

# Funktionale Programmierung

## Grundlegende Datentypen

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik  
Hochschule München

Letzte Änderung: 22.10.2018 10:53

### Inhaltsverzeichnis

Typen . . . . .	1
Datentypen definieren . . . . .	2
Numerische Typen . . . . .	2
Typklassen . . . . .	3
Werte vergleichen . . . . .	3
<code>if-then-else</code> als Ausdruck . . . . .	3
Tupel . . . . .	3
Listen . . . . .	4
Listen-Konkatenation . . . . .	4
<b>Strings</b> sind Listen von <b>Char</b> . . . . .	5
Strenge Typisierung . . . . .	5
Mehr Listenfunktionen . . . . .	5
Listenkompensation . . . . .	6
<code>print</code> vs. <code>putStr</code> . . . . .	6
Und was ist mit I/O? . . . . .	6
Mehr to do . . . . .	7
So wenig wie möglich, so viel wie nötig . . . . .	7

### Typen

- in Haskell ist alles streng typisiert
- Haskell verfügt über einen Typinferenz-Mechanismus, der den Typ berechnet

- im GHCi können Sie sich den Typ für jeden beliebigen (korrekten) Ausdruck anzeigen lassen
- Beispiele:

```
Prelude> :t 'a'  
'a' :: Char  
Prelude> :t "Hallo"  
"Hallo" :: [Char]
```

## Datentypen definieren

- neue Datentypen werden in Haskell mit dem Schlüsselwort **data** definiert
- beispielsweise wird in der Standardbibliothek definiert:

```
data Bool = False | True
```

- Bool ist der **Typkonstruktor** der z.B. in Typsignaturen auftaucht
- False und True sind **Datenkonstruktoren**
- das Pipe-Symbol | zeigt an, dass es sich bei Bool um einen sog. **Summentypen** (*sum type*) handelt
  - ein Wert vom Typ Bool ist entweder False oder True

## Numerische Typen

- Typklasse Integral
  - Int — ganze Zahlen, feste Länge
  - Integer — ganze Zahlen, beliebig groß/klein
- Typklasse Fractional
  - Float, Double — as usual
  - Ratio – Brüche (in Data.Ratio)
  - Scientific — speichereffiziente, nahezu beliebig genaue Zahlen (im Package scientific)
- alle sind gemeinsam in der Typklasse Num

## Typklassen

- Typklassen sind dazu da um Funktionen zu überladen
- in den Typsignaturen können sog. **Typconstraints** angegeben werden, z.B.

```
(/) :: Fractional a => a -> a -> a
```

- das bedeutet die Funktion (/) kann auf Werte aller Typen a angewendet werden mit der Einschränkung, dass a in der Typklasse Fractional ist
- d.h. insbesondere (/) kann nicht auf Int und Integer angewendet werden, dazu gibt es in Haskell die Funktion

```
div :: Integral a => a -> a -> a
```

- heisst **parametrischer Polymorphismus**

## Werte vergleichen

- um Werte auf Gleichheit zu vergleichen, gibt es in Haskell die Funktionen

```
(==) :: Eq a => a -> a -> Bool
```

```
(/=) :: Eq a => a -> a -> Bool
```

- die Relationen <, <=, ... sind in der Typklasse Ord, z.B.

```
(>=) :: Ord a => a -> a -> Bool
```

## if-then-else als Ausdruck

- Haskell hat keine Kontrollstrukturen
- if-then-else ist in Haskell als Ausdruck (*expression*) verfügbar, z.B.

```
if x == 3 then "three" else "not three"
```

- analog zu ternärem Operator ?: in anderen Programmiersprachen

## Tupel

- Tupel kombinieren zwei oder mehr Werte zu einem Tupel-Wert
- Datendefinition für 2-Tupel:

```
data (,) a b = (,) a b
```

- der Typkonstruktor hat zwei Typen als Parameter, d.h. er konstruiert auf Typebene aus zwei Typen einen neuen Typ

– ein Tupel ist ein sog. **Produkttyp**, d.h. wir benötigen Werte von beiden Typen `a` und `b` um einen Wert vom Typ `(,)` `a b` zu erzeugen

- Anmerkung: Üblicherweise schreiben wir die Werte und auch die Typen innerhalb der Klammern:

```
(True, "Hallo") :: (Bool, String)
```

- für 2-Tupel gibt es die Funktionen

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

- auf die Komponenten von Tupeln greifen wir üblicherweise mit sog. **Pattern Matching** zu, z.B.

```
tupFun :: (Int, [a]) -> (Int, [a]) -> (Int, [a])
tupFun (a, b) (c, d) = ((a + c), (b ++ d))
```

- ginge auch so, ist aber eher unüblich

```
tupFun' :: (Int, [a]) -> (Int, [a]) -> (Int, [a])
tupFun' t s = ((fst t + fst s), (snd t ++ snd s))
```

