Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 10: *Monad Transformers*
Stand: 17. Juni 2016

Wiederholung: State

Definition

State war defeniert als:

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

Definition

State war defeniert als:

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

Eine State-Berechnung produziert uns somit ein `a` mit Hilfe des versteckten Parameters `s`.

Definition

State war defeniert als:

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

Eine State-Berechnung produziert uns somit ein a mit Hilfe des versteckten Parameters s.
Wir erhalten also:

```
State    :: (s -> (a,s)) -> State s a
runState :: State s a    ->  s -> (a,s)
```

Definition

State war defeniert als:

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

Eine State-Berechnung produziert uns somit ein a mit Hilfe des versteckten Parameters s.
Wir erhalten also:

```
State    :: (s -> (a,s)) -> State s a
runState :: State s a     ->  s -> (a,s)
```

Wenn wir State monadisch nutzen, benutzen wir Funktionen der folgenden Form:

Definition

State war defeniert als:

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

Eine State-Berechnung produziert uns somit ein a mit Hilfe des versteckten Parameters s.
Wir erhalten also:

```
State    :: (s -> (a,s)) -> State s a
runState :: State s a     ->  s -> (a,s)
```

Wenn wir State monadisch nutzen, benutzen wir Funktionen der folgenden Form:

```
foo :: a -> State s b
```

Definition

State war defeniert als:

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

Eine State-Berechnung produziert uns somit ein a mit Hilfe des versteckten Parameters s.
Wir erhalten also:

```
State    :: (s -> (a,s)) -> State s a
runState :: State s a     ->  s -> (a,s)
```

Wenn wir State monadisch nutzen, benutzen wir Funktionen der folgenden Form:

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

Definition

State war defeniert als:

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

Eine State-Berechnung produziert uns somit ein a mit Hilfe des versteckten Parameters s.
Wir erhalten also:

```
State    :: (s -> (a,s)) -> State s a
runState :: State s a    ->  s -> (a,s)
```

Wenn wir State monadisch nutzen, benutzen wir Funktionen der folgenden Form:

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

Definition

State war defeniert als:

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

Eine State-Berechnung produziert uns somit ein `a` mit Hilfe des versteckten Parameters `s`.
Wir erhalten also:

```
State    :: (s -> (a,s)) -> State s a
runState :: State s a    ->  s -> (a,s)
```

Wenn wir State monadisch nutzen, benutzen wir Funktionen der folgenden Form:

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

State fügt somit einen Funktionsparameter `s` hinzu, und erstellt den Rückgabewert `b` und einen aktualisierten Zustand.

Frage

Gegeben

```
type Position = V2 Int
type Boardsize = V2 Int
data GameState = GameState {pos :: Position, board :: Boardsize }
```

was macht folgender Code?

```
rechts :: State GameState Bool
rechts = do
       (GameState (V2 x y) (V2 sx sy)) <- get
       case x+1 > sx of
         True  -> return False
         False -> do
               put (GameState (V2 (x+1) y) (V2 sx sy))
               return True
```

Frage

Gegeben

```
type Position = V2 Int
type Boardsize = V2 Int
data GameState = GameState {pos :: Position, board :: Boardsize }
```

was macht folgender Code?

```
rechts :: State GameState Bool
rechts = do
       (GameState (V2 x y) (V2 sx sy)) <- get
       case x+1 > sx of
         True  -> return False
         False -> do
                put (GameState (V2 (x+1) y) (V2 sx sy))
                return True
```

Diese Funktion bewegt die Spielfigur nach rechts - aber prüft, dass sie nicht aus dem Feld hinauslaufen kann.

Frage

Da wir in der zwischenzeit schon `lens` besprochen haben, gibt es auch „State-aware“-Lenses. Hiermit können wir die Funktion `rechts` zusammenfassen auf:

```
rechts :: State GameState Bool
rechts = do
       x  <- use $ pos . _x
       sx <- use $ board . _x
       let valid = x+1 <= sx
       when valid $ pos . _x += 1
       return valid
```

Motivation

Ein Problem, was nun auftaucht ist, dass wir zwar z.B. ein

```
State (Either e a)
```

erstellen können, aber wir verlieren die ganzen monadischen Eigenschaften von `Either e a`, da wir das `(>>=)` von `State` benutzen.

