Monaden

6. August 2009

1 Motivation

Monaden sind ein Konzept aus der Mathematik.Viele glauben, Monaden seien schwierig zu verstehen, oder dass die Benutzung von Monaden den Prinzipien der funktionalen Programmierung widerspricht. Das ist ist grundweg falsch. Monaden sind einfach eine Möglichkeit, Funktionen elegant miteinander zu kombinieren, bei denen es auf Grund ihrer Typen zu Problemen bei der Benutzung der normalen Funktionskomposition kommt.

Damit das Konzept klarer wird, machen wir zunächst ein paar Beispiele.

Debug-Ausgaben Wenn man komplexere Probleme lösen will, kommt man schnell an einen Punkt, wo das Programm, das man geschrieben hat, sich anders verhält, als man es erwartet. Hier wäre es hilfreich, wenn man neben den eigentlichen Berechnungen auch noch eine Mitschrift des Programmablaufs erstellen könnte, damit man sieht, wo der Fehler auftritt.

Da Funktionen in Haskell aber keinen globalen Zustand verändern können, in dem man eine solche Mitschrift speichern könnte und auch immer nur einen Wert zurückgeben können, muss man den Typen seiner Funktionen zwangsweise ändern. Nehmen wir an, wir haben drei Funktionen:

```
f :: a -> b
g :: b -> c

h :: a -> c
h = g . f
```

Damit wir eine Mitschrift generieren können, müssen sie neben ihren ursprünglichen Rückgabewerten noch ein Stückchen Mitschrift liefern. Wir benutzen beispielhaft Strings:

```
f' :: a -> (b, String)
g' :: b -> (c, String)
```

Soweit so gut, jetzt können wir in `f'` zum Beispiel noch einen String "f' aufgerufen" zurückgeben. Allerdings wird es umständlich `h` zu definieren, weil wir uns darum kümmern müssen, dass die Mitschriften aus `f` und `g` zusammengeführt werden. Einfache Funktionskomposition funktioniert nicht mehr, weil der Input von g nicht mehr zum Output von f passt. Stattdessen müssen wir schreiben:

```
h' x= let
        (fErg, fMit) = f x
        (gErg, gMit) = g fErg in (gErg,gMit ++ fMit)
```

In einem echten Programm tritt dieser Fall natürlich häufig auf. Es wäre sehr lästig, wenn man jedes Mal diese Konstruktion neu hinschreiben müsste. Man kann sich aber eine Funktion schreiben, die den Eingabetypen anpasst und sich um die nötige Verknüpfung der Mitschriften kümmert. Weil die Erstellung der Mitschriften einen sequentiellen Ablauf impliziert, stellen wir das Argument vom Typen `(a,String)` an die erste Stelle, damit wir bei einer Verkettung von Funktionen von links nach rechts lesen können

```
verbinde :: (a,String) -> (a -> (b,String)) -> (b,String)
verbinde (x,s) f  = let (fErg, fMit) = f x in
        (fErg,s++fMit)
```

Mit dieser Funktion können wir jetzt wieder komfortabel arbeiten

```
h'' x = f x `verbinde` g
```

Oft ist es nützlich, wenn man eine Art Identitätsfunktion hat. Die normale Identität können wir aber nicht mit `verbinde` mit unseren Funktionen verknüpfen, weil der Typ nicht passt. Wir nennen unsere neue Funktion `einheit`, weil sie das neutrale Element bezüglich `verbinde` ist (genau wie die normale Identität das neutrale Element für die normale Funktionskomposition ist). Mit `einheit` können wir auch leicht eine Funktion lift schreiben, die normale Funktionen kompatibel mit `verbinde` macht.

```
einheit x = (x,"")
lift f = einheit . f
```

