

Kapitel 3

Monaden und IO

3.1 Monaden

Eine Monade ist ein Datentyp für Aktionen. Bereitgestellt werden Methoden, Aktionen zu kombinieren. Gekapselt wird i.a. der Zustand und das Weiterreichen des veränderten Zustands an die nächste Aktion. Eine Monade ist i.a. parametrisiert mit dem Typ des Zustands und dem Ausgabetyt der Aktionen. Eine wichtige Verwendung von Monaden ist die Einbindung von imperativen Konzepten und Seiteneffekten in eine nicht-strikte funktionale Programmiersprache wie Haskell. Insbesondere ist es damit leicht und sicher möglich, Ein-Ausgabe zu programmieren, oder zustandsbehaftete Berechnungen durchzuführen, und trotzdem weiterhin eine referentiell transparente, pure funktionale Programme zu haben. Die Unterstützung der Monaden in Haskell von der Syntax und vom Compiler her bewirkt, dass die entsprechenden Programmteile dann wie imperative Programme sind, aber die lazy funktionalen Berechnungen weiterhin genutzt werden können. In Haskell sind Monaden als Typklasse implementiert. Das Typklassensystem in Haskell ist wesentlich für die korrekte und flexible Verwendung von Monaden.

Die wahrscheinlich häufigste Verwendung der Klasse `Monad` ist die Darstellung von Zustands-Übergangssystemen, zu denen auch Ein-/Ausgabe in Haskell zählt.

Aber es gibt auch Beispiele für Interaktionen, für die andere Modelle besser geeignet sind; z.B. wenn sich die Außenwelt verändert, ohne dass das Programm dafür verantwortlich war.

Es gibt auch nicht-monadische Ansätze zu Ein-Ausgabe in funktionalen Programmiersprache, aber in diesem Teil des Skriptes behandeln wir nur die in Haskell verwendete Methode.

3.1.1 Typklasse `Monad`

Monadisches Programmieren meint die Verwendung eines speziellen Datentyps von Funktionen (Aktionen) der es erlaubt, Programme für single-threaded ver-

wendbare Objekte zu behandeln. In Haskell kann es dank des Typ- und Typklassensystem so eingebettet werden, dass der Typchecker in Zusammenarbeit mit dem Compiler die unzulässigen Verwendungen verbietet.

Ein Monade (mit Typkonstruktor `m`) ist in Haskell so definiert, dass die Objekte vom Typ `(m a)` als Aktionen mit Ausgabety `a` angesehen werden können. Die Monaden-Methoden sind sequentielle Verknüpfung von zwei Aktionen `>>=`, und Erzeugung einer Aktion (`return`). Das Programmieren besteht dann in der Konstruktion zusammengesetzter Aktionen. Die Verwendung dieser Aktion in einem Haskell-Programm besteht darin, diese auf eine Eingabe anzuwenden, und die Endergebnisse zu verwenden. Man hat, außer auf durch Programmieren von Aktionen, keinen direkten Zugriff (im Programm) auf die inneren Datenstrukturen der Aktionen.

Betrachte die Definition der Typklasse `Monad`, wie sie im Haskell-Prelude steht. Beachte, dass diese eine Konstruktorklasse ist.

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

  -- Minimal complete definition: (>>=), return
  p >> q = p >>= \ _ -> q
  fail s = error s
```

Der Operator `>>=` wird auch „Bind“ genannt. In Haskell ermöglicht die `do`-Notation, Verkettung von Aktionen einfacher zu schreiben:

```
a1 >>= (\x -> a2 >>= (\y -> a3 >>= \_ -> return (x,y)))
```

kann man einfacher schreiben als:

```
do {x <- a1; y <- a2; a3; return (x,y)}
```

Die Übersetzungsregeln für die `do`-Notation sind:

```
do {x <- e; s} = e >>= \x -> do { s }
do {e; s}     = e >> do { s }
do {e}        = e
```

Die Eigenschaften der Methoden der Typklasse `Monad` sind folgendermaßen algebraisch definiert (Monadengesetze):

```
1: return x >>= f           = f x
2: m >>= return             = m
3: m1 >>= (\x -> m2 >>= \y -> m3) = (m1 >>= \x -> m2) >>= \y -> m3
    Falls x ∉ FV(m3)
```

Die Eigenschaft der Bind-Operation kann man auch mit `do` notieren; es ist eine leicht abgewandelte Assoziativität der Hintereinanderausführung:

$$\begin{array}{l} \text{do } \{ \quad x \leftarrow m_1; \\ \quad \quad y \leftarrow m_2; \\ \quad \quad m_3 \end{array} = \begin{array}{l} \text{do } \{ \quad y \leftarrow \text{do } \{ \quad x \leftarrow m_1; \\ \quad \quad m_2 \}; \\ \quad \quad m_3 \end{array}$$

Leider kann das Typsystem bzw. der Compiler die Gültigkeit dieser Gesetze nicht prüfen, wenn ein Datentyp zur Instanz von Monade erklärt wird. Hierfür ist der Programmierer verantwortlich.

Beispiele: Datenstrukturen als Monaden

