Intermediate Functional Programming in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

1. Record-Syntax
2. State-Monad
3. Monad-Transformer
4. Monad-Transformer cont.

Nehmen wir an, wir wollen den folgenden Produkttypen definieren:

```
data V2 a = V2 a a
```

Nehmen wir an, wir wollen den folgenden Produkttypen definieren:

```haskell
data V2 a = V2 a a
```

Nun wollen wir auch einzelne Elemente auslesen und setzen. Dafür müssen wir ein paar Hilfsfunktionen definieren

Zum Auslesen:

Zum Auslesen:

```
x :: V2 a -> a
x (V2 x' _) = x'
```

Zum Auslesen:

```
x :: V2 a -> a
x (V2 x' _) = x'

y :: V2 a -> a
y (V2 _ y') = y'
```

Zum Auslesen:

```
x :: V2 a -> a
x (V2 x' _) = x'

y :: V2 a -> a
y (V2 _ y') = y'
```

Und setzen:

Zum Auslesen:

```
x :: V2 a -> a
x (V2 x' _) = x'

y :: V2 a -> a
y (V2 _ y') = y'
```

Und setzen:

```
sx :: a -> V2 a -> V2 a
sx x (V2 _ y) = V2 x y
```

Zum Auslesen:

```
x :: V2 a -> a
x (V2 x' _) = x'

y :: V2 a -> a
y (V2 _ y') = y'
```

Und setzen:

```
sx :: a -> V2 a -> V2 a
sx x (V2 _ y) = V2 x y

sy :: a -> V2 a -> V2 a
sy y (V2 x _) = V2 x y
```

Nun sind diese Funktionen stupide zu schreiben. Also kann man das ganze automatisieren.

Nun sind diese Funktionen stupide zu schreiben. Also kann man das ganze automatisieren.
Dies nennt sich Record-Syntax:

Nun sind diese Funktionen stupide zu schreiben. Also kann man das ganze automatisieren.
Dies nennt sich Record-Syntax:

```
data V2 a = V2 { x :: a
               , y :: a
               }
```

Nun sind diese Funktionen stupide zu schreiben. Also kann man das ganze automatisieren.
Dies nennt sich Record-Syntax:

```
data V2 a = V2 { x :: a
               , y :: a
               }
```

welches uns automatisch die Funktionen

```
x :: V2 a -> a
y :: V2 a -> a
```

generiert.

Auch werden setter generiert, die wie folgt zu verwenden sind:

Auch werden setter generiert, die wie folgt zu verwenden sind:

```
let a = V2 1 2
let b = a { x = 0 }
-- b = V2 0 2
```

Auch werden setter generiert, die wie folgt zu verwenden sind:

```
let a = V2 1 2
let b = a { x = 0 }
-- b = V2 0 2
```

Wir geben also einfach die Struktur an (hier: a) und in den geschweiften Klammern alle Parameter, die wir ändern wollen.

Nochmal ein komplizierteres Beispiel:

Nochmal ein komplizierteres Beispiel:
Welche Funktionen generiert folgender Code?

```
data D a = K a (a -> a)
```

Nochmal ein komplizierteres Beispiel:
Welche Funktionen generiert folgender Code?

```
data D a = K a (a -> a)

K :: a -> (a -> a) -> D a
```

Nochmal ein komplizierteres Beispiel:
Welche Funktionen generiert folgender Code?

```
data D a = K a (a -> a)

K :: a -> (a -> a) -> D a
```

Welche Funktionen generiert die Record-Syntax?

```
data D a = K { x :: a, y :: (a -> a) }
```

Nochmal ein komplizierteres Beispiel:
Welche Funktionen generiert folgender Code?

```
data D a = K a (a -> a)

K :: a -> (a -> a) -> D a
```

Welche Funktionen generiert die Record-Syntax?

```
data D a = K { x :: a, y :: (a -> a) }

K :: a -> (a -> a) -> D a
x :: D a -> a
y :: D a -> (a -> a)
```

Nochmal ein komplizierteres Beispiel:
Welche Funktionen generiert folgender Code?

```
data D a = K a (a -> a)

K :: a -> (a -> a) -> D a
```

Welche Funktionen generiert die Record-Syntax?

```
data D a = K { x :: a, y :: (a -> a) }

K :: a -> (a -> a) -> D a
x :: D a -> a
y :: D a -> (a -> a)
```

