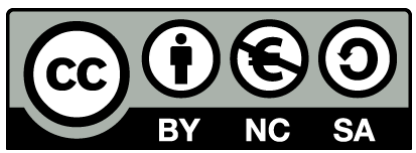
Vorlesung
Programmierung und Modellierung mit Haskell

6 Typklassen
7 Funktionen höherer Ordnung
(Version vom 14.05.2021)

François Bry

Sommersemester 2021



Zusammenfassung der letzten Vorlesung

4 Auswertung – Verzögerte Auswertung mit Konstruktoren

- Verzögerte Auswertung mit Konstruktoren

5 Typen

- Typprüfung
- Vordefinierte elementare Typen
- Vordefinierte algebraische Typen
- Funktionstypen
- Benutzer-definierte Typen
- Pattern Matching

Quiz Lernfortschritte

Welche Aussagen sind korrekt?

A. Der Ausdruck `(+) 1` ist ist in Weak Head Normalform (WHNF).
B. Der Ausdruck `(\x -> x * x) 55` ist in Weak Head Normalform (WHNF).
C. Der Ausdruck `take 2 [0..]` ist in Weak Head Normalform (WHNF).
D. Der Ausdruck `[0..]` ist in Weak Head Normalform (WHNF).
E. Die Auswertung von `head (tail [0..])` terminiert.
F. Die Auswertung von `tail [0..]` terminiert.
G. Der Ausdruck `if x > 2 then y else z` hat keinen Typ.
H. Die Auswertung von `1 == "1"` liefert `True`.
I. Die Auswertung von `1.0 == 1e0` liefert `True`.
J. Eine Funktion wie `\x -> x + 1` hat keinen Typ.

Antworten: https://lmu.onlineted.de

Inhalt der heutigen Vorlesung

6 Typklassen

- Wozu Typ-Klassen?
- Was ist eine Typ-Klasse?
- Der Polymorphismus von Haskell
- Grundlegende Typ-Klassen
- Eine Typ-Klasse und Instanzen davon definieren
- Die Typ-Klasse Monoid
- Die Typ-Klasse Foldable

7 Funktionen höherer Ordnung

- Was sind Funktionen höherer Ordnung?
- Bekannte Funktionen höherer Ordnung

Wozu Typ-Klassen?

```
-- laenge1.hs (nicht endrekursiv)
laenge :: [a] -> Int
laenge [] = 0
laenge (_:t) = 1 + laenge t

-- laenge2.hs (endrekursiv)
laenge :: [a] -> Int
laenge liste = hlaenge 0 liste

hlaenge :: Int -> [a] -> Int
hlaenge akk [] = akk
hlaenge akk (_:t) = hlaenge (akk+1) t
```

Um `laenge` auf `Integer` anzupassen, reicht es im Grunde aus,
`laenge' :: [a] -> Int`
durch
`laenge :: [a] -> Integer`
zu ersetzen. Die Funktionsdefinitionen bleiben dabei gleich. Sie müssen aber wiederholt werden und die Funktionsnamen müssen verschieden sein.

Wozu Typ-Klassen?

Die Typ-Klasse `Num` ermöglicht, eine einzige Funktion `laenge` für alle Zahlen-Typen zu definieren:

```
-- laenge3.hs
laenge :: Num a => [b] -> a
laenge liste = hlaenge 0 liste

hlaenge :: Num a => a -> [b] -> a
hlaenge akk [] = akk
hlaenge akk (_:t) = hlaenge (akk+1) t
```

`Num a =>` ist eine **Klassen-Constraint** für die Typ-Variable `a`: Sie kann nur durch einen Typ der Typklasse `Num` (wie `Int`, `Integer` oder `Double`) ersetzt werden.

In ghci `laenge3.hs` laden und das folgende aufrufen:

```
laenge [1,2,3,4] :: Int
laenge [1,2,3,4] :: Double
laenge [1,2,3,4] :: String
show (laenge [1,2,3,4])
:t laenge [1,2,3,4]
```

Was ist eine Typ-Klasse?

Eine Typ-Klasse

- spezifiziert eine Menge von Typen,
- spezifiziert eine Menge von Funktionen.

Jeder Typ einer Typ-Klasse implementiert jede Funktion der Typ-Klasse in seiner Weise.

Eine Typ-Klasse spezifiziert eine Schnittstelle für Typen, die ähnliche Funktionen (üblicherweise in verschiedenen Weise) implementieren.