Mit den drei Funktionen `verbinde`, `einheit` und `lift` können wir jetzt fast genauso komfortabel arbeiten wie zuvor, nur dass wir jetzt auch Debug-Ausgaben machen können.

```
f x = (x, "f aufgerufen.")
g x = (x, "g aufgerufen.")

h x = f `verbinde` g `verbinde` (\x -> (x,"fertig."))
```

```
Hugs> h 5
(5,"f aufgerufen.g aufgerufen.fertig.")
```

Zufallszahlen Ebenfalls problematisch ist die Benutzung von Zufallszahlen. Ihre Verwendung scheint geradezu ein Widerspruch zum Konzept der mathematischen Funktionen zu sein, die nur von ihren Eingaben abhängig sind. Tatsächlich ist es aber so, dass Computer nur in den aller seltensten Fällen *echte* Zufallszahlen benutzen. Meistens benutzt man Pseudozufallszahlen, die nach einer mathematischen Vorschrift generiert werden und nur so aussehen als wären sie zufällig. Dazu benutzt man einen Generator, der einen internen Zustand hat, der nach jeder Abfrage einer Zufallszahl verändert wird. Ein Generator im selben

Zustand liefert immer die selbe Zahl, so dass immer noch eine vollständige Abhängigkeit des Ergebnisses einer Funktion von ihren Eingaben besteht. Haskell bietet zur Erzeugung eines Zufallswerts die in `Data.Random` definierte Funktion `random :: StdGen -> (a, StdGen)`. Normale Funktionen können also Zufallszahlen benutzen, wenn sie noch einen Wert vom Typ StdGen bekommen.

```
f :: a -> b
-- wird zu

f' :: a -> StdGen -> (b,StdGen)
```

Natürlich möchte man den veränderten Zufallszahlengenerator wieder als Ergebnis zurückgeben, damit die nächste Funktion ihn benutzen kann.

Jetzt ist es aber wieder schwierig geworden, zwei Funktionen, die vormals leicht miteinander zu verknüpfen waren, aneinander zu reihen.

```
f :: a -> StdGen -> (b,StdGen)
g :: b -> StdGen -> (c,StdGen)

h :: a -> StdGen -> (c,StdGen)
h a gen= let
        (fErg, fGen) = f a gen in g fErg fGen
```

Wir wollen also wie auch zuvor eine Funktion schreiben, die die gewöhnliche Funktionskomposition für uns ersetzt. Wenn wir schon dabei sind, schreiben wir auch gleich noch eine Funktion, die es uns ermöglicht Funktionen, die deterministisch sind, in eine Kette von nichtdeterministischen Funktionen einzureihen.

```
verbinde :: (StdGen ->(a,StdGen))
        -> (a -> StdGen -> (b,StdGen))
        -> StdGen -> (b,StdGen)
verbinde g f gen = let (a,gen') = g gen in f a gen'

einheit x gen = (x,gen)

lift f = einheit.f
```

1.1 Übungen

1. Angenommen wir wollen Funktionen schreiben, die manchmal bei ihren Berechnungen scheitern. Zum Beispiel scheitert die Berechnung einer Wurzel aus einer negativen Zahl, oder das Finden eines Eintrags im Telefonbuch, wenn die gesuchte Person nicht existiert.

   Wir benutzen den `Maybe` Datentypen um das auszudrücken. Funktionen, die scheitern können, haben den Typen

   ```
   f :: a -> Maybe b
   ```

   Wenn in einer Kette von Berechnungen eine scheitert, wollen wir, dass die ganze Kette scheitert.

Schreiben Sie `verbinde` und `einheit` für Funktionen dieses Typs.

2. Angenommen wir wollen Funktionen schreiben, die mehrere Ergebnisse haben können. Zum Beispiel hat die n-te Wurzel im Komplexen n Lösungen, oder beim Durchmustern eines Graphen mit Grad n können wir n Folgezustände haben.

