(Simile Free) Monad Recipes

Aditya Siram (@deech)

July 7, 2013

Outline

Brief IO Example

- This small function writes a text file, uppercases its contents & prints them..

```
import Data.Char
main :: IO ()
main = do
    writeFile "test.txt" "a,b,c,d,e"
    x <- readFile "test.txt"
    let up_cased = map toUpper x
    y <- return up_cased
    print y
=> "A,B,C,D,E"
```

Brief IO Example

- With types added (don't forget the pragma) ...

```
{-# LANGUAGE ScopedTypeVariables #-}
main :: IO ()
main = do
  writeFile "test.txt" "a,b,c,d,e" :: IO ()
  x :: String <- readFile "test.txt" :: IO String
  let upCased :: String = map toUpper x
  y :: String <- return upCased :: IO String
  print y :: IO ()
```

Brief IO Example

```
main = do
  writeFile "test.txt" "a,b,c,d,e" :: IO ()
  ...
```

Brief IO Example

```
main = do
  ...
  x :: String <- readFile "test.txt" :: IO String
  let upCased :: String = map toUpper x
  ...
```

Brief IO Example

```
main = do
  let upCased = ...
  y :: String <- return upCased :: IO String
  ...
```

Brief IO Example

```
main :: IO ()
main = do
  y <- ...
  print y :: IO ()
```

Brief IO Example

- Querying a Sqlite database

```
get_users :: IO [(String,String)]
get_users = do
  rows :: [[SqlValue]] <- dbQuery
                            "select * from users"
                            []
  let marshalled :: [(String,String)] =
          map (\(user:pass:[]) ->
                 (fromSql user, fromSql pass))
              rows
  return marshalled
  where
    dbQuery sql values = ...
```

Brief IO Example

- Querying a Sqlite database

```
get_users = do
  rows :: [[SqlValue]] <- dbQuery
                              "select * from users"
                              []
  ...
```

Brief IO Example

- Querying a Sqlite database

```
get_users = do
  rows <- ...
  let marshalled :: [(String,String)] =
          map (\(user:pass:[]) ->
                   (fromSql user, fromSql pass))
              rows
  ...
```

Brief IO Example

- Querying a Sqlite database

```
get_users :: IO [(String,String)]
get_users = do
  ...
  let marshalled = ...
  return marshalled
  where ...
```

Brief IO Example

- The implementation of dbQuery isn't important, but here it is . . .

```
dbQuery :: String -> [SqlValue] -> IO [[SqlValue]]
dbQuery sql values =
      bracket dbConnect disconnect
        (\conn -> quickQuery' conn sql values)
dbConnect :: IO Connection
dbConnect = connectSqlite3 "test.sqlite"
```

Reader (1/2)

- Reader = Read-only State + Result
- 'runReader' :: Reader Monad -> Read-Only State -> Result
- 'ask' extracts the state from the monad for inspection.

Reader (2/2)

- Authenticating users

```
simple_auth :: (String,String) ->
               Reader [(String,String)] Bool
simple_auth (user,pass) = do
  users :: [(String,String)] <- ask
  case (lookup user users) of
    Nothing -> return False
    Just p -> return (p == pass)

main =
    let my_auth = ("deech","deechpassword") in
    do users :: [(String,String)] <- get_users
       print (runReader (simple_auth my_auth) users)
=> True
```

Reader (2/2)

- Authenticating users

```
simple_auth :: (String,String) ->
               Reader [(String,String)] Bool
simple_auth (user,pass) = do ...
main = ...
```

Reader (2/2)

- Authenticating users

```
simple_auth :: (String,String) ->
                  Reader [(String,String)] Bool
simple_auth (user,pass) = do ...
main =
    let my_auth = ("deech","deechpassword") in
    do users :: [(String,String)] <- get_users
       print (runReader (simple_auth my_auth) users)
```

Reader (2/2)

- Authenticating users

```
simple_auth :: (String,String) ->
                  Reader [(String,String)] Bool
simple_auth (user,pass) = do
  users :: [(String,String)] <- ask
  ...
main = ...
```

Reader (2/2)

- Authenticating users

```
simple_auth :: (String,String) ->
               Reader [(String,String)] Bool
simple_auth (user,pass) = do
  users <- ...
  case (lookup user users) of
    Nothing -> return False
    Just p -> return (p == pass)
main = ...
```

Writer

- Writer = Append-Only State + Result
- ‘runWriter’ :: Writer Monad -> (Result, Accumulated State)
- State is accumulated using ‘tell’

Writer

- Validating input

```
validate :: String -> Writer [String] ()
validate input =
    let hasNumbers = (>= 2) . length . filter isDigit
        hasUppers  = (>= 1) . length . filter isUpper
        noSpaces   = null . filter (== ' ')
        check f msg = if (not (f input))
                      then tell [msg]
                      else return ()
    in do check hasNumbers "Needs 2+ numbers"
          check hasUppers  "Needs 1+ capitals"
          check noSpaces   "Has spaces"
```

