Lecture 8 – Reader, Writer, and State Monad

- Reader monad maintains an environment that contains read-only values that can be read by the computation.

```haskell
newtype Reader r a = Reader { runReader :: r -> a }
```

- Writer monad maintains a log to record information during the computation.

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }
```

- State monad maintains a mutable state so that the computation can get or update the state.

```haskell
newtype State s a = State { runState :: s -> (a, s) }
```

Example with an interpreter

- Implement an interpreter for a nano language.

```haskell
-- abstract syntax
data Term = Const Integer
          | Var String
          | Plus Term Term
          | Times Term Term
          | LE Term Term
          | IF Term Term Term
          | App Term Term
          | Fn (String, Term)
          | Fun (String, String, Term) deriving (Show)
```

- Concrete syntax.

$$
\begin{array}{lcl}
t & ::= & n \mid x \mid t + t \mid t * t \mid t \leq t \\
 & \mid & \text{if } t_0 \text{ then } t_1 \text{ else } t_2 \\
 & \mid & t_1 \; t_2 \\
 & \mid & \text{fn } x \Rightarrow t \\
 & \mid & \text{fun } f \; x \; = \; t
\end{array}
$$

Example with an interpreter

- Implement an interpreter for a nano language.

```
-- abstract syntax
data Term = Const Integer
          | Var String
          | Plus Term Term
          | Times Term Term
          | LE Term Term
          | IF Term Term Term
          | App Term Term
          | Fn (String, Term)
          | Fun (String, String, Term) deriving (Show)
```

- Concrete syntax.

```
a = Plus (Const 10) (Const 20)}                  -- 10 + 20
f = Fn ("x", Plus(Var "x") (Const 5))} -- fn x => x + 5
t = App f a                  -- (fn x => x + 5) (10 + 20)
```

Example with an interpreter

- Implement an interpreter for a nano language.

```
-- abstract syntax
data Term = Const Integer
          | Var String
          | Plus Term Term
          | Times Term Term
          | LE Term Term
          | IF Term Term Term
          | App Term Term
          | Fn (String, Term)
          | Fun (String, String, Term) deriving (Show)
```

- Concrete syntax.

```
let fact = Fun ("fact", "x",
               IF (LE (Var "x") (Const 1))
                  (Const 1)
                  (Times (Var "x")
                         (App (Var "fact")
                              (Plus (Var "x") (Const (-1))))))
-- fun fact x = if x <= 1 then 1 else x * fact (x - 1)
```

Example with an interpreter

- Implement an interpreter for a nano language.

```
-- abstract syntax
data Term = Const Integer
          | Var String
          | Plus Term Term
          | Times Term Term
          | LE Term Term
          | IF Term Term Term
          | App Term Term
          | Fn (String, Term)
          | Fun (String, String, Term) deriving (Show)
```

- Define a type for variable context and a type for results.

```
type Context = [(String, Val)]

data Val = IntVal Integer
         | BoolVal Bool
         | FVal (Maybe String, String, Term) Context
           deriving (Show)
```

Example with an interpreter

- Define a type for variable context and a type for results.

```
type Context = [(String, Val)]

data Val = IntVal Integer
         | BoolVal Bool
         | FVal (Maybe String, String, Term) Context
           deriving (Show)
```

- Evaluate simple terms

```
lookup :: String -> Context -> Maybe Val

lookup x [] = Nothing          -- lookup variable in context
lookup x ((y, v) : xs)
  | x == y    = Just v
  | otherwise = lookup x xs

eval :: Term -> Context -> Maybe Val

eval (Var x) ctx = lookup x ctx        -- evaluate variable
```

Example with an interpreter

- Define a type for variable context and a type for results.

```
type Context = [(String, Val)]

data Val = IntVal Integer
         | BoolVal Bool
         | FVal (Maybe String, String, Term) Context
             deriving (Show)
```

- Evaluate simple terms

```
eval :: Term -> Context -> Maybe Val

eval (Const x) _ = Just x     -- evaluate constant

eval (Plus t1 t2) ctx = do    -- evaluate plus
    v1 <- eval t1 ctx
    case v1 of                -- t1 has to be an int
      (IntVal c1) -> do
        v2 <- eval t2 ctx
        case v2 of            -- t2 has to be an int
          (IntVal c2) -> Just $ IntVal (c1+c2)
          _ -> Nothing
      _ -> Nothing
```

Example with an interpreter

- Recap Maybe Monad

```haskell
instance Monad Maybe where
   return a = Just a

   Just x  >>= f = f x
   Nothing >>= _ = Nothing

instance MonadFail Maybe where
   fail _ = Nothing -- called if <- fails to pattern match
```

- Evaluate simple terms

