

Intermediate Functional Programming in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

- 1 Typen
- 2 Typklassen
- 3 Typklassen auf Maybe
- 4 Typklassen auf Either
- 5 Typklassen auf Listen
- 6 Praktische Arbeit

Primitive Datentypen sind eine Annotation, wie die Bits im Speicher interpretiert werden sollen.

Primitive Datentypen sind eine Annotation, wie die Bits im Speicher interpretiert werden sollen.

Einige primitive Datentypen sollten euch aus anderen Programmiersprachen schon bekannt sein:

- Zahlen (z.B. Int, Integer, Float, Double, ...)
- Zeichenketten (z.B. String, UTF-8-Strings, ...)
- Bool

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:

(a,k,v steht hier jeweils für einen (primitiven) Datentypen)

- Liste von a
- Hashmap von k und v
- Vektor von a
- Tree von a
- Zusammengesetzte Typen (z.B. Structs in C/C++)

Es gibt auch Datentypen höherer Ordnung. Diese zeichnen sich dadurch aus, dass sie alleine nicht vollständig sind.

Auch hier sollten schon einige bekannt sein:

(a,k,v steht hier jeweils für einen (primitiven) Datentypen)

- Liste von a
- Hashmap von k und v
- Vektor von a
- Tree von a
- Zusammengesetzte Typen (z.B. Structs in C/C++)

Im folgenden gehen wir auf 2 wesentliche zusammengesetzte Typen in Haskell ein: Maybe und Either.

Einen neuen Datentypen definieren wir in Haskell mit dem Keyword `data`:

```
data Maybe a = Nothing  
             | Just a
```


Einen neuen Datentypen definieren wir in Haskell mit dem Keyword `data`:

```
data Maybe a = Nothing  
             | Just a
```

Was hat das für einen Sinn?

Einen neuen Datentypen definieren wir in Haskell mit dem Keyword `data`:

```
data Maybe a = Nothing  
             | Just a
```

Was hat das für einen Sinn?

Maybe gibt das Ergebnis einer Berechnung an, die fehlschlagen kann.

In klassischen Sprachen wird hier meist ein „abgesprochener“ Fehlerzustand zurückgegeben (0, -1, null, ...). In Haskell wird dies über den Rückgabetyt deutlich gemacht.

Nachteile

- Ein neuer Datentyp, den man kennen muss

Nachteile

- Ein neuer Datentyp, den man kennen muss

Vorteile

- keine Absprachen, die man vergessen kann
- einheitliche Behandlung aller Fälle
- mehrere möglicherweise fehlschlagende Operationen gruppieren und nur solange evaluieren, bis die erste fehlschlägt oder alle erfolgreich sind

Beispiel: Finden eines Elementes in einer Liste

Beispiel: Finden eines Elementes in einer Liste

```
ghci > import Data.List
ghci > findIndex (== 5) [1..10]
Just 4                -- [1..10] !! 4 => 5
```

```
ghci > findIndex (== 1000) [1..10]
Nothing
```

Beispiel: Finden eines Elementes in einer Liste

```
ghci > import Data.List
ghci > findIndex (== 5) [1..10]
Just 4                -- [1..10] !! 4 => 5
```

```
ghci > findIndex (== 1000) [1..10]
Nothing
```

Da wir 1000 in der Liste der Zahlen von 1-10 nicht finden können, haben wir keinen gültigen Index, daher bekommen wir ein Nothing.

```
data Either a b = Left a
                | Right b
```



```
data Either a b = Left a  
                | Right b
```

Was hat das für einen Sinn?

```
data Either a b = Left a  
                | Right b
```

Was hat das für einen Sinn?

Either benutzt man, wenn man ein erwartetes Ergebnis `Right b` vom Typen `b` hat **oder** einen Fehlerzustand `Left a` vom Typen `a`. Meistens ist das erste Argument `String` um eine lesbare Fehlermeldung zu bekommen.

```
data Either a b = Left a  
                | Right b
```

Was hat das für einen Sinn?

Either benutzt man, wenn man ein erwartetes Ergebnis `Right b` vom Typen `b` hat **oder** einen Fehlerzustand `Left a` vom Typen `a`. Meistens ist das erste Argument `String` um eine lesbare Fehlermeldung zu bekommen.

Einfach zu merken: „`Right`“ ist der „richtige“ Fall.

Beispiele für eine Benutzung von Either:

```
parse5 :: String -> Either String Int
```

```
parse5 "5" = Right 5
```

```
parse5 _   = Left "Could not parse 5"
```

```
parse5 "5"    -- Right 5
```

```
parse5 "abc"  -- Left "Could not parse 5"
```

Viele Typen haben ähnliche oder gleiche Eigenschaften. Diese Eigenschaften fasst man zu Typklassen zusammen.

