Abschnitt 3.2

Polymorphie und Funktionen höherer Ordnung

Abstraktion mittels Polymorphie und Funktionen höherer Ordnung

Überblick:

- Grundbegriffe der Typisierung
- Polymorphie als Abstraktionsmittel
- Typsysteme und Typinferenz
- Einführung in Funktionen höherer Ordnung
- Wichtige Funktionen höherer Ordnung

Unterabschnitt 3.2.1

Typisierung

Typisierung

Inhalte:

- Was ist ein Typ?
- Ziele der Typisierung
- Polymorphie und parametrische Typen
- Typsystem von Haskell und Typinferenz

Fast alle modernen Spezifikations- und Programmiersprachen besitzen ein Typsystem.

Was ist ein Typ?

Ein Typ beschreibt Eigenschaften von Elementen der Modellierung oder Programmierung:

- Elementare Typen stehen häufig für die Eigenschaft, zu einer bestimmten Wertemenge zu gehören (`Bool`, `Int`, `Char`, `String`, ...).
- Zusammengesetzte Typen beschreiben die genaue Struktur ihrer Elemente (Tupel-, Listen- und Funktionstypen).

Was ist ein Typ? (2)

- Parametrische Typen beschreiben bestimmte Eigenschaften und lassen andere offen; z.B.:
  - Elemente vom Typ `[a]` sind homogene Listen; man kann also `null`, `head`, `tail` anwenden. Offen bleibt z.B. der Ergebnistyp von `head`.
  - Elemente vom Typ `(a,[a])` sind Paare, so dass die Funktionen für Paare angewendet werden können. Außerdem besitzen sie die Eigenschaft, dass die zweite Komponente immer eine Liste ist, deren Elemente vom selben Typ sind wie die erste Komponente:

```
somefun :: ( a, [a] ) -> [a]

somefun p = let (fstc,sndc) = p
            in  fstc : sndc
```

Ziele der Typisierung

Die Typisierung verfolgt drei Ziele:

- Automatische Erkennung von Programmierfehlern (durch Übersetzer, Interpreter);
- Verbessern der Lesbarkeit von Programmen;
- Ermöglichen effizienterer Implementierungen.

Wir konzentrieren uns hier auf das erste Ziel.

Zentrale Idee:

- Für jeden Ausdruck und jede Funktion wird ein Typ festgelegt.
- Prüfe, ob die Typen der aktuellen Parameterausdrücke mit der Signatur der angewendeten Funktion übereinstimmen.

Beispiele: (Typprüfung von Ausdrücken)

```
f :: Int -> Int
```

dann sind:

`f 7, f (head [1,2,3]), f (f 78)+ 9` typkorrekt;

`f True, [f,5.6], f head` nicht typkorrekt.

Bemerkung:

Typisierung war lange Zeit nicht unumstritten.
Hauptgegenargumente sind:

- zusätzlicher Schreib- und Entwurfsaufwand
- Einschränkung der Freiheit:
  - inhomogene Listen
  - Nutzen der Repräsentation von Daten im Rechner

Beispiel:

Es gibt viele Programme, die nicht typkorrekt sind, sich aber trotzdem zur Laufzeit gutartig verhalten; z.B:

Aufgabe 1:
Schreibe eine Funktion frp:

- Eingabe: Liste von Paaren entweder vom Typ `(Bool,Int)` oder `(Bool,Float)`
- Zulässige Listen: Wenn 1. Komponente `True`, dann 2. Komponente vom Typ `Int`, sonst vom Typ `Float`
- Summiere die Listenelemente und liefere ein Paar mit beschriebener Eigenschaft.

Beispiel: (2)

Realisierung in Haskell-Notation (kein Haskell-Programm, da nicht typkorrekt!):

```
frp :: ( Bool, ? ) -> ?

frp [ ]              =  (True, 0)
frp ((True,n):xs)    =
       case  frp xs  of
         (True,k)   ->  (True, k+n )
         (False,q)  ->  (False, q+(fromInteger n))
frp ((False,r):xs)   =
       case  frp xs  of
         (True,k)   ->  (False, (fromInteger k)+r )
         (False,q)  ->  (False, q+r )
```

Beispiel: (3)

Aufgabe 2:
Schreibe eine Funktion,

- die ein n-Tupel (n≥2) nimmt und
- die erste Komponente des Tupels liefert.

