Monaden und IO in Haskell

Sebastian Lindemeier

Magnus Giesbert

SS 2017

VORWORT

Diese Ausarbeitung für das Proseminar „Fortgeschrittene Programmierkonzepte“ wird, mit besonderem Fokus auf IO, eine Einführung in das Thema der Monaden in Haskell bieten. Hierbei wird es nicht möglich sein, alle Bereiche und existierenden Monaden zu behandeln. Darum werden die Grundkonzepte und eine kompakte Zusammenfassung zu den weiteren Konzepten der Monaden in Haskell behandelt.

Inhaltlich werden zunächst Begriffe eingeführt, bevor die Monade an sich vorgestellt wird und ihr Nutzen anhand eines Beispiels gezeigt wird. Danach wird eine simplere Schreibweise für das Arbeiten mit Monaden eingeführt welche die Lesbarkeit von geschriebenem Quelltext erhöht.
Mit diesem Wissen über die grundlegenden Funktionsweisen der Monade, wird anschließend gezeigt, wie selbst eine Monade geschrieben werden kann und welche Änderungen seit der Version GHCI 7.10 existieren. Nachdem die Monade erklärt ist, wird das Thema IO in Haskell erklärt. Dafür wird zunächst eine besondere Monade eingeführt und anschließend betrachtet welche Funktionen es für IO bereits gibt und welche vom Programmierer selbst erstellt werden können.

1 DIE MONADE

1.1 NEUE BEGRIFFE IN HASKELL

Zusätzlich zu den bereits bekannten Symbolen und Aufrufen gibt es in Haskell noch *`bind`* (>>=), *`then`* (>>) und *`return`*, welche im Folgenden erläutert werden.

Das Symbol `bind (>>=)` übernimmt die Funktion einer Pipe, die aus der Bash-Sprache bekannt ist. Mit `bind` lassen sich Aktionen verketten und die Ausgabe der einen Funktion in die Nächste übertragen. So würde `plusZwei(-4) >>= plusZwei` Null ergeben, da wir auf -4 zweimal Zwei addieren. Das `bind` wertet standardmäßig immer von links nach rechts aus und nur der erste Aufruf der Funktion bekommt einen Parameter, da der Zweite seinen Parameter erst durch das `bind` zugewiesen bekommt. Die genaue Definition von `bind` lautet `m a -> (a -> m b) -> m b`, wobei `m` ein Objekt darstellt und `a` beziehungsweise `b` die zugehörigen Datentypen. Das `bind` bildet also das Objekt `m` mit Datentyp `a` auf das Objekt `m` mit Datentyp `b` ab, sodass der Datentyp `a` zu dem Datentyp `b` transferiert wird.

Das Symbol `then (>>)` ist `bind` sehr ähnlich, der Unterschied liegt in der Übertragung des Ergebnisses. Beim `then` wird dieses im Gegensatz zum `bind` nicht in die nächste Funktion übertragen. So können verschiedene Funktionen nacheinander ausgeführt werden, ohne Parameter auszutauschen. Wird das vorherige Beispiel als `plusZwei(-4) >> plusZwei` geschrieben, wirft der Compiler einen Fehler aus, da das zweite `plusZwei` keinen Parameter hat.

Mit dem veränderten Code `plusZwei(-4) >> plusZwei(-4)` wertet Haskell den Aufruf zu -2 aus, da auf -4 nur einmal Zwei addiert wird. Das Symbol `then` bildet also das Objekt `m` mit Datentyp `a` auf das Objekt `m` mit Datentyp `b` ab, ohne dass der Datentyp `a` zum Datentyp `b` transferiert wird.

Der Aufruf `return` ist aus anderen Programmiersprachen bereits bekannt und hat in Haskell eine ähnliche Funktion. Es gibt den Wert zurück wie in der Funktion angegeben. Dementsprechend ordnet es jedem Parameter `a` ein entsprechendes Objekt `m` mit Parameter `a` zu.
Diese Objekte können sowohl Datenstrukturen, als auch Funktionen sein.

Damit ergibt sich für die Funktion:

```
liste :: a -> [a]
liste x = return x
```
würde `return` nun `[x]` zurückgeben.

Wichtig ist, dass `return` niemals den anfänglichen Wert verändert, sondern nur „neu“ verpackt.