   Wir benutzen Listen um das auszudrücken. Funktionen, die mehrere Ergebnisse haben können, haben den Typen

   ```
   f :: a -> [b]
   ```

   In einer Kette von Berechnungen wollen wir, dass die nächste Funktion auf alle Ergebnisse der vorherigen Funktion angewendet wird. `sqrt ... sqrt` soll also alle vier vierten Wurzeln liefern.

   Schreiben Sie `verbinde` und `einheit` für Funktionen diesen Typs.

2 Monaden sind eine Typklasse

Die Beispiele aus dem vorhergehenden Abschnitt haben eine gemeinsame Struktur, die man in einer Typklasse zusammenfassen kann. Dazu muss man sich natürlich erst einen neuen Typen überlegen.

Wenn wir schon die record-Syntax eingeführt haben, hier benutzen!

```
data Debug a = D (a,String) -- für Debug-Ausgaben
data Random a = R (StdGen -> (a, StdGen)) -- für Zufallszahlen
```

Die Typklasse, die eine Funktion zum Verbinden und einem neutralen Element bezüglich dieser Funktion beinhaltet ist in Haskell bereits vordefiniert und heißt `Monad`. Neben diesen beiden Funktionen enthält sie, nicht unumstritten, auch noch eine Funktion `fail`, die bei Fehlern aufgerufen werden kann. Im nächsten Abschnitt wird klar, warum man sie braucht.

Die Typklasse sieht also so aus:

```
infixl 1  >>, >>=
class  Monad m  where
        -- "verbinde"
        (>>=):: m a -> (a -> m b) -> m b
        {- "verbinde aber ignoriere
           das Ergebnis der ersten Funktion" -}
        (>>):: m a -> m b -> m b
        -- neutrales Element bzgl. >>=
        return:: a -> m a
        fail:: String -> m a

        m >> k =  m >>= \_ -> k
        fail = error
```

Wie immer bei Typklassen erklärt die Klassendefinition zwar welchen Typ die gefordeten Funktionen haben müssen, ihre Semantik bleibt aber dem Programmierer überlassen. Damit ein Typ sich aber rechtmäßig Monade nennen darf, müssen die Funktionen folgende Gesetze erfüllen

1. Identität von rechts

   ```
   m >>= return = m
   ```

2. Identität von links

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

3. Assoziativität

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

Wir können zum Beispiel unseren Typen für Funktionen, die Zufallszahlen benutzen, zu einer Instanz dieser Klasse machen:

```
instance Monad Random where
        (R m) >>= f = R $ \gen -> (let
                (a,gen') = m gen
                (R b) = f a in b gen')
        return x = R $ \gen -> (x,gen)
```

Die Definitionen von `>>=` und `return` sind genauso wie vormals `verbinde` und `einheit`. Wir mussten uns nur noch um den Konstruktor kümmern, was die Notation ein bisschen schwieriger zu verstehen gemacht hat.

2.1 Übungen

1. Definieren Sie eine geeignete Instanz von Monad für diesen Datentypen:

   ```
   data M a = M a
   ```

   Und beweisen Sie die drei Monadengesetze.

2. Definieren Sie geeignete Instanzen von Monad für

   - `Debug a`
   - `Maybe a`
   - [a]

3. Zeigen Sie, dass die drei Monadengesetze für Ihre Instanz für Maybe gelten.

3 do-Notation

Weil Monaden so häufig vorkommen (insbesondere funktioniert die ganze Ein- und Ausgabe in Haskell über Monaden) gibt es besondere Syntax, damit man nicht mit so langen Ketten aus >>= arbeiten muss.

Eine Reihe von monadischen Anweisungen wird als do-Block geschrieben, in dem die Anweisungen von oben nach unten[1] abgearbeitet werden. Der Haskell

[1] tatsächlich hängt das von der verwendeten Monade und der Definition von >>= darin ab

Report hat eine extra Sektion darüber link. Dort wird erklärt wie man die uns bereits bekannte Syntax mit >>= in die do-Notation umwandelt.

