Monads

Nils Decker, <wi4157@fh-wedel.de> 30. April 2002

Simulation von Programmzuständen, Ein- und Ausgabe, Ausnahmen und Nichtdeterminismus

Inhaltsverzeichnis

[PDF] `<monads.pdf>` *[Präsentation]* `<monads-pres.html>`

1 Einleitung

Haskell ist eine *pure* funktionale Sprache. Das bedeutet, daß das Ergebnis einer Funktion nur von ihren Parametern abhängt. Eine Funktion kann keinerlei Seiteneffekte auslösen, oder von äußeren Zuständen abhängig sein. Dadurch ist es einfach, über eine Funktion zu Argumentieren, oder deren Korrektheit zu beweisen.

Weil die Sprache *pur* ist, macht es keinen Unterschied, in welcher Reihenfolge Werte berechnet werden. Wenn ein Wert für das Ergebnis des gesamten Programms nicht benötigt wird, muß dieser auch nicht berechnet werden. Es ist also nicht vorherzusehen, wann und ob, eine bestimmte Funktion ausgewertet wird.

Mit einem *Monad* ist es unter Anderem möglich, die Reihenfolge von Berechnungen festzulegen, und deren Berechnung zu erzwingen.

Die hier vorgestellten Konzepte von Monads sind in der Sprache Haskell verdeutlicht, können aber direkt auf die meisten anderen funktionalen Sprachen, wie z.B. ML, Erlang oder Scheme [2] übertragen werden.

1.1 Das Problem der Seiteneffekte bei der Ein- und Ausgabe

Weil sich die Reihenfolge der Auswertung nicht vorhersehen lässt, hätte das Einführen von verborgenen Seiteneffekten schwere Auswirkungen auf die Semantik der Sprache.

Es wäre z.B. nicht mehr möglich mathematische Aussagen über das Verhalten von Funktionen aufzustellen. Als Beispiel sei eine Funktion `readInt` definiert, die einen Integer-Wert einliest, und eine Funktion `foo`, die `readInt` verwendet.

```
readInt :: Int

foo = readInt - readInt
```

Es ist nicht möglich über das Verhalten der Funktion `foo` eine Aussage zu treffen. Das Ergebnis dieser Funktion ist nicht nur von einer Eingabe von außen, sondern auch von der Reihenfolge der Auswertung abhängig. Ein optimierender Compiler könnte folgern, daß `foo = 0` ist, und somit alle Seiteneffekte verhindern.

Es ist also notwendig, eine Struktur einzuführen, die es ermöglicht, sequentielle Abläufe zu beschreiben.

2 Struktur von Monads

Ein Monad ist ein Typ auf den die Operationen *bind* und `return` definiert sind. In Haskell wird für den Operator *bind* der Operator >>= verwendet. Der Operator >>= wird verwendet, um 2 Berechnungen aneinander zu binden, und eine neue, zusammengesetzte, Berechnung zu formen. Das Ergebnis der 1. Berechnung wird der 2. Berechnung als Parameter übergeben. `return` fügt das Ergebnis einer Berechnung in das Monad ein.

Jede Berechnung die in einem Monad verwendet wird, liefert ihr Ergebnis in dem Monad *verpackt* zurück.

Die Definition der Klasse `Monad` in der Hakell Prelude ist

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

  m >> k = m >>= \_ -> k
  fail s = error s
```

Die Operatoren >> und `fail` sind für die Definition eines Monads nicht notwendig, können aber praktisch sein. Der Operator >> bindet, genau wie >>= zwei Berechnungen aneinander, verwirft aber das Ergebnis der ersten Berechnung. `fail` kann verwendet werden, um das Fehlschlagen einer Berechnung zu signalisieren.

In der Definition aus der Prelude ruft `fail` die Haskell-Funktion `error` auf und führt somit zum Abbruch des Programms. Es ist aber möglich, wie z.B. bei dem Exception-Monad, die `fail`-Funktion neu zu definieren, um das Fehlschlagen der Berechnung an einer anderen Stelle des Programmes behandeln zu können.

`return` ist das Nullelement der Operation >>=, und >>= ist assoziativ:

