Ein Interpreter mit Erweiterungen

Wiederholung: Monaden

November 24, 2008

Eine einfache Termsprache

Definition

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

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

Ein einfacher Interpreter

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
```

Erweitern des Interpreters

Mögliche Erweiterungen

- Fehlerbehandlung
- Zählen der Auswertungsschritte
- Variable, Zustand
- Ausgabe

. . . ohne das Grundgerüst des Interpreters zu verändern!

Interpreter mit Fehlerbehandlung

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)
```

Die Rettung: Monaden

Die Klasse Monad

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

Ein einfacher Auswerter in monadischer Notation

Die Identitätsmonade

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

Monadischer Interpreter mit Fehlerbehandlung

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)
```

Interpreter mit Protokoll

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"
```

Monadischer Interpreter mit Protokoll I

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)
```

Monadischer Interpreter mit Protokoll II

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
```

Interpreter mit Reduktionszähler

Count

```
type Count a =  Int -> (a, Int)

eval                :: Term -> Count Integer
eval (Con n)        = \i -> (n, i)
eval (Bin t op u) = \i -> let (v, j) = eval t i in
                          let (w, k) = eval u j in
                           (sys op v w, k + 1)
```

Monadischer Interpreter mit Reduktionszähler

Die Zustandsmonade

State

```
data ST s a = ST (s -> (a, s))
exST (ST sas) = sas

instance Monad (ST s) where
  return a = ST (\s -> (a, s))
  m >>= f  = ST (\s -> let (a, s') = exST m s in
                       exST (f a) s')

type Count a = ST Int a

incr :: Count ()
incr =  ST (\i -> ((), i + 1))
```

Monadischer Interpreter mit Reduktionszähler

Implementierung

Auswertung

```
eval :: Term -> Count Integer
eval (Con n)       =  return n
eval (Bin t op u) =  eval t >>= \v ->
                     eval u >>= \w ->
                     incr   >> ->
                     return (sys op v w)
```

Typische Monaden

Bisher verwendet

- Identitätsmonade (Identity)
- Exception Monade
- Zustandsmonade (State)
- Ausgabemonade (Writer)

Nicht jeder Datentyp kann eine Monade sein

Monadengesetze

return ist Linkseinheit

```
return x >>= f          == f x
```

return ist Rechtseinheit

```
m >>= return            == m
```

bind ist assoziativ

```
m >>= \x -> (n >>= f) == (m >>= \x -> n) >>= f
```

Die Maybe Monade

Praktisch für

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

Die Maybe Monade

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
```

Die List Monade

Praktisch für

- Berechnungen über Folgen von nicht-deterministischen Ergebnissen
- Backtracking, z.B. Parsen von mehrdeutigen Grammatiken

Die List Monade

Praktisch für

- Berechnungen über Folgen von nicht-deterministischen Ergebnissen
- Backtracking, z.B. Parsen von mehrdeutigen Grammatiken

Definition

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

where

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

Die IO Monade

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]
```