Einführung in die Funktionale Programmierung mit Haskell

Monaden

Steffen Jost

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

27. Juni 2013

Monaden

Was wir bisher über Monaden herausgefunden haben:

- `IO` ist eine Monaden
- Monaden sind im Modul `Control.Monad` implementiert
- Monaden bieten Operation zur Hintereinanderausführung zweier monadischer Aktionen an
- Monaden bieten Operation zur Komposition von Aktionen an; für `IO` bedeutet dies, dass der Zustand der Welt zwischen den Aktionen automatisch durchgefädelt wird.
- `Monad` ist eine Typklasse ähnlich zu `Functor`, d.h. der Typklasse gehören keine Typen, sondern Typkonstruktoren an (nicht verwechseln mit Datenkonstruktoren)
- Alle Monaden sind auch Funktoren

Typklasse Monad

Aus dem Modul `Control.Monad`:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y

  fail :: String -> m a
  fail msg = error msg
```

- `(>>=)` sprich "bind" ist Komposition mit "fädeln"
- `(>>)` ist Hintereinanderausführung, entspricht Komposition mit Wegwerfen des Zwischenergebnis
- `fail` erlaubt gesonderte Fehlerbehandlung
- Instanz `Functor` automatisch generiert

⇒ Folie 9/S.37

DO-Notation für alle Monaden

Die Do-Notation wird für jede Monade unterstützt:

```
foo = do
  x <- action1
  y <- action2
  z <- action3 x y
  action4 z
  return $ bar x y z
```

wird automatisch behandelt wie

```
foo =
  action1     >>= (\x ->
  action2     >>= (\y ->
  action3 x y >>= (\z ->
  action4     >>
  return $ bar x y z)))
```

Monaden Gesetze

Instanzen der Typklasse `Monad` **sollten** folgenden Gesetze einhalten.
Wie bei den Gesetzen der Typklasse `Functor` ist der Programmierer der Instanz für die Einhaltung dieser Gesetze zuständig!

1. **Links-Identität** `return x >>= f` macht das Gleiche wie `f x`
2. **Rechts-Identität** `m >>= return` macht das Gleiche wie `m`
3. **Assoziativität** `m >>= (\x-> f x >>= g)` macht das gleiche wie `(m >>= f) >>= g`

Alles was diesen Gesetzen genügt, ist eine **Monade**!

Maybe als Monade

```
data Maybe a = Nothing | Just a
```

Diesen Datentyp können wir zur Monade machen:

```
instance Monad Maybe where
    return x  = Just x
    (>>=) x f = case x of
                  Nothing  -> Nothing
                  (Just x) -> f x
    fail _    = Nothing
```

Monade modelliert Berechnungen, welche fehlschlagen können:

1. "Aktion" ist Berechnung, welche Wert liefert oder fehlschlägt.
2. Wenn eine Berechnung einen Wert liefert, kann damit weiter gerechnet werden.
3. Wenn eine einzelne Berechnung fehlschlägt, so schlägt auch die gesamte Berechnung fehl.

Maybe erfüllt Monaden-Gesetze

1. Links-Identität

```
return x >>= f
```

```
⇝ case (Just x) of (Just x) -> f x
```

```
⇝ f x
```

2. Rechts-Identität

```
m >>= return
```

```
⇝ case m of Nothing -> Nothing; (Just x) -> return x
```

```
⇝ case m of Nothing -> Nothing; (Just x) -> (Just x)
```

```
⇝ m
```

3. Assoziativität

```
m >>= (\x-> f x >>= g)
```

```
⇝ ...
```

```
⇝ (m >>= f) >>= g
```

Maybe-Monade: Beispiele (1)

```
> Just 9 >>= \x -> return (x*10)
Just 90
> Nothing >>= \x -> return (x*10)
Nothing
```

```
> :t sequence
sequence :: Monad m => [m a] -> m [a]

> sequence [Just 1, Just 2]
Just [1,2]
> sequence [Just 1, Just 2, Nothing, Just 4]
Nothing
```

Maybe-Monade: Beispiele (2)

```
> :t forM
forM :: Monad m => [a] -> (a -> m b) -> m [b]
> :t maybeRead
maybeRead :: Read a => String -> Maybe a

> forM ["1","2","3"] maybeRead :: Maybe [Int]
Just [1,2,3]

> forM ["1","2x","3"] maybeRead x :: Maybe [Int]
Nothing
```

Maybe-Monade: Beispiele (3)

```
mmult mx my = do
  x <- mx
  y <- my
  return $ x * y

> mmult (Just 4) (Just 5)
Just 20
> mmult Nothing (Just 5)
Nothing
> mmult (Just 4) Nothing
Nothing
```

Das Beispiel ist vielleicht etwas unsinnig, aber in der Praxis erspart die DO-Notation für Maybe-Monaden wiederholte pattern-matches mit `(Just x)` — wenn man denn nur das Ergebnis haben will, falls alle Zwischenergebnis nicht `Nothing` waren.

Listen als Monade

Auch Listen können wir als Monaden auffassen:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
  fail _ = []
```

Die Listen-Monade kann nicht-deterministische Berechnungen simulieren:

- Anstatt eines einzelnen Wertes wird mit einer Liste von Werten gleichzeitig gerechnet wird.
- Schlägt eine Berechnung fehl, so wird die Menge der möglichen Ergebniswerte für diese Berechnung leer.

Listen-Monade: Beispiele (1)

Unser Input ist entweder 3, 4 oder 5. Was kommt heraus, wenn wir darauf eine Funktion anwenden, welche entweder die Zahl negiert oder unverändert zurück gibt?

```
> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Wenn wir keinen Input bekommen, so kann auch nichts berechnet werden:

```
> [] >>= \x -> [1..5]
[]
```

Listen-Monade: Beispiele (2)

Mehrere Alternativen müssen alle miteinander kombiniert werden:

```
> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

In DO-Notation geschrieben sieht das so aus:

```
do
  n  <- [1,2]
  ch <- ['a','b']
  return (n,ch)
```

Kommt uns diese Notation nicht bekannt vor?

Listen-Monade: Beispiele (2)

List-Comprehension sind in der Tat indentisch zu DO-Notation:

```
foo1 :: [Int] -> [Int] -> [(Int,Int)]
foo1 xs ys = [(z,y)|x<-xs, x/=5, let z=x*10, y<-ys]

foo2 :: [Int] -> [Int] -> [(Int,Int)]
foo2 xs ys = do
  x <- xs
  unless (x/=5) (fail "Oops!")
  let z = x*10
  y <- ys
  return (z,y)

> foo1 [4..6] [7..9]
[(40,7),(40,8),(40,9),(60,7),(60,8),(60,9)]
> foo2 [4..6] [7..9]
[(40,7),(40,8),(40,9),(60,7),(60,8),(60,9)]
```

MonadPlus

Das Konzept der Monade kann verfeinert werden.
Listen und Maybe sind beide Instanzen der Typklasse `MonadPlus`:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Die Monade besitzt also zusätzlich die Struktur eines **Monoids**:

- Assoziative binäre Operation
  $\Rightarrow$ Assoziativität muss der Programmierer sicherstellen!
- Neutrales Element zu dieser Operation

Typklasse `Monoid` erfasst nur diese Struktur ohne Monade.

Listen-Monade: Beispiele (3)

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

Dies erlaubt Verfeinerung des Codes für List-Comprehensions:

```
foo1 xs ys = [(z,y)|x<-xs, x/=5, let z=x*10, y<-ys]
foo3 xs ys = do
  x <- xs
  guard (x/=5)
  let z = x*10
  y <- ys
  return (z,y)
```

Schlägt ein Pattern-Match mit `<-` oder `let` fehl, wird `fail` aufgerufen, dessen Nutzen nun klar ist: In der Listen-Monade muss nur die Berechnung des aktuellen Wertes abgebrochen werden!

Weitere Monadische Funktionen

`Control.Monad` bietet monadische Varianten für üblichen Funktionen:

```
replicateM :: Monad m => Int -> m a -> m [a]

mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a

foldM :: Monad m =>
              (a -> b -> m a) -> a -> [b] -> m a

zipWithM :: Monad m =>
              (a -> b -> m c) -> [a] -> [b] -> m [c]
```

Verschachtelte Monaden kann man hiermit auswickeln:

```
msum :: MonadPlus m => [m a] -> m a

join :: Monad m => m (m a) -> m a
```

Einführung in die Funktionale Programmierung mit Haskell

Monaden

Steffen Jost

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

3. Juli 2013

Zusammenfassung Monaden

- Monaden sind ein Programmierschema für zusammengesetzte Berechnungen mit Kontext oder Nebeneffekten
- Monaden separieren das Fädeln des Kontexts von der eigentlich Berechnung;
  d.h. Hintereinanderausführung mehrerer Berechnungen mit Kontext/Nebeneffekten wird vereinfacht.
- Monadenoperation müssen drei Gesetze erfüllen:
  Links-/Rechtsidentität und **Assoziativität**.
- Monaden sind keine Spracherweiterung
- GHC bietet syntaktische Unterstützung (DO-Notation)
  Bibliotheken für andere Sprachen inzwischen verfügbar

Übersicht Monaden

Gebräuchliche Monaden-Sichtweisen:

- **Fehlermonade** für Berechnungen, welche Fehler werfen können z.B. **Maybe-Monade, MonadError**
- **Nichtdeterminismus** für Berechnungen, welche mit mehreren Alternativen gleichzeitig rechnen z.B. **Listen-Monade**
- **Zustandsmonade** für Berechnungen mit veränderlichen Kontext z.B. **Control.Monad.State**
  Spezialfälle:
  - **Lesemonade** liest Zustand nur aus **Control.Monad.Reader**
  - **Schreibmonade** beschriebt Zustand nur, z.B. logging einer Berechnung, **Control.Monad.Writer**
- **I/O Monade** ermöglicht funktionale User-Interaktion.
  Kann grob als spezielle Zustandsmonade aufgefasst werden.

Zustandsmonade

Die Zustandsmonade gibt einer Berechnung einen Kontext; Berechnung darf Kontext verändern, z.B. können Variablen gesetzt und gelesen werden.

- Kann imperative Programmierweise simulieren, bzw. in die Funktionale Welt einbetten
- Typinformation gibt genau an, welche Information verfügbar ist, d.h. Seiteneffekte können genau eingegrenzt werden
- Oft für selten benutzte globale Parameter gebraucht

Beispiel

```
stackOperation = do
  push 2              --     [2]
  push 3              --   [3,2]
  push 4              -- [4,3,2]
  r <- pop            --   [3,2]; r=4
  pop                 --     [2]; return 3
```

Implementation

Auch wenn es imperative aussieht: Hinter der Monaden-Kulisse geht alles rein funktional zu!

```
newtype State a = State { getState :: Int -> (a, Int) }

instance Monad State where
  return x  = State $ \s -> (x,s)
  (State x) >>= f = State $ \s0 ->
                      let (v1,s1)   = x s0
                      in  getState (f v1) s1

runState :: Int -> State a -> a
runState i (State s) = fst $ s i

getState :: State Int
getState    = State $ \s -> (s,s)

putState :: Int -> State ()
putState s = State $ \_ -> ((),s)

inc :: State ()
inc = State $ \s -> ((),s+1)
```

```
demo :: (Int,Int,Int)
demo = runState 0 $ do
  x <- readState
  writeState $ x + 10
  y <- readState
  writeState $ x + 100
  inc
  inc
  z <- readState
  return (x,y,z)

> demo
(0,10,102)
```

Implementation

Auch wenn es imperative aussieht: Hinter der Monaden-Kulisse geht alles rein funktional zu!

```
newtype State a = State { getState :: Int -> (a, Int) }

instance Monad State where
  return x  = State $ \s -> (x,s)
  (State x) >>= f = State $ \s0 ->
                      let (v1,s1)   = x s0
                      in  getState (f v1) s1

runState :: Int -> State a -> a
runState i (State s) = fst $ s i

getState :: State Int
getState    = State $ \s -> (s,s)

putState :: Int -> State ()
putState s = State $ \_ -> ((),s)

inc :: State ()
inc = State $ \s -> ((),s+1)
```

```
demo :: (Int,Int,Int)
demo = runState 0 $ do
  x <- readState
  writeState $ x + 10
  y <- readState
  writeState $ x + 100
  inc
  inc
  z <- readState
  return (x,y,z)

> demo
  (0,10,102)
```

Control.Monad.State

Da Haskell sehr modular ist, können wir uns (wie so oft) in den Bibliotheken bedienen, anstatt alles von Hand zu programmieren:

```
type State s a = ...Instanz von MonadState...

class Monad m => MonadState s m | m -> s where
  get :: m s                          -- Zustand lesen
  put :: s -> m ()                    -- Zustand schreiben
  state :: (s -> (a, s)) -> m a       -- Funktion zu Monade

runState :: State s a -> s -> (a, s)
modify   :: MonadState s m => (s -> s) -> m ()
gets     :: MonadState s m => (s -> a) -> m a
```

- `m -> s` erzwingt, dass `s` durch `m` eindeutig bestimmt ist
- `runState` wendet Zustandsberechnung auf Anfangszustand an, liefert Endzustand und Ergebnis
- `gets` erlaubt bequemes auslesen des Zustands, z.B. wenn der Zustand ein Verbund von mehreren Werten ist (Record)

Bibliotheksunterstützung

Die Bibliotheksunterstützung vereinfacht die Verwendung der Zustandsmonade sehr:

```
import Control.Monad.State

demo :: (Int,Int,Int)
demo = fst $ (flip runState) 0 $ do
  x <- get
  put $ x + 10
  y <- get
  put $ x + 100
  inc
  inc
  z <- get
  return (x,y,z)

inc :: State Int ()
inc = modify (+1)
```

Verwendung der Module `Control.Monad.Reader` und

Control.Monad.Error

Die Fehlermonade erlaubt neben dem Werfen von Ausnahmen auch das Abfangen und die Behandlung von Fehlern.

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

Eine wichtige Instanz dieser Monade ist
`MonadError IOException IO`

Dies funktioniert ganz ähnlich wie bei der Maybe-Monade, anstatt `Maybe` verwenden wir jedoch den Datentyp `Either String`:

```
data Either a b = Left a | Right a

instance Monad (Either a) 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
```

Nichtdeterminismus

Diese Sichtweise kann manche Berechnungen vereinfachen:

Beispiel

Die Berechnung der Potenzmenge, also aller Teilmengen einer Menge, kann man mit einem nicht-deterministischen Filter erreichen:

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs

> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

- Fehlermonade kann damit auch verstanden werden: Berechnung hat 0 oder 1 Ergebnis

Beispiel

Wir modellieren mehrere Ergebniswerte mit Wahrscheinlichkeiten.

```
import Data.Ratio

newtype Prob a = Prob { getProb :: [(a,Rational)] } der

example = [("Blue",1%2),("Red",1%4),("Green",1%4)]
```

Bedeutung ist, dass Ergebnis zu $\frac{1}{2}$ =50% Blau ist, zu $\frac{1}{4}$ = 25% Grün, usw.
Wie machen wir dies zur Monade?
Wir nutzen folgendes Gesetz aus:

- `m >>= f` macht das gleiche wie `(fmap f m) f`

Liften

Ein interessantes Paar genereischer, monadischer Funktionen sind:

```
liftM :: Monad m =>   (a -> b) -> m a -> m b
ap    :: Monad m => m (a -> b) -> m a -> m b
```

`liftM` liftet eine Funktion zwischen zwei Typen zu einer Funktion zwischen Monaden von diesen Typen; `ap` erlaubt die Anwendung einer "verpackten" Funktion.
Damit sehen monadische Operationen wieder funktionaler aus:

```
> (+) `liftM` [10,20,30] `ap` [4,5,6]
[14,15,16,24,25,26,34,35,36]

> (+) `liftM` (Just 3) `ap` (Just 4)
Just 7
```

Appliktive Funktoren

Modul `Control.Applicative` erfasst Konzept von `liftM` und `ap`:

```
class Functor f => Applicative f where
  pure  :: a -> f a                    -- == return
  (<*>) :: f (a -> b) -> f a -> f b    -- application
```

Jede Monade ist auch Instanz von `Applicativ`, aber nicht umgekehrt. Aus historischen Gründen setzt `Monad` aber `Applicativ` nicht voraus.

Identität `pure id <*> v = v`

Komposition `pure (.) <*>u <*>v <*>w = u <*>(v<*>w)`

Homomorphie `pure f <*> pure x = pure (f x)`

Interchange `u <*> pure y = pure ($ y) <*> u`

Reicht `Applicative`, dann ist dies meist bessere Wahl, da es ein einfacheres Konzept ist.

Applicative: Beispiele

`Applicative` ist meist bessere Wahl als Monade, wenn möglich:

```
> (+) <$> [10,20,30] <*> [4,5,6]
[14,15,16,24,25,26,34,35,36]

> (+) <$> (Just 3) <*> (Just 4)
Just 7
```

versus

```
> (+) `liftM` [10,20,30] `ap` [4,5,6]
[14,15,16,24,25,26,34,35,36]

> (+) `liftM` (Just 3) `ap` (Just 4)
Just 7
```

Da alle Monaden ja auch applikative Funktoren sind, dürfen wir auch dort die kürzere Schreibweise verwenden, wenn das Modul `Control.Applicativ` eingebunden wurde.

Applicative: Beispiele

```
sequence :: [IO a] -> IO [a]
sequence []  = return []
sequence (c : cs) = do
  x <- c
  xs <- sequence cs
  return (x : xs)
```

kann man im applikativen Stil auch so schreiben:

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (c : cs) = (:) <$> c <*> sequence cs
```

Erinnerung: `<$>` ist Infix-Synonym für `fmap`

```
> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```