Intermediate Functional Programming in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

Wir hatten in der letzten Vorlesung die State-Monade kurz angesprochen.
Heute wenden wir uns der Definition zu und werden herausfinden, wie man noch weiter abstrahieren kann.

Beispiel:

```
countme :: a -> State Int a
countme a = do
              modify (+1)
              return a

example :: State Int Int
example = do
              x <- countme (2+2)
              y <- return (x*x)
              z <- countme (y-2)
              return z

examplemain = runState example 0
-- -> (14,2), 14 = wert von z, 2 = interner counter
```

Beispiel 2:

```
module Main where
import Control.Monad.State
type CountValue = Int
type CountState = (Bool, Int)

startState :: CountState
startState = (False, 0)

play :: String -> State CountState CountValue
--play  ...
```

```haskell
play []     = do
              (_, score) <- get
              return score
play (x:xs) = do
 (on, score) <- get
 case x of
   'C' -> if on then put (on, score + 1) else put (on, score)
   'A' -> if on then put (on, score - 1) else put (on, score)
   'T' -> put (False, score)
   'G' -> put (True, score)
   _   -> put (on, score)
 playGame xs

main = print $ runState (play "GACAACTCGAAT") startState
-- -> (-3,(False,-3))
```

Die State-Monade „packt“ einen State in jeden Funktionsaufruf:

```
newtype State s a = State { runState :: s -> (a,s) }

foo :: a -> State s b
foo :: a -> (s -> (b,s))
foo :: a -> s -> (b,s)
```

Die State-Monade „packt“ einen State in jeden Funktionsaufruf:

```
newtype State s a = State { runState :: s -> (a,s) }
```

```
foo :: a -> State s b
foo :: a -> (s -> (b,s))
foo :: a -> s -> (b,s)
```

Wir sehen, dass eine Funktion, die in die State-Monade aufgewertet wurde einfach nur ein weiteres Funktionsargument (den State `s`) mitgegeben wird und wir statt dem Ergebnis `b` ein `(b,s)` bekommen, was den neuen Zustand enthält.

Man kann sich die State-Monade als Berechnung vorstellen, die noch nicht ausgeführt werden kann, weil der initiale State nicht gesetzt ist. Dies spiegelt sich auch in der Funktor-Instanz wieder:

Man kann sich die State-Monade als Berechnung vorstellen, die noch nicht ausgeführt werden kann, weil der initiale State nicht gesetzt ist. Dies spiegelt sich auch in der Funktor-Instanz wieder:

```
instance Functor (State s) where
  fmap f (State rs) = State $ \s ->
                      let (a,s') = rs s in (f a, s')
```

Man kann sich die State-Monade als Berechnung vorstellen, die noch nicht ausgeführt werden kann, weil der initiale State nicht gesetzt ist. Dies spiegelt sich auch in der Funktor-Instanz wieder:

```
instance Functor (State s) where
  fmap f (State rs) = State $ \s ->
                      let (a,s') = rs s in (f a, s')
```

Wir sehen, dass wir erst mit `rs s` den State, den wir bekommen „ausführen“ müssen um ein a zu generieren, auf das wir die Funktion anwenden können.
Anschließend verpacken wir in unserem Ergebnis den modifizierten State und die angewendete Funktion.

Man kann sich die State-Monade als Berechnung vorstellen, die noch nicht ausgeführt werden kann, weil der initiale State nicht gesetzt ist. Dies spiegelt sich auch in der Funktor-Instanz wieder:

```
instance Functor (State s) where
  fmap f (State rs) = State $ \s ->
                      let (a,s') = rs s in (f a, s')
```

Wir sehen, dass wir erst mit `rs s` den State, den wir bekommen „ausführen“ müssen um ein a zu generieren, auf das wir die Funktion anwenden können.

Anschließend verpacken wir in unserem Ergebnis den modifizierten State und die angewendete Funktion.
Wichtig ist hier, dass wir wieder eine Funktion in `State` verpackt zurückgeben müssen, die einen State nimmt:

```
newtype State s a = State { runState :: s -> (a,s) }
```

Ganz analog funktioniert die Applicative-Instanz:

```
instance Applicative (State s) where
    pure a = State $ \s -> (a,s)
    (State rs) <*> (State rest) =
                  State $ \s ->
                        let (f,s') = rs s
                            (a,s'') = rest s'
                        in
                            (f a,s'')
```

Ganz analog funktioniert die Applicative-Instanz:

```
instance Applicative (State s) where
    pure a = State $ \s -> (a,s)
    (State rs) <*> (State rest) =
                  State $ \s ->
                        let (f,s') = rs s
                            (a,s'') = rest s'
                        in
                            (f a,s'')
```

Hier müssen wir den State 2x ausführen. Einmal um an das f zu kommen und dann verketten wir dies mit der restlichen State-Berechnung um auch noch an unser a zu kommen. Zurück geben wir den doppelt bearbeiteten State und den bearbeiteten Wert.

