Funktionale Programmierung

Monaden II

Andreas Abel und Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

24. April 2012

Monaden als Dienstleister

- Eine Monade bietet ihrer Funktion nach gewisse Dienste:
- Beispiel: Zustandsmonade
  - Lesen des Zustandes
  - Schreiben des Zustandes
- Beispiel: Fehlermonade
  - Auslösen einer Ausnahme
  - Abfangen von Ausnahmen
- Diese Funktionalität kapseln wir in Typklassen.
- Typklassen sind hier vergleichbar mit JAVA-Interfaces.

Dienstleister Zustandsmonade

Quelle *Control.Monad.State* im Paket *mtl*.

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies
-- A monad m with mutable state s.
class Monad m ⇒ MonadState s m | m → s where
  get :: m s          -- Read the state.
  put :: s → m ()     -- Write the new state.
```

Die *funktionale Abhängigkeit* $m \to s$ bedeuted, dass s durch m eindeutig bestimmt sein muss.
Beispielskunde: Zähler.

```
{-# LANGUAGE FlexibleContexts #-}
inc :: MonadState Int m ⇒ m ()
inc = do
  counter ← get
  put (counter + 1)
```

Dienstleister Zustandsmonade

Kanonische Instanz: *State*.

```
{-# LANGUAGE FlexibleInstances #-}
newtype State s a = State {runState :: s → (a, s)}
instance Monad (State s) where
  return a = State $ λs → (a, s)
  m >>= k = State $ λs →
    let (a, s') = runState m s
    in runState (k a) s'
instance MonadState s (State s) where
  get   = State $ λs → (s, s)
  put s = State $ λ_ → ((), s)
```

Gesetze der Zustandsmonade

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

Die Zustandsmonade kennt zwei (oder mehr?) Identitäten:

- Gelesenes schreiben.

$$get \gg\!= put = return\ ()$$

- Geschriebenes lesen.

$$put\ s \gg get = put\ s \gg return\ s$$

Hier verwenden wir den Sequenzoperator $\gg$, der das Ergebnis der ersten Berechnung verwirft.

```
(>>) :: Monad m ⇒ m a → m b → m b
m >> k = m >>= const k
    -- const k _ = k
```

Dienstleister Fehlermonade

Eine Fehlermonade ermöglicht das Werfen und (Ab)fangen von Ausnahmen.

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

invert :: (Fractional a, MonadError String m) ⇒ a → m a
invert x = if x ≡ 0 then throwError "division by zero"
                    else return $ 1 / x

showInverse :: (Fractional a, MonadError String m) ⇒ a → m String
showInverse x =
  do q ← invert x
     return $ show q
  `catchError` λe → return "infinity"
```

Dienstleister Fehlermonade

Kanonische Instanz: Either.

```
data Either e a = Left e | Right a
instance Monad (Either e) where
  return a       = Right a
  Left  e >>= k = Left e
  Right a >>= k = k a

instance MonadError e (Either e) where
  throwError e             = Left e
  Left e  `catchError` k = k e
  Right a `catchError` k = Right a
```

Varianten der Zustandsmonade

- Typ der Zustandsmonade: $m\ a = s \to (a, s)$
- Variante 1: $m\ a = s \to a$ "nur lesen"
  Lese-/Kontextmonade
- Variante 2: $m\ a = (a, s)$ "nur schreiben"
  Schreibe-/Ausgabemonade
- In einer Kontextmonade kann man Funktionsparameter verstecken:
  - Funktionsparameter, die selten gebraucht oder verändert werden.
  - Z.B. Programmoptionen, globale Parameter, etc.

Motivation Kontextmonade

```
import Debug.Trace (trace)   -- trace :: String → a → a
class Monad m ⇒ MonadReader r m | m → r where
  ask :: m r   -- corresponds to get of MonadState

data Options = Options {optDebug :: Bool}
debugMessage :: MonadReader Options m ⇒ String → m ()
debugMessage s = do
  r ← ask
  when (optDebug r) $ trace s $ return ()
square :: (Num a, MonadReader Options m) ⇒ a → m a
square x = do
  debugMessage $ "entering: square " ++ show x
  return $ x ∗ x
```

Lesen aus einem Verbund

- Typischer Fall: Zustand ist ein Verbund.
- Oft gebraucht: Lesen des Zustands und selektieren eines Verbundfeldes.

$$\begin{aligned}&\mathit{asks} :: \mathit{MonadReader}\ r\ m \Rightarrow (r \to a) \to m\ a\\&\mathit{asks}\ f = \mathbf{do}\ \{r \leftarrow \mathit{ask}; \mathit{return}\ (f\ r)\}\end{aligned}$$

