Praktische Informatik 3: Funktionale Programmierung

Vorlesung 11 vom 25.01.2021: Monaden als Berechnungsmuster

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

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 $f : \alpha \to \beta$ mit Seiteneffekt in **Zustand** σ:

$$f : \alpha \times \sigma \to \beta \times \sigma$$
$$\cong$$
$$f : \alpha \to \sigma \to \beta \times \sigma$$

- Datentyp für Zustand σ: $\sigma \to \beta \times \sigma$
- Komposition: Funktionskomposition und `uncurry`

```
curry   :: ((α, β) → γ)→ α→ β→ γ
uncurry :: (α→ β→ γ)→ (α, β) → γ
```

In Haskell: Zustände **explizit**

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

```
type State σ α = σ→ (α, σ)
```

- Komposition zweier solcher Berechnungen:

```
comp :: State σ α→ (α→ State σ β)→ State σ β
comp f g = uncurry g ∘ f
```

- Trivialer Zustand:

```
lift :: α→ State σ α
lift = curry id
```

- Lifting von Funktionen:

```
map :: (α→ β)→ State σ α→ State σ β
map f g = (λ(a, s)→ (f a, s)) ∘ g
```

Zugriff auf den Zustand

- Zustand lesen:

```
get :: (σ→ α)→ State σ α
get f s = (f s, s)
```

- Zustand setzen:

```
set :: (σ→ σ)→ State σ ()
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?

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):

$$\texttt{fmap id} = \texttt{id}$$

$$\texttt{fmap f} \circ \texttt{fmap g} = \texttt{fmap (f} \circ \texttt{g)}$$

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

Monaden in Haskell

- Verkettung ($\gg=$) und Lifting (`return`):

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

$\gg=$ 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
```

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` :: $\alpha \rightarrow$ `Maybe` β ist Berechnung mit möglichem (unspezifiertem) Fehler,
  - `f` :: $\alpha \rightarrow$ `Either` ϵ α ist Berechnung mit möglichem Fehler vom Typ ϵ
  - Fehlerfreie Berechnungen werden verkettet
  - Fehler (`Nothing` oder `Left x`) werden propagiert

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

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
```

Jetzt seit ihr dran.

Übung 11.2: Komposition in der Listenmonade

Betrachten wir noch mal die Komposition in der Listenmonade:

$$a : as \gg= g = g\ a ++ (as \gg= g)$$
$$[\,] \gg= g = [\,]$$

Welche uns (hoffentlich) wohlbekannte Funktion versteckt sich dahinter?

Jetzt seit 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:

$$\texttt{concatMap} :: (\alpha \rightarrow [\beta]) \rightarrow [\alpha] \rightarrow [\beta]$$
$$\texttt{concatMap f} = \texttt{concat} \circ \texttt{map f}$$

$$(\gg\!=) == \texttt{flip concatMap}$$

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

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

Aktionen als Zustandstransformationen

- **Idee**: Aktionen sind Transformationen auf Systemzustand S
- S beinhaltet
  - Speicher als Abbildung $A \rightharpoonup V$ (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

IV. Fallbeispiel: Auswertung von Ausdrücken

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
```

Monaden im Einsatz

- Auswertung von Ausdrücken:

  Algebraische Ausdrücke:

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

  Auswertung ohne Effekte:

```haskell
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

Monaden im Einsatz

- Auswertung von Ausdrücken:

Algebraische Ausdrücke:

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

Auswertung ohne Effekte:

```haskell
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

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
```

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
```

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]
```

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.

Food For Thought.

Übung 11.3: Andere Kombinationen sind möglich:

(i) `data Res` σ α = `Res (`$\sigma \rightarrow$ `Exn [`α`])`

(ii) `data Res` σ α = `Res (Exn [`$\sigma \rightarrow$ α`])`

(iii) `data Res` σ α = `Res ([`$\sigma \rightarrow$ `Exn` α`])`

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

Food For Thought.

Übung 11.3: Andere Kombinationen sind möglich:

1. `data Res` $\sigma\ \alpha$ `= Res (`$\sigma\rightarrow$ `Exn [`α`])`
2. `data Res` $\sigma\ \alpha$ `= Res (Exn [`$\sigma\rightarrow\ \alpha$`])`
3. `data Res` $\sigma\ \alpha$ `= Res ([`$\sigma\rightarrow$ `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. Berechnung 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.

Nachtisch

Übung 11.4: Bonusfrage

Wir hatten also als Kombinationen

1. `data Res` σ α = `Res` ($\sigma \to$ `[Exn` α`]`)
2. `data Res` σ α = `Res` ($\sigma \to$ `Exn [`α`]`)
3. `data Res` σ α = `Res (Exn [`$\sigma \to \alpha$`])`
4. `data Res` σ α = `Res [`$\sigma \to$ `Exn` α`]`

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

Nachtisch

Übung 11.4: Bonusfrage

Wir hatten also als Kombinationen

1. `data Res` σ α $=$ `Res (`$\sigma\to$ `[Exn` α`])`
2. `data Res` σ α $=$ `Res (`$\sigma\to$ `Exn [`α`])`
3. `data Res` σ α $=$ `Res (Exn [`$\sigma\to$ α`])`
4. `data Res` σ α $=$ `Res [`$\sigma\to$ `Exn` α`]`

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

Lösung: Es fehlen noch

5. `data Res` σ α $=$ `Res [Exn (`$\sigma\to$ α`)]`
6. `data Res` σ α $=$ `Res (Exn (`$\sigma\to$ `[`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 "Divison␣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