1.2 Einführung in die Monaden

Die gerade eingeführten Begriffe sind recht praktisch, um beispielsweise mehrere Funktionen nacheinander aufzurufen. Sie scheinen aber nicht modifizierbar zu sein. Dieses Problem lässt sich mit Hilfe einer Konstruktion, die *Monade* genannt wird, lösen. Monaden sind nichts anderes als eine Containerklasse der beiden Symbole und dem Aufruf aus Kapitel 1.1, auf Basis einer Datenstruktur. Daher können diese Werte mithilfe einer Monade geändert werden, wenn eine passende Datenstruktur vorhanden ist. Das simpelste Beispiel um dies zu erklären ist die Maybe – Monade, welche schon in Haskell existiert. Sie wird dazu genutzt Fehler abzufangen, das Programm dabei aber nicht zu unterbrechen.

Die Implementierung der Datenstruktur ist im Folgenden aufgeführt:

```
data Maybe a = Nothing | Just a deriving (Show).
```

Nothing steht hierbei für den auftretenden Fehler und Just für einen gültigen Wert.

Eine solche Implementierung ist zum Beispiel bei der Division nützlich. Wird in Haskell eine Zahl x durch Null dividiert, wobei $x \neq 0$, ist das Ergebnis *Infinity.* Wird jedoch Null durch Null dividiert, ist das Ergebnis *NaN* (kurz für Not a Number). Soll nun ein Dividieren durch Null generell verboten werden, böte sich `Nothing` an, um diesen Fehler zu simulieren. Außerdem kann mit Monaden recht platzsparend und vereinheitlichend gearbeitet werden, was bei Fehlerwartung und Leserlichkeit wichtig ist.

Als Beispiel wird nun ein Programm genommen, das einen Term darstellt und mit diesem umgehen kann [GI16]. Dieses Programm wird auf die Fähigkeit zu dividieren beschränkt. Zunächst wird die Implementierung ohne Monade betrachtet. Dafür wird `data Term = Con Float | Div Term Term` als Datenstruktur des Terms verwendet, wobei `Con` für eine Konstante steht und `Div` das mathematische Zeichen für die Division ersetzt.

Des Weiteren wird `data Value a = Result a deriving (Show)` verwendet, um das Ergebnis des Terms auszugeben.

Die einfachste Implementierung einer Methode, die diese Terme auswerten kann, ist:

```
divide1 :: Term -> Float
divide1 (Con x) = x
divide1 (Div x y) = Result(x/y)
                    where Result x = divide x
                          Result y = divide y
```

In diesem Programmtext werden aber noch keine Fehler abgefangen. Zum Abfangen möglicher Fehler wird die Datenstruktur Value durch Maybe ersetzen und eine neue `divide`-Funktion geschrieben, die den Fehler `Nothing` so auswertet, dass, wenn er einmal auftritt, er zum Schluss ausgegeben wird.

```
divide2 :: Term -> Maybe Float
divide2 (Con x) = Just x
divide2 (Div t u) = case divide2 t of
                    Nothing -> Nothing
                    Just x -> case divide2 u of
                              Nothing -> Nothing
                              Just y -> if y == 0 then Nothing
                                                  else Just (x/y)
```

Trotz dieser kleinen funktionellen Erweiterung hat sich der Code drastisch verändert und ist sehr unleserlich geworden. Das Ergebnis stellt allerdings eine korrekte Lösung der gestellten Aufgabenstellung dar. Diese soll weiterhin leserlicher gestaltet werden. Außerdem ist bekannt, dass wenn der Fehler Nothing einmal vorkommt, er beibehalten wird, sodass bei der Auswertung ein Fehler auftritt.
Zur Verbesserung der Leserlichkeit können Monaden eingesetzt werden. Monaden zu instanziieren geht mit dem folgenden Schema:

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

Konkret für diesen Fall angewendet folgt damit:

```
instance Monad Maybe where
  return = Just
  Nothing >>= q = Nothing
  (Just x) >>= q = q x
```

Im Kopf dieser Anweisung wird Haskell mitgeteilt, dass eine neue Instanz einer Monade angelegt wird und diese auf der Datenstruktur von Maybe beruht. Danach wird der Aufruf `return` modifiziert. In diesem Fall soll mit `return` das Ergebnis zurückgeben werden. Daher wird `Just` verwendet, da dies einem fehlerfreien Term entspricht.

