

# Tutorium 03: Typen und Typklassen

---

David Kaufmann

16. November 2022

Tutorium Programmierparadigmen am KIT

## Letztes Mal...

`foldl f z [] = z`

`foldl f z (x:xs) = f (foldl f z xs) x`

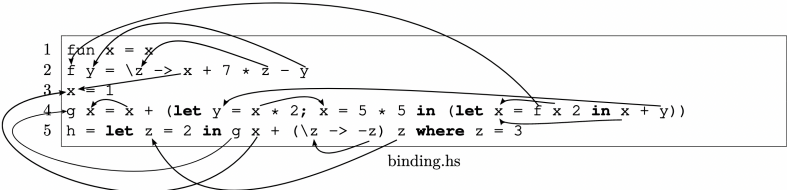
`foldl f z (x:xs) = foldl f (op x z) xs`

`foldr f z (x:xs) = f x (foldr f z xs)`

# Übungsblatt 2

---

## 2.1 – Bindung und Gültigkeitsbereiche



```
1 fun x = x
2 f y = \z -> x + 7 * z - y
3 x = 1
4 g x = x + (let y = x * 2; x = 5 * 5 in (let x = f x 2 in x + y))
5 h = let z = 2 in g x + (\z -> -z) z where z = 3
```

The diagram illustrates variable binding and scope resolution in the provided Haskell code. Arrows indicate the following:

- Line 1: `fun x = x`. An arrow points from `x` to its definition.
- Line 2: `f y = \z -> x + 7 * z - y`. An arrow points from `x` to its definition in line 1. Another arrow points from `z` to its lambda binding.
- Line 3: `x = 1`. An arrow points from `x` to its definition.
- Line 4: `g x = x + (let y = x * 2; x = 5 * 5 in (let x = f x 2 in x + y))`. An arrow points from the first `x` to its definition in line 3. Another arrow points from the `x` inside the `let` block to its definition in line 3. A third arrow points from `f x 2` to its definition in line 2.
- Line 5: `h = let z = 2 in g x + (\z -> -z) z where z = 3`. An arrow points from the `z` in `let z = 2` to its definition. Another arrow points from the `z` in `where z = 3` to its definition. A third arrow points from the `z` in `(\z -> -z)` to its lambda binding.

binding.hs

- Größte Fehlerquelle: `x * 2` und `f x 2` in Zeile 4
- Beide zeigen auf Definition im selben `let`-Block
- $\leadsto$  Allgemein: Variablen zeigen möglicherweise auf eine Definition im selben `let`-Block, selbst wenn es ihre eigene ist.

## 2.2.{1,2,3} – Polynome

```
module Polynom where

type Polynom = [Double]

cmult polynom c = map (* c) polynom

eval polynom x = foldr go 0 polynom
  where go a_n acc = acc * x + a_n

deriv [] = []
deriv polynom = zipWith (*) [1..] $ tail polynom
```

`foldr op i [] = i`

`foldr op i (x:xs) = op x (foldr op i xs)`

Beispiel Polynom [1,2,3,4]

`1 + x(2+x(3+x(4+x*0)))`

# **Wiederholung:**

## **Typen und Typklassen**

---

# Cheatsheet: Typen

- Char, Int, Integer, ...
- String
- *Typvariablen/Polymorphe Typen:*
  - (a, b): Tupel
  - [a]: Listen
  - a -> b: Funktionen
  - Vgl. Java: List<A>, Function<A, B>
- *Typsynonyme:* type String = [Char]



# Cheatsheet: Algebraische Datentypen in Haskell

- *data-Definitionen, Datenkonstrukturen*
- Algebraische Datentypen: *Produkttypen* und *Summentypen*
  - Produkttypen  $\approx$  structs in C
  - Summentypen  $\approx$  enums
- *Typkonstrukturen*, bspw. `[] :: * -> *`
- *Polymorphe* Datentypen, bspw. `[a]`, `Maybe a`
- Beispiel:

```
module Shape where

data Shape
  = Circle Double -- radius
  | Rectangle Double Double -- sides
  | Point -- technically equivalent to Circle 0
```

# Cheatsheet: Typklassen 1

- *Klasse, Operationen/Methoden, Instanzen*
- Beispiele:
  - `Eq t, {(==), (/=)}, {Eq Bool, Eq Int, Eq Char, ...}`
  - `Show t, {show}, {Show Bool, Show Int, Show Char, ...}`
- Weitere Typklassen: `Ord`, `Num`, `Enum`
- Deklaration/Implementierung:

```
module Truthy where

class Truthy t where
  toBool :: t -> Bool

instance Truthy Int where
  toBool x = x /= 0
```

