Monaden und IO in Haskell

David Ehrenberg Philip Niederprüm

15. Mai 2019

betreut von Marcel Hark
Rheinisch Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet Informatik 2 (Programmiersprachen und Verifikation)
Prof. Dr. Jürgen Giesl

1 Einleitung

Haskell ist eine rein funktionale Sprache. Dies bedeutet unter anderem, dass keine Seiteneffekte erlaubt sind. Eine Funktion hat also immer das gleiche Ergebnis, wenn sie mit den gleichen Parametern aufgerufen wurde. Das Ergebnis ist zudem unabhängig vom Zeitpunkt der Ausführung oder davon, welche Funktion vorher ausgeführt wurde. Dies hat zum Beispiel den Vorteil, dass sich sämtliche äquivalente Teilausdrücke eines Programms durch die gleiche Variable ersetzten lassen. Das ermöglicht wiederum, dass das Programm leicht umstrukturierbar ist. Gegebenenfalls ist auch eine Validierung des Programms einfacher. Dieses Konzept trägt den Namen `referentielle Transparenz`. Das Problem das sich aus dieser ergibt ist, dass man in der Realität auf gewisse Seiteneffekte angewiesen ist. So könnte man zum Beispiel einen Zufallsgenerator gebrauchen, oder auf programmexterne Ressourcen zugreifen wollen. In der folgenden Ausarbeitung befassen wir uns insbesondere damit, wie in Haskell Ein- und Ausgabe so realisiert sind, dass die `referentielle Transparenz` erhalten bleibt.

2 Die IO-Monade

Der Typkonstruktor `IO` ist eine der vordefinierten Monaden in Haskell. Was genau eine Monade bedeutet, werden wir im nächsten Kapitel erläutern. `IO` dient dazu, Ein- und Ausgaben zu verarbeiten. Hierbei sind die jeweiligen Werte einer Instanz nicht direkt zugänglich, sondern ausschließlich über Funktionen der Klasse `IO` zu bearbeiten. Ein Datentyp von `IO` repräsentiert hier eine Aktion, zum Beispiel eine Eingabe vom Typ `a` der in der Aktion `IO a` gespeichert wird. Diese Aktion kann weiterhin nur von Funktionen bearbeitet werden, die dessen Wert nicht entkapseln. Das bedeutet, dass keine Funktion `f :: IO a -> b` existiert, die zur Berechnung des Typs `b` den Wert des Typs `a` heranzieht.

Dadurch ist die referentielle Transparenz gesichert. Alle Aktionen über `IO a` die keinen Input verwalten, haben den Typ `IO ()`. `()` ist in Haskell das leere Tupel. `IO ()` ist also die leere Aktion die keinen Wert kapselt. Ein Beispiel hierfür ist die Funktion putChar:

```
putChar :: Char -> IO () -- Ausgabeaktion als Resultat
```

2.1 Grundlegende vordefinierte Funktionen von IO

Interaktion in der Konsole:

```
putStr :: String -> IO() --String ausgeben
putStrLn :: String -> IO() --Ausgabe mit Zeilenumbruch
getLine :: IO String -- String einlesen
getChar :: IO Char -- Char einlesen
```

Auch Textdateien lassen sich über die Klasse `IO` verwalten:

```
writeFile :: FilePath -> String -> IO() -- erstellt Datei ggf.
doesFileExist :: FilePath -> IO Bool
readFile :: FilePath -> IO String
removeFile :: FilePath -> IO () -- weiteres Bsp. für IO ()
```

Dies ist nur ein kleiner Ausschnitt der vordefinierten Funktionen. Weitere Funktionen lassen sich zum Beispiel in der Online Suchmaschine Hoogle finden, welche die Haskell API durchsucht.