```
return a >>= k   = k a
m >>= return     = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Die Signatur einer Funktion, die innerhalb des Monads arbeitet, ist `(Monad a) => a b`. Das Ergebnis der Berechnung wird in das Monad verpackt.

```
hello :: String -> IO ()
hello x = putStr ("Hello " ++ x)
```

Für einig Monads ist es noch sinnvoll, eine Funktion *extract* zu definieren. Diese Funktion extrahiert das Ergebnis einer Berechnung aus dem Monad, damit das Ergebnis in funktionalen Programmteilen benutzt werden kann, ohne das Monad zu verwenden. Die Funktion *extract* darf aber nur definiert werden, wenn alle Berechnungen innerhalb des Monads ineinander abgeschlossen sind, und somit als Gesamtheit gesehen funktional sind. Ein Beispiel für eine solche Berechnung ist ein Programmteil, der einen internen Status verwaltet, der am Anfang der Berechnung initialisiert wird, und nach der Berechnung nicht mehr gebraucht wird. Es ist nicht möglich, die Funktion *extract* für Berechnungen mit externen Seiteneffekten zu definieren.

2.1 Das Identity-Monad

Das einfachste Monad ist das Identity-Monad. Die verwendung dieses Monads hat keine besonderen Auswirkungen. Berechnungen werden wie sonst auch in Haskell durchgeführt. Es kann jedoch trotzdem sinnvoll sein, dieses Monad zu verwenden, um das Monad später um weitere Eigenschaften zu erweitern.

Das Identity-Monad ist definiert durch

```
data Id a  = Id a
instance Monad Id where
  (Id x) >>= f  = f x
  return x      = Id x
  (Id _) >>  f  = f
  fail s        = error s

extract :: Id a -> a
extract (Id x) = x
```

Ein Beispiel für eine Berechnung mit dem Identity-Monad ist:

```
mul      :: Int -> Int -> Id Int
mul x y  =  return (x * y)

div_     :: Int -> Int -> Id Int
div_ _ 0 =  fail "div 0"
div_ x y =  return (x `div` y)
```

```
one      :: Int -> Id Int
one x    =  mul x x >>= \_ ->
            mul x 1 >>= \y ->
            div_ y x

> one 5 ==> 1
> one 0 ==> Fehler des Interpreters: "div 0"
```

Die Berechnungen in diesem Beispiel lassen sich durch einfaches Anwenden der Operatoren auf `one x = (x * 1) / x` reduzieren. Später werden wir jedoch weitere Monads vorstellen, die das Verhalten verändern, so dass ganz andere Effekte entstehen.

2.2 die `do`-Notation

Um dem Programmierer das schreiben von den vielen Lambda-Funktionen abzunehmen, gibt es in Haskell das `do`-Konstrukt. Es erlaubt eine einfachere und kürzere Schreibweise von Berechnungen mit Monads.

Diese beiden Funktionsdefinitionen sind identisch.

```
one x    =  mul x x >>
            mul x 1 >>= \y ->
            div_ y x
```

```
one x   = do mul x x
             y <- mul x 1
             div_ y x
```

Es ist möglich Zwischenergebnisse mit `let` an einen Namen zu binden.

```
one x   = do mul x x
             let eins = 1 in
             y <- mul x eins
             div_ y x
```

Die `do`-Notation verändert die Ausdruckskraft der Sprache nicht, weil sich Ausdrücke in der `do`-Notation sehr einfach in Ausdrücke mit `>>=,` und `return` umwandeln lassen.

Programmstücke, die in der `do`-Notation geschrieben sind, sehen imperativen Programmen in normalen, imperativen Programmiersprachen sehr ähnlich.

3 Verschiedene Anwendungen von Monads

Monads erlauben nicht nur die Reihenfolge von Berechnungen festzulegen. Ein Monad kann beliebig viele Informationen enthalten und dem Programm die möglichkeit bieten diese auszulesen oder zu verändern. Desweiteren ist es möglich, nur durch Veränderung der Struktur des Monads, Ausnahmen und Nichtdeterminismus zu implementieren. Mit Hilfe von Monads ist sogar das destruktive Verändern von Daten möglich.

3.1 Status

Durch die Verwendung eines Monads ist genau beschrieben, in welcher Reihenfolge Berechnungen erfolgen sollen. Es ist deshalb möglich, einen Status in einem Monad einzuführen, der in der festgelegten Reihenfolge von Berechnungen gelesen oder verändert werden kann.

