Haskell - Angewandter „abstract nonsense“

Luca Leon Happel

Bachelorseminar: Programmiersprachen
Wintersemester 2022

Zusammenfassung

Haskell ist eine Sprache, welche vor mehr als 30 Jahren zuerst erschien und dennoch selbst heutzutage ihrer Zeit in manchen Bereichen voraus ist. Trotz dessen ist sie in der Industrie wenig verbreitet, obwohl Programmierer viel von ihr lernen können. Inwiefern Programmierer davon profitieren können, wird in dieser Arbeit untersucht, indem Parallelen zwischen Haskell und anderen Sprachen gezogen werden. Wir erkennen dabei, dass Haskell mächtige Strukturen besitzt, welche in anderen Sprachen nicht oder nur teilweise existieren, da Haskell Datenstrukturen auf einem viel abstrakteren Level behandelt, wodurch Probleme mit Haskell weitaus effizienter und eleganter gelöst werden können.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen von Haskell **3**
2.1 Typen und Typklassen 4
2.2 Funktoren 5
2.2.1 Currying 7
2.2.2 Anwendung der Functor Typenklasse 10
2.2.3 Funktoren von **Hask** in andere Kategorien 12

3 Monaden und unser erstes Haskell Programm **14**
3.1 Listen als Monade 15
3.2 Die IO Monade und Hello World in Haskell 17
3.3 Warum Monaden? Oder: Warum Haskell? 18

Abbildungsverzeichnis **19**

Literatur **21**

1 Einleitung

Im September 1987 wurde die „Conference on Functional Programming Languages and Computer Architecture“in Portland, Oregon, USA, abgehalten. Ziel war es, die zersplitterte funktionale Programmiersprachen-Community zu vereinen und eine gemeinsame Standardsprache zu entwickeln. Die Sprache sollte alle Vorteile der bis

dahin existierenden funktionalen Sprachen, insbesondere der zu der Zeit populären Sprache „Miranda“, vereinen und gleichzeitig die Nachteile beseitigen. Die Sprache, welche dabei entstand, ist Haskell [8].

Während ihrer Entwicklung wurde Haskell stark von Akademikern beeinflusst, wobei insbesondere Themen wie Typentheorie, Kategorientheorie und das Lambda-Kalkül eine große Rolle spielten. Daraus resultierte, dass Haskell einige Besonderheiten aufweist, welche es von anderen Sprachen unterscheidet. Diese Besonderheiten sind es, welche Haskell zu einer so interessanten Sprache machen, da somit Haskell eine der wenigen reinen Programmiersprachen ist, also keine Seiteneffekte vorhanden sind. Eine unintuitive Konsequenz daraus ist, dass Haskell keine Variablen besitzt, welche verändert werden können, da es keine Zuweisung gibt, da Zuweisungen in Haskell Seiteneffekte erzeugen würden. Somit existieren in Haskell nur Konstanten, welche nicht verändert werden können. Dies wiederum führt dazu, dass klassische Schleifen in Haskell nicht existieren, da diese durch Seiteneffekte, insbesondere dem Inkrementieren einer Laufvariable, implementiert werden. Stattdessen werden in Haskell Funktionen verwendet, welche rekursiv aufgerufen werden, um die gewünschte Schleifenfunktionalität zu erreichen.

Statt der Objektorientierung verwendet Haskell die funktionale Programmierung, was durch die Reinheit bedeutet, dass Haskell sehr gut durch kategorientheoretische Konzepte betrachtbar ist. “Kategorientheoretische Konzepte“beziehen sich auf die Kategorientheorie, welche eine Theorie ist, welche eine alternative Grundlage für die Mathematik darstellt, im Gegensatz zu der klassischen Mengenlehre. Oft werden Konzepte aus der Kategorientheorie auch als „abstract nonsense “bezeichnet [3]. Diese besagt, dass es Kategorien gibt, in welchen es sogenannte Objekt gibt, welche durch Morphismen verbunden sind. Existieren Objekte A, B und C in einer Kategorie $\mathcal{C}$, sowie Morphismen $f : A \to B$ und $g : B \to C$, so gibt es einen Morphismus $g \circ f : A \to C$ und außerdem muss es für jedes Objekt A einen Morphismus $id_A : A \to A$ geben, welcher das Objekt A nicht verändert. In dem Fall von Haskell sind die Objekte die Typen und die Morphismen die Funktionen, wobei dies noch weitreichende Konsequenzen hat, welche im Folgenden noch aufgefasst werden. Funktionale Programmierung bedeutet, dass Programme durch Funktionen definiert werden, welche Daten als Eingabe erhalten und Daten als Ausgabe liefern. Geschickte Komposition von Funktionen ermöglicht es, komplexe Probleme durch hintereinander aufrufende einfache Funktionen zu lösen. Dies hat zur Folge, dass Programmierer, welche Haskell lernen, automatisch besser in abstrakter Mathematik werden. Außer der Reinheit und der funktionalen Programmierung, ist Haskell eine polymorphe, statisch-typisierte Programmiersprache mit lazy-evaluation, wobei diese Begriffe im Folgenden erläutert werden.

Polymorphie bedeutet, dass Funktionen und Datenstrukturen generisch sind, was bedeutet, dass sie mit verschiedenen Typen verwendet werden können. Statische Typisierung bedeutet, dass der Typ einer Variable zur Kompilierzeit festgelegt wird, wodurch Fehler, welche durch dynamische Typisierung entstehen, vermieden werden. Durch das algebraische Typensystem von Haskell, was bedeutet, dass Typen A und B zu Produkttypen $A \times B$ und Summentypen $A|B$ kombiniert werden können,

ist es möglich, Funktionen und Datenstrukturen mit verschiedenen Typen zu kombinieren. Hier sehen wir wieder, dass ein starker Fokus auf dem Kombinieren von einfacheren Strukturen liegt, wodurch komplexere Strukturen entstehen. Da Haskell auch eine funktionale Programmiersprache ist, haben auch Funktionen einen Typen, wodurch Funktionen, Konstanten und Variablen alle durch die starke Typisierung von Haskell definiert werden können und somit eine Typenstruktur aufweisen. Da wir bereits wissen, dass Haskell eine stark-typisierte Programmiersprache ist, bedeutet dies, dass Programmierfehler in Haskell fast immer von dem Compiler erkannt werden. Sollte eine Funktion mit einem falschen Typen aufgerufen werden, oder einen falschen Typen zurückgeben, so wird dies vom Compiler erkannt. Diese Eigenschaft von Haskell ist sehr nützlich, da es so möglich ist, sicherheitskritische Programme zu schreiben, welche mit erhöhter Wahrscheinlichkeit korrekt sind als Programme in anderen Sprachen.

Die lazy-evaluation von Haskell bedeutet, dass Ausdrücke erst dann ausgewertet werden, wenn sie benötigt werden. Dies hat zur Folge, dass Ausdrücke, welche nie benötigt werden, nicht ausgewertet werden. Dies ist sehr nützlich, da es so möglich ist, unendliche Datenstrukturen zu definieren, welche nur dann ausgewertet werden, wenn sie benötigt werden. Beispielsweise kann eine Liste von allen Primzahlen definiert werden, wobei man nun die n-te Primzahl aus dieser Liste erhalten kann, ohne dass die Liste vollständig ausgewertet werden muss. Auch wird Code somit idiomatischer, da man analog zur natürlichen Sprache „die n-te Primzahl“verwenden kann, statt diese durch eine Funktion zu erhalten, welche die n-te Primzahl zurückgibt.