2.2 Operation in IO

Als Basis für weitere Programme implementieren wir zunächst eine Funktion, die Fragen an den Nutzer stellt und eine Antwort erwartet. Dazu verknüpfen wir putStrLn und getLine. Die Funktion bildet demnach von einem String für die Frage, auf eine `IO`-Aktion, die die Antwort des Typs String kapselt, ab.

```
ask :: String -> IO String
```

Zwei Aktionen lassen sich mit dem Operator `>>` (genannt `then`) wie folgt verknüpfen.

```
ask  s = putStrLn s >> getLine
```

`Then` führt dabei die erste Aktion aus, verwirft dessen Ergebnis und führt anschließend die zweite Aktion aus.

```
(>>)  :: IO a -> IO b -> IO b
```

In unserem Fall unterschlägt `then` der Funktion getLine das Ergebnis von putStrLn, welches die Rückgabeaktion `IO ()` ist. Da getLine keine Parameter erwartet, ist dies ein gewünschter Effekt. Angenommen wir wollen nun aber den Benutzer etwas fragen lassen - unser Programm `ask` sozusagen umdrehen -, könnte man auf ein Problem stoßen. Wir nennen das neue Programm `listen`, welches folgende Abbildungsvorschrift hat:

```
1    listen :: IO String -> IO () --Eingabe zu Ausgabe
```

Wenn wir in `listen` kontextbezogene Antworten geben wollen, hängt die Antwort von der Frage ab. Somit können wir nicht mehr den `then` Operator nutzen, da dieser Aktionen zwar hintereinander ausführt, allerdings ein Zwischenergebnis dabei verwirft. Um dies zu vermeiden, benötigt man den Operator `>>=` (genannt `bind`) mit folgender Abbildungsvorschrift:

```
1    (>>=) :: IO a -> (a -> IO b) -> IO b
```

`Bind` führt die erste Aktion aus, übergibt dessen Resultat an die zweite Aktion und führt diese anschließend aus. (Betrachtet man die Aktionsbeschreibung `b = x >>= f` müsste x also eine Aktion sein, und `f` eine Funktion, die auf eine Aktion abbildet, zu der `b` ausgewertet wird.) Mit folgender Implementierung erwartet `listen` eine Eingabe und gibt diese aus.

```
1    listen = getLine >>= putStrLn
```

Wenn wir nun zurück an die Funktion `ask` denken, fällt auf, dass wir diese auch mit `bind` implementieren können. Dazu nutzen wir eine λ-Funktion mit einer Wildcard, da die Rückgabe von `putStrLn` hier wie erwähnt nicht weiter von Interesse ist.

```
1    ask  s = putStrLn s >> getLine
2    ask2 s = putStrLn s >>= \_ -> getLine
```

`Then` ist sozusagen syntaktischer Zucker, denn `bind` ist mächtiger als `then` und allgemein gilt `p >> q = p >>= \_ -> q`.
Neben `bind` ist in der IO Monade auch die Funktion `return` definiert.

```
1    return :: a -> IO a
```

`Return` hebt Werte auf IO Typen an, kapselt ihre Werte also in Aktionen. Man könnte annehmen, `return` funktioniert wie in imperativen Sprachen.

```
GHCI Prelude> return 'a'
     'a'
```

Es scheint, als würde return den Char 'a' zurückgeben.

```
GHCI Prelude> 'a'
     'a'
```

Wenn man sich jedoch diese Eingabe anschaut sieht man, dass in `Prelude` auf alle Unterklassen von `Show` die Funktion `show()` aufgerufen wird. `IO` ist keine Unterklasse von `Show`, jedoch ist `show()` überladen und lässt sich somit ebenfalls über `show()` ausgeben. `Return 'a'` ist somit vom Typ `IO Char` und nicht vom Typ `Char`.
Als weitere Version der Funktion `ask` implementieren wir nun ein Programm, das eine Frage an den Nutzer stellt, ob dieser mit einem Prozess fortfahren möchte.

```
1 askv2 :: String -> IO Bool
2 askv2 s = putStrLn (s ++ " Fortfahren?[j|n]")
```

```
3              >> getChar >>= \x -> case x of
4                                                  'j' -> return True
5                                                  _   -> return False
```

