Heutiges Thema

Mehr über Haskell, Teil 2

Von Basistypen (Int, Bool, Char,...) zu...

- *zusammengesetzten* Datentypen und Funktionen darauf
  $\leadsto$ *Tupel* und *Listen*

...und Weiterführendes/Ergänzendes über

- Listen, insbesondere
  - Listenkomprehension
  - (Polymorphe) Funktionen auf Listen
  - Vordefinierte Funktionen auf Listen
  - Polymorphie vs. Überladen
  - Listen und Muster

Tupel

Tupel ...fassen eine festgelegte Zahl von Werten möglicherweise verschiedener Typen zusammen.

$\leadsto$ Tupel sind *heterogen*!

Beispiele:

- ...Modellierung von Studentendaten

```
("MaxMuster","e0123456@student.tuwien.ac.at",534) ::
                    Num a => ([Char], [Char], a)
```

- ...Modellierung von Bibliotheksdaten

```
("PeytonJones","Implementing Funct. Lang.",1987,True) ::
                    Num a => ([Char], [Char], a, Bool)
```


Tupel...

- Allgemeines Muster

```
(v1,v2,...,vk) :: (T1,T2,...,Tk)
```

wobei

```
v1 :: T1, v2 :: T2,..., vk :: Tk
```

mit `v1,...,vk` Werten und `T1,...,Tk` Typen

- Standardkonstruktor

```
(. , . , ... , . )
```


Spezialfall: Paare ("Zweitupel")

- Beispiele

```
type Point = (Float, Float)

(0.0,0.0) :: Point
(3.14,17.4) :: Point
```

- Standardselektoren (für Paare)

```
fst (x,y) = x
snd (x,y) = y
```

- Anwendung der Standardselektoren

```
fst (0.0,0.0) = 0.0
snd (3.14,17.4) = 17.4
```

Hilfreich...

Typsynonyme

```
type Student = (String, String, Int)
type Buch = (String, String, Int, Bool)
```

...erhöhen die Transparenz in Programmen.

Beachte: Typsynonyme definieren *keine* neuen Typen, sondern einen Namen für einen schon existierenden Typ.

Tupel...

Selbstdefinierte Selektorfunktionen...

```
type Student = (String, String, Int)

name  :: Student -> String
email :: Student -> String
kennzahl :: Student -> Int

name (n,e,k)     = n
email (n,e,k)    = e
kennZahl (n,e,k) = k
```

...mittels Mustererkennung (engl. *pattern matching*)

Selbstdefinierte Selektorfunktionen...

Ein weiteres Beispiel...

```
autor :: Buch -> String
kurzTitel :: Buch -> String
erscheinungsJahr :: Buch -> Int
ausgeliehen :: Buch -> Bool

autor (a,t,j,b)  = a
kurzTitel (a,t,j,b)  = t
erscheinungsJahr (a,t,j,b)  = j
ausgeliehen (a,t,j,b)  = b

autErsch (a,t,j,b) = (autor (a,t,j,b), ausgeliehen (a,t,j,b))
```

...auch hier mittels Mustererkennung

Am Rande bemerkt...

Auch die Deklaration...

```
binom :: (Integer,Integer) -> Integer
binom (n,k)
  | k==0 || n==k   = 1
  | otherwise      = binom (n-1,k-1) + binom (n-1,k)
```

...ist Beispiel einer *musterbasierten* Funktionsdefinition.

Zum Vergleich...

mit Standardselektoren ohne Muster...

```
binom :: (Integer,Integer) -> Integer
binom p
  | snd(p)==0 || snd(p)==fst(p)  = 1
  | otherwise  = binom (fst(p)-1,snd(p)-1)
                        + binom (fst(p)-1,snd(p))
```

...offenbar weniger elegant und weniger gut lesbar.

Zurück zu Typsynonymen

Beachte...

- `type` ...führt neue Namen für bereits existierende Typen ein (Typsynonyme!), keine neuen Typen.