Diese Eigenschaften von Haskell machen es zu einer sehr ungewöhnlichen Programmiersprache, welche sich von anderen Programmiersprachen wie C, Java oder Python unterscheidet. Diese Unterschiede werden im Folgenden noch näher erläutert, wobei wir sehen werden, dass viele Konzepte von Haskell über die Jahre durch ihre praktische Anwendung in anderen Programmiersprachen übernommen wurden. Der tiefe mathematische Hintergrund von Haskell ist jedoch immer noch außergewöhnlich und erlaubt durch die abstrakten Strukturen darin, Probleme sehr elegant zu lösen, wobei dies auch ein zweiseitiges Schwert ist, da es für Programmierer, welche nicht darin geschult sind, sehr schwer zu verstehen ist. Hier werden wir auch sehen, dass andere Programmiersprachen, welche die Konzepte von Haskell übernommen haben, diese Konzepte nicht so abstrakt und rigoros implementiert haben wie Haskell, wodurch die Konzepte dort auf bestimmte Anwendungen spezialisiert sind, und in ihrem Umfang eingeschränkt sind. Als Beispiel dafür werden wir den Typen `Option<u32>` in Rust betrachten, welcher ähnlich dem Typen `Maybe Int` in Haskell ist, jedoch aufgrund seiner fehlenden Funktor-Struktur unhandlicher ist und mehr Fallunterscheidungen erfordert.

2 Grundlagen von Haskell

Wir möchten Haskell mit anderen Programmiersprachen vergleichen. Interessant ist, dass wir Konzepte, welche in Haskell verwendet werden und aus der Kategorientheo-

rie stammen, dazu verwenden können, um dieses Ziel zu verwirklichen.

2.1 Typen und Typklassen

Betrachten wir zuerst eine Typenklasse, welche in Haskell definiert ist. Dazu müssen wir jedoch zuerst definieren, was eine Typenklasse ist.

Definition 2.1. Eine Typenklasse ist eine Sammlung von Typen, welche eine bestimmte Eigenschaft besitzen.

Nach der Einleitung wissen wir bereits, was ein Typ ist. Diese kommen aus der Typentheorie und ordnen Werten in unserer Programmiersprache eine Eigenschaft genannt „Typ“zu. Allgemeiner bezeichnen wir als ein „Typensystem“eine konsistente Theorie T, für welche zu jeder Aussage ϕ der Form „x hat den Typ y“, mit x ein Wert und y ein Typ, entweder ϕ gilt in T, oder $\neg\phi$ gilt in T, wobei „$\neg$“die logische Negierung ist, gilt. Dabei bedeutet „konsistente“, dass es keine Kontradiktion in dieser Theorie gibt, also keine Aussage ϕ existiert, für welche sowohl ϕ in T als auch in $\neg\phi$ in T gilt. Ein Beispiel für einen Typ ist der Typ $Bool$ welcher den Werten $True$ und $False$ in Haskell zugeordnet ist. Ein weiterer Typ wäre der Typ Int, welcher den ganzen Zahlen zugeordnet ist. Ein Beispiel für eine Typentheorie ist die Typentheorie von Haskell, welche unter anderem die Typen $Bool$ und Int enthält. Inkonsistent, also nicht konsistent, wäre die Typentheorie von Haskell, wenn beispielsweise die Aussage „$True$ hat den Typ Int“in dieser Theorie und die Aussage „$True$ hat nicht den Typ Int“gelten würde, was jedoch nicht gilt.

Die Definition einer Typenklasse als Sammlung von Typen ist jedoch auf den ersten Blick nicht sonderlich aufschlussreich. Der Begriff „Sammlung“bezieht sich in dem Kontext von Haskell auf eine Menge von Typen im Sinne der Mengenlehre. Wir können sogar weiter eingrenzen und stark davon ausgehen, dass die Menge von Typen, welche eine Typenklasse definiert, endlich in Haskell ist, da ein Programm in Haskell nur endlich viele Typen definieren kann.

Wir können nun die Implementation von Typen und Typenklassen in Haskell genauer betrachten. Wir können in Abbildung 1 sehen, wie ein einfacher Typ in Haskell definiert wird und in Abbildung 2 eine Funktion, welche einen Typen als Parameter verwendet. Insbesondere bezeichnen wir die erste Zeile `f :: MyType -> Bool` als „Typsignatur“und die zweite Zeile `f (MyConstruktor b i) = b` als „Definition“der Funktion.

```
data MyType = MyConstruktor Bool Int
```

Abbildung 1: Definition eines Typens genannt „MyType“mit Konstruktor „MyConstruktor“, welcher einen „Bool“und einen „Int“speichert [5]

Eine Typenklasse wird nun wie in Abbildung 3 definiert. Wir können sehen, dass eine Typenklasse eine Sammlung von Typen parametrisiert durch einen Typenvariablen a ist, wobei eine Funktion `myFunction :: a -> Bool` existieren muss. Diese

```
f :: MyType -> Bool
f (MyConstruktor b i) = b
```

Abbildung 2: Definition einer Abbildung, welche einen Wert von Typ „MyType“auf einen Wert von Typ „Bool“schickt.

Typenklasse ist jedoch bisher nur eine Definition, wobei kein Typ ein Teil dieser Typenklasse ist. Damit wir einen Typen in diese Typenklasse hinzufügen können, müssen wir beweisen, dass ein gegebener Typ die Eigenschaften der Typenklasse erfüllt. Dies wird in Abbildung 4 gezeigt. Wir können sehen, dass wir die Typenklasse `MyTypeClass` mit dem Typen `MyType` verbinden, indem wir den Typen `MyType` als Instanz der Typenklasse `MyTypeClass` definieren. Insbesondere haben wir für alle Typensignaturen, welche nach dem `where` in Abbildung 3 stehen, eine Definition, welcher der entsprechenden Typensignatur zustimmt, in der Instanziierung gegeben.

```
class MyTypeClass a where
  myFunction :: a -> Bool
```

Abbildung 3: Definition einer Typenklasse, welche einen Typen „a“als Parameter besitzt.

```
instance MyTypeClass MyType where
  myFunction (MyConstruktor b i) = b
```

Abbildung 4: Definition einer Typenklasse, welche einen Typen „a“als Parameter besitzt.

Als ein pathologisches Beispiel für eine Typenklasse, welche von jedem Typen a implementiert werden kann, wobei „implementiert“gleichzusetzen mit „a ist ein Typ in der entsprechenden Typenklasse“ist, ist die Typenklasse `Trivial`, welche in Abbildung 5 definiert ist. Die entsprechende Implementation für den Typen `MyType` ist in Abbildung 6, wobei wir sehen, dass wir keine Implementationen für Funktionen vornehmen müssen.

2.2 Funktoren

Als besondere Typenklasse ist die `Functor` Typenklasse [6] zu erwähnen. Die `Functor` Typenklasse ist eine Typenklasse, welche im „Prelude“von Haskell [1] definiert ist. Diese Definition können wir in Abbildung 7 genauer betrachten. Die erste Zeile ist ein

[1] der Prelude ist gewissermaßen die Standartbibliothek von Haskell, welche automatisch in jedem Haskellprogramm eingebunden wird

```haskell
class Trivial a
```

Abbildung 5: Definition einer pathologischen Typenklasse, welche für jeden Typen „a“als Parameter ohne Einschränkung implementiert werden kann.

```haskell
instance Trivial MyType
```

Abbildung 6: Definition einer Typenklasse, welche einen Typen „a“als Parameter besitzt.

