

Functional Programming

Interpreters and Monads

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

A simple expression language

Definition

```
1 data Term = Con Integer
2           | Bin Term Op Term
3           deriving (Eq, Show)
4
5 data Op = Add | Sub | Mul | Div
6         deriving (Eq, Show)
```

A simple interpreter

Evaluation

```
1 eval :: Term -> Integer
2 eval (Con n) = n
3 eval (Bin t op u) = sys op (eval t) (eval u)
4
5 sys Add = (+)
6 sys Sub = (-)
7 sys Mul = (*)
8 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

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

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

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

Interpreter with error handling

Exception

```
1 eval :: Term -> Exception Integer
2 eval (Con n) = Return n
3 eval (Bin t op u) = case eval t of
4     Raise s -> Raise s
5     Return v -> case eval u of
6         Raise s -> Raise s
7         Return w ->
8             if (op == Div && w == 0)
9             then
10                 Raise "div by zero"
11             else
12                 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

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

Interpreter with tracing

Trace

```
1 eval :: Term -> Trace Integer
2 eval e@(Con n) = Trace (n, trace e n)
3 eval e@(Bin t op u) =
4     let Trace (v, x) = eval t in
5     let Trace (w, y) = eval u in
6     let r = sys op v w in
7     Trace (r, x ++ y ++ trace e r)
8
9 trace t n = "eval (" ++ show t ++ ") = "
10            ++ 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

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

Interpreter with reduction count

Count

```
1 eval :: Term -> Count Integer
2 eval (Con n) = Count $ \i -> (n, i)
3 eval (Bin t op u) = Count $ \i -> let (v, j) = exCount (eval t) i in
4                                     let (w, k) = exCount (eval u) j in
5                                     (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

```
1 class Applicative m => Monad m where  
2   (>>=) :: m a -> (a -> m b) -> m b  
3   return :: a -> m a  
4   fail :: String -> m a
```

Here, $m :: * \rightarrow *$ is a variable that can stand for IO, Gen, and other type constructors.

Monads to the rescue!

The type class Monad

```
1 class Applicative m => Monad m where
2   (>>=) :: m a -> (a -> m b) -> m b
3   return :: a -> m a
4   fail :: String -> m a
```

Here, $m :: * \rightarrow *$ 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

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

Monadic evaluator

Monadic interpreter

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

The identity monad

```
1 newtype Id a = Id a
2
3 instance Monad Id where
4     return x = Id x
5     x >>= f = let Id y = x in f y
```

Monadic interpreter with error handling

Exception

```
1 instance Monad Exception where
2   return a = Return a
3   m >>= f = case m of
4               Raise s -> Raise s
5               Return v -> f v
6   fail s = Raise s -- the extra operation
```

Interpreter

```
1 eval :: Term -> Exception Integer
2 eval (Con n) = return n
3 eval (Bin t op u) = eval t >>= \v ->
4                       eval u >>= \w ->
5                           if (op == Div && w == 0)
6                               then fail "div by zero"
7                               else return (sys op v w)
```


A monad for tracing

Trace

```
1 instance Monad Trace where
2   return a = (a, "")
3   m >>= f = let Trace (a, x) = m in
4             let Trace (b, y) = f a in
5             Trace (b, x ++ y)
6
7 output :: String -> Trace ()
8 output s = Trace ((), s)
```

Monadic interpreter with tracing

Evaluation

```
1 eval :: Term -> Trace Integer
2 eval e@(Con n) = output (trace e n) >>
3     return n
4 eval e@(Bin t op u) = eval t >>= \v ->
5     eval u >>= \w ->
6     let r = sys op v w in
7     output (trace e r) >>
8     return r
```

A monad for counting

The state monad

State

```
1 data ST s a = ST (s -> (a, s))
2 exST (ST sas) = sas
3
4 instance Monad (ST s) where
5     return a = ST (\s -> (a, s))
6     m >>= f = ST (\s -> let (a, s') = exST m s in
7                           exST (f a) s')
8
9 type Count a = ST Int a
10
11 incr :: Count ()
12 incr = ST (\i -> ((), i + 1))
```

Monadic interpreter with reduction count

Implementation

Evaluation

```
1 eval :: Term -> Count Integer
2 eval (Con n) = return n
3 eval (Bin t op u) = eval t >>= \v ->
4                       eval u >>= \w ->
5                       incr >>
6                       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

1 **return** x >>= f == f x

return is a right unit

1 m >>= **return** == m

bind is associative

1 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)

```
1 data Maybe a = Nothing | Just a
2
3 instance Monad Maybe where
4     return x = Just x
5
6     Nothing >>= f = Nothing
7     (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)

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

where

```
1 concatMap :: (a -> [b]) -> [a] -> [b]
2 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