Monaden

Björn Peemöller Stefan Roggensack

Fachhochschule Wedel – University of Applied Sciences

08. Januar 2008

Agenda

Agenda

Wiederholung

- ermöglichen eine Simulation imperativer Programmierung
- viele Monaden besitzen einen Zustand
- Monaden ermöglichen IO, ohne funktionale Prinzipien zu verletzen
- Funktionen, die auf Monaden arbeiten, sind sequentialisiert
- bind (»=) Funktion zur Verknüpfung monadischer Funktionen

```
(>>=)  :: m a -> (a -> m b) -> m b
```

- `return`, um einen Wert zu „liften“

```
return :: a -> m a
```

Wiederholung

Ausgabe eines Strings

```
write :: String -> IO ()
write [] = return ()
write (c:cs) = putChar c >> write cs
```

```
write' :: String -> IO ()
write' = foldr (>>) (return ()) . map putChar
```

do-Notation

```
readn :: Int -> IO String
readn 0 = return []
readn (n + 1) = getChar >>= q
        where q c = readn n >>= r
              where r cs = return (c:cs)
```

```
readn' :: Int -> IO String
readn' 0 = return ""
readn' (n + 1) = do {
           c <- getChar;
           cs <- readn' n;
           return (c:cs)}
```

Übersetzen in »=

```
do{r} = r
do{x <- p; C; r} = p >>= q where q x = do {C; r}
```

- Die `do`-Notation ist keine Erweiterung der Sprache

```
readn' :: Int -> IO String
readn' 0 = return ""
readn' (n + 1) = do {
           c <- getChar;
           cs <- readn' n;
           return (c:cs)}
```

Agenda

IO-Beispiel

```
palin :: String -> Bool
palin xs = (ys == reverse ys)
    where ys = map toUpper (filter isLetter xs)


palindrome :: IO ()
palindrome = do
        putStr "Enter a string: "
        cs <- getLine
        if palin cs
          then putStrLn "Yes"
          else putStrLn "No"
```

IO-Beispiel 2

```
guess :: String -> IO ()
guess word = do
      putStr "rate: "
      cs <- getLine
      if cs == word
        then putStrLn "Jeah, geschafft!"
        else do
          putStrLn (compareWord word cs)
          guess word
```

Agenda

Ein einfacher Evaluierer ohne Monaden

- Datenstruktur der Exception

```
data Term                 =  Con Int
                          |  Div Term Term
                             deriving Show

answer, wrong             :: Term
answer                    =  Div (Div (Con 1972) (Con 2)) (Con 23)
wrong                     =  Div (Con 2) (Div (Con 1) (Con 0))

eval                      :: Term -> Int
eval (Con x)              =  x
eval (Div t u)            =  (eval t) `div` (eval u)
```

- Verschiedene Erweiterungen sinnvoll:
  - Behandlung von Ausnahmen
  - Erweiterung um einen Zustand (z. B. zählen der Divisionen)

Erweiterung des Evaluierers um Exceptions

- Motivation
  - Eine Division durch 0 soll nicht zu einem Programmabbruch führen
  - Stattdessen soll eine passende Meldung ausgegeben werden
- Datenstruktur

```
data Exc a                  =  Raise Exception
                            |  Return a
type Exception              =  String

instance Show a             => Show (Exc a) where
    show (Raise e)          =  "exception: " ++ e
    show (Return x)         =  "value: " ++ show x
```

Erweiterung des Evaluierers um Exceptions

- Die modifizierte Berechnung

```
evalExc                 :: Term -> Exc Int
evalExc (Con x)         =  Return x
evalExc (Div t u)       =  h (evalExc t)
                           where
                           h (Raise e)  = Raise e
                           h (Return x) = h' (evalExc u)
                             where
                             h' (Raise e') = Raise e'
                             h' (Return y)
                               | y == 0 = Raise "division by zero"
                               | y /= 0 = Return (x `div` y)
```

Erweiterung des Evaluierers um einen Zustand

- Motivation
  - Die Anzahl an durchgeführten Divisionen soll gezählt werden
  - Hier wird ein Zähler als zusätzliche Komponente eingeführt
- Datenstruktur

```
newtype St a              =  MkSt (State -> (a,State))
type State                =  Int

apply                     :: St a -> State -> (a,State)
apply (MkSt f) s          =  f s

instance Show a           => Show (St a) where
    show f                =  "value: " ++ show x ++
                             ", count: " ++ show s
                             where (x,s) = apply f 0
```

Erweiterung des Evaluierers um einen Zustand

- Die modifizierte Berechnung

```
evalSt                  :: Term -> St Int
evalSt (Con x)          =  MkSt f
                           where f s = (x,s)

evalSt (Div t u)        =  MkSt f
                           where
                           f s = (x `div` y, s'' + 1)
                                 where
                                 (x,s')  = apply (evalSt t) s
                                 (y,s'') = apply (evalSt u) s'
```

Probleme der Erweiterungen

- Die Struktur des ursprünglichen Evaluierers wird „zerstört“
- Der Grad der Wiederverwendung ist sehr gering
- Die neue Struktur ist unübersichtlich und unflexibel
- Lösung: Evaluierer mit Hilfe von Monaden entwickeln

Der einfache Evaluierer mit Monaden

- Die Datenstruktur ist identisch geblieben

```
data Term          =  Con Int
                   |  Div Term Term
                      deriving Show
```

- Die Berechnung ist geringfügig komplexer

```
eval               :: Monad m => Term -> m Int
eval (Con x)       =  return x
eval (Div t u)     =  do
                      x <- eval t
                      y <- eval u
                      return (x `div` y)
```

- Die geänderte Struktur erfordert für Variationen jedoch nur geringfügige Anpassungen

Evaluierer mit ID-Monade

- Deklaration der ID-Monade

```
newtype Id a              =  MkId a

instance Monad Id where
    return x              =  MkId x
    (MkId x) >>= q        =  q x

instance Show a => Show (Id a) where
    show (MkId x)         =  "value: " ++ show x
```

- Berechnung des Ergebnisses

```
evalId                    :: Term -> Id Int
evalId                    =  eval
```

Evaluierer mit Exception-Monade

- Die Datenstruktur ist identisch geblieben

```
data Exc a                  =  Raise Exception
                            |  Return a
type Exception              =  String
```

- Installation als Monade, Einführung einer spezifischen Funktion

```
instance Monad Exc where
    return x                =  Return x
    (Raise e) >>= q         =  Raise e
    (Return x) >>= q        =  q x

raise                       :: Exception -> Exc a
raise e                     =  Raise e
```

Evaluierer mit Exception-Monade

- Kleine Änderung an dem Basis-Evaluierer

```
evalExc                 :: Term -> Exc Int
evalExc (Con x)         =  return x
evalExc (Div t u)       =  do
                           x <- evalExc t
                           y <- evalExc u
                           if y == 0
                              then raise "division by zero"
                              else return (x `div` y)
```

- Änderung ist notwendig, da sich die Funktionalität erweitert!

Evaluierer mit State-Monade

- Die Datenstruktur ist identisch geblieben

```
newtype St a          =  MkSt(State -> (a,State))
type State            =  Int
```

- Installation als Monade, Einführung einer spezifischen Funktion

```
instance Monad St where
    return x          =  MkSt f where f s = (x,s)
    p >>= q           =  MkSt f
                         where
                         f s = apply (q x) s'
                               where (x,s') = apply p s

tick                  :: St()
tick                  =  MkSt f where f s = ((),s+1)
```

Evaluierer mit State-Monade

- Kleine Änderung an dem Basis-Evaluierer

```
evalSt                  :: Term -> St Int
evalSt (Con x)          =  return x
evalSt (Div t u)        =  do
                           x <- evalSt t
                           y <- evalSt u
                           tick
                           return (x `div` y)
```

- Änderung ist notwendig, da sich die Funktionalität erweitert!

Vorteile der Lösung mit Monaden

- Die Struktur des ursprünglichen Evaluierers bleibt erhalten
- Erweiterungen finden nur an wenigen Stellen der Basisstruktur statt
- Der Großteil der Logik ist in der Monade enthalten

Agenda

Gesetze von Monaden

- Gesetze

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b

p >>= return = p
(return e) >>= q = q e
(p >>= q) >>= r = p >>= s where s x = (q x >>= r)
```

- State Monade

```
apply                    :: St a -> State -> (a,State)
apply (MkSt f) s         =  f s
instance Monad St where
    return x             =  MkSt f where f s = (x,s)
    p >>= q              =  MkSt f
                            where
                            f s = apply (q x) s'
                                  where (x,s') = apply p s
```

Agenda

Kombination von Monaden

- Motivation
  - Die bisherigen Monaden decken nur jeweils einen Aspekt ab
  - Um Exception-Handling und einen Divisionszähler zu implementieren, müssen die Monaden kombiniert werden
- Dies ist möglich durch
  - Schreiben einer neuen Monade
    - `data ExcSt a = ...`
    - Aber: unflexibel
  - Entwicklung einer allgemeinen Regel zur Kombination von Monaden
    - *Monaden-Transformer*
    - zu Beginn aufwändiger, aber flexibel

Einführung von Typklassen

- Eine *Exception-Monade* ist eine Monade mit einer Operation zum Auslösen einer Ausnahme

```
class Monad m => ExMonad m where
    raise                :: Exception -> m a
```

- Eine *State-Monade* ist eine Monade mit einer Operation zum Erhöhen des Zählers

```
class Monad m => StMonad m where
    tick                 :: m ()
```

Einführung von Typklassen

- Eine *Show-Monade* ist eine Monade mit einer Operation zum Anzeigen eines Wertes

```
class Monad m => ShowMonad m where
    showMonad           :: m String -> String
```

- Ein *Monaden-Transformer* ist eine Operation, die aus Monaden neue Monaden erzeugt mit der Möglichkeit, Berechnungen der inneren Monade in die äußere Monade zu befördern

```
class Transformer t where
    promote             :: Monad m => m a -> t m a
```

Der neue Evaluierer

```
eval                    :: (ExMonad m, StMonad m) => Term -> m Int
eval (Con x)            =  return x
eval (Div t u)          =  do
                           x <- eval t
                           y <- eval u
                           tick
                           if y == 0
                              then raise "division by zero"
                              else return (x `div` y)
```

Der Exception-Transformer

```
newtype EXC m a          =  MkEXC (m (Exc a))

recover                  :: EXC m a -> m (Exc a)
recover (MkEXC g)        =  g

instance Monad m => Monad (EXC m) where
    return x             =  MkEXC (return (Return x))
    p >>= q              =  MkEXC (recover p >>= r)
                            where
                            r (Raise e)  = return (Raise e)
                            r (Return x) = recover (q x)

instance Monad m => ExMonad (EXC m) where
    raise e              =  MkEXC (return (Raise e))

instance Transformer EXC where
    promote g            =  MkEXC (do {x <- g; return (Return x)})
```

Der Exception-Transformer

```
instance StMonad m => StMonad (EXC m) where
    tick                = promote tick

instance ShowMonad m => ShowMonad (EXC m) where
    showMonad p         = showMonad (recover p >>= q)
                          where
                          q (Raise e)  = return ("exception: " ++ e)
                          q (Return x) = return x

instance (ShowMonad m, Show a) => Show (EXC m a) where
    show                = showMonad . mapMonad show


mapMonad                :: Monad m => (a -> b) -> m a -> m b
mapMonad f p            = do {x <- p; return (f x)}
```

Die State-Monade

```
newtype STT m a          =  MkSTT (State -> m (a, State))
type State               =  Int

apply                    :: STT m a -> State -> m (a, State)
apply (MkSTT f)          =  f

instance Monad m => Monad (STT m) where
    return x             =  MkSTT f where f s = return (x,s)
    p >>= q              =  MkSTT f
                            where
                            f s = do
                                  (x,s') <- apply p s
                                  apply (q x) s'

instance Monad m => StMonad (STT m) where
    tick                 =  MkSTT f
                            where
                            f s = return ((),s+1)
```

Die State-Monade

```
instance ShowMonad m => ShowMonad (STT m) where
    showMonad p          =  showMonad q
                            where
                            q = do
                                (x,s) <- apply p 0
                                return ("value: " ++ x ++
                                  ", count: " ++ show s)

instance (ShowMonad m, Show a) => Show (STT m a) where
    show                 =  showMonad . mapMonad show
```

Kombination der einzelnen Monaden

```
newtype Id a              =  MkId a

instance Monad Id where
    return x              =  MkId x
    (MkId x) >>= q        =  q x

instance ShowMonad Id where
    showMonad (MkId cs) =  cs

evalExSt                  :: Term -> EXC (STT Id) Int
evalExSt                  =  eval
```

Erweiterung: State-Transformer

- Installation des State-Transformers

```
instance Transformer STT where
    promote g           =  MkSTT f
                           where
                           f s = do {x <- g; return (x,s)}

instance ExMonad m => ExMonad (STT m) where
    raise e             =  promote (raise e)
```

- Kombination mit Exception-Monade

```
evalStEx                :: Term -> STT (EXC Id) Int
evalStEx                =  eval
```

Fazit

- Es gibt keine feste Regel zur Kombination von Monaden
- Monaden bieten jedoch eine Möglichkeit zur Strukturierung der Komplexität
  - Durch eine allgemeine Kombinationsvorschrift können Monaden flexibel geschachtelt werden
  - Jede Monade behandelt einen einzelnen Aspekt, kann jedoch auch andere Aspekte weiterleiten

Agenda

Zusammenfassung

- Monaden dienen zum Sequentialisieren von Operationen
- Monaden bietet zusätzliche Abstraktionsmöglichkeiten
  - Erhaltung der Basisstruktur
  - Flexible Erweiterung um zusätzliche Aspekte
- Monaden lassen sich für unterschiedlichste Bereiche einsetzen
  - IO-Operationen
  - Verwaltung von Zuständen
  - Exception-Handling
  - Tracing
  - Parsing
  - Simulation von Nichtdeterminismus
  - ...

Fragen?

Vielen Dank für die Aufmerksamkeit!

Literatur

BIRD, Richard:
Introduction to Functional Programming using Haskell.
2nd Edition.
Harlow (England), London, New York : Prentice Hall, 1998

HUTTON, Graham:
Programming in Haskell.
1st Edition.
Cambridge, New York, Melbourne : Cambridge University Press, 2007