## Cheatsheet: Typklassen 2

- *Vererbung*: Typklassen mit Voraussetzungen

```
module Truthy2 where

class Truthy t where
  toBool :: t -> Bool

instance Truthy Int where
  toBool x = x /= 0

instance Truthy t => Truthy (Maybe t) where
  toBool Nothing  = False
  toBool (Just x) = toBool x
```

## type: Namen für Typen

```
type String    = [Char]
type Rational  = Ratio Integer
type FilePath  = String
type IOError   = IOException
```

- `type N = T` definiert einen neuen Typpnamen `N` für den Typen `T`
- `N` kann nun überall verwendet werden wo auch `T` es kann
- $\leadsto$  Bessere Lesbarkeit  
(bspw. `readFile :: FilePath -> IO String`)

## data: Neue Typen

data definiert einen neuen Typen  $t$  durch die Aufzählung aller seiner „Konstruktoren“  $c_i$ :

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

```
data t     = c1
          | c2
          | ...
          | cn
```

Jeder Konstruktor  $c_i$  hat einen Namen und ggf. Parameter.

## data: Beispiel komplexe Zahlen

```
module Complex where

data Complex = Algebraic Double Double
              | Polar Double Double

real (Algebraic a b) = a
real (Polar r phi)   = r * cos phi
```

- Zwei Darstellungen:  $z = a + bi = r * (\cos \phi + i \sin \phi)$
- Beide Darstellungen bestehen aus zwei reellen Zahlen
- $\leadsto$  Durch unterschiedliche Konstruktornamen unterscheiden

## data: **Beispiel Bruchzahlen**

- Bruch ist ein Tupel von Integern
- Wie würde die Multiplikation aussehen

## data: Beispiel Bruchzahlen

- Bruch ist ein Tupel von Integern
- Wie würde die Multiplikation aussehen

```
module Fraction

data Fraction = Fraction Integer Integer

mul (Fraction a b) (Fraction a' b') =
    Fraction (a * a') (b * b')
```

- Definition von Fraction gibt uns Typsicherheit:  
Ein Bruch bleibt ein Bruch
- Konvention: Hat ein Typ nur einen Konstruktor, benennen wir diesen nach dem Typen.



# Übung

---

- Liste von Klassen, Name, Geburtsdatum (als Tupel)
- es gibt Klassen A, B, BE, C, D
- Klasse B kann Zusatzziffer B96 haben (über boolean angegeben).
- Beispiel:  
`DriversLicense [A, B True] "Arthur" (1, 1, 1970)`

```
module DriversLicense where

data DriversLicense = DriversLicense
  [VehicleClass]
  String
  (Int, Int, Int)

data VehicleClass = A | B Bool | BE | C | D
```

- Typ Playingcard hat Suit und Rank
- Suit kann Hearts, Diamonds, Clubs, Spades sein
- Rank kann Rank7, Rank8, Rank9, Rank10, Jack, Queen, King, Ace sein

```
module PlayingCard where

data PlayingCard = PlayingCard Suit Rank

data Suit = Hearts | Diamonds | Clubs | Spades
data Rank
  = Rank7 | Rank8 | Rank9 | Rank10
  | Jack | Queen | King | Ace
```

Warteschlangen lassen sich in Haskell, nicht effizient als Liste implementieren, weil die enqueue Operation immer die gesamte Liste durchlaufen muss. Folgende Definition schafft abhilfe

```
data Queue a = Q [a] [a]
```

`Q front back` stellt die Queue dar, die durch Konkatenation von `front` und der Umkehrung von `back` entsteht

Implementieren sie `fromList :: [a] -> Queue a` und `toList :: Queue a -> [a]`

Implementieren sie folgende Funktionen:

- `enqueue :: a -> Queue a` mit konstanter Anzahl Speicherzugriffe
- `dequeue :: Queue a -> Maybe (a, Queue a)`

Instanzieren sie die Typklasse Eq für den Datentyp Queue

**Hinweis:** sie müssen nur die Funktion (==) implementieren



Implementieren sie eine Funktion `bfs :: Tree a -> a`, die einen Binärbaum in dem bekannten Datentyp aus der VL entgegennimmt. Dabei gibt `bfs` die Knotenlabels des übergebenen Baums in Breitenordnung zurück

```
data Tree t = Leaf | Node (Tree t) t (Tree t)
```

Beispiel: `Node Leaf 1 (Node (Node Leaf 2 Leaf) 3 Leaf)`

```
>>> [1,3,2]
```

**Hinweis:** Implementieren sie eine Funktion `go :: Queue (Tree t) -> [a]`, die die Queue von Knoten abarbeitet