Funktionale Programmierung

Monaden

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 14.01.2019 21:21

Inhaltsverzeichnis

Eine Monade

- Typklasse

```haskell
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b -- heisst bind
  (>>) :: m a -> m b -> m b          -- heisst then
  return :: a -> m a                -- analog zu pure
  fail :: String -> m a
```

Vergleich mit Funktoren:

- wesentliche Funktionalität

```haskell
fmap  :: Functor f     =>   (a -> b) -> f a        -> f b
(<*>) :: Applicative f => f (a -> b) -> f a        -> f b
(>>=) :: Monad m       => m a        -> (a -> m b) -> m b
```

- Beispiel

```haskell
Prelude> fmap (+1) [1..3]
[2,3,4]
Prelude> [1..3] >>= return . (+1)
[2,3,4]
```

`IO` ist eine Monade

- in Haskell ist `IO` eine Monade
- über diesen *Trick* ist I/O auch **pure**
- `IO` ist ein abstrakter Datentyp
  - der *zur Laufzeit* Actions ausführen kann
- Beispiel:

```haskell
Prelude> :t [putStr "Hallo", putStr "Welt"]
[putStr "Hallo", putStr " Welt"] :: [IO ()]
```

`do`-Notation

- Beispiel mit IO

```haskell
f = getLine >>= putStrLn
```

- mit einem λ

```
f = getLine >>= \s -> putStrLn s
```

- anders angeordnet

```
f = getLine >>= \s ->
    putStrLn s
```

- und jetzt mit *syntactic sugar*:

```
f = do
  s <- getLine
  putStrLn s
```

Keine veränderliche Variablen!

- Folgendes sieht aus als ob `s` ein neuer Wert zugewiesen wird

```
f = do
  s <- getLine
  putStrLn s
  s <- getLine
  putStrLn s
```

- aber in Wirklichkeit wird `s` durch eine **neue** gebundene Variable mit selben Namen verdeckt

```
f = getLine >>= (\s ->
    putStrLn s >>
    getLine >>= (\s ->
    putStrLn s))
```

(Klammern nicht notwendig, nur zur Veranschaulichung)

Auch `Maybe` ist eine Monade

```
instance Monad Maybe where
  return = Just
  fail _ = Nothing
  Nothing >>= _ = Nothing
  Just x >>= f = f x
```

Beispiel für die Maybe-Monade

- `Data.Map` definiert `lookup`:

```
lookup :: Ord k => k -> Map k a -> Maybe a
```

- wir haben folgende `Maps`:

```
students :: Map MtkNr Student
exams    :: Map Ancode Exam
rooms    :: Map (TimeSlot, StudentName) Room
```

- wir wollen für eine Matrikelnummer und eine Prüfung herausfinden in welchem Raum die Prüfung statt findet

Ohne Monade

```
roomForStud :: MtkNr -> Ancode -> Maybe RoomName
roomForStud mtknr ancode =
  case lookup mtknr students of
    Nothing -> Nothing
    Just student -> case lookup ancode exams of
      Nothing -> Nothing
      Just exam ->
        let studentName' = studentName student
            timeSlot' = timeSlot exam
        in case lookup (timeSlot', studentName')
                       rooms of
            Nothing -> Nothing
            Just room -> Just $ roomName room
```

Mit Hilfe der `Maybe`-Monade

```
roomForStud :: MtkNr -> Ancode -> Maybe RoomName
roomForStud mtknr ancode = do
  student <- lookup mtknr students
  exam <- lookup ancode exams
  let studentName' = studentName student
      timeSlot' = timeSlot exam
  room <- lookup (timeSlot', studentName') rooms
  return $ roomName room
```

Monaden Gesetze

- Identität

```
m >>= return === m
return x >>= f === f x
```

- Assoziativität

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

Viele Libs sind als Monaden realisiert

- z.B. die HSpec-Tests
- z.B. Scotty (ein Webserver)

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty
import Data.Monoid (mconcat)
main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"
    html
    (mconcat
      [ "<h1>Scotty, "
      , beam
      , " me up!</h1>"])
```

Viele auch als Applikative Funktoren

- z.B. `optparse-applicative` zum Parsen von Kommmandozeilenargumenten