Ganz analog funktioniert die Applicative-Instanz:

```
instance Applicative (State s) where
    pure a = State $ \s -> (a,s)
    (State rs) <*> (State rest) =
                State $ \s ->
                      let (f,s') = rs s
                          (a,s'') = rest s'
                      in
                          (f a,s'')
```

Hier müssen wir den State 2x ausführen. Einmal um an das f zu kommen und dann verketten wir dies mit der restlichen State-Berechnung um auch noch an unser a zu kommen. Zurück geben wir den doppelt bearbeiteten State und den bearbeiteten Wert.
Wichtig ist hier die Reihenfolge! Wir hätten es auch umdrehen können:

```
let (f,s'') = rs s'
    (a,s') = rest s
in
    (f a,s'')
```

allerdings arbeitet <*> immer von links nach rechts!

Am wichtigsten ist die Monad-Instanz:

```
instance Monad (State s) where
    return a = State $ \s -> (a,s)
    (State rs) >>= f =
                 State $ \s ->
                       let (a,s') = rs s
                           (State rs') = f a
                       in
                           rs' s'
```

Am wichtigsten ist die Monad-Instanz:

```
instance Monad (State s) where
    return a = State $ \s -> (a,s)
    (State rs) >>= f =
                  State $ \s ->
                        let (a,s') = rs s
                            (State rs') = f a
                        in
                            rs' s'
```

Wir müssen wieder zuerst den State ausführen um an unser a zu gelangen. Danach können wir unsere Funktion f ausführen um eine neue Funktion zu bekommen, die wir auch aus dem State auspacken. Eine kleine Anwendung des erhaltenen States hierauf gibt uns schlussendlich unser Ergebnis.

Wir hatten letzte Woche die Maybe-Monade mit dem folgenden Anwendugsfall:

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

Nun möchten wir aus irgendeinem Grund (Logging, Netzwerk, ..) zwischen dem `getInbox` und dem `getFirstMail` eine IO-Aktion ausführen.

Problem: `IO /= Maybe`

Wir hatten letzte Woche die Maybe-Monade mit dem folgenden Anwendugsfall:

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

Nun möchten wir aus irgendeinem Grund (Logging, Netzwerk, ..) zwischen dem `getInbox` und dem `getFirstMail` eine IO-Aktion ausführen.

Problem: `IO /= Maybe`
Als Konsequenz können wir die do-notation nicht verwenden - wir fallen also wieder zurück auf die hässliche Notation:

```haskell
f :: IO (Maybe Header)
f = case getInbox of
      (Just folder) ->
          do
            putStrLn "debug"
            return $ case getFirstMail folder of
              (Just mail) ->
                  case getHeader mail of
                    (Just head) -> return head
                    Nothing     -> Nothing
              Nothing     -> Nothing
      Nothing         -> return Nothing
```

Dieser Code ist ohne Frage hässlich. Stellt sich die Frage, ob wir nicht soetwas, wie `MaybeIO` bauen können, sodass wir wieder do-notation verwenden können.

Dieser Code ist ohne Frage hässlich. Stellt sich die Frage, ob wir nicht soetwas, wie `MaybeIO` bauen können, sodass wir wieder do-notation verwenden können.
Also kombinieren wir es (ähnlich zur State-Monade):

```
data MaybeIO a = MaybeIO { runMaybeIO :: IO (Maybe a) }
```

Dieser Code ist ohne Frage hässlich. Stellt sich die Frage, ob wir nicht soetwas, wie `MaybeIO` bauen können, sodass wir wieder do-notation verwenden können.
Also kombinieren wir es (ähnlich zur State-Monade):

```
data MaybeIO a = MaybeIO { runMaybeIO :: IO (Maybe a) }
```

Dieses liefert uns 2 Funktionen:

```
MaybeIO    :: IO (Maybe a) -> MaybeIO a
runMaybeIO :: MaybeIO a -> IO (Maybe a)
```

Also eine Funktion, um in unsere neue Monade zu kommen und eine Funktion um dieses wieder Rückgängig zu machen.

Fangen wir mit der Functor-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = wrapped
                 where
                   unwrapped = runMaybeIO input
                   -- IO (Maybe a) auspacken
                   fmapped = fmap (fmap f) unwrapped
                   -- erstes fmap mapped durch IO,
                   -- zweites fmap durch Maybe
                   wrapped = MaybeIO fmapped
                   -- einpacken in den richtigen Typen
```

Fangen wir mit der Functor-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = wrapped
                 where
                   unwrapped = runMaybeIO input
                   -- IO (Maybe a) auspacken
                   fmapped = fmap (fmap f) unwrapped
                   -- erstes fmap mapped durch IO,
                   -- zweites fmap durch Maybe
                   wrapped = MaybeIO fmapped
                   -- einpacken in den richtigen Typen
```

oder kurz:

```
instance Functor MaybeIO where
  fmap f = MaybeIO . fmap (fmap f) . runMaybeIO
```

Applicative:

```
instance Applicative MaybeIO where
  pure     = MaybeIO . return . Just
             -- in Just packen, mit return in IO heben
             -- und den Typen mit MaybeIO aliasen
  f <*> x = MaybeIO $ (<*>) <$> f' <*> x'
             where
               f' = runMaybeIO f -- IO (Maybe f)
               x' = runMaybeIO x -- IO (Maybe x)
```

Das erste (<*>) ist Applicative auf Maybe und es wird in Applicative <*> von IO hineingemappt.