Die Record-Syntax beschert uns also kostenlos Getter-Funktionen und bietet die Möglichkeit eines Setzens über die update-Notation `a { x = y }`

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))
```

Definition von State:

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

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

`runState` benötigt also 2 Argumente, damit es ein `(a,s)` liefert.

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

`runState` benötigt also 2 Argumente, damit es ein `(a,s)` liefert.

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

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

`runState` benötigt also 2 Argumente, damit es ein `(a,s)` liefert.

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

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

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

`runState` benötigt also 2 Argumente, damit es ein `(a,s)` liefert.

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

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

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

`runState` benötigt also 2 Argumente, damit es ein `(a,s)` liefert.

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

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

Definition von State:

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

Diese (Record-)Notation liefert uns 2 Funktionen:

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

`runState` benötigt also 2 Argumente, damit es ein `(a,s)` liefert.

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

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

State in der monadischen Form fügt einfach nur einen Funktionsparameter `s` hinzu und versteckt das `(b,s)` und gibt lediglich das `b` in der do-Notation zurück.

Hilfreich ist es, sich die State-Monade als Berechnung vorzustellen, die noch nicht ausgeführt werden kann, weil der **initiale** State noch nicht bekannt ist.

Hilfreich ist es, sich die State-Monade als Berechnung vorzustellen, die noch nicht ausgeführt werden kann, weil der **initiale** State noch nicht bekannt ist.

Man **bekommt** also erst einen State, bearbeitet ihn ggf. und gibt dann den geänderten State weiter.

Hilfreich ist es, sich die State-Monade als Berechnung vorzustellen, die noch nicht ausgeführt werden kann, weil der **initiale** State noch nicht bekannt ist.

Man **bekommt** also erst einen State, bearbeitet ihn ggf. und gibt dann den geänderten State weiter.
Dies spiegelt sich auch in der Funktor-Instanz wieder:

```haskell
instance Functor (State s) where
  fmap f rs = _
```

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

```haskell
instance Functor (State s) where
  fmap f rs = _
```

```haskell
State :: (s -> (b,s)) -> State s b
```

```haskell
instance Functor (State s) where
  fmap f rs = State $ _
```

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

```haskell
instance Functor (State s) where
  fmap f rs = State $ \s -> _
```

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

```haskell
instance Functor (State s) where
  fmap f rs = State $ \s -> _
```

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

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

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

```
Found hole '_' with type: (b, s)
Where: 's' is a rigid type variable
       'b' is a rigid type variable
Relevant bindings include
  a :: a
  s' :: s
  s :: s
  rs :: State s a
  f :: a -> b
  fmap :: (a -> b) -> State s a -> State s b
```

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

```
Found hole '_' with type: s
Where: 's' is a rigid type variable
Relevant bindings include
  a :: a
  s' :: s
  s :: s
  rs :: State s a
  f :: a -> b
  fmap :: (a -> b) -> State s a -> State s b
```

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

Danke, typed holes!

Ganz analog funktioniert die Applicative-Instanz:

Ganz analog funktioniert die Applicative-Instanz:

```
instance Applicative (State s) where
    pure a    = _
    rf <*> rs = undefined
```

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

Ganz analog funktioniert die Applicative-Instanz:

```
instance Applicative (State s) where
    pure a    = State $ \s -> _
    rf <*> rs = undefined
```

```
Found hole ‘_’ with type: (a, s)
Where: ‘s’ is a rigid type variable
       ‘a’ is a rigid type variable
Relevant bindings include
  s :: s
  a :: a
  pure :: a -> State s a
```

Ganz analog funktioniert die Applicative-Instanz:

```
instance Applicative (State s) where
    pure a    = State $ \s -> (a,s)
    rf <*> rs = State $ \s -> _
```

```
Found hole ‘_’ with type: (b, s)
Where: ‘s’ is a rigid type variable
       ‘b’ is a rigid type variable
Relevant bindings include
  s :: s
  rs :: State s a
  rf :: State s (a -> b)
  (<*>) :: State s (a -> b) -> State s a -> State s b
```

Ganz analog funktioniert die Applicative-Instanz:

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

Wichtig: Erst das `rf` ausführen, dann das `rs`, da <*> von links-nach-rechts arbeitet.

Ganz analog funktioniert die Applicative-Instanz:

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