Kann in Haskell nicht definiert werden:

```
arbitraryfst :: (a,b,...) -> a
arbitraryfst  (n,...) = n
```

Polymorphie und parametrische Typen

Programmierer möchten vom Typsystem nicht weiter eingeengt werden als nötig. Ziel:

- mächtige und flexible Typsysteme
- insbesondere Polymorphie und Parametrisierung

Im Allg. bedeutet Polymorphie Vielgestaltigkeit.

In der Programmierung bezieht sich Polymorphie auf die Typisierung bzw. das Typsystem.

Begriffsklärung: (polymorphes Typsystem)

Das ***Typsystem*** einer Sprache *S* beschreibt,

- welche Typen es in *S* gibt bzw. wie neue Typen deklariert werden;
- wie den Ausdrücken von *S* ein Typ zugeordnet wird;
- welche Regeln typisierte Ausdrücke erfüllen müssen.

Ein Typsystem heißt ***polymorph***, wenn Ausdrücke zu Werten bzw. Objekten unterschiedlichen Typs ausgewertet werden können.

Bemerkung:

- Man unterscheidet:
  - ***Parametrische Polymorphie***
  - ***Subtyp-Polymorphie*** (vgl. Typisierung in Java)

- Oft spricht man im Zusammenhang mit der Überladung von Funktions- oder Operatorsymbolen von Ad-hoc-Polymorphie.

  Beispiel:
  Dem +-Operator könnte man in Haskell den Typ
  `Int ->Int ->Int` oder `Float ->Float ->Float` geben.

Bemerkung: (2)

- In polymorphen Typsystemen gibt es meist eine Relation „ist_spezieller_als“ zwischen Typen T1, T2:

  T1 heißt spezieller als T2, wenn die Eigenschaften, die T1 garantiert, die Eigenschaften von T2 implizieren (umgekehrt sagt man: T2 ist allgemeiner als T1).

  Beispiel:
  Der Typ `[Int]` ist spezieller als der parametrische Typ `[a]` .
  Insbesondere gilt:

  Jeder Wert vom Typ `[Int]` kann überall dort benutzt werden, wo ein Wert vom Typ `[a]` erwartet wird.

Typsystem von Haskell und Typinferenz

Typen werden in Haskell durch Typausdrücke beschrieben:

- Typkonstanten sind die Basisdatentypen: `Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`
- Typvariablen: `a`, `meineTypvar`, `gTyp`
- Ausdrücke gebildet mit Typkonstruktoren:

  Seien TA, TA1, TA2, TA3, ... Typausdrücke, dann sind die folgenden Ausdrücke auch Typausdrücke:

`( TA1, TA2 )`	Typ der Paare
`( TA1, TA2, TA3 )`	Typ der Triple
...	
`[ TA ]`	Listentyp
`TA1 -> TA2`	Funktionstyp

Einem Typausdruck TA kann ein Typconstraint für Typvariablen `a` vorangestellt werden: `(Eq a) => TA`

Beispiele: (Typausdrücke)

```
Int,  Bool

a,  b

(Int,Bool) , (Int,a) ,  (Int,a,b,a)

[Integer] , [a] , [(Float, b, a)]

Int -> Int , [a] -> [(Float, b, a)]

(Char -> Char) -> (Int -> Int)

(Eq a) => a -> [a] -> Bool
```

Hinweis:
-> ist rechtsassoziativ.

Begriffserklärung: (Vergleich von Typen)

Seien *TA* und *TB* Typausdrücke und *var*(*TA*) die Menge der Typvariablen, die in *TA* vorkommen.

Eine *Variablensubstitution* ist eine Abbildung β von *var*(*TA*) auf die Menge der Typausdrücke.

Bezeichne TA_β den Typausdruck, den man aus *TA* erhält, wenn man alle Variablen *v* in *TA* konsistent durch $\beta(v)$ ersetzt.

TB **ist spezieller als** *TA*, wenn es eine *Variablensubstitution* gibt, so dass $TA_\beta = TB$.

TA und *TB* bezeichnen den gleichen Typ, wenn *TA* spezieller als *TB* ist und *TB* spezieller als *TA*.

Beispiele:

`[Int]` ist spezieller als `[a]`

`[a]` ist spezieller als `[b]` und umgekehrt

`a -> a` ist spezieller als `a -> b`

`([Int],b)` und `([c],Bool)` sind nicht vergleichbar, d.h. der erste Ausdruck ist nicht spezieller als der zweite und der zweite nicht spezieller als der erste.

`(Eq a) => TA` ist spezieller als `TA`

Typen in Haskell

Bezeichne TE die Menge der Typausdrücke in Haskell.