Monad:

```
instance Monad MaybeIO where
  return = pure
  x >>= f = MaybeIO $ x''
            where
              x' = runMaybeIO x
              x'' = x' >>= runMaybeIO . mb . fmap f
              mb :: Maybe (MaybeIO a) -> MaybeIO a
              mb (Just a) = a
              mb Nothing = MaybeIO $ return Nothing
```

Zuerst packen wir das MaybeIO aus. `fmap f` bringt uns ein Maybe (MaybeIO a), welches wir mittels der Hilfsfunktion `mb` auspacken oder einen leeren Wert konstruieren.
Dieses jagen wir noch durch runMaybeIO um wieder ein `IO (Maybe a)` zu bekommen, auf das wir dann den >==-Operator von IO anwenden können. Das Ergebnis verpacken wir noch in `MaybeIO` und sind fertig.

Da wir nun eine Monade definiert haben, können wir ja wieder do nutzen:

```
f = do
      i <- getInbox
      putStrLn "debug"
      m <- getFirstMail i
      h <- getHeader m
      return h
```

Allerdings:

```
Couldn't match type \T1\textquoteleft Maybe\T1\textquoterigh
Expected type: MaybeIO Inbox
  Actual type: Maybe Inbox
In a stmt of a 'do' block: in <- getInbox

Couldn't match type \T1\textquoteleft IO\T1\textquoteright
Expected type: MaybeIO ()
  Actual type: IO ()
In a stmt of a 'do' block: putStrLn "debug"

Couldn't match type \T1\textquoteleft Maybe\T1\textquoterigh
Expected type: MaybeIO Mail
  Actual type: Maybe Mail
In a stmt of a 'do' block: m <- getFirstMail i

Couldn't match type \T1\textquoteleft Maybe\T1\textquoterigh
Expected type: MaybeIO Header
  Actual type: Maybe Header
In a stmt of a 'do' block: h <- getHeader m
```

Wir brauchen also Konverter:

- `Maybe -> MaybeIO`
- `IO -> MaybeIO`

Wir brauchen also Konverter:

- `Maybe -> MaybeIO`
- `IO -> MaybeIO`

Aber wir haben schon alles, was wir brauchen, wenn wir uns nur klar machen:

```
return  :: Maybe a -> IO (Maybe a) -- return von IO
MaybeIO :: IO (Maybe a) -> MaybeIO a
```

Wir brauchen also Konverter:

- `Maybe -> MaybeIO`
- `IO -> MaybeIO`

Aber wir haben schon alles, was wir brauchen, wenn wir uns nur klar machen:

```
return  :: Maybe a -> IO (Maybe a) -- return von IO
MaybeIO :: IO (Maybe a) -> MaybeIO a
```

und

```
Just       :: a -> Maybe a
fmap Just :: IO a -> IO (Maybe a)
```

Somit wird unser Code von oben:

```
f = do
      i <- MaybeIO (return (getInbox))
      MaybeIO (fmap Just (putStrLn "debug"))
      m <- MaybeIO (return (getFirstMail i))
      h <- MaybeIO (return (getHeader m))
      return h
```

Somit wird unser Code von oben:

```
f = do
      i <- MaybeIO (return (getInbox))
      MaybeIO (fmap Just (putStrLn "debug"))
      m <- MaybeIO (return (getFirstMail i))
      h <- MaybeIO (return (getHeader m))
      return h
```

Zwar können wir nun do nutzen, aber das sieht doch eher hässlich aus. Außerdem ist so viel Code doppelt!

Wenn wir Muster finden, dann faktorisieren wir sie doch raus!

```
liftMaybe :: Maybe a -> MaybeIO a
liftMaybe x = MaybeIO (return x)

liftIO :: IO a -> MaybeIO a
liftIO x = MaybeIO (fmap Just x)
```

Wenn wir Muster finden, dann faktorisieren wir sie doch raus!

```
liftMaybe :: Maybe a -> MaybeIO a
liftMaybe x = MaybeIO (return x)

liftIO :: IO a -> MaybeIO a
liftIO x = MaybeIO (fmap Just x)
```

und wir erhalten:

```
f = do
      i <- liftMaybe getInbox
      liftIO $ putStrLn "debug"
      m <- liftMaybe $ getFirstMail i
      h <- liftMaybe $ getHeader m
      return h
```