Writer

- Validating input

```
validate :: String -> Writer [String] ()
validate input = ...
```

Writer

- Validating input

```
validate :: String -> Writer [String] ()
validate input =
    let hasNumbers = (>= 2) . length . filter isDigit
        hasUppers  = (>= 1) . length . filter isUpper
        noSpaces   = null . filter (== ' ')
        ...
```

Writer

- Validating input

```
validate :: String -> Writer [String] ()
validate input =
    let hasNumbers = ...
        hasUppers  = ...
        noSpaces   = ...
        check f msg = if (not (f input))
                      then tell [msg]
                      else return ()
    in do ...
```

Writer

- Validating input

```
validate :: String -> Writer [String] ()
validate input =
    let hasNumbers = ...
        hasUppers  = ...
        noSpaces   = ...
        check f msg = ...
    in do check hasNumbers "Needs 2+ numbers"
          check hasUppers  "Needs 1+ capitals"
          check noSpaces   "Has spaces"
```

Writer

- Running

```
main = do
  let ((),errs) = runWriter (validate "abcde1")
      valid     = null errs
  if (not valid) then print errs else print "Valid!"
=> ["Needs 2+ numbers","Needs 1+ capitals"]
```

State

- State Monad = Mutable State + Result
- ‘get’, ‘put’ do what they sound like
- ‘runState’ :: State Monad -> Initial State -> (Result, New State)
- Initial State is **required**.

State

- Finding the minimum imperatively. Buggy!

```
minimum :: [Int] -> State Int ()
minimum [] = error "Empty List."
minimum xs =
    forM_ xs (\curr -> do
                old_min <- get
                if (curr < old_min)
                then put curr
                else return ())
main = let numbers = [3,2,1] in
       print (runState (Main.minimum numbers) (-1))
  => -1
```

State

- Finding the minimum imperatively. Buggy!

```
minimum :: [Int] -> State Int ()
...
main = ...
```

State

- Finding the minimum imperatively. Buggy!

```
minimum :: [Int] -> State Int ()
minimum [] = error "Empty List."
...
main = ...
```

State

- Finding the minimum imperatively. Buggy!

```
minimum xs =
    forM_ xs (\curr -> do
                old_min <- get
                ...)
```

State

- Finding the minimum imperatively. Buggy!

```
minimum xs =
    forM_ xs (\curr -> do
                old_min <- ...
                if (curr < old_min)
                then put curr
                else return ())
```

State

- Finding the minimum imperatively. Buggy!

```
minimum :: [Int] -> State Int ()
minimum [] = ...
minimum xs = ...
main = let numbers = [3,2,1] in
       print (runState (Main.minimum numbers) (-1))
  => -1
```

State

- ‘trace‘ and ‘printf‘ are your friends

```
import Debug.Trace
import Text.Printf
-- trace :: String -> a -> a
println msg = trace msg (return ())
printf_test = printf "Welcome to %s %d" "LambdaJam" 2013
   => "Welcome to LambdaJam 2013"
```

State

```
minimum xs = ...
    forM_ xs (\curr -> do
                old_min <- get
                println (printf "old_min: %d curr: %d"
                                 old_min curr)
                ...)
  => ((), old_min: -1 curr: 3
          old_min: -1 curr: 2
          old_min: -1 curr: 1
          -1)
```

State

- Fixed!

```
-- main = let numbers = [3,2,1] in
--        print (runState (Main.minimum numbers) (-1))
main = let (n:ns) = [3,2,1] in
       print (runState (Main.minimum ns) n)
```

Transformers

- Use all at once.
- The Good: Combining monads is easy.
- The Bad: Type sigs. and runners are more complicated.
- The Sorta Good: It's pretty mechanical

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts     msg = liftIO (putStrLn msg)
      wait_for msg = do {puts msg; liftIO getLine}
      log_failed   = tell ["Failed login attempt"]
      set_user u   = do {puts "Welcome!"; put u}
  in do users    <- ask
        user     <- wait_for "Username:"
        password <- wait_for "Password:"
        case (lookup user users) of
          Nothing -> do puts "Invalid Login!"
                        log_failed
          Just p  -> if (p == password)
                     then set_user user
                     else log_failed
```

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts      msg = liftIO (putStrLn msg)
      wait_for msg = ...
      log_failed   = ...
      set_user u   = ...
  in do ...
```

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts     msg = liftIO (putStrLn msg)
      wait_for msg = do {puts msg; liftIO getLine}
      log_failed   =
      set_user u   =
  in do ...
```

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts     msg = ...
      wait_for msg = ...
      log_failed   = tell ["Failed login attempt"]
      set_user u   = ...
  in do ...
```

