III.6 Funktionale Programmiertechniken

Montag, 23. Januar 2017 17:30

Viele Prog-Techniken kann man sowohl in Fkt. als auch in imperat. Sprachen benutzen.

Hier: 2 Prog-Techniken, die typisch für funktionale Sprachen sind

- Funktionen höherer Ordnung
- Unendliche Datenstrukturen

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

```
square :: Int -> Int
```
Fkt. erster Ordnung

```
plus :: Int -> (Int -> Int)
```
Fkt. höherer Ordnung

comp: Bsp für eine Fkt.

höherer Ordnung, da der sowohl Argument als auch Resultat eine Fkt ist.

Funktionskomposition $\circ$

$f \circ g$ ist die Funktion, die erst g anwendet und dann f

Typ $b \to c$ Typ $a \to b$

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

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

berechnet also

$$(\backslash x.\ \frac{x^2}{2})$$

comp ist vordefiniert in Haskell, heißt dort.

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

Bsp: curry (uncurry f) = f

uncurry (curry f) = f

Curry + uncurry haben Funktionen als Ein- und Ausgabe

Fkt. höherer Ordnung als Prog-Technik

- Formuliere ein häufig verwendetes Rekursionsmuster als Fkt. höherer Ordnung
- Verwende diese Fkt. höherer Ordnung dann immer wieder in konkreten Algorithmen
- Vorteil: Wiederverwendbarkeit + bessere Lesbarkeit

Bsp: `suc :: Int -> Int`

`suc = plus 1`

`suclist [1,4,9] = [2,5,10]`

`sqrtlist [1,4,9] = [1,2,3]`

Erkenntnis: suclist + sqrtlist verwenden dasselbe Rekursionsmuster: "Durchlaufe eine Liste u. wende eine Fkt. auf jedes Element d. Liste an."
Dieses Rekursionsmuster sollte man einmal als Fkt. höherer Ordnung implementieren und dann immer wiederverwenden.
Finde das Rekursionsmuster, indem man von den Unterschieden zw. suclist u. sqrtlist abstrahiert:

- Ersetze den Typ der Elemente (Int bzw. Float) durch Typvariable. (Geht nur in Sprachen mit param. Polymorph.)
- Ersetze die Fkt., die auf Elemente angewendet wird

(suc bzw. sqrt) durch Variable g. (Gelt nur in Sprachen mit Fkt. höherer Ord.)

Die entstehende Fkt f kann man so nicht in Haskell schreiben, denn g sollte ein weiterer Eingabeparameter von f sein.

Die Fkt map auf Listen ist in Haskell vordef.

Analoge map-Fkten lassen sich natürlich auch für eigene Datenstrukturen schreiben.

map ist Fkt. höherer Ordnung, die das obige Rekursionsmuster implementiert.

Konkrete Alg. wie suclist oder sqrtlist sollte man daher mit Hilfe von map implementieren:

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

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

Bsp: Weiteres Rekursionsmuster (filter)

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

(filtert alle ungeraden Zahlen heraus, d.h., löscht alle geraden Zahlen)

```
dropUpper "GmbH" = "mb"
```

(filtert alle Kleinbuchstaben heraus, d.h., löscht alle Großbuchstaben)

Haskell hat ähnliches Bibl.-System wie Java

(Pakete ≙ Module)
Java Haskell

Fkt. isLower steht im

Modul Data.Char.
⇒ am Anfang des Files
import Data.Char oder
import Data.Char (isLower)

Implementiere das verwendete Rekursionsmuster als Fkt höherer Ordng: abstrahiere v.d. Unterschieden zw. dropEven und dropUpper. Mit der (vordef.) Fkt filter kann man nun konkrete Alg. wie dropEven u. dropUpper direkt implementieren.

2. Prog-Technik:

Programmieren mit unendlichen Datenobjekten

(ist in Haskell möglich
wegen der outermost
Auswertungsstrategie)

from 2 = [2, 3, 4, ...]

terminiert nicht
(ist unendliche Liste)

take n xs liefert die
Liste mit den ersten n
Elementen der Liste xs
(in Haskell vordef.)

Nutze die Auswertungsstrategie
und unendl. Datenobjekte
als Prog-Technik:

- Berechne zunächst unendl.
Approximation an gewünschte
Lösung.
- Filtere daraus schrittweise

die gewünschte Lösung heraus.

Bsp: Sieb der Eratosthenes

Ziel: Berechne Liste aller Primzahlen

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

} drop_mult

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

drop_mult x xs löscht alle Vielfachen der Zahl x aus d. Liste xs.

Idee: filtere die Zahlen y aus der Liste xs heraus, bei denen

$y \bmod x \neq 0$.

Damit drop_mult immer wieder ausgeführt wird: dropall

Eindruck von fkt. Prog

- war nur erster Eindruck
- es ex. zahlreiche weitere Konzepte + Prog-Techniken in fkt. Sprachen

(Vorlesung "Funktionale Prog", "Termersetzung")