Sven Eric Panitz

Lehrbriefe
Funktionale Programmierung

6 Zustand in Haskell

Zustand in Haskell

Sven Eric Panitz

9. Juni 2025

Inhaltsverzeichnis

1 Zustandsberechnungen

Ja es gibt primär in diesem Lehrbrief um Zustandsberechnungen[**?**][**?**], aber wir wollen ehrlich sein: es geht auch um Monaden, ein Kapitel in Haskell, das als notorisch schwer gilt. Die Monaden sind insbesondere für IO Operationen in Haskell eingeführt worden[**?**].

In der imperativen Programmierung sind wir veränderbare Zustände als allgegenwärtig gewohnt. Es gibt globale Variablen, auf die jede Anweisung zugreifen kann und deren Werte von überall verändert werden können, indem neue Werte in diesen Variablen gespeichert werden.

Dieses geht in Haskell nicht. Es gibt gar keine Anweisungen im klassischen Sinne. Es gibt nur Ausdrücke, die zu einem Wert auswerten, unabhängig von einem globalen Zustand.

Mit dem Fehlen von Anweisungen fehlt nicht nur die Zuweisung sondern auch sämtliche Kontrollstrukturen wie Schleifen und die klare Sequenzierung von Anweisungen. In imperative Sprache geht stets alles der Reihe nach. Zeile für Zeile wird in einer festgelegten Sequenz eine Folge von Anweisungen abgearbeitet. In Haskell gibt es dieses nicht. Es gibt keine Sequenzierung. Insbesondere die lazy Auswertungsstrategie macht es fast unmöglich zu sagen, in welcher Reihenfolge Teilausdrücke ausgewertet werden. Das ist für einen veränderbaren Zustand natürlich tödlich, denn hier kommt es darauf an, in welcher Reihenfolge die Schreib- und Leseanweisungen auf dem veränderbaren Zustand ausgeführt werden.

Trotzdem ist auch für einen Haskellprogrammierer ein Programmierparadigma mit einem veränderbaren Zustand manchmal wünschenswert. Dieses fällt insbesondere auf, wenn man eine imperative Bibliothek, die als eine Zustandsmaschine entworfen ist, auf Haskell portieren möchte. Eine solche Bibliothek ist insbesondere OpenGL.

In diesem Lehrbrief zeigen wir, wie man auch in einer rein funktionalen Welt, eine Zustandsbehaftetes Paradigma implementieren kann.

Schreiben wir also ein kleines Modul zur Arbeit mit Zuständen:

Zustand.lhs

```
> module Zustand where
```

Wir werden im hinteren Teil ein paar Standardfunktionen für die Transformation von Monaden benötigen:

Zustand.lhs

```
> import Control.Monad.Trans.Maybe
> import Control.Monad.Trans.Class
> import Control.Monad
> import Control.Monad.IO.Class
```

Da es in Haskell keinen globalen Zustand gibt, muss eine Funktion einen Zustand, den sie bei der Auswertung berücksichtigen soll, als Argument erhalten. Da in Haskell keine veränderbaren Variablen existieren, muss eine Funktion, die einen Zustand verändern soll, neben ihrem eigentlichen Ergebnis den veränderten neuen Zustand als Funktionsergebnis haben.

Eine Anweisung, die einen globalen Zustand hat, ist somit eine Funktion, die für einen Eingabezustand ein Ergenis berechnet und einen neuen veränderten Zustand liefert. Somit ist es eine Funktion `st -> (st,r)`, wobei `st` eine Typvariable für den Zustand (*State*) und `r` eine Typvariable für das Ergebnis der Anweisung ist. Wir definieren solche Funktionen als den Typ `Anweisung`.

Zustand.lhs

```
> newtype Anweisung st r =  Z (st -> (st,r))
```

Wenn der Zustand eine Liste von Paaren aus String und Integer-Werten ist, dann lässt sich eine Anweisung schreiben, die in die Umgebung für eine Variable einen neuen Wert einträgt, als:

Zustand.lhs

```
> setvar :: String -> Integer -> Anweisung [(String,Integer)] Integer
> setvar x v = Z (\env -> ((x,v):env,v))
```

Das soll jetzt einer Zuweisung in imperativen Sprachen entsprechen. Statt `x = 42` in einer imperativen Sprache, schreiben wir `setvar "x" 42` und sprechen dabei von einer Anweisung.

