dann sollte man darauf achten, dass diese immer nur endlich viele Schritte ausgewertet werden.

Bsp: Liste aller Primzahlen

Algorithmus: Sieb des Eratosthenes

Prinzip: Erstelle erst potentiell unendl. Liste von Approximationen an die Lösung und filtere anschließend daraus Schritt für Schritt die wirkliche Lösung heraus.

1. Erstelle Liste aller natürlichen Zahlen ab 2.
2. Markiere die erste unmarkierte Zahl in der Liste.
3. Streiche alle Vielfachen der gerade markierten Zahl.
4. Gehe zurück zu Schritt 2.

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

Liste aller Zahlen ab 2:

from 2 (oder [2..])

Streichen von Vielfachen:

drop_mult x xs

Int [Int]

löscht alle Vielfachen von x aus der Liste xs

Zahl y ist kein Vielfaches v. x

falls $y \bmod x \neq 0$
(dh: mod y x /= 0)

dropall :: [Int] -> [Int]
ruft drop_mult erst mit dem 1. Listenelement auf (u. streicht alle seine Vielfachen), dann mit dem 2. Listenelement etc.

---

primes selbst terminiert nicht

[2, 3, 5, 7, ...]

Aber z.B.
take 100 primes
terminiert u. berechnet die ersten 100 Primzahlen.