Funktionales Programmieren

Teil 3

Carl Philipp Reh

Universität Siegen

22. Mai 2020

Functor

Die Funktion `map :: (a -> b) -> [a] -> [b]` bekommt eine Funktion `f :: a -> b` und wendet diese auf alle Elemente einer Liste an. Zum Beispiel gilt

```
map show [1,2] = ["1","2"]
map (\x -> mod x 2 == 0) [1,2,3]
   = [False,True,False]
```

Dies funktioniert nicht bloß auf Listen, sondern auf vielen verschiedenen Datenstrukturen, die Elemente „enthalten“. Hierfür hat Haskell eine eigene Typklasse:

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

Die Implementierung für Listen ist:

```
instance Functor [] where
  fmap = map
```

Functor

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

Da hier `f a` und `f b` gebildet werden, muss `f :: * -> *` gelten, was zum Beispiel passend ist für `[] :: * -> *`.
Ein Functor ist also ein Typkonstruktor mit Kind `* -> *`, d.h. eine Abbildung von Typen auf Typen, und eine Funktion, die Funktionen vom Typ `a -> b` auf Funktionen vom Typ `f a -> f b` abbildet.

Functor

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

Beispielsweise sieht die Implementierung für `Maybe` folgendermaßen aus:

```
instance Functor Maybe where
  fmap f m = case m of
               Nothing -> Nothing
               Just x -> Just (f x)
```

Zum Beispiel gilt

```
fmap (\x -> x + 1) Nothing = Nothing
fmap (\x -> x + 1) (Just 3) = Just 4
```

Functor

Anwendungsbeispiel: Wir wollen ausrechnen, ob ein Benutzer volljährig ist. Dieser gibt sein Alter als String ein, den wir erst umwandeln müssen mit der Funktion

```
stringToInt :: String -> Maybe Int
```

Diese gibt `Nothing` zurück, wenn der String kein gültiger `Int` ist. Zum Testen, ob jemand mindestens 18 ist, definieren wir

```
isAdult :: Int -> Bool
isAdult x  = x >= 18
```

Dann gilt zum Beispiel

```
fmap isAdult (stringToInt "abc") = Nothing
fmap isAdult (stringToInt "10") = Just False
fmap isAdult (stringToInt "20") = Just True
```

Applicative

`Applicative` ist eine Erweiterung von `Functor`, die es erlaubt, über mehr als einen Wert auf einmal zu „mappen“. Als Beispiel wollen wir ausgeben, ob ein Benutzer älter ist als der andere. Dazu definieren wir

```
isOlderThan :: Int -> Int -> Bool
```

Mit `fmap` auf `Maybe` erhalten wir

```
fmap isOlderThan
  :: Maybe Int -> Maybe (Int -> Bool)
```

Um damit weiter zu arbeiten, brauchen wir eine Funktion

```
Maybe (Int -> Bool) -> Maybe Int
                    -> Maybe Bool
```

die uns von `Applicative` bereitgestellt wird:

```
(<*>) :: f (a -> b) -> f a -> f b
```

Applicative

Die volle Definition von `Applicative` ist

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

Die Funktion `pure` „bettet einen Wert ein“.
Die Implementierung für `Maybe` sieht folgendermaßen aus

```
instance Applicative Maybe where
  pure = Just
  f <*> m = case f of
    Nothing -> Nothing
    Just f' -> case m of
      Nothing -> Nothing
      Just m' -> Just (f' m')
```

Es gilt also für `g :: a -> b` und `x :: a`, dass

```
Just g <*> Just x = Just (g x)
```

Applicative

Wir können in unserem Beispiel dann schreiben:

```
compareAges ::
  String -> String -> Maybe Bool
compareAges x y =
  (fmap isOlderThan) (stringToInt x)
  <*> (stringToInt y)
```

Um diesen Code besser zu verstehen, ist es wichtig, sich die Typen, die auftreten, anzuschauen: Zunächst haben wir, dass

```
fmap isOlderThan ::
  Maybe Int -> Maybe (Int -> Bool)
```

Dieses wenden wir dann auf `stringToInt x :: Maybe Int` an und erhalten `Maybe (Int -> Bool)`. Als Nächstes übergeben wir dies zusammen mit `stringToInt y :: Maybe Int` an `<*>` und erhalten `Maybe Bool`.

Applicative

`Applicative` erlaubt es, statt `<*>` die Funktion `liftA2` zu implementieren:

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

Als Beispiel betrachten wir wieder `Maybe`:

```
instance Applicative Maybe where
  liftA2 f m m' = case m of
    Nothing -> Nothing
    Just x -> case m' of
      Nothing -> Nothing
      Just x' -> Just (f x x')
```

Wir können dann auch schreiben

```
compareAges x y =
  liftA2 isOlderThan
    (stringToInt x) (stringToInt y)
```

Applicative

Als Anwendungsbeispiel von `Applicative` betrachten wir die Funktion `sequenceA` auf Listen:

```
sequenceA :: Applicative f => [f a] -> f [a]
```

Mit `Maybe` ergibt sich

```
sequenceA :: [Maybe a] -> Maybe [a]
```

Hierbei gilt, dass `sequenceA l = Nothing` genau dann, wenn mindestens ein Element in `l` `Nothing` ist.
Beispielsweise gilt

```
sequenceA [Nothing, Just 1] = Nothing
sequenceA [Just 1, Just 2] = Just [1,2]
```

Wichtig: Die Implementierung von `sequenceA` auf Listen funktioniert mit jedem `Applicative`.

Monad

`Monad` ist eine Erweiterung von `Applicative`, die es erlaubt, „sequentielle Abhängigkeit“ auszudrücken.
Als Beispiel wollen wir die Fehlerbehandlung beim Einlesen eines Alters erweitern. Wir wollen `Int` durch `Nat` ersetzen, sodass das Alter nicht mehr negativ sein kann. Zum Konvertieren von `Int` zu `Nat` verwenden wir die Funktion

```
intToNat :: Int -> Maybe Nat
```

Die Funktion `stringToInt` liefert uns `Maybe Int`. Dieses wollen wir nur an `intToNat` übergeben, wenn es nicht `Nothing` ist. Wir brauchen also eine Funktion

```
Maybe Int -> (Int -> Maybe Nat) -> Maybe Nat
```

Diese wird uns von `Monad` bereitgestellt:

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

Monad

Die allgemeine Definition von `Monad` ist

```
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b
```

Zum Beispiel sieht die Implementierung von `Maybe` so aus:

```
instance Monad Maybe where
  m >>= f = case m of
    Nothing -> Nothing
    Just x -> f x
```

Die Funktion `f` kann nur einen Wert produzieren, wenn `m` schon nicht `Nothing` ist. Diese „sequentielle Abhängigkeit“ kann man nicht mit `Applicative` alleine ausdrücken. Zum Beispiel gilt:

```
stringToInt "x" >>= intToNat = Nothing
stringToInt "-5" >>= intToNat = Nothing
stringToInt "2" >>= intToNat
  = Just (Succ (Succ Zero))
```

Monad

Als Anwendungsbeispiel von `Monad` betrachten wir die Funktion

```
join :: Monad f => f (f a) -> f a
```

Diese „entfernt“ eine Schicht eines Typkonstruktors. Zum Beispiel erhalten wir für `Maybe` und `[]`, dass

```
join (Just (Just 10)) = Just 10
join (Just Nothing) = Nothing
join [[]] = []
join [[1,2],[],[3,4],[5]] = [1,2,3,4,5]
```

Auch hier gilt natürlich, dass `join` mit jeder Monade funktioniert.