Daher...

Typsyonyme...

- tragen zur Dokumentation bei und
- erleichtern (i.a.) das Programmverständnis

aber...

- führen nicht zu (zusätzlicher) Typsicherheit!

Ein Beispiel

```
type Euro        = Float
type Yen         = Float
type Temperature = Float

myPi    :: Float
daumen  :: Float
maxTemp :: Temperature
myPi    = 3.14
daumen  = 5.55
maxTemp = 43.2

currencyConverter :: Euro -> Yen
currencyConverter x = x + myPi * daumen
```

Mit obigen Deklarationen...

```
currencyConverter maxTemp => 60.627
```

...werden 43.2 °C in 60.627 Yen umgerechnet. Typsicher?

Listen

Listen ...fassen eine beliebige/unbestimmte Zahl von Werten gleichen Typs zusammen.

$\rightsquigarrow$ Listen sind *homogen*!

Beispiele:

- ...Modellierung von Vorlesungshörern

```
[("MaxMuster","e0123456@student.tuwien.ac.at",534),
 ("MoritzMuster","e0987654@student.tuwien.ac.at",538),
 ("MimiMuster","e0987654@student.tuwien.ac.at",536)] :: [Student]
```

- ...Modellierung von Bibliotheken

```
[("PeytonJones","Implementing Funct. Lang.",1987,True),
 ("Pepper","Funkt. Prog.",2003,False)] :: [Buch]
```

Listen

Bereits vordefiniert...

```
Type String = [Char]
```

Somit...

```
['h','e','l','l','o'] == "hello"
"hello" ++ "world" -> "hello world"  (++: vordef. Konkatenationsop.)
```

Auch möglich...

- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [ [Int] ]`
- Listen von Paaren
  `[(3.14,42.0),56.1,51.3),(1.12,2.22)] :: [ Point ]`
- ...
- Listen von Funktionen
  `[fact, fib, fun91] :: [ Integer -> Integer ]`

Erinnerung aus der Mathematik...

Eine Funktion f ist ein Tripel (D,W,G) mit einer Definitionsmenge (-bereich) D, einer Wertemenge (-bereich) W und einer rechtseindeutigen Relation G mit $G \subseteq D \times W$, dem sog. Funktionsgraphen von f.

Mithin...

Funktionen sind spezielle Relationen; spezielle Teilmengen *eines kartesischen Produkts*

Damit intuitiv naheliegend...

Listen von Funktionen ... "Listen von Listen von Paaren"

Schlagwort...

Funktionen als *"first-class citizens"*

Weitere vordef. Funktionen auf Listen

length :: [a] −> Integer
length [] = 0
length (x:xs) = 1 + length xs

length [1, 2, 3] ⇒ 3 ([1,2,3] = 1:2:3:[])
length ['a','b','c'] ⇒ 3
length [[1],[2],[3]] ⇒ 3

head :: [a] −> a
head (x:xs) = x

tail :: [a] −> [a]
tail (x:xs) = xs

Auch möglich (und vorzuziehen):

head :: [a] −> a
head (x:_) = x

tail :: [a] −> [a]
tail (_:xs) = xs

Spezielles über Listen

Einige...

- Spezialfälle (i.w. für Listen über Zahlen und Zeichen)
  ...`[2 .. 6]` kurz für `[2,3,4,5,6]`
  ...`[11,9 .. 2]` kurz für `[11,9,7,5,3]`
  ...`['a','d' .. 'j']` kurz für `['a','d','g','j']`
  ...`[0.0,0.3 .. 1.0]` kurz für `[0.0,0.3,0.6,0.9]`

Listenkomprehension

...ein für funktionale Prog.sprachen charakteristisches Ausdrucksmittel.

- Listenkomprehension
  ...ein erstes Beispiel:
  `[3*n | n <- list]` kurz für `[3,6,9,12]`, falls `list` vom Wert `[1,2,3,4]` ist.