Das Monad wird so definiert, daß jede Berechnung des Monads eine Funktion ist, die den Ausgangsstatus in einen Endstatus überführt.

```
-- a ist der Typ des Status
-- b ist der Typ des Ergebnisses der Berechnung
data State a b = State ( a -> (a, b) )

instance Monad (State a) where
  return x = State (\s -> (s,x) )

  (>>=)  :: (State a) -> ( a -> (State b) ) -> (State b)
  (State f1) >>= f2
     = State ( \st1 -> let (st2, y)       = f1 st1
                           (State trans ) = f2 y
                       in trans st2 )
```

Nach dieser Definition des State-Monads wird ein Status von Berechnung zu Berechnung durchgereicht.

Es können jetzt zwei weitere Berechnungen definiert werden, die den Status lesen oder schreiben.

```
readState    :: State a a
readState    =  State   (\st -> (st, st) )

writeState   :: a -> State a ()
writeState s = State (\_ -> ( s, () ))
```

Es ist jetzt möglich, z.B. das Beispiel um einen Zähler erweitern, wieviele Multiplikationen stattgefunden haben.

```
type CountState a = State Int a

tick :: CountState ()
tick = do a <- readState
          writeState (a+1)

extract :: CountState a -> (Int, a)
extract (CountState st) = st 0

mul      :: Int -> Int -> CountState Int
mul x y  =  do tick
               return (x * y)

test = extract . one

> test 5 ==> (2, 1)
```

In diesem Beispiel liefert `test` ein Tupel zurück, daß die Anzahl der Aufrufe von `mul` und das Ergebnis der Berechnung enthält.

Bemerkenswert ist, daß sich die Definitionen von `div` und `one` nicht geändert haben, obwohl sie jetzt, im Gegensatz zum Beispiel mit dem Identity-Monad, einen Status durchreichen.

3.2 Ausnahmen

In C wird häufig der Fehlerfall durch einen speziellen Rückgabewert gezeigt. Dies hat den Nachteil, daß der Wertebereich des Ergebnisses beschränkt wird, und nach jedem Funktionsaufruf getestet werden muß, ob der Funktionsaufruf erfolgreich war. In C++, Java und anderen Sprachen wurden Exceptions eingeführt, die einen Fehler darstellen, und *nach oben* durchgereichten, bis der Fehler behandelt wird. Diese Exceptions können als ein spezieller Rückgabewert gesehen werden, auf den nach jedem Funktionsaufruf geprüft wird.

Die polymorphen Datentypen von Haskell erlauben es, diesen speziellen Fehlertyp zu verwenden, ohne den Wertebereich des Ergebnisses einzuschränken. Die Definition des Monads kann verwendet werden, um die Fehlerübeprüfung nach jedem *Aufruf* vorzunehmen.

Eine Ausnahmen-Behandlung kann einfach realisiert werden, in dem >>= so definiert wird, daß bei einer fehlgeschlagenen Berechnung, die darauf folgenden Berechnungen nicht mehr berechnet werden, und sofort der Fehler zurückgegeben wird.

```
data Exception a = Success a | Error String

instance Monad Exception where
  return x = Success x
  fail s   = Error s

  (Success x) >>= f  = f x
  (Error s  ) >>= f  = (Error s)

catch_ :: Exception a -> (Exception a -> Exception a) -> Exception a
catch_ (Success a) _ = (Success a)
catch_ (Error s)   f = f (Error s)

handler :: Num a => Exception a -> Exception a
handler (Error s) = Success 999

div_     :: Int -> Int -> Exception Int
div_ _ 0 =  fail "div 0"
div_ x y =  return (x `div` y)

> one 5 ==> (Success 1)
> one 0 ==> (Error "div 0")
> catch_ (one 0) handler ==> 999
```

In diesem Beispiel wird der Fehler in `div_` mit `catch_` abgefangen, und von `handler` bearbeitet. Obwohl sich die Definitionen von `mul` und `one` nicht verändert haben, wird in ihnen die Ausnahme korrekt behandelt.