```
config :: Parser Config
config = Config <$> optional
    (strOption
       (long "output" <>
        short 'o' <>
        metavar "OUTFILE" <>
        help "output to file instead of stdout")) <*>
  switch (short 'v' <>
          long "verbose" <>
          help "turn on verbosity")
```

- oder `aeson` für JSON

```
instance FromJSON Payload where
  parseJSON (Object v) =
    Payload <$> v .: "from"
            <*> v .: "to"
            <*> v .: "subject"
            <*> v .: "body"
```

```
                <*> v .: "offset_seconds"
      parseJSON v = typeMismatch "Payload" v
```

Ein paar nützliche Monaden

Reader

- die Reader-Monade ermöglicht auf einen Wert lesend zuzugreifen
  - d.h. Funktionen zu nutzen, die diesen Wert als Argument geliefert bekommen, ohne ihn anzugeben
- nützlich zum Beispiel um das Programm mit einer Konfiguration zu starten
  - in der Readermonade wird die Konfiguration implizit immer weiter gereicht
  - wenn ein Wert aus der Konfiguration benötigt wird, kann einfach darauf zugegriffen werden
- die Idee basiert darauf, dass Funktionen eine Monade sind

Reader: Beispiel

```
newtype HumanName = HumanName String
newtype DogName = DogName String
newtype Address = Address String
data Person = Person
  { humanName :: HumanName
  , dogName :: DogName
  , address :: Address
  }
data Dog = Dog
  { dogsName :: DogName
  , dogsAddress :: Address
  }
```

- normaler Zugriff:

```
getDog :: Person -> Dog
getDog p = Dog (dogName p) (address p)
```

- über den applikativen Reader-Funktor

```
getDogR :: Person -> Dog
getDogR = Dog <$> dogName <*> address
```

- über die Reader-Monade

```haskell
getDogM :: Person -> Dog
getDogM = do
  dName <- dogName
  dAddress <- address
  return $ Dog dName dAddress
```

Writer

- in der Writer-Monade können Werte geloggt werden
- der Typ für das Log muss in der Typklasse `Monoid` sein
- Beispiel **ohne Monade**: Größter gemeinsamer Teiler

```haskell
gcd' :: Int -> Int -> Int
gcd' m n
  | m == n = m
  | m > n = gcd' (m - n) n
  | otherwise = gcd' m (n - m)
```

Beispiel: Größter gemeinsamer Teiler mit Log

```haskell
gcd'' :: Int -> Int -> Writer [String] Int
gcd'' m n
  | m == n = return m
  | m > n = do
    tell ["Calling gcd'' " ++
          show (m - n) ++ " " ++ show n]
    gcd'' (m - n) n
  | otherwise = do
    tell ["Calling gcd'' " ++
          show m ++ " " ++ show (n - m)]
    gcd'' m (n - m)
```

```
> runWriter $ gcd'' 22 18
(2, ["Calling gcd'' 4 18"
    ,"Calling gcd'' 4 14"
    ,"Calling gcd'' 4 10"
    ,"Calling gcd'' 4 6"
    ,"Calling gcd'' 4 2"
    ,"Calling gcd'' 2 2"])
```

```
> fst $ runWriter $ gcd'' 22 18
2
```

```
> mapM_ putStrLn $ snd $ runWriter $ gcd'' 22 18
Calling gcd'' 4 18
Calling gcd'' 4 14
Calling gcd'' 4 10
Calling gcd'' 4 6
Calling gcd'' 4 2
Calling gcd'' 2 2
```

State-Monade

- Beispiel **ohne Monade**: Stack

```haskell
type Stack = [Int]

pop' :: Stack -> (Int, Stack)
pop' (x:xs) = (x, xs)

push' :: Int -> Stack -> ((), Stack)
push' a xs = ((), a : xs)

stackManip :: Stack -> (Int, Stack)
stackManip stack =
  let ((), newStack1) = push' 3 stack
      (a, newStack2) = pop' newStack1
  in pop' newStack2
```

Beispiel: Stack mit der State-Monade

```haskell
pop :: State Stack Int
pop = state $ \(x:xs) -> (x, xs)

push :: Int -> State Stack ()
push a = state $ \xs -> ((), a : xs)

stackManip' :: State Stack Int
stackManip' = do
  push 3
  pop
  pop
```

```
> runState stackManip' [5,8,2,1]
(5,[8,2,1])
```