Als Parameter erwartet `askv2` dabei einen String, welcher daraufhin ausgegeben wird. Gibt der Benutzer anschließend den Char 'j' ein, liefert `askv2` den in IO gekapselten Boolean `True`. In allen anderen Fällen liefert `askv2` den gekapselten Wert von `False`. Wie in Zeile 4 zu sehen, kann man innerhalb von Funktionen eine Aktion entkapseln, um das Verarbeiten von Eingaben zu ermöglichen. Jedoch muss gewährleistet sein, dass die Funktion zum Beispiel durch ein `return` zu einem `IO` Typ ausgewertet wird. Da die Schreibweise mit `then`, `bind` und λ-Ausdrücken sich bei komplexeren Programmen immer tiefer verschachtelt und somit unübersichtlich wird, gibt es hierfür eine alternative Notation.

2.3 Die do-Notation

Die do-Notation ist ebenso mächtig, wie die bisher kennengelernte Schreibweise. Die Funktion `ask` sieht beispielsweise folgendermaßen aus:

```
1      ask3 s = do {   putStrLn s;
2                      getLine;    }
```

Dieser Stil weist erneut hohe Ähnlichkeit zu imperativen Sprachen auf. Die Semikola sind in einem {}-Block notwendig, um eine Aktion abzuschließen. Aktionen können demnach auch in einer Zeile gestaffelt werden. Ein Effekt dieser Schreibweise ist es, die Sequentialität der Aktionen zu wahren. So müssen wir uns nicht mehr darum kümmern, ob durch Lazy-Evaluation putStrLn vor getLine aufgerufen wird. Als weiter vereinfachte Darstellung der do-Notation kann man die geschweiften Klammern weglassen.

```
1      ask4 s = do putStrLn s; getLine;
```

Auch die Semikola können weggelassen werden, dabei muss eine Aktion allerdings durch einen Zeilenumbruch abgeschlossen sein. Auch unser Beispiel `listen` lässt sich in neuer Art schreiben.

```
1      listen2 = do x <- getLine
2                   putStrLn x
```

Damit ein Programm kompiliert, muss darauf geachtet werden, dass Aktionen in einer Zeile mindestens nach dem `do` stehen. Wir geben das Resultat von getLine nun nicht mehr durch `bind` weiter, sondern speichern die (ungekapselte) Eingabe in x, welches anschließend an putStrLn übergeben wird. Die Rückgabe von putStrLn x ist zum Schluss wieder die leere Aktion `IO ()`.
Wir können drei Regeln festhalten (E' ist hierbei eine Ableitung des Ausdrucks E und p darf eine leere Aktion sein):

```
E := do {p}           = p
E := do {p; E'}       = p >> do {E'}
E := do {x <- p; E'} = p >>= \x -> do {E'}
```

Betrachten wir nun ein Programm, über das der Benutzer Dateien erstellen kann. Den Namen der Datei, als auch dessen Inhalt, lesen wir dabei über unsere Funktion `ask` ein.

```
createFile = ask "Gebe einen Dateipfad ein:"
             >>= \path -> ask "Gebe Inhalt ein:"
             >>= \content -> writeFile path content
```

Im nächsten Schritt vergleichen wir dies mit einer äquivalenten Implementierung mit der do-Notation. Hier kann pro Zeile ein neuer Funktionsaufruf stehen. Dies ist zwar etwas länger, macht uns die Struktur der Funktion jedoch deutlich.

```
createFile2 = do    putStrLn "Gebe einen Dateipfad ein:"
                    path <- getLine
                    putStrLn "Gebe Inhalt ein:"
                    content <- getLine
                    writeFile path content
```

In einer do-Notation lassen sich auch Variablen über let-Statements definieren. Dies hat die Form: `do {...; let var = x; ...}`

