Mit Monaden die Zukunft im Blick

Joachim Breitner*

19. Februar 2016
BobKonf 2016, Berlin

Herzlich willkommen zu meinem Vortrag über MonadFix. Es geht – wie so oft – um Monaden. Ich drücke mich dabei jetzt um die Frage, was eine Monade ist, sondern zeige, was man damit machen kann: Nämliche schwierige und hässliche Programmierprobleme durch Abstraktion einfach und schön zu machen.

Sowohl das Problem als auch die Lösung entspringt dabei einer „echten“ Anwendung. Wer Kinder hat wird den Tiptoi-Stift von Ravensburger kennen: Der kann optische Codes in Büchern erkennen und darauf hin Audio-Dateien abspielen. Die Zuordnung und Logik ist dabei in einem proprietärem Dateiformat gespeichert, welches ich Reverse-Engineered habe. Ich habe dann ein Programm[1] geschrieben, was dieses Dateiformat analysieren und erzeugen kann.

Heute will ich erzählen und zeigen, wie mir Monaden, und insbesondere die etwas unbekannte Typklasse MonadFix, dabei geholfen haben, Daten in diesem Binärformat zu serialisieren..

1 Ohne Monaden

Im Folgenden bauen wir also Haskell-Code, der eine Reihe von einfachen Binärformaten erzeugen kann.

Das erste ist ganz einfach: Ein 8-Bit-Wort und zwei 16-Bit-Wörter, direkt aufeinander folgend. Das ist schnell hingeschrieben:

```
module Writer1 where

import Data.Word

type ByteString = [Word8]

writeFormat1 :: (Word8, Word16, Word16) -> ByteString
writeFormat1 (w1, w2, w3) =
    [ w1
    , fromIntegral $ w2 `div` 256
    , fromIntegral $ w2 `mod` 256
    , fromIntegral $ w3 `div` 256
    , fromIntegral $ w3 `mod` 256
    ]
```

*breitner@kit.edu, http://www.joachim-breitner.de/
[1] http://tttool.entropia.de/

Der Einfachheit halber benutze ich hier Listen von Bytes als Typ für die Binärdaten, aber der Name Bytestring legt ja schon nahe dass man eigentlich gern etwas effizienteres haben will.

Das funktioniert auch wie gewünscht:

```
*Writer1> writeFormat1 (23,42,2015)
[23,0,42,7,223]
```
GHCi

Außerdem habe ich im Modul `Parser` einen Parser vorbereitet, mit dem wir den Code testen können:

```
*Writer1> import Parser
*Writer1 Parser> Test.QuickCheck.quickCheck (\i -> parseFormat1 (writeFormat1 i) == i)
+++ OK, passed 100 tests.
```
GHCi

Was die Funktionalität angeht, sind wir also zufrieden. Aber es ist klar, dass dieser Stil nicht skalieren wird: Zu viel Code-Duplikation, zu wenig Abstraktion.

2 Mit Monaden

Also auf zu Monaden! Dies soll kein umfassendes Monaden-Tutorial werden, deswegen überspringe ich die pädagogische Herleitung, und schreibe gleich den Code, mit dem man dieses Dateiformate dann schön serialisieren lassen kann.

```
import Data.Word

type ByteString = [Word8]

newtype Write a = Write (ByteString, a)

runWriter :: Write a -> (ByteString, a)
runWriter (Write result) = result

execWriter :: Write a -> ByteString
execWriter w = let (bytes,_) = runWriter w in bytes

instance Functor Write where
    fmap f w = Write $
        let (bytes, x) = runWriter w
        in (bytes, f x)

instance Applicative Write where
    pure x = Write ([], x)
    w1 <*> w2 = Write $
        let (bytes1, f) = runWriter w1
            (bytes2, x) = runWriter w2
        in (bytes1 ++ bytes2, f x)

instance Monad Write where
    w1 >>= w2 = Write $
        let (bytes1, x1) = runWriter w1
            (bytes2, x2) = runWriter (w2 x1)
        in (bytes1 ++ bytes2, x2)
```

```Haskell
```

Seit GHC-7.10 braucht man für eine Monad-Instanz auch Functor- und Applicative-Instanzen. Ich schreibe die Implementierungen bewusst so ähnlich, dass man schön sieht worin sich fmap, <*> und >>= unterscheiden.