Motivation

Ein Problem, was nun auftaucht ist, dass wir zwar z.B. ein

```
State (Either e a)
```

erstellen können, aber wir verlieren die ganzen monadischen Eigenschaften von `Either e a`, da wir das `(>>=)` von `State` benutzen.

Wie können wir das lösen? Kann man das irgendwie kombinieren?

Kombination von Monaden

Oder: Wieso nur eine Monade, wenn man alle haben kann?

Beispiel

Wir hatten in einer Übung ein einfaches Beispiel in der `Maybe`-Monade mit folgendem Code:

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

Beispiel

Wir hatten in einer Übung ein einfaches Beispiel in der `Maybe`-Monade mit folgendem Code:

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

Nun ändern wir das Szenario:

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

Beispiel

Wir hatten in einer Übung ein einfaches Beispiel in der `Maybe`-Monade mit folgendem Code:

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

Nun ändern wir das Szenario:

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

Problem: `IO /= Maybe`

Beispiel

Wir hatten in einer Übung ein einfaches Beispiel in der `Maybe`-Monade mit folgendem Code:

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

Nun ändern wir das Szenario:

Wir möchten 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.

Beispiel

Wir fallen also wieder zurück auf die hässliche Notation:

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

Beispiel

Dieser Code ist ohne Frage umständlich und unschön. Stellt sich die Frage, ob wir nicht so etwas wie `MaybeIO` bauen können, sodass wir wieder `do`-Notation verwenden können.

Beispiel

Dieser Code ist ohne Frage umständlich und unschön. Stellt sich die Frage, ob wir nicht so etwas wie `MaybeIO` bauen können, sodass wir wieder `do`-Notation verwenden können.

Also kombinieren wir es (ähnlich zur `State`-Monade) in einen neuen Typen:

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

Beispiel

Dieser Code ist ohne Frage umständlich und unschön. Stellt sich die Frage, ob wir nicht so etwas wie `MaybeIO` bauen können, sodass wir wieder `do`-Notation verwenden können.

Also kombinieren wir es (ähnlich zur `State`-Monade) in einen neuen Typen:

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

und bekommen so diese zwei Funktionen:

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

Beispiel

Dieser Code ist ohne Frage umständlich und unschön. Stellt sich die Frage, ob wir nicht so etwas wie `MaybeIO` bauen können, sodass wir wieder `do`-Notation verwenden können.

Also kombinieren wir es (ähnlich zur `State`-Monade) in einen neuen Typen:

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

und bekommen so diese zwei Funktionen:

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

Nun müsser wir „nur“ die Monaden-Instanz (inkl. Vorraussetzungen) schreiben, die das tut, was wir wollen.

Fangen wir mit der `Functor`-Instanz an:

Fangen wir mit der `Functor`-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = _
```

```
Found hole ‘_’ with type: MaybeIO b
Where: ‘b’ is a rigid type variable
Relevant bindings include
  input :: MaybeIO a
  f :: a -> b
  fmap :: (a -> b) -> MaybeIO a -> MaybeIO b
```

Fangen wir mit der `Functor`-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = _
                 where
                   unwrapped = runMaybeIO input
```

```
Found hole ‘_’ with type: MaybeIO b
Where: ‘b’ is a rigid type variable
Relevant bindings include
  unwrapped :: IO (Maybe a)
  input :: MaybeIO a
  f :: a -> b
  fmap :: (a -> b) -> MaybeIO a -> MaybeIO b
```

Fangen wir mit der `Functor`-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = _
                 where
                   unwrapped = runMaybeIO input
                   fmapped   = fmap (fmap f) unwrapped
```

```
Found hole ‘_’ with type: MaybeIO b
Where: ‘b’ is a rigid type variable
Relevant bindings include
  fmapped :: IO (Maybe b)
  unwrapped :: IO (Maybe a)
  input :: MaybeIO a
  f :: a -> b
  fmap :: (a -> b) -> MaybeIO a -> MaybeIO b
```

Fangen wir mit der `Functor`-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = _
                 where
                   unwrapped = runMaybeIO input
                   fmapped   = fmap (fmap f) unwrapped
                   wrapped   = MaybeIO fmapped
```

```
Found hole ‘_’ with type: MaybeIO b
Where: ‘b’ is a rigid type variable
Relevant bindings include
  wrapped :: MaybeIO b
  fmapped :: IO (Maybe b)
  unwrapped :: IO (Maybe a)
  input :: MaybeIO a
  f :: a -> b
  fmap :: (a -> b) -> MaybeIO a -> MaybeIO b
```