```
instance Monad Maybe where
  Just x  >>= k = k x
  Nothing >>= k = Nothing
  return  = Just
  fail s   = Nothing
```

Für die Typklasse `Maybe` kann man die Monadenregeln nachrechnen. Man muss allerdings eine call-by-need Auswertung verwenden statt einer call-by-name-Auswertung, d.h. keine einsetzenden Reduktionen wie (beta) und (case), sondern muss sharing (i.a. mit `let`) im Kalkül eingebaut haben. Der Grund ist, dass die Implementierung der Monaden in Haskell nur auf der Basis von call-by-need korrekt ist. Damit die Rechnungen unten mathematisch fundiert sind, muss man u.a. folgendes Wissen voraussetzen:

- Die call-by-name Reduktionen sind korrekt, d.h. es sind Gleichheitstransformationen.
- Die Einsetzung von Variablen für Variablen ist korrekt: d.h. die Beta-Reduktion $(\lambda x.t) z \rightarrow t[z/x]$ ist korrekt für Variablen z .
- Die Einsetzung für x ist korrekt, wenn eine Variable x maximal einmal vorkommt. d.h. die Beta-Reduktion $(\lambda x.t) s \rightarrow t[s/x]$ ist korrekt wenn die Variablen x nur einmal in t vorkommt.
- Schieben einer let-Bindung nach außen ist korrekt. Es gibt verschiedene Varianten dieses Verschiebens, die alle korrekt sind.

1. `return x >>= f = f x`:
`Just x >>= f` ist gerade als `f x` definiert.

2. `m >>= return = m`:
Hier ist `m >>= Just` zu analysieren: `m` kann `Just s` sein; Die Pattern-Variable des Bind kommt nur einmal vor, somit ist das Ergebnis `Just s`, d.h. `m`. Im Fall `m = Nothing` ergibt sich als Resultat `Nothing`, also ebenfalls `m`. Der dritte Fall ergibt sich, wenn `m = bot`. In dem Fall ergibt sich ebenfalls `bot` auf beiden Seiten

3. Wir verifizieren das dritte Monadengesetz:

Der bot-Fall geht genauso wie oben.

Ist `m_1 = Nothing`, so ergibt die linke Seite `Nothing`. Die rechte Seite ergibt nach zweimaliger Anwendung der Definition ebenfalls `Nothing`.

Ist `m_1 = Just s`, dann ergibt sich links:

```
let x = s in m_2 >>= \y -> m_3.
```

Auf der rechten Seite ergibt sich `(let x = s in m_2) >>= \y -> m_3`.

Nach dem Schieben der `let`-Bindung nach außen stimmt die Gleichung.

Listen als Monade

Listen können als Monaden gesehen werden mit den Definitionen:

```
instance Monad [ ] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  [] >>= f = []
  return x = [x]
  fail s = []
```

Nachrechnen der Monadengesetze:

1. `[x] >>= f = f x ++ ([] >>= f) = f x`. Als Begründung: die Variablen in die eingesetzt wird, kommen nur einmal vor.

2. `m >>= \x -> [x]` ergibt für `m = []` wieder `[]` und für `m = s : zs` den Ausdruck `[s] ++ (zs >>= \x -> [x])`. (Auch hier kommen die ersetzten Variablen nur einmal vor).

Wenn `m = bot`, dann gilt die Gleichung. Für endliche Listen kann man Induktion durchführen, für unendliche Listen kann man ebenfalls Induktion verwenden, wenn man den Basisfall `m = bot` dazu nimmt.¹

3. (Übungsaufgabe, wobei man die `let`-Schiebe-Regeln verwenden muss.)

Die Interpretation als Aktionen ist hier etwas abwegig, aber die `do`-Notation kann man direkt mit der List-Comprehensions-Notation vergleichen:

Die Schreibweise

`[e | x <- xs; y <- ys]` entspricht der Notation:

```
do {x <- xs;
    y <- ys;
    return e}
```

Dazu passt die äquivalente Bind-Definition, die zu den Transformationsregeln für List-Comprehensions passt:

```
xs >>= f = concat (map f xs)
```

List Comprehensions können etwas mehr als die `do`-Notation: Prädikate und Pattern. Diese lassen sich nicht in die `do`-Notation übertragen.

¹Genauere Begründung für diese Vorgehensweise in einem späteren Abschnitt des Skriptes

3.1.2 Zustandsmonade

Eine Zustandsmonade ist eine Datenstruktur, die Funktionen (Aktionen) kapselt, die einen Zustand verändern, und die es erlaubt aus kleinen Aktionen mittels sequentieller Komposition größere Aktionen zusammensetzen.

