Reaktive Programmierung
Vorlesung 2 vom 10.04.2019
Monaden und Monadentransformer

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

Fahrplan

Inhalt

- Monaden zusammensetzen
- Monadentransformer
- Monaden in Scala

Monaden

Beispiele für Monaden

- Zustandstransformer: Reader, Writer, State
- Fehler und Ausnahmen: Maybe, Either
- Mehrdeutige Berechnungen: List, Set

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
- 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])
```

```
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

Kombination von Monaden

Das Problem

- Monaden sind nicht **kompositional:**

```
type mn a = m (n a)
instance (Monad m, Monad n)⇒ Monad mn
```

- Warum?
  - Wie wären $\gg=$, return definiert?
- Funktoren **sind** kompositional.

Die “Lösung”

- Monadentransformer
  - Monaden mit einem “Loch” (i.e. parametrisierte Monaden)

Beispiel

- Zustandsmonadentransformer: StateMonadT

```
data StateT m s a = St { runSt :: s→ m (a, s) }
```

- Ausnahmenmonadtransformer: ExnMonadT

```
data ExnT m e a = ExnT { runEx :: m (Either e a) }
```

- Komposition:

```
type ResMonad a = StateT (ExnT Identity Error) State a
```


Probleme

- “Lifting” von Hand
- Komposition muss fallweise entschieden werden:
  - Exception und Writer kann kanonisch mit allen kombiniert werden
  - State und List nicht mit allen, oder unterschiedlich

Monadtransformer in Haskell: mtl

- Klassendeklarationen erlauben Typinferenz für automatisches Lifting
- Zustandsmonaden, Exceptions, Reader, Writer, Listen, IO
- Fallbeispiel: Interpreter für eine imperative Sprache

Zusammenfassung

- Monaden sind **Muster** für **Berechnungen** mit **Seiteneffekten**
- Beispiele:
  - Zustandstransformer
  - Fehler und Ausnahmen
  - Nichtdeterminismus
- Kombination von Monaden: **Monadentransformer**
  - Monadentransformer: parametrisierte Monaden
  - mtl-Bücherei erleichtert Kombination
  - Prinzipielle Begrenzungen
- Grenze: Nebenläufigkeit $\longrightarrow$ Nächste Vorlesung