  $\leadsto$ Listenkomprehension ist ein sehr elegantes und ausdruckskräftiges Sprachkonstrukt!

Mehr zu Listenkomprehension (1)

Anwendungsbeispiele: ...wobei `lst = [1,2,4,7,8,11,12,42]`

a) `[ square n | n <- lst ]` $\Rightarrow^*$ `[1,4,16,49,64,121,144,1764]`

b) `[ n | n <- lst, isPowOfTwo n ]` $\Rightarrow^*$ `[1,2,4,8]`

c) `[ n | n <- lst, isPowOfTwo n, n>=5 ]` $\Rightarrow^*$ `[8]`

d) `[ isPrime n | n <- lst ]` $\Rightarrow^*$

`[False,True,False,True,False,True,False,False]`

Mehr zu Listenkomprehension (2)

Weitere Beispiele...

e)
```
addCoordinates :: [Point] -> [Float]
addCoordinates pLst = [ x+y | (x,y)<-pLst, (x>0||y>0) ]
```

`addCoordinates [(0.0,0.5),(3.14,17.4),(-1.5,-2.3)]` $\Rightarrow^*$ `[0.5,20.54]`

f)
```
allOdd :: [Integer] -> Bool
allOdd xs = ( [ x | x<-xs, isOdd x ] == xs )

allEven :: [Integer] -> Bool
allEven xs = ( [ x | x<-xs, isOdd x ] == [] )
```


Mehr zu Listenkomprehension (3)

g)
```
grabCapVowels :: String -> String
grabCapVowels s = [ c | c<-s, isCapVowel c ]

isCapVowel :: Char -> Bool
isCapVowel 'A' = True
isCapVowel 'E' = True
isCapVowel 'I' = True
isCapVowel 'O' = True
isCapVowel 'U' = True
isCapVowel  c  = False
```

Listenkomprehension “at work” (1)

...am Beispiel von “Quicksort”.

Aufgabe: Sortiere eine Liste L ganzer Zahlen aufsteigend.

Lösung mittels Quicksort:

- *Teile*: Wähle ein Element l aus L und partitioniere L in zwei (möglicherweise leere) Teillisten L_1 und L_2 so, dass alle Elemente von L_1 (L_2) kleiner oder gleich (größer) dem Element l sind.

- *Herrsche*: Sortiere L_1 und L_2 mit Hilfe rekursiver Aufrufe von Quicksort.

- *Zusammenführen der Teilergebnisse*: Trivial, die Gesamtliste entsteht durch Konkatenation der sortierten Teillisten.

Listenkomprehension “at work” (2)

Quicksort: Eine typische imperative Realisierung...

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition(L,low,high)
         quickSort(L,low,splitInd-1)
         quickSort(L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i=low+1 to high do
    if L[i] <= l then left = left+1
                      swap(L[i],L[left]) fi od
  swap(L[low],L[left])
  return left
```

...und dem initialen Aufruf `quickSort(L,1,length(L))`.

Listenkomprehension “at work” (3)

QuickSort: Eine typische Realisierung in Haskell...

```
quickSort :: [Int] -> [Int]