Es gibt 4 Umwandlungsregeln

1. Einzelne Anweisungen brauchen keine Umformung. Man lässt das do einfach weg.

   ```
   do {e} = e
   ```

2. ```
   do {e; anweisungen} = e >>= _ -> do {anweisungen}
   ```

3. ```
   do {pattern <- e; anweisungen} =
           let ok pattern = do {anweisungen}
                   ok _ = fail "pattern match failure" in
                           e >>= ok
   ```

4. ```
   do { let deklarationen in anweisungen} =
           let deklarationen in do {anweisungen}
   ```

An dieser Stelle wird auch klar, warum `fail` Teil der Typklasse Monad sein muss. Dadurch, dass man pattern-matches in der do-Syntax erlaubt, kann es passieren, dass Fehler auftreten, die irgendwie behandelt werden müssen. Die Möglichkeit fail durch eine passende Funktion zu überschreiben, erlaubt es Programmierern sinnvoll mit solchen Fehlern umzugehen. Zum Beispiel kann fail in der Maybe-Monade Nothing zurückgeben und so die ganze Berechnung zum Scheitern bringen.

Vorsicht ist geboten, wenn man `if ... then ... else ...` in einem do-Block benutzt.

```
{- das folgende ist FALSCH -}
f = do
        anweisung1
        if p then
                anweisung2
                anweisung3
        else
                anweisung4
                anweisung5

{- so ist es richtig -}
f = do
        anweisung1
        if p then do
                anweisung2
                anweisung3
        else do
                anweisung4
                anweisung5
```
```

Außerdem ist es auch in do-Blöcken wichtig darauf zu achten, dass beide Zweige in der `if ... then ... else ...` Anweisung den selben Rückgabewert haben. Insbesondere kann man also den else Zweig nicht leer lassen, wenn man im then Zweig etwas zurückgibt.

Solcherlei Fehler können vermieden werden, wenn man sich stets vor Augen hält, wie do-Blöcke vom Compiler zu Anweisungen mit `>>=` umgewandelt werden.

3.1 Übungen

1. Wandeln Sie folgenden Ausdruck in einen do-Block um

   ```
   f x y= g x >>= h >> j y
   ```

2. Wandeln Sie folgenden do-Block zurück in eine Verkettung von Funktionen mit `>>=`

   ```
   f x y = do
           z <- g x
           let i = h y
           j z
   ```

4 Vordefinierte Monaden

Es gibt eine große Zahl vordefinierter Monaden. Wir stellen hier nur eine Auswahl vor. Für ausführlicheres siehe man All about Monads. Dort gibt es auch Beispiele für alle hier vorgestellten Monaden.

4.1 Writer

Die Writer Monade ist im Prinzip das Gleiche wie die am Anfang definierte Debug Monade. Allerdings ist die Writer Monade nicht nur auf Strings als Mitschrift beschränkt. Stattdessen kann man alle Typen verwenden, die eine Instanz von Monoid sind. Das ist eine Klasse, die die mathematische Struktur "Monoid" darstellt. Ein Monoid ist eine Menge von Elementen (z.B. Strings), einer assoziativen Operation mappend um zwei Elemente der Menge zu einem neuen Element zusammenzuziehen (etwa die ++ Operation) und einem neutralen Element mempty bezüglich dieser Operation (dem leeren String).

