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  ...
```

```
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)
    f >>= (State rs) =
                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)
    f >>= (State rs) =
                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.