

Monads : why should you care?

harold.carr@gmail.com / @haroldcarr / haroldcarr.com

<http://www.degoesconsulting.com/lambdaconf/> - 2014-04-19

Outline

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

Monads : why should you care?

Most monad explanations

- use analogies like "container" or "context"
- show how monads are implemented
- how they relate to category theory

This presentation shows

- the usefulness of monads—what they can do for you
- (not how they work)

Goal

- provide a glimpse of benefits of monads
- so you can use them in your work
- motivate you to understand how they work on your own

Source of examples: Martin Grabmuller :

- <http://www.grabmuller.de/martin/www/pub/Transformers.en.html>

- WebLogic InfiniBand Communication at Oracle
- SOAP Web Services Technology at Oracle/Sun
- CORBA/IIOP, RMI-IIOP Technology at Sun
- Visual LISP Technology at Autodesk
- Distributed C++ at University of Utah/HP
- (Concurrent) Utah Scheme, Utah Common Lisp, Portable Standard Lisp at University of Utah/HP
- Logic Simulation at Patel Systems/Cirrus Logic

initial intuition

- a Monad is used to pipe the *output* of one function into another
- a monadic "pipe" has code that executes "behind the scenes"
 - rather than embedded in main lines of program

monads and side-effects

- a Monad has *nothing* to do with "real" side-effects
 - e.g., reading/writing a file
- a monadic type is often used to simulate side-effects in a purely functional way
 - aka "effectful"
- the IO monad does "real" side-effects

combining monads

- use monad "transformers" to combine two or more monads

- 1 abstract, me, what you will learn
- 2 **non-monadic evaluator**
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

```
1  type Name    = String          -- variable names
2
3  data Exp     = Lit Integer     -- expressions
4              | Var Name
5              | Plus Exp Exp
6              | Abs Name Exp
7              | App Exp Exp
8              deriving (Eq, Show)
9
10 data Value    = IntVal Integer  -- values
11              | FunVal Env Name Exp
12              deriving (Eq, Show)
13
14 type Env      = Map.Map Name Value -- var names to vals
```



```
1  eval0                                :: Env -> Exp -> Value
2
3
4  eval0 env (Lit i)                    = IntVal i
5
6  eval0 env (Var n)                    = fromJust (Map.lookup n env)
7
8  eval0 env (Plus e1 e2) = let IntVal i1 = eval0 env e1
9                               IntVal i2 = eval0 env e2
10                              in IntVal (i1 + i2)
11
12 eval0 env (Abs n e) = FunVal env n e
13
14 eval0 env (App e1 e2) = let v1 = eval0 env e1
15                          v2 = eval0 env e2
16                          in case v1 of
17                              FunVal env' n body ->
18                                  eval0 (Map.insert n v2 env') body
```

```
1  -- 12 + (\x -> x) (4 + 2)
2  exampleExp
3  => Plus (Lit 12)
4         (App (Abs "x" (Var "x"))
5              (Plus (Lit 4) (Lit 2)))
6
7  eval0 Map.empty exampleExp
8  => IntVal 18
9
10 eval0 Map.empty (Plus (Lit 2) (Abs "x" (Lit 1)))
11 => IntVal *** Exception: m.hs:59:31-55: Irrefutable pattern failed for p
12
13 eval0 Map.empty (Var "x")
14 => *** Exception: Maybe.fromJust: Nothing
```

"fixing" unbound variable handling using Either

```
1  eval0e                :: Env -> Exp -> Either String Value
2
3  eval0e env (Lit  i)    = Right $ IntVal i
4
5  eval0e env (Var  n)    = case Map.lookup n env of
6                          Nothing -> Left $ "unbound: " ++ n
7                          Just v   -> Right v
8
9  eval0e env (Plus e1 e2) = let Right (IntVal i1) = eval0e env e1
10                          Right (IntVal i2) = eval0e env e2
11                          in Right $ IntVal (i1 + i2)
12
13 eval0e env (Abs  n  e)  = Right $ FunVal env n e
14
15 eval0e env (App  e1 e2) = let Right v1 = eval0e env e1
16                          Right v2 = eval0e env e2
17                          in case v1 of
18                              FunVal env' n body ->
19                                  eval0e (Map.insert n v2 env')
20                                  body
```

```
1  (eval0e Map.empty (Var "x"))  
2  => (Left "unbound: x")
```

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator**
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

```
1  type Eval1 alpha  =  Identity alpha
2
3  runEval1           :: Eval1 alpha -> alpha
4  runEval1 ev        =  runIdentity ev
5
6  eval1              :: Env -> Exp -> Eval1 Value
```

```
1  eval1 env (Lit i)      = return $ IntVal i
2
3  eval1 env (Var n)      = return $ fromJust (Map.lookup n env)
4
5  eval1 env (Plus e1 e2) = do IntVal i1 <- eval1 env e1
6                             IntVal i2 <- eval1 env e2
7                             return $ IntVal (i1 + i2)
8
9  eval1 env (Abs n e)    = return $ FunVal env n e
10
11 eval1 env (App e1 e2) = do v1 <- eval1 env e1
12                             v2 <- eval1 env e2
13                             case v1 of
14                               FunVal env' n body ->
15                                 eval1 (Map.insert n v2 env') body
```