Viele Typen haben ähnliche oder gleiche Eigenschaften. Diese Eigenschaften fasst man zu Typklassen zusammen.

- Zahlen kann man alle verrechnen, auch wenn z.B. Int und Double verschiedene Typen haben
- Listen, Vektoren, Arrays haben alle Elemente, über die man z.B. iterieren kann
- Maybe, Either, Listen, etc. haben (vielleicht) Elemente, die man verändern kann

Viele Typen haben ähnliche oder gleiche Eigenschaften. Diese Eigenschaften fasst man zu Typklassen zusammen.

- Zahlen kann man alle verrechnen, auch wenn z.B. Int und Double verschiedene Typen haben
- Listen, Vektoren, Arrays haben alle Elemente, über die man z.B. iterieren kann
- Maybe, Either, Listen, etc. haben (vielleicht) Elemente, die man verändern kann

Warnung: Typklassen haben nichts mit den Klassen der Objektorientierung zu tun, sondern eher mit Templates und abstrakten Klassen

```
class Eq a where
    (==) :: a -> a -> Bool
    --or (/=) :: a -> a -> Bool

class Eq a => Ord a where
    (<=) :: a -> a -> Bool
    -- definiert automatisch: compare, >=, <, >, max, min
```



```
class Eq a where
    (==) :: a -> a -> Bool
--or (/=) :: a -> a -> Bool
```

```
class Eq a => Ord a where
    (<=) :: a -> a -> Bool
-- definiert automatisch: compare, >=, <, >, max, min
```

Im folgenden stellen wir 3 Zentrale Typklassen vor: Functor,
Applicative, Monad

Ein Funktor F lässt sich auf jedem Datentypen definieren, der sowas wie einen „Inhalt“ hat.

Ein Funktor F lässt sich auf jedem Datentypen definieren, der sowas wie einen „Inhalt“ hat.

Genauer: Er wird definiert über die Funktion `fmap`, die es erlaubt eine Funktion auf den Inhalt anzuwenden.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

`f` heisst hier **der Kontext**, in dem `a` existiert.

Ein Funktor F lässt sich auf jedem Datentypen definieren, der sowas wie einen „Inhalt“ hat.

Genauer: Er wird definiert über die Funktion `fmap`, die es erlaubt eine Funktion auf den Inhalt anzuwenden.

```
class Functor f where  
    fmap :: (a -> b) -> f a -> f b
```

`f` heisst hier **der Kontext**, in dem `a` existiert.

Spoiler: Liste ist ein Funktor. Maybe auch. Wieso?

Hilfreiche Analogie:

Für den Anfang kann man sich den **Kontext** als eine Kiste vorstellen, in der etwas liegt.

Hilfreiche Analogie:

Für den Anfang kann man sich den **Kontext** als eine Kiste vorstellen, in der etwas liegt.

`fmap` wertet also nur Funktionen, die auf dem *Inhalt* der Kiste funktionieren würden, zu Funktionen auf, die auf *Kisten mit Dingen* funktionieren.

Hilfreiche Analogie:

Für den Anfang kann man sich den **Kontext** als eine Kiste vorstellen, in der etwas liegt.

`fmap` wertet also nur Funktionen, die auf dem *Inhalt* der Kiste funktionieren würden, zu Funktionen auf, die auf *Kisten mit Dingen* funktionieren.

Man kann `fmap` daher auch etwas anders Klammern:

```
fmap :: (a -> b) -> (f a -> f b)
```

`fmap` nimmt somit eine Funktion und gibt eine neue Funktion zurück, die auf dem Kontext `f` funktioniert.

Functor-Instanz von Maybe:

```
instance Functor Maybe where
    fmap f (Just a) = Just (f a)
    fmap _ Nothing  = Nothing
```


Functor-Instanz von Maybe:

```
instance Functor Maybe where
    fmap f (Just a) = Just (f a)
    fmap _ Nothing  = Nothing
```

Functor-Instanz von Either:

```
instance Functor Either where
    fmap f (Right a) = Right (f a)
    fmap _ (Left b)  = Left b
```

Functor-Instanz von Maybe:

```
instance Functor Maybe where
    fmap f (Just a) = Just (f a)
    fmap _ Nothing  = Nothing
```

Functor-Instanz von Either:

```
instance Functor Either where
    fmap f (Right a) = Right (f a)
    fmap _ (Left b)  = Left b
```