Fangen wir mit der `Functor`-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = wrapped
                 where
                   unwrapped = runMaybeIO input
                   fmapped   = fmap (fmap f) unwrapped
                   wrapped   = MaybeIO fmapped
```

Fangen wir mit der `Functor`-Instanz an:

```
instance Functor MaybeIO where
  fmap f input = wrapped
                 where
                   unwrapped = runMaybeIO input
                   fmapped   = fmap (fmap f) unwrapped
                   wrapped   = MaybeIO fmapped
```

oder kurz:

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

Functor / Applicative / Monad

`Applicative`:

Functor / Applicative / Monad

Applicative:

```
instance Applicative MaybeIO where
  pure a  = _
  f <*> x = undefined
```

```
Found hole ‘_’ with type: MaybeIO a
Where: ‘a’ is a rigid type variable
Relevant bindings include
  a :: a
  pure :: a -> MaybeIO a
```

Applicative:

```
instance Applicative MaybeIO where
  pure a  = MaybeIO $ _
  f <*> x = undefined
```

```
Found hole ‘_’ with type: IO (Maybe a)
Where: ‘a’ is a rigid type variable
Relevant bindings include
  a :: a
  pure :: a -> MaybeIO a
```

Functor / Applicative / Monad

Applicative:

```
instance Applicative MaybeIO where
  pure a  = MaybeIO $ pure $ _
  f <*> x = undefined
```

```
Found hole ‘_’ with type: Maybe a
Where: ‘a’ is a rigid type variable
Relevant bindings include
  a :: a
  pure :: a -> MaybeIO a
```

Functor / Applicative / Monad

Applicative:

```
instance Applicative MaybeIO where
  pure a  = MaybeIO $ pure $ pure $ _
  f <*> x = undefined
```

```
Found hole ‘_’ with type: a
Where: ‘a’ is a rigid type variable
Relevant bindings include
  a :: a
  pure :: a -> MaybeIO a
```

Applicative:

```
instance Applicative MaybeIO where
  pure a  = MaybeIO $ pure $ pure $ a
  f <*> x = undefined
```

Applicative:

```
instance Applicative MaybeIO where
  pure a  = MaybeIO . pure . pure $ a
  f <*> x = undefined
```

Functor / Applicative / Monad

Applicative:

```
instance Applicative MaybeIO where
  pure    = MaybeIO . pure . pure
  f <*> x = _
```

```
Found hole ‘_’ with type: MaybeIO b
Where: ‘b’ is a rigid type variable
Relevant bindings include
  x :: MaybeIO a
  f :: MaybeIO (a -> b)
  (<*>) :: MaybeIO (a -> b) -> MaybeIO a -> MaybeIO b
```

Functor / Applicative / Monad

Applicative:

```
instance Applicative MaybeIO where
  pure    = MaybeIO . pure . pure
  f <*> x = MaybeIO $ _
```

```
Found hole ‘_’ with type: IO (Maybe b)
Where: ‘b’ is a rigid type variable
Relevant bindings include
  x :: MaybeIO a
  f :: MaybeIO (a -> b)
  (<*>) :: MaybeIO (a -> b) -> MaybeIO a -> MaybeIO b
```

Applicative:

```
instance Applicative MaybeIO where
  pure    = MaybeIO . pure . pure
  f <*> x = MaybeIO $ _
            where
             f' = runMaybeIO f
             x' = runMaybeIO x
```

```
Found hole '_' with type: IO (Maybe b)
Where: 'b' is a rigid type variable
Relevant bindings include
  f' :: IO (Maybe (a -> b))
  x' :: IO (Maybe a)
  x :: MaybeIO a
  f :: MaybeIO (a -> b)
  (<*>) :: MaybeIO (a -> b) -> MaybeIO a -> MaybeIO b
```

Applicative:

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

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

Functor / Applicative / Monad

`Monad`:

Functor / Applicative / Monad

Monad:

```
instance Monad MaybeIO where
  return  = pure
  x >>= f = MaybeIO $ _
```

```
Found hole ‘_’ with type: IO (Maybe b)
Where: ‘b’ is a rigid type variable
Relevant bindings include
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Functor / Applicative / Monad

Monad:

```
instance Monad MaybeIO where
  return  = pure
  x >>= f = MaybeIO $ _
            where
              x'  = runMaybeIO x
```

```
Found hole '_' with type: IO (Maybe b)
Where: 'b' is a rigid type variable
Relevant bindings include
  x' :: IO (Maybe a)
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Functor / Applicative / Monad

Monad:

```
instance Monad MaybeIO where
  return  = pure
  x >>= f = MaybeIO $ x' >>= _
            where
              x'  = runMaybeIO x
```

```
Found hole '_' with type: Maybe a -> IO (Maybe b)
Where: 'a' is a rigid type variable
       'b' is a rigid type variable
Relevant bindings include
  x' :: IO (Maybe a)
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Functor / Applicative / Monad

Monad:

```
instance Monad MaybeIO where
  return  = pure
  x >>= f = MaybeIO $ x' >>= _ . fmap f
            where
              x'  = runMaybeIO x
```

```
Found hole '_' with type: Maybe (MaybeIO b) -> IO (Maybe b)
Where: 'b' is a rigid type variable
Relevant bindings include
  x' :: IO (Maybe a)
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Functor / Applicative / Monad

Monad:

```
instance Monad MaybeIO where
  return  = pure
  x >>= f = MaybeIO $ x' >>= runMaybeIO . _ . fmap f
            where
              x'  = runMaybeIO x
```

```
Found hole '_' with type: Maybe (MaybeIO b) -> MaybeIO b
Where: 'b' is a rigid type variable
Relevant bindings include
  x' :: IO (Maybe a)
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Monad:

```
instance Monad MaybeIO where
  return  = pure
  x >>= f = MaybeIO $ x' >>= runMaybeIO . mb . fmap f
            where
              x'  = runMaybeIO x
              mb :: Maybe (MaybeIO a) -> MaybeIO a
              mb a = _
```

```
Found hole '_' with type: MaybeIO a1
Where: 'a1' is a rigid type variable
Relevant bindings include
  a :: Maybe (MaybeIO a1)
  mb :: Maybe (MaybeIO a1) -> MaybeIO a1
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Monad:

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

```
Found hole '_' with type: MaybeIO a1
Where: 'a1' is a rigid type variable
Relevant bindings include
  a :: MaybeIO a1
  mb :: Maybe (MaybeIO a1) -> MaybeIO a1
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Monad:

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

```
Found hole '_' with type: MaybeIO a1
Where: 'a1' is a rigid type variable
Relevant bindings include
  mb :: Maybe (MaybeIO a1) -> MaybeIO a1
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Monad:

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

```
Found hole '_' with type: IO (Maybe a1)
Where: 'a1' is a rigid type variable
Relevant bindings include
  mb :: Maybe (MaybeIO a1) -> MaybeIO a1
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Monad:

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

```
Found hole '_' with type: Maybe a1
Where: 'a1' is a rigid type variable
Relevant bindings include
  mb :: Maybe (MaybeIO a1) -> MaybeIO a1
  f :: a -> MaybeIO b
  x :: MaybeIO a
  (>>=) :: MaybeIO a -> (a -> MaybeIO b) -> MaybeIO b
```

Monad:

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

Beispiel revisited

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
```

Beispiel revisited

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
```

Beispiel revisited

Wir brauchen also zwei Konverter:

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

Beispiel revisited

Wir brauchen also zwei Konverter:

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

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

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

Beispiel revisited

Wir brauchen also zwei Konverter:

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

Aber wir haben schon alles, was wir brauchen, wenn wir uns nur Folgendes 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)
```

Beispiel revisited

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
```

Beispiel revisited

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!

Finale Version

Wenn wir Muster finden, dann lagern wir sie doch einfach in Funktionen aus!

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

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

Finale Version

Wenn wir Muster finden, dann lagern wir sie doch einfach in Funktionen aus!

```
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
```

Recap

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

Recap

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 `Functor`

Recap

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 `Functor`

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

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

Recap

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 Functor

```
instance Applicative MaybeIO where
  pure    = MaybeIO . pure . pure
  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 als Monad

Recap

Uns fällt auf: Wir verwenden gar keine intrisischen Eigenschaften von `IO`.

Also können wir `IO` auch durch jede andere Monade ersetzten. Dies nennt man dann *Monad Transformer*.

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

Und der Code von eben...

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

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

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
```

...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 . pure
  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
```

Recap

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

Recap

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

Über Typklassen! Dafür sind sie schließlich da!