```haskell
eval :: Term -> Context -> Maybe Val

eval (Const x) _ = Just x      -- evaluate constant

eval (Plus t1 t2) ctx = do     -- evaluate plus
     v1 <- eval t1 ctx
     case v1 of                -- t1 has to be an int
       (IntVal c1) -> do
          v2 <- eval t2 ctx
          case v2 of           -- t2 has to be an int
             (IntVal c2) -> Just $ IntVal (c1+c2)
             _ -> Nothing
       _ -> Nothing
```

Example with an interpreter

- Recap Maybe Monad

```haskell
instance Monad Maybe where
   return a = Just a

   Just x   >>= f = f x
   Nothing >>= _ = Nothing

instance MonadFail Maybe where
   fail _ = Nothing -- called if <- fails to pattern match
```

- Simplify case analysis based on default 'fail' behavior

```haskell
eval :: Term -> Context -> Maybe Val

eval (Const x) _ = Just x     -- evaluate constant

eval (Plus t1 t2) ctx = do
   (IntVal c1) <- eval t1 ctx -- if fails, return Nothing
   (IntVal c2) <- eval t2 ctx -- if fails, return Nothing
   return $ IntVal (c1 + c2)  -- pattern matching succeeds
```

Example with an interpreter

- Evaluate simple terms

```
eval :: Term -> Context -> Maybe Val

eval (Times t1 t2) ctx = do      -- evaluate times
   (IntVal c1) <- eval t1 ctx
   (IntVal c2) <- eval t2 ctx
   return $ IntVal (c1 * c2)

eval (Plus t1 t2) ctx = do       -- evaluate plus
   (IntVal c1) <- eval t1 ctx
   (IntVal c2) <- eval t2 ctx
   return $ IntVal (c1 + c2)

eval (LE t1 t2) ctx = do         -- evaluate less/than/equal
   (IntVal c1) <- eval t1 ctx
   (IntVal c2) <- eval t2 ctx
   return $ BoolVal (c1 <= c2)

eval (IF t0 t1 t2) ctx = do      -- evaluate if/then/else
   (BoolVal b) <- eval t0 ctx
   if b then eval t1 ctx else eval t2 ctx
```

Example with an interpreter

- Evaluate function value and function call.

```
eval :: Term -> Context -> Maybe Val

eval (App t1 t2) ctx = do
    -- evaluate function value
    fun@(FVal (f, x, t0) ctx0) <- eval t1 ctx

    -- evaluate argument value
    arg <- eval t2 ctx

    -- check whether the function is named
    let ctx = case f of Just name -> [(name, fun)]
                        Nothing -> []

    -- evaluate function body with new context
    eval t0 $ ctx ++ (x, arg) : ctx0

-- anonymous function
eval (Fn (x, t))      ctx = return $ FVal (Nothing, x, t) ctx
-- named function
eval (Fun (f, x, t)) ctx = return $ FVal (Just f, x, t) ctx
```

Example with an interpreter

- Do some tests

```
eval :: Term -> Context -> Maybe Val

main :: IO ()
main = do
         let x = Plus (Const 10) (Const 20)
         let f = Fn ("x", Plus(Var "x") (Const 5))
         print $ eval (App f x) []    -- Just (IntVal 35)

         let fact = Fun ("fact", "x",
                         IF (LE (Var "x") (Const 1))
                            (Const 1)
                            (Times (Var "x")
                                   (App (Var "fact")
                                        (Plus (Var "x")
                                              (Const (-1))))))
         let f10 = App fact (Const 10)
         print $ eval f10 []          -- Just (IntVal 3628800)
```

Example with an interpreter

- Avoid passing context parameter with reader monad (hand-rolled version).

```haskell
newtype Result r a =  Result {runResult :: r -> Maybe a}

instance Functor (Result r) where
   fmap f (Result g) = Result (\ctx -> fmap f $ g ctx)

instance Applicative (Result r) where
   pure x = Result (\_ -> pure x)
   Result f <*> Result a = Result (\ctx -> f ctx <*> a ctx)

instance Monad (Result r) where
   Result f >>= k = Result $ \ctx -> do
                                 a <- f ctx
                                 (runResult $ k a) ctx

instance Fail.MonadFail (Result r) where
   fail s = Result $ \_ -> Fail.fail s
```

Example with an interpreter

- eval function no longer takes context explicitly.

```
newtype Result r a =  Result {runResult :: r -> Maybe a}

eval :: Term -> Result Context Val

eval (Const a) = return $ IntVal a

eval (Var s) = Result $ \ctx -> lookup s ctx
```

Example with an interpreter

- eval function no longer takes context explicitly.