Mit der Funktion `lookup` aus dem Standard-Prelude lässt sich aus der Umgebung auch wieder für eine dort gespeicherte Variable der Wert auslesen. Das Ergebnis ist in unserem Fall vom Typ `Maybe Integer`, je nachdem ob für die Variable eine Wert n in der Umgebung gespeichert war (dann ist es `Just n`) oder eben nicht (dann ist es `Nothing`).

Zustand.lhs

```
> getvar :: String -> Anweisung [(String,Integer)] (Maybe Integer)
> getvar x = Z (\env -> (env,lookup x env))
```

Damit können wir Variablen in die Umgebung speichern und aus ihr wieder lesen.

Variablen setzen und sie wieder verwenden, nun haben wir eigentlich alles, was eine imperative Sprache ausmacht. Betrachten wir eine entsprechende Java-Methode:

Zustand.java

```
class C{
  static int f(){
    var x = 17;
    var y = 4;
    var z = 2;
    return (x+y)*z;
  }
}
```

Wie können diese jetzt versuchen mit dem Datentyp `Anweisung` nachzubauen. Erst haben wir drei Anweisungen, die eine Variable speichern und dann greifen wir mit drei Anweisungen auf diese wieder zu. So lassen sich sechs Anweisungen definieren:

Zustand.lhs

```
> ex1 =
>    let
>      Z anw1 = setvar "x" 17
>      Z anw2 = setvar "y" 4
>      Z anw3 = setvar "z" 2
>      Z anw4 = getvar "x"
>      Z anw5 = getvar "y"
>      Z anw6 = getvar "z"
```

Durch diese sechs Anweisungen ist jetzt ein Zustand, der durch die Anweisungen verändert werden kann, durchzureichen. Der durch eine Vorhergehende Anweisung erzielte Zustand wird die Eingabe für die nächste Anweisung. Der durchgereichte Zustand führt also zu Sequentierung der Anweisungen. Die erste Anweisung wird auf die leere Liste, in der noch keine Variable gespeichert wurde, ausgeführt. Die zweite auf das Zustandsergebnis der ersten, die dritte auf das Zustandsergebnis der zweiten usw.

Zustand.lhs

```
>      (st1,_) = anw1 []
>      (st2,_) = anw2 st1
>      (st3,_) = anw3 st2
>      (st4,Just x) = anw4 st3
>      (st5,Just y) = anw5 st4
>      (st6,Just z) = anw6 st5
>    in (st6,(x+y)*z)
```

Das Durchreichen des Zustand ist etwas umständlich, gewährt aber die Sequenzierung. Die Sequenzierung der Anweisungen ist gewährleistet, weil kein Zustand mehr als einmal als Argument für eine Anweisung verwendet wird. In der Programmiersprache Clean hat man für Zustände, die sequentiell durchgereicht werden eine Typannotation verwendet und dann statisch geprüft, dass solche Argumente nicht mehrfach weitergereicht werden. Man bezeichnet es dort als *unique types*[**?**].

Soweit kann man also selbst seinen Zustand sichtbar durchreichen. Es wirkt aber insbesondere umständlich, wenn der Zustand gar nicht verändert wird. Es geht hauptsächlich darum, auszudrücken, in welcher Sequenz die Anweisungen ausgeführt werden sollen.

Wir können hierfür eine Funktion vorsehen, die aus zwei Anweisungen die Sequenz zweier Anweisungen liefert und definieren eine Kombinatorfunktion `undDann`, die die Sequenz von Anweisungen ausdrückt.

Zustand.lhs

```
> undDann:: (Anweisung st r1) -> (Anweisung st r2) -> (Anweisung st r2)
> (Z anw1)  `undDann` (Z anw2)
>   = Z (\s -> let (s1,r1) = anw1 s
>                  (s2,r2) = anw2 s1
>              in (s2,r2) )
```

Mit dieser Kombinatorfunktion lassen sich die ersten drei Anweisungen zu einer Sequenz aus diesen Anweisungen direkt zusammenfassen:

Zustand.lhs

```
> ex2 =
>    let
>      Z anw123 =
>        setvar "x" 17  `undDann`
>        setvar "y" 4   `undDann`
>        setvar "z" 2
>      Z anw4 = getvar "x"
>      Z anw5 = getvar "y"
>      Z anw6 = getvar "z"
>      (st3,_) = anw123 []
>      (st4,Just x) = anw4 st3
>      (st5,Just y) = anw5 st4
>      (st6,Just z) = anw6 st5
>    in (st6,(x+y)*z)
```

Die ersten Anweisunge werden mit `undDann` verknüpft, ohne dass wir die Zwischenzustände noch zu sehen bekommen.