2.4 unsafePerformIO

Zu Beginn haben wir begründet, weshalb keine Funktion existieren darf, welche Aktionen entkapselt. Haskell gilt als funktionale Sprache, obwohl genau genommen, eine solche „illegale“ Funktion existiert. Sie heißt `unsafePerformIO` und muss mit `import System.IO.Unsafe(unsafePerformIO)` eingebunden werden. Sie behandelt eine IO Aktion wie einen normalen Wert.

```
nichtNachMachen :: String -- Beispiel einer Funktion, welche die
    referentielle Transparenz verletzt
nichtNachMachen = let x = unsafePerformIO getChar in ['a'..x]
```

Darüber hinaus existieren weitere Funktionen, wie `unsafeInterleaveIO`, welche IO Aktionen mit Lazy Evaluation auswerten lässt, was in Sonderfällen wie dem Auslesen großer Dateien angewendet werden kann. Weiter gehen wir jedoch nicht auf diese Funktionen einen, da Sie schnell zu Fehlern führen können, nicht zum normalen Programmieren in Haskell gedacht sind und außerdem die referentielle Transparenz verletzen.

3 Programmieren mit Monaden

Im Folgenden werden nun zunächst die formale Definition von Monaden, sowie weitere vordefinierte Monaden behandelt. Es ist zudem auch möglich selber Monaden im Rahmen der in diesem Abschnitt aufgezeigten Einschränkungen zu definieren. Außerdem wird die Nützlichkeit von Monaden beim modularen Programmieren anhand von Beispielen gezeigt.

3.1 Monaden-Definition

Monaden bilden in Haskell eine Typklasse. Diese Klasse sieht wie folgt aus:

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

Sie ist in Haskell vordefiniert. Die Funktionen `return` und `>>=` sind hierbei Funktionen der Klasse `Monad`, welche je nach Monade unterschiedlich definiert werden können, allerdings den oben angegebenen Typ haben müssen. Die Funktion `return` gibt von einem Wert den im Datentypkonstruktor `m` „gekapselten“ Wert zurück. Folglich muss `m` ein Typkonstruktor sein. Wie bereits in der IO-Monade gesehen, kann mit `>>=` eine Funktion den gekapselten Wert des Typs a benutzen, um eine vom Wert abhängigen anderen gekapselten Wert des Typs b zu berechnen. Die Pünktchen symbolisieren hierbei, dass es noch weitere Funktionen gibt, die definiert werden können. Diese werden allerdings, wenn man sie nicht gesondert implementiert, von Haskell vordefiniert. Ein Beispiel [1] dafür ist `>>`, welches standardmäßig folgendermaßen implementiert ist (wie auch bei der IO-Monade):

```
f >> g = f >>= \_ -> g
```

Des Weiteren müssen bestimmte Gesetze beim Programmieren mit Monaden beachtet werden. Die Begründung der Gesetze liegt in der ursprünglichen mathematischen Definition von Monaden und wird im Rahmen dieser Ausarbeitung nicht weiter behandelt (vgl. [1,2]). Beim Programmieren eigener Monaden müssen diese beachtet werden und bei allen vordefinierten Monaden sind diese natürlich erfüllt. Zum Verständnis der folgenden Beispiele sind sie allerdings nicht notwendig.

3.2 Die Value-Monade

Zunächst wollen wir nun eine sehr simple Monade selbst implementieren. Der dazugehörige Datentypkonstruktor ist wie folgt definiert:

```
data Value a = Result a
```

Die dazugehörige Monade definieren wir wie folgt. (Eigentlich müsste man jetzt auch die Monadengesetze für die Monade überprüfen):

```
instance Monad Value where
return x = Result x
Result x >>= f = f x
```

Die Funktion `return` kapselt hierbei einfach den bloßen Wert in Result und bei der Nutzung von `>>=` wird der Wert an eine Funktion f übergeben, wobei diese ihn bearbeiten, aber nicht entkapseln kann. Hat `x` den Typ `a`, so muss `f:: a -> Value b` gelten, da dies sonst nicht der generellen Definition der Monade entsprechen würde. Trotzdem ergibt diese bloße Definition der Monade beim Kompilieren (mit bspw. GHC) einen Fehler. Dies liegt daran, dass man den Datenkonstruktor Value, bevor man ihn als Instanz der Klasse Monade

