Informatik-Seminar 2003 - Thema: Monaden (Kapitel 10)

Stefan Neumann

2. Dezember 2003

Inhalt

Einleitung

Gegeben seien folgende Funktionen:

```
inputInt  :: Int

inputDiff :: Int
inputDiff =  inputInt - inputInt
```

- Ergebnis ist von der Auswertungsreihenfolge abhängig
- Compiler kann durch “Optimierung” konstant 0 zurückliefern

Die IO()-Notation

- Der Ausdruck IO() verweist auf eine **Aktion**
- Auswertung dieser Aktion ist die Durchführung
- Bsp:

```
done :: IO()
done = return()
```

- Abstrakter Datentyp(Kapitel 8)
- Allgemeiner Typ: IO a

Ausgabe von Char

- Schreiben eines Char

```
putChar :: Char -> IO()
```

- Beispiel

```
? putChar '!'
!
```

Kombinieren von Kommandos

- Operator zum sequentialisieren
- Ein Operator wäre (>>)

```
(>>) :: IO() -> IO() -> IO()
```

- p >> q: Erst wird das Kommando p ausgeführt, dann q

Beispiele für >>

- Implementation von putStr durch write

```
write          :: String -> IO()
write []       =  done
write (c:cs)   =  putChar c >> write cs
```

- Oder die Implementation mit foldr

```
write :: foldr (>>) done.map putChar
```

- Einen String mit Zeilenumbruch ausgeben

```
writeln :: String ->IO()
writeln cs = write cs >> putChar '\n'
```

Eingabe eines Char

- Einlesen eines Buchstaben durch getChar

```
getChar:: IO Char
```

- Die Funktion getChar liest einen Buchstaben ein und gibt ihn wieder aus
- Expressions vom Typ IO a mit a $/=$ () können nicht ausgegeben werden

```
? getChar
ERROR: Cannot find show function for IO Char
```

- Mit done kombinieren

```
? getChar >> done
x
```

Der Bind-Operator (>>=)

- Operator >> nur interessant, wenn bei p >> q der Rückgabewert von p uninteressant ist
- Signatur der Funktion:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Wenn p und q Funktionen sind, dann wird bei p >>= q
p ausgeführt und der Rückgabewert x vom Typ a an q übergeben,
also wird q mit x aufgerufen: q x

Beispiele für Bind-Operator (>>=)

- Zeichen einlesen und sofort wieder ausgeben:

```
? getChar >>= putChar
xx
```

- n-Zeichen einlesen

```
readn :: Int -> IO String
readn 0 = return []
readn (n+1) = getChar >>= q
   where q c = readn n >>= r
      where r cs = return (c:cs)
```

- Eine ganze Zeile einlesen

```
readln :: IO String
readln = getChar >>= q
   where q c = if c ==’\n’
               then return []
               else readln >>= r
                  where r cs = return (c:cs)
```

Die Do-Notation

- Alternative schreibweise für den Bind-Operator mit geschachtelten where-clauses
- In dem *do*-Block werden alle Kommandos sequentiell ausgeführt
- Ergebnis einer Funktion kann mit $<-$ an eine Variable gebunden (**single** assignment)

Beispiele für do-Notation

- n Zeichen einlesen

```
readn :: Int -> IO String
readn 0     = return []
readn (n+1) = do c <- getChar
                 cs <- readn n
                 return (c:cs)
```

- Eine ganze Zeile einlesen

```
readln :: IO String
readln = do c <- getChar
            if c =='\n'
            then return []
            else do cs <- readln
                    return (c:cs)
```

Zusammenfassung IO-Monad

- Was können wir aus dem IO-Monad lernen?
- Klasssendefinition eines Monad:

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

- Den >>-Operator durch den Bind-Operator ausdrücken (Default-Implementation):

```
m >> k =  m >>= \ _ -> k
```

Der einfache Auswerter

- Datenstruktur:

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

- Funktion eval berechnet das Ergebnis:

```
eval :: Term -> Int
eval (Con x)   = x
eval (Div t u) = (eval t) `div` (eval u)
```

Erweiterung des Evaluierers um Exceptionbehandlung

- Problem beim einfachen Evaluierer?
  - Kein Abfangen der Division durch null
- Folge: unkontrollierter Programmabbruch
- Lösung: Exceptionbehandlung

- Datenstruktur der Exception

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

- der modifizierte Evaluierer

```
evalExc :: Term -> Exc Int
evalExc (Con x) = Return x
evalExc (Div t u)
    = h (evalExc t)
      where
      h (Raise e) = Raise e
      h (Return x) = h'(evalExc u)
                     where
                     h' (Raise e) = Raise e
                     h' (Return y)
                        = if y == 0
                          then Raise "Division by zero"
                          else Return (x `div` y)
```

Erweiterung des Evaluierers um einen Trace-Output

- Was ist das Ziel?
  - Jeden Auswertungsschritt und dessen Ergebnis ausgeben
- Alle Zeilen sammeln und als String zusammenbasteln

- Datenstruktur des Output-Traces

```
newtype Out a = MkOut(Output,a)
type Output = String
```

- zusätzliche Funktion nötig

```
line :: Term -> Int -> Output
line t x =
   "Term: " ++ show t ++ " ergibt: " ++ show x ++ "\n"
```

