Reactive Programmierung

Vorlesung 2 vom 09.04.2017: Monaden als Berechnungsmuster

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017

Fahrplan

- Einführung
- Monaden als Berechnungsmuster
- Nebenläufigkeit: Futures and Promises
- Aktoren I: Grundlagen
- Aktoren II: Implementation
- Bidirektionale Programmierung
- Meta-Programmierung
- Reaktive Ströme I
- Reaktive Ströme II
- Functional Reactive Programming
- Software Transactional Memory
- Eventual Consistency
- Robustheit und Entwurfsmuster
- Theorie der Nebenläufigkeit, Abschluss

Inhalt

- Monaden als allgemeine Berechnungsmuster
- Beispielmonaden, und wie geht das mit IO?
- Monaden in Scala

Monaden als allgemeine Berechnungsmuster

Berechnungsmuster

- Eine Programmiersprache hat ein grundlegendes Berechnungsmodell und darüber hinaus Seiteneffekte
- Seiteneffekte sind meist implizit (Bsp: exceptions)
- Monaden verkapseln Seiteneffekt in einem Typ mit bestimmten Operationen:
  1. Komposition von Seiteneffekten
  2. Leere Seiteneffekte
  3. Basisoperationen
- Idee: Seiteneffekt explizit machen

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 bestimmten Eigenschaften
- In Scala: ein Typ mit bestimmten Operationen

Beispiel: Funktionen mit Zustand

- 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: $S \to B \times S$
- Operationen:
  - Komposition von zustandsabhängigen Berechnungen:
$$\begin{array}{cc} f : A \times S \to B \times S & g : B \times S \to C \times S \\ \cong & \cong \\ f' : A \to S \to B \times S & g' : B \to S \to C \times S \end{array}$$
$$g' \cdot f' = (g \cdot f)'$$
  - Basisoperationen: aus dem Zustand lesen, Zustand verändern

Monaden in Haskell

Monaden in Haskell

- Monaden in Haskell als Verallgemeinerung von Aktionen

Aktionen:	Monade m:
`type IO α`	`type m α`
Komposition:	Komposition:
`(≫=) :: IO α → (α → IO β) → IO β`	`(≫=) :: m α → (α → m β) → m β`
Leere Aktion:	Leerer Seiteneffekt:
`return :: α → IO α`	`return :: α → m α`
Aktion für Funktionen:	Seiteneffekt auf Funktionen:
`fmap :: (α → β) → IO α → IO β`	`fmap :: (α → β) → m α → m β`

Beispiel für eine Konstruktorklasse.

Monadengesetze I

- Monaden müssen bestimmte Eigenschaften erfüllen.
- Für Funktionen:

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

fmap bewahrt Identität und Komposition:

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

- Folgendes gilt allgemein (für r :: f α → g α, h :: α → β):

```
fmap h ∘ r == r ∘ fmap h
```


Monadengesetze II

- Für 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
```

- Folgendes gilt allgemein (naturality von return und ≫=):

```
  fmap f ∘ return == return ∘ f
m ≫= (fmap f ∘ p) == fmap f (m ≫= p)
```

- Den syntaktischen Zucker (**do**-Notation) gibt's dann umsonst dazu.

Zustandsabhängige Berechnungen in Haskell

- Modellierung: Zustände explizit in Typ σ (polymorph über α)

```
data ST σ α = St { run :: σ → (α, σ) }
```

- Komposition zweier solcher Berechnungen:

```
f ≫= g = St $ λs → let (a, s') = run f s in run (g a) s'
```

- Leerer Seiteneffekt:

```
return a = St $ λs → (a, s)
```

- Lifting von Funktionen:

```
fmap f g = St $ λs → let (a, s1) = run g s in (f a, s1)
```


Basisoperationen: Zugriff auf den Zustand

- Zustand lesen:

```
get :: (σ → α) → ST σ α
get f = St $ λs → (f s, s)
```

- Zustand setzen:

```
set :: (σ → σ) → ST σ ()
set g = St $ λs → ((), g s)
```


Benutzung von ST: einfaches Beispiel

- Zähler als Zustand:

```
type WithCounter α = ST Int α
```

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

```
cntToL :: String → WithCounter String
cntToL [] = return ""
cntToL (x:xs)
  | isUpper x = do ys ← cntToL xs
                   set (+1)
                   return (toLower x: ys)
  | otherwise = do { ys ← cntToL xs; return (x: ys) }
```

- Hauptfunktion:

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


Implizite vs. explizite Zustände

- Nachteil von ST: Zustand ist explizit
  - Kann dupliziert werden
- Daher: Zustand implizit machen
  - Datentyp verkapseln
  - Zugriff auf Zustand nur über elementare Operationen
  - Zustand wird garantiert nicht dupliziert oder weggeworfen.

Zustandstransformer mit impliziten Zustand

- Impliziter Zustand und getypte Referenzen:

```
newtype Ref α = Ref { addr :: Integer } deriving (Eq, Ord)
type Mem α = M.Map Integer α
```

- Lesen und Schreiben als Operationen auf Data.Map
- Impliziten Zustand und Basisoperationen verkapseln:

```
newtype ST α β = ST { state :: State.ST (Mem α) β }
```

- Exportschnittstelle: state wird nicht exportiert
- runST Kombinator:

```
runST :: ST α β → β
runST s = fst (State.run (state s) M.empty)
```

- Mit dynamischen Typen können wir den Zustand monomorph machen.

Weitere Beispiele für Monaden

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

Unveränderliche Zustände: Reader

- Die Reader-Monade:

```
newtype Reader σ α = Rd { run :: σ → α }

instance Functor (Reader σ) where
  fmap f r = Rd (f. run r)

instance Monad (Reader σ) where
  return a = Rd (λs→ a)
  r >>= f  = Rd (λs→ run (f ((run r) s)) s)
```

- Berechnungsmodell: Zustand aus dem nur gelesen wird
  - Vereinfachter Zustandsmonade
  - Basisoperation: read, local
  - Es gibt auch das "Gegenstück": Writer

Fehler und Ausnahmen: Maybe

- 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
```

- Berechnungsmodell: Fehler
  - f :: α→ Maybe β ist Berechnung mit möglichem Fehler
  - Fehlerfreie Berechnungen werden verkettet
  - Fehler (Nothing) werden propagiert

Fehler und Ausnahmen: Either

- Either α als Monade:

```
data Either δ β = Left δ | Right β

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

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

- Berechnungsmodell: Ausnahmen
  - f :: α→ Either δ β ist Berechnung mit Ausnahmen vom Typ δ
  - Ausnahmefreie Berechnungen (Right a) werden verkettet
  - Ausnahmen (Left e) werden propagiert

Mehrdeutigkeit

- List als Monade:
  - Können wir so nicht hinschreiben, Syntax vordefiniert
  - Aber siehe `ListMonad.hs`

```
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)

Aktionen als Zustandstransformationen

- Idee: Aktionen sind Zustandstransformationen 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

```
type IO α = ST RealWorld α
```


Monaden in Scala

Monaden in Scala

- Seiteneffekte sind in Scala implizit
- Aber Monaden werden implizit unterstützt
- "Monadische" Notation: for

Monaden in Scala

- Für eine Monade in Scala:

```
abstract class T[A] {
  def flatMap[B](f: A⇒ T[B]): T[B]
  def map[B](f: A⇒ B): T[B]
  }
```

- Gegebenfalls noch

```
  def filter (f: A⇒ Bool): T[A]
  def foreach (f: A⇒ Unit): Unit
```


do it in Scala!

- Übersetzung von `for` mit einem Generator:

```
for (x← e1) yield r   ⟹   e1.map(x⇒ r)
```

- `for` mit mehreren Generatoren:

```
for (x1← e1; x2← e2; s) yield r
⟹
e1.flatMap(x⇒ for (y← e2; s) yield r)
```

- Wo ist das `return`? Implizit:

```
  e1.map(x⇒ r) == e1.flatMap(x⇒ return r)
```

```
        fmap f p == p >>= return ∘ f
```


Beispiel: Zustandsmonade in Scala

- Typ mit `map` und `flatMap`:

```
case class State[S,A](run: S ⇒ (A,S)) {
```

```
  def flatMap[B](f: A ⇒ State[S,B]): State[S,B] =
    State { s ⇒  val (a,s2) = run(s)
                 f(a).run(s2)
          }
```

```
  def map[B](f: A ⇒ B): State[S,B] =
    flatMap(a ⇒ State(s ⇒ (f(a),s)))
```

- Beispielprogramm: Ein Stack

Zusammenfassung

- Monaden sind Muster für Berechnungen mit Seiteneffekten
- Beispiele:
  - Zustandstransformer
  - Fehler und Ausnahmen
  - Nichtdeterminismus
- Nutzen von Monade: Seiteneffekte explizit machen, und damit Programme robuster
- Was wir ausgelassen haben: Kombination von Monaden (Monadentransformer)
- Grenze: Nebenläufigkeit ⟶ Nächste Vorlesung