```
newtype StateTransformer s a = ST (s -> (a, s))

-- s    Zustands-Typ
-- a    Typ des Ausgabewerts
-- s -> (a, s)  Typ einer Aktion

apply :: StateTransformer s a -> s -> (a, s)
apply (ST f) x = f x

instance Monad (StateTransformer s) where
  return x = ST $ \s -> (x, s)
  a1 >>= aa2 = ST action
    where
      action = \s -> let (res, s') = apply a1 s
                      in apply (aa2 res) s'
```

Eine beispielhafte Benutzung ist ein Zustand, der nur aus einer Zahl besteht, und die man mittels einiger Aktionen verändern kann. Durch die Konstruktion werden alle Aktionen sequenzialisiert, d.h. man muss bereits sequentiell programmieren. Um den Wert zu erhalten, muss man die monadische Berechnung abbrechen.

```
--- Operationen auf dem Zustand:
incr = ST $ \s -> ((), s+1)
decr = ST $ \s -> ((), s-1)
hole = ST $ \s -> (s, s)
add n = ST $ \s -> ((), s +n)
sub n = ST $ \s -> ((), s - n)
clear = ST $ \_ -> ((), 0)
check p = ST $ \s -> (p s, s)

anfang = ST (\ein -> ((),ein))

test = (anfang >> incr >> incr >> add 3 >> hole)
zeigetest = (apply test 4444)
> (4449,4449) :: (Integer,Integer)

test2 = anfang >> incr >> add 3 >> sub 5
zeigetest2 = (apply test2 3333)
> ((),3332) :: ((),Integer)
```

3.1.3 Kontrollstrukturen auf Aktionen

Da wir nun eine Schicht haben, die eine imperative Sicht erlaubt, sollte es auch entsprechende Kontrollstrukturen geben. Einige sind beispielhaft angegeben und programmiert:

```
forever :: Monad m => m a -> m b
forever a = a >> forever a

repeatN :: (Monad a, Num b) => b -> a c -> a ()

repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

Damit kann man kompliziertere Aktionen erzeugen, z.B. in der Zustandsmonade:

```
repeatN :: (Monad m, Num a) => a -> m b -> m ()

repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a

test3 = anfang >> incr >> (repeatN 5 (add 3)) >> sub 5
zeigetest3 = (apply test3 6666)
> ((),6677) :: ((),Integer)
```

Die Funktion `for` macht zuerst ein `map` über eine Liste, erzeugt dadurch eine Liste von IO-Aktionen, die dann sequentiell abgearbeitet werden.

```
-- Aktion pro Element einer Liste
for :: Monad m => [a] -> (a -> m c) -> m ()
for []      fa = return ()
for (n:ns) fa = fa n >> for ns fa
----alternativ:
for ns fa = sequence_ (map fa ns)

-- wobei:  sequence_ :: Monad a => [a b] -> a ()
--         sequence_ as = foldr (>>) (return ()) as

-- print :: Show a => a -> IO ()

testprintZahlen = for [1..10] print
```

Will man die Ergebnisse der Aktionen in einer Liste haben, dann verwendet man `sequence`.

```
sequence []      = return []
sequence (a:as) = do { r <- a;
```

```
rs <- sequence as;
return (r:rs) }
```

Man kann auch ein `while` in der Zustandsmonade programmieren:

```
while :: Monad m => m Bool -> m a -> m ()
while check a = do {
    b <- check;
    if b then a >> while check a
    else return ()
}
```

```
test4 = anfang >> (add 10) >> while (check (\x -> x > 0)) decr
zeigetest4 = (apply test4 7777)
> ((),0) :: ((),Integer)
```

3.2 Monadisches IO in Haskell

Ein/-Ausgabe geschieht in Form von Aktionen vom Typ `IO a`, wobei die Aktion auf der `World` operiert, und `a` der Typ des zurückgelieferten Wertes ist. Ein Haskell-Programm ist dann z.B. eine Aktion vom Typ `IO ()`. Das Objekt `World` kann man im Programm nicht selbst erzeugen, es wird beim Programmstart dem Programm übergeben, am Ende liefert das Programm das veränderte Objekt `World` zurück.

Die einfachste Vorstellung ist, dass der Typ `IO a` eine Abkürzung ist:

```
type IO a = World -> (a, World)
```

Das bedeutet aus Programmsicht: es gibt die Außenwelt, und eine Aktion vom Typ `IO a` liefert ein Paar: den Wert vom Typ `a` und eine evtl. veränderte Außenwelt.²

Wir gehen einige Funktionen durch, um ein Gefühl dafür zu bekommen, was machbar ist. Das einfachste Beispiel sind die zwei Funktionen

```
getChar :: IO Char          -- World -> (Char,World)
putChar :: Char -> IO ()    -- Char -> (World -> ((),World))
```

`getChar` liest ein Zeichen von der Eingabe, `putChar` druckt ein Zeichen in die Standardausgabe. Beide verändern die Außenwelt: `getChar` entfernt das gelesene Zeichen von der Standardeingabe und `putChar` fügt ein Zeichen zur Standardausgabe hinzu.