Bei den Anweisungen, die ein Ergebnis erzeugen, an dem wir interessiert sind, reicht die Kombinatorfunktion `undDann` nicht, denn sie ignoriert das Ergebnis der ersten Anweisung und es gibt keine Möglichkeit, dieses Ergebnis in der zweiten Anweisung zu verwenden. Hierfür können wir uns eine zweite Kombinatorfunktion definieren. Bei dieser soll die zweite Anweisung im Abhängigkeit des Ergebnisses der ersten Anweisung sein.

Zustand.lhs

```
> undMit::(Anweisung st r1)->(r1->Anweisung st r2)->(Anweisung st r2)
> (Z st1)  `undMit` fs
>   = Z (\s -> let (s1,r1) = st1 s
>                  (Z anw2) = fs r1
>              in anw2 s1)
```

Manchmal braucht man eine Anweisung, die direkt unabhängig vom Zustand ein Ergebnis liefert. Auch das können wir in einer Funktion ausdrücken.

Zustand.lhs

```
> ergebnis :: r -> Anweisung st r
> ergebnis r = Z (\st -> (st,r))
```

Jetzt lässt sich die Sequenz der sechs Anweisungen fast wie ein imperatives Programm schreiben.

Zustand.lhs

```
> ex3 =
>    let
>      Z anws =
>        setvar "x" 17 `undDann`
>        setvar "y" 4  `undDann`
>        setvar "z" 2  `undDann`
>        getvar "x"    `undMit` \(Just x) ->
>        getvar "y"    `undMit` \(Just y) ->
>        getvar "z"    `undMit` \(Just z) ->
>        ergebnis ((x+y)*z)
>   in anws []
```

Auf magische Weise ist der Zustand komplett verschwunden, oder er ist zumindest unsichtbar geworden und wird unter der Hand von den beiden Kombinatorfunktionen `undDann` und `undMit` weitergereicht oder gar verändert.

Noch eleganter wird das Programm, wenn wir uns daran erinnern, dass in Haskell beliebige Operatoren definiert werden können. Wir können die beiden Kombinatorfunktionen durch Operatoren ausdrücken.

Zustand.lhs

```
> infixl 1 -->
> infixl 1 ->-

> (-->) = undDann
> (->-) = undMit
```

Jetzt wird die Sequenz der Anweisungen auf den Zustand noch einmal sichtbarer:

Zustand.lhs

```
> ex4 = let
>   Z anws =
>     setvar "x" 17 -->
>     setvar "y" 4  -->
>     setvar "z" 2  -->
>     getvar "x"    ->- \(Just x) ->
```

```
>     getvar "y"    ->- \(Just y) ->
>     getvar "z"    ->- \(Just z) ->
>     ergebnis ((x+y)*z)
>   in anws []
```

Warum nicht auch der Funktion `setvar` einen Operator spendieren?

Zustand.lhs

```
> (=:) = setvar
```

In Haskell lassen sich nur zweistellige Operatoren definieren. Der Zugriff auf eine Variable der Umgebung mit der Funktion `getvar` ist eine einstellige Funktion. Es gibt in Haskell aber den etwas künstlichen Typ `()` des nullstelligen Tupels, der nur den Wert `()` hat. Mit diesem können wir auch einen Operator für den Variablenzugriff definieren als.

Zustand.lhs

```
> x ! () = getvar x
```

Dann wird das Programm noch einmal deutlicher in der Sequenz von Setzen und Lesen der Variablen

Zustand.lhs

```
> ex5 = let
>   Z anws =
>     "x" =: 17 -->
>     "y" =: 4  -->
>     "z" =: 2  -->
>     "x"!()    ->- \(Just x) ->
>     "y"!()    ->- \(Just y) ->
>     "z"!()    ->- \(Just z) ->
>     ergebnis ((x+y)*z)
>   in anws []
```

Es bietet sich an, das Deklarieren eines Programms auf einem Zustand von der Ausführung zu trennen. Die Ausführung wird durch die Funktion `run` ausgedrückt.

Zustand.lhs

```
> run (Z stm) = stm []
```

Das auszuführende Programm ist dann die Sequenz von Anweisungen.

Zustand.lhs

```
> ex6 =
>   "x" =: 17 -->
>   "y" =: 4  -->
>   "z" =: 2  -->
>   "x"!()    ->- \(Just x) ->
>   "y"!()    ->- \(Just y) ->
>   "z"!()    ->- \(Just z) ->
>   ergebnis ((x+y)*z)
```

Das sieht schon fast aus wie ein imperatives Programm. Insbesondere der Zustand, der unter der Hand weitergereicht wird, ist komplett unsichtbar geworden. Die Sequenzierung der Anweisungen erzwingen die Operatoren `-->` und `->-`.

