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:

```haskell
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:

```haskell
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 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 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:

```
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 . pure . Just
            -- in Just packen, mit pure 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 Maybe with MaybeIO
Expected type: MaybeIO Inbox
  Actual type: Maybe Inbox
In a stmt of a 'do' block: in <- getInbox

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

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

Couldn't match type Maybe with MaybeIO
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
```

Wenn wir uns nochmals ansehen, welche Eigenschaft der IO-Monade wir genutzt haben, dann fällt uns auf:

Wenn wir uns nochmals ansehen, welche Eigenschaft der IO-Monade wir genutzt haben, dann fällt uns auf:

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

`fmap` von IO als Funktor

Wenn wir uns nochmals ansehen, welche Eigenschaft der IO-Monade wir genutzt haben, dann fällt uns auf:

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

`fmap` von IO als Funktor

```
instance Applicative MaybeIO where
  pure    = MaybeIO . pure . Just
  f <*> x = MaybeIO $ (<*>) <$> (runMaybeIO f)
                            <*> (runMaybeIO x)
```

`pure` und `<*>` von IO als Applicative

Wenn wir uns nochmals ansehen, welche Eigenschaft der IO-Monade wir genutzt haben, dann fällt uns auf:

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

fmap von IO als Funktor

```
instance Applicative MaybeIO where
  pure    = MaybeIO . pure . Just
  f <*> x = MaybeIO $ (<*>) <$> (runMaybeIO f)
                            <*> (runMaybeIO x)
```

pure und <*> von IO als Applicative

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

return und »= von IO

Uns fällt auf: Wir verwenden gar keine intrisischen Eigenschaften von IO.
Also können wir IO auch durch jede Monade ersetzten. Dies nennt man dann Monad Transformer.

```
data MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

Und der Code von eben wird zu:

```
instance Functor m => Functor (MaybeT m) where
  fmap f = MaybeT . fmap (fmap f) . runMaybeT

instance Applicative m => Applicative (MaybeT m) where
  pure    = MaybeT . pure . Just
  f <*> x = MaybeT $ (<*>) <$> (runMaybeT f)
                           <*> (runMaybeT x)

instance Monad m => Monad (MaybeT m) where
  return = pure
  x >>= f = MaybeT $ (runMaybeT x)
                     >>= runMaybeT . mb . fmap f
            where
              mb (Just a) = a
              mb Nothing = MaybeT $ return Nothing
```

Und der Code von eben wird zu:

```
instance Functor m => Functor (MaybeT m) where
  fmap f = MaybeT . fmap (fmap f) . runMaybeT

instance Applicative m => Applicative (MaybeT m) where
  pure    = MaybeT . pure . Just
  f <*> x = MaybeT $ (<*>) <$> (runMaybeT f)
                           <*> (runMaybeT x)

instance Monad m => Monad (MaybeT m) where
  return = pure
  x >>= f = MaybeT $ (runMaybeT x)
                     >>= runMaybeT . mb . fmap f
            where
              mb (Just a) = a
              mb Nothing = MaybeT $ return Nothing
```

Wir haben nur die Monade heraus faktorisiert.

Frage: Wie realisieren wir nun `liftIO` etc.?

Frage: Wie realisieren wir nun `liftIO` etc.?
Über Typklassen!

```
class Monad m => MonadIO m where
    liftIO :: IO a -> m a
```

Wir verlangen einfach, dass IO irgendwie verarbeitet werden muss.

Frage: Wie realisieren wir nun `liftIO` etc.?
Über Typklassen!

```
class Monad m => MonadIO m where
    liftIO :: IO a -> m a
```

Wir verlangen einfach, dass IO irgendwie verarbeitet werden muss.

IO ist ein Spezialfall, da IO als Monade nicht additiv ist. Es gibt somit keinen IOT.

Frage: Wie realisieren wir nun `liftIO` etc.?
Über Typklassen!

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

Frage: Wie realisieren wir nun `liftIO` etc.?
Über Typklassen!

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

Dies ist die allgemeine Form für additive Monaden. Mit `lift` heben wir uns eine monadische Ebene höher.
Doch was bedeutet das?

Wir haben schon ein paar Monaden kennengelernt. Diese sind fast alle additiv. Wir können somit folgendes bauen:

Wir haben schon ein paar Monaden kennengelernt. Diese sind fast alle additiv. Wir können somit folgendes bauen:

```
data MyMonadStack a = StateT (EitherT (MaybeT (IO a)))
```

Wir haben schon ein paar Monaden kennengelernt. Diese sind fast alle additiv. Wir können somit folgendes bauen:

```
data MyMonadStack a = StateT (EitherT (MaybeT (IO a)))
```

Wie schreiben wir nun hierin Code?