- Verwendung von *asks fieldName* erhöht die Lesbarkeit des Codes:

$$\begin{aligned}&\mathit{traceM} :: \mathit{Monad}\ m \Rightarrow \mathit{String} \to m\ ()\\&\mathit{traceM}\ \mathit{msg} = \mathit{trace}\ \mathit{msg}\ \$\ \mathit{return}\ ()\\&\mathit{debugMessage}\ \mathit{msg} = \mathbf{do}\\&\quad \mathit{debug} \leftarrow \mathit{asks}\ \mathit{optDebug}\\&\quad \mathit{when}\ \mathit{debug}\ \$\ \mathit{traceM}\ \mathit{msg}\end{aligned}$$

Lokale Parameteränderungen

```
class Monad m ⇒ MonadReader r m | m → r where
  ask    :: m r
  local  :: (r → r) → m a → m a

noDebugging :: MonadReader Options m ⇒ m a → m a
noDebugging k = local unSetDebug k
  where unSetDebug r = r {optDebug = False}

power :: (Num a, MonadReader Options m) ⇒ Int → a → m a
power n x = do
  debugMessage $ "entering: power " ++ show n ++ "  " ++ show
  noDebugging $ loop n
  where loop 0 = return 1
        loop 1 = return x
        loop n = do y ← square =≪ loop (n `div` 2)
                    return $ if n `mod` 2 ≡ 1 then x ∗ y else y
```

Implementierung der Kontextmonade

Kanonische Instanz: *Reader*.

```
newtype Reader r a = Reader {runReader :: r → a}
instance Monad (Reader r) where
  return a = Reader $ λr → a
  m >>= k  = Reader $ λr → runReader (k $ runReader m r) r
instance MonadReader r (Reader r) where
  ask       = Reader $ λr → r
  local f k = Reader $ λr → runReader k (f r)
```

Listenmonade und Nichtdeterminismus

- Fehlermonade: Es wird 1 oder 0 Ergebnisse geliefert.
- Verallgemeinerung: 0..n Ergebnisse.
- Z.B. Datenbankanfrage liefert mehrere passende Antworten.
- Spielstrategie liefert mehrere mögliche Züge, evtl. mit Bewertung.
- 8-Damenproblem: mehrere Lösungen.
- Wechsel der Perspektive: mehrere Ergebnisse $\longleftrightarrow$ Ergebnisliste

Listenmonade

Während *Monad* sequentielle Berechnungen klassifiziert...

```
instance Monad [] where
  return a = [a]
  m >>= k = concat $ map k m
```

klassifiziert *MonadPlus* alternative Berechnungen.

```
class MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Funktionen mit mehreren Ergebnissen

Liefere alle Quadratwurzeln einer Fließkommazahl:

```
nsqrt :: Float → [Float]
nsqrt x = if (x < 0) then [] else
  if (x < 1 e − 20) then [0] else
    let q = sqrt x in [q, −q]
```

Mit monadischer Notation:

```
nsqrt :: MonadPlus m ⇒ Float → m Float
nsqrt x = do
  when (x < 0) mzero
  if (x < 1 e − 20) then return 0 else do
    let q = sqrt x
    return q `mplus` return (−q)
```

Monaden- und Listenkomprehension

Alle Ergebnisse der Rechnung $\sqrt{x} + \sqrt{x/3}$.

```
f :: MonadPlus m ⇒ Float → m Float
f x = do
  q ← nsqrt x
  r ← nsqrt (x / 3)
  return $ q + r
```

Das entspricht einer Listenkomprehension.

```
g :: Float → [Float]
g x = [q + r | q ← nsqrt x, r ← nsqrt (x / 3)]
```

Wächter in Komprehension

Bool'sche Wächter in Listenkomprehensionen....

$$g' :: Float \to [Float]$$
$$g'\ x = [q + r \mid q \leftarrow nsqrt\ x, r \leftarrow nsqrt\ (x\ /\ 3), r > 0]$$

lassen sich auch in Monadenkomprehensionen realisieren...

$$f' :: MonadPlus\ m \Rightarrow Float \to m\ Float$$
$$f'\ x = \mathbf{do}$$
$$\quad q \leftarrow nsqrt\ x$$
$$\quad r \leftarrow nsqrt\ (x\ /\ 3)$$
$$\quad guard\ (r > 0)$$
$$\quad return\ \$\ q + r$$