

# Tutorium 03: Typen und Typklassen

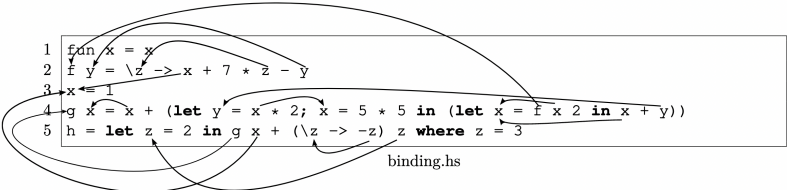
---

Paul Brinkmeier

15. November 2022

Tutorium Programmierparadigmen am KIT

## 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`. Arrows point from `x` to line 1 and from `z` to the lambda parameter.
- 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))`. Arrows show `x` binding to the parameter `x`, `y` binding to `x * 2`, the inner `x` binding to `f x 2`, and the outer `x` binding to line 3.
- Line 5: `h = let z = 2 in g x + (\z -> -z) z where z = 3`. Arrows show `z` binding to 2, `z` binding to the lambda parameter, and the `where z = 3` clause binding to `z`.

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.

# **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 Typnamen `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.

```
module Fraction

data Fraction = Fraction Integer Integer

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

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

## 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

# Typen selbst definieren

---

# Typen selbst definieren

Modelliert mit data:

- Brüche
- Führerschein
- Spielkarten
- Monopolykarten
- Boolesche Ausdrücke
- MiMa-Instruktionen
- (Typen in Haskell)

Vorlagen: [pad.pbrinkmeier.de/pp-tut](http://pad.pbrinkmeier.de/pp-tut)

```
module DriversLicense where

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

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

- Klasse B kann Zusatzziffer B96 haben.
- Für Daten gibt es natürlich auch eigene Typen.
- Beispiel:

```
DriversLicense [A, B True] "Arthur" (1, 1, 1970)
```

```
module PlayingCard where

data PlayingCard = PlayingCard Suit Rank

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

# Monopolykarten

```
module Monopoly where

data MonopolyCard
  = Street String Rent Int Color
  | Station String
  | Utility String

data Rent = Rent Int Int Int Int Int Int

data Color
  = Brown | LightBlue | Pink | Orange
  | Red | Yellow | Green | Blue
```



# Boolesche Logik

```
module BoolExpr where

data BoolExpr
  = Const Bool
  | Var String
  | Neg BoolExpr
  | BinaryOp BoolExpr BinaryOp BoolExpr

data BinaryOp = AND | OR | XOR | NOR
```

Beispiele:

- $a \wedge b$  entspricht `BinaryOp (Var "a") AND (Var "b")`
- $a \vee (b \wedge 0)$  entspricht  
`BinaryOp (Var "a") AND (BinaryOp (Var "b") OR (Const False))`

```
module MiMa where
```

```
data MimaInst
```

```
  = LDC Int | LDV Int | STV Int  
  | ADD Int | AND Int | OR Int  
  | EQL Int | JMP Int | JMN Int  
  | HALT | NOT | RAR
```

```
-- Alternatively:
```

```
data MimaInst' =
```

```
  Nullary MimaNullary | Unary MimaUnary Int
```

```
data MimaUnary = LDC' | LDV' | STV' | ADD' -- ...
```

```
data MimaNullary = HALT' | NOT' | RAR'
```

# Typklassen implementieren

---

# Implementierung von Typklassen

Implementiert:

- Monopolykarten: `Eq MonopolyCard`, `Show MonopolyCard`
- Boolesche Ausdrücke: `Show BoolExpr`
- Typen in Haskell: `Show Type`
- MiMa-Instruktionen: `Show MiMaInst`
- Spielkarten: `Eq` und `Ord` für `Suit`, `Rank` und `PlayingCard`
  - `Ord PlayingCard`: Zuerst nach Farbe, dann nach Wert.

Aufgabe: Spielkarten + eine weitere

## Ord PlayingCard – Vorlage

```
module PlayingCard2 where
import PlayingCard
instance Eq Suit where
    Hearts    == Hearts    = True
    Diamonds == Diamonds = True
    Clubs     == Clubs     = True
    Spades    == Spades    = True
    _         == _         = False
instance Ord Suit where
    s1 <= s2 = toInt s1 <= toInt s2
    where toInt Hearts    = 0
          toInt Diamonds = 1
          toInt Clubs     = 2
          toInt Spades    = 3
```

```
module PlayingCard3 where

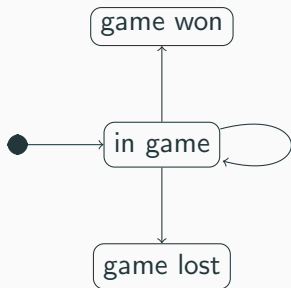
data PlayingCard = PlayingCard Suit Rank
  deriving (Eq, Ord, Show)

data Suit = Hearts | Diamonds | Clubs | Spades
  deriving (Eq, Ord, Show, Enum)
data Rank
  = Rank7 | Rank8 | Rank9 | Rank10
  | Jack | Queen | King | Ace
  deriving (Eq, Ord, Show, Enum)
```

Enum ermöglicht es, bspw. [Hearts .. Spades] zu schreiben.

# Hangman 2.0

```
data Hangman = Hangman
  String -- ^ Geheimes Wort.
  [Char] -- ^ Geratene Buchstaben.
```



- Summentypen eignen sich für Automaten
- `demos/Hangman.hs`
- Baut die Vorlage so um, dass der Typ `Hangman` einen Konstruktor pro Zustand hat
- $\leadsto$  Verlagerung von Fallunterscheidungen auf die Typebene (vgl. dynamische Bindung bei OOP)