```
bsp :: MyMonadStack ()
bsp = do
    a <- fun                    -- fun  :: MyMonadStack Int
    b <- lift $ fun2            -- fun2 :: EitherT (MaybeT (IO Int))
    c <- lift . lift $ fun3 -- fun3 :: MaybeT (IO Int))
    liftIO $ putStrLn "foo"
```

Um auf spezielle Ebenen im Monad-Stack zuzugreifen gibt es für jeden Zweck eine Typklasse. Beispielsweise:

Um auf spezielle Ebenen im Monad-Stack zuzugreifen gibt es für jeden Zweck eine Typklasse. Beispielsweise:

```
instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

Um auf spezielle Ebenen im Monad-Stack zuzugreifen gibt es für jeden Zweck eine Typklasse. Beispielsweise:

```
instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

Nun können wir einfach

```
bsp :: MyMonadStack ()
bsp = do
    state <- get
    put $ manipulateState state
    liftIO $ putStrLn "foo"
```

Um auf spezielle Ebenen im Monad-Stack zuzugreifen gibt es für jeden Zweck eine Typklasse. Beispielsweise:

```
instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

Nun können wir einfach

```
bsp :: MyMonadStack ()
bsp = do
    state <- get
    put $ manipulateState state
    liftIO $ putStrLn "foo"
```

Analog gibt es dies auch für `ReaderT` (`env`), welches ein read-only-Environment bereitstellt (z.b. eine Konfiguration) oder für `WriterT` (`tell`), welches ein write-only-Environment zur Verfügung stellt (z.b. Logging). Häufig findet man daher einen Read-Write-State-Transformer, kurz `RWST`-Stack.

Um auf spezielle Ebenen im Monad-Stack zuzugreifen gibt es für jeden Zweck eine Typklasse. Beispielsweise:

```
instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

Nun können wir einfach

```
bsp :: MyMonadStack ()
bsp = do
    state <- get
    put $ manipulateState state
    liftIO $ putStrLn "foo"
```

Analog gibt es dies auch für `ReaderT` (`env`), welches ein read-only-Environment bereitstellt (z.b. eine Konfiguration) oder für `WriterT` (`tell`), welches ein write-only-Environment zur Verfügung stellt (z.b. Logging). Häufig findet man daher einen Read-Write-State-Transformer, kurz `RWST`-Stack. Echtweltprogramme sind meist durch eine `RWST IO` mit der Außenwelt verbunden.

Ein weiteres Echtwelt-Beispiel könnte etwa der folgende Aufruf sein:

```
data Env = Env { filename :: String }

readInputs :: ReaderT Env IO String
readInputs = do
             e <- env
             f <- liftIO $ readFile (filename e)
             return f
```

Ein weiteres Echtwelt-Beispiel könnte etwa der folgende Aufruf sein:

```
data Env = Env { filename :: String }

readInputs :: ReaderT Env IO String
readInputs = do
             e <- env
             f <- liftIO $ readFile (filename e)
             return f
```

Dieser Aufruf liest einen Dateinamen aus einem Environment, kann per `liftIO` IO-Aktionen ausführen und das Ergebnis (den String mit dem Dateiinhalt) zurückliefern.

Noch ein Beispiel aus einem Spiel könnte sein:

```
mainLoop :: RWST Env () State IO ()
mainLoop = do
          e <- env
          f <- liftIO $ getUserInput (keySettings e)
          newWorld <- updateWorld f
                      --holt sich die Welt mittels 'get'
          put newWorld
          unless (f == endKey e) mainLoop
```

Noch ein Beispiel aus einem Spiel könnte sein:

```
mainLoop :: RWST Env () State IO ()
mainLoop = do
          e <- env
          f <- liftIO $ getUserInput (keySettings e)
          newWorld <- updateWorld f
                      --holt sich die Welt mittels 'get'
          put newWorld
          unless (f == endKey e) mainLoop
```

Dies ist ein klassisches Game-Loop, bestehend aus Konfigurationen im Env (Key settings), IO (User-Input abfragen), Update des internen Zustands (updateWorld) und das schreiben des neuen Zustandes (put newWorld). Sofern dann das Spiel nicht beendet ist loopen wir.