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts     msg = liftIO (putStrLn msg)
      wait_for msg = ...
      log_failed   = ...
      set_user u   = do {puts "Welcome!"; put u}
  in do ...
```

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts     msg = ...
      wait_for msg = ...
      log_failed   = ...
      set_user u   = ...
  in do users    <- ask
        ...
```

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts     msg = ...
      wait_for msg = do {puts msg; liftIO getLine}
      log_failed   = ...
      set_user u   = ...
  in do users    <-  ...
        user     <- wait_for "Username:"
        password <- wait_for "Password:"
        ...
```

Transformers

- An interactive version of auth

```
interactive_auth =
  let puts     msg = liftIO (putStrLn msg)
      wait_for msg = ...
      log_failed   = tell ["Failed login attempt"]
      set_user u   = do {puts "Welcome!"; put u}
  in do users    <- ...
        user     <- ...
        password <- ...
        case (lookup user users) of
          Nothing -> do puts "Invalid Login!"
                        log_failed
          Just p  -> if (p == password)
                     then set_user user
                     else log_failed
```

Transformers

```
interactive_auth :: ReaderT [(String,String)]
                            (WriterT [String]
                                     (StateT String
                                             IO))
                            ()
```

- Transformer = Nested Monads
- Monad Transformer = MonadT + Monad Params + M + Result
- 'runMonadT' :: MonadT -> Monad Params -> M (Computation Result)

Transformers

```
interactive_auth :: ReaderT [(String,String)]
                        (WriterT [String]
                                 (StateT String
                                         IO))
                        ()
```

- Reader Transformer = ReaderT + Read-Only State + M + Result
- 'runReaderT' :: ReaderT Monad -> Read-Only State -> M Result

```
let writer :: WriterT [String] (StateT ...) () =
   runReaderT interactive_auth users
```

Transformers

```
interactive_auth :: ReaderT [(String,String)]
                            (WriterT [String]
                                     (StateT String
                                             IO))
                            ()
```

- WriterT Transformer = WriterT + Append-Only State + M + Result
- 'runWriterT' :: WriterT Monad -> M (Result, Accumulated State)

```
let writer = runReaderT interactive_auth users
let state :: (StateT String ...) ((), [String])
    = runWriterT writer
```

Transformers

```
interactive_auth :: ReaderT [(String,String)]
                            (WriterT [String]
                                     (StateT String
                                             IO))
                            ()
```

- State Transformer = StateT + Mutable State + M + Result
- 'runStateT' :: StateT Monad -> Mutable State -> M (Result, New State)

```
let writer = runReaderT interactive_auth users
let state  = runWriterT writer
let io :: IO (((), [String]), String) =
   runStateT state ""
```

Running

- Using ‘interactive_auth‘

```
interactive_auth_driver = do
    let my_auth = ("deech","deechpassword")
    users <- get_users
    let writer = runReaderT interactive_auth users
    let state  = runWriterT writer
    let io     = runStateT  state ""
    final <- io
    print final
```

Running

- Running with Control.Monad.RWS

```
-- runRWST :: RWST Monad ->
              Read-Only State ->
              Mutable State ->
              Lowest Monad
interactive_auth_driver' = do
    let my_auth = ("deech","deechpassword")
    users <- get_users
    final <- runRWST interactive_auth users ""
    print final
```

Running

- Sample session 1

```
Username:
deech
Password:
wrongpassword
(((),["Failed login attempt"]),"")
```

- Sample session 2

```
Username:
deech
Password:
deechpassword
Welcome!
(((),[]),"deech")
```

Transformers

- Multiple States, Readers, Writers?
- An 'interactive_auth' with an attempt counter

```
interactive_auth :: ReaderT [(String,String)]
                      (WriterT [String]
                          (StateT String
                             (StateT Int
                                 IO)))
                      ()
```

- Not recommended!

Transformers

```
interactive_auth :: ReaderT [(String,String)]
                      (WriterT [String]
                          (StateT String
                             (StateT Int
                                 IO)))
                      ()
```

- 'lift' "removes" a monadic layer
- Accessing the counter:

```
do ...
   counter <- lift -- ReaderT
              (lift -- WriterT
               (lift -- StateT String
                get))
   ...
```

Transformers

- Better off using a record:

```
data Auth_State = Auth_State {
                              counter :: Int,
                              current_user :: String
                            }
increment_attempt_counter = do
  auth_state <- get
  put auth_state{counter = (counter auth_state + 1)}
```

End

- Real World Uses
  - Yesod
  - Snap
  - Parsec
  - XMonad
  - Many more . . .
- Happy Haskelling!