„Type-Constraint“und besagt, dass nur Typen der Art `*->*` in die `Functor` Typenklasse implementiert werden können. Die „Art“ist ein Konzept aus der Typentheorie [7], welches wir hier nicht näher betrachten. Das einzige wichtige für uns ist, dass die Art `*->*` besagt, dass der Typenparameter a ein Typenkonstruktor ist, welcher einen Typen reinnimmt und einen Typen zurückgibt. Hierbei sind Typenkonstruktoren bereits aus Abbildung 1 bekannt, wobei dort der Typenkonstruktor `MyConstruktor` war. Dieser hätte die Art `*`, da er keine freien Typenparameter gibt, wohingegen `data MyType a b = MyConstruktor a b` von der Art `*->*->*` ist, da er zwei freie Typenparameter hat.

```haskell
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
    -- Defined in 'GHC.Base'
```

Abbildung 7: Definition der „Functor“Typenklasse.

In der Definition der `Functor` Typenklasse sehen wir, dass nach dem `where` eine Typensignatur `fmap` und eine Typensignatur `<$` definiert sind. Diese Typensignaturen sind die Funktionen, welche die Instanzen der `Functor` Typenklasse implementieren müssen. Die Typensignatur `fmap` besagt, dass wir eine Funktion `fmap` definieren müssen, welche eine Funktion vom Typen `(a -> b)` und einen Wert vom Typen `f a` einnimmt und einen Wert vom Typen `f b` zurückgibt. Wichtig für das Verständnis dieser Typensignaturen, welche mehr als einen Pfeil besitzen, ist das sogenannte Konzept des Currying [2].

[2] Auch „Schönfinkeln“in dem Bereich der Linguistik gennant

2.2.1 Currying

Das Konzept des Curryings ist ein Konzept aus der Mathematik, welches in der Programmierung verwendet wird, um Funktionen mit mehreren Argumenten mittels einer natürlichen Bijektion zwischen Hom-Mengen elegant zu definieren. Was wir damit meinen, möchten wir in diesem Unterabschnitt näher erläutern, genauso wie, welche Konsequenzen das Currying für die Typensignaturen der `Functor` Typenklasse hat.

Nach der Einleitung dieser Arbeit wissen wir bereits, was eine Kategorie $\mathcal{C}$ ist. Wir wissen, dass diese Kategorie Objekte A, B beinhalten kann, sowie Morphismen $f : A \to B$. Wir einigen uns auf die Konvention, dass wir $A \in \text{ob}(\mathcal{C})$ für ein Objekt A in der Kategorie $\mathcal{C}$ schreiben und $f : A \to B \in \text{Hom}_{\mathcal{C}}(A, B)$ für einen Morphismus von A nach B in der Kategorie $\mathcal{C}$. Allgemein sind $\text{ob}(\mathcal{C})$ und $\text{Hom}_{\mathcal{C}}(A, B)$ echte Klassen, also keine Mengen im Sinne der Mathematik. Wir werden jedoch nicht näher auf den Unterschied zwischen Mengen und Klassen eingehen, da dies nicht relevant für die Kategorie **Hask** ist, welche die Kategorie der Typen und Funktionen in Haskell ist[3]. Die genaue Konstruktion der Kategorie **Hask** lässt sich in der offiziellen Dokumentation von Haskell [4] nachlesen. Wichtig ist nur, dass Typen in Haskell Objekte in der Kategorie **Hask** sind und Funktionen zu Morphismen in der Kategorie **Hask** unter bestimmten Voraussetzungen korrespondieren. In der Kategorie **Hask** gilt, dass $\text{ob}(\mathbf{Hask})$ und $\text{Hom}_{\mathbf{Hask}}(A, B)$ Mengen für alle $A, B \in \text{ob}(\mathbf{Hask})$ sind.

Definition 2.2. Wir definieren den Exponenten Z^Y von Objekten $Z, Y \in \text{ob}(\mathbf{Hask})$ als die Menge aller Morphismen $h : Z \to Y$ vom Objekt Z nach Y.

Wir können nun das Konzept des Curryings in der Kategorie **Hask** definieren. Dazu betrachten wir uns folgende Eigenschaft der Kategorie **Hask**:

Theorem 2.1. *Für alle Objekte $X, Y, Z \in ob(\boldsymbol{Hask})$ gilt:*

$$Hom_{\boldsymbol{Hask}}(X \times Y, Z) \cong Hom_{\boldsymbol{Hask}}(X, Y^Z) \tag{1}$$

Beweis. Sei $f : X \times Y \to Z$ ein Morphismus in der Kategorie **Hask**. Dieser korrespondiert zu einer Funktion $f : X \times Y \to Z$ in Haskell, also eine Funktion, welche ein Tupel aus X und Y einnimmt und etwas vom Typ Z zurückgibt. Wir definieren nun eine Funktion $g : X \to Y^Z$ als $g(x) = f(x, \cdot)$, wobei $\cdot$ ein Platzhalter für ein Argument vom Typ Y ist, also $f(x, \cdot) = (y \mapsto f(x, y))$.

Es ist klar, dass g eine Funktion vom Typ $X \to Y^Z$ ist, da g einen Wert vom Typ X einnimmt und eine Funktion von Y nach Z zurückgibt. Dies sind aber genau die Werte vom Typ Y^Z.

Allgemein können wir eine Abbildung $\text{Hom}_{\mathbf{Hask}}(X \times Y, Z) \tilde{\to} \text{Hom}_{\mathbf{Hask}}(X, Y^Z)$ konstruieren, indem wir für jeden Morphismus $f : X \times Y \to Z \in \text{Hom}_{\mathbf{Hask}}(X \times Y, Z)$ die Abbildung $\phi(f) = g$ definieren, wobei g die Funktion $g : X \to Y^Z$, $g(x) =$

[3] Per Konvention nehmen wir an, dass Funktionen in Haskell terminieren, also endliche Laufzeit haben

[4] `https://wiki.haskell.org/Hask#\%22Platonic\%22_Hask`

$(y \mapsto f(x, y)$ ist. Die Bijektivität ist nach Konstruktion gegeben, was wir durch $\tilde{\to}$ symbolisieren. □

Wir können noch viel mehr über diese Abbildung ϕ sagen, jedoch beschränken wir uns darauf, dass dies als Implikation bedeutet, dass die Funktionen in Abbildung 12 praktisch identisch sind. Für den interessierten Leser bietet sich hierbei [1, S. 53, Proposition 3.3.2] an. Man kann also `f` und `g` als die gleiche Funktion betrachten, da man durch Anwenden von ϕ auf `f` die Funktion `g` erhält und umgekehrt durch Anwenden von ϕ^{-1} auf `g` die Funktion `f` erhält. Wichtig ist hierbei, dass wir `g x`, also die Funktion `g` angewandt auf den Parameter `x`, durch eine Lambda-Funktion `(\y -> f (x y))` definiert haben, welche also ein `y` nimmt und `f` auf `(x, y)` angewandt ausgibt.

```
f :: (a, b) -> c
g :: a -> (b -> c)
g x = \y -> f (x, y)
```