Eine Typ-Klasse ermöglicht einen Polymorphismus, der über den Polymorphismus hinaus geht, der mit Typ-Variablen möglich ist.

Der Polymorphismus von Haskell

- **Parametrischer Polymorphismus** mit Typ-Variablen (wie z.B. `[a]`)
- **"Ad hoc"-Polymorphismus** mit Typ-Klassen: Eine Funktion kann verschiedene Typen haben, weil ihre Definition sich auf Variablen bezieht, die verschiedene Definitionen für verschiedene Typen haben.

Die folgende Funktion `laenge` ist in diesen beiden Weisen polymorph.

```
-- laenge3.hs
laenge :: Num a => [b] -> a
laenge liste = hlaenge 0 liste

hlaenge :: Num a => a -> [b] -> a
hlaenge akk [] = akk
hlaenge akk (_:t) = hlaenge (akk+1) t
```

(+) ist "Ad hoc"-polymorph: inlinecode(+) ist in verschiedenen Weisen für die verschiedenen Zahlen-Typen implementiert.

Klassen-Constraint

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

Steht `a` für ein Typ der Typklasse `Eq`, so gibt es eine Funktion `(==)` vom Typ `a -> a -> Bool`.

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

Steht `a` für ein Typ der Typklasse `Ord`, so gibt es eine Funktion `(>)` vom Typ `a -> a -> Bool`.

Links von `=>` steht ein Klassen-Constraint über eine oder mehrere Typ-Variablen, rechts ein polymorpher Typ, in dessen Definition diese Typ-Variablen vorkommen.

Grundlegende Typ-Klassen

Eq `==` `/=`

Ord `<` `>` `<=` `>=` `compare`

`compare` gibt GT (greater than), LT (less than) oder EQ (equal) zurück.

Nur Typen in Eq können Typen in Ord sein.

Show `show`

Die Werte eines Typs der Typklase Show haben eine String-Darstellung.

Grundlegende Typ-Klassen

`Num` Zahlen-Typen

In ghci aufrufen:

```
:t 2
2 :: Double
2 :: Int
2 :: Integer
```

Eine ganze Zahl ist eine polymorphe Konstante.

Eine Typ-Klasse und Instanzen davon definieren

```
-- binbaumklasse.hs
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses #-}

class Eq a => BinBaum b a where
    -- istIn oder istNichtIn muss definiert werden
    -- beide duerfen definiert werden
    istIn, istNichtIn :: a -> b a -> Bool
    istIn      wert baum = not (istNichtIn wert baum)
    istNichtIn wert baum = not (istIn    wert baum)
```

Eine Typ-Klasse und Instanzen davon definieren

```
data BBKM a = L | MK a (BBKM a) (BBKM a) deriving (Show)

instance Eq a => BinBaum BBKM a where
    istIn x (MK w _ _)           | x == w    = True
    istIn x (MK _ links rechts)  | otherwise =
                                   (istIn x links) || (istIn x rechts)
    istIn _ _                    | otherwise = False

data BBBM a = LL | MB a | K (BBBM a) (BBBM a)
              deriving (Show)

instance Eq a => BinBaum BBBM a where
    istIn x (MB w)               | x == w    = True
    istIn x (K links rechts)     | otherwise =
                                   (istIn x links) || (istIn x rechts)
    istIn _ _                    | otherwise = False
```

Die Typ-Klasse Monoid

Ein Monoid wird definiert durch:

- Einen Typ
- Ein Operator über diesen Typ mit den Eigenschaften:
  - der Operator hat ein neutrales Element.
  - der Operator ist assoziativ.

Der Operator eines Monoids kann – muss aber nicht – kommutativ sein.

```
-- vordefiniert
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

Die Typ-Klasse Monoid

Die Monoid-Gesetze:

1. `mempty ‘mappend‘ x` liefert `x` – `mempty` ist Links-Neutrum
2. `x ‘mappend‘ mempty` liefert `x` – `mempty` ist Rechts-Neutrum
3. `(x ‘mappend‘ y) ‘mappend‘ z` liefert dasselbe wie `x ‘mappend‘ (y ‘mappend‘ z)` – `mappend` ist assoziativ

Es ist nur dann sinnvoll, eine Instanz der Typ-Klasse `Monoid` zu definieren, wenn diese Gesetze erfüllt sind.

Die Typ-Klasse Monoid

Zahlen bilden Monoide sowohl für (+) wie für (*).
In `Data.Monoid` ähnlich wie folgt vordefiniert:

```
-- productsum.hs
newtype Sum a =  Sum { getSum :: a }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)
```

In ghci testen:

```
getSum $ Sum 1 `mappend` Sum 2
getSum $ Sum 1 `mappend` mempty
getSum $ mconcat [Sum 0, Sum 1, Sum 2, Sum 3]
```

$ wird verwendet um Klammern zu vermeiden: `f $ g x` druckt aus `f (g x)`.

