

Christoph Lüth



Wintersemester 2020/21



Organisatorisches

- ▶ Die Klausur am 03.02. ist gestern von der Uni **abgesagt** worden.
- ▶ Es bleibt der Klausurtermin am 21.04.2020.
- ▶ Wir bemühen uns um eine zusätzlichen Wiederholungstermin im Sommersemester.



Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**
 - ▶ Aktionen und Zustände
 - ▶ **Monaden als Berechnungsmuster**
 - ▶ Funktionale Webanwendungen
 - ▶ Scala — Eine praktische Einführung
 - ▶ Rückblick & Ausblick



Inhalt

- ▶ Wie geht das mit IO?
- ▶ Monaden als allgemeines Berechnungsmuster
- ▶ Fallbeispiel: Auswertung von Ausdrücken

Lernziele

Wir verstehen, wie wir Berechnungsmuster wie Seiteneffekte, Partialität oder Mehrdeutigkeit in Haskell funktional modellieren.



I. Zustandsabhängige Berechnungen



Funktionen mit Zustand

- ▶ Idee: Seiteneffekt **explizit** machen
- ▶ Funktion $f : \alpha \rightarrow \beta$ mit Seiteneffekt in **Zustand** σ :

$$\begin{aligned} f &: \alpha \times \sigma \rightarrow \beta \times \sigma \\ &\cong \\ f &: \alpha \rightarrow \sigma \rightarrow \beta \times \sigma \end{aligned}$$

- ▶ Datentyp für Zustand $\sigma : \sigma \rightarrow \beta \times \sigma$
- ▶ Komposition: Funktionskomposition und **uncurry**

```
curry  :: (( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ )  $\rightarrow$   $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$   $\gamma$   
uncurry :: ( $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$   $\gamma$ )  $\rightarrow$  ( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ 
```



In Haskell: Zustände **explizit**

- ▶ **Zustandstransformer**: Berechnung mit Seiteneffekt in Typ σ (polymorph über α)

```
type State  $\sigma$   $\alpha$  =  $\sigma$   $\rightarrow$  ( $\alpha$ ,  $\sigma$ )
```

- ▶ Komposition zweier solcher Berechnungen:

```
comp :: State  $\sigma$   $\alpha$   $\rightarrow$  ( $\alpha$   $\rightarrow$  State  $\sigma$   $\beta$ )  $\rightarrow$  State  $\sigma$   $\beta$   
comp f g = uncurry g  $\circ$  f
```

- ▶ Trivialer Zustand:

```
lift ::  $\alpha$   $\rightarrow$  State  $\sigma$   $\alpha$   
lift = curry id
```

- ▶ Lifting von Funktionen:

```
map :: ( $\alpha$   $\rightarrow$   $\beta$ )  $\rightarrow$  State  $\sigma$   $\alpha$   $\rightarrow$  State  $\sigma$   $\beta$   
map f g = ( $\lambda$ (a, s)  $\rightarrow$  (f a, s))  $\circ$  g
```



Zugriff auf den Zustand

- ▶ Zustand lesen:

```
get :: ( $\sigma$   $\rightarrow$   $\alpha$ )  $\rightarrow$  State  $\sigma$   $\alpha$   
get f s = (f s, s)
```

- ▶ Zustand setzen:

```
set :: ( $\sigma$   $\rightarrow$   $\sigma$ )  $\rightarrow$  State  $\sigma$  ()  
set g s = ((), g s)
```



Einfaches Beispiel

- ▶ Zähler als Zustand:

```
type WithCounter α = State Int α
```

- ▶ Beispiel: Funktion, die in Kleinbuchstaben konvertiert und **zählt**

```
cntToL :: String → WithCounter String
cntToL [] = lift ""
cntToL (x:xs)
  | isUpper x = cntToL xs 'comp'
                λys → set (+1) 'comp'
                λ() → lift (toLower x: ys)
  | otherwise = cntToL xs 'comp' λys → lift (x: ys)
```

- ▶ Hauptfunktion (verkapselt State):

```
cntToLower :: String → (String, Int)
cntToLower s = cntToL s 0
```

Food for Thought

Übung 11.1: Verkapselung

Warum **müssen** wir den Datentyp `State σ α` in einen Datentyp verkapseln, und wie sieht dessen Signatur aus?