```
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! Dafür sind sie schließlich da!

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

Wir verlangen einfach, dass `IO` irgendwie verarbeitet werden muss.

Genereller:

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

Recap

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

Über Typklassen! Dafür sind sie schließlich da!

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

Wir verlangen einfach, dass `IO` irgendwie verarbeitet werden muss.

Genereller:

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

Dies ist die allgemeine Form für verknüpfbare Monaden (composable monads; monad-transformers). Mit `lift` heben wir uns eine monadische Ebene höher.

Recap

Können wir das nun für jede Kombination von Monaden machen?

Können wir das nun für jede Kombination von Monaden machen?
Nein.

Recap

Können wir das nun für jede Kombination von Monaden machen?
Nein.

Für Funktoren und Applicatives geht das. Für Monaden nicht.

Recap

Bei einem Funktor reicht folgedes aus:

```haskell
newtype Compose f g a = Compose (f (g a))

instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose a) = Compose $ fmap (fmap f) a
```

[1] https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Functor-Compose.html

Recap

Bei einem Funktor reicht folgedes aus:

```haskell
newtype Compose f g a = Compose (f (g a))

instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose a) = Compose $ fmap (fmap f) a
```

Analog bei einem Applicative:

```haskell
instance (Applicative f, Applicative g)
         => Applicative (Compose f g) where
  pure = Compose . pure . pure
  (Compose f) <*> (Compose x) = Compose $ (<*>) <$> f <*> x
```

[1] https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Functor-Compose.html

Bei einem Funktor reicht folgedes aus:

```haskell
newtype Compose f g a = Compose (f (g a))

instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose a) = Compose $ fmap (fmap f) a
```

Analog bei einem Applicative:

```haskell
instance (Applicative f, Applicative g)
         => Applicative (Compose f g) where
  pure = Compose . pure . pure
  (Compose f) <*> (Compose x) = Compose $ (<*>) <$> f <*> x
```

Seit GHC 8.0 sind diese auch im Modul `Data.Functor.Compose`[1] vertreten. Vorher wurden diese über eine Library bereitgestellt.

[1] https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-Functor-Compose.html

Bei Monaden liegt der Knackpunkt in der Definition von (>>=):

```
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
```

In der Hilfsfunktion `mb` müssen wir auf die Eigenschaften der inneren Monade zugreifen und in allen Fällen einen gültigen Wert konstruieren.

Bei Monaden liegt der Knackpunkt in der Definition von (>>=):

```
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
```

In der Hilfsfunktion `mb` müssen wir auf die Eigenschaften der inneren Monade zugreifen und in allen Fällen einen gültigen Wert konstruieren.
Für `IO` z.B. klappt so etwas nicht!

Beispiele

Wir haben schon ein paar Monaden kennengelernt. Diese kann man *fast* alle kombinieren. Wir können somit folgendes bauen:

Beispiele

Wir haben schon ein paar Monaden kennengelernt. Diese kann man *fast* alle kombinieren. Wir können somit folgendes bauen:

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

Beispiele

Wir haben schon ein paar Monaden kennengelernt. Diese kann man *fast* alle kombinieren. Wir können somit folgendes bauen:

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

Wie schreiben wir nun Code dafür?

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

Beispiele

Weitere Monaden, die hierbei häufig vorkommen, sind z.B.

- `ReaderT` für ein read-only-Environment (z.B. Konfiguration)

Beispiele

Weitere Monaden, die hierbei häufig vorkommen, sind z.B.

- `ReaderT` für ein read-only-Environment (z.B. Konfiguration)
- `WriterT` für ein write-only-Environment (z.B. für Logging)

Beispiele

Weitere Monaden, die hierbei häufig vorkommen, sind z.B.

- `ReaderT` für ein read-only-Environment (z.B. Konfiguration)
- `WriterT` für ein write-only-Environment (z.B. für Logging)
- `StateT` für einen globalen State

Beispiele

Weitere Monaden, die hierbei häufig vorkommen, sind z.B.