2 Monadische Implementierung

Was wir im letzten Abschnitt entwickelt haben, ist eine Struktur, die als Monade bezeichnet wird und wir können sie in die Monadenbibliothek von Haskell einbinden.

2.1 Monade

Monaden sind in Haskell eine Typklasse mit drei zu implementierenden Funktionen. Diese drei Funktionen haben wir bereits für unseren Typ `Anweisung st` umgesetzt.

Somit können wir die drei Funktionen `undDann`, `undMit` sowie `ergebnis` direkt nutzen für die Implementierung der Typklasse Monade. Da es Bestrebungen gibt, die Funktionen `(>>)` und `return` direkt über die Klasse `Applicative` zu implementieren, kommentieren wir die Definitionen für diese beiden Funktionen jedoch aus.

Zustand.lhs

```
> instance Monad (Anweisung st) where
>   -- (>>)   = undDann
>   -- return = ergebnis
>   (>>=)  = undMit
```

2.2 Funktor

Die Definition der Typklasse `Monad` verlangt, dass auch die Typklasse `Functor` implementiert wurde. Ein Funktor enthält die Funktion `fmap` mit der das Ergebnis einer Anweisung manipuliert werden kann. Sie ist die Verallgemeinerung der Funktion `map` auf Listen.

Für Anweisungen ist die Funktion `fmap` zu implementieren als:

Zustand.lhs

```
> instance Functor (Anweisung st) where
>   fmap f (Z st) = Z (\s -> let (s1,r1) = st s in (s1,f r1))
```

2.3 Applicative

Es wird für jesen Typ der eine Monade ist, auch verlangt die Typklasse `Applicative` zu implmentieren. Wir werden davon zwar vorerst für unsere Anweisungen keinen Gebrauch machen, geben aber hier die Umsetzung an:

Zustand.lhs

```
> instance Applicative (Anweisung st) where
>   pure  = ergebnis
>   (Z gf) <*> (Z ga)
>     = Z (\s -> let (s1,f) = gf s
>                    (s2,a) = ga s1
>                in (s2,f a))
```

Im Prinzip wurden nur die Operatoren umbenannt, so dass unser Beispiel jetzt geschrieben werden kann als:

Zustand.lhs

```
> ex7 =
>   "x" =: 17  >>
>   "y" =: 4   >>
>   "z" =: 2   >>
>   "x"!()     >>= \(Just x)->
>   "y"!()     >>= \(Just y)->
>   "z"!()     >>= \(Just z)->
>   return ((x+y)*z)
```

Man beachte bei diesem Programm, dass in den λ-Ausdrücken zum Extrahieren der Variablen wie in `"x"!()  >>= \(Just x)->`, das Pattern `(Just x)` fest davon ausgeht, dass die erfragte Variable im globalen Zustand gespeichert ist. Eine Alternative für `Nothing` ist nicht vorgesehen und würde zum Abbruch des Programms führen.

2.4 Do-Notation

Es gibt für alle Typen, die die Typklasse `Monad` implementieren, in Haskell eine bestimmte Syntax. Eine solche Syntax wird gerne auch aus syntaktischer Zucker bezeichnet. Das kennt man

auch aus anderen Sprachen. In Java ist die foreach-Schleife syntaktischer Zucker für alle Typen, die die Schnittstelle `Iterable` implementieren.

Der syntaktische Zucker für Monaten in Haskell ist die do-Notation. Sie wird eingeleitet von dem Schlüsselwort `do` und anschließend kommen untereinander die Ausdrücke vom Typ der Monade. Ist man an das Ergebnis der Monade interessiert, so kann man dieses an eine neue Variable binden, indem man diese dem monadichen Ausdruck voran stellt und mit dem stilisierten Elementsymbol `<-` vor den monadischen Ausdruck stellt. Links vom Symbol `<-` darf allerdings nur eine Variable stehen, kein beliebiges Pattern.

Dafür können in der Sequenz der monadischen Ausdrücke beliebige Bindungen mit `let` vorgenommen werden.

In der do-Notation präsentiert sich unser Beispiel als:

Zustand.lhs

```
> ex8 = do
>   "x" =: 17
>   "y" =: 4
>   "z" =: 2
>   mx <- "x"!()
>   let (Just x) = mx
>   my <- "y"!()
>   let (Just y) = my
>   mz <- "z"!()
>   let (Just z) = mz
>   return ((x+y)*z)
```

Das Auspacken der erfolgreichen Abfrage von Variablenwerten aus dem Typ `Maybe` muss nun in eigenen `let`-Bindungen geschehen. Dieses kann weiterhin fehlschlagen. Dieses werden wir im nächsten Abschnitt koordinierter behandeln.