Lösung: Wenn wir den Zustand explizit durch die Gegend reichen, können wir ihn beliebig kopieren — das ist sicherlich nicht beabsichtigt, es sollte immer nur genau eine Kopie des Zustands geben.

Die Signatur besteht aus `comp`, `lift`, `map`, `get` und `set` — siehe nächsten Abschnitt.

II. Monaden

Monaden als Berechnungsmuster

- ▶ In `cntToL` werden zustandsabhängige Berechnungen verkettet.
- ▶ So ähnlich wie bei Aktionen!

State:

```
type State σ α
```

```
comp :: State σ α →
      (α → State σ β) →
      State σ β
```

```
lift :: α → State σ α
```

```
map :: (α → β) → State σ α → State σ β
```

Aktionen:

```
type IO α
```

```
(=>) :: IO α →
      (α → IO β) →
      IO β
```

```
return :: α → IO α
```

```
fmap :: (α → β) → IO α → IO β
```

Berechnungsmuster — **Monade**

Was ist ein Berechnungsmuster?

- ▶ Ein **Berechnungsmuster** hat eine **Einheit** und kann **verknüpft** werden.
- ▶ Beispiele:
 - ▶ **Seiteneffekte** (Zustand),
 - ▶ **Fehler** (Partialität),
 - ▶ **Mehrdeutigkeit**,
 - ▶ **Aktionen**.
- ▶ Eine Monade ist ein **Typkonstruktor**, der zu einem Typ **Berechnungsmuster** **hinzufügt**.
- ▶ **Mathematisch** ist eine Monade eine **verallgemeinerte algebraische Theorie** (durch Operationen und Gleichungen definiert).

Monaden in Haskell

- ▶ Monaden sind erstmal Funktoren:

```
class Functor f where
  fmap :: (α → β) → f α → f β
```

- ▶ Es sollte gelten (kann nicht geprüft werden):

```
fmap id = id
fmap f ∘ fmap g = fmap (f ∘ g)
```

- ▶ Standard: "Instances of Functor should satisfy the following laws."

Monaden in Haskell

- ▶ Verkettung (`>>=`) und Lifting (`return`):

```
class (Functor m, Applicative m) => Monad m where
  (>>=) :: m α → (α → m β) → m β
  return :: α → m α
```

`>>=` ist assoziativ und `return` das neutrale Element:

```
return a >>= k == k a
m >>= return == m
m >>= (x → k x >>= h) == (m >>= k) >>= h
```

- ▶ Auch diese Eigenschaften können nicht geprüft werden.
- ▶ Den syntaktischen Zucker (`do`-Notation) gibt's umsonst dazu.

Beispiele für Monaden

- ▶ Zustandsmonaden: `ST`, `State`, `Reader`, `Writer`
- ▶ Fehler und Ausnahmen: `Maybe`, `Either`
- ▶ Mehrdeutige Berechnungen: `List`, `Set`

Die Reader-Monade

- Aus dem Zustand wird nur gelesen:

```
data Reader σ α = R {run :: σ → α}
```

- Instanzen:

```
instance Functor (Reader σ) where
  fmap f (R g) = R (f . g)
```

```
instance Monad (Reader σ) where
  return a = R (const a)
  R f >>= g = R $ λs → run (g (f s)) s
```

- Nur eine elementare Operation:

```
get :: (σ → α) → Reader σ α
get f = R $ λs → f s
```

PI3 WS 20/21

17 [37]



Fehler und Ausnahmen

- Maybe und Either als Monade:

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing = Nothing
```

```
instance Functor (Either ε) where
  fmap f (Right b) = Right (f b)
  fmap f (Left a) = Left a
```

```
instance Monad Maybe where
  Just a >>= g = g a
  Nothing >>= g = Nothing
  return = Just
```

```
instance Monad (Either ε) where
  Right b >>= g = g b
  Left a >>= _ = Left a
  return = Right
```

- Berechnungsmodell: **Ausnahmen** (Fehler)

- $f :: \alpha \rightarrow \text{Maybe } \beta$ ist Berechnung mit möglichem (unspezifiziertem) Fehler,
- $f :: \alpha \rightarrow \text{Either } \epsilon \alpha$ ist Berechnung mit möglichem Fehler vom Typ ϵ
- Fehlerfreie Berechnungen werden verkettet
- Fehler (`Nothing` oder `Left x`) werden propagiert

PI3 WS 20/21

18 [37]



Mehrdeutigkeit

- List als Monade:

- Können wir so nicht hinschreiben, Syntax vordefiniert

```
instance Functor [α] where
  fmap = map
```

```
instance Monad [α] where
  a : as >>= g = g a ++ (as >>= g)
  [] >>= g = []
  return a = [a]
```

- Berechnungsmodell: Mehrdeutigkeit

- $f :: \alpha \rightarrow [\beta]$ ist Berechnung mit **mehreren** möglichen Ergebnissen
- Verkettung: Anwendung der folgenden Funktion auf **jedes** Ergebnis

PI3 WS 20/21

19 [37]



Beispiel

- Berechnung aller Permutationen einer Liste:

- 1 Ein Element überall in eine Liste einfügen:

```
ins :: α → [α] → [[α]]
ins x [] = return [x]
ins x (y:ys) = [x:y:ys] ++ do
  is ← ins x ys
  return $ y:is
```

- 2 Damit Permutationen (rekursiv):

```
perms :: [α] → [[α]]
perms [] = return []
perms (x:xs) = do
  ps ← perms xs
  is ← ins x ps
  return is
```

PI3 WS 20/21

20 [37]



Jetzt seid ihr dran.

Übung 11.2: Komposition in der Listenmonade

Betrachten wir noch mal die Komposition in der Listenmonade:

```
a : as >>= g = g a ++ (as >>= g)
[] >>= g = []
```

Welche uns (hoffentlich) wohlbekannte Funktion versteckt sich dahinter?

Lösung: Das ist dasselbe wie `concatMap`, nur mit umgedrehten Argumenten:

```
concatMap :: (α → [β]) → [α] → [β]
concatMap f = concat ∘ map f
```

```
(>>=) = flip concatMap
```

PI3 WS 20/21

21 [37]



Der Listenmonade in der Listenkomprehension

- Berechnung aller Permutationen einer Liste:

- 1 Ein Element überall in eine Liste einfügen:

```
ins' :: α → [α] → [[α]]
ins' x [] = [[x]]
ins' x (y:ys) = [x:y:ys] ++ [ y:is | is ← ins' x ys ]
```

- 2 Damit Permutationen (rekursiv):