```
newtype Result r a =  Result {runResult :: r -> Maybe a}

eval :: Term -> Result Context Val

eval (Times t1 t2) = do
   (IntVal c1) <- eval t1
   (IntVal c2) <- eval t2
   return $ IntVal (c1 * c2)

eval (Plus t1 t2) = do
   (IntVal c1) <- eval t1
   (IntVal c2) <- eval t2
   return $ IntVal (c1 + c2)

eval (LE t1 t2) = do
   (IntVal c1) <- eval t1
   (IntVal c2) <- eval t2
   return $ BoolVal (c1 <= c2)

eval (IF t0 t1 t2) = do
   (BoolVal b) <- eval t0
   if b then eval t1 else eval t2
```

Example with an interpreter

- eval function no longer takes context explicitly.

```haskell
newtype Result r a =  Result {runResult :: r -> Maybe a}

eval :: Term -> Result Context Val

eval (App t1 t2) = do
    fun@(FVal (f, x, t0) ctx0) <- eval t1 -- eval function

    arg <- eval t2                         -- eval argument

    -- check whether the function is named
    let ctx = case f of Just name -> [(name, fun)]
                        Nothing   -> []

    -- evaluate function body with new context
    Result $ \_ ->
             runResult (eval t0) $ ctx ++ (x, arg) : ctx0

eval (Fn (x,t)) = Result (\ctx ->
                              return $ FVal (Nothing, x, t) ctx)

eval (Fun (f,x,t)) = Result (\ctx ->
                              return $ FVal (Just f, x, t) ctx)
```

Example with an interpreter

- Do some tests

```
eval :: Term -> Context -> Maybe Val

main :: IO ()
main = do
         let x = Plus (Const 10) (Const 20)
         let f = Fn ("x", Plus(Var "x") (Const 5))
         print $ runResult (eval (App f x)) []
         -- Just (IntVal 35)

         let fact = Fun ("fact", "x",
                         IF (LE (Var "x") (Const 1))
                            (Const 1)
                            (Times (Var "x")
                                   (App (Var "fact")
                                        (Plus (Var "x")
                                              (Const (-1))))))
         let f10 = App fact (Const 10)
         print $ runResult (eval f10) []
         -- Just (IntVal 3628800)
```

MonadReader

- MonadReader defines the standard Reader interface.

```
newtype Result r a =  Result {runResult :: r -> Maybe a}

class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r          -- extract the current context

  -- replace the context with a new one and use it locally
  local :: (r -> r) -> m a -> m a

  -- run a function with the current context
  reader :: (r -> a) -> m a
  {-# MINIMAL (ask | reader), local #-}
```

- To make the Result type a proper Reader, we need to define its MonadReader instance.

MonadReader

- MonadReader defines the standard Reader interface.

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

newtype Result r a =  Result {runResult :: r -> Maybe a}

class Monad m => MonadReader r (m :: * -> *) | m -> r where
     ask :: m r
   local :: (r -> r) -> m a -> m a
  reader :: (r -> a) -> m a

instance MonadReader r (Result r) where
        ask = Result $ \ctx -> return ctx
  local f m = Result $ \ctx -> (runResult m) (f ctx)
   reader f = Result $ \ctx -> return (f ctx)

-- lift a Maybe value to a Result value.
lift' :: Maybe a -> Result r a
lift' mb = Result $ \_ -> mb
```

MonadReader

- Using standard Reader interface

```
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

lift' :: Maybe a -> Result r a

-- use 'ask' to extract context and 'lift' to promote Maybe
eval (Var s) = ask >>=
                       \ctx ->
                               lift' $ lookup s ctx
```

MonadReader

- Using standard Reader interface

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

eval (App t1 t2) = do
    fun@(FVal (f, x, t0) ctx0) <- eval t1

    arg <- eval t2

    let ctx = case f of Just name -> [(name, fun)]
                        Nothing   -> []

    -- use 'local' to make local changes to the context
    local (\_ -> ctx ++ (x, arg) : ctx0)
          (eval t0)
```

MonadReader

- Using standard Reader interface

```
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

-- use 'ask' to extract context
eval (Fn (x,t)) =
    ask >>=
           \ctx ->
                 return $ FVal (Nothing, x, t) ctx

eval (Fun (f,x,t)) =
    ask >>=
           \ctx ->
                 return $ FVal (Just f, x, t) ctx
```

Reader Transformer

- Monad can be layered using Monad transformers.

```
-- T1, T2, T3 are transformers
-- M' is the new monad built from M
type M' a = T1 (T2 (T3 M)) a
```

- ReaderT transforms another monad into a Reader monad.