Abbildung 8: Typensignaturen von Funktionen, welche äquivalent bezüglich curryings sind

Wir erkennen aus dieser Tatsache, dass es effizienter ist, die Funktionen `f` und `g` direkt ohne Klammern zu schreiben. So erhalten wir eine Kurzschreibweise für die Funktionen `f` und `g`, welche wir in Abbildung 9 als Funktion `h` sehen können. Dem Verständnis geschuldet, haben wir auch eine äquivalente Definition von `h` in Abbildung 10 gegeben.

```
f :: (a, b) -> c
g :: a -> (b -> c)
g x = \y -> f (x, y)

h :: a -> b -> c
h x y = f (x, y)
```

Abbildung 9: Kurzschreibweise resultierend aus dem Currying basierend basierend auf der Definition von f

Die Vorteile hierbei sind vielfältig. Ein offensichtlicher Vorteil ist, dass Funktionen effizienter und lesbarer dargestellt werden können, wie wir in Abbildung 11 sehen können. Hier nehmen wir ans, dass die Funktion `callServer` zwei Parameter vom Typ `String` nimmt und einen Wert vom Typ `String` zurückgibt. Sie könnte beispielsweise eine URL auf einen Server, definiert im ersten Parameter, mit einem bestimmten Header, definiert im zweiten Parameter, aufrufen und den Inhalt der Seite zurückgeben.

```haskell
f :: (a, b) -> c
g :: a -> (b -> c)
g x = \y -> f (x, y)

h :: a -> b -> c
h x y = (g x) y
```

Abbildung 10: Kurzschreibweise resultierend aus dem Currying basierend basierend auf der Definition von g

```haskell
callServer :: String -> String -> String
callServer url header = ...

apiCall :: String -> String
apiCall = callServer "https://api.example.com"

apiCallWithHeader :: String
apiCallWithHeader = apiCall "example header"
```

Abbildung 11: So binden Sie Code ein

Des Weiteren kann man durch currying Funktionen höhere Ordnung eleganter verwenden. Am Beispiel der Funktion `map : (a -> b) -> [a] -> [b]`, welche eine Funktion vom Typ `a -> b` auf eine Liste vom Typ `[a]` anwendet und eine Liste vom Typ `[b]` zurückgibt, können wir sehen, dass das Currying in Anbetracht der Funktion `+ : Num a => a -> a -> a` sehr nützlich ist. Hierbei ist `+` eine Funktion vom Typ `Num a => a -> a -> a`, was bedeutet, dass `+` zwei Werte des Typs `a` nimmt und einen Wert des Typs `a` zurückgibt, wobei `a` ein Typ ist, der die Klasse `Num` implementiert. `Num` ist eine Typenklasse, welche die Grundrechenarten implementiert. Also sagen wir mit `Num a => ...` aus, dass `+` auf dem Typen `a` definiert sein muss.

```haskell
increaseByOne :: [Int] -> [Int]
increaseByOne = map (+ 1)
```

Abbildung 12: Wir komponieren die map Funktion geschickt mit einer partiellen Anwendung der gecurryten Funktion + um eine Funktion zu erhalten, welche alle Elemente einer Liste von Zahlen um 1 zu erhöhen.

2.2.2 Anwendung der Functor Typenklasse

Nachdem wir nun das notwendige Wissen über currying haben um die Typensignaturen der Functor Typenklasse in Abbildung 7 zu verstehen, können wir uns nun mit der Implementierung der Functor Typenklasse beschäftigen und herausfinden, warum diese so wichtig ist.

```
fmap (\x -> x+1) [1..10]
```

Abbildung 13: Anwendung der kanonischen Funktor Instanz auf der Liste von ganzen Zahlen z mit $0 \leq z < 10$, wobei jede Zahl in der Liste um eins erhöht wird.

Betrachten wir dafür Abbildung 13, wobei wir erkennen, dass wir durch `fmap` eine Funktion vom Typ `a -> b` auf eine Liste vom Typ `[a]` anwenden können. Wir haben also unsere Funktion vom Typ `a -> b` auf einen Typen, welcher unseren ursprünglichen Typen `a` beinhaltet „hochgehoben“. Das wichtige hierbei ist, dass wir erkannt haben, dass Listen von einem beliebigen Typen `[a]` eine „kanonische“, beziehungsweise offensichtliche, Möglichkeit besitzen, Funktionen vom Typ `a -> b` auf sich anzuwenden.

Dieses Konzept tritt überall in der Programmierung auf. Wir möchten mit Strukturen arbeiten, indem wir ihre Bestandteile manipulieren. Um dieses Konzept zu festigen, betrachten wir einen weiteren Typen in Haskell, welcher durch seine Nützlichkeit auch in der Programmiersprache Rust anzutreffen ist. Dieser Typ wird in Haskell `Maybe` genannt und in Rust `Option` [4].

```
data Maybe a = Nothing | Just a

meineDivision :: Int -> Int -> Maybe Int
meineDivision x y = if y == 0 then Nothing else Just (x `div` y)
```

Abbildung 14: Der Maybe Datentyp und wie man eine einfachere Funktion mit diesem schreibt.

Der Maybe Typ, wie er in Abbildung 14 definiert ist, ist auch Bestandteil des `Prelude` Pakets und somit auch in jedem Haskell Programm verfügbar. Er ist ein Typ, welcher entweder den Wert `Nothing` oder den Wert `Just a` beinhaltet, wobei `a` ein beliebiger Typ ist. Diese Definition basiert auf dem algebraischen Typensystem von Haskell, welches wir in der Einleitung kennengelernt haben. Außerdem lernen wir nun die `if` Anweisung kennen, welche in Haskell durch `if ... then ... else ...` verwendet wird. Diese ist klar verständlich und ähnlich wie in anderen Programmiersprachen. Die Funktion `meineDivision` nimmt zwei Zahlen vom Typ `Int` und gibt entweder `Nothing` oder `Just Int` zurück, je nachdem ob der zweite Parameter `y` gleich 0 ist oder nicht. Dies erfasst die allgemein bekannte Idee, dass „etwas

schlimmes passiert, wenn man durch 0 teilt“.

```
erhoeheMaybe :: Maybe Int -> Maybe Int
erhoeheMaybe Nothing = Nothing
erhoeheMaybe (Just x) = Just (x + 1)
```

Abbildung 15: Naive Definition von „erhoeheMaybe“

Wir können nun die Funktion `erhoeheMaybe` wie in Abbildung 15 definieren, welche eine Funktion vom Typ `Maybe Int -> Maybe Int` ist. Diese Funktion erhöht den Wert eines Wertes von Typ `Maybe Int` um eins, falls dieser nicht `Nothing` ist, also `Just Int` ist und somit einen Wert vom Typ `Int` beinhaltet. Diese Funktion lässt somit `+ 1` auf kanonische Weise auf dem Typen `Maybe Int` wirken, wobei dies ein Stichwort dafür ist, dass `Maybe` eine Instanz der Functor Typenklasse ist. Sieht man in der Haskell-Dokumentation nach, so findet man heraus, dass `Maybe` eine Instanz der Functor Typenklasse ist, und somit unser Instinkt richtig war, wodurch wir Funktionen wie `(+ 1)` vom Typ `Int -> Int` auf `Maybe Int` anwenden können. Durch ein geschicktes Refactoring können wir die Funktion `erhoeheMaybe` somit in eine Zeile umschreiben, statt eine Fallunterscheidung zu schreiben. Wir können dies in Abbildung 16 sehen.

```
erhoeheMaybe = fmap (+ 1)
```