[1] Eine weitere Funktion ist ”fail”. Für genauere Informationen siehe [`https://www.techfak.uni-bielefeld.de/ags/pi/lehre/AuDIWS14/Slides/Haskell/hask15.pdf`]

implementiert, zunächst als Instanz der Klassen `Functor` sowie `Applicative` instanziieren muss. Es existiert sozusagen eine Rangfolge. Bevor etwas als Monade deklariert werden darf, muss es erst als `Applicative` und davor als `Functor` deklariert werden. Dies soll an dieser Stelle jedoch nicht weiter behandelt werden. Damit das Programm - wenn auch mit Warnungen - kompiliert, reicht es Value als Instanz von `Functor` und `Applicative` zu deklarieren, es muss keine Funktionalität spezifiziert werden.

```
1  instance Functor Value
2  instance Applicative Value
```

Schauen wir uns nun ein kurzes Beispiel an, wobei wir wieder die „do-Notation verwenden:

```
1 ausdruck1 = do x <- Result 4
2                y <- Result 2
3                return (x/y)
4 ausdruck2 = do x <- Result 4
5                y <- Result 0
6                return (x/y)
```

Wie erwartet wertet `ausdruck1` zu 2.0 aus, wohingegen `ausdruck2` Infinity oder einen „divide-by-zero“-Error zurückgibt. (Dies ist abhängig vom verwendeten Compiler - Der GHC gibt Infinity zurück.)

3.3 Die vordefinierte Maybe-Monade

Neben der IO-Monade gibt es noch weitere vordefinierte Monaden in Haskell, wie beispielsweise die `Maybe-Monade`. Die Deklaration des Datentyps `Maybe` sieht dabei wie folgt aus:

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

Neben dem einstelligen Datenkonstruktor „Just“ gibt es hierbei noch den nullstelligen Datenkonstruktor „Nothing“ welcher für Fehlerwerte steht. Dieser ist vergleichbar mit „null“ in Java. Die Maybe-Monade wird nun wie folgt definiert:

```
1  instance Monad Maybe where
2  return x  =  Just x
3  Just x   >>= f  = f x
4  Nothing  >>= f  = Nothing
```

Die Funktion `return` soll ein Objekt x „kapseln“, d.h. es gibt dieses einfach nur an `Just` weiter. Die Implementation von `>>=` hängt nun davon ab, ob wir einen Fehlerwert, also `Nothing` haben. In diesem Fall soll es bei dem Fehlerwert bleiben und dementsprechend `Nothing` zurückgegeben werden. Falls allerdings ein „gekapselter“ Wert übergeben wird, so soll dieser entkapselt und auf die Funktion angewendet werden. Wenn x den Typ a hat, muss die Funktion f hierbei allerdings den Typ `f:: a -> Maybe b` haben (dies geht aus der Definition der Monade hervor, genau wie bei der Value-Monade).

Schauen wir uns nun beispielhaft zwei Ausdrücke an, um die Funktionsweise nachzuvollziehen. Wir nutzen hierbei wieder den syntaktischen Zucker der „do-Notation“:

```
ausdruck1 = do x <- Just 4
               y <- Just 2
               return (x/y)
ausdruck2 = do x <- Just 1
               y <- Nothing
               return (x/y)
```

Wie erwartet wertet `ausdruck1` zu 2.0 aus, wohingegen `ausdruck2` zu `Nothing` auswertet.