Man macht den Typ `IO` zu einer Monade durch Bereitstellen von primitiven Funktionen (siehe Prelude), aber wir zeigen unten auch noch eine mögliche funktionale Kodierung:

²Die prelude-Definition hat einen etwas allgemeineren Typ, da dort rein technisch anders verfahren wird

```
instance Monad IO where
  (>>=) = primbindIO
  return = primretIO
```

Die Aktionen `getChar` und `putChar` kann man kombinieren zu `echo` mittels `>>=`:

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

```
echo :: IO ()
echo = getChar >>= putChar
```

Der Operator `Bind` bewirkt sequentielle Ausführung, wobei zusätzlich das Ergebnis der ersten Aktion als Argument der zweiten Aktion verwendet wird. Das stimmt mit dem instanziierten Typ des `Bind` überein:

```
IO Char -> (Char -> IO ()) -> IO ()
```

Beispiel 3.2.1 *Ein Doppelecho kann man so erreichen:*

```
getChar >>= (\c -> putChar c >> putChar c)
```

Vordefinierte Funktionen, die eine Zeile einlesen und ausgeben, sind:

```
getLine :: IO [Char]
```

```
getLine = getChar >>= \c ->
  if c == '\n'
  then return []
  else getLine >>= \cs ->
  return (c:cs)
```

```
putLine :: [Char] -> IO ()
```

```
putLine [] = return ()
putLine (c:cs) = putChar c >> putLine cs
```

Notation mit `do`

Als Beispiel nochmal die Übersetzung der `>>=`-Notation in die `do`-Notation.

```
getLine = getChar >>= \c ->
  if c == '\n'
  then return []
  else getLine >>= \cs ->
  return (c:cs)

getLine = do { c <- getChar;
  if c == '\n'
  then return []
```

```
        else    do { cs <- getLine;
                  return (c:cs)
                }
      }
    }
```

3.2.1 Zellen mittels IORef

Im Modul IOExt sind weitere Funktionen vordefiniert. Eine interessante Erweiterung sind IORefs, die sich wie veränderbare Speicherzellen verhalten, d.h. wie Variablen in imperativen Programmen. Allerdings kann man diese nur im IO-Datentyp verwenden.

```
data IORef a    -- wie in ML
newIORef  :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef:: IORef a -> a -> IO ()

count :: Int -> IO Int
count n = do { r <- newIORef 0;
              loop r 1 }
  where
    loop r i | i > n = readIORef r
              | otherwise = do { v <- readIORef r;
                                writeIORef r (v+i);
                                loop r (i+1)}
```

3.2.2 Behandlung der Außenwelt

Das Objekt „World“ wird nicht direkt sichtbar gemacht, sondern kann nur durch IO-Aktionen abgefragt bzw. verändert werden. Folgende Bedingung muss gelten:

Es gibt immer nur ein Objekt World im gerade aktuellen Zustand.

D.h. das Objekt World darf nicht kopiert werden; es dürfen keine alten Versionen aufgehoben werden. D.h. es darf nicht gespeichert werden bzw. in einem let referenziert werden, denn nach der Veränderung durch eine Aktion hat man nur noch die Welt nach der Änderung. Dies wird gewährleistet durch eine single-threaded Behandlung der Welt (ein Berechnungsfaden).

Diese single-threaded Behandlung wird in Haskell durch die Benutzung von Monaden für IO gewährleistet. Es gibt noch andere Datentypen, die single-threaded benutzt werden sollen, bzw. bei einer solchen Benutzung leichter zu optimieren sind: z.B. Arrays.

3.2.3 Eigenschaften der IO-Monade

Damit Ein-/Ausgabe in Haskell korrekt funktioniert und das Programmieren pur bleibt, gelten folgende Eigenschaften bzw. Beschränkungen:

- Ein Haskell-Programm ist selbst eine IO-Aktion, d.h. vom Typ `IO ()`.
- IO-Aktionen kann man nur mit speziellen Operatoren wieder kombinieren: `>>=` und `return`.
- IO-Aktionen können im Programm herumgereicht werden, können kombiniert werden und Resultate von anderen Funktionen sein.
- Man kommt aus der IO-Monade nicht heraus, erst wenn man am Ende das Ergebnis von `main` hat, d.h. das Ergebnis des Programms.

Der wesentliche Effekt der Bind-Operation ist die zuverlässige Sequentialisierung der zugehörigen IO-Aktionen und die single-threaded Behandlung der `World`. Ein weiterer Effekt des Bind ist die Bereitstellung von Umgebungen, mit der man Ergebnisse von IO-Aktionen an spätere IO-Aktionen weitergeben kann. Es gibt zwar eine Möglichkeit, `unsafePerformIO`, um diese Beschränkungen zu umgehen, allerdings wird die Reihenfolge der IOs und Zugriffe nicht vom Compiler gewährleistet.

3.2.4 Eine funktionale Kodierung der IO-Operatoren

Wenn wir `World` als ein nichtkopierbares Objekt vom Typ `World` annehmen, dann kann man Bind und `return` implementieren:

```
return :: a -> IO a
return a = \w -> (a,w)

(>>=) :: IO a -> (a -> IO b) -> IO b
m >>= k = \w -> case (m w) of (r,w') -> k r w'
```

Hier ist `w'` die neue `World`. Auch dafür kann man die Monadengesetze verifizieren, wenn man annimmt, dass Aktionen deterministisch sind.