Abbildung 16: Durchdachtere Definition von „erhoeheMaybe“

Das interessante ist, dass wir hierbei durch unser Refactoring nicht nur die Lesbarkeit verbessert haben, sondern auch deutlich weniger Code geschrieben haben, sowie durch das Erkennen der unterliegenden Struktur in der Lage sind, die Funktion `erhoeheMaybe` statt nur auf `Maybe Int` zu definieren, allgemeiner zu definieren. Indem wir die Funktion `fmap` verwenden und somit die Funktorialität von `Maybe` ausnutzen, mussten wir gar keinen konkreten Bezug auf den Typen Maybe mehr herstellen. Wir können also eine neue Typensignatur für `erhoeheMaybe` definieren, welche in Abbildung 17 zu sehen ist.

```
erhoeheMaybe :: (Functor f, Num b) => f b -> f b
erhoeheMaybe = fmap (+ 1)
```

Abbildung 17: Typensignatur von „erhoeheMaybe“nach einem Refactoring. „erhoeheMaybe“ist nun auf allen „f“anwendbar, welche Funktoren sind und einen Typen „b“, welcher eine Zahl ist, im ersten Parameter haben.

Hiermit haben wir eine enorme Verallgemeinerung erreicht, wodurch unsere Funk-

tion `erhoeheMaybe` beispielsweise im Spezialfall von `f = []` und `b = Int`, also auf Listen von Integern, angewendet werden kann. In diesem Spezialfall erhöht `erhoeheMaybe` die Werte in einer Liste von Zahlen jeweils um eins, was bedeutet, dass `erhoeheMaybe` als Spezialfall die Funktion `increaseByOne` aus Abbildung 11 ist. Wir können dies in Abbildung 18 sehen.

```
erhoeheMaybe [1, 2, 3] = [2, 3, 4]
erhoeheMaybe Just 1 = Just 2
```

Abbildung 18: Anwendung von „erhoeheMaybe“auf Listen von Zahlen und auf „Maybe Int“.

Wir können dieses Schema fortführen und beispielsweise einen Datentypen für Bäume definieren, wobei ein Knoten jeweils einen Wert vom Typ `a` beinhalten kann. So könnten wir auch eine Funktorinstanz für diesen Datentypen definieren, wodurch die Funktion `erhoeheMaybe` auch auf Bäume angewendet werden kann. Wir sehen also, dass die Funktorialität von Datentypen ein sehr mächtiges Konzept ist.

2.2.3 Funktoren von Hask in andere Kategorien

Wichtig ist, dass wir bisher nur Funktoren von **Hask** in **Hask** gesehen haben. Wir nennen solche Funktoren „Endofunktoren“. Interessant ist, dass wir jedoch die Ähnlichkeiten zwischen Haskell und Rust auch durch Funktoren beschreiben können. Wir können allgemeiner „Ähnlichkeiten“zwischen Haskell und anderen Programmiersprachen mit den mathematischen Konzepten, welche das Fundament von Haskell bilden, kurz und bündig beschreiben.

Wir nehmen dazu an, dass wir analog zu der Kategorie **Hask** eine Kategorie **Rust** haben, welche analog zu **Hask** definiert ist. Die Objekte der Kategorie **Rust** sind die Typen und die Morphismen sind die Funktionen zwischen den Typen in Rust. Im allgemeinen würde dies keine Kategorie im Sinne der Definition von Kategorien aus der Einleitung sein, weshalb wir analog zu der Definition von **Hask** beispielsweise nur terminierende Funktionen in **Rust** zulassen. Die Konzepte, welche wir hier erschließen, lassen sich jedoch auch auf nicht-terminierende Funktionen erweitern, wobei wir darauf nicht weiter eingehen möchten.

Wir haben gesehen, dass für Typen, welche eine Funktor-Instanz sind gilt, dass sie die Funktion `fmap :: (a -> b) -> f a -> f b` haben, welche eine Funktion vom Typ `a -> b` auf einen Funktion vom Typ `f a -> f b` bis auf Currying schickt. Das wichtige dabei ist, dass wir dies „strukturerhaltend“machen möchten. Betrachten wir dazu das Beispiel aus Abbildung 16. Wir möchten, dass wenn `x` ein Wert vom Typ `Int` ist, dann soll zuerst das Anwenden von `(+ 1)` auf `x` und dann das Anwenden von `Just` auf das Ergebnis das selbe sein, wie wenn man zuerst `Just` auf `x` anwendet und dann `fmap (+ 1)` auf das Ergebnis. Wir können dies in Abbildung 19 sehen.

Eine andere Art diese ausschlaggebende Eigenschaft von Funktor-Instanzen zu beschreiben, ist indem wir ein sogenanntes „kommutatives Diagramm“aufstellen. Wir

```
fmap (+ 1) (Just 1) = Just 2
Just ((+ 1) 1) = Just 2
```

Abbildung 19: Beispiel der grundlegenden Rechenregel für die Funktor-Instanz von „Maybe“.

können dies in Abbildung 20 sehen. Was wir damit aussagen möchten ist, wenn ich zuerst rechts und dann runter gehe, indem ich die entsprechenden Funktionen anwende, ist es das selbe, wie wenn ich zuerst runter und dann rechts gehe.

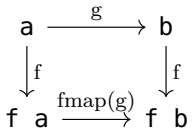
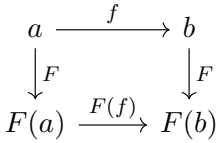


Abbildung 20: kommutatives Diagramm, welches die Eigenschaften von „fmap“darstellt.

Für Funktoren in **Hask** ist dies die einzige Eigenschaft, welche sie erfüllen müssen. Allgemein ist dies sogar die kategorientheoretische Definition eines Funktors. Ein Funktor F von einer Kategorie $\mathcal{C}$ in eine Kategorie $\mathcal{D}$ ordnet jedem Objekt $a \in \mathcal{C}$ ein Objekt $F(a) \in \mathcal{D}$ zu und jedem Morphismus $f : a \to b$ in $\mathcal{C}$ einen Morphismus $F(f) : F(a) \to F(b)$ in $\mathcal{D}$ zu, sodass das folgende Diagramm kommutiert:



Besitzt ein Typ in **Hask** eine Funktor-Instanz, so sagen wir damit aus, dass die konstruieren dieses Typens Funktoren im Sinne der Kategorientheorie sind

Dies erlaubt es uns nun auch einen abstrakteren Funktor F von **Hask** nach **Rust** zu definieren, indem wir jeden Typen a in **Hask** mit einem Typen $F(a)$ in **Rust** assoziieren. Beispielhaft könnte der Typ `Int` mit dem Typen `u32` assoziiert werden und so weiter. Insbesondere müsste dann auch F(`Maybe Int`) mit einem Typen in **Rust** assoziiert werden, welcher die Funktionalität von `Just` erhält. Dafür bietet sich der Typ `Option<u32>` in **Rust** an. Der Typ `Option<u32>` in Rust ist dadurch definiert, dass er entweder den Wert `None` oder den Wert `Some(u32)` annehmen kann. Wir können somit F(`Just x`) auf `Some(x)` für ein beliebiges x von Typ `Int` in **Hask** abbilden, sowie F(`Nothing`) auf `None` abbilden. Nun müssen wir nur noch für eine Abbildung `g :: Int -> b` in **Hask** die zugehörige Abbildung `F(g)` in **Rust** finden. Dies ist jedoch dadurch klar, indem wir durch Pattern-Matching auf `Some(x)` und `None` unterscheiden und für beide Fälle die entsprechende Abbildung `g` anwenden. Dies ist in Abbildung 21 dargestellt.

```
// result ist ein Option<u32>
match result {
  Some(x) => F(h(x)),
  None    => None
  }
```

Abbildung 21: Implementation von $F(\text{g} :: \text{Maybe Int} \to \text{a})$, wobei h die eindeutig von f induzierte Funktion $\text{h} :: \text{Just Int} \to a$ ist.

Was dies bedeutet ist, dass wir mit den Konzepten, welche in Haskell existieren vergleiche mit anderen Programmiersprachen ziehen können. Insbesondere können wir sehen, das **Rust** Strukturen besitzt, welche denen in **Hask** ähneln. Auch können wir sehen, dass es in Haskell einfacher möglich ist mit verkapselten Typen zu arbeiten, da wir auf dem `Maybe` Typen eine Funktor-Instanz besitzen. Diese fehlt jedoch in Rust, wodurch wir nur durch Pattern-Matching ein ähnliches Verhalten auf den `Option` Typen erreichen können. Hier zeigt sich jedoch ein Muster, welches Haskell vollkommen durchzieht: Konzepte, welche in Haskell zum eleganten Programmieren existieren haben auch praktische Anwendung in anderen Bereichen, wie zum Beispiel dem Vergleichen von Programmiersprachen. Haskell zu lernen trägt somit dazu bei, generell besser Strukturen zu erkennen, was im Endeffekt auch zu einem besseren Programmieren führt.

3 Monaden und unser erstes Haskell Programm

> If you're approaching Haskell monads a little nervously, that's understandable. You've probably heard that monads are a very powerful code-structuring technique, but ... sometimes, with great power comes great (apparent) nonsensicality.
>
> *Haskell Wiki*
> *All About Monads*

Ein sehr Populäres Konzept in Haskell, woran viele Programmierer beim Erlernen der Sprache scheitern, sind Monaden. Monaden sind ein weiteres Konzept aus der Kategorientheorie, welches in Haskell allgegenwärtig ist. Fürwahr, wir sind nicht in der Lage ein laufendes Programm in Haskell zu schreiben, ohne ein zumindest grundlegendes Verständnis von Monaden vorzuweisen. Monaden in Haskell sind ein weiterer Datentyp `type Monad :: (* -> *) -> Constraint`, welche eine Funktor-Instanz besitzen, also auch Funktoren sind, und zusätzlich eine Operation

`return :: a -> m a` und eine Operation `>>= :: m a -> (a -> m b) -> m b` besitzen. Die Operation `return` ist eine sogenannte „natürliche Transformation„, welche ein Objekt in eine Monade steckt. Wir werden nicht weiter darauf eingehen, was eine natürliche Transformation ist, jedoch ist es nützlich zu wissen, dass diese genau so wie Funktoren als eine Art von Abbildung zwischen Kategorien (in unserem Fall von **Hask** nach **Hask**) definiert werden können, welche gewisse Eigenschaften besitzen. Die Operation `>>=` ist eine sogenannte „Bindung„, welche es uns erlaubt „gefangene Werte“in einer Monade zu manipulieren.

Betrachten wir jedoch ein paar Beispiele, um uns diesem gefürchteten Konzept zu nähern und zu sehen, dass es gar nicht so angsteinflößend ist.

3.1 Listen als Monade

Wir haben bereits ein inniges Verständnis von Listen in Haskell in den vorherigen Abschnitten aufgebaut. Wir wissen, dass Listen eine Funktor-Instanz besitzen, welche uns erlaubt Listen zu manipulieren, indem wir Funktionen für die Unterliegenden Datentypen in der Liste definieren.

Wir möchten nun einen Schritt weiter gehen und uns anschauen, warum Listen auch eine kanonische Monaden-Instanz in Haskell besitzen, also eine Instanz, welche in der Standardbibliothek `Prelude` von Haskell definiert ist. Außerdem stellt sich die Frage, warum wir überhaupt eine Monaden-Instanz für Listen brauchen.

Um Listen zu einer Monade zu machen, müssen wir uns zuerst genauer ansehen, was eine Monade ist. Eine Monade ist ein Datentyp, dessen Definition in Abbildung 22 dargestellt ist.

```
type Monad :: (* -> *) -> Constraint
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  {-# MINIMAL (>>=) #-}
  -- Defined in 'GHC.Base'
```

Abbildung 22: Definition der „Monaden“Typenklasse.

Die Definition einer Monade ist hierbei genau so wie die in Abbildung 7 aufgeschrieben und sollte auch so verstanden werden. Wir haben hier jedoch eine Einschränkung, welche dadurch gegeben ist, dass hier `Applicative m => Monad m` steht. Dies stellt einen Type-Constraint da, welche und bereits bekannt aus dem Abschnitt über Funktoren sind. Dieser Type-Constraint besagt, dass ein Typ, welcher eine Monade ist, nur dann eine Monade sein kann, wenn dieser auch ein „Applicative“ist. Die Definition eines Applicatives ist in Abbildung 23 auffindbar. Relevant ist hierbei nur, dass auch ein Applicative einen Type-Constraint besitzt, welcher besagt, dass

jeder Typ, welcher eine Instanz von Applicative ist, auch eine Instanz von Functor sein muss. Die Operationen `<*>`, `GHC.Base.liftA2`, `*>` und `<*` sollten nach unserer Analyse der Funktor Typenklasse bereits verständlich sein, wobei ihr Verständnis keine Anforderung für das Verständnis von Monaden für den Rest dieses Abschnittes ist.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  GHC.Base.liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
    -- Defined in 'GHC.Base'
```

Abbildung 23: Definition der „Applicative“Typenklasse.

Wir nehmen nun an, dass Listen eine Instanz von Applicative sind. Dies ist auch der Fall, jedoch werden wir dies nicht beweisen. Der Beweis, dass Listen eine Instanz von Monaden sind wird dadurch geführt, indem wir konkrete Implementierungen für die Operationen `>>=` und `return` definieren.

Betrachten wir zunächst die Operation `return`, welche einen Wert in eine Monade steckt. Was dies bedeutet, ist bei Listen relativ selbsterklärend. Sei nun `x` ein Wert vom Typ `a`, dann ist `return x` eine Liste, welche nur den Wert `x` enthält. Dies stimmt auch mit der Typen-Signatur von `return` überein, welche `return :: a -> m a` ist.

Nun müssen wir nur noch die Operation `>>=` definieren. Diese Operation soll uns erlauben, einen Wert aus einer Monade zu ziehen, dann zu bearbeiten und dann zurück in die Monade zu legen. Sei nun `xs` eine Liste vom Typ `[a]` und `f` eine Funktion vom Typ `a -> b`. So können wir `(>>=) xs f = fmap f xs`, beziehungsweise kompakter `>>= = fmap` definieren. Dies ist uns bereits aus dem Abschnitt über Funktoren bekannt.

Wir sehen hiermit, dass Monaden es uns erlauben eine Art „Gefängnis“für unsere Werte zu schaffen, in welchem wir die Werte manipulieren können. Jedoch gibt es keine Operation mit der Typen-Signatur `get :: m a -> a`, welche uns erlaubt, einen Wert aus der Monade zu ziehen. Dies ist auch bei allgemeinen Monaden nicht möglich, wobei wir eine solche Funktion bei Listen relativ leicht wie in Abbildung 24 definieren können. Wichtig ist in diesem Beispiel, dass die Funktion nicht total ist, also nur für den Fall einer einelementigen Liste definiert ist. Sollte man eine Liste mit mehr oder weniger als einem Element übergeben, so wird eine Laufzeitfehler auftreten.

```haskell
get :: [a] -> a
get [x] = x
```

Abbildung 24: Eine Funktion, welche einen Wert aus einer Liste und somit aus einem Objekt, welches eine Monadenstruktur besitzt, herausnimmt.

3.2 Die IO Monade und Hello World in Haskell

Die wohl wichtigste Monade ist die `IO` Monade. Diese Monade erlaubt es uns, Eingaben und Ausgaben zu machen, woher auch der Name `IO` kommt. Ein wichtiges Beispiele für die Verwendung der `IO` Monade ist in Abbildung 25 zu finden. Hier definieren wir den Wert `main :: IO ()`, welcher der Start eines jeden Haskell Programms ist. Der Klarheit halber sei erwähnt, dass `IO` also mit dem Typen `()` als Parameter verwendet wird, welcher den Typen `Unit` in anderen Sprachen entspricht und lediglich ein leeres Tupel der Länge 0 darstellt.

```haskell
main :: IO ()
main = putStrLn "Hello World"
```

Abbildung 25: Hello World in Haskell

Zeigen wir eine Parallelität zwischen der `IO` Monade und der `[]` Monade, der Monade der Listen, auf um die Funktionalität der `IO` Monade zu verstehen. Wir können uns vorstellen, dass `IO` eine Art „Gefängnis“für Werte ist, in diesem Fall von `IO ()` für leere Tupel, genau so wie `[a]` eine Art „Gefängnis“für Werte vom Typ `a` ist. Allerdings gibt es einen bedeutenden Unterschied zwischen den beiden Monaden. Während wir in `[a]` wissen, was genau in unserem „Gefängnis“ist, so wissen wir dies bei `IO` nicht. In `[Int]` befindet sich für ein Element `[1,2,3]` beispielsweise nur die Zahl 1, die Zahl 2 und die Zahl 3 in dem Gefängnis. In dem Typen `IO ()` gibt es konstruktionsgemäß, weil der Typ `()` nur das Element `()` besitzt, nur das Element `IO ()`. Dieses Element kann jedoch neben sich in seinem „`IO` Gefängnis“noch weitere Sachen sitzen haben. Das einzige was unser „`IO` Gefängnis“garantieren muss ist, dass der Binding-Operator `>>=` und der `return` Operator ihre Rechenregeln beibehalten müssen. An dem Beispiel von Hello World in Abbildung 25 können wir uns dies gut vorstellen. Lässt man dieses Programm laufen, so wird die Ausgabe `Hello World` auf dem Bildschirm erscheinen. Wir haben somit einen Seiteneffekt beim ausführend unseres Programmes, also beim „Aufrufen des Wertes `main`“erzeugt.

Nun gibt es verschiedene philosophische Herangehensweisen, wie wir dies Interpretieren können. Wir können uns beispielsweise vorstellen, dass in unserem „`IO` Gefängnis“der Wert `()` nun zusammen mit dem Seiteneffekt, welcher „`Hello World`“auf dem Bildschirm ausgibt, sitzt. Dies bedeutet, wenn der Wert `main` aufgerufen wird, so wird das gesamte „`IO` Gefängnis“betrachtet, was den Seiteneffekt anzeigt.

Das wichtige hierbei ist jedoch, dass wir nun eine rigorose Theorie für Seiteneffekte haben, welche wir in Haskell nun nutzen können um mit Seiteneffekten, wie Beispielsweise der Ausgabe auf dem Bildschirm, umzugehen.

Möchte man Beispielsweise mehrere Seiteneffekte in seinem Programm haben, so können wir den uns bekannten Binding-Operator `>>=` verwenden. In Abbildung 26 sehen wir ein Beispiel, welches zwei Seiteneffekte in den Wert `main` einbindet. Lässt man dieses Programm nun laufen, so wird zuerst `Hello` auf dem Bildschirm ausgegeben und anschließend `World`.

```
main :: IO ()
main = print "Hello" >>= \x -> print "World"
```

Abbildung 26: Hello World in Haskell, indem wir die Monadenstruktur durch den Binding-Operator ausnutzen.

Zuallerletzt können wir auch den `return` Operator nutzen, um unser Programm nichts machen zu lassen. In Abbildung 26 sehen wir ein Beispiel dafür. Das Konkatinieren von Seiteneffekten zu neuen Seiteneffekten und die Existenz eines Seiteneffekts, welcher nichts macht, also der „nichts-machende Seiteneffekt„`return ()` vom Typen `IO ()`, sind die Eigenschaften, welche intuitiv für Seiteneffekte klar sind und Seiteneffekte ausmachen. Und diese sind genau die Eigenschaften, welche wir mit der Monadenstruktur erfassen.

Eine interessante Folgerung, welche aus dem vorherigen Paragraphen klar erkenntlich ist, ist hierbei dass Monaden ein Monoidobjekt in der Kategorie der Endofunktoren sind. Wir werden diesen Satz nicht beweisen, jedoch in den Zusammenhang mit der Kategorie **Hask** und der Monade `IO` bringen. Ein Objekt besitzt eine Monoidstruktur, wenn man je zwei Elemente daraus zu einem dritten Element daraus verknüpfen kann. Für die Monade `IO` übernimmt diese Rolle der Verknüpfung der Binding-Operator. Dieser muss zudem noch assoziativ sein, was intuitiv für den Binding-Operator klar ist, da es das selbe ist, wenn a, b, c drei Seiteneffekte sind, ob man zuerst a gefolgt von b mit c ausführt oder zuerst a mit b gefolgt von c ausführt. Die letzte Eigenschaft welche für eine Monoidstruktur benötigt wird ist die Existenz eines neutralen Elements, welches also nichts macht. Dieses haben wir bereits im vorherigen Paragraphen konstruiert, nämlich den Wert `return ()` vom Typen `IO ()`.

3.3 Warum Monaden? Oder: Warum Haskell?

Schlussendlich stellt sich jedoch die Frage, warum wir einen solchen Aufwand betreiben, um beispielsweise mit Seiteneffekten umzugehen. Die Antwort ist jedoch ganz einfach. Wir haben am Anfang damit begonnen, einfache Typen zu erstellen. Stets waren wir bemüht, die Theorie, welche wir aufbauen rigoros und wohldefiniert zu halten. Haben wir eine Struktur gefunden, so haben wir versucht dieser einen Namen zu

geben und diese zu verallgemeinern, ganz im Sinne der „Modularität“und „Ausdrucksfähigkeit“, so wie es in [2] empfohlen wird. Speziell haben wir dies bei der `Functor` Typenklasse gesehen, welche das Konzept „Abbildungen zwischen Strukturen können auf ähnliche Abbildungen auf ähnlichen Strukturen geschickt werden„darstellt. Nun war es am Ende eigentlich nur ein glücklicher Zufall, dass unsere Struktur der Monaden genau die Eigenschaften, welche Seiteneffekte ausmachen, besitzt.

Die Bedeutung letztendlich ist, dass wir in Haskell genau wissen möchten, womit wir arbeiten. Während man sich in anderen Programmiersprachen wie Python oder Java damit begnügt, dass Seiteneffekte existieren und immer auftreten können, selbst wenn wir explizit angeben, dass unsere Funktion einen Wert von Typ a einnimmt und einen Wert von Typ b zurückgibt, kann es immer noch passieren, dass die Funktion plötzlich als Seiteneffekt etwas anderes macht. In Haskell wissen wir jedoch, dass wenn wir eine Typensignatur angeben, die Funktion auch genau diesen Typ hat[5].

Genau so, wie wir sagen könnten „uns ist egal was ein 'Körper‘ ist, hauptsache wir wissen was 1,2,3, . . . sind„, könnten wir auch sagen „uns ist egal was eine Monade ist, hauptsache wir wissen dass es Seiteneffekte gibt„. Wir können auch ohne das nötige Hintergrundwissen Programmieren, jedoch können wir deutlich eleganteren und präziseren Code schreiben, wenn wir die den Problemen grundlegenden Strukturen kennen.

Haskell hilft uns dadurch besser zu programmieren, indem es uns dazu zwingt diese Konzepte zu verstehen und umzusetzen. Hier können wir nun auch sehen, warum Haskell zwar sehr eleganten und effizienten Code erzeugen kann, jedoch in der Industrie nicht weit verbreitet ist. Oft ist es einfacher und schneller, eine „quick and dirty“Lösung zu finden welche „einfach funktioniert“, als sich mit den Grundlagen seiner Probleme auseinanderzusetzen. Insbesondere ist die Hürde für die Einarbeitung in Haskell sehr hoch, was bei anderen populären Sprachen wie Python oder JavaScript nicht der Fall ist.

Schlussendlich ist es für jeden Programmierer ratsam in seinem Abenteuer durch die Welt der Programmiersprachen auch einen Zwischenhalt bei Haskell einzulegen. Die Konzepte welche in Haskell behandelt werden, können den Programmierstil in anderen Sprachen, welche der Programmierer verwendet, positiv beeinflussen. Auch wenn der Programmierer nicht unbedingt Haskell am Ende in Production verwendet, so bieten die Ideen, welche Haskell vermittelt, allein schon genügend Anreiz zum erlernen der Sprache.

Anhang

Abbildungsverzeichnis

1 Definition eines Typens genannt „MyType“mit Konstruktor „MyConstruktor“, welcher einen „Bool“und einen „Int“speichert [5] 4

[5] außer bei partiellen Funktionen, welche Definitionslückene aufweisen, oder nicht-terminierenden Funktionen

2 Definition einer Abbildung, welche einen Wert von Typ „MyType“auf einen Wert von Typ „Bool“schickt. 5
3 Definition einer Typenklasse, welche einen Typen „a“als Parameter besitzt. 5
4 Definition einer Typenklasse, welche einen Typen „a“als Parameter besitzt. 5
5 Definition einer pathologischen Typenklasse, welche für jeden Typen „a“als Parameter ohne Einschränkung implementiert werden kann. . . 6
6 Definition einer Typenklasse, welche einen Typen „a“als Parameter besitzt. 6
7 Definition der „Functor“Typenklasse. 6
8 Typensignaturen von Funktionen, welche äquivalent bezüglich curryings sind 8
9 Kurzschreibweise resultierend aus dem Currying basierend basierend auf der Definition von f 8
10 Kurzschreibweise resultierend aus dem Currying basierend basierend auf der Definition von g 9
11 So binden Sie Code ein 9
12 Wir komponieren die map Funktion geschickt mit einer partiellen Anwendung der gecurryten Funktion + um eine Funktion zu erhalten, welche alle Elemente einer Liste von Zahlen um 1 zu erhöhen. 9
13 Anwendung der kanonischen Funktor Instanz auf der Liste von ganzen Zahlen z mit $0 \leq z < 10$, wobei jede Zahl in der Liste um eins erhöht wird. 10
14 Der Maybe Datentyp und wie man eine einfachere Funktion mit diesem schreibt. 10
15 Naive Definition von „erhoeheMaybe“ 11
16 Durchdachtere Definition von „erhoeheMaybe“ 11
17 Typensignatur von „erhoeheMaybe“nach einem Refactoring. „erhoeheMaybe“ist nun auf allen „f“anwendbar, welche Funktoren sind und einen Typen „b“, welcher eine Zahl ist, im ersten Parameter haben. . 11
18 Anwendung von „erhoeheMaybe“auf Listen von Zahlen und auf „Maybe Int“. 12
19 Beispiel der grundlegenden Rechenregel für die Funktor-Instanz von „Maybe“. 13
20 kommutatives Diagramm, welches die Eigenschaften von „fmap“darstellt. 13
21 Implementation von F(g :: Maybe Int $\rightarrow$ a), wobei h die eindeutig von f induzierte Funktion h :: Just Int $\rightarrow a$ ist. 14
22 Definition der „Monaden“Typenklasse. 15
23 Definition der „Applicative“Typenklasse. 16
24 Eine Funktion, welche einen Wert aus einer Liste und somit aus einem Objekt, welches eine Monadenstruktur besitzt, herausnimmt. 17
25 Hello World in Haskell 17

26 Hello World in Haskell, indem wir die Monadenstruktur durch den Binding-Operator ausnutzen. 18

Literatur

[1] Andrea Asperti, Giuseppe L.: *CATEGORIES TYPES AND STRUCTURES: An Introduction to Category Theory for the working computer scientist.* M.I.T. PRESS, 1991

[2] Harold Abelson, Julie S. Gerald Jay Sussman S. Gerald Jay Sussman: *Struktur und Interpretation von Computerprogrammen: Eine Informatik-Einführung.* Springer-Lehrbuch, 1998

[3] Henderson, Christopher: Generalized abstract nonsense: Category theory and adjunctions / Technical report, University of Chicago. 2008. – Forschungsbericht

[4] Offizielle Rust Dokumentation: *Module std::option.* https://doc.rust-lang.org/std/option/, 2023. – [Online; accessed 14-January-2023]

[5] Offizielles Haskell Wiki: *Constructor - HaskellWiki.* https://wiki.haskell.org/Constructor, 2023. – [Online; accessed 14-January-2023]

[6] Offizielles Haskell Wiki: *Functor - HaskellWiki.* https://wiki.haskell.org/Functor, 2023. – [Online; accessed 14-January-2023]

[7] Offizielles Haskell Wiki: *Kind - HaskellWiki.* https://wiki.haskell.org/Kind, 2023. – [Online; accessed 14-January-2023]

[8] Philip Wadler, Paul H. u.: A history of Haskell: being lazy with class. In: *HOPL III* (2007). http://dx.doi.org/https://doi.org/10.1145/1238844.1238856. – DOI https://doi.org/10.1145/1238844.1238856