3.4 Modularität mit Monaden

Im Folgenden nutzen wir ineinander verschachtelte Modulo-Terme[2] und zeigen mit diesen anhand der im vorigen Teil bereits vorgestellten Value- und Maybe-Monade, inwiefern Monaden dazu beitragen können, modulares, änderungsfreundliches Programmieren zu unterstützen. Wir definieren uns für die verschachtelten Modulo-Terme die folgende Datenstruktur:

```
data Modulo = Straight Int| Mod Modulo Modulo
```

Es wäre natürlich wesentlich sinnvoller in einer Datenstruktur von Termen mehrere Rechenarten zuzulassen, wie bspw.:

```
data Term     = Straight Int| Chain Operator Term Term
data Operator   = Add|Sub|Mult|Div|Mod
```

Dadurch wäre das Beispiel [3] allerdings nicht mehr so anschaulich. Zudem würde es nicht zum Verstehen von Monaden beitragen, weswegen dieser praktischere Ansatz zwar erwähnt, aber im Folgenden nicht mehr berücksichtigt werden soll. Um die Funktionsweise des Modulo-Terms zu zeigen definieren wir folgende Beispiele:

```
modTerm1 = Mod (Straight 10) (Straight 2)
modTerm2 = Mod (Straight 5) modTerm1
modTerm3 = Mod (Straight 2) (Straight (-10))
```

Der erste Term steht für 10 mod 2 der zweite Term für 5 mod (10 mod 2), also 5 mod 0 der letzte für 2 mod (-10). Wir wollen nun mit Hilfe der Datenstruktur Value und der ihr zugehörigen Monade die Modulo-Terme auswerten. Dies soll folgende Funktion gewährleisten:

```
moca1 :: Modulo -> Value Int
moca1 (Straight x) = return x
moca1 (Mod n m) = do x <- moca1 n
                     y <- moca1 m
                     return (x `mod` y)
```

Sie ist aufgrund der „do-Notation“ sehr gut lesbar - zudem wird auch die sequentielle Abarbeitung der einzelnen „Unterterme“ deutlich. Der Aufruf `moca1 modTerm1` liefert wie

[2] Dieses Beispiel ist stark angelehnt, an ein Beispiel aus dem Skript (siehe [2])

[3] Ein weiteres komplexeres Beispiel mit euklidischem Algorithmus ist hier zu finden: [`https://diego.codes/post/learning-monads/`]

erwartet 0, der Aufruf `moca1 modTerm2`, welcher 5 mod 0 entspricht, liefert einen „divided-by-zero“-Fehler und `moca1 modTerm3` liefert -8.

Um einen Fehler zu umgehen, implementieren wir nun eine neue Funktion mit Hilfe der Datenstruktur Maybe. Wenn an einer Stelle des Terms modulo 0 gerechnet wird, so soll der gesamte Term `Nothing` ergeben. Wenn an einer Stelle modulo n mit $n < 0$ gerechnet wird, so soll mit Hilfe der Funktion `askv2` (aus dem IO-Teil) der Benutzer gefragt werden, ob er wirklich modulo einer negativen Zahl rechnen will. Je nach Eingabe, soll dann `Nothing` ausgegeben, oder die Berechnung normal weiter ausgeführt werden. Eine auf den ersten Blick passende Funktion könnte wie folgt aussehen:

```
1 s= "Sie rechnen modulo einer negativen Zahl"
2 moca2 :: Modulo -> Maybe Int
3 moca2 (Straight x) = return x
4 moca2 (Mod n m) = do x <- moca2 n
5                      y <- moca2 m
6                      if (y==0) then return Nothing else
7                                      if (y<0 && not(askv2 s)) then
                                          return (x `mod` y) else
                                          return (x `mod` y)
```

Hierbei ist `s` der String, der `askv2` übergeben wird. Die Funktion `moca2` wird nun allerdings nicht kompilieren. Wenn sie es tun würde, so würde die referentielle Transparenz in Haskell verletzt, da der Rückgabeparameter von `moca2` unter anderem von der Eingabe des Nutzers abhängt. Der Grund warum sie nicht kompiliert ist, dass `askv2 :: String -> IO Bool` definiert ist, wir den Rückgabewert von `askv2` in `moca2` allerdings behandeln, als wäre es Bool. Die einzige Möglichkeit eine Funktion zu programmieren, die eine derartige Funktionalität bietet, wäre entweder den gesamten Output wiederum in IO zu kapseln, oder aber die Funktion `unsafePerformIO: IO a -> a` zu benutzen.