Nun wird das Symbol `bind` für die beiden möglichen Fälle angepasst. Diese beiden Fälle sind zum einen, dass ein Fehler aufgetreten ist, zum anderen, dass kein Fehler aufgetreten ist. Wird `Nothing` mit einer weiteren Funktion `q` verbunden (engl. bind), dann soll als Ergebnis `Nothing` auftreten. Wird `Just` mit einer Funktion verbunden, dann soll diese Funktion den Wert von `Just` weiterverwenden. Somit ist auch das Problem gelöst, wenn `Just` mit `Nothing` verbunden wird, da `Nothing` sich im nächsten Schritt wieder durchsetzt.

Nun muss die Monade in das Programm integriert werden.

```
divide2 :: Term -> Maybe Float
divide2 (Con x) = return x
divide2 (Div t u) = divide2 t >>= \x
                                -> divide2 u >>= \y
                                             -> if y /= 0 then return(x/y)
                                                          else Nothing
```

In diesem Fall verbinden wir das Ergebnis von `divide2 t` mit der Variable `x` und danach das Ergebnis von `divide2 u` mit `y`. Diese Werte werden dann weiter übergeben, sodass mit if getestet werden kann, ob `y` ungleich Null ist. Wenn dies gegeben sein sollte, dann geben wir das Ergebnis mit `return` zurück. Ist dem allerdings nicht so, wird `Nothing` zurückgegeben. Das Symbol `Nothing` setzt sich im Folgenden überall durch, sodass `Nothing` das Endergebnis darstellt. Somit ist eine Monade in das Programm integriert, ohne Funktionalität zu beeinträchtigen. Allerdings ist die Lesbarkeit nicht signifikant gestiegen, dafür aber besser zu vereinheitlichen.

Im Anschluss können die drei weiteren Grundrechenarten hinzugefügt werden. Die Methoden dafür würden sich nur darin unterscheiden, dass der if – Teil modifiziert werden muss. Dabei wird im `return` jeweils die Rechenart eingesetzt. Außerdem ist zu beachten, dass diese hier beschriebene Methode eine Monade zu schreiben nur vor der Version GHC 7.10.* gilt. Seit GHC 7.10.* brauchen Monaden ausführlichere Grundlagen, die programmiert werden müssen. Diese werden in Kapitel 1.4 betrachtet. Das Ziel, die Lesbarkeit zu verbessern, kann allerdings auf diesem Weg nicht erreicht werden. Hierzu sind weitere Maßnahmen erforderlich.

1.3 Die do-Notation

Zur Verbesserung der Lesbarkeit des oben aufgeführten Programmcodes wird eine alternative Syntax verwendet, welche als *do – Notation* bezeichnet wird. Bisher werden die mit Monaden arbeitenden Funktionen mit `bind` oder `then` verbunden. Dieses Vorgehen kann durch die Verwendung der neuen Syntax umgangen werden.

Mithilfe der do-Notation lässt sich im Vergleich zum obigen Beispiel der Code wie folgend vereinfachen:

```
do x <- divide2 t
   y <- divide2 u
   if y /= 0 then return(x/y)
             else Nothing
```

Es kann beobachtet werden, dass die alternative Schreibweise für `bind` und `then` die Lesbarkeit des Programmcodes deutlich erhöht, da die Länge der Zuweisungen deutlich reduziert werden kann.

Für die do-Notation gibt es Transferregeln:

1. Der Anfang eines solchen Befehlblocks ist stets do gefolgt von den Befehlen.
2. Anstelle von `x >>= y >>= z` schreibt man
   ```
   do a <- x
      b <- y a
      z b.
   ```
3. Anstelle von `x >>= \a -> y b >>= \c -> z a c` schreibt man
   ```
   do a <- x
      c <- y b
      z a c.
   ```
   Wobei `z` hier eine Funktion ist, die `a` und `c` als Eingabe fordert.
4. Anstelle von `x >> y >> z` schreibt man einfach
   ```
   do x
      y
      z.
   ```

Einer der Vorteile dieser Notation ist, dass Variablen leichter definiert und verwendet werden können. Die do-Notation erscheint sehr sequentiell aufgebaut und ist von der Richtung der Zuordnungspfeile her stark an imperative Programmierung angelehnt. Dies bedingt, dass die do-Notation bei Haskell Einsteigern sehr beliebt ist.

