Monaden

Sebastian Wagner

26.11.07

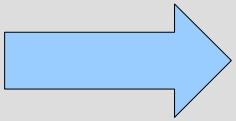
Motivation

Haskell ist eine reine funktionale Programmiersprache, d.h. Programme werden als mathematische Funktionen aufgefasst.

Es werden also Berechnungen nur durch Funktionsanwendungen durchgeführt, wodurch einige erwünschte Nebenwirkungen nicht zulässig sind.

Motivation

Um diese erwünschten Wirkungen, wie z.B. implizite Zustandsänderungen oder Exception-Behandlungen beschreiben zu können, wurde eine neues Konzept entwickelt.



Monaden

Was ist eine Monade?

Eine Monade besteht aus dem Tripel:

(M, return, >>=)

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

Die 3 Monaden Gesetze

1. (return x) >>= f == f x

2. m >>= return == m

3. (m >>= f) >>= g == m >>= (\x -> f x >>= g)

Die do-Notation

Durch die do-Notation wird das schreiben des bind (>>=) durch do ersetzt und somit das Programmieren erleichtert.

```
foo r = do
    x <- f r
    x <- g x
    x <- h x
    return x
```

```
foo r = f r >>= ((\x -> g x)
            >>= ((\x -> h x)
            >>= (\x -> return x)))
```

Was gibt es für Monaden-Typen?

Identity Monade
Maybe Monade
List Monade
Exception Monade
I/O Monade
State Monade
Reader Monade
Writer Monade
Continuation Monade

Die Exception Monade

```
data Term = Con Int | Div Term Term

eval :: Term
eval(Con a) = a
eval(Div t u) = eval t / eval u
```

Die Exception Monade

```
answer, error :: Term

answer = (Div(Div(Con 16)(Con 2))(Con 2))
error = (Div(Con 1)(Con 0))

data M a = Raise Exception | Return a
type Exception = String
```

Die Exception Monade

```
eval :: Term -> M Int

    eval (Con a) = Return a
    eval (Div t u) = case eval t of
          Raise e -> Raise e
          Return a -> case eval u of
             Raise e -> Raise e
             Return b -> if b == 0
                         then Raise „divide by zero“
                         else Return(a/b)
```

Die Exception Monade

Auf unsere beiden Beispiele angewendet erhalten wir als Ergebnis:

eval answer = (Return 4)
eval error = (Raise „divide by zero“)

Die Exception Monade

```
data M a = Raise Exception | Return a
type Exception = String

retrun :: a -> M a
return a = Return a

raise :: Exception -> M a
raise e = Raise e

(>>=) :: m a -> (a -> m b) -> m b
m >>= k = case m of
Raise e -> Raise e
Return a -> k a
```

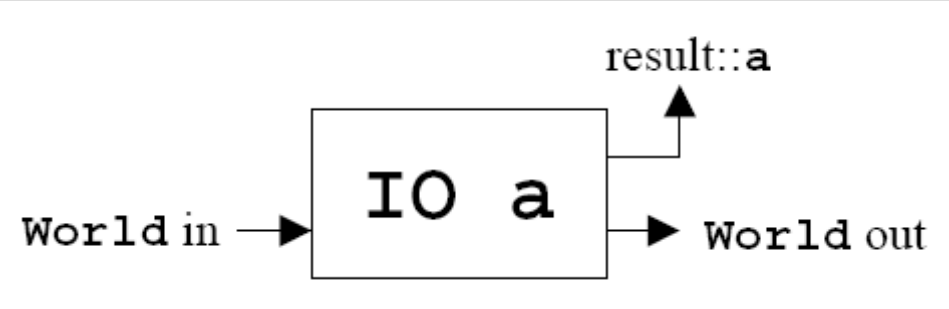
Die Exception Monade

Um nun die Fehlerbehandlung zu unserer Monade hinzuzufühgen erweitern wir:

if b == 0
then raise „divide by zero“
else return (a/b)

Die I/O Monade

Type IO a = World -> (a, World)



Quelle: Tackling the Awkward Squad, by Simon PEYTON JONES

Anwendung der I/O Monade

```
getChar :: IO Char
getChar = do x <- gehtChar
             putChar x
             return x
```

Anwendung der I/O Monade

```
getLine :: IO String

    getLine =
    getChar >>=
    \x -> if x == ’\n’
          then return []
               else
                   getLine >>= \xs -> return (x:xs)
```

('\n' steht für den Zeilenumbruch)

Zusammenfassung

Mit Monaden lassen sich erwünschte Nebeneffekte modellieren.

Eine Monade ist ein Typkonstruktor auf den return und bind (>>=) definiert ist.

Eine Monade muss gewisse Richtlinien einhalten (Monaden Gesetzte)

Zusammenfassung

Das schreiben von Programmen, welche Monaden benutzen, wird durch die do-Notation vereinfacht.

Wir haben viele verschiedene Monadentypen wie die Exception- und die I/O – Monade kennengelernt.

Durch Monaden können Programme geschrieben werden, bei denen der Benutzer interaktive Eingaben tätigen kann.

Danke für Ihre Aufmerksamkeit!