Wir wollen uns nun stattdessen aber in der Funktion `moca2` die vorletzte Zeile (6) wegdenken, sodass eine Fallunterscheidung bei negativen Zahlen nicht eintritt. Auf diese Art und Weise kompiliert `moca2` und liefert auch das gewünschte Ergebnis. Der Aufruf `moca2 modTerm1` liefert also `0`, `moca2 modTerm2` liefert `Nothing`, `moca2 modTerm3` liefert `-8`. Es fällt hierbei auf, dass obwohl wir die Fehlerbehandlung für den Fall y==0 eingefügt haben, sich die Funktion bloß um eine weitere if-Abfrage von `moca1` unterscheidet. Vergleichen wir dies nun mit einer Implementierung der Funktionen `moca1` bzw. `moca2` ohne dabei die Funktionen der Monaden zu benutzen. Hierbei sollen `calc1` und `moca1` sowie `calc2` und `moca2` dieselbe Funktionalität besitzen.

```
1 calc1 :: Modulo -> Value Int
2 calc1 (Straight x) = Result x
3 calc1 (Mod n m) = Result (x `mod` y)
4                   where Result x = calc1 n
5                         Result y = calc1 m
```

```haskell
calc2 :: Modulo -> Maybe Int
calc2 (Straight x) = Just x
calc2 (Mod n m) = case calc2 n of
                        Nothing -> Nothing
                        Just x -> case calc2 m of
                                        Nothing -> Nothing
                                        Just y -> if y==0 then Nothing
                                                  else Just (x `mod` y)
```

Während sich die Funktionen `moca1` und `moca2` nur in einer if-Abfrage unterscheiden, kann man eindeutig sehen, dass bei `calc2` die einzelnen Fälle für `Nothing` erneut abgearbeitet werden müssen, was dazu führt, dass `calc2` nicht auf die Implementierung von `calc1` aufbauen kann. Bei der Implementation mit den Funktionen der Monaden werden die einzelnen Fälle für Nothing bereits in den Monaden durch die unterschiedliche Implementierung von `>>=` und `return` behandelt, sodass `moca1` und `moca2` dieselbe Struktur besitzen können.

Das Programmieren mit Monaden bietet also nicht nur den Vorteil der Beibehaltung von referentieller Transparenz, sondern es unterstützt auch Änderungsfreundlichkeit sowie Modularität in den einzelnen Funktionen.

4 Zusammenfassung

Abschließend wollen wir nun die Vor- und Nachteile von Monaden in Haskell kurz zusammenfassen. Der wichtigste Vorteil von Monaden ist der Erhalt der referentiellen Transparenz, und das gleichzeitige Ermöglichen von IO-Aktionen. Dies ist der Grund gewesen, Monaden in Haskell einzuführen. Monaden besitzen in der funktionalen Programmierung allerdings noch weitere Vorteile, die in dieser Ausarbeitung aufgezeigt wurden. Zu ihnen gehören sowohl die Änderungsfreundlichkeit, als auch die Modularität, was zwei sehr wichtige Konzepte in der Programmierung sind. Allerdings sind Monaden durchaus komplex und nicht einfach zu Erlernen für Programmierer imperativer Sprachen. Deshalb empfehlen wir für ein besseres Verständnis von Monaden weiterführende Literatur, welche an manchen Stellen, an denen wir uns kurzgehalten haben, in den Fußnoten erwähnt wurde. Diese sind im untenstehenden Literaturverzeichnis ebenfalls zu finden.

Literatur

[1] Bird, Richard. `Thinking functionally with Haskell` Cambridge University Press, 2014.

[2] Giesl, Jürgen. `Funktionale Programmierung` RWTH Aachen, 2016

[3] Hudak, Paul; Peterson, John; Fasel, Joseph; `A gentle introduction to Haskell Version 98`, 2000. URL `https://www.haskell.org/tutorial/monads.html` (Zugriff: 02.05.2019)