Bei all der Einfachheit sollte man aber auch ein paar Dinge beachten:

1. Redundanz vermeiden: Es lohnt sich nicht für eine einzige Aktion einen do Block zu verwenden.
   Anstelle von:
   ```
   foo = do x <- bar
            return x
   ```
   verwendet man besser: `foo = bar.`
2. Durch die Verwendung der do-Notation versteckt man die dahintersteckende Funktionalität. Dies mag eigentlich eher nebensächlich sein, aber vor allem, wenn man den Umgang mit Monaden üben will, sollte man auf diese Notation zuerst verzichten.
   (Es gibt sogar Programmierer, die die do-Notation als schädlich ansehen) [LE16]

Unter der Verwendung der do-Notation ergibt sich für das Programm der folgende Code:

```
divide2 :: Term -> Maybe Float
divide2 (Con x) = return x
divide2 (Div t u) = do x <- divide2 t
                       y <- divide2 u
                       if y /= 0 then return(x/y)
                                 else Nothing.
```

Sofort fällt auf, dass eine deutliche Erhöhung der Lesbarkeit erreicht wird. Somit ist das Ziel aus Kapitel 1.2 erfüllt. Auch auf die anderen Rechenarten kann diese Schreibweise angewendet werden und somit ist eine einheitliche und gut lesbare Schreibweise für alle Rechenarten gefunden, indem Monaden benutzt werden.

Abschließend ist noch darauf hinzuweisen, dass in Haskell Tabs nicht ohne Grund zu Warnungen des Compilers führen. Wird beim Untereinanderschreiben der Aktionen in der do – Notation ein Tab benutzt, kann es unter Umständen dazu führen, dass der do-Block vor dem Tab Zeichen endet und somit der Rest der Aktionen nicht mehr in dem do-Block steht.

1.4 Erstellen einer eigenen Monade

Nachfolgend wird die Maybe-Monade aus Kapitel 1.2 auf den Stand ab GHC 7.10.* erweitert und somit auch für diese Versionen nutzbar gemacht. Bisher gab es nur die Klasse Monade, die wie in Kapitel 1.2 initiiert wurde. Diese hat mit den neuen Versionen eine Erweiterung erhalten und ist zu einem Spezialfall eines *Applicatives* geworden. Somit ergibt sich für die Implementierung der Monade nun folgende Klassenbezeichnung.

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

Anschließend wird die, durch die neue Klassenbezeichnung herbeigeführte, Erstellung der Monaden-Typklasse der Maybe-Monade betrachtet.

```
instance Monad Maybe where
  return = pure
  Nothing >>= q = Nothing
  (Just x) >>= q = q x
```

Auffallend ist, dass dieser Code dem aus den älteren GHC Versionen, bis auf das `pure`, exakt gleicht. Der einzige sonstige Unterschied ist, dass in der Klassenbezeichnung nun die Klasse Applicative zu finden ist. Daraus resultiert, dass die Applicative Klasse auch vom Programmierer erstellt werden muss.

Die Klassenbezeichnung eines solchen Applcative ist im Folgenden aufgeführt.

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

Der Aufruf `pure` ist, tauscht man das `f` gegen das `m` aus, deckungsgleich zu dem Aufruf `return` aus der Klassenbezeichnung der Monade. Auch bei dem neuen Zeichen `(<*>)` ist eine Ähnlichkeit mit dem Symbol `bind` zu erkennen. In diesem Fall sind es allerdings keine Objekte mehr, sondern Funktionen mit bestimmten Datentypen. Das Zeichen `(<*>)` bildet also die applicative Funktion `f` mit Datentyp `a` auf die applicative Funktion `f` mit Datentyp `b` ab, sodass der Datentyp `a` zu dem Datentyp `b` transferiert wird. Aus dieser Klassenbezeichnung ergibt sich nun der nachfolgende Code für das Applivative der Maybe-Monade.

```
instance Applicative Maybe where
  pure = Just
  Just f <*> Just x = Just (f x)
  _ <*> _ = Nothing
```

Zu sehen ist, dass `pure` nun die gleiche Zuweisung hat wie `return` in der Monade der alten Version. Aus diesem Grund wird auch `return` mit `pure` innerhalb der neueren Maybe-Monade gleichgesetzt. Die Zuweisung des Zeichens `(<*>)` weicht von der Vorlage der Monade ab. Auch dort wird in zwei Fällen unterschieden. Einerseits, wenn auf beiden Seiten mit `Just` gültige Begriffe stehen, andererseits, wenn es nicht der Fall ist. Tritt der Fall ein, dass auf der einen Seite eine gültige Funktion und auf der anderen Seite ein gültiger Wert ist, wird der Wert mit der Funktion ausgewertet. Ist dies nicht der Fall muss auf einer der beiden Seiten `Nothing` vorhanden sein. Es wurde vorher festgelegt, dass, sollte an einer Stelle innerhalb der Monade ein `Nothing` vorhanden sein, die Funktion dann nach `Nothing` auswertet. Dies ist mit der zweiten Definition des Zeichens `(<*>)` gegeben.