- Die „ist_spezieller_als“ Relation auf $TE \times TE$ ist reflexiv und transitiv, aber nicht antisymmetrisch.
- Identifiziert man alle Typausdrücke, die den gleichen Typ bezeichnen, erhält man die Menge T der Typen. (T, ist_spezieller_als) ist eine partielle Ordnung.

Damit ist gesagt, was Typen in Haskell sind.

Bemerkung:
Das Typsystem von Haskell ist feiner als hier dargestellt (Typklassen und benutzerdefinierte Typconstraints).

Beispiel:

```
foldplus :: (Num a) => [a] -> a
-- Addiert alle Elemente einer Liste.
-- Listenelemente muessen alle vom gleichen Zahltyp sein

foldplus []     =  0
foldplus (x:xs) =  x + foldplus xs
```

Typregeln in Haskell:

Wir nennen eine Ausdruck *A typannotiert*, wenn *A* und allen Teilausdrücken von *A* ein Typ zugeordnet ist.

Beispiel:

Die Typannotationen von `abs (-2)` ist `((abs::Int->Int)(-2::Int))::Int`

Die Typregeln legen fest, wann ein typannotierter Ausdruck typkorrekt ist. In Hakell muss gelten:

- Die Typen der formalen Parameter müssen gleich den Typen der aktuellen Parameter sein.
- Bedingte Ausdrücke müssen korrekt typisiert sein.

Begriffserklärung: (Typinferenz)

Typinferenz bedeutet das Ableiten der Typannotation für Ausdrücke aus den gegebenen Deklarationsinformationen.

Bemerkung:
In Programmiersprachen mit parametrischem Typsystem kann Typinferenz sehr komplex sein.

Beispiele:

1. Leere Liste:

```
x = [ ]
```

   Inferierter Typ für `x :: [a]`

2. Enthalten sein in Liste:

```
enthalten p1 []       =  False
enthalten p1 (x:xs)
         | p1 == x    =  True
         | otherwise  =  enthalten p1 xs
```

   Inferierter Typ für `enthalten :: (Eq a) => a -> [a] -> Bool`

Beispiele: (2)

3. Einsortieren in geordnete Liste mit Vergleichsfunktion:

```
einsortieren vop p1 []     =  [p1]
einsortieren vop p1 (x:xs)
        | p1 `vop` x   =  p1:x:xs
        | otherwise    =  x : (einsortieren vop p1 xs)
```

Inferierter Typ für `einsortieren ::`

```
(t -> t -> Bool) -> t -> [t] -> [t]
```

Bemerkung:
Bei der Typinferenz versucht man immer den allgemeinsten Typ herauszufinden.

Parametrische markierte Binärbaumtypen:

1. Alle Markierungen sind vom gleichen Typ:

```
data  BBaum a =
        Blatt a
      | Zweig a (BBaum a) (BBaum a)
```

2. Blatt- und Zweigmarkierungen sind möglicherweise von unterschiedlichen Typen:

```
data  BBaum a b =
        Blatt a
      | Zweig b (BBaum a b) (BBaum a b)
```

Parametrische markierte Binärbaumtypen: (2)

3. Und was passiert hier?

```
data  BBaum a b =
        Blatt a
      | Zweig b (BBaum b a) (BBaum a b)
```

Unterabschnitt 3.2.2

Funktionen höherer Ordnung

Funktionen höherer Ordnung

Überblick:

- Einführung in Funktionen höherer Ordnung
- Wichtige Funktionen höherer Ordnung

Einführung

Funktionen höherer Ordnung sind Funktionen, die

- Funktionen als Argumente nehmen und/oder
- Funktionen als Ergebnis haben.

Selbstverständlich sind auch Listen oder Tupel von Funktionen als Argumente oder Ergebnisse möglich.

Eine Funktion F, die Funktionen als Argumente nimmt und als Ergebnis liefert, nennt man häufig auch ein **Funktional**.
Funktionale, die aus der Schule bekannt sind, sind

- Differenzial
- unbestimmtes Integral

Sprachliche Aspekte:

Alle wesentlichen Sprachmittel zum Arbeiten mit Funktionen höherer Ordnung sind bereits bekannt:

- Funktionsabstraktion
- Funktionsdeklaration
- Funktionsanwendung
- Funktionstypen

Konzeptionelle Aspekte:

Zwei konzeptionelle Aspekte liegen der Anwendung von Funktionen höherer Ordnung in der Software- Entwicklung zugrunde:

1. Abstraktion und Wiederverwendung
2. Metaprogrammierung, d.h. das Entwickeln von Programmen, die Programme als Argumente und Ergebnisse haben.

