Fortgeschrittene Techniken der Funktionalen Programmierung

Vorlesung vom 03.11.09: Mehr über Monaden

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Heute gibt's:

- Monaden, mehr Monaden, und noch mehr Monaden
  - Die Monaden der Standardbücherei
  - Referenz: `http://www.haskell.org/all_about_monads/html`

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
  - Einführung, Wiederholung
  - Zustände, Zustandsübergänge und `IO`
  - Reader/Writer, Nichtdeterminismus, Ausnahmen
  - Monadentransformer
- Teil II: Fortgeschrittene Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming

Die Identitätsmonade

- Die allereinfachste Monade:

```
type Id a = a

instance Monad Id where
  return a = a
  b >>= f  = f b
```

Fehlermonaden

- Erste Nährung: Maybe
- Maybe kennt nur Nothing, daher strukturierte Fehler:

```
data Either a b = Left a  | Right b
type Error  a   = Either String a

instance Monad (Either String) where
   (Right a) >>= f = f a
   (Left l)  >>= f = Left l
   return b        = Right b
```

- Nachteil: Fester Fehlertyp
- Lösung: Typklassen

Die Monade Control.Monad.Error

- Typklasse Error für Fehler

```
class Error a where
    noMsg  :: a
    strMsg :: String → a
```

- Fehlermonade parametrisiert über e:

```
class (Monad m) ⇒ MonadError e m where
    throwError :: e → m a
    catchError :: m a → (e → m a) → m a
```

```
instance MonadError (Either e) where
    throwError = Left
    (Left e) `catchError` handler = handler e
    a        `catchError` _       = a
```

Die Zustandsmonade

- Zustandsübergang als Funktion

```
newtype State s a = State {unwrap :: (s → (a,s))}

instance Monad (State s) where
    return a          = State $ λs → (a,s)
    (State g) >>= f = State (uncurry g . unwrap f)
```

- Nachteil 1: Zustandsübergang nicht-strikt (insbesondere lazy)!
  - Lösung: Strikter Zustandsübergang— Control.Monad.ST
- Nachteil 2: Zustandsübergang ≡ Funktion
  - Lösung: Typklassen

Die Monad Control.Monad.State

- Typklasse State für Zustand lesen/schreiben:

```
class MonadState m s where
    get :: m s
    put :: s → m ()
```

- Zustandsmonade parametrisiert über State:

```
instance MonadState (State s) s where
    get   = State $ λs → (s,s)
    put s = State $ λ_ → ((),s)
```

- Aber: manchmal liest man nur, manchmal schreibt man nur...

Die Lesemonade

- Intuition: Werte der Eingabe e lesen und verabeiten.
  - Lese-Teil der Zustandsmonade

```
newtype Reader e a = Reader (e → a)

instance Monad (Reader e) where
    return a          = Reader $ λe → a
    (Reader r) >>= f = Reader $
        λe → let Reader g = f (r e) in  g e
```

- Eingabe wird nicht modifiziert.
- Beispiel: Lesen aus Symboltabelle (Gegenbeispiel: Datei)

Die Monade Control.Monad.Reader

- Wie vorher: Abstraktion der Leseoperationen
- Neu: Lokaler Zustand

```
class MonadReader e m where
    ask   :: m e
    local :: (e → e) → m a → m a

instance MonadReader (Reader e) where
    ask       = Reader id
    local f c = Reader $ λe → runReader c (f e)

asks :: (MonadReader e m) ⇒ (e → a) → m a
asks sel = ask ≫= return . sel
```

Die Schreibmonade

- Produziert einen Strom von Werten
- Kein Zugriff auf geschriebene Werte möglich
- Beispiel: “Logging”

```
newtype Writer w a = Writer (a, [w])

instance Monad (Writer w) where
  return a               = Writer (a, [])
  (Writer (a,w)) >>= f = let Writer (a',w') = f a
                         in Writer (a', w++ w')
```

- Abstraktion: auch über Listen von Ausgabewerten

Die Monade Control.Monad.Writer

- Typklasse `Monoid`: Verallgemeinerte Listen

```
class (Monoid w, Monad m) ⇒ MonadWriter w m where
  pass   :: m (a, w → w) → m a
  listen :: m a → m (a, w)
  tell   :: w → m ()
```

```
instance MonadWriter (Writer w) where
  pass   (Writer ((a,f),w)) = Writer (a,f w)
  listen (Writer (a,w))     = Writer ((a,w),w)
  tell   s                  = Writer ((), s)
```

```
listens :: (MonadWriter w m) ⇒
           (w → w) → m a → m (a,w)
listens f m = do (a,w) ← m; return (a,f w)

censor :: (MonadWriter w m) ⇒
          (w → w) → m a → m a
censor f m = pass $ do a ← m; return (a,f)
```

Die Listenmonade

- Listen sind Monaden:

```
instance Monad [] where
  m >>= f   = concatMap f m
  return x = [x]
  fail s    = []
```

- Intuition: f :: a→ [b] Liste der möglichen Resultate

- Reihenfolge der Möglichkeiten relevant?

Der Monade Set

- Data.Set sind Monaden:

```
instance Monad Set where
  m >>= f   = Set.unions (Set.map f m)
  return x = Set.singleton x
  fail s    = Set.empty
```

- Nicht vordefiniert ...

Der Continuationmonade

- Auswertungskontext wird explizit modelliert.

```
newtype Cont r a =
  Cont { runCont :: ((a → r) → r) }
```

- r ist der Typ der gesamten Berechnung
- a→ r ist der momentane Kontext

```
instance Monad (Cont r) where
  return a       = Cont $ λk → k a
  (Cont c) >>= f = Cont $
     λk → c (λa → runCont (f a) k)
```

Control.Monad.Cont

- callCC: GOTO für funktionale Sprachen

```
class (Monad m) ⇒ MonadCont m where
  callCC :: ((a → m b) → m a) → m a

instance MonadCont (Cont r) where
  callCC f = Cont $
    λk → runCont (f (λa → Cont $ λ_ → k a)) k
```

- Lieber nicht benutzen!

Exkurs: Was ist eigentliche eine Monade?

- Monade: Konstrukt aus Kategorientheorie
- Monade $\cong$ (verallgemeinerter) Monoid
- Monade: gegeben durch algebraische Theorien
  - Operationen endlicher (beschränkter) Aritität
  - Gleichungen
- Beispiele: Maybe, List, Set, State, ...
- Monaden in Haskell: computational monads
  - Strukturierte Notation für Berechnungsparadigmen
  - Beispiel: Rechner mit Fehler, Nichtdeterminismus, Zustand, ...

Kombination von Monaden: Das Problem

- Gegeben zwei Monaden:

```
class Monad m1 where ...

class Monad m2 where ...
```

- Es gelten weder

```
instance Monad (m1 (m2 a))
instance Monad (m2 (m1 a))
```

- Problem: Monadengesetze gelten nicht.
- Lösung: Nächste Vorlesung

Zusammenfassung

- Monaden sind praktische Abstraktion
- Wir haben kennengelernt:
  - Fehlermonaden: Maybe, Either, MonadError
  - Zustandsmonaden: State, ST, IO
  - Lese/Schreibmonade: ReaderMonad, WriterMonad
  - Nichtdeterminusmus: [a], Data.Set
  - Explizite Sprünge: Continuation
- Wichtiges Strukturierungsmittel für funktionale Programme
- Kombination bereitet (noch) Probleme ...