3.3 Nichtdeterminismus

Um Nichtdeterminismus zu implementieren, wird das Exception-Monad verändert. Eine Funktion liefert eine Liste aller erfolgreich berechneten Ergebnisse. Eine fehlgeschlagene Berechnung hat die leere Liste als Ergebnis. Die nachfolgenden Berechnungen werden auf alle Ergebnisse der voherigen Berechnungen angewendet.

```
data Nd a = Nd [a]
```

```
instance Monad Nd where
  return x = Nd [x]
  fail s   = Nd []

  (Nd [] )    >>= f = Nd []
  (Nd (x:xs)) >>= f = let (Nd ys) = f x
                          (Nd zs) = (Nd xs) >>= f in
                          (Nd (ys ++ zs) )


fromTo   :: Int -> Int -> Nd Int
fromTo a b = Nd [a..b]

many     :: Int -> Nd Int
many x   = do y <- fromTo 0 x
              one y

> many 2 ==> [1, 1]
> many 0 ==> []
```

In diesem Beispiel liefert `fromTo` eine Liste von Ergebnissen der Funktion. Diese werden jeweils einzeln von `one` verarbeitet. Die erfolgreichen Berechnungen von `one` werden gesammelt, und bilden das Ergebnis von `many`.

Wenn am Ende einer solchen Berechnung nur ein gültiges Ergebnis benötigt wird, werden durch die lazy evaluationnur die benötigten Zwischenergebnisse berechet. Durch diese Eigenschaft lassen sich z.B. einfach nichtdeterministische Parser bauen.

3.4 IO

Das Monad `IO` ähnelt dem Status-Monad. Der Status innerhalb des Monads entspricht der Umgebung des Programmes, also der Welt. Jede Funktion, die IO-Operationen ausführt, verändert diesen Status der "Welt". Das `IO`-Monad ist ein ADT, es ist also nicht möglich, Teile der *Welt* aus dem Monad zu extrahieren, und außerhalb des Monads zu verändern.

Alle Berechnungen, die IO-Operationen ausführen, haben als Ergebnistyp das `IO`-Monad.

```
-- data IO a = IO(Welt -> (Welt, a))

mul      :: Int -> Int -> IO Int
mul x y  =  do putStrLn( "mul " ++ (show x) ++ " " ++ (show y) )
               return (x * y)
```

Für das `IO`-Monad gibt es keine `extract`-Funktion. Es ist nicht möglich, ein Ergebnis, daß in dem `IO`-Monad berechnet wurde, zu extrahieren. Alle IO-Operationen eines Programmes sind durch dieses Monad genau vorhersagbar.

3.4.1 unsafePerformIO

Es gibt eine Funktion, die für das `IO`-Monad wie die `extract`-Funktion arbeitet. Diese Funktion heißt `unsafePerformIO`. IO-Operationen die mittels `unsafePerformIO` ausgeführt werden, sind nicht vorhersagbar. Sie können zu jedem Zeitpunkt der Programmausführung in beliebiger Reihenfolge auftreten.

Die Benutzung von `unsafePerformIO` sollte, soweit möglich, vermieden werden. Durch unvorsichtige Benutzung ist es möglich, das Typsystem von Haskell auszuhebeln, und Programme zu schreiben, die auf ungültige Speicherbereiche zugreifen.

Es gibt jedoch ein paar Anwendungen, bei denen `unsafePerformIO` praktisch sein kann.

- Trace-Nachrichten

```
trace :: String -> a -> a
trace s x = unsafePerformIO ( putStrLn s >> return x )
```

- Alloziierung einer globalen Variable (globale Variablen sollten möglichst vermieden werden!)

```
noOfOpenFiles :: IORef Int
noOfOpenFiles =  unsafePerformIO (newIORef 0)
```

- Aushebeln des Typsystems [5]

```
cast :: a -> b
cast x = let bot = bot
             r = unsafePerformIO (newIORef bot)
         in  unsafePerformIO ( do writeIORef r x
                                  readIORef r )
```

4 Kontrollstrukturen mit Monads

Mit den Monads ist es möglich, imperative Programme in einer funktionalen Programmiersprache zu schreiben. Es ist, durch die Verwendung von *higher order functions*, einfach möglich die üblichen Kontrollstrukturen der imperativen Programmiersprachen nachzubilden.

```
foreach :: (Monad m) => [a] -> (a -> m ()) -> m ()
foreach []     _ = return ()
foreach (x:xs) f = do f x
                      foreach xs f

> foreach [1..10] (\x ->
>   putStrLn (show x)
> )

while :: (Monad m) => m Bool -> m () -> m ()
while cond f = do res <- cond
                  if res then (f >> (while cond f))
                         else (return ())

> while (return True) (
```

```
>   putStrLn "Hello world"
> )
```

