Praktische Informatik 3: Funktionale Programmierung
Vorlesung 11 vom 08.01.2019: Monaden als Berechnungsmuster

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Frohes Neues Jahr!

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
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Inhalt

- Wie geht das mit IO?
- Das M-Wort
- Monaden als allgemeine Berechnungsmuster
- Fallbeispiel: Auswertung von Ausdrücken

Zustandsabhängige Berechnungen

Funktionen mit Zustand

- Idee: Seiteneffekt **explizit** machen
- Funktion $f : A \to B$ mit Seiteneffekt in **Zustand** S:

$$f : A \times S \to B \times S$$
$$\cong$$
$$f : A \to S \to B \times S$$

- Datentyp: $S \to B \times S$
- 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
```

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**

Monaden als Berechngsmuster

Eine Monade ist:

- **mathematisch**: durch Operationen und Gleichungen definiert (verallgemeinerte algebraische Theorie)
- als **Berechnungsmuster**: **verknüpfbare** Berechnungen mit einem **Ergebnis**
- in **Haskell**: durch mehrere Typklassen definierte Operationen mit **Eigenschaften**

Monaden in Haskell

- Aktion auf Funktionen:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

fmap bewahrt Identität und Komposition:

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

- Die Eigenschaften **sollten** gelten, können aber nicht überprüft werden.
  - 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 a → (a → m b) → m b
  return   :: a → m a
```

$\gg=$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

- Zustandstransformer: 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 als Monade:

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

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

  - Ähnlich mit Either

- Berechnungsmodell: **Ausnahmen** (Fehler)
  - f :: $\alpha \rightarrow$ Maybe β ist Berechnung mit möglichem Fehler
  - 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 :: α→ [β]` ist Berechnung mit **mehreren** möglichen Ergebnissen
  - Verkettung: Anwendung der folgenden Funktion auf **jedes** Ergebnis (concatMap)

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
```

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] ++ map (y :) (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

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

Fallbeispiel: Auswertung von Ausdrücken

Monaden im Einsatz

- Auswertung von Ausdrücken:

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

- Mögliche Arten von Effekten:
  - Partialität (Division durch 0)
  - Zustände (für die Variablen)
  - Mehrdeutigkeit

Monaden im Einsatz

- Auswertung von Ausdrücken:

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

- Mögliche Arten von Effekten:
  - Partialität (Division durch 0)
  - Zustände (für die Variablen)
  - Mehrdeutigkeit

- 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
```

Auswertung mit Fehlern

- Partialität durch Maybe-Monade

```
eval :: Expr → Maybe Double
eval (Var _)   = 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; if y == 0 then Nothing else Just $ 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

```
data Res σ α = Res { run :: σ→ [Maybe α] }
```

- Andere Kombinationen möglich:

```
data Res σ α = Res (σ→ Maybe [α])
```

Kombination der Effekte

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

```
data Res σ α = Res { run :: σ→ [Maybe α] }
```

- Andere Kombinationen möglich:

```
data Res σ α = Res (σ→ Maybe [α])
```

```
data Res σ α = Res (σ→ [α])
```

Kombination der Effekte

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

```
data Res σ α = Res { run :: σ→ [Maybe α] }
```

- Andere Kombinationen möglich:

```
data Res σ α = Res (σ→ Maybe [α])
```

```
data Res σ α = Res (σ→ [α])
```

```
data Res σ α = Res ([σ→ α])
```

Res: Monadeninstanz

- Functor durch Komposition der fmap:

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

- Monad ist Kombination

```
instance Monad (Res σ) where
  return a = Res (const [Just a])
  Res f >>= g = Res $ λs→ do ma← f s
                             case ma of
                               Just a→ run (g a) s
                               Nothing→ return Nothing
```

Res: Operationen

- Zugriff auf den Zustand:

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

- Fehler:

```
fail :: Res σ α
fail = Res $ const [Nothing]
```

- Mehrdeutige Ergebnisse:

```
join :: α→ α→ Res σ α
join a b = Res $ λs→ [Just a, Just 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 (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
                      if y == 0 then fail else return $ x / y
eval (Pick a b)  = do x← eval a; y← eval b; join x y
```

- Systematische Kombination durch **Monadentransformer**
  - Monade mit Platzhalter für weitere Monaden

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