## Listen

- Listen sind ein weiterer first-class Containertyp in Haskell
- Listen enthalten beliebig viele Werte des selben Typs
- Tupel enthalten eine feste Anzahl Werte (potentiell) verschiedener Typen
- selbstverständlich kann beides kombiniert werden, z.B.

```
(True
, [ (1, "Hallo")
, (2, " ")
, (3, "Welt")
, (4, "!")
]
) :: Num a => (Bool, [(a, [Char])])
```

## Listen-Konkatenation

- betrachten wir die Typen<sup>1</sup>

---

<sup>1</sup>Der Typ von `concat` ist sogar allgemeiner, aber dazu später mehr

```
(++) :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
```

- d.h. wir können die beiden Funktionen auf Listen beliebiger Typen anwenden

```
Prelude> [1,2,3] ++ [4,5]
[1,2,3,4,5]
Prelude> concat [[1, 2], [3, 4, 5], [6, 7]]
[1,2,3,4,5,6,7]
```

## Strings sind Listen von Char

- einzeln mit

```
(++) :: String -> String -> String
Prelude> "Hallo" ++ " " ++ "Welt" ++ "!"
"Hallo Welt!"
```

- oder eine Liste mit

```
concat :: [String] -> String
Prelude> concat ["Hallo", " ", "Welt", "!"]
"Hallo Welt!"
```

## Strenge Typisierung

- Listen können alles enthalten, aber nie gemischt:

```
Prelude> "hello" ++ [1, 2, 3]
<interactive>:14:13:
  No instance for (Num Char) arising
    from the literal '1'
  In the expression: 1
  In the second argument of '(++)',
    namely '[1, 2, 3]'
  In the expression: "hello" ++ [1, 2, 3]
```

## Mehr Listenfunktionen

- *cons*: `(:) :: a -> [a] -> [a]`
- *head*
- *tail*

- take, drop
- (!!)
- ...

## Listenkomprension

- Haskell bietet mit der Listenkomprension eine Notation für Listen, die an die Mengenkomprension der Mathematik angelehnt ist
- Beispiel für Mengenkomprension:

$$\{(x, y) | \forall x \in \{1, \dots, 10\}, \forall y \in \{x, \dots, 10\}, 2 \cdot x \neq y\}$$

- in Haskell:

```
[ ( x, y) | x <- [1..10], y <- [x..10], 2 * x /= y ]
```

- obiges Beispiel zeigt außerdem, dass in Haskell Listen durch die Verwendung von .. kürzer geschrieben werden können

## print vs. putStr

- Werte aller Typen die in der **Typklasse Show** sind, können mit `print` ausgegeben werden, z.B.

```
Prelude> print 'a'
'a'
Prelude> print 2
2
Prelude> print "Hallo"
"Hallo"
```

- ausschließlich `Strings` können mit `putStr` ausgegeben werden

```
Prelude> putStrLn "Hallo"
Hallo
```

- Beachten Sie den Unterschied in der Ausgabe von `Hallo`

## Und was ist mit I/O?

- Haskell ist rein funktional
  - keine Seiteneffekte

- also keine I/O?
- IO ist für den Haskell Compiler ein Container, wie z.B. Listen
  - über IO weiß der Compiler aber nicht mehr als das
- nachdem die sequentielle Ausführung von I/O ein wesentliches Merkmal ist, gibt es in Haskell die sog. `do`-Notation

```
do
  putStrLn "Hello"
  putStrLn "World"
```

## Mehr to do

- innerhalb eines `do`-Blockes gibt es einige Besonderheiten zu beachten
- jede Zeile muss vom Typ `IO x` für ein beliebiges `x` sein
- der Typ der letzten Zeile ist der Ergebnistyp der Funktion
- soll ein Ausdruck einen Namen bekommen, muss ein `let` voran gestellt werden, z.B.

```
do
  ...
  let x = y * 13 + z
      z = 25 * y
```

- wird ein Wert durch innerhalb einer I/O-Action berechnet, kann er mit `<-` einen Namen bekommen, z.B.

```
do
  ...
  str <- getLine
```

- hat ein berechneter Wert nicht den Typ `IO x` für ein beliebiges `x`, muss er mit der Funktion

```
return :: x -> IO x
```

zu einem Wert vom Typ `IO x` gemacht werden

## So wenig wie möglich, so viel wie nötig

- halten Sie Ihren I/O-Anteil so klein wie möglich
- vermeiden Sie umfangreichere *reine* Berechnungen innerhalb eines `do`-Blockes
  - lagern Sie diese in eine eigene Funktion **ohne** I/O aus

- Funktionen ohne I/O können viel leichter getestet werden!
- mit Hilfe der `do`-Notation können Sie in Haskell Programme schreiben, die wie die in einer imperativen Sprache **aussehen**<sup>2</sup>
  - **widerstehen Sie dieser Versuchung**

---

<sup>2</sup>Die `do`-Notation ist nur *syntactic sugar*, imperativ ist da gar nichts (siehe später bei *Monaden*).