2.5 Monadentransformer

2.5.1 Die Monade `Maybe`

Ein Blick in das Prelude zeigt, dass auch der Datentyp `Maybe` die Typklasse `Monad` implementiert. Somit steht auch für den Typ `Maybe` die do-Notation zur Verfügung.

Zustand.lhs

```
> eval1 env = do
>   x <- lookup "x" env
>   y <- lookup "y" env
>   z <- lookup "z" env
>   return ((x+y)*z)
```

Die Verwendung von `Maybe` als Monade in der do-Notation, sorgt dafür, dass auch die Fehlerwerte, also Ausdrücke die zu `Nothing` auswerten, sauber durchgereicht werden und nicht zu einem Programmabbruch führen.

Zustand

```
*Zustand> eval1 [("x",17),("y",4),("z",2)]
Just 42
*Zustand> eval1 [("x",17),("y",4)]
Nothing
*Zustand> eval1 [("x",17),("z",2)]
Nothing
```

Das ist übrigens unabhängig davon, ob die gelesene Variable gebraucht wird oder nicht. Hier schlägt lazy Auswertung eben nicht zu, weil wir mit den Monaden ja explizit eine Sequenzierung erzwungen haben.

Das folgende Beispiel errechnet kein Ergebnis.

Zustand.lhs

```
> bsp1 = do
>   x <- lookup "x" [("y",42),("x",17)]
>   y <- lookup "y" [("y",42),("x",17)]
>   z <- lookup "z" [("y",42),("x",17)]
>   return (x+y)
```

Obwohl die Variable `z` für das Ergebnis nicht verwendet wird, wertet das Beispiel zu `Nothing` aus.

Zustand

```
*Zustand> bsp1
Nothing
```

Für unser Beispiel können wir jetzt nutzen, dass auch `Maybe` eine Monade ist, indem wir die beiden Monaden mit jeweils einem eigenen do-Ausdruck behandeln. Erst gibt es die Anweisungen auf unserem Zustand, und dann werden in der zweiten do-Notation die dort erhaltenen `Maybe`-Werte weiterverarbeitet.

Zustand.lhs

```
> ex9 = do
>   "x" =: 17
>   "y" =: 4
>   "z" =: 2
```

```
>   mx <- "x"!()
>   my <- "y"!()
>   mz <- "z"!()
>   return $ do
>     x<-mx
>     y<-my
>     z<-mz
>     return ((x+y)*z)
```

Dieses Programm bricht jetzt nicht mehr ab, sollte eine Variable gelesen werden, die nicht in der Umgebung vorher gespeichert wurde.

2.5.2 Der `Maybe`-Transformer

Im letzten Abschnitt landeten wir bei zwei unterschiedlichen monadischen Typen. Zum einen unserem eigenen Typen `Anweisung` zum anderen dem Standardtypen `Maybe`. Beide Implementieren zwar die Typklasse `Monad` haben aber sonst überhaupt nichts miteinander zu tun, und können deshalb auch nicht gemeinsam verbunden werden, weder mit den Operatoren `(>>)` und `(>>=)` noch über die do-Notation.

Deshalb brauchten wir zum Schluss zwei getrennten do-Ausdrücke. Je einen für die beiden monadischen Typen.

Das ist schade. Schöner wäre, wenn wir die beiden Monaden irgendwie zu einer vereinigen könnten. Was wir suchen ist eine dritte Monade, auf der wir rechnen können wie mit der Monade `Maybe`, die aber auch die Monade `Anweisung` in sich vereinigen kann.

Wir wollen unsere Monade `Anweisung` so transformieren, dass mit ihren Werten wie auf den Werten der `Maybe`-Monade gerechnet werden kann.

Für solche Situationen lassen sich sogenannte Transformer-Monaden definieren. Diese haben im Standard immer einen Namen der mit einem großen `T` endet. Wenn wir in einer Monade wie unserer Monade `Anweisung` mit `Maybe`-Werten rechnen wollen, dann können wir die Anweisung mit der Monade `MaybeT` transformieren.

Diese ist deiniert als:

Zustand.lhs

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

`m` steht dabei für den Monadentyp in unserem Fall `Anweisung st` und `a` steht für den Elementtyp, in unserem Fall ist das jeweils `Integer`. Einen Ausdruck `e` vom Typ `Anweisung st (Maybe Integer)`, solche liegen bei uns mit `"x"!()` vor, lässt sich mit dem Konstruktor `MaybeT` zu einem Ausdruck des Typs `MaybeT (Anweisung st) Integer` transformieren.

Für einen Ausdruck des Typs `Anweisung st Integer`, solche liegen bei uns mit `"x" =: 17` vor, müssen wir zunächst aus dem Element einen Wert des Typs `Maybe Integer` machen. Dieses lässt sich mit der Anwendung der Funktorfunktion durchführen: `fmap Just`.

Wenn wir unsere Monade auf die `MaybeT` Monade transformieren, dann kommen wir in unserem Beispiel mit nur einem do-Ausdruck aus:

Zustand.lhs

```
> ex10 :: MaybeT (Anweisung [(String,Integer)]) Integer
> ex10 = do
>   MaybeT$fmap Just ("x" =: 17)
>   MaybeT$fmap Just ("y" =: 4)
>   MaybeT$fmap Just ("z" =: 2)
>   xm <- MaybeT ("x"!())
>   ym <- MaybeT ("y"!())
>   zm <- MaybeT ("z"!())
>   return  ((xm+ym)*zm)
```

Um diesen Ausdruck nun auszuwerten, ist zunächst die Funktion aus der Monade `MaybeT` auszupacken, und diese dann zu starten:

Zustand

```
*Zustand> run (runMaybeT ex10)
([("z",2),("y",4),("x",17)],Just 42)
```

Für monadische Typen, die als Transformertypen gedacht sind, das obige `MaybeT` soll ja eine beliebige Monade in die `Maybe` Funktionalität bringen, für solche Typen existiert eine weitere Typklasse, die Klasse `MonadTrans`.

In dieser ist die Funktion `lift` definiert, mit der Ausdrücke, die keinen `Maybe`-Wert enthalten auf die Transformer-Moade gehoben werden. In unserem Beispiel ersetzt die Funktion `lift` den Aufruf von `(MaybeT.fmap Just)`, sodass unser Beispiel schließlich geschrieben werden kann als:

Zustand.lhs

```
> ex11 :: MaybeT (Anweisung [(String,Integer)]) Integer
> ex11 = do
>   lift ("x" =: 17)
>   lift ("y" =: 4)
>   lift ("z" =: 2)
>   xm <- MaybeT ("x"!())
>   ym <- MaybeT ("y"!())
>   zm <- MaybeT ("z"!())
>   return  ((xm+ym)*zm)
```

2.6 Beispiele weiterer Monaden

Wir haben bisher drei Monaden gesehen. Die Monade `Maybe`, die Transformer-Monade `MaybeT` und unsere eigene Monade für Anweisungen, die einen Zustand bearbeiten: `Anweisung`.

Monaden sind in Haskell allgegenwärtig und insbesondere in real life programming mit viel Interaktion, viel IO und bei Webprogrammierung geht es nicht ohne.

2.6.1 IO

Der Hauptgrund für Monaden in Haskell war, ein gutes Modell für IO-Operationen zu haben. In den ersten Haskellversionen war dieses mehr als umständlich umgesetzt.

Seither sind alle IO-Operationen in der IO-Monade gekapselt. Sie können nur innerhalb der Monade ausgeführt werden. Sei es im Rahmen einer do-Notation oder direkt durch Kombination mit den Operatoren >> und >>=.

Hier ein kleines Beispiel für ein paar IO-Operationen. Eine Ausgabe auf der Konsole wird gemacht, eine Zahl von der Konsole eingelesen, weitere Ausgaben, dann eine Datei gelesen und schließlich eine Kopie der Datei in eine andere Datei geschrieben:

Zustand.lhs

```
> bsp2 = do
>   print "geben sie eine Zahl ein"
>   i <- (readLn::IO Integer)
>   print "das Quadrat ist"
>   print (i*i)
>   xs <- readFile "Zustand.lhs"
>   writeFile "newZust.hs"  xs
```

IO lebt immer in der IO-Monade. Wenn wir IO-Operationen in anderen Berechnungen integrieren wollen, brauchen wir wieder Monaden-Transformer. Dieses werden wir in der abschließenden Aufgabe illustrieren.

2.6.2 Listen

Auf dem ersten Blick etwas überraschend, bei Betrachtung eines Beispiels sehr naheliegend, auch die Standardlisten in Haskell implementieren die Typklasse `Monad`.

So lassen sich auch Listen mit der do-Notation verarbeiten. Hier ein kleines Beispiel.