Der Typ Writer führt eine Abstraktion ein, und gute Abstraktionen haben eine kleine Oberfläche. Das heißt, dass wir anstreben, möglichst wenig Funktionen so zu definieren, dass wir in den Typ Write „hinen schauen“ müssen. Tatsächlich genügt dazu eine Funktion, nämlich writeWord8, die ein Byte schreibt. Schon die Funktion writeWord16, mit der wir ein 16-Bit-Wort schreiben, können wir aus writeWord8 und den Monad-Funktionen zusammensetzen:

```Haskell
writeWord8 :: Word8 -> Write ()
writeWord8 b = Write ([b], ())

writeWord16 :: Word16 -> Write ()
writeWord16 w = do
    writeWord8 (fromIntegral w1)
    writeWord8 (fromIntegral w2)
  where (w1, w2) = w `divMod` 256
```

Haskell

Damit können wir unser Dateiformat sehr schön deklarativ in **do**-Notation herausschreiben:

```Haskell
writeFormat1 :: (Word8, Word16, Word16) -> ByteString
writeFormat1 (w1, w2, w3) = execWriter $ do
    writeWord8 w1
    writeWord16 w2
    writeWord16 w3
```

Haskell

Schön! Auf zum nächsten Format. Nun sollen wir eine Liste von 16-Bit-Wörtern herausschreiben, und ihre Anzahl vorne weg. Da das ein wiederkehrendes Muster ist, macht es Sinn, sich erst einmal das als abstrakte Definition herauszuschreiben:

```Haskell
writeList :: (a -> Write ()) -> [a] -> Write ()
writeList write ws = do
    writeWord8 (fromIntegral (length ws))
    mapM_ write ws
```

Haskell

Nun ist das Format im Handumdrehen implementiert und getestet:

```Haskell
writeFormat2 :: [Word16] -> ByteString
writeFormat2 ws = execWriter $ do
    writeList writeWord16 ws
```

Haskell

```GHCi
*Writer2 Parser> Test.QuickCheck.quickCheck (\i -> parseFormat2 (writeFormat2 i) == i)
+++ OK, passed 100 tests.
```

GHCi

3 Positionslichter

Soweit so gut. Nun enhalten Binärdateien oft Querverweise: Wenn ich etwas zwei solche Listen speichern will, dann will ich nicht die erste komplett durchgehen müssen, um zu wissen, wo die zweite beginnt. Also will ich eine Art Tabelle, die z.B. am Ende der Datei stehen kann.

Das könnte ich z.B. so implementieren:

```Haskell
writeFormat3 :: ([Word16], [Word16]) -> ByteString
writeFormat3 (ws1, ws2) = execWriter $ do
    writeList writeWord16 ws1
    writeList writeWord16 ws2
    writeWord8 0
    writeWord8 $ 1 + fromIntegral (length ws1)
```

Das Problem ist: So Index-Berechnungen sind hochgradig fehleranfällig:

```GHCi
*Writer2 Parser> Test.QuickCheck.quickCheck (\i -> parseFormat3 (writeFormat3 i) == i)
*** Failed! Falsifiable (after 3 tests and 1 shrink):
([0],[0])
*Writer2 Parser> writeFormat3 ([0],[0])
[1,0,0,1,0,0,0,2]
*Writer2 Parser> parseFormat3 (writeFormat3 ([0],[0]))
([0],[])
```

Was ging dabei schief? Wir haben die Länge der ersten Liste falsch berechnet, und deswegen war alles kaputt. Richtig wäre folgende Zeile gewesen, und damit ist auch QuickCheck happy.

```Haskell
    writeWord8 $ 1 + 2 * fromIntegral (length ws1)
```

Bei solchen Probleme sollte man sich als guter Haskell-Programmierer die Frage stellen, wie man sie prinzipiell vermeiden kann, statt sie jedesmal aufs neue falsch machen zu können. Wir würden also gerne im Code fragen, welche Position in der Datei wir gerade haben, um das später verwenden zu können. Der Code sollte also so aussehen:

```Haskell
writeFormat3 :: ([Word16], [Word16]) -> ByteString
writeFormat3 (ws1, ws2) = execWriter $ do
    p1 <- getPos
    writeList writeWord16 ws1
    p2 <- getPos
    writeList writeWord16 ws2
    writeWord8 p1
    writeWord8 p2
```