```
($) :: (a -> b) -> (a -> b)
f $ x = f x
```

Die Typ-Klasse Monoid

Listen bilden ein Monoid für (++).

```
-- vordefiniert
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

In ghci testen:

```
[1,2] `mappend`[3,4,5]
[1, 2] `mappend` mempty
mconcat [[1,2], [3,4,5], [6]]
```

(++) ist nicht kommunikativ.

Die Monoid-Bezeichner `mempty`, `mappend` und `mconcat` wurden nach den Listen-Bezeichnern `[]`, `append` und `concat` gebildet.

Die Typ-Klasse Monoid

`Maybe` zur Fehlerbehandlung:

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

In ghci testen:

```
Just "abc" `mappend` Nothing
Just "abc" `mappend` mempty
Just "" `mappend` mempty

mconcat [Just "abc", Nothing, Just "def"]
```

Quiz `Maybe`

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

`Maybe Int` und `Maybe String` sind beide Monoide, d.h.

- `Nothing` ist Links- und Rechtsneutrum für `mappend`,
- `mappend` ist assoziativ.

Ist die Aussage korrekt?

A. Ja
B. Nein

Antworten: https://lmu.onlineted.de

Die Typ-Klasse Foldable

Monoide ermöglichen, Daten zu „falten“, d.h. aggregieren. Dafür sind aber Funktionen nötig.

Die Typ-Klasse `Foldable` wird für Monoide verwendet, die Funktionen `foldr` (falten vom rechts her) und `foldl` (falten vom links her) besitzen.

In ghci testen:

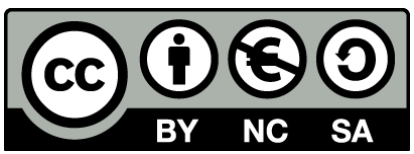
```
import qualified Data.Foldable as F
:t foldr
:t F.foldr

foldr (+) 0 [1,2,3]
F.foldr (+) 0 [1,2,3]

F.foldl (||) False (Just True)
F.foldl (&&) True (Just True)

F.foldl (+) 2 (Just 40)
F.foldl (*) 2 (Just 40)
```

Vorlesung
Programmierung und Modellierung mit Haskell

7 Funktionen höherer Ordnung

François Bry

Sommersemester 2021



Inhalt

7 Funktionen höherer Ordnung

- Was sind Funktionen höherer Ordnung?
- Bekannte Funktionen höherer Ordnung
- Vorteile von Curried-Funktionen

Was sind Funktionen höherer Ordnung?

In funktionalen Programmiersprachen sind Funktionen Werte, die wie andere Werte auch Parameter oder Ergebnisse von Funktionsanwendungen sein können.

Eine Funktion höherer Ordnung ist eine Funktion, die

- eine Funktion als Parameter hat,
- und/oder eine Funktion als Wert zurückgibt.

Was sind Funktionen höherer Ordnung?

Die Bezeichnung Funktion höherer Ordnung wird wie folgt erklärt:

- Ordnung 0: Konstanten wie `1`, `'b'`, und `[]`.
- Ordnung 1: Funktionen, deren Parameter und Werte der Ordnung 0 sind.
- Höhere Ordnung: Alle weiteren Funktionen.

Was sind Funktionen höherer Ordnung?

```
zweimalAnwenden f = f . f
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
flip :: (a -> b -> c) -> b -> a -> c
```

In ghci das Folgende ausführen:

```
(zweimalAnwenden (\x -> 3*x)) 5

let f x y = 2*x + 3*y
f 1 2
uncurry f (1, 2)
let g (x, y) = 2*x + 3*y
g (1, 2)
curry g 1 2
f 2 1
flip f 1 2
```

Quiz `flip`

```
f x y = 2*x + 3*y
```

Welche Aussagen sind korrekt?

A. `f 1` ist eine Funktion höherer Ordnung.

B. `(\x -> f x)` ist eine Funktion höherer Ordnung.

C. `flip f` ist eine Funktion höherer Ordnung.

D. `flip` ist eine Funktion höherer Ordnung.

Antworten: https://lmu.onlineted.de

Quiz Funktionskomposition

Ist die Funktionskomposition `(.)` eine Funktion höhere Ordnung?