Interessanterweise ist interne Reduktion in der Kernsprache eingeschränkt: Der Compiler muss darauf achten, dass nicht die Beta-Reduktion verwendet wird, sondern verzögerte Auswertung mit Sharing. D.h. normale Beta-Reduktion ergibt falsche Auswertung, insbesondere gilt der Satz von Church-Rosser nicht mehr für die Kombination der (kopierenden) Beta-Regel mit IO-Aktionen: Ebenso ist die Beta-Reduktion nicht korrekt, und auch nicht mehr die Ersetzung von gleichen Termen durch gleiche:

Beispiel 3.2.2 *Hier ein Beispiel von Simon Peyton Jones.*

```
do {c <- getChar; putChar c; putChar c}
```

übersetzt man das unter Benutzung der Definition von `do` und `>>=`, ergibt sich:

```
\w -> case getChar w of
  (c,w1) -> case putChar c w1 of
    (_,w2) -> putChar c w2
```

Würde der Satz von Church-Rosser und obige Ersetzungsmöglichkeiten gelten, wäre es dem Compiler erlaubt, Ersetzung von gleichen Ausdrücken durchzuführen, brigens auch ohne die Annahme, dass

```
\w -> case getChar w of
      (c,w1) -> case putChar c w1 of
                (_,w2) -> putChar (fst (getChar w)) w2
```

Hier sind Bedingungen beim Umgang mit IO und der World verletzt: die World-Referenz w wurde kopiert, aber schlimmer ist, dass aus einem `getChar w`-Aufruf zwei geworden sind. Die neue Funktionen hat dadurch ein anderes Verhalten als die erste.

Durch verzögerte Auswertung mit Sharing passiert so etwas nie, aber man muss vorsichtig sein mit den Transformationen, die ein Compiler durchführt (u.a. sogenannte partielle Auswertung); auch hierbei darf das Verhalten der Programme nicht verändert werden.

Dieses Problem tritt auch auf, wenn man die Implementierung nicht voraussetzt: Offenbar sind die beiden folgenden `do`-Ausdrücke nicht gleich, aber nach den Regeln der *call-by-name*-Ersetzung sollten diese gleich sein:

```
do {c <- getChar; putChar c; putChar c}
do {c <- getChar; d <- getChar; putChar c; putChar d}
```

3.3 Beispiele für Monaden: Taschenrechner

3

Im folgenden wird ein einfacher Taschenrechner programmiert als Beispiel für die Verwendung einer Zustandsmonade und als Erweiterung für die Kombination einer Zustandsmonade mit der IO-Monade.

Nehmen wir an, die Aufgabe ist einen Taschenrechner zu simulieren. Die Eingabe stellen wir durch eine Zeichenkette dar. Jedes einzelne Zeichen soll einen Zustands-Übergang bewirken. Die Zeichen aus denen die Eingabe besteht beschränken wir auf die Ziffern '0'-'9', die Operatoren '+', '-', '*' und '/', den Dezimalpunkt und die Zeichen 'c' und '='. Weiterhin soll der Taschenrechner die Infix-Schreibweise verwenden, wir wollen also z.B. "123*456=" eingeben und 56088.0 als Ergebnis erhalten.

Wir könnten einen Zustand verwenden, in dem der Operator und die beiden Operanden gespeichert werden, aber da nach Eingabe des Operators der erste Operand nicht mehr verändert wird, speichern wir im Zustand eine Funktion und einen Operator. Wir definieren also

```
module Main where
```

```
type CalcState = (Float -> Float, Float)
```

```
type CalcTransformer a = CalcState -> (a, CalcState)
```

³Dank an Marko Schütz

Folgende CalcTransformer benötigen wir für die Simulation

```
clear :: CalcTransformer ()
clear (g, 0.0) = ((), (id, 0.0))
clear (g, z)   = ((), (g, 0.0))

digit :: Float -> CalcTransformer ()
digit d (g, z) = ((), (g, z * 10.0 + d))

oper :: (Float -> Float -> Float) -> CalcTransformer ()
oper o (g, z) = ((), (o (g z), 0.0))

total :: CalcTransformer ()
total (g, z) = ((), (id, g z))

readResult :: CalcTransformer Float
readResult (g, z) = (g z, (g, z))

startState :: CalcState
startState = (id, 0.0)
```

Ein Schritt unseres Rechners besteht dann im Auswählen der passenden Operation zum Eingabe-Zeichen.

```
calcStep :: Char -> CalcTransformer ()
calcStep x
  | isDigit x = digit (fromInt (ord x - ord '0'))
  | x == '+'  = oper (+)
  | x == '-'  = oper (-)
  | x == '*'  = oper (*)
  | x == '/'  = oper (/)
  | x == 'c'  = clear
  | x == '='  = total
```