Um dieses hilfreiche `getPos` implementieren zu können, müssen wir unseren Parser aufbohren, und ihm die aktuelle Position in der Datei zur Verfügung stellen. Da das den Typ des Parsers ändert, müssen wir die Instanzen und `writeWord8` anpassen. Dank der sauber gezogenen Abstraktion müssen wir allerdings allen anderen Code *nicht* anpassen:

```haskell
type ByteString = [Word8]
type Index = Word8

newtype Write a = Write (Index -> (ByteString, a))

runWriter :: Write a -> Index -> (ByteString, a)
runWriter (Write result) = result

execWriter :: Write a -> ByteString
execWriter w = let (bytes, _) = runWriter w 0 in bytes

instance Functor Write where
    fmap f w = Write $ \i ->
        let (bytes, x) = runWriter w i
        in (bytes, f x)

instance Applicative Write where
    pure x = Write $ \_ -> ([], x)
    w1 <*> w2 = Write $ \i0 ->
        let (bytes1, f) = runWriter w1 i0
            i1 = i0 + fromIntegral (length bytes1)
            (bytes2, x) = runWriter w2 i1
        in  (bytes1 ++ bytes2, f x)

instance Monad Write where
    w1 >>= w2 = Write $ \i0 ->
        let (bytes1, x) = runWriter w1 i0
            i1 = i0 + fromIntegral (length bytes1)
            (bytes2, y) = runWriter (w2 x) i1
        in  (bytes1 ++ bytes2, y)

writeWord8 :: Word8 -> Write ()
writeWord8 b =  Write $ \_ -> ([b], ())
```
Haskell

getPos ist jetzt denkbar einfach zu implementieren: Wir geben den Index zurück (und schreiben keine zusätzlichen Bytes raus):

```haskell
getPos :: Write Index
getPos = Write $ \i -> ([],i)
```
Haskell

4 Der Blick in die Zukunft

Nun hat das echte Format die etwas unschöne Eigenschaft, dass die Tabelle mit den Verweisen am Anfang der Datei steht. Das heißt, was wir eigentlich schreiben wollen, ist

```haskell
writeFormat4 :: ([Word16], [Word16]) -> ByteString
writeFormat4 (ws1, ws2) = execWriter $ do
    writeWord16 p1
```

```
    writeWord16 p2
    p1 <- getPos
    writeList writeWord16 ws1
    p2 <- getPos
    writeList writeWord16 ws2
```
Haskell

Aber das kann doch nicht funktionieren, oder? Doch, es kann! Man muss nur die Spracherweiterung RecursiveDo aktivieren, und statt **do** schreiben wir **mdo** (wobei das „m“ für ein „μ“ steht).

Das war nicht ganz alles: Diese Fähigkeit, in die Zukunft zu blicken, hat nicht jede Monade, sondern nur die, für die es eine Instanz der Typklasse MonadFix gibt. Wir importieren also Control.Monad.Fix und implementieren sie, wobei wir uns an der Struktur der anderen Instanzen orientieren, und die Variablen so zusammen stecken, dass es halt passt:

```
instance MonadFix Write where
    -- mfix :: (a -> m a) -> m a
    mfix f = Write $ \i ->
        let (bytes,x) = runWriter (f x) i
        in (bytes, x)
```
Haskell

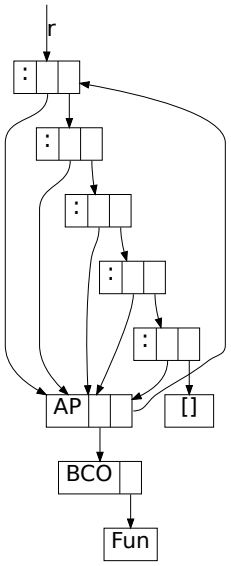
Und siehe da, die Magie funktioniert:

```
*Writer3> writeFormat4 ([23,24],[42,43])
[4,9,2,0,23,0,24,2,0,42,0,43]
*Writer3> import Parser
*Writer3 Parser> Test.QuickCheck.quickCheck (\i -> parseFormat4 (writeFormat4 i) == i)
+++ OK, passed 100 tests.
```
GHCi

5 Ein Blick unter die Abstraktion

Was passiert hier? Eigentlich nichts spannendes. Wichtig hier ist die Erkenntnis: Monaden (und andere Typklassen der Art) sind nur Abstraktionen, sie bieten an sich keine neue Funktionalität. Wenn wir also verstehen wollen, was hier passiert, können wir die Abstraktionsschicht entfernen und schauen, was darunter passiert. Ich schreibe jetzt die Funktion writeFormat4 so um, dass wir auf dieser Ebene keine Monadenoperationen mehr haben:

```
writeFormat4' :: ([Word16], [Word16]) -> ByteString
writeFormat4' (ws1, ws2) =
    let i0 = 0
        (words1, _) = runWriter (writeWord8 p1) i0
        i1 = i0 + fromIntegral (length words1)
        (words2, _) = runWriter (writeWord8 p2) i1
        i2 = i1 + fromIntegral (length words2)
        p1 = i2
        (words3, _) = runWriter (writeList writeWord16 ws1) i2
        i3 = i2 + fromIntegral (length words3)
        p2 = i3
        (words4, _) = runWriter (writeList writeWord16 ws2) i3
    in (words1 ++ words2 ++ words3 ++ words4)
```



