Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

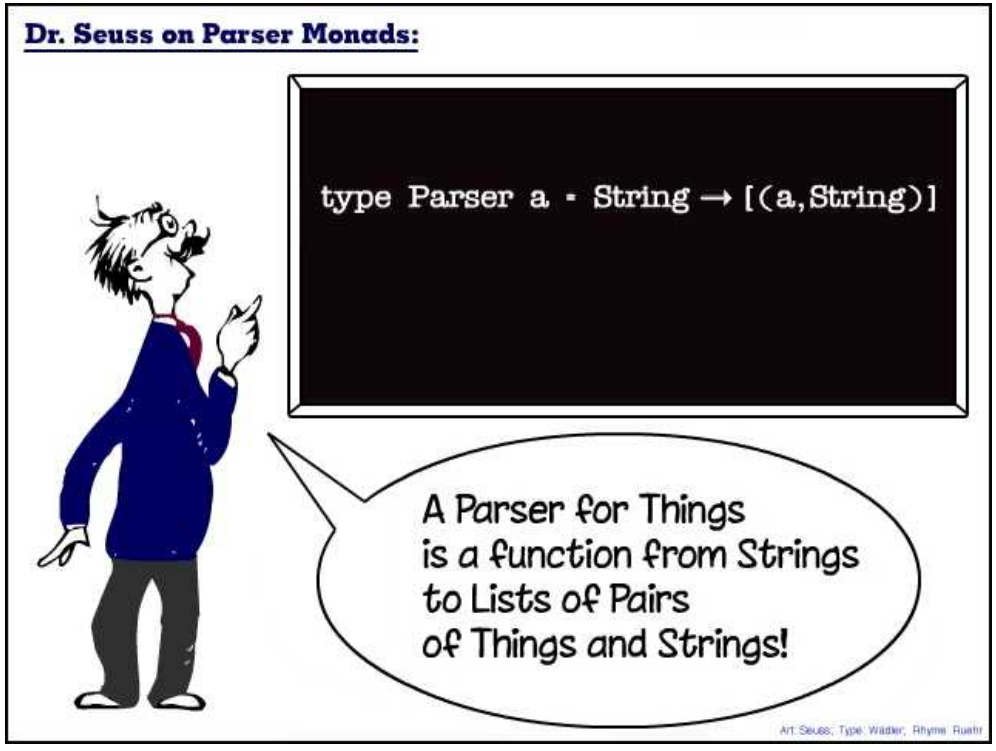
Vorlesung 6: *State und Monad Transformer*
Stand: 20. Mai 2016

Wiederholung: Parser

Was war ein Parser?

Was war ein Parser?

Die Möglichkeit, die in der Vorlesung vorgestellt wurde, war:



Quelle: `http://www.willamette.edu/~fruehr/haskell/seuss.html`

Was war ein Parser?

Wir benutzen Parser in Haskell klassischerweise als monadische Parserkombinatioren.
Das bedeutet, dass wir nur die elementarsten Parser selbst schreiben (Zeichen, Bitmuster, ...) und diese dann zu größeren Parsern zusammenstecken.

Was war ein Parser?

Wir benutzen Parser in Haskell klassischerweise als monadische Parserkombinatioren.
Das bedeutet, dass wir nur die elementarsten Parser selbst schreiben (Zeichen, Bitmuster, ...) und diese dann zu größeren Parsern zusammenstecken.

Hier helfen uns die Eigenschaften von `Applicative` und `Monad`:

Was war ein Parser?

Wir benutzen Parser in Haskell klassischerweise als monadische Parserkombinatioren.
Das bedeutet, dass wir nur die elementarsten Parser selbst schreiben (Zeichen, Bitmuster, ...) und diese dann zu größeren Parsern zusammenstecken.

Hier helfen uns die Eigenschaften von `Applicative` und `Monad`:

- `Monad`-Instanz erlaubt es uns Parser auszuführen und nur die Erfolgsfälle zu berücksichtigen
- `(<|>)` erlaubt uns Parser zu probieren, bis einer funktioniert
- weitere Kombinatoren wie `many`, `some`, etc. decken automatisch Wiederholungen ab.

Was war ein Parser?

Wir benutzen Parser in Haskell klassischerweise als monadische Parserkombinatioren.
Das bedeutet, dass wir nur die elementarsten Parser selbst schreiben (Zeichen, Bitmuster, ...) und diese dann zu größeren Parsern zusammenstecken.