```
perms' :: [α] → [[α]]
perms' [] = [[]]
perms' (x:xs) = [ is | ps ← perms' xs, is ← ins' x ps ]
```

- Listenkomprehension \cong Listenmonade

PI3 WS 20/21

22 [37]



III. IO ist keine Magie

Implizite vs. explizite Zustände

- Wie funktioniert jetzt IO?
- Nachteil von State: Zustand ist **explizit**
 - Kann dupliziert werden
- Daher: Zustand **implizit** machen
 - Datentyp verkapseln (kein `run`)
 - Zugriff auf State nur über elementare Operationen

PI3 WS 20/21

23 [37]



PI3 WS 20/21

24 [37]



Aktionen als Zustandstransformationen

- ▶ **Idee:** Aktionen sind Transformationen auf Systemzustand S
- ▶ S beinhaltet
 - ▶ Speicher als Abbildung $A \rightarrow V$ (Adressen A , Werte V)
 - ▶ Zustand des Dateisystems
 - ▶ Zustand des Zufallsgenerators
- ▶ In Haskell: Typ `RealWorld`
 - ▶ "Virtueller" Typ, Zugriff nur über elementare Operationen
 - ▶ Entscheidend nur Reihenfolge der Aktionen

PI3 WS 20/21

25 [37]



IV. Fallbeispiel: Auswertung von Ausdrücken

PI3 WS 20/21

26 [37]



Monaden im Einsatz

- ▶ Auswertung von Ausdrücken:
Algebraische Ausdrücke:

```
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

- Auswertung ohne Effekte:

```
eval :: Expr -> Double
eval (Var _) = 0
eval (Num n) = n
eval (Plus a b) = eval a + eval b
eval (Minus a b) = eval a - eval b
eval (Times a b) = eval a * eval b
eval (Div a b) = eval a / eval b
```

- ▶ Mögliche Arten von Effekten:

- ▶ Partialität (Division durch 0)
- ▶ Zustände (für die Variablen)
- ▶ Mehrdeutigkeit

PI3 WS 20/21

27 [37]



Auswertung mit Fehlern

- ▶ Partialität durch Fehlermonade (Either):

```
eval :: Expr -> Either String Double
eval (Var x) = Left $ "No_variable_" ++ x
eval (Num n) = return n
eval (Plus a b) = do x <- eval a; y <- eval b; return $ x + y
eval (Minus a b) = do x <- eval a; y <- eval b; return $ x - y
eval (Times a b) = do x <- eval a; y <- eval b; return $ x * y
eval (Div a b) = do
  x <- eval a; y <- eval b;
  if y == 0 then Left "Division_by_zero" else Right $ x / y
```

PI3 WS 20/21

28 [37]



Auswertung mit Zustand

- ▶ Zustand durch Reader-Monade

```
import ReaderMonad
import qualified Data.Map as M
```

```
type State = M.Map String Double
```

```
eval :: Expr -> Reader State Double
eval (Var i) = get (M.! i)
eval (Num n) = return n
eval (Plus a b) = do x <- eval a; y <- eval b; return $ x + y
eval (Minus a b) = do x <- eval a; y <- eval b; return $ x - y
eval (Times a b) = do x <- eval a; y <- eval b; return $ x * y
eval (Div a b) = do x <- eval a; y <- eval b; return $ x / y
```

PI3 WS 20/21

29 [37]



Mehrdeutige Auswertung

- ▶ Dazu: Erweiterung von Expr:

```
data Expr = Var String
          | ...
          | Pick Expr Expr
```

```
eval :: Expr -> [Double]
eval (Var i) = return 0
eval (Num n) = return n
eval (Plus a b) = do x <- eval a; y <- eval b; return $ x + y
eval (Minus a b) = do x <- eval a; y <- eval b; return $ x - y
eval (Times a b) = do x <- eval a; y <- eval b; return $ x * y
eval (Div a b) = do x <- eval a; y <- eval b; return $ x / y
eval (Pick a b) = do x <- eval a; y <- eval b; [x, y]
```

PI3 WS 20/21

30 [37]



Kombination der Effekte

- ▶ Benötigt **Kombination** der Monaden.
- ▶ Monade `Res`:
 - ▶ Zustandsabhängig
 - ▶ Mehrdeutig
 - ▶ Fehlerbehaftet

```
type Exn α = Either String α
data Res σ α = Res { run :: σ -> [Exn α] }
```

- ▶ Berechnungen sind von einem Zustand abhängig, der mehrere Ergebnisse geben kann, von denen einige Fehler sein können.
- ▶ Andere Kombinationen möglich.

PI3 WS 20/21

31 [37]



Food For Thought.

Übung 11.3: Andere Kombinationen sind möglich:

- ① `data Res σ α = Res (σ -> Exn [α])`
- ② `data Res σ α = Res (Exn [σ -> α])`
- ③ `data Res σ α = Res ([σ -> Exn α])`

Was für eine Art Berechnung modellieren diese, und was ist hier der Unterschied?

Lösung:

- ① Berechnungen sind von einem Zustand abhängig, und geben entweder einen Fehler oder eine Liste von Ergebnissen;
- ② Berechnungen sind entweder fehlerhaft, oder eine Liste von Funktionen, die zu jedem Zustand ein Ergebnis liefern;
- ③ Berechnungen sind eine Liste von Funktionen, die zu jedem Zustand entweder ein Fehler oder ein Ergebnis liefern können.

Unterschied zwischen (i) und (ii)/(iii): für (i) kann es für einen Zustand mehrere Ergebnisse geben, bei (ii)/(iii) für einen Zustand nur ein Ergebnis/Fehler.

PI3 WS 20/21

32 [37]



Nachtisch

Übung 11.4: Bonusfrage

Wir hatten also als Kombinationen

- ❶ `data Res σ α = Res (σ → [Exn α])`
- ❷ `data Res σ α = Res (σ → Exn [α])`
- ❸ `data Res σ α = Res (Exn [σ → α])`
- ❹ `data Res σ α = Res [σ → Exn α]`

Sind das alle, fehlen noch welche, und wenn ja wieviele?

Lösung: Es fehlen noch

- ❺ `data Res σ α = Res [Exn (σ → α)]`
- ❻ `data Res σ α = Res (Exn (σ → [α]))`



Res: Monadeninstanz

- ▶ `Res α` ist Reader (`List (Exn α)`)

- ▶ Functor durch Komposition der `fmap`:

```
instance Functor (Res σ) where
  fmap f (Res g) = Res $ fmap (fmap f) . g
```

- ▶ Monad durch Kombination der jeweiligen Operationen `return` und `>>=`:

```
instance Monad (Res σ) where
  return a = Res (const [Right a])
  Res f >>= g = Res $ λs → do ma ← f s
                        case ma of
                          Right a → run (g a) s
                          Left e → return (Left e)
```



Res: Operationen

- ▶ Zugriff auf den Zustand:

```
get :: (σ → Exn α) → Res σ α
get f = Res $ λs → [f s]
```

- ▶ Fehler:

```
fail :: String → Res σ α
fail msg = Res $ const [Left msg]
```

- ▶ Mehrdeutige Ergebnisse:

```
join :: α → α → Res σ α
join a b = Res $ λs → [Right a, Right b]
```



Auswertung mit Allem

- ▶ Im Monaden `Res` können alle Effekte benutzt werden:

```
type State = M.Map String Double

eval :: Expr → Res State Double
eval (Var i) = get (λs → case M.lookup i s of
                        Just x → return x
                        Nothing → Left $ "No_such_variable_" ++ i)
eval (Num n) = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x + y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x - y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x * y
eval (Div a b) = do x ← eval a; y ← eval b
                  if y == 0 then fail "Division_by_zero." else return $ x / y
eval (Pick a b) = do x ← eval a; y ← eval b; join x y
```

- ▶ Systematische Kombination durch **Monadentransformer**



Zusammenfassung

- ▶ Monaden sind **Muster** für **Berechnungen** mit **Seiteneffekten**

- ▶ Beispiele:

- ▶ Zustandstransformer (`State`)
- ▶ Fehler und Ausnahmen (`Maybe`, `Either`)
- ▶ Nichtdeterminismus (`List`)

- ▶ Fallbeispiel Auswertung von Ausdrücken:

- ▶ Kombination aus Zustand, Partialität, Mehrdeutigkeit

- ▶ Grenze: Nebenläufigkeit

