1 Monaden, IO

Oft ist eine natürliche Formulierung eines Programmes eine, die einen Zustand verwendet und diesen schrittweise verändert. Wir werden uns nun ein solches ansehen.

Nehmen wir an, die Aufgabe ist einen Taschenrechner zu simulieren. Die Eingabe stellen wir durch eine Zeichenkette dar und 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)
```

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 = 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

```
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.

Zum Hintereinanderschalten definieren wir eine Operation `(>>=)`. Diese 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 eben den aus der ersten Aktion entstehenden Zustand als Ausgangszustand verwendet. Würden wir das direkt so definieren, so würden wir das evtl. berechnete Ergebnis nach der ersten Aktion verlieren. Daher soll das zweite Argument von `(>>=)` nicht bloß eine Aktion sein, sondern diese Aktion 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:

```
(>>=) :: StateTransformer s a -> (a -> StateTransformer s b)
         -> StateTransformer s b
m >>= k = ST action
  where
  action s = apply (k res) s'
    where
    (res, s') = apply m s
```

Außerdem definieren wir noch `(>>)`, wenn es wirklich nur um zwei Aktionen gehen soll, wobei die zweite von dem möglicherweise berechneten Ergebnis der ersten unabhängig ist.

```
(>>) :: StateTransformer s a -> StateTransformer s b
        -> StateTransformer s b
m >> k = m >>= \_ -> k
```

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.

1.1 Monaden

Für Haskell ist eine Monade zunächst einmal nichts anderes als eine Instanz der Klasse `Monad`. Instanzen dieser Klasse müssen die Funktionen `return`, `(>>=)` und `(>>)` definieren und zwar mit den folgenden Typen.

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

Offensichtlich können wir `StateTransformer s` als Instanz der Klasse `Monad` deklarieren.

```
instance Monad (StateTransformer s) where
  return x = ST $ \s -> (x, s)
  m >>= k = ...
```

Die wahrscheinlich häufigste Verwendung der Klasse `Monad` ist Zustands-Übergangssysteme der einen oder anderen Art darzustellen und als ein solches wird in Haskell auch die Ein-/Ausgabe dargestellt, aber es gibt auch Beispiele für die eine andere Intuition besser geeignet ist. Tatsächlich stellt jeder Typ-Konstruktor, der eine Instanz der Klasse `Monad` ist und für den darüber hinaus gilt:

```
p >>= return = p
```

und

```
return e >>= q = q e
```

und

```
(p >>= q) >>= r = p >>= s where s x = (q x >>= r)
```

eine Monade dar.

In Haskell gibt es eine besondere Syntax, um monadische Berechnungen aufzuschreiben: die `do`-Notation. Mit ihr können wir `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 }
```

Das ist aber nur syntaktischer Zucker, denn

```
do { m; k } = m >> do { k }
```

Hier kann man noch nicht auf das möglicherweise vorhandene Ergebnis der Berechnung `m` zugreifen. Hat `m` ein Ergebnis (zusätzlich zum neuen Zustand), dann kann man `do { r <- m; k }` schreiben und das entspricht `m >>= \r -> do { k }`.

Wir werden zunächst aber bei den Zustandsmonaden bleiben, denn wie schon erwähnt wird IO in Haskell als eine Zustandsmonade angesehen.

1.2 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
readLine :: Read a => IO a
```

und für Dateien

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

Eine Variation des Rechners mit IO ist

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

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 `k` 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.

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 ()

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

1.3 Andere Monaden

Wie bereits erwähnt, gibt es auch Monaden, die keine Zustandsübergänge darstellen. Ein Beispiel sind Listen. In Haskell haben wir den Typ

```
data [a] = []       -- nil
         | a : [a]  -- cons a (list a)
```

Die Monaden-Gesetze wurden bereits erwähnt: `p >>= return = p`, `return e >>= q = q e` und `(p >>= q) >>= r = p >>= s where s x = (q x >>= r)`. Würden wir `return x = []` definieren, dann könnten wir zwar das erste Gesetz erfüllen, aber i.a. nicht das zweite, denn die Information über `e` würden wir verlieren. Also wird `return x = [x]` definiert. Weiterhin ist für Listen definiert

```
[]     >>= f  =  []
(x:xs) >>= f  =  f x ++ (xs >>= f)
```

Man überzeugt sich leicht davon, daß diese Definition die Monaden-Gesetze erfüllt.