Wir betrachten im Folgenden den ersten Aspekt.

Beispiel: (Abstraktion, Wiederverwendung)

Aufgabe:
Sortiere eine Liste xl von Zahlen durch Einfügen (insertion sort)

Rekursionsidee:

- Sortiere zunächst den Rest der Liste.
- Das ergibt eine sortierte Liste xs.
- Sortiere das erste Element von xl in xs ein.

Beispiel: (Abstraktion, Wiederverwendung) (2)

```
einsortieren :: Int -> [Int] -> [Int]

einsortieren p1 []       =  [p1]
einsortieren p1 (x:xs)
        | p1 <= x    =  p1:x:xs
        | otherwise  =  x : (einsortieren p1 xs)

sort []     = []
sort (x:xr) = einsortieren x (sort xr)
```

Beispiel: (Abstraktion, Wiederverwendung) (3)

Frage:
Was ist zu tun, damit `sort` auch Werte der Typen `Char`, `String`, `Float`, etc. sortieren kann?

Antwort:
Abstraktion des Algorithmus: Führe die Vergleichsoperation als weiteren Parameter ein.

```
einsortieren :: (t -> t -> Bool) -> t -> [t] -> [t]

einsortieren vop p1 []     =  [p1]
einsortieren vop p1 (x:xs)
        | p1 `vop` x   =  p1:x:xs
        | otherwise    =  x : (einsortieren vop p1 xs)

sort vop []     = []
sort vop (x:xr) = einsortieren vop x (sort vop xr)
```

Anwendung:

```
l1 = sort (<=) [ 2,3, 968,-98,34,0 ]
l2 = sort (>=) [ 2,3, 968,-98,34,0 ]
l3 = sort ((>=)::Float -> Float -> Bool) [1.0,1e-4]
l4 = sort (>=) [1.0,1e-4]

strcmp :: String -> String -> Bool
strcmp = (<=)

l5 = sort strcmp ["Abbay","Abba","Ara","ab"]
```

Bemerkung:

Polymorphe Funktionen können häufig auch Funktionen als Parameter nehmen.

Beispiele:

1. Funktion cons auf Listen : `abs :fac :[]`
2. Identitätsfunktion: `id =(x->x)(x->x)`

Wichtige Funktionen höherer Ordnung

Dieser Abschnitt betrachtet einige Beispiele für Funktionen höherer Ordnung und diskutiert das Arbeiten mit solchen Funktionen.

Applikationsoperator:

```
($) :: (a -> b) -> a -> b
```

rechtsassoziativ; erlaubt andere Schreibweise/Klammersetzung:
`fac $ fac $ n+1` statt `fac (fac (n+1))`

Funktionskomposition:

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

Wichtige Funktionen höherer Ordnung (2)

Map:
Anwendung einer Funktion auf die Elemente einer Liste:
map f [x1,x2,x3] == [f x1, f x2, f x3]

```
map :: (a -> b) -> [a] -> [b]

map f []     = []
map f (x:xs) = (f x): map f xs
```

Beispiele:

```
map length ["Schwerter","zu","Pflugscharen"]

double n = 2*n
map (map double)  [ [1,2], [34829] ]
```

Currying und Schönfinkeln:

Funktionen mit einem Argumenttupel kann man die Argumente auch sukzessive geben. Dabei entstehen Funktionen höherer Ordnung.

Beispiele:

```
times :: Integer -> Integer -> Integer
times m n =  m * n

double :: Integer -> Integer
double = times 2

double 5
```

Zwei Varianten von map im Vergleich:

1. Die beiden Argumente als Paar:

```
mapp :: (b -> a, [b]) -> [a]

mapp (f,[])    = []
mapp (f,x:xs) = (f x): mapp (f,xs)
```

2. Die Argumente nacheinander („gecurryt“):

```
map :: (a -> b) -> [a] -> [b]

map f []      = []
map f (x:xs) = (f x): map f xs
```

Bemerkung:

Die gecurryte Fassung ist flexibler: Sie kann nicht nur auf ein vollständiges Argumententupel angewendet werden, sondern auch zur Definition neuer Funktionen mittels partieller Anwendung benutzt werden.