quickSort []     = []
quickSort (x:xs) =
    quickSort [ y | y<-xs, y<=x ] ++
                     [x] ++ quickSort [ y | y<-xs, y>x ]
```

Beachte: Funktionsanwendung bindet stärker als Listenkonstruktion. Deshalb Klammerung des Musters `x:xs` in `quickSort (x:xs) = ...`

Rückblick auf Teil 1 der Vorlesung...

Ein Vergleich - imperative vs. funktionale Programmierung

Gegeben ein Problem P.

Imperativ: Typischer Lösungsablauf besteht aus folgenden Schritten

1. Beschreibe eine(n) Lösung(sweg) L für P
2. Gieße L in die Form einer Menge von Anweisungen für den Rechner und organisiere dabei die Speicherverwaltung

Funktional:

- ...das “was” statt des “wie” in den Vordergrund stellen
- $\leadsto$ etwas von der Eleganz der Mathematik in die Programmierung bringen!

Quicksort: Ein eindrucksvolles Beispiel? Urteilen Sie selbst...

Generische Funktionen, Polymorphie, Typvariablen (1)

Wir haben Listen sehr unterschiedl. Typs kennengelernt, z.B.

- Listen von Basistypelementen
  `[2,4,23,2,53,4] :: [Int]`
- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [ [Int] ]`
- Listen von Paaren
  `[(3.14,42.0),(56.1,51.3),(1.12,2.22)] :: [ Point ]`
- Listen von Funktionen
  `[fact, fib, fun91] :: [ Integer -> Integer ]`
- ...

Ziel: ...Gemeinsamkeiten unterstreichen und ausnutzen!

Generische Funktionen, Polymorphie, Typvariablen (2)

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Sprechweisen:

- `a` in `length :: [a] -> Int` heißt *Typvariable*. Typvariablen werden gewöhnlich mit Kleinbuchstaben vom Anfang des Alphabets bezeichnet: `a`, `b`, `c`,...
- Typen wie...

  ```
  length :: [Bool] -> Int
  length :: [ [Int] ] -> Int
  length :: [ Integer -> Integer ] -> Int
  ...
  ```

  heißen *Instanzen* des Typs `[a] -> Int`. Letzterer heißt *allgemeinster Typ* der Funktion `length`.

Bem.: Das Hugs-Kommando `:t expr` liefert den allgemeinsten Typ eines Haskell-Ausdrucks `expr`.

Weitere Beispiele polymorpher Funktionen

```
id :: a -> a
id x = x

ignoreSecond :: Num a => (Bool,b) -> a
ignoreSecond (x,y) = if x then 1 else -1

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _           = []

unzip :: [(a,b)] -> ([a],[b])
unzip []  = ([],[])
unzip ((x,y):ps) = (x:xs,y:ys)
                   where
                   (xs,ys) = unzip ps
```


In Haskel vordefinierte (polymorphe) Funktionen auf Listen...

`:`	`a->[a]->[a]`	Listenkonstruktor (rechtsassoziativ)
`!!`	`[a]->Int->a`	Proj. auf i-te Komp., Postfixop.
`length`	`[a]->Int`	Länge der Liste
`++`	`[a]->[a]->[a]`	Konkatenation zweier Listen
`concat`	`[[a]]->[a]`	Konkatenation mehrerer Listen
`head`	`[a]->a`	Listenkopf
`last`	`[a]->a`	Listen“endelement”
`tail`	`[a]->[a]`	Liste ohne Listenkopf
`init`	`[a]->[a]`	Liste ohne Listenendelement
`splitAt`	`Int->[a]->[[a],[a]]`	Aufspalten einer Liste an Stelle i
`reverse`	`[a]->[a]`	Umdrehen einer Liste
...		

Randbemerkungen... (1)

Listenkonstruktor vs. *Listenoperator*

: ist Listenkonstruktor, ++ ist Listenoperator...

Abgrenzung: Konstruktoren führen zu eindeutigen Darstellungen, Nichtkonstruktoren i.a. nicht.

Veranschaulichung:

`42:17:4:[] == (42:(17:(4:[])))`, aber...

`[42,17] ++ [] ++ [4] == [42,17,4] == [42] ++ [17,4] ++ []`

Bemerkung: `(42:(17:(4:[])))` deutet an, dass eine Liste *ein* Objekt ist, erzwungen durch die Typstruktur. Anders in imperativen/objektorientierten Sprachen: Listen nur indirekt existent, nämlich bei "geeigneter" Verbindung von Elementen durch Zeiger.

Randbemerkungen... (2)

"Verstecken" von in `Prelude.hs` vordefinierten Funktionen...

Ergänze...