Die Typklasse für Monoide sieht folglich so aus:

```
class Monoid a where
        mempty  :: a
        -- ^ Identity of 'mappend'
        mappend :: a -> a -> a
        -- ^ An associative operation
        mconcat :: [a] -> a
        -- ^ Fold a list using the monoid.
```

```
        -- For most types, the default definition for 'mconcat' will be
        -- used, but the function is included in the class definition so
        -- that an optimized version can be provided for specific types.
        mconcat = foldr mappend mempty
```

Die Instanz für Monad für die Writer Monade ist praktisch identisch zur Instanz der Debug Monade:

```
newtype Writer w a = Writer { runWriter :: (a, w) }

instance (Monoid w) => Monad (Writer w) where
        return a = Writer (a, mempty)
        m >>= k  = Writer $ let
                (a, w)  = runWriter m
                (b, w') = runWriter (k a)
                in (b, w `mappend` w')
```

4.2 Reader

```
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
        return a = Reader $ \_ -> a
        m >>= k  = Reader $ \r -> runReader (k (runReader m r)) r
```

4.3 State

Genau wie Random

```
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
        return a = State $ \s -> (a, s)
        m >>= k  = State $ \s -> let
                (a, s') = runState m s
                in runState (k a) s'
```

4.4 List

```
instance Monad [ ] where
    (x:xs) >>= f = f x ++ (xs >>= f)
    []     >>= f = []
    return x     = [x]
    fail s       = []
```

5 Ein- und Ausgabe

Ein und Ausgabe ist problematisch in Haskell, weil Funktionen, die Ein- und Ausgabe machen keine mathematischen Funktionen sind. Vielmehr hängt ihr

Ergebnis vom "Zustand der Welt" ab. Zum Beispiel verändert eine Funktion, die den Benutzer um eine Eingabe bittet den Zustand den Benutzers.

Zum Glück haben wir mit der State-Monade schon eine Möglichkeit kennengelernt, wie man sich in Haskell um das Weiterreichen eines Zustands kümmern kann. Die IO-Monade macht eigentlich nichts anderes als die State-Monade, nur dass der Zustand der Welt weitergereicht wird. Dieses Zustandsobjekt kann nicht vom Programmierer erzeugt werden[2] sondern wird der main Funktion als versteckter Parameter vom Compiler übergeben. Das bedeutet, dass es kein `runIO` geben kann (diese Aufgabe ist der `main` Funktion vorbehalten).

Damit ist die IO-Monade, die einzige Monade, bei der etwas Hintergrundmagie passiert.

Damit wissen wir durch unsere Kenntnis der State-Monade eigentlich schon alles, was wir brauchen um Ein- und Ausgaben zu machen. Wir müssen uns lediglich ansehen, was für Funktionen uns schon vordefiniert gegeben werden.

5.1 Bildschirmausgaben

getChar, putChar, getLine, putStr, putStrLn, Buffering

5.1.1 Beispiel: Hangman

5.2 Dateien

readFile, writeFile, hGetContents

5.2.1 Beispiel: Wörter zählen?

5.3 Netzwerkkommunikation

```
import Network.Socket
import System
main = do
        leSocket <- socket AF_INET Stream defaultProtocol
        addr <- getAddrInfo Nothing (Just "time.fu-berlin.de") (Just "13")
        let leAddr = addrAddress (head addr)
        putStrLn $ show leAddr
        connect leSocket leAddr
        tmp <- recv leSocket 100
        putStrLn tmp
        sClose leSocket

{- todo: hierarchische packages -}
import Network
import IO
import System
main = do
```

[2] es sei denn er benutzt Funktionen deren Namen mit unsafe beginnen, wie unsafePerformIO

```
args <- getArgs
case args of
        (host:port:[]) -> do
                          handle <- connectTo host (PortNumber (fromIntegral $ re
                          hSetBuffering handle LineBuffering
                          putStrLn =<< hGetLine handle
                          hClose handle
        _ -> putStrLn "musst so krass host und port angeben!"
```

5.3.1 Beispiel: Webserver

5.4 Exceptions

H98 Exceptions, neue Exceptions?

5.5 Stil

http://www.haskell.org/haskellwiki/Avoiding_IO
http://www.haskell.org/haskellwiki/How_to_get_rid_of_IO
http://haskell.org/haskellwiki/Iteratee_I/O

6 Beispiel: Backtracking

http://logic.csci.unt.edu/tarau/research/PapersHTML/monadic.html