```
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                -- m is a monad (& a type operator)
                ReaderT {runReaderT :: r -> m a}
```

Reader Transformer

- (ReaderT r m) is a Monad
- (ReaderT r m) is a also MonadReader

```haskell
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT r m) is an instance of Monad
instance Monad m => Monad (ReaderT r m)

-- (ReaderT r m) is an instance of MonadReader
instance Monad m => MonadReader r (ReaderT r m)
```

Reader Transformer

- (ReaderT r) is a Monad transformer.
- Any (m a) value can be lifted into a (ReaderT r m a) value

```haskell
newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT r) is an instance of MonadTrans
instance MonadTrans (ReaderT r)

-- lift any type (m a) into (t m a)
lift :: (MonadTrans t, Monad m) => m a -> t m a

lift $ Just (IntVal 1) :: ReaderT Context Maybe Val
```

Reader Transformer

- No need to 'hand-roll' a Reader.

```
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT Context) transforms Maybe
--     into a (Reader Context) + Maybe Monad
eval :: Term -> ReaderT Context Maybe Val

eval (Const a) = return $ IntVal a

-- ask and lift come for free
eval (Var s) = ask >>=
                      \ctx ->
                           lift $ lookup s ctx
```

Reader Transformer

- No need to ‘hand-roll’ a Reader.

```
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT Context) transforms Maybe
--    into a (Reader Context) + Maybe Monad
eval :: Term -> ReaderT Context Maybe Val

eval (App t1 t2) =
  do
    fun@(FVal (f, x, t0) ctx0) <- eval t1
    arg <- eval t2
    let ctx = case f of Just name -> [(name, fun)]
                        Nothing -> []

    -- local is free too
    local (\_ -> ctx ++ (x, arg) : ctx0)
          (eval t0)
```

Reader Transformer

- No need to 'hand-roll' a Reader.

```
class Monad m => MonadReader r (m :: * -> *) | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

newtype ReaderT r (m :: * -> *) a =
                ReaderT {runReaderT :: r -> m a}

-- (ReaderT Context) transforms Maybe
--    into a (Reader Context) + Maybe Monad
eval :: Term -> ReaderT Context Maybe Val

-- everything remains the same
eval (Fn (x, t)) =
       ask >>=
              \ctx ->
                     return $ FVal (Nothing, x, t) ctx
eval (Fun (f, x, t)) =
       ask >>=
              \ctx ->
                     return $ FVal (Just f, x, t) ctx
```

Exception handling with Either Monad

- Either represents a binary choice – but left is usually bad

```haskell
data Either a b = Left a | Right b

instance Functor (Either e) where
        fmap _ (Left a) = Left a
        fmap f (Right a) = Right (f a)

instance Monad (Either e) where
        return = Right
        Right m >>= k = k m
        Left e  >>= _ = Left e

instance Applicative (Either e) where
        pure = Right
        a <*> b = do x <- a; y <- b; return (x y)
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```
data Either a b = Left a | Right b

class Monad m => MonadError e (m :: * -> *) | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a

instance MonadError e (Either e) where
    throwError              = Left

    Left  l `catchError` h = h l

    Right r `catchError` _ = Right r
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

-- customized error type
data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

-- (ReaderT Context) transforms (Either EvalError) monad
eval :: Term -> ReaderT Context (Either EvalError) Val
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if variable 's' is not found
eval (Var s) = do
       ctx <- ask
       case lookup s ctx of
           Just a -> return a
           Nothing -> throwError $ VariableNotFound s
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if 'fun' is not a function
eval (App t1 t2) =
  do
    fun <- eval t1
    case fun of
      (FVal (f, x, t0) ctx0) -> do
          arg <- eval t2
          let ctx = case f of Just name -> [(name, fun)]
                              Nothing   -> []
          local (\_ -> ctx ++ (x, arg) : ctx0) $ eval t0
      _ -> throwError $ NotAFun fun
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if either operand of + is not an int
eval (Plus t1 t2) = do
    v1 <- eval t1
    v2 <- eval t2
    case (v1, v2) of
      (IntVal c1, IntVal c2) -> return $ IntVal (c1 + c2)
      (_, IntVal _) -> throwError $ NotAnInt v1
      (_, _) -> throwError $ NotAnInt v2
```

Exception handling with Either Monad

- throwError and catchError using Either monad

```haskell
data Either a b = Left a | Right b

data EvalError = VariableNotFound String
               | NotAnInt Val
               | NotABool Val
               | NotAFun Val deriving (Show)

eval :: Term -> ReaderT Context (Either EvalError) Val

-- throwError if condition of branch is not a bool
eval (IF t0 t1 t2) = do
    v <- eval t0
    case v of
        (BoolVal b) -> if b then eval t1 else eval t2
        _ -> throwError $ NotABool v
```

Exception handling with Either Monad

- run ReaderT

```
main :: IO ()
main = do
          let x = Plus (Const 10) (Const 20)
          let f = Fn ("x", Plus(Var "y") (Const 5))
          let fact = Fun ("fact", