Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS) Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

January 26, 2011

Today's Topics

- An 'Imperative' Evaluator
- Monads
- A Monadic Evaluator

An ‘Imperative’ Evaluator

The Basic Evaluator

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

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

The Basic Evaluator

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

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

Example Terms

```haskell
answer, failure :: Term
answer    =  Div (Div (Con 1972) (Con 2)) (Con 23)
failure   =  Div (Con 1) (Con 0)
```

```
> eval answer
42
> eval failure
*** Exception: divide by zero
```

Extending a Purely Functional Evaluator

- error handling - modify each recursive call to check for and handle errors

Extending a Purely Functional Evaluator

- error handling - modify each recursive call to check for and handle errors
- operation count - modify each recursive call to pass around current count

Extending a Purely Functional Evaluator

- error handling - modify each recursive call to check for and handle errors
- operation count - modify each recursive call to pass around current count
- execution trace - modify each recursive call to pass around the trace

Extending a Purely Functional Evaluator

- error handling - modify each recursive call to check for and handle errors
- operation count - modify each recursive call to pass around current count
- execution trace - modify each recursive call to pass around the trace
- in impure languages we could use: exceptions, global variables, output (not nice for mathematical reasoning, but easy to integrate)

Variation One - Exception Handling

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

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 `div` b)
```

Variation Two - State

```haskell
type M a    = State -> (a, State)
type State = Int

eval :: Term -> M Int
eval (Con a) x    = (a, x)
eval (Div t u) x = let (a, y) = eval t x in
                   let (b, z) = eval u y in
                   (a `div` b, z+1)
```

Variation Three - Tracing

```haskell
type M a     = (Output, a)
type Output = String

eval :: Term -> M Int
eval (Con a)    = (line (Con a) a, a)
eval (Div t u) =
  let (x, a) = eval t in
  let (y, b) = eval u in
  (x ++ y ++ line (Div t u) (a `div` b), a `div` b)

line :: Term -> Int -> Output
line t a =
  "eval(" ++ show t ++ ") <= " ++ show a ++ "\n"
```

Monads

Common Structure of Variations

- type `M a` of effects/computations

Common Structure of Variations

- type `M a` of effects/computations
- original evaluator of type `Term -> Int`

Common Structure of Variations

- type `M a` of effects/computations
- original evaluator of type `Term -> Int`
- all variations have type `Term -> M Int`

Common Structure of Variations

- type `M a` of effects/computations
- original evaluator of type `Term -> Int`
- all variations have type `Term -> M Int`
- accept argument of type `Term` and return result of type `Int` with possible additional effect captured by `M`

Common Structure of Variations

- type `M a` of effects/computations
- original evaluator of type `Term -> Int`
- all variations have type `Term -> M Int`
- accept argument of type `Term` and return result of type `Int` with possible additional effect captured by `M`
- which operations are required on `M`?

Common Structure of Variations

- type `M a` of effects/computations
- original evaluator of type `Term -> Int`
- all variations have type `Term -> M Int`
- accept argument of type `Term` and return result of type `Int` with possible additional effect captured by `M`
- which operations are required on `M`?
- turn arbitrary value into computation returning that value

```
return :: a -> M a
```

Common Structure of Variations

- type `M a` of effects/computations
- original evaluator of type `Term -> Int`
- all variations have type `Term -> M Int`
- accept argument of type `Term` and return result of type `Int` with possible additional effect captured by `M`
- which operations are required on `M`?
- turn arbitrary value into computation returning that value

```
return :: a -> M a
```

- apply function of type `a -> M b` to computation of type `M a`

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

Common Structure of Variations

- type `M a` of effects/computations
- original evaluator of type `Term -> Int`
- all variations have type `Term -> M Int`
- accept argument of type `Term` and return result of type `Int` with possible additional effect captured by `M`
- which operations are required on `M`?
- turn arbitrary value into computation returning that value

```
return :: a -> M a
```

- apply function of type `a -> M b` to computation of type `M a`

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

- `M` together with `return` and `(>>=)` (‘*bind*’) form a monad

Rewrite `eval` in Terms of Monad Abstractions

```haskell
eval :: Term -> M Int
eval (Con a)    = return a
eval (Div t u) =
  eval t >>= \a -> eval u >>= \b -> return (a `div` b)
```

Rewrite `eval` in Terms of Monad Abstractions

```
eval :: Term -> M Int
eval (Con a)    = return a
eval (Div t u) =
  eval t >>= \a -> eval u >>= \b -> return (a `div` b)
```

Recall

$$
\begin{aligned}
\texttt{do}\ \{\texttt{let}\ x \texttt{ = } e\,;M\} &= \texttt{let}\ x \texttt{ = } e\ \texttt{in}\ \texttt{do}\ \{M\} \\
\texttt{do}\ \{x\ \texttt{<-}\ m\,;M\} &= m\ \texttt{>>=}\ (\texttt{\textbackslash}x\ \texttt{->}\ \texttt{do}\ \{M\}) \\
\texttt{do}\ \{m\,;M\} &= m\ \texttt{>>=}\ (\texttt{\textbackslash\_}\ \texttt{->}\ \texttt{do}\ \{M\}) \\
\texttt{do}\ \{M\} &= M
\end{aligned}
$$

Rewrite eval in Terms of Monad Abstractions

```
eval :: Term -> M Int
eval (Con a)   = return a
eval (Div t u) =
  eval t >>= \a -> eval u >>= \b -> return (a `div` b)
```

Recall

$$
\begin{aligned}
\texttt{do}\ \{\texttt{let}\ x = e; M\} &= \texttt{let}\ x = e\ \texttt{in}\ \texttt{do}\ \{M\} \\
\texttt{do}\ \{x\ \texttt{<-}\ m; M\} &= m\ \texttt{>>=}\ (\backslash x\ \texttt{->}\ \texttt{do}\ \{M\}) \\
\texttt{do}\ \{m; M\} &= m\ \texttt{>>=}\ (\backslash\_\ \texttt{->}\ \texttt{do}\ \{M\}) \\
\texttt{do}\ \{M\} &= M
\end{aligned}
$$

Syntactic Sugar

```
eval (Div t u) = do
  a <- eval t
  b <- eval u
  return (a `div` b)
```

Monad Laws

1. left identity

$$\texttt{return}\ a \mathbin{\texttt{>>=}} f \quad = \quad f\ a$$

2. right identity

$$m \mathbin{\texttt{>>=}} \texttt{return} \quad = \quad m$$

3. associativity

$$(m \mathbin{\texttt{>>=}} f) \mathbin{\texttt{>>=}} g \quad = \quad m \mathbin{\texttt{>>=}} (\backslash x \mathbin{\texttt{->}} f\ x \mathbin{\texttt{>>=}} g)$$

A Monadic Evaluator

The Basic Evaluator, Revisited - The Identity Monad

```
type M a = a

return :: a -> M a
return x = x

(>>=) :: M a -> (a -> M b) -> M b
x >>= f = f x
```

Variation One, Revisited - The Exception Monad

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

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

(>>=) :: M a -> (a -> M b) -> M b
m >>= f = case m of Raise e  -> Raise e
                    Return x -> f x

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

eval (Div t u) = do
  a <- eval t
  b <- eval u
  if b == 0 then raise "divide by zero"
            else return (a `div` b)
```

Variation Two, Revisited - The State Monad

```
type M a     = State -> (a, State)
type State = Int

return :: a -> M a
return a = \x -> (a, x)

(>>=) :: M a -> (a -> M b) -> M b
m >>= f = \x -> let (a, y) = m x   in
                let (b, z) = f a y in
                (b, z)

tick :: M ()
tick = \x -> ((), x+1)

eval (Div t u) = do {
  a <- eval t; b <- eval u;
  tick; return (a `div` b)
}
```

Variation Three, Revisited - The Writer Monad

```haskell
type M a      = (Output, a)
type Output = String

return :: a -> M a
return a = ("", a)

(>>=) :: M a -> (a -> M b) -> M b
m >>= f = let (x, a) = m   in
          let (y, b) = f a in
          (x ++ y, b)

out :: Output -> M ()
out x = (x, ())

eval (Con a)   = do { out (line (Con a) a); return a }
eval (Div t u) = do {
  a <- eval t; b <- eval u;
  out(line(Div t u) (a `div` b)); return(a `div` b) }
```

Bibliography

Philip Wadler.
Monads for functional programming.
In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.