Functional Programming

Interpreters and Monads

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

A simple expression language

Definition

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

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

A simple interpreter

Evaluation

```
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
```

Extending the interpreter

Possible extensions

- Error handling
- Counting evaluation steps
- Variables, state
- Output

. . . but without changing the structure of the interpreter!

Interpreter with error handling

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)
```

Monads to the rescue!

The type class Monad

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

Here, `m` is a variable that can stand for `IO`, `Gen`, and other **type constructors**.

Monadic evaluator

The identity monad

```
newtype Id a = Id a

instance Monad Id where
    return x = Id x
    x >>= f = let Id y = x in f y
```

Monadic 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)
```

Monadic interpreter with error handling

Exeception

```
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 with tracing

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"
```

A monad for tracing

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)
```

Monadic interpreter with tracing

Evaluation

```
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 with reduction count

Count

```haskell
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)
```

A monad for counting

The state monad

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))
```

Monadic interpreter with reduction count

Implementation

Evaluation

```
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)
```

Typical monads

Already used

- Identity monad
- Exception monad
- State monad
- Writer monad

Not every type constructor can be a monad

Monad laws

return is a left unit

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

return is a right unit

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

bind is associative

```
m1 >>= \x -> (m2 >>= f) == (m1 >>= \x -> m2) >>= f
```

The Maybe monad

More useful than you think

- Computation that may or may not return a result
- Database queries, dictionary operations, ...

The Maybe monad

More useful than you think

- Computation that may or may not return a result
- Database queries, dictionary operations, ...

Definition (predefined)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return x = Just x

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

The List monad

Useful for

- Handling multiple results
- Backtracking

The List monad

Useful for

- Handling multiple results
- Backtracking

Definition (predefined)

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

where

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

The IO Monad

Required for

- any kind of I/O
- side effecting operation
- implementation is machine dependent

Challenges

- what if there are multiple effects?

Challenges

- what if there are multiple effects?
- need to combine monads

Challenges

- what if there are multiple effects?
- need to combine monads
- sequence matters (e.g., exception and state)

Challenges

- what if there are multiple effects?
- need to combine monads
- sequence matters (e.g., exception and state)
- some monads do not combine at all

Challenges

- what if there are multiple effects?
- need to combine monads
- sequence matters (e.g., exception and state)
- some monads do not combine at all
- BUT we can go for something weaker