Tutorium 03: Typen in Haskell

Paul Brinkmeier

05. November 2019

Tutorium Programmierparadigmen am KIT

Übungsblätter

- ÜB1 korrigiert
- ÜB2 korrigiert
- ÜB3: Bis Donnerstag Mittag

Heutiges Programm

Programm

- Übungsblatt 2
- Algebraische Datentypen
- Implementierung eines einfach verketteten Liste
- Typklassen
- Implementierung einer Queue

Übungsblatt 2

1 — Gültigkeitsbereiche

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

- Zeile 3: Welches `x` wird in `f x 2` verwendet?
- Zeile 3: Welches `x` wird in `y` verwendet?
- Zeile 4: Bindet `let` oder `where` stärker?

1 — Gültigkeitsbereiche, Z.3

```
module LetXInFx where


x = 42
f = (*2)


y = let x = f x in x
```

1 — Gültigkeitsbereiche, Z.3

```
module LetXInFx where


x = 42
f = (*2)


y = let x = f x in x
```

- `y` hängt in GHCi
- $\rightsquigarrow$ unendliche Schleife
- $\rightsquigarrow$ das im `let` definierte `x` wird verwendet

1 — Gültigkeitsbereiche, Z.3

```
module LetX where


x = 42
y = let z = 2 * x; x = 100 in z
```

1 — Gültigkeitsbereiche, Z.3

```
module LetX where


x = 42
y = let z = 2 * x; x = 100 in z
```

- `y = 200` in GHCi
- $\rightsquigarrow$ das im `let` definierte `x` wird verwendet

1 — Gültigkeitsbereiche, Z.4

```
module LetVsWhere where


stronger =
  let
    x = "let"
  in
  x
    where x = "where"
```

1 — Gültigkeitsbereiche, Z.4

```
module LetVsWhere where


stronger =
  let
    x = "let"
  in
  x
    where x = "where"
```

- `stronger = "let"` in GHCi
- $\rightsquigarrow$ `let` ist stärker
- `let ... in ...`: normaler Ausdruck
- `where`: nur in Definition

2.1 — Tiefgründige Typen

```
module DeepTypes where


vA  = (\x -> x)    :: a -> a
vB  = (\x -> x)    :: (a -> b) -> a -> b
vC  = (\x -> x)    :: (a -> b) -> (a -> b)
vD  = (\x _ -> x)  :: (a -> b -> a)
vE1 = (\x _ -> x)  :: (a -> a -> a)
vE2 = (\_ y -> y)  :: (a -> a -> a)
vF  = []           :: [a]
vG  = [1]          :: [Int]
vH  = []           :: [[a]]
vI  = [[]]         :: [[a]]
-- Alternativ:
-- vA x = x, vB g x = g x, etc.
```

2.2 — Tiefgründige Typen

Warum gibt keinen Wert, der `[a]` darstellt?

> `a` ist eine Typvariable, kein spezifischer Typ. Bspw. sind `[1]` und `["test"]` beide gültige Werte mit dem Typ `[a]`. Es gibt aber keine Wert, der für alle möglichen Typen ein `a` dartstellt (anders als bspw. `null` in Java).

2.3 — Tiefgründige Typen

```
module NestedList where

type Nested a = [[[[[[[[[[a]]]]]]]]]]

a1 = [[[[[[[[[[0]]]]]]]]]] -- :: Nested a
a2 = [[[[[[[[[[ ]]]]]]]]]]    :: Nested a
a3 =  [[[[[[[[[ ]]]]]]]]]     :: Nested a
a4 =     [[[[[[ ]]]]]]        :: Nested a
a5 =          [ ]             :: Nested a
a6 =          []              :: Nested a
```

- Eine Liste von Listen von Listen... ist immer noch eine Liste
- $\rightsquigarrow$ Kürzester Wert: leere Liste

3 — Listenkombinatoren

```
module Polynom where

type Polynom = [Double]

cmult :: Polynom -> Double -> Polynom
cmult poly c = map (* c) poly

eval :: Polynom -> Double -> Double
eval poly x = foldr evalFold 0 poly
  where evalFold aN acc = aN + x * acc

deriv :: Polynom -> Polynom
deriv [] = []
deriv poly = zipWith (*) [1..] $ tail poly
```

Algebraische Datentypen

Algebraische Datentypen

```
module DataExamples where


data Bool = True | False


data Category = Jackets | Pants | Shoes
data Filter
  = InSale
  | IsCategory Category
  | PriceRange Float Float
```

- Keyword data definiert *neuen* Typ
- „enum auf Meth“
- Ersetzt oft Vererbung im Entwurfsprozess

Algebraische Datentypen

```
module DataExamples2 where

data UserRole = Student | Mitarbeiter | Admin
data StudentId = UAccount String | MatNum Int

data List a = Null | Cons a (List a)
```

- Typen werden definiert als Menge von Werten
- Auch: Summentypen
  - Produkttypen: `struct`, Tupel, etc.
  - Menge der möglichen Werte: Summe der möglichen Werte der Konstruktoren

Übungsaufgabe: Liste implementieren

```
module List where

data List a = Null | Cons a (List a)
    deriving (Show)

map' f Null = Null
map' f (Cons x xs) = Cons (f x) (map' f xs)

filter' f Null = Null
filter' f (Cons x  xs) = if f x then (Cons x (filter' f x
else filter' f xs

all' f Null = True
all' f (Cons x xs) = if f x then all' xs else False
```

Übungsaufgabe: Liste implementieren

```
module List where

data List a = Null | Cons a (List a)
    deriving (Show)

map' f Null = Null
map' f (Cons x xs) = Cons (f x) (map' f xs)

filter' f Null = Null
filter' f (Cons x  xs) = if f x then (Cons x (filter' f x
else filter' f xs

all' f Null = True
all' f (Cons x xs) = if f x then all' xs else False
```

Übungsaufgabe: Liste implementieren

```
module List where

data List a = Null | Cons a (List a)
    deriving (Show)

map' f Null = Null
map' f (Cons x xs) = Cons (f x) (map' f xs)

filter' f Null = Null
filter' f (Cons x  xs) = if f x then (Cons x (filter' f x
else filter' f xs

all' f Null = True
all' f (Cons x xs) = if f x then all' xs else False
```

Typklassen

Typklassen

```
module Classes where


max :: Ord a => a -> a -> a
max x y
  | x > y     = x
  | otherwise = y


findIndex :: Eq a => a -> [a] -> Int
findIndex element (x : xs)
  | x == element = 0
  | otherwise    = 1 + findIndex element xs
```

- Typklasse $\rightsquigarrow$ Einschränkung für einen Typ
  - `Ord a` $\rightsquigarrow$ as müssen sortierbar sein
  - `Eq a` $\rightsquigarrow$ as müssen gleich oder ungleich sein

Definition von Typklassen

```
module MyEq where

data UserRole = Student | Mitarbeiter

class MyEq a where
  is :: a -> a -> Bool

  isnt :: a -> a -> Bool
  x `isnt` y = not (x `is` y)

instance MyEq UserRole where
  Student     `is` Student     = True
  Mitarbeiter `is` Mitarbeiter = True
  _           `is` _           = False
```

Übungsaufgabe: Queue implementieren

```
module Queue where

import List

data Queue a = Queue (List a) (List a)

pushHead x (Queue front back) =
  Queue (Cons x front) back
pushLast x (Queue front back) =
  Queue front (Cons x back)
```

- `head :: Queue a -> a`
- `popHead :: Queue a -> Queue a`
- `last :: Queue a -> a`
- `popLast :: Queue a -> Queue a`

Übungsaufgabe: Typklasse Mappable

```
module Mappable where


class Mappable m where
  mmap :: (a -> b) -> m a -> m b
```

- Implementiert diese Typklasse für `List` und `Queue`
- `Mappable m` $\rightsquigarrow$ Ein `m` enthält Elemente, auf die eine Funktion angewendet werden kann.
- Bspw.:
  - `mmap (+1) Null = Null`
  - `mmap (*2) (Cons 1 (Cons 2 (Cons 3 Null))) = (Cons 2 (Cons 4 (Cons 6 Null)))`