```
Found hole '_' with type: (b, s)
Where: 's' is a rigid type variable
       'b' is a rigid type variable
Relevant bindings include
  a :: a
  s'' :: s
  f :: a -> b
  s' :: s
  s :: s
  rs :: State s a
  rf :: State s (a -> b)
  (<*>) :: State s (a -> b) -> State s a -> State s b
```

Ganz analog funktioniert die Applicative-Instanz:

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

Am wichtigsten ist die Monad-Instanz:

Am wichtigsten ist die Monad-Instanz:

```
instance Monad (State s) where
    return   = pure
    rs >>= f = State $ \s -> _
```

```
Found hole '_' with type: (b, s)
Where: 's' is a rigid type variable
       'b' is a rigid type variable
Relevant bindings include
  s :: s
  f :: a -> State s b
  rs :: State s a
  (>>=) :: State s a -> (a -> State s b) -> State s b
```

Am wichtigsten ist die Monad-Instanz:

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

```
Found hole '_' with type: (b, s)
Where: 's' is a rigid type variable
       'b' is a rigid type variable
Relevant bindings include
  a :: a
  s' :: s
  s :: s
  f :: a -> State s b
  rs :: State s a
  (>>=) :: State s a -> (a -> State s b) -> State s b
```

Am wichtigsten ist die Monad-Instanz:

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

```
Found hole '_' with type: (b, s)
Where: 's' is a rigid type variable
       'b' is a rigid type variable
Relevant bindings include
  rs' :: State s b
  a :: a
  s' :: s
  s :: s
  f :: a -> State s b
  rs :: State s a
  ...
```

Am wichtigsten ist die Monad-Instanz:

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

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

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

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.

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"
           case getFirstMail folder of
             (Just mail) ->
                case getHeader mail of
                  (Just head) -> return $ return head
                  Nothing     -> return Nothing
             Nothing     -> return 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):

```
newtype 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):

```
newtype 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:

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:

```haskell
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
```

Applicative:

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
```

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
```

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
```

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
```

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.

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
```

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
```

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
```

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
```

Monad:

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

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

```haskell
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
```

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 . pure
  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 . 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

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

```
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
```

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

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

```haskell
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.
Genereller:

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

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.
Genereller:

```
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.

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.
Genereller:

```
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.

Wichtig: IO ist nicht additiv! Es gibt keinen IO-T!

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 MyState
                             (EitherT String
                                      (MaybeT (IO a)))
```

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

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

Wie schreiben wir nun hierin Code?

```
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 ()
```

Um auf spezielle Ebenen im Monad-Stack zuzugreifen gibt es (z.B. in der Bibliothek `mtl`) für jeden Zweck eine Typklasse.
Beispielsweise:

Um auf spezielle Ebenen im Monad-Stack zuzugreifen gibt es (z.B. in der Bibliothek `mtl`) 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 (z.B. in der Bibliothek `mtl`) 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
    -- manipulateState :: MyState -> MyState
    liftIO $ putStrLn "foo"
```

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

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

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. fürs Logging)

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. fürs Logging)
- `StateT` für einen globalen State

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. fürs Logging)
- `StateT` für einen globalen State
- `EitherT` für fehlschlagbare Operationen (mit Fehlermeldung)

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. fürs Logging)
- `StateT` für einen globalen State
- `EitherT` für fehlschlagbare Operationen (mit Fehlermeldung)
- `MaybeT` für fehlschlagbare Operationen (ohne Fehlermeldung)

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. fürs Logging)
- `StateT` für einen globalen State
- `EitherT` für fehlschlagbare Operationen (mit Fehlermeldung)
- `MaybeT` für fehlschlagbare Operationen (ohne Fehlermeldung)

Jenachdem, welche Möglichkeiten man haben möchte, kann man diese kombinieren.

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 heran

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 heran, wohingegen

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

nur die Fehlermeldung liefert und den State schon entsorgt hat.

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 heran, 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`.

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 heran, 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.

Ein weiteres 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
```

Ein weiteres 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
          newWorld <- updateWorld f oldWorld
          put newWorld
          unless (f == endKey e) mainLoop
```

Noch ein Beispiel aus einem Spiel könnte sein:

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

Noch ein Beispiel aus einem Spiel könnte sein:

```
mainLoop :: RWST Env () State IO ()
mainLoop = do
         e <- ask
         f <- liftIO $ getUserInput (keySettings e)
         oldWorld <- get
         newWorld <- updateWorld f oldWorld
         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).
Wichtig: `updateWorld` ist pure.