```
import Prelude hiding (reverse,tail,zip)
```

...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden), um `reverse`, `tail` und `zip` zu verbergen.

Polymorphie vs. Überladen (1)

- Polymorphie ("Vielgestaltigkeit")
- Überladen (engl. Overloading) oder ad-hoc Polymorphie

Vordergründig...

...ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.

Bsp.:

```
Polymorphie                                Ueberladen
-----------                                -----------
length :: [a] -> Int                       gleich :: a -> a -> Bool
length []     = 0                 "naiv":  gleich x y = x == y
length (_:xs) = 1 + length xs
```

Beachte: Obige Implementierung von `gleich` ist naiv und in dieser Generalität nicht möglich!

Polymorphie vs. Überladen (2)

Informell...

- *Polymorphie* ("Vielgestaltigkeit")...
  $\leadsto$ gleicher Code trotz unterschiedlicher Typen
- *Überladen* (engl. Overloading) / *ad-hoc Polymorphie*...
  $\leadsto$ unterschiedlicher Code trotz gleichen Namens (und sinnvollerweise ähnlicher Funktionalität)

Beachte:

- Anders als für `length` ist für `gleich` keine typunabhängige einheitliche Implementierung möglich (Hinweis: *denke an Gleichheit auf Basistypen, Paaren, Listen, Bäumen, etc.*)
- Tatsächlich ist `gleich` nicht einmal typabhängig stets implementierbar. Warum? (Hinweis: *denke an Gleichheit auf Funktionen!*)

Zurück zu Listen, Mustern und Funktionen darauf... (1)

Weitere Beispiele...

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs

head :: [a] -> a
head (x:_)  =  x

tail :: [a] -> [a]
tail (_:xs) = xs

null :: [a] -> Bool
null []    = True
null (_:_) = False
```


Zurück zu Listen, Mustern und Funktionen darauf... (2)

Muster, Wild Cards und Typvariablen...

```
sign               :: Integer -> Integer
sign x | x > 0     = 1
       | x == 0    = 0
       | x < 0     = -1


takeFirst          :: Integer -> [a] -> [a]
takeFirst m ys = case (m,ys) of
                   (0,_)    -> []
                   (_,[])   -> []
                   (n,x:xs) -> x : takeFirst (n - 1) xs

ifThenElse :: Bool -> a -> a -> a
ifThenElse c t e = case c of True  -> t
                             False -> e
```

Somit als Fortschreibung...

Muster können sein...

Bislang:

- *Werte* (z.B. 0, 'c', True)
  ...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.
- *Variablen* (z.B. n)
  ...jedes Argument passt.
- *Wild card* "_"
  ...jedes Argument passt.

Neu:

- *Konstruktormuster* (hier über Listen; z.B. [], (p:ps))
  - Eine Liste passt auf das Muster [], wenn sie leer ist.
  - Eine Liste passt auf (p:ps), wenn sie nicht leer ist und der Listenkopf auf p und der Listenrest auf ps passt.

Hinweis: Im Fall von (p:ps) reicht es, dass die Argumentliste nicht leer ist.

Oft nützlich...

...das sog. as-Muster:

Beispiel: ...mit @ gelesen als "as".

```
listTransform :: [a] -> [a]
listTransform l@(x:xs) = (x : l) ++ xs
```

Zum Vergleich ohne as-Muster...

```
listTransform :: [a] -> [a]
listTransform (x:xs) = (x : (x : xs)) ++ xs
```

...weniger elegant und weniger gut lesbar.

Fazit für heute...

Beachte...

Musterbasierte Funktionsdefinitionen sind (i.a.)...

- elegant und
- führen zu knappen, gut lesbaren Spezifikationen.

Aber: Musterbasierte Funktionsdefinitionen...

- können zu subtilen Fehlern führen und
- erschweren (oft) Programmänderungen/-weiterentwicklungen (bis hin zur Tortur (vgl. Pepper [4]): denke etwa an das Hinzukommen eines weiteren Parameters)