Hier helfen uns die Eigenschaften von `Applicative` und `Monad`:

- `Monad`-Instanz erlaubt es uns Parser auszuführen und nur die Erfolgsfälle zu berücksichtigen
- `(<|>)` erlaubt uns Parser zu probieren, bis einer funktioniert
- weitere Kombinatoren wie `many`, `some`, etc. decken automatisch Wiederholungen ab.

Kurzum: Man gibt nur an, welches Ergebnis akzeptabel ist und die Monadeninstanz kümmert sich um alle Fehlerbehandlungen.

Verständnisfrage

Was tut dieser Parser?

```
unknown = do
        a <- option '+' (char '+' <|> char '-')
        b <- decimal
        c <- option 0 (do
                        char '.'
                        decimal)
        return $ UnknownType a b c
```

Verständnisfrage

Was tut dieser Parser?

```
unknown = do
        a <- option '+' (char '+' <|> char '-')
        b <- decimal
        c <- option 0 (do
                        char '.'
                        decimal)
        return $ UnknownType a b c
```

Dies ist ein Parser für Zahlen der Form ± 123.456.

Parser mal anders

Allerdings gibt es auch noch eine andere Möglichkeit Parser zu definieren:

```
data Parser a = Parser (String -> Either String (a,String))
```

Parser mal anders

Allerdings gibt es auch noch eine andere Möglichkeit Parser zu definieren:

```
data Parser a = Parser (String -> Either String (a,String))
```

Hier haben wir statt einer „List of Successes“ nur einen Erfolg und andernfalls eine Fehlermeldung.
Dies ist aber kein Problem, da wir mehrere Erfolge ganz einfach über `(<|>)` aus der `Alternative`-Typklasse simulieren können.

Parser mal anders

Gegeben nun eine Funktion

```
fun :: a -> b -> Parser c
```

Parser mal anders

Gegeben nun eine Funktion

```
fun :: a -> b -> Parser c
```

Dann kann man die Definition einsetzen und erhält:

```
fun :: a -> b -> (String -> Either String (c,String))
```

Parser mal anders

Gegeben nun eine Funktion

```
fun :: a -> b -> Parser c
```

Dann kann man die Definition einsetzen und erhält:

```
fun :: a -> b -> (String -> Either String (c,String))
```

und durch weglassen der Klammern erhalten wir:

```
fun :: a -> b -> String -> Either String (c,String)
```

Also fügt der Parser einen String als letzten Funktionsparameter hinzu und hat statt `c` einen erweiterten Rückgabetypen mit Fehlerzustand.

Parser mal anders

Da wir gesehen haben, dass der Parser eine Monade ist und die Information des Reststrings einfach nur „durchreicht“, können wir uns fragen, ob das nicht für alle Typen geht.

Parser mal anders

Da wir gesehen haben, dass der Parser eine Monade ist und die Information des Reststrings einfach nur „durchreicht“, können wir uns fragen, ob das nicht für alle Typen geht.

Stellt sich heraus: Natürlich geht das.

State

Oder: Wie bekomme ich eigentlich „richtige“ Variablen?

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

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

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

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

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

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

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

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

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

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

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

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

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

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

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

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

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

Definition

Die Definition des `State`-Datentyps sieht sehr ähnlich zum Parser aus:

```
newtype State s a = State (s -> (a,s))
```

Wir übergeben nun einen zweiten Parameter `s`, der die Art des Zustandes übergibt. (Im Falle des Parsers war dies ein String)
Zusätzlich zu dem Konstruktor definieren wir uns noch eine Funktion, die die interne Funktion „auspackt“.

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

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

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

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

State in der monadischen Form fügt somit ebenfalls einfach nur einen Funktionsparameter `s` hinzu, versteckt das Ergebnis `(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 ein **Anfangzustand** (initial State) noch nicht bekannt ist.

Hilfreich ist es, sich die `State`-Monade als Berechnung vorzustellen, die noch nicht ausgeführt werden kann, weil ein **Anfangzustand** (initial State) noch nicht bekannt ist.

Man bekommt also erst *später* irgendwann 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 ein **Anfangzustand** (initial State) noch nicht bekannt ist.

