Functional Programming

Interpreters and Monads

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2024/25

A simple expression language

Definition

```haskell
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!

Feature: Error handling

- A computation either returns a value or it fails on the way
- Could be expressed with the **Maybe** type
  - Returning the value v is **Just** v
  - Signalling an error is **Nothing**
- But it would be better to have an error message

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

- Equivalently we might use the predefined type **Either** a b

```
type Exception' a = Either String a
```

Interpreter with error handling

Exception

```
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)
```

Feature: Tracing

- We would like to produce a trace of the computation
- Each step of the computation should be documented
- Such a computation returns the computed value **and** the trace
- Could be expressed with the following type

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

Interpreter with tracing

Trace

```
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"
```

Feature: Reduction count

- We want to keep track of the number of reductions during evaluation of an expressions
- To this end, a counter value must be passed as a parameter to the `eval` function, which must return the updated counter
- A type like the following would do

```
newtype Count a = Count { exCount :: Int -> (a, Int) }
```

Interpreter with reduction count

Count

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

Monads

- There is an abstract concept of computation behind these examples: Monads
- Originates in category theory
- Mathematicians did not even bother to give it a proper name, they called it “standard construction”
- Discovered for computations by Eugenio Moggi
- Popularized for programming by Phil Wadler
- In Haskell: the **Monad** type class

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**.

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**.

Purpose of the operations

- Bind (>>=) is for **composing** computations
  In **IO** composing means sequencing
- **return** is for injecting values into computations
  A trivial computation that does nothing but return the value
- Monads live for their other features!

Monadic evaluator

Monadic interpreter

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

Monadic evaluator

Monadic interpreter

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

The identity monad

```haskell
newtype Id a = Id a

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

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 -- the extra operation
```

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)
```

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
```

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

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

return is a right unit

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

bind is associative

```haskell
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