Als letztes ist in der Bezeichnung des Applicative die Klasse *Functor* genannt. Auch diese muss vom Programmierer selber erstellt werden. Die Klassenbezeichnung eines Functors ist wie folgt.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Aus der Vorlesung Programmierung ist die Map-Funktion in Haskell bekannt. Diese ist nun die Funktion `fmap` in der Klasse des Functors und sagt aus, dass jeder Functor seinen eigenen Datentypen auf sich selber abbilden (engl. map) kann. Somit ist die Functor-Klasse der Maybe-Monade wie folgt zu deklarieren.

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

Wie zu sehen ist wird, wie bei der Monade und dem Applicative zuvor auch, in zwei Fälle unterteilt. Wird `Just` mit einem beliebigen Wert auf `f` abgebildet wird ein Ergebnis ausgegeben, indem die Funktion `f` auf `x` angewendet wird. Das daraus resultierende Ergebnis wird dann mittels `Just` in den Maybe-Datentyp verpackt. Soll allerdings `Nothing` auf die Funktion `f` abgebildet werden, ist das Ergebnis der Operation stets `Nothing`. Beide Functionen entsprechen somit der aufgestellten Definition der Maybe-Monade.

Werden die drei Instanzen von Functor, Applicative und Monade in ein Haskell Programm übertragen, ist die Erstellung der Monade erfolgreich.

Diese Vorgehensweise ist von der GHC Version 7.10.* an bei jeder Monade notwendig. Bei Bedarf kann nun die Monade um weitere Funktionalitäten erweitert werden. Diese Erweiterung kann dann zum Beispiel wie folgt aussehen.

```
instance Monad Maybe where
  return = pure
  Nothing >>= q = Nothing
  (Just x) >>= q = q x
  Nothing >> g = Nothing
  (Just x) >> g = g
```

In diesem Beispiel ist zu sehen, dass auch `then` in dieser Monade angepasst wird. So wird der linke Wert von `then`, der sonst vernachlässigt worden wäre, nun bei dem Aufruf mit `Nothing` mit einbezogen, sodass sich `Nothing` fortsetzt.

2 Input und Output in Haskell

2.1 IO Monade

Im vorangegangenen Kapitel wurden der Aufbau und die Funktionsweise von Monaden beschrieben. Dieses Vorwissen wird nun auf eine spezielle Monade angewandt. Diese ist die Monade, die in Haskell den Input beziehungsweise den Output regelt. Benannt wird diese im Folgenden als IO-Monade.

Im Gegensatz zur Maybe-Monade sind bei der IO Monade die entsprechenden Werte, wie die Datenstruktur oder der Aufbau der Monade mit `bind`, `then` und `return`, nicht bekannt. Daraus folgt, dass diese Monade nicht vom Nutzer geändert werden kann. Wäre dies möglich, könnte die referenzielle Transparenz zerstört werden. Die Verwendung dieser IO-Monade ist dennoch möglich, da bekannt ist, dass die IO-Monade Aktionen anstatt Werte verwendet. Dies geschieht dadurch, dass Aktionen Seiteneffekte haben können, die nicht im Programm selber stattfinden, sodass die referenzielle Transparenz erhalten bleibt.

Diese Aktionen sind im Einzelnen das Einlesen und das Ausgeben. Der Befehl der zum Ausgeben verwendet wird ist stets IO gefolgt von einem leeren Tupel, also `IO()`. Anders ist der Befehl zum Einlesen immer spezifiziert mit IO und einem beliebigen Datentypen a, also `IO a`. Der Datentyp `a` muss, sobald er in einem Programm vorkommt, immer genau genannt werden. Er darf somit nie das unspezifische `a` bleiben, sondern muss einen Wert annehmen.

2.2 Basis IO-Funktionen

Haskell bietet für die „Basis“ IO Operationen, wie Zeichen in der Konsole ausgeben oder einlesen, bereits vordefinierte Funktionen. Diese sind im Folgenden nachzulesen.

1. `putChar :: Char -> IO()`

   Gibt einen Char in der Konsole aus.

2. `getChar :: IO Char`

   Liest einen Char aus der Konsole ein und gibt ihn zurück.

3. `putStrLn :: String -> IO()`

   Gibt einen String mit Absatz in der Konsole aus.