Functor-Instanz von Listen:

```
instance Functor [] where
    fmap f (x:xs) = f x : (fmap f xs)
    fmap _ []     = []
```

fmap auf Listen ist einfach das bekannte map

Beispiel:

```
ghci > fmap (+1) (Just 3)
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
```

```
      Just 4
```

```
ghci > fmap (+1) Nothing
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
```

```
      Just 4
```

```
ghci > fmap (+1) Nothing
```

```
      Nothing
```

```
ghci > fmap (+1) [1..10]
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
```

```
      Just 4
```

```
ghci > fmap (+1) Nothing
```

```
      Nothing
```

```
ghci > fmap (+1) [1..10]
```

```
      [2,3,4,5,6,7,8,9,10,11]
```

```
ghci > fmap (+1) (Right 2)
```

Beispiel:

```
ghci > fmap (+1) (Just 3)  
      Just 4
```

```
ghci > fmap (+1) Nothing  
      Nothing
```

```
ghci > fmap (+1) [1..10]  
      [2,3,4,5,6,7,8,9,10,11]
```

```
ghci > fmap (+1) (Right 2)  
      Right 3
```

```
ghci > fmap (+1) (Left 2)
```

Beispiel:

```
ghci > fmap (+1) (Just 3)
      Just 4
```

```
ghci > fmap (+1) Nothing
      Nothing
```

```
ghci > fmap (+1) [1..10]
      [2,3,4,5,6,7,8,9,10,11]
```

```
ghci > fmap (+1) (Right 2)
      Right 3
```

```
ghci > fmap (+1) (Left 2)
      Left 2
```


Funktoren sind mathematische Objekte mit Eigenschaften. Damit der Compiler auch die richtigen Optimierungen machen kann, muss jeder Functor folgende Regeln erfüllen:

Funktoren sind mathematische Objekte mit Eigenschaften. Damit der Compiler auch die richtigen Optimierungen machen kann, muss jeder Functor folgende Regeln erfüllen:

```
-- Strukturerhaltung  
fmap id = id
```

Die Datenstruktur darf sich nicht ändern.

Funktoren sind mathematische Objekte mit Eigenschaften. Damit der Compiler auch die richtigen Optimierungen machen kann, muss jeder Funktor folgende Regeln erfüllen:

```
-- Strukturerhaltung
```

```
fmap id = id
```

Die Datenstruktur darf sich nicht ändern.

```
-- Composability
```

```
fmap (f . g) = fmap f . fmap g
```

Mehrere `fmaps` hintereinander dürfen zusammengefasst werden, ohne, dass sich das Ergebnis ändert.

Negativbeispiel

```
fmap' f []      = []  
fmap' f (a:as) = (f a):a:(fmap' f as)
```

Negativbeispiel

```
fmap' f [] = []
```

```
fmap' f (a:as) = (f a):a:(fmap' f as)
```

```
fmap' id [1,2,3] = [1,1,2,2,3,3] /= [1,2,3]
```

Negativbeispiel

```
fmap' f Nothing = Nothing  
fmap' f (Just a) = Just (f (f a))
```

Negativbeispiel

```
fmap' f Nothing = Nothing
fmap' f (Just a) = Just (f (f a))

(fmap' (+1) . fmap' (*2)) (Just 1)
= fmap' (+1) (Just ((*2) ((*2) 1)))
= fmap' (+1) (Just 4)
= Just 6

(fmap' ((+1).(*2)) (Just 1)
= Just (((+1).(*2)).(+1).(*2)) 1)
= Just 7
```

Applicative funktioniert ähnlich zu Funktor. Hierbei kann man auch mit Funktionen in einem Kontext arbeiten.

Applicative funktioniert ähnlich zu Funktor. Hierbei kann man auch mit Funktionen in einem Kontext arbeiten.

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Applicative funktioniert ähnlich zu Funktor. Hierbei kann man auch mit Funktionen in einem Kontext arbeiten.

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

pure bringt etwas in den Standardkontext (z.B. Liste mit 1 Element, Just x, Right x, ..).

<*> ist fast ein fmap, nur dass die Funktion auch in demselben Kontext liegt.