5 Kombinieren von Monads

Es ist einfach möglich, die verschiedenen Mondads, die hier vorgestellt wurden, zu kombinieren. In diesem Beispiel wird das Status-Monad mit dem IO-Monad kombiniert. Eine Berechnung in dem IOState-Monad ist eine Berechnung in dem IO-Monad, die einen Ausgangsstatus in einen Endstatus überführt.

```
data IOState a b = IOState ( a -> IO (a, b) )

instance Monad (IOState a) where
    return v = IOState ( \s -> return (s, v))

    IOState f1 >>= f2
      = IOState ( \st1 -> do (st2, r) <- f1 st1
                             let (IOState trans ) = f2 r in
                               trans st2 )

readState    :: IOState a a
readState    =  IOState   (\st -> return (st, st) )

writeState   :: a -> IOState a ()
writeState s = IOState (\_ -> return ( s, () ))
```

Bei `readState` und `writeState` ist zu beachten, daß das `return` zu dem IO-Monad gehört.

`liftIO` fügt eine Berechnung in dem IO-Monad in das IOState-Monad ein.

```
liftIO       :: IO b -> IOState a b
liftIO ioa   =  IOState ( \s -> do a <- ioa
                                   return (s, a) )

extractIO    :: a -> IOState a b -> IO b
extractIO init (IOState f)
             =  do ( st, res ) <- f init
                   return res

> foo x = do
>   writeState x
>   while ( do { nr <- readState; return (nr /= 1) } ) ( do
>     nr <- readState
>     liftIO (putStrLn (show nr))
>     let nr´ = if ( nr ´mod´ 2 == 0 ) then (nr ´div´ 2)
>                                      else (3 * nr + 1)
>     writeState nr´
>     return ()
>    )
> extractIO 0 ( foo 10 )
```

Durch die Verwendung des IOState-Monads ist es möglich, imperative Programme zu schreiben, die sich kaum von Programmen in normalen imperativen Sprachen unterscheiden.

6 Vergleich mit imperativen Programmiersprachen

Durch die Verwendung von Monads werden in einer funktionalen Programmiersprache Berechnungen in einer vorbestimmten Reihenfolge ausgeführt. Dies kommt dem Einbetten einer imperativen Programmiersprache in die funktionale Sprache gleich. Die eingebette Sprache ist jedoch flexibler als die meisten *normalen* imperativen Programmiersprachen. Durch verändern der Definition des Monads lässt sich die Semantik der eingebetteten Sprache an das jeweilige Problem anpassen. Dies liegt daran, daß in imperativen Programmiersprachen Berechnungen immer implizit die Welt, also das IO-Monad, verwendet wird. Das Exception-Monad wird bei vielen Sprachen durch Spracherweiterungen unterstützt. Andere Monads, wie z.B. das Nichtdeterminismus-Monad läßt sich nur sehr schwierig in imperativen Programmiersprachen nachbilden.

7 Literatur

[1]

Simon Thomson, The Craft of Functional Programming, Addison-Wesley 1999, S. 383 ff.

[2]

Unbekannter Author, *Monadic Programming in Scheme* <http://okmij.org/ftp/Scheme/monad-in-Scheme.html>

[3]

Philip Wadler, *The essence of functional programming* <http://www.research.avayalabs.com/user/wadler/topics/monads.html>

[4]

Simon Peyton Jones, *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions and foreign-language calls in Haskell* <http://research.microsoft.com/users/simonpj/Papers/marktoberdorf/>

[5]

John Launchbury, Jeffrey R. Lewis, Byron Cook, *On embedding a microarchitectural design language within Haskell* <http://www.cse.ogi.edu/~jlewis/hawkpaper.ps.gz>

[6]

John Hughes, Magnus Carlsson, *Systematic Design of Monads* <http://www.cs.chalmers.se/~augustss/AFP/monads.html>

Alle hier angegebenen URLs waren am 28. April 2002 erreichbar.