Funktionale Programmierung

Functor, Applicative und Monad

1. Functor
2. Applicative Functor
   - Beispiel
   - Applicative Regeln
3. Monad
   - Beispiel
   - Monad Regeln

Functor

Wir kennen bereits die Functor Typklasse.

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

Die Funktion fmap hat auch eine infix Variante:

```haskell
(<$>) = fmap
```

Beispiel:

```haskell
(+1) <$> [1,2,3] == [2,3,4]
```

Functor

Wir kennen auch bereits die zur Functor Klasse gehörenden Regeln:

- Identität

```
fmap id = id
id <$> x = x
```

- Komposition

```
fmap (f . g) = (fmap f) . (fmap g)
(f . g) <$> x = f <$> (g <$> x)
```

Applicative Functor

Wir haben eine Funktion, die den "Vorgänger" einer natürlichen Zahl berechnet:

```haskell
predec :: Int -> Maybe Int
predec x
    | x <= 1    = Nothing
    | otherwise = Just $ x - 1
```

Weil Maybe ein Functor ist, können wir diese Funktion angenehm mit Funktionen kombinieren, die eigentlich einen `Int` statt eines `Maybe Int` konsumieren.

```haskell
-- f x = 2 * (x - 1)
f x = (*2) <$> predec x
```

Applicative Functor

Können wir die Funktion `predec` auch im Kontext einer zweistelligen Funktion verwenden?

```
1 -- h x y = (x - 1) * (y - 1)
2 h x y = (*) (predec x) (predec y)
```

funktioniert nicht, weil wir `*` nicht direkt auf `Int` anwenden können.

```
1 h x y = (*) <$> (predec x) (predec y)
```

geht ebenfalls nicht, weil

```
1 (*) <$> (predec x) :: Maybe (Int -> Int)
2 predec y :: Maybe Int
```

Applicative Functor

Wir brauchen eine Funktion vom Typ

```
(<*>) :: Maybe (Int -> Int) -> Maybe Int ->
   Maybe Int
```

dann können wir (zumindest von den Typen her) unsere Funktion `h` wie folgt schreiben:

```
h x y = (*) <$> predec x <*> predec y
```

Applicative Functor

Die Applicative Functor Klasse

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

Die Applicative Instanz von Maybe

```haskell
pure = Just
Just f <*> x  = f <$> x
Nothing <*> _ = Nothing
```

bietet genau was wir brauchen.

Applicative Functor

Sobald wir zweistellige Funktionen beherrschen, beherrschen wir *n* stellige Funktionen:

```
1 mul3 x y z = x * y * z
2
3 -- d x y z = (x - 1) * (y - 1) * (z - 1)
4 d x y z =
5     mul3 <$> predec x <*> predec y <*> predec z
```

Applicative Functor

Beispiel

```haskell
data User = User
    { uName  :: String
    , uEmail :: String
    , uCity  :: String
    }

type Profile = [(String, String)]

petersProfile :: Profile
petersProfile =
    [ ("name", "peter")
    , ("email", "peter@peter.com")
    , ("city", "zueri")
    ]
```

Applicative Functor

Beispiel

```haskell
myLookup :: String -> Profile -> Maybe String
myLookup str [] = Nothing
myLookup str ((key, value):assocs)
    | str == key = Just value
    | otherwise = myLookup str assocs
```

Aufgabe

Implementieren Sie eine Funktion

```
1 buildUser :: Profile -> Maybe User
```

die aus einem Profil einen User erstellt wenn alle nötigen Angaben vorhanden sind (Name, Email, Stadt). Verwenden Sie die Applicative Instanz von Maybe noch nicht sondern arbeiten Sie alle Fälle “von Hand” durch.

Applicative Functor

Beispiel

```haskell
buildUser :: Profile -> Maybe User
buildUser prof = case myLookup "name" prof of
   Nothing -> Nothing
   Just name -> case myLookup "email" prof of
      Nothing -> Nothing
      Just email -> case myLookup "city" prof of
         Nothing -> Nothing
         Just city -> Just $ User name email
            city
```

Aufgabe

Versuchen Sie Ihre Funktion mithilfe von `(<$>)` und `(<*>)` zu deklarieren.

Applicative Functor

Beispiel

```
1 buildUser profile = User
2     <$> myLookup "name" profile
3     <*> myLookup "email" profile
4     <*> myLookup "city" profile
```

Applicative Regeln

Die Applicative Klasse hat folgende Regeln:

- Identität

```haskell
pure id <*> vv = vv
```

- Komposition

```haskell
pure (.) <*> f <*> g <*> x = f <*> (g <*> x)
```

- Homomorphismus

```haskell
pure f <*> pure v = pure (f v)
```

- Interchange

```haskell
f <*> pure x = pure ($ x) <*> f
```

Wir wollen unser Beispiel zum "Bauen" von `User` aus einem Profil wie folgt erweitern:

- Die Stadt eines Benutzers ist nicht mehr direkt in seinem Profil gespeichert. Die Städte werden den Email Adressen in einer separaten Liste zugeordnet.
- Daraus folgt, dass zum Herausfinden des `uCity` Feldes die `uEmail` Daten verwendet und daher bereits vorhanden sein müssen.
- Die Daten können also nicht unabhängig voneinander hergestellt werden, wir können das Problem also nicht "nur" mit einem Applicative lösen.

```haskell
type CityBase = [(String, String)]

annasProfile :: Profile
annasProfile =
    [ ("name", "Anna")
    , ("email", "anna@nasa.gov")
    ]

citiesB :: CityBase
citiesB =
    [ ("anna@nasa.gov", "Washington")
    , ("peter@peter.com", "Zueri")
    ]
```

Von “Hand”

```haskell
buildUserC :: Profile -> CityBase -> Maybe User
buildUserC p cities = case myLookup "name" p of
    Nothing -> Nothing
    Just name -> case myLookup "email" p of
        Nothing -> Nothing
        Just email -> case myLookup email cities
            of
            Nothing -> Nothing
            Just city -> Just $
                User name email city
```

Weil wir die `email` Daten brauchen um auf die `city` Daten zugreifen zu können, benötigen wir eine Funktion, die folgende Signatur hat:

```
bind :: Maybe String -> (String -> Maybe
    String) -> Maybe String
```

Weil wir damit Funktionen vom Typ

```
String -> Maybe String
```

"hintereinander" ausführen (komponieren) können.

Die Monad Klasse bietet uns genau das passende Interface:

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

```haskell
buildUserB :: Profile -> CityBase -> Maybe User
buildUserB profile cities =
    myLookup "name" profile
        >>= \n -> myLookup "email" profile
        >>= \e -> myLookup e cities
        >>= \c -> pure $ User n e c
```

Mit der “do Notation”:

```haskell
buildUserCM :: Profile -> CityBase -> Maybe User
buildUserCM profile cities = do
    name  <- myLookup "name" profile
    email <- myLookup "email" profile
    city  <- myLookup email cities
    pure $ User name email city
```

Die Monad Klasse hat auch Regeln.

- "left identity"

```
1 pure a >>= f = f a
```

- "Right identity"

```
1 m >>= pure = m
```

- Assoziativität:

```
1 (m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

Bemerkung

Die Funktion `pure` wird im Zusammenhang mit Monaden auch `return` genannt.