Nun ist es eine Kleinigkeit den ganzen Rechner zu definieren.

```
calc :: String -> CalcTransformer Float
calc "" s      = readResult s
calc (x:xs) s = let (_,newState) = calcStep x s

                in calc xs newState

main = fst (calc "123*456=" startState)
```

Wir sehen hier ein häufig wiederkehrendes Konzept: Funktionen, die Zustände transformieren und ggf. zu einem neuen Zustand auch noch einen Wert berechnen. In Haskell können wir CalcTransformer verallgemeinern zu einer Zustandsmonade:

```
newtype StateTransformer s a = ST (s -> (a, s))
```

Wir verwenden hier `newtype`, weil man in Haskell kein Typsynonym zu einer Instanz einer Klasse machen kann, was wir später vorhaben. Für diese Einschränkung gibt es gute Gründe, die hier aber nicht besprochen werden sollen. Da wir zwar *im Inneren* eines `StateTransformer s a` eine Funktion haben, aber diese von einem Daten-Konstruktor “verdeckt” wird, definieren wir eine Funktion, die uns die verdeckte Funktion anwendet.

```
apply :: StateTransformer s a -> s -> (a, s)
apply (ST action) = action
```

Es hilft sich den Typ von `StateTransformer s a` als *Aktion* auf Zuständen vorzustellen, wobei die Aktion auch noch ein Ergebnis berechnen kann. Hat ein Ausdruck einen solchen Typ, dann ist das Ergebnis eine Aktion, die noch auf einen Zustand “wartet”, bevor sie ausgeführt werden kann.

Zwei Operationen, die wir naheliegenderweise für solche `StateTransformer s a` zur Verfügung haben wollen sind zum einen zwei `StateTransformer s a` hintereinander zu schalten und zum anderen eine `StateTransformer s a` zu erzeugen, der den Zustand unverändert läßt, aber einen Wert vom Typ `a` berechnet.

Wir machen `StateTransformer s` zur Instanz von `Monad`. Dazu sind die Operationen `(>>=)` und `return` zu definieren. `(>>=)` erhält im wesentlichen zwei `StateTransformer s a`, also zwei Aktionen und erzeugt daraus eine Aktion, die, wenn sie einen Zustand als Argument erhält, beide Aktionen hintereinander auf diesem Zustand ausführt, wobei die zweite Aktion den aus der ersten Aktion entstehenden Zustand als Ausgangszustand verwendet. Das zweite Argument von `(>>=)` soll nicht bloß eine Aktion sein, sondern soll vom Ergebnis der ersten Aktion abhängen. Dieses Argument ist also eine Funktion, die das evtl. berechnete Ergebnis der ersten Aktion verwendet, um eine Aktion zu erzeugen. Das sieht in Haskell so aus:

```
instance Monad (StateTransformer s) where
  return x = ST $ \s -> (x, s)
  m >>= k = ST action
    where
      action s = apply (k res) s'
        where
          (res, s') = apply m s
```

Der Kombinator `(>>)` braucht nicht definiert zu werden, da er bereits durch die Instanz-Definition zur Verfügung steht.

Wenn wir nun unseren Rechner mit `StateTransformer s a` darstellen wollen, erhalten wir folgendes.

```
type CalcTransformer = StateTransformer CalcState
```

```
clear :: CalcTransformer ()
clear = ST action
  where
    action (g, 0.0) = ((), (id, 0.0))
    action (g, z)   = ((), (g, 0.0))

digit :: Float -> CalcTransformer ()
digit d = ST $ \ (g, z) -> ((), (g, z * 10.0 + d))

oper :: (Float -> Float -> Float) -> CalcTransformer ()
oper o = ST $ \ (g, z) -> ((), (o (g z), 0.0))

total :: CalcTransformer ()
total = ST $ \ (g, z) -> ((), (id, g z))

readResult :: CalcTransformer Float
readResult = ST $ \ (g, z) -> (g z, (g, z))
```

An `calcStep` ändert sich nichts, aber wir können (`>>=`) verwenden, um `calc` neu zu definieren.

```
calc :: String -> CalcTransformer Float
calc ""      = readResult
calc (x:xs) = calcStep x >>
               calc xs
```

Das sieht zwar zunächst nicht unbedingt einfacher aus, aber wir haben erreicht, dass Aktionen und deren Komposition unabhängig vom Zustand definiert sind. Mit der monadischen Syntax kann man `calc` so schreiben:

```
calc ""      = readResult
calc (x:xs) = do calcStep x
                  calc xs
```

Macht man die lexikalischen Elemente sichtbar, die hier durch Layout eingefügt werden, dann hat man

```
calc (x:xs) = do { calcStep x; calc xs }
```

3.3.1 IO

Die Vorstellung, die der Ein-/Ausgabe in Haskell zugrundeliegt ist, dass ein Zustand der Umgebung verändert wird, dass z.B. eine Ausgabe-Aktion den Zustand der Umgebung gerade so verändert, dass nach der Aktion die Ausgabe in der Umgebung beobachtbar ist. Ganz entsprechend verändert eine Eingabe-Aktion den Zustand der Umgebung.