A. Ja
B. Nein

Antworten: `https://lmu.onlineted.de`

Bekannte Funktionen höherer Ordnung

`map` ist eine vordefinierte Funktion, die der folgenden Funktion `myMap` entspricht.

```
-- myMap.hs
myMap :: (a -> b) -> [a] -> [b]
myMap f []     = []
myMap f (x:xs) = (f x):myMap f xs
```

In ghci wie testen:

```
myMap succ [0, 1, 2]
myMap length ["ab", "cde", "fghij"]
map length ["ab", "cde", "fghij"]

import Data.Char
toUpper 'a'
myMap toUpper "abcd"
map toUpper "abcd"
```

Bekannte Funktionen höherer Ordnung

`filter` ist eine vordefinierte Funktion, die der folgenden Funktion `myFilter` entspricht.

```
-- myFilter.hs
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter f [] = []
myFilter f (x:xs) | f x       = x : myFilter f xs
myFilter f (x:xs) | otherwise = myFilter f xs
```

In ghci testen:

```
myFilter (\x -> x `mod` 2 == 0) [0,1,2,3,4]
filter (\x -> x `mod` 2 == 0) [0,1,2,3,4]

import Data.Char
myFilter isLower "aBcDeF"
filter isLower "aBcDeF"
```

Quiz `filter`

Ist `filter (\x -> x >= 0)` ist eine Funktion höherer Ordnung.?

A. Ja

B. Nein

Antworten: `https://lmu.onlineted.de`

Bekannte Funktionen höherer Ordnung

`foldl` ist eine vordefinierte Funktion, die der folgenden Funktion entspricht:

```
-- myFoldl.hs
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl f akk []     = akk
myFoldl f akk (x:xs) = myFoldl f (f akk x) xs
```

In ghci wie folgt testen:

```
myFoldl (+) 0 [1, 2, 3]
foldl (+) 0 [1, 2, 3]

myFoldl (-) 0 [1, 2, 3]
foldl (-) 0 [1, 2, 3]
```

`foldl` wird auch `reduce` genannt.

Quiz `foldl`

```
-- myFoldl.hs
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl f akk []     = akk
myFoldl f akk (x:xs) = myFoldl f (f akk x) xs
```

Welche Aussagen sind korrekt?

A. `myFoldl` ist rekursiv.

B. `myFoldl` ist endrekursiv.

C. `myFoldl` ist eine Funktion höherer Ordnung.

Antworten: https://lmu.onlineted.de

Bekannte Funktionen höherer Ordnung

`foldr` ist eine vordefinierte Funktion, die der folgenden Funktion `myFoldr` entspricht:

```
-- myFoldr.hs
myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr f akk []     = akk
myFoldr f akk (x:xs) = f x (myFoldr f akk xs)
```

In ghci wie folgt testen:

```
myFoldr (+) 0 [1, 2, 3]
foldr (+) 0 [1, 2, 3]
foldl (+) 0 [1, 2, 3]

myFoldr (-) 0 [1, 2, 3]
foldr (-) 0 [1, 2, 3]
foldl (-) 0 [1, 2, 3]
```

Quiz `foldr`

```
-- myFoldr.hs
myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr f akk []     = akk
myFoldr f akk (x:xs) = f x (myFoldr f akk xs)
```

A. `myFoldr` ist rekursiv.

B. `myFoldr` ist endrekursiv.

C. `myFoldr` ist eine Funktion höherer Ordnung.

Antworten: https://lmu.onlineted.de

Bekannte Funktionen höherer Ordnung

Weil `foldr` nicht endrekursive, verursacht
`foldr (-) 0 [0..1000000000]`
einen Pufferüberlauf (stack overflow).

Für Hinweise zu Pufferüberläufe bei der Verwendung von `foldl` und `foldr` sowie Lösungen dazu siehe den Artikel „Stack Overflow“ im Haskell Wiki (`https://wiki.haskell.org/Stack_overflow`)

Zusammenfassung der heutigen Vorlesung

6 Typklassen

- Wozu Typ-Klassen?
- Was ist eine Typ-Klasse?
- Der Polymorphismus von Haskell
- Grundlegende Typ-Klassen
- Eine Typ-Klasse und Instanzen davon definieren
- Die Typ-Klasse Monoid
- Die Typ-Klasse Foldable

7 Funktionen höherer Ordnung

- Was sind Funktionen höherer Ordnung?
- Bekannte Funktionen höherer Ordnung