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

Dies spiegelt sich auch in der `Functor`-Instanz wieder:

Functor/Applicative/Monad

```
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
```

Functor/Applicative/Monad

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

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

Functor/Applicative/Monad

```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
```

Functor/Applicative/Monad

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

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

Functor/Applicative/Monad

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

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

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

Functor/Applicative/Monad

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

Functor/Applicative/Monad

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

```
Found hole '_' with type: s
Where: 's' is a rigid type variable
Relevant bindings include
  a :: a
  st' :: s
  st :: 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 $ \st -> let (a,st') = runState rs st
                             in (f a, st')
```

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 $ _
    rf <*> rs = undefined
```

```
Found hole '_' with type: s -> (a, s)
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 $ \st -> _
    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
  st :: s
  a :: a
  pure :: a -> State s a
```

Ganz analog funktioniert die `Applicative`-Instanz:

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

```
Found hole ‘_’ with type: (b, s)
Where: ‘s’ is a rigid type variable
       ‘b’ is a rigid type variable
Relevant bindings include
  st :: 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 $ \st -> (a,st)
    rf <*> rs = State $ \st ->
                  let (f,st')  = runState rf st
                      (a,st'') = runState rs st'
                  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 $ \st -> (a,st)
    rf <*> rs = State $ \st ->
                  let (f,st')  = runState rf st
                      (a,st'') = runState rs st'
                  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
  st'' :: s
  f :: a -> b
  st' :: s
  st :: 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 $ \st -> (a,st)
    rf <*> rs = State $ \st ->
                  let (f,st')  = runState rf st
                      (a,st'') = runState rs st'
                  in (f a, st'')
```

Am wichtigsten ist die `Monad`-Instanz:

Am wichtigsten ist die `Monad`-Instanz:

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

```
Found hole ‘_’ with type: (b, s)
Where: ‘s’ is a rigid type variable
       ‘b’ is a rigid type variable
Relevant bindings include
  st :: 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 $ \st ->
                 let (a,st') = runState rs st
                 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
  st' :: s
  st :: 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 $ \st ->
                let (a,st') = runState rs st
                    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
  st' :: s
  st :: 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 $ \st ->
                let (a,st') = runState rs st
                    rs'     = f a
                in runState rs' st'
```

Anwendung

Nun haben wir das alles definiert, aber wie wenden wir das nun an?

Anwendung

Nun haben wir das alles definiert, aber wie wenden wir das nun an?

Ein paar Kleinigkeiten fehlen noch:

- Wie kommen wir an den Zustand dran?
- Wie können wir ihn ersetzen?
- Wie können wir ihn modifizieren?

Anwendung

Nun haben wir das alles definiert, aber wie wenden wir das nun an?

Ein paar Kleinigkeiten fehlen noch:

- Wie kommen wir an den Zustand dran?
- Wie können wir ihn ersetzen?
- Wie können wir ihn modifizieren?

Also benötigen wir noch

```
get    ::                State s s
put    ::          s  -> State s ()
modify :: (s -> s) -> State s ()
```

Anwendung

Ein `get` bedeutet nur, dass wir den internen Zustand in die Ausgabe kopieren müssen:

```
get :: State s s
get = State $ (\s -> (s,s))
```

Anwendung

Ein `get` bedeutet nur, dass wir den internen Zustand in die Ausgabe kopieren müssen:

```
get :: State s s
get = State $ (\s -> (s,s))
```

Bei `put` bekommen wir einen Zustand rein und schmeissen den alten weg:

```
put :: s -> State s ()
put s = State $ (\_ -> ((),s))
```

Anwendung

Ein `get` bedeutet nur, dass wir den internen Zustand in die Ausgabe kopieren müssen:

```
get :: State s s
get = State $ (\s -> (s,s))
```

Bei `put` bekommen wir einen Zustand rein und schmeissen den alten weg:

```
put :: s -> State s ()
put s = State $ (\_ -> ((),s))
```

Und bei `modify` wenden wir einfach die Funktion an, die wir bekommen:

```
modify :: (s -> s) -> State s ()
modify f = State $ (\s -> ((),f s))
```

Anwendung

Zur Illustration einer Anwendung schreiben wir nun ein einfaches Spiel mit einer Figur:

- Diese Figur kann sich auf einem begrenztem 2-Dimensionalen Spielfeld in die vier Hauptrichtungen bewegen
- Wir möchten die Position der Figur gerne „verstecken“, sodass wir nicht aus versehen ungültige Züge machen
- Wir möchten dies „automatisch“ kombinieren

Anwendung

Zunächst beginnen wir wieder mit einer Datendefinition:

```
type Position = (Int, Int)
type Dimension = (Int, Int)
data GameState = GameState Position Dimension
```

Anwendung

Zunächst beginnen wir wieder mit einer Datendefinition:

```
type Position = (Int, Int)
type Dimension = (Int, Int)
data GameState = GameState Position Dimension
```

Dies ist für das Beispiel extra schlicht gehalten. In einem echten Projekt würde das etwa so aussehen:

```
data GameState = GameState
               { playerPosition  :: Position
               , obstacles       :: [Position]
               , boardDimensions :: Dimension
               , score           :: Int
               -- .....
               }
```

Anwendung

Zum Bewegen müssen wir die vier Richtungen definieren. Dies funktioniert bei allen Richtungen ähnlich, sodass wir hier nur exemplarisch eine präsentieren:

Anwendung

Zum Bewegen müssen wir die vier Richtungen definieren. Dies funktioniert bei allen Richtungen ähnlich, sodass wir hier nur exemplarisch eine präsentieren:

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

Anwendung

Zum Bewegen müssen wir die vier Richtungen definieren. Dies funktioniert bei allen Richtungen ähnlich, sodass wir hier nur exemplarisch eine präsentieren:

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

Wenn wir nun diese Funktion `rechts` benutzen, dann können wir nicht (nach rechts) aus dem Spielfeld herauslaufen.

Anwendung

Zum Bewegen müssen wir die vier Richtungen definieren. Dies funktioniert bei allen Richtungen ähnlich, sodass wir hier nur exemplarisch eine präsentieren:

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

Wenn wir nun diese Funktion `rechts` benutzen, dann können wir nicht (nach rechts) aus dem Spielfeld herauslaufen.

In einem „richtigen“ Spiel würden hier statt `case x+1 > dx of` noch weitere Abfragen gemacht - etwa ob man das Feld betreten kann oder ob dort ein Hindernis ist etc.

Anwendung

Zum Bewegen müssen wir die vier Richtungen definieren. Dies funktioniert bei allen Richtungen ähnlich, sodass wir hier nur exemplarisch eine präsentieren:

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

Wenn wir nun diese Funktion `rechts` benutzen, dann können wir nicht (nach rechts) aus dem Spielfeld herauslaufen.

In einem „richtigen“ Spiel würden hier statt `case x+1 > dx of` noch weitere Abfragen gemacht - etwa ob man das Feld betreten kann oder ob dort ein Hindernis ist etc.

Wichtig ist, dass wir die Veränderung des Zustandes getrennt haben von dem Rückgabewert, der hier z.B. den Erfolg der Aktion angibt.

Anwendung

Kompliziertere Bewegungen und Bewegungsmuster kann man nun ganz einfach kombinieren:

```
laufKomisch = do
              rechts; rechts
              hoch;   hoch
              links;  links
              runter
```

Anwendung

Kompliziertere Bewegungen und Bewegungsmuster kann man nun ganz einfach kombinieren:

```
laufKomisch = do
              rechts; rechts
              hoch;   hoch
              links;  links
              runter
```

Auch können wir so etwas schreiben wie:

```
bisZumObjekt :: Monad m => m Bool -> Bool -> m ()
bisZumObjekt dir c = do
              if c then
                dir >>= bisZumObjekt dir
              else
                return ()
```

Dieses läuft solange in Richtung `dir`, bis ein `False` zurückkommt.

Anwendung

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.

Anwendung

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 (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 (IO (Maybe a))
```

und benutzen im Folgenden diese zwei Funktionen:

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

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

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 (IO (Maybe a))
```

und benutzen im Folgenden diese zwei Funktionen:

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

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

Nun müsser wir lediglich die Monaden-Instanz (inkl. Vorraussetzungen) schreiben.

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?

Recap

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

Recap

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

Man kann aus zwei beliebigen Funktoren einen neuen Funktor machen.

Recap

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

Man kann aus zwei beliebigen Funktoren einen neuen Funktor machen.
Ebenfalls kann man aus zwei Applicatives einen neuen Applicative machen.

Recap

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

Man kann aus zwei beliebigen Funktoren einen neuen Funktor machen.
Ebenfalls kann man aus zwei Applicatives einen neuen Applicative machen.
Aber man kann nicht Monaden beliebig verbinden.

Recap

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

Man kann aus zwei beliebigen Funktoren einen neuen Funktor machen.
Ebenfalls kann man aus zwei Applicatives einen neuen Applicative machen.
Aber man kann nicht Monaden beliebig verbinden.

Der Knackpunkt liegt 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.

Recap

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

Man kann aus zwei beliebigen Funktoren einen neuen Funktor machen.
Ebenfalls kann man aus zwei Applicatives einen neuen Applicative machen.
Aber man kann nicht Monaden beliebig verbinden.

Der Knackpunkt liegt 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!

Ausblick

Vorschau: Was machen wir nächste Woche?

- Wiederholung: Vorlesung 6
- Record-Syntax
- Lenses
- (automatisiertes) Testing

Fragen?