4. `getLine :: IO String`

   Liest einen String aus der Konsole ein und gibt ihn zurück.

5. `print :: Show a => a -> IO()`

   Gibt einen Wert a in der Konsole aus.

6. `readFile :: FilePath -> IO String`

   Liest eine Datei und gibt ihren Inhalt als String zurück.

7. `writeFile :: FilePath -> String -> IO()`

   Schreibt einen String in eine Datei.

2.3 Erstellen eigener Programme mit IO

Um die Anwendung der IO-Monade zu erläutern, wird im Folgenden ein spezifisches Problem unter Zuhilfenahme dieser Monade-gelöst. Das zu lösende Problem ist einen String aus einer vom Benutzer festzulegenden Datei einzulesen und alle Fälle von 'A' zu zählen. Um nur alle Vorkommnisse des Buchstaben 'A' zu zählen, genügt eine Methode die wie folgt aufgebaut ist.

```
countA :: [Char]  -> Int
countA [] = 0
countA ('A':xs) = 1 + countA xs
countA (x:xs) = 0 + countA xs
```

Da IO Aufrufe nicht mehr zu dem „sauberen“ Code in Haskell gehören, darf während des Programms der IO Teil nie verlassen werden. Um dies zu erreichen muss der IO-Teil des Programms in der main-Methode auftreten. Diese wird wie nachfolgend zu sehen ist implementiert.

```
main :: IO()
main = do
       putStrLn "Enter Filepath:"
       pfad ← getLine
       input ← readFile pfad
       let out = countA input
       putStrLn (show out)
```

Diese gibt mit `putStrLn` zunächst aus, dass von dem User ein Dateipfad erwünscht ist und liest ihn dann in die Variable `pfad` ein. Dann wird in der Variablen `input` der String, aus der in `pfad` gespeicherten Datei, eingelesen. Nun enthält `input` den String, der mit der `countA`-Methode bearbeitet werden soll. Mithilfe von `let`, einer für Monaden ideal geeigneten Form von `where`, wird `out` auf den Wert des Ergebnisses der `countA`–Methode gesetzt. Da `out` nun vom Typ Int ist, kann es nicht ohne weiteres als String ausgeben werden. Dazu wird Funktion `show`, definiert als `show :: a -> String`, benutzt, um aus dem Int einen String zu formen, der mit `putStrLn` ausgeben werden kann.

FAZIT

Monaden sind eine mächtige Möglichkeit um Funktionalitäten wie IO zu realisieren, ohne dabei die referentielle Transparenz zu verletzen. Ihr Einsatzgebiet erstreckt sich dabei von simpler Fehlererkennung mittels Maybe bis hin zur Erkennung vom ‚State' einer Funktion. Durch ihr Modulares erscheinen ermöglichen sie außerdem, dass bei einer nötigen Änderung ihrer Funktionsweise nur Teile der Monade geändert werden müssen und nicht mehr Teile des restlichen Quelltextes.

Wenn also mit Haskell gearbeitet wird, führt der Weg früher oder später zur Verwendung von Monaden, sei es der Einfachheit halber oder weil IO im Programm benötigt wird.
Mithilfe der vorgestellten do-Notation und der gezeigten Beispiele sollte es nun einfacher sein, Monaden zu verwenden oder selber zu Schreiben.

Für weitergehende Literatur zu Monaden und Haskell sei auf [GO08, FA00, LI11] verwiesen.

QUELLEN

1. [GI16] Giesl, J.:Funktionale Programmierung. Skript zur Vorlesung, RWTH Aachen, 2016.

2. [GO08] Goerzen, J.;O'Sullivan, B.; Stewart, D.: Real world Haskell. O'Reilly Media, Sebastopol (Kalifornien) November 2008. URL:http://book.realworldhaskell.org/ (letzter Abruf: 21.6.2017)

3. [FA00] Fasel, J.; Hudak, P.; Peterson, J.: A Gentle Introduction to Haskell. Stand: 06.2000. URL: https://www.haskell.org/tutorial/index.html (letzter Abruf: 21.6.2017)

4. [LE16] Lemming:Do Notation considered harmful. Stand: 3.2016. URL: https://wiki.haskell.org/Do_notation_considered_harmful (letzter Abruf: 21.6.2017)

5. [LI11] Lipovača, M.: Learn you a Haskell for greater good. No Starch Press, San Francisco, April 2011. URL: http://learnyouahaskell.com/ (letzter Abruf: 21.6.2017)