- der modifizierte Evaluierer

```
evalOut :: Term -> Out Int
evalOut (Con x)   = MkOut(line (Con x) x,x)
evalOut (Div t u) = MkOut(ox ++ oy ++ line (Div t u) z,z)
   where MkOut (ox,x) = evalOut t
         MkOut (oy,y) = evalOut u
         z = x `div` y
```

Erweiterung des Evaluierers um einen State

- Was ist das Ziel?
  - Anzahl der Divisionen zählen.
- Erste Idee: Inkrementieren einer globalen Variablen
- Nicht möglich: State als Parameter durchschleifen

- Datenstruktur des States

```
newtype St a = MkSt(State -> (a,State))
type State = Int
```

- zusätzliche Funktion nötig

```
apply :: St a -> State -> (a,State)
apply (MkSt f) s = f s
```

- der modifizierte Evaluierer

```
evalSt :: Term -> St Int
evalSt (Con x)   = MkSt f
   where f s = (x,s)
evalSt (Div t u) = MkSt f
   where
   f s = (x `div` y, s'' +1)
      where
      (x,s')  = apply (evalSt t) s
      (y,s'') = apply (evalSt u) s'
```

- Implementation der show-funktion

```
instance Show a => Show (St a) where
    show f = "value: " ++ show x ++ "\ncount: " ++ show s
     where (x,s) = apply f 0
```

Probleme

- Die Struktur des einfachen Evaluierers wird durch hinzufügen von neuen “Modifikationen” zerstört.
- Struktur wird unübersichtlich, schwerer zu lesen
- Lösung: Evaluierer mit Hilfe von Monaden entwickeln

Evaluierer mit Monad

- Datenstruktur ist die selbe

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

- Der Basis-Evaluierer sieht wie folgt aus:

```
eval :: Monad m => Term -> m Int
eval (Con x)   = return x
eval (Div t u) = do x <- eval t
                    y <- eval u
                    return (x `div` y)
```

- Struktur komplexer, jedoch flexibler

Evaluierer mit ID-Monad

- Evaluierer, der nur das Ergebnis berechnet
- Neuen Typ deklarieren

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

- Instanz von dem Monad

```
instance Monad Id where
   return x       = MkId x
   (MkId x) >>= q = q x
```

- Verhalten wie eval

```
evalId :: Term -> Id Int
evalId = eval
```

- Anzeigen des Ergebnisses

```
instance Show a => Show (Id a) where
    show (MkId x) = "Ergebnis: " ++ show x
```

Evaluierer mit Exception

- Datenstruktur ist identisch geblieben:

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

- Instanz von dem Monad

```
instance Monad Exc where
    return x          = Return x
    (Raise e) >>= q   = Raise e
    (Return x) >>= q = q x
```

- Spezifische Operation für den Monad zum Auslösen einer Exception

```
raise :: Exception -> Exc a
raise e = Raise e
```

- Kleinere Änderung an dem Basis-Evaluierer:

```
evalExc :: Term -> Exc Int
evalExc (Con x) = return x
evalExc (Div t u) = do x <- evalExc t
                       y <- evalExc u
                       if y == 0
                          then raise "division by zero"
                          else return (x `div` y)
```

Evaluierer mit Trace-Output

- Datenstruktur ist identisch geblieben:

```
newtype Out a = MkOut(Output,a)
type Output = String
```

- Instanz von dem Monad

```
instance Monad Out where
    return x = MkOut("",x)
    p >>= q = MkOut (ox ++ oy,y)
              where MkOut (ox,x) = p
                    MkOut (oy,y) = q x
```

- Spezifische Operation für den Monad zum Generieren des Outputs

```
out :: Output -> Out()
out ox = MkOut (ox,())
```

- Kleinere Änderung an dem Basis-Evaluierer:

```
evalOut :: Term -> Out Int
evalOut (Con x) = do out(line (Con x) x)
                     return x
evalOut (Div t u) = do x <- evalOut t
                       y <- evalOut u
                       out(line (Div t u) (x `div` y))
                       return (x `div` y)
```

Evaluierer mit States

- Datenstruktur ist identisch geblieben:

```
newtype St a = MkSt(State -> (a,State))
type State = Int
```

- Instanz von dem Monad

```
instance Monad St where
    return x = MkSt f where f s = (x,s)
    p >>= q  = MkSt f
               where
               f s = apply (q x) s'
                     where (x,s') = apply p s
```

- Spezifische Operation für den Monad zum Inkrementieren des States

```
tick :: St()
tick = MkSt f where f s = ((),s+1)
```

- Kleinere Änderung an dem Basis-Evaluierer:

```
evalSt :: Term -> St Int
evalSt (Con x) = return x
evalSt (Div t u) = do x <- evalSt t
                      y <- evalSt u
                      tick
                      return (x `div` y)
```

Was haben wir gelernt?

- Monaden dienen zum Sequentialisieren und Festlegen der Auswertereihenfolge
- Monaden dienen zur Abstraktion
  - Die Basisstruktur bleibt erhalten
  - Änderungen nur an wenigen Stellen in der Basisstruktur
  - Keine Umstrukturierung in der Basisstruktur nötig
  - Die Hauptänderungen nur in der Monade
  - Der Zugriff auf “globale” Variablen nur innerhalb des Monads

Ende

Vielen Dank für Eure Aufmerksamkeit.
Fragen, Anmerkungen, Kritik...?