1	Lit : IntVal i	return \$ IntVal i
2		
3	Var : fromJust (Map.lookup n env)	return \$ fromJust (Map.lookup n env)
4		
5	Plus: let IntVal i1 = eval0 env e1	do IntVal i1 <- eval1 env e1
6	IntVal i2 = eval0 env e2	IntVal i2 <- eval1 env e2
7	in IntVal (i1 + i2)	return \$ IntVal (i1 + i2)
8		
9	Abs : FunVal env n e	return \$ FunVal env n e
10		
11	App : let v1 = eval0 env e1	do v1 <- eval1 env e1
12	v2 = eval0 env e2	v2 <- eval1 env e2
13	in case v1 of	case v1 of
14	FunVal env' n body ->	FunVal env' n body ->
15	eval0 (Map.insert n v2 env')	eval1 (Map.insert n v2 env')
16	body	body


```
1  eval1 env (Lit i)      = return $ IntVal i
2
3  eval1 env (Var n)      = return $ fromJust (Map.lookup n env)
4
5  eval1 env (Plus e1 e2) = eval1 env e1 >>= \dummy ->
6                          case dummy of
7                              IntVal i1 -> eval1 env e2 >>= \dummy ->
8                                  case dummy of
9                                      IntVal i2 -> return $ IntVal (i1 + i2)
10                                     _         -> fail "pattern match failure"
11                                     _         -> fail "pattern match failure"
12
13 eval1 env (Abs n e)    = return $ FunVal env n e
14
15 eval1 env (App e1 e2) = eval1 env e1 >>= \v1 ->
16                          eval1 env e2 >>= \v2 ->
17                          case v1 of
18                              FunVal env' n body ->
19                                  eval1 (Map.insert n v2 env') body
```

```
1  runEval1 (eval1 Map.empty exampleExp)
2  => IntVal 18
3
4  runEval1 (eval1 Map.empty (Var "x"))
5  => *** Exception: Maybe.fromJust: Nothing
```

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling**
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

```
1  type Eval2 alpha = ErrorT String Identity alpha
2
3  runEval2          :: Eval2 alpha -> Either String alpha
4  runEval2 ev       = runIdentity (runErrorT ev)
5
6  eval2a            :: Env -> Exp -> Eval2 Value
7
8
9  eval2a env (Var n) = case (Map.lookup n env) of
10                        Nothing -> fail $ "unbound: " ++ n
11                        Just v   -> return v
```

```
1  runEval2 (eval2a Map.empty exampleExp)
2  => Right (IntVal 18)
3
4  runEval2 (eval2a Map.empty (Var "no-way"))
5  => Left "unbound: no-way"
6
7  -- type error, but not apparent in error message
8  runEval2 (eval2a Map.empty (Plus (Lit 12) (Abs "x" (Var "x"))))
9  => Left "Pattern match failure in do expression at transformers.hs:138:3"
```

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors**
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

handle dynamic type errors : code

no change in types

```
1  eval2b env (Plus e1 e2) = do e1' <- eval2b env e1
2                                e2' <- eval2b env e2
3                                case (e1', e2') of
4                                  (IntVal i1, IntVal i2)
5                                    -> return $ IntVal (i1 + i2)
6                                  _ -> throwError "dyn type err: Plus"
7
8  eval2b env (App e1 e2)  = do v1 <- eval2b env e1
9                                v2 <- eval2b env e2
10                               case v1 of
11                                 FunVal env' n body
12                                   -> eval2b (Map.insert n v2 env') body
13                                 _ -> throwError "dyn type err: App"
```

```
1  runEval2 (eval2b Map.empty (Plus (Lit 12) (Abs "x" (Var "x"))))
2  => Left "dyn type err: Plus"
```


- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment**
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

Env only

- extended in App
- used in Var and Abs

```
1  type Eval3 alpha = ReaderT Env (ErrorT String Identity) alpha
2
3  runEval3          :: Env -> Eval3 alpha -> Either String alpha
4  runEval3 env ev   = runIdentity (runErrorT (runReaderT ev env))
5
6  eval3             :: Exp -> Eval3 Value
```

```
1  eval3 (Var n)      = do env <- ask
2                        case Map.lookup n env of
3                          Nothing -> throwError ("unbound: " ++ n)
4                          Just val -> return val
5
6  eval3 (Abs n e)    = do env <- ask
7                        return $ FunVal env n e
8
9  eval3 (App e1 e2) = do v1 <- eval3 e1
10                        v2 <- eval3 e2
11                        case v1 of
12                          FunVal env' n body
13                            -> local (const (Map.insert n v2 env'))
14                                (eval3 body)
15                          _ -> throwError "dyn type err: App"
```

```
1  runEval3 Map.empty (eval3 exampleExp)
2  => Right (IntVal 18)
```