- `ReaderT` für ein read-only-Environment (z.B. Konfiguration)
- `WriterT` für ein write-only-Environment (z.B. für Logging)
- `StateT` für einen globalen State
- `EitherT` für fehlschlagbare Operationen (mit Fehlermeldung)

Beispiele

Weitere Monaden, die hierbei häufig vorkommen, sind z.B.

- `ReaderT` für ein read-only-Environment (z.B. Konfiguration)
- `WriterT` für ein write-only-Environment (z.B. für Logging)
- `StateT` für einen globalen State
- `EitherT` für fehlschlagbare Operationen (mit Fehlermeldung)
- `MaybeT` für fehlschlagbare Operationen (ohne Fehlermeldung)

Beispiele

Weitere Monaden, die hierbei häufig vorkommen, sind z.B.

`ReaderT`	für ein read-only-Environment (z.B. Konfiguration)
`WriterT`	für ein write-only-Environment (z.B. für Logging)
`StateT`	für einen globalen State
`EitherT`	für fehlschlagbare Operationen (mit Fehlermeldung)
`MaybeT`	für fehlschlagbare Operationen (ohne Fehlermeldung)

Je nachdem, welche Möglichkeiten man haben möchte, kann man diese miteinander kombinieren.

Beispiele

Auch kommt es auf die Reihenfolge an:

```
StateT MyState (EitherT String (Identity a))
```

kann fehlschlagen, aber man kommt nach dem Fehlschlag noch an den `State` dran,

Beispiele

Auch kommt es auf die Reihenfolge an:

```
StateT MyState (EitherT String (Identity a))
```

kann fehlschlagen, aber man kommt nach dem Fehlschlag noch an den `State` dran, wohingegen

```
EitherT String (StateT MyState (Identity a))
```

nur die Fehlermeldung liefert und den State schon entsorgt hat.

Beispiele

Auch kommt es auf die Reihenfolge an:

```
StateT MyState (EitherT String (Identity a))
```

kann fehlschlagen, aber man kommt nach dem Fehlschlag noch an den `State` dran, wohingegen

```
EitherT String (StateT MyState (Identity a))
```

nur die Fehlermeldung liefert und den State schon entsorgt hat.
Häufig findet man einen Read-Write-State-Transformer, kurz `RWST`.

Beispiele

Auch kommt es auf die Reihenfolge an:

```
StateT MyState (EitherT String (Identity a))
```

kann fehlschlagen, aber man kommt nach dem Fehlschlag noch an den `State` dran, wohingegen

```
EitherT String (StateT MyState (Identity a))
```

nur die Fehlermeldung liefert und den State schon entsorgt hat.
Häufig findet man einen Read-Write-State-Transformer, kurz `RWST`.
Echtweltprogramme sind oft durch einen `RWST IO` mit der Außenwelt verbunden.

Beispiele

Ein Echtwelt-Beispiel könnte etwa der folgende Aufruf sein:

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

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

Beispiele

Ein Echtwelt-Beispiel könnte etwa der folgende Aufruf sein:

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

readInputs :: ReaderT Env IO String
readInputs = do
           e <- ask
           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 <- ask
         f <- liftIO $ getUserInput (keySettings e)
         oldWorld <- get
         let newWorld = updateWorld f oldWorld
         put newWorld
         unless (f == endKey e) mainLoop
```

Beispiele

Noch ein Beispiel aus einem Spiel könnte sein:

```
mainLoop :: RWST Env () State IO ()
mainLoop = do
         e <- ask
         f <- liftIO $ getUserInput (keySettings e)
         oldWorld <- get
         let newWorld = updateWorld f oldWorld
         put newWorld
         unless (f == endKey e) mainLoop
```

Dies ist eine klassische 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`).

Beispiele

Noch ein Beispiel aus einem Spiel könnte sein:

```
mainLoop :: RWST Env () State IO ()
mainLoop = do
         e <- ask
         f <- liftIO $ getUserInput (keySettings e)
         oldWorld <- get
         let newWorld = updateWorld f oldWorld
         put newWorld
         unless (f == endKey e) mainLoop
```

Dies ist eine klassische 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`).

Wichtig: `updateWorld` ist pure!

Fragen?