Der Code wird so natürlich nicht schöner. Aber was sehen wir hier? Ganz normale Lazyness am Werk. Im Grunde passiert hier auch nichts anderes als in dem kleinen Codeschnippsel

```
> let r = replicate 5 (length r)
> r
[5,5,5,5,5]                                                    GHCi
```

wo wir ausnutzen, dass wir in Haskell die Länge einer Liste berechnen können, ohne dazu den Wert ihrer Elemente zu kennen. Es ist daher an sich kein Problem, wenn die Elemente der Liste sich aus der Länge der Liste berechnen. Im Hauptspeicher sieht das dann so aus, dass, solange man nicht auf einen Wert der Liste zugreift, jeder Eintrag in der Liste ein Thunk, also ein unausgewerteter Wert, ist, der selbst wieder auf die Liste verweist.

Wie sieht das jetzt bei unserem writeFormat4 aus? Schauen wir uns an, was passiert, wenn wir uns die Ausgabe erst ab dem 2. Element anzeigen lassen, also noch nicht die Berechnung der Position angestoßen haben. Wir sehen, dass die ersten Elemente noch Thunks sind. Der erste verweist auf zwei Listen mit je einem Element, die selbst Teil der Ausgabe sind: Das sind die Stückchen der Ausgabe, deren Länge wir berechnen und zusammenzählen müssen, um die Position der ersten Liste zu wissen. Der zweite Eintrag verweist zum einen auf den ersten, denn er zählt ja von der Position aus weiter, als auch auf eine Kopie der ersten serialisierten Liste.

6 Das geht nur lazy

Wir sehen, das dies alles also nicht wirklich Zauberei ist, und natürlich können wir uns damit in den Fuß schießen. Zum Beispiel funktioniert folgender Code nicht:

```
writeBroken :: ByteString
writeBroken = execWriter $ mdo
    writeWord8 pos
    if even pos then writeWord8 0 else writeWord16 0
    pos <- getPos
    return ()                                                  Haskell
```

Denn nun muss der Wert von pos berechnet werden, um den Wert von pos zu berechnen, und das hängt einfach nur:

```
Writer3> writeBroken
[^CInterrupted.                                                GHCi
```

Wie sieht das denn mit diesem Code aus, wo nicht nur der Wert des herausgeschriebenen Bytes von dem Wert aus der Zukunft abhängt, sondern auch wieviele Bytes ausgeführt werden?
(Für forM wird **import** Control.Monad benötigt.)

```
writeFormat5 :: [[Word16]] -> ByteString
writeFormat5 wss = execWriter $ mdo
    writeList writeWord8 ps
    ps <- forM wss $ \ws -> do
```

```haskell
        pos <- getPos
        writeList writeWord16 ws
        return pos
    return ()
```
Haskell

Wer glaubt dass das funktioniert? Richtig, es funktioniert. Aber damit das funktioniert muss die Monad-Instanz von Write lazy genug sein. Wenn ich den Tuple-Pattern-Matches im Bind-Operator wie folgt strict mache, dann geht es schon nicht mehr:

```haskell
instance Monad Write where
    return = pure
    w1 >>= w2 = Write $ \i0 ->
        let !(bytes1, x) = runWriter w1 i0
            i1 = i0 + fromIntegral (length bytes1)
            !(bytes2, y) = runWriter (w2 x) i1
        in  (bytes1 ++ bytes2, y)
```
Haskell

```
Prelude Parser Writer> writeFormat5 [[23,24],[42,43]]
*** Exception: «loop»
```
GHCi

Und natürlich geht das alles auch nur, weil Listen so schön lazy sind. In einem echten Programm will man aber nicht mit Listen arbeiten, sondern Bytestrings bauen, die aber zu strikt für diesen Trick sind. Daher führt der richtige Code die aktuelle Position als Zustand im Sinne der State-Monade mit.

7 Abschließene Gedanken

Ich hoffe, nach dieser kleinen Präsentation aus der Haskell-Trickkiste sind Sie alle in der richtigen Laune für die weiteren Vorträge. Außerdem sollten Sie folgende Erkenntnisse mitgenommen haben:

- Monaden sind nichts: Sie bieten keine neue Funktionalität, sondern nur eine Abstraktion.
- Monaden sind viel: Eine geeignete Abstraktion macht Programme schöner, kürzer und weniger fehleranfällig.
- Haben Sie keine Scheu davor, Monaden selbst zu implementieren.
- Mit MonadFix und **mdo** kann man lustige und sinnvolle Sachen machen.