Functor-Instanz von Maybe:

```
instance Applicative Maybe where
    pure                = Just
    Nothing <*> _       = Nothing
    (Just f) <*> something = fmap f something
```

Functor-Instanz von Maybe:

```
instance Applicative Maybe where
    pure                = Just
    Nothing <*> _       = Nothing
    (Just f) <*> something = fmap f something
```

Functor-Instanz von Either:

```
instance Applicative Either where
    pure                = Right
    (Right f) <*> something = fmap f something
    (Left e) <*> _       = Left e
```

Functor-Instanz von Maybe:

```
instance Applicative Maybe where
    pure                = Just
    Nothing <*> _      = Nothing
    (Just f) <*> something = fmap f something
```

Functor-Instanz von Either:

```
instance Applicative Either where
    pure                = Right
    (Right f) <*> something = fmap f something
    (Left e) <*> _      = Left e
```

Functor-Instanz von Listen:

```
instance Functor [] where
    pure a = [a]
    (f:fs) <*> x = fmap f x ++ (fs <*> x)
    [] <*> _ = []
```

```
ghci > import Control.Applicative  
ghci > Just (+1) <*> Just 3
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
      Just 4
ghci > Nothing <*> Just 3
```

```
ghci > import Control.Applicative
```

```
ghci > Just (+1) <*> Just 3  
      Just 4
```

```
ghci > Nothing <*> Just 3  
      Nothing
```

```
ghci > pure (+1) <*> Right 2
```



```
ghci > import Control.Applicative
```

```
ghci > Just (+1) <*> Just 3  
      Just 4
```

```
ghci > Nothing <*> Just 3  
      Nothing
```

```
ghci > pure (+1) <*> Right 2  
      Right 3
```

```
ghci > pure (+1) <*> Just 2
```

```
ghci > import Control.Applicative
```

```
ghci > Just (+1) <*> Just 3  
      Just 4
```

```
ghci > Nothing <*> Just 3  
      Nothing
```

```
ghci > pure (+1) <*> Right 2  
      Right 3
```

```
ghci > pure (+1) <*> Just 2  
      Just 3
```

```
ghci > [(+1), (*2)] <*> [1..5]
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
      Just 4
ghci > Nothing <*> Just 3
      Nothing
ghci > pure (+1) <*> Right 2
      Right 3
ghci > pure (+1) <*> Just 2
      Just 3
ghci > [(+1), (*2)] <*> [1..5]
      [2,3,4,5,6,2,4,6,8,10]
ghci > pure (*) <*> [1..3] <*> [1..3]
```

```
ghci > import Control.Applicative
ghci > Just (+1) <*> Just 3
      Just 4
ghci > Nothing <*> Just 3
      Nothing
ghci > pure (+1) <*> Right 2
      Right 3
ghci > pure (+1) <*> Just 2
      Just 3
ghci > [(+1), (*2)] <*> [1..5]
      [2,3,4,5,6,2,4,6,8,10]
ghci > pure (*) <*> [1..3] <*> [1..3]
      [1,2,3,2,4,6,3,6,9]
```

Wozu Monaden?

Wozu Monaden?

- Man kann auch ohne funktional programmieren

Wozu Monaden?

- Man kann auch ohne funktional programmieren
- Monaden verhalten sich wie ein Semikolon in anderen Sprachen

Wozu Monaden?

- Man kann auch ohne funktional programmieren
- Monaden verhalten sich wie ein Semikolon in anderen Sprachen
- Monaden arbeiten im Hintergrund

Beispiel

```
f                :: Maybe Header
getInbox         :: Maybe Folder
getFirstMail    :: Folder -> Maybe Mail
getHeader       :: Mail -> Maybe Header
```

Beispiel

```
f                :: Maybe Header
getInbox         :: Maybe Folder
getFirstMail    :: Folder -> Maybe Mail
getHeader       :: Mail -> Maybe Header
```

Ohne Monaden:

```
f = case getInbox of
    (Just folder) ->
        case getFirstMail folder of
            (Just mail) ->
                case getHeader mail of
                    (Just head) -> return head
                    Nothing      -> Nothing
            Nothing      -> Nothing
    Nothing             -> Nothing
```

Beispiel

```
f                :: Maybe Header  
getInbox         :: Maybe Folder  
getFirstMail    :: Folder -> Maybe Mail  
getHeader       :: Mail -> Maybe Header
```

Mit Monaden:

```
f = do  
    folder <- getInbox  
    mail   <- getFirstMail folder  
    header <- getHeader mail  
    return header
```

foo

foo

foo

foo

foo

foo

foo

foo

foo

foo

foo

Wie kriege ich sachen aus einem `f a`? -> Alle Fälle behandeln mit `case`.

Wie kriege ich sachen aus einem Maybe a? `case x of Just a -> foo;
Nothing -> bar`

Wie kriege ich sachen aus einem Either a? case x of Left a -> foo;
Right b -> bar