Beispiele:

```
double :: Integer -> Integer
double =  times 2

intListSort :: [Int] -> [Int]
intListSort =  sort ((<=)::Int->Int->Bool)

doublelist :: [Int] -> [Int]
doublelist  =  map double
```

Curryen von Funktionen:

Die Funktion `curry` liefert zu einer Funktion auf Paaren die zugehörige gecurryte Funktion:

```
curry :: ((a, b) -> c) -> a -> b -> c

curry f x y = f (x,y)
```

Prüfen und Filtern von Listen:

```
any :: (a -> Bool) -> [a] -> Bool
--  Pruefe, ob Liste ein Element enthaelt, das
--  das gegebene Praedikat erfuellt

any pred [ ]     = False
any pred (x:xs) = pred x  ||  any pred xs

all :: (a -> Bool) -> [a] -> Bool
--  Pruefe, ob jedes Listenelement das gegebene
--  Praedikat erfuellt

all pred [ ]     = True
all pred (x:xs) = pred x  &&  all pred xs
```

Prüfen und Filtern von Listen: (2)

```
filter :: (a -> Bool) -> [a] -> [a]
--  Liste alle Elemente, die das gegebene Praedikat
--  erfuellen

filter pred [ ]    = []
filter pred (x:xs)
    | pred x       = x : (filter pred xs)
    | otherwise    = (filter pred xs)
```

Beispiele:

```
ismember x xs = any (\y-> x==y) xs

split p xs =
     (filter (\y-> y<p) xs, filter (\y-> p<=y) xs)
```

Punktweise Veränderung von Funktionen:

Die "Veränderung" einer Funktion an einem Punkt des Argumentbereichs:

```
update :: (Eq a) => (a -> b) -> a -> b -> a -> b

update f x v y
  |  x == y     =  v
  |  otherwise =  f y
```

Falten von Listen:

Eine häufig benötigte Funktion ist das Falten einer Liste mittels einer binären Funktion und einem neutralen Element bzw. Anfangselement:

foldr $\otimes$ n [e1, e2, . . . ,en] = e1 $\otimes$ (e2 $\otimes$ (...(en $\otimes$ n) . . .))

Deklaration von foldr:

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f n [ ]     = n
foldr f n (x:xs) = f x (foldr f n xs)
```

Falten von Listen: (2)

Üblicherweise steht auch eine Funktion für das Falten von links zur Verfügung:

foldl $\otimes$ n [e1, e2, . . . ,en] = (. . . ((n $\otimes$ e1) $\otimes$ e2) ... $\otimes$ en)

Deklaration von foldl:

```
foldl :: (b -> a -> b) -> b -> [a] -> b

foldl f n [ ]     = n
foldl f n (x:xs) = foldl f (f n x) xs
```

Falten von Listen: (3)

Mittels der Faltungsfunktionen `foldr` und `foldl` lassen sich viele Listenfunktionen direkt, d.h. ohne Rekursion definieren:

```
sum, product :: (Num a) => [a] -> a
sum     = foldr (+) 0
product = foldr (*) 1

(++) :: [a] -> [a] -> [a]
(++) l1 l2 = foldr (:) l2 l1
```

Falten von Listen: (4)

Bäume mit variabler Kinderzahl lassen sich allgemeiner und kompakter behandeln (vgl. Folien 337f):

```
data VBaum = Kn Int [VBaum]  deriving (Eq,Show)

zaehleKnVBaum :: VBaum -> Int
zaehleknVBaum (Kn _ xs) =
              foldr (+) 1 (map zaehleknVBaum xs)
```

Bemerkungen: (Funktionen höherer Ordnung)

Die Programmentwicklung mittels Funktionen höherer Ordnung (funktionale Programmierung) ist ein erstes Beispiel für:

- das Zusammensetzen komplexerer Bausteine,
- programmiertechnische Variationsmöglichkeiten,
- die Problematik der Wiederverwendung:
  - die Bausteine müssen bekannt sein,
  - die Bausteine müssen ausreichend generisch sein.

Bemerkungen: (zur Haskell-Einführung)

- Ziel des Kapitels war es nicht, eine umfassende Haskell-Einführung zu geben. Haskell dient hier vor allem als Hilfsmittel, wichtige Konzepte zu erläutern.
- Die meisten zentralen Konstrukte wurden behandelt.
- Es fehlt insbesondere:
  - Fehlerbehandlung
  - Typklassen
  - Aspekte imperativer und interaktiver Programmierung
- Viele Aspekte der Programmierumgebung wurden nicht näher erläutert.