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)**
- 8 add logging
- 9 add IO
- 10 final version
- 11 resources

```
1  type Eval4 alpha =
2      ReaderT Env (ErrorT String (StateT Integer Identity)) alpha
3
4  runEval4 :: Env
5             -> Integer
6             -> Eval4 alpha
7             -> (Either String alpha, Integer)
8  runEval4 env st ev =
9      runIdentity (runStateT (runErrorT (runReaderT ev env)) st)
10
11 eval4      :: Exp -> Eval4 Value
```

```
1  tick :: (Num s, MonadState s m) => m ()
2  tick = do st <- get
3           put (st + 1)
4
5
6  eval4 (Lit i)      = do tick
7                     return $ IntVal i
8
9  eval4 (Var n)      = do tick
10                      env <- ask
11                      ...
```

```
1  runEval4 Map.empty 0 (eval4 exampleExp)
2  => (Right (IntVal 18),8) -- 8 reduction steps
```


- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging**
- 9 add IO
- 10 final version
- 11 resources

add logging : types

```
1  type Eval5 alpha =
2      ReaderT Env
3          (ErrorT String (WriterT [String]
4                                  (StateT Integer Identity)))
5          alpha
6
7  runEval5 :: Env
8            -> Integer
9            -> Eval5 alpha
10           -> ((Either String alpha, [String]), Integer)
11
12 runEval5 env st ev =
13     runIdentity (runStateT (runWriterT (runErrorT
14                                         (runReaderT ev env)))
15                             st)
16
17 eval5      :: Exp -> Eval5 Value
```

```
1 eval5 (Var n)      = do tick
2                     tell [n] -- write name vars seen during eval
3                     env <- ask
4                     case Map.lookup n env of
5                       Nothing  -> throwError ("unbound: " ++ n)
6                       Just val  -> return val
```

```
1  runEval5 Map.empty 0 (eval5 exampleExp)
2  => ((Right (IntVal 18),["x"]),8)
```

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO**
- 10 final version
- 11 resources

```
1  type Eval6 alpha =
2      ReaderT Env
3          (ErrorT String (WriterT [String] (StateT Integer IO)))
4          alpha
5
6  runEval6 :: Env
7            -> Integer
8            -> Eval6 alpha
9            -> IO ((Either String alpha, [String]), Integer)
10 runEval6 env st ev =
11     runStateT (runWriterT (runErrorT (runReaderT ev env))) st
12
13 eval6      :: Exp -> Eval6 Value
```

```
1  eval6 (Lit i)      = do tick
2                      liftIO $ print i -- print each int
3                      return $ IntVal i
```

```
1  runEval6 Map.empty 0 (eval6 exampleExp)
2  12
3  4
4  2
5  => IO ((Right (IntVal 18),["x"]),8)
```


- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version**
- 11 resources

```

1  eval6 (Lit i)      = do tick          -- profiling
2                      liftIO $ print i -- print each int
3                      return $ IntVal i
4
5  eval6 (Var n)      = do tick
6                      tell [n]          -- log var
7                      env <- ask        -- consult env
8                      case Map.lookup n env of
9                        Nothing -> throwError ("unbound: " ++ n)
10                       Just val -> return val
11
12 eval6 (Plus e1 e2) = do tick
13                     e1' <- eval6 e1
14                     e2' <- eval6 e2
15                     case (e1', e2') of
16                       (IntVal i1, IntVal i2)
17                         -> return $ IntVal (i1 + i2)
18                       _ -> throwError "dyn type err: Plus"

```

```
1  eval6 (Abs n e)    = do tick
2                      env <- ask
3                      return $ FunVal env n e
4
5  eval6 (App e1 e2)  = do tick
6                      v1 <- eval6 e1
7                      v2 <- eval6 e2
8                      case v1 of
9                        FunVal env' n body
10                         -> local (const (Map.insert n v2 env'))
11                             (eval6 body)
12                         _ -> throwError "dyn type err: App"
```

- 1 abstract, me, what you will learn
- 2 non-monadic evaluator
- 3 equivalent monadic evaluator
- 4 add unbound variable error handling
- 5 handle dynamic type errors
- 6 hide the environment
- 7 add state (i.e., simulated mutability)
- 8 add logging
- 9 add IO
- 10 final version
- 11 **resources**

- interactive version of this presentation at FPComplete :
 - <https://www.fpcomplete.com/user/haroldcarr/example-of-why-to-use-monads-what-they-can-do>
- Maybe, Either, [], IO monads (and more to come) :
 - <http://haroldcarr.com/posts/2014-02-19-monad-series.html>
- Phillip Wadler : covers similar ground : in Scala : must see :
 - <http://yow.eventer.com/yow-2013-1080/the-first-monad-tutorial-by-phillip-wadler-1359>
- Dan Piponi's :
 - <http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html>
- illustrated :
 - http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
- Brent Yorgey :
 - <http://www.haskell.org/haskellwiki/Typeclassopedia>
- deep dive with Mike Vanier :
 - <http://mvanier.livejournal.com/3917.html>