In Haskell ist dazu der Datentyp `IO a` verfügbar, um Ein-/Ausgabe-Aktionen zu verwenden, die auch noch ein Ergebnis von einem beliebigen Typ `a` haben

können. Der Typ-Konstruktor `IO` und die Funktionen, die ihn zu einer Instanz der Klasse `Monad` machen sind in Compiler und Interpreter eingebaut und abstrakt, d.h. ein Programmierer hat keine Möglichkeit die Konstruktoren des Typs `IO a` direkt zu verwenden, sie stehen im Programm nur durch Funktionen zur Verfügung, die auf Objekten vom Typ `IO a` operieren können und von der Prelude bereitgestellt werden.

Als Ausgabe-Funktionen gibt es z.B.:

```
putChar :: Char -> IO ()
putStr  :: String -> IO ()
putStrLn :: String -> IO ()
print  :: Show a => a -> IO ()
```

An Eingabe-Funktionen gibt es z.B.:

```
getChar :: IO Char
getLine :: IO String
getContents :: IO String
interact :: (String -> String) -> IO ()
readIO  :: Read a => String -> IO a
getLine :: Read a => IO a
```

und für Dateien

```
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile  :: FilePath -> IO String
```

Eine Variation des Rechners mit IO ist

```
main = do k <- getLine
         let (g, z) = apply (calc k) startState
         print $ g
```

Lässt man dieses Programm laufen, kann man eine Zeile eingeben und nach dem Betätigen der Eingabe-Taste erhält man das Ergebnis. Die Funktionen `getLine` und `print` erzeugen IO-Aktionen, also Aktionen, die den Ein-/Ausgabe-Zustand des Programmes verändern. Dadurch, dass der von einer IO-Aktion veränderte Zustand der Zustand ist, den die darauffolgende IO-Aktion als Anfangs-Zustand vorfindet, erhalten wir eine Sequentialisierung. Es kann z.B. keine Ausgabe durch `print` stattfinden, solange nicht die ganze Zeile eingegeben worden ist. Es würde dem Modell eines Taschenrechners wohl eher entsprechen, wenn unmittelbar nach dem Betätigen der Taste = das Zwischenergebnis ausgegeben würde. Dazu müssen wir einzelne Zeichen einlesen, mit `calcStep` sofort verarbeiten und da wir ggf. das Zwischenergebnis ausgeben wollen müssen wir aus `calcStep` auf den Ein-/Ausgabe-Zustand zugreifen können. Das Ziel ist also, Aktionen erzeugen zu können, die sowohl einen Zustandsübergang verursachen, als auch Ein- und Ausgabe bewirken. Dieses Ziel erreichen wir indem wir aus dem Typ `CalcTransformer` eine Kombination zweier Monaden machen, der IO-Monade und der Zustandsmonade.

3.3.2 Kombination von Monaden mit IO

Die Zustandsmonade kann man mit einer beliebigen anderen Monade `m` kombinieren und zwar so

```
newtype STT s m a = STT (s -> m (a, s))

apply :: STT s m a -> s -> m (a, s)
apply (STT f) = f
```

Hier haben wir auch gleich den Zustand `s` zu einem Parameter gemacht. Im Haskell Typ-System können nicht beliebige Typen zu Instanzen einer Klasse gemacht werden, sondern es gibt Beschränkungen. Wir wollen `STT` zu einer Instanz eines Typs machen und müssen deswegen `newtype` und den zusätzlichen Daten-Konstruktor `STT` verwenden, anstatt `type`, um diese Beschränkungen zu erfüllen.

```
instance Monad m => Monad (STT s m) where
  return x = STT $ \ s -> return (x, s)
  m >>= k = STT action
    where action s = do (x, s') <- apply m s
                      apply (k x) s'
```

Mit `STT` und der bisherigen Definition von `CalcState` definieren wir den Typ eines Rechner-Zustandsübergangs als

```
type CalcTransformer = STT CalcState IO
```

Wir wollen Ein-/Ausgabe-Aktionen, also Elemente des Typs `IO a` zu Zustands-Transformationen umwandeln, die Ein-/Ausgabe beinhalten. Das Ergebnis einer solchen Umwandlung kann man dann sowohl als eine Ein-/Ausgabe-Aktion auffassen, als auch als einen Zustandsübergang. Wir verallgemeinern von der `IO`-Monade auf eine Klasse `IOMonad`, damit wir die Funktionen, die von einer Monade bereitgestellt werden und die Ein-/Ausgabe-Aktionen ergeben, gleich benennen können. Wir verwenden hier nur `ioPrint` und `ioGetChar`. Für andere Funktionen geht das dann ganz analog.

```
class Monad m => IOMonad m where
  ioGetChar :: m Char
  ioPrint :: Show a => a -> m ()
```

Natürlich soll `IO` eine Instanz dieser Klasse sein. Da `IO` der Prototyp dieser Klasse ist, sind die Definitionen besonders einfach.

```
instance IOMonad IO where
  ioGetChar = getChar
  ioPrint = print
```

Um nun `STT s m` zu einer Instanz der Klasse `IOMonad` zu machen verwenden wir ein allgemeineres Konzept, mit dem eine Aktion einer "inneren" Monade in eine triviale Aktion einer "äußeren" Monade sozusagen eingewickelt wird.

```
class MonadTransformer t where
  promote :: Monad m => m a -> t m a

instance MonadTransformer (STT s) where
  promote g = STT $ \ s -> do {x <- g; return (x, s)}

instance IOMonad m => IOMonad (STT s m) where
  ioGetChar = promote ioGetChar
  ioPrint x = promote $ ioPrint x
```

Wir erhalten also eine Instanz der Klasse `IOMonad` für `STT s m`, wenn `m` bereits eine Instanz dieser Klasse ist, indem wir die entsprechenden Aktionen von `m` einbetten in eine triviale Aktion der äußeren Zustandsmonade. Anders ausgedrückt erhalten wir eine Ein-/Ausgabe-Aktion der kombinierten Monade indem wir ihren Zustands-Teil unverändert lassen, aber dabei die entsprechende Aktion ihres Ein-/Ausgabe-Teils verwenden.

Als nächstes werden wir die Operationen des Rechners neu definieren. Wir haben nun erreicht, dass wir diese Operationen für beliebige Monaden, insbesondere für die IO-Monade, verwenden können. Wollen wir im Inneren einer solchen Aktion Ein-/Ausgaben veranlassen müssen wir lediglich sicherstellen, dass die verwendete innere Monade eine Instanz der Klasse `IOMonad` ist.

```
clear :: Monad m => STT CalcState m ()
clear = STT action
  where
    action (g, 0.0) = return (), (id, 0.0)
    action (g, z)   = return (), (g, 0.0)

total :: IOMonad m => STT CalcState m ()
total   = STT $ \(g, z) -> do {ioPrint $ g z; return (), (id, g z)}}

digit :: Monad m => Float -> STT CalcState m ()
digit d   = STT $ \(g, z) -> return (), (g, z*10.0 + d)

oper :: Monad m => (Float -> Float -> Float) -> STT CalcState m ()
oper o    = STT $ \(g, z) -> return (), (o (g z), 0.0)

readResult :: Monad m => STT CalcState m Float
readResult = STT $ \(g, z) -> return (g z, (g, z))
```

Schließlich können wir den Rechner auf den Anfangszustand anwenden. Da `IO ()` der Typ von `main` sein muss, haben wir die letzte Zeile eingefügt, die gerade dies sicherstellt.

```
main = do  apply calc startState
          return ()

----  (apply calc startState) :: IO ((),(Float -> Float,Float))

calc :: CalcTransformer ()
calc = do {k <- ioGetChar;
          if k == '\n'
            then do {x <- readResult;
                    ioPrint x}
            else do {calcStep k;
                    calc}}
```

3.3.3 Test, Kritik und Änderung

Der jetzige Taschenrechner ist natürlich in vieler Hinsicht erweiterbar und verbesserbar.

Um zu zeigen, wie man ihn in dem allgemeinen Rahmen leicht ändern kann, beheben wir den Fehler, der in einer alten Variante auftrat bei folgender Eingabe:

```
Main> main
123*456=56088.0
1+1=560882.0
```

Offenbar kommt das daher, dass `digit` einfach eine Ziffer an das Ergebnis anhängt. Um das zu beheben, kann man sich den letzten Befehl merken. Dazu ist nur die Zustandsmonade zu ändern, alles andere bleibt gleich.

```
type CalcState = (String,Float -> Float, Float)

startState :: CalcState
startState = (" ",id, 0.0)

calcStep .....

clear :: Monad m => STT CalcState m ()
clear = STT action
  where
    action (str,g, 0.0) = return ((), ("c",id, 0.0))
    action (str,g, z)   = return ((), ("c",g, 0.0))

total :: IOMonad m => STT CalcState m ()
total = STT $ \(_,g, z) -> do {ioPrint $ g z;
                             return ((), ("t",id, g z))}

digit :: Monad m => Float -> STT CalcState m ()
```

```
digit d    = STT $ \(str,g, z) ->
              if str == "t" then return ((), ("d",g, d))
              else return ((), ("d",g, z*10.0 + d))

tunichts :: Monad m => STT CalcState m ()
tunichts  = STT $ \(str,g, z) -> return ((), (str,g, z))

oper :: Monad m => (Float -> Float -> Float) -> STT CalcState m ()
oper o    = STT $ \(_,g, z) -> return ((), ("o",o (g z), 0.0))

readResult :: Monad m => STT CalcState m Float
readResult = STT $ \(_,g, z) -> return (g z, ("r",g, z))
```

Das Verhalten ist jetzt so:

```
Main> main
123*456=56088.0
1+1=2.0
```

```
-----
Main> main
123*456=56088.0
+1=56089.0
```

Der aktuelle Taschenrechner ist `Calculator6.hs`, der auch Komma-Eingaben korrekt berechnet, aber keine Exponenten.