Praktische Informatik 3: Funktionale Programmierung
Vorlesung 11 vom 10.01.2017: Monaden als Berechnungsmuster

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Frohes Neues Jahr!

Organisatorisches

- Fachgespräche: 1.–3. Februar (letzte Semesterwoche)
- Mündliche Prüfung: entweder in der Zeit, oder individuell zu vereinbaren.
- Prüfungen müssen bis zum 31.03.2017 (Semesterende) stattgefunden haben.

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: Interpreter für IMP

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:	Aktionen:
`type State σ α`	`type IO α`
`comp :: State σ α → (α→ State σ β)→ State σ β`	`(>>=) :: IO α → (α→ IO β) → IO β`
`lift :: α→ State σ α`	`return :: α→ IO α`
`map :: (α→ β)→ State σ α→ State σ β`	`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
```

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

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

>>= ist assoziativ und return das neutrale Element:

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

- 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

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 :: α→ 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)

IO ist keine Magie

Implizite vs. explizite Zustände

- Wie funktioniert jett 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: Die Sprache IMP

Monaden im Einsatz

- Gegeben: imperative Programmiersprache IMP
- Ein Interpreter für IMP benötigt:
  - Parser
  - Interpreter zur Auswertung

IMP — Grammatik

```
identifier ::= Char (Char | Digit)*
    number ::= Digit+ ( . Digit+ )?
      expr ::= aterm <= expr | aterm = expr | aterm
     aterm ::= term + aterm | term
      term ::= factor * term | factor / term | factor
    factor ::= identifier | number | ( expr ) | - expr
      expr ::= expr <= expr | expr = expr
             | expr + expr
             | expr * expr | expr / expr
             | identifier | number | ( expr ) | - expr
       cmd ::= identifier := expr
             | while expr { cmds }
             | if expr { cmds } (else{cmds })?
             | print expr
      cmds ::= cmd ; cmds | cmd
      decl ::= var identifier ;
      Prog ::= decl* cmds
```

Beispielprogramm: Fakultät

```
var fak;
var n;

n := 10;

fak := 1;
while 1 ≤ n {
  print fak;
  fak := fak * n;
  n := n+(−1)
  }
```

Parser

- Monadischer Kombinatorparser
  - nach Graham Hutton, Erik Meijer: *Monadic parsing in Haskell*, J. Funct. Program. **8**:4, 1998, p 437-444.
- Eingabe ist Sequenz von Eingabetoken (Char), Rückgabe ist abstrakter Syntaxbaum (AST)
- Zustand des Parsers: noch zu lesende Eingabesequenz
- Typ (generisch über Eingabetoken α und AST β):

```
data Parser α β = Parser { parse :: [α] → [(β, [α])] }
```

  - Kombination aus State-Monade (Zustand) und Listen-Monade (Nichtdeterminismus)

Parser

- Basisparser: satisfy, erkennt einzelne Token

```
satisfy :: (α→ Bool)→ Parser α α
```

- Kombinator: Sequenzierung des Monaden:

```
(>>=) :: Parser α β→ (β→ Parser α γ)→ Parser α γ
```

- Kombinator: (+++) ist Auswahl

```
(+++) :: Parser a b → Parser a b → Parser a b
```

- Darauf aufgebaut: optional, Kleene-Stern, ...

```
opt :: Parser a b → Parser a (Maybe b)

many :: Parser a b → Parser a [b]

sepby :: Eq a⇒ Parser a b → a → Parser a [b]
```

Auswertung

- Auswertung: Systemzustand und eventueller Fehler:

```
data State = State { vars :: M.Map Id Val
                   , output :: [String]
                   }
data St a = St { run :: State→ Error (a, State) }
```

- Ausführung von Kommandos:

```
exec :: Cmd → St ()
```

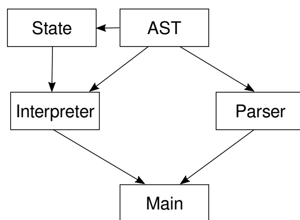
- Auswertung von Ausdrücken (keine Änderung des Systemzustands):

```
eval :: Expr→ State→ Error Val
```

Ausführung von Kommandos

```
exec :: Cmd → St ()
exec w@(While e cs) = do
  v← evaluate e
  vs ← get vars
  if isTrue v then do {execs cs; exec w} else return ()
exec (If e cs1 cs2) = do
  v← evaluate e
  if isTrue v then execs cs1 else execs cs2
exec (Assign i e) = do
  v← evaluate e
  set $ λs→ s{vars= M.insert i v (vars s)}
exec (Print e) = do
  v← evaluate e
  set $ λs→ s{output= show v : output s}
```

IMP-Interpreter: Modulstruktur



Zusammenfassung

- Monaden sind Muster für Berechnungen mit Seiteneffekten
- Beispiele:
  - Zustandstransformer (State)
  - Fehler und Ausnahmen (Maybe, Either)
  - Nichtdeterminismus (List)
- Fallbeispiel IMP:
  - Parser ist Kombination aus State und List
  - Auswertung ist Kombination aus State und Either
- Grenze: Nebenläufigkeit