Zustand.lhs

```
> bsp3 = do
>   x <- [1,2,3,4]
>   y <- [5,6,7]
>   return (x,y)
```

Was macht dieser Ausdruck? Schauen wir uns das Ergebnis einmal an:

Zustand

```
Zustand> bsp3
[(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),(3,7),(4,5),(4,6),(4,7)]
```

Es werden also alle x-y-Paare gebildet. Dieses erinnert sehr an ein anderes syntaktisches Konstrukt für Listen, der Mengenschreibweise. Den gleichen Ausdruck kann man analog mit der Mengenschreibweise schreiben:

Zustand

```
*Zustand> [(x,y)|x<-[1,2,3,4],y<-[5,6,7]]
[(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),(3,7),(4,5),(4,6),(4,7)]
*Zustand> bsp3 == [(x,y)|x<-[1,2,3,4],y<-[5,6,7]]
True
```

Fehlen eigentlich nur noch die Bedingungen, die aus den generierten Listenelementen wieder filtern, so wie beispielweise im folgenden Ausruck verwendet:

Zustand

```
*Zustand> [(x,y)|x<-[1,2,3,4],y<-[5,6,7],x`mod`2==0]
[(2,5),(2,6),(2,7),(4,5),(4,6),(4,7)]
```

Auch innerhalb der do-Notation lässt sich dieser Filter integrieren. Hierbei hilft die Standardfunktion `guard`.

Zustand.lhs

```
> bsp4 = do
>   x <- [1,2,3,4]
>   y <- [5,6,7]
>   guard (x`mod`2==0)
>   return (x,y)
```

Damit ist die do-Notation gleich mächtig in ihrer Ausdrucksmöglichkeit wie die Mengenschreibweise auf Listen.

Zustand

```
*Zustand> bsp4
[(2,5),(2,6),(2,7),(4,5),(4,6),(4,7)]
```

3 Aufgaben

Aufgabe 1 In dieser Aufgabe soll eine kleine Imperative Sprache entwickelt werden, in der es veränderbare Variablen gibt, Einlesen von Werten aus der Konsole, Ausgabe auf die Konsole und eine einfache Schleife.

Hierfür sei der folgende Datentyp gegeben:

Zustand.lhs

```
> data Imperator =
>   Lese String
>  |Drucke Imperator
>  |Zuweisung String Imperator
>  |Literal Integer
>  |Var String
>  |Arith Imperator (Integer->Integer->Integer) Imperator
>  |While Imperator Imperator
>  |Sequenz [Imperator]
```

Die Fakultät lässt sich inklusive Einlesen des Arguments in diesem Datentyp wie folgt codieren:

Zustand.lhs

```
> factorial = Sequenz
>   [Lese "x"
>   ,Zuweisung "r" (Literal 1)
>   ,While
>      (Var "x")
>      (Sequenz
>        [Zuweisung "r" (Arith (Var "r")(*)(Var "x"))
>        ,Zuweisung "x" (Arith (Var "x")(-)(Literal 1))
>        ]
>      )
>   ,Drucke (Var "r")
>   ]
```

Um Programme des Datentyps `Imperator` auszuführen, benötigen wir die Monade `Anweisung`, die Monade `Maybe` und die Monade `IO` für Ein- und Ausgabeoperationen.

Wir können mit der Transformer-Monade `MaybeT` Monaden mit der Monade `Maybe` verheiraten. Eine solche Transformer-Monade benötigen wir jetzt auch, um die IO-Operationen mit den Anweisungen zu verheiraten.

Hierzu sei zunächst der neue Typ `AnweisungT` definiert.

Zustand.lhs

```
> newtype AnweisungT st m r
>    = AnweisungT {runAnweisungT::(st -> m (st, r))}
```

Dieser Typ soll ermöglichen, dass `IO` und `Anweisung` zusammen ein Monadentyp werden können.

Zunächst machen wir diesen Typ zu einen Funktor:

Zustand.lhs

```
> instance Monad m =>  Functor (AnweisungT st m) where
>   fmap f (AnweisungT anw)
>     = AnweisungT (\x-> fmap (\(s',a)->(s', f a)) (anw x))
```

Dann zu einer Monade:

Zustand.lhs

```
> instance Monad m =>  Monad (AnweisungT st m) where
>   (AnweisungT anw1) >>= k  = AnweisungT $ \ s -> do
>      ~(s',a) <- anw1 s
>      let (AnweisungT anw2) = (k a)
>      anw2 s'
>
>   -- fail str = AnweisungT $ \ _ -> fail str
```

Und schließlich auch eine Instanz von `Applicative`:

Zustand.lhs

```
> instance (Functor m, Monad m) => Applicative (AnweisungT st m) where
>   pure a = AnweisungT $ \ s -> return (s, a)
>
>   AnweisungT mf <*> AnweisungT mx = AnweisungT $ \ s -> do
>     ~(s', f) <- mf s
>     ~(s'', x) <- mx s'
>     return (s'', f x)

>   m *> k = m >>= \_ -> k
```

Damit wir sie als Transformer-Monade verwenden können und eine entsprechende Funktion `lift` erhalten, wir die Klasse `MonadTrans` implementiert:

Zustand.lhs

```
> instance MonadTrans (AnweisungT st) where
>   lift m = AnweisungT $ \ st -> do
>     a <- m
>     return (st, a)
```

Dieses benötigen wir, damit wir auch die IO-Monade mit den Anweisungen verknüpfen können:

Zustand.lhs

```
> instance (MonadIO m) => MonadIO (AnweisungT st m) where
>  liftIO io = lift  (liftIO io)
```

Um unsere Anweisungen ohne IO-Operatoren zu der Transformer-Monade zu machen, hilft diese kleine Hilfsfunktion:

Zustand.lhs

```
> liftAnweisung (Z f) = AnweisungT (\st -> return (f st))
```

Implementieren Sie jetzt schrittweise die Funktion, die die Ausfürhung eines `Imperator`-Programms beschreibt:

Zustand.lhs

```
> execute :: Imperator
>    ->  MaybeT (AnweisungT [(String, Integer)] IO) Integer
```

Die eigentliche Ausführung des ist dann die Komposition aus:
`(runAnweisungT.runMaybeT.execute)`.

Zustand

```
*Zustand> (runAnweisungT.runMaybeT.execute) (Literal 42) []
([],Just 42)
```

a) Implementieren Sie die Funktion `execute` für den Fall der Zahlenliterale:

Zustand.lhs

```
> execute (Literal n) = return 0 -- ToDo
```

b) Implementieren Sie jetzt die Funktion für den Fall eines Variablenzugriffs. Hierzu müssen Sie die Anweisung einmal zu einem Anweisungs-Transformer und diesen dann wieder ins Maybe transformieren:

Zustand.lhs

```
> execute (Var v) = do
>   return 0 -- ToDo
```

c) Als nächstes ist die Druckfunktion dran. Mit `print` können Sie beliebige Ausdrücke auf der Konsole ausgeben. Diese IO-Funktion müssen Sie aber dann noch auf die AnweisungT-Monade liften.

Zustand.lhs

```
> execute (Drucke imp) = do
>   return 0 -- ToDo
```

d) Als nächstes ist die Zuweisungsanweisung dran.

Zustand.lhs

```
> execute (Zuweisung v imp) = do
>   return 0 -- ToDo
```

e) Zum Lesen einer Variablen in der Konsole, ist die entsprechende IO-Operation zu liften und dann die Variable im Zustand zu setzen. Das Typsystem hilft Ihnen wahrscheinlich dabei, die korrekten Liftings zu machen.

Zustand.lhs

```
> execute (Lese v) = do
>   return 0 -- ToDo
```

f) Am einfachsten sind wahrscheinlich arithmetische Ausdrücke, weil hier keinerlei Liftings zu machen sind. Linker und rechter Operand sind auszuführen und der Operator zum Berechnen des Ergebnisses anzuwenden.

Zustand.lhs

```
> execute (Arith l op r) = do
>   return  0 -- ToDo
```

g) Auch bei der While-Schleife brauchen wir keine Liftings. Zunächst wird die Bedingung ausgeführt. Ist diese ungleich 0, so wird ausgeführt und anschließend wieder die ganze While-Schleife. Ist das Ergebnis der Bedingung 0, so wird einfach diese 0 zurück gegeben.

Zustand.lhs

```
> execute w@(While cond body) = do
>   return 0 -- ToDo
```

h) Und schließlich die Sequenz mehrerer Anweisungen. Auch hier brauchen wir keine Liftings und können uns ganz auf die Ausführung konzentrieren. Ergebnis soll das Ergebnis der letzten Anweisung in der Sequenz sein.

Zustand.lhs

```
> execute (Sequenz [i]) = return 0 -- ToDo
> execute (Sequenz (i:is)) = do
>   return 0 -- ToDo
> execute (Sequenz []) = return 0
```

4 Lernzuwachs

- Die Zustandsmonade
- Do-Notation

Literatur

Funktoren
Monaden
Do-Notation