

Ein Interpreter mit Erweiterungen

Wiederholung: Monaden

December 17, 2007

Definition

```
data Term = Con Integer
          | Bin Term Op Term
          deriving (Eq, Show)
```

```
data Op = Add | Sub | Mul | Div
        deriving (Eq, Show)
```

Auswertung

```
eval          :: Term -> Integer
eval (Con n)  = n
eval (Bin t op u) = sys op (eval t) (eval u)

sys Add      = (+)
sys Sub      = (-)
sys Mul      = (*)
sys Div      = div
```

Wie war das doch gleich mit den Monaden?

Die Klasse Monad

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

Die Identity Monade

```
newtype Id a = Id a
```

```
instance Monad Id where
```

```
    return x      = Id x
```

```
    x >>= f      = let Id y = x in f y
```

Monadischer Interpreter

```
eval :: Term -> Id Integer
```

```
eval (Con n)      = return Id n
```

```
eval (Bin t op u) = eval t >>= \v ->
                      eval u >>= \w ->
                      return Id (sys op v w)
```

Exception

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

eval             :: Term -> Exception Integer
eval (Con n)    = Return n
eval (Bin t op u) = case eval t of
                      Raise s -> Raise s
                      Return v -> case eval u of
                                    Raise s -> Raise s
                                    Return w ->
                                        if (op == Div && w == 0)
                                        then
                                            Raise "div by zero"
                                        else
                                            Return (sys op v w)
```

Exception

```
instance Monad Exception where
  return a  = Return a
  m >>= f   = case m of
                Raise s -> Raise s
                Return v -> f v
  fail s    = Raise s
```

Interpreter

```
eval :: Term -> Exception Integer
eval (Con n)      = return n
eval (Bin t op u) = eval t >>= \v ->
                      eval u >>= \w ->
                      if (op == Div && w == 0)
                        then fail "div by zero"
                        else return (sys op v w)
```

Trace

```
newtype Trace a = Trace (a, String)

eval :: Term -> Trace Integer
eval e@(Con n)      = Trace (n, trace e n)
eval e@(Bin t op u) =
  let Trace (v, x) = eval t in
  let Trace (w, y) = eval u in
  let r = sys op v w in
  Trace (r, x ++ y ++ trace e r)

trace t n = "eval (" ++ show t ++ ") = "
           ++ show n ++ "\n"
```


Trace

```
instance Monad Trace where
  return a = (a, "")
  m >>= f  = let Trace (a, x) = m in
              let Trace (b, y) = f a in
              Trace (b, x ++ y)

output    :: String -> Trace ()
output s  = Trace ((), s)
```

Auswertung

```
eval :: Term -> Trace Integer
eval e@(Con n) = output (trace e n) >>
                  return n
eval e@(Bin t op u) = eval t >>= \v ->
                       eval u >>= \w ->
                       let r = sys op v w in
                       output (trace e r) >>
                       return r
```

Bisher verwendet

- Identity Monade
- Exception Monade
- State Monade
- Writer Monade

Praktisch für

- Erstellen von Berechnungen, die "Nothing" als Wert zurückgeben
- komplexe Datenbankabfragen, Operationen auf Wörterbüchern, ...

Praktisch für

- Erstellen von Berechnungen, die "Nothing" als Wert zurückgeben
- komplexe Datenbankabfragen, Operationen auf Wörterbüchern, ...

Definition

```
data Maybe a = Nothing | Just a
```

```
instance Monad Maybe where  
    return x      = Just x
```

```
Nothing  >>= f = Nothing  
(Just x) >>= f = f x
```

Praktisch für

- Berechnungen über Folgen von nicht-deterministischen Ergebnissen
- Parsen von mehrdeutigen Grammatiken

Praktisch für

- Berechnungen über Folgen von nicht-deterministischen Ergebnissen
- Parsen von mehrdeutigen Grammatiken

Definition

```
instance Monad [] where
  return x = [x]
  m >>= f = concatMap f m
```

where

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

Notwendig für

- jegliche Art von Input und Output
- Seiteneffekt-behaftete Operationen
- Implementierung ist Maschinenabhängig!

Was ist so speziell an der IO Monade?

Angenommen,

```
getChar :: Char
```

Wie soll dann der folgende Ausdruck ausgewertet werden?

```
get2chars = [getchar, getchar]
```


Was macht eigentlich Hugs/Ghci?

Programmcode in Datei test.hs

```
x = 1 + 1
```

Im Interpreter

```
Prelude> :l test.hs  
[1 of 1] Compiling Main (test.hs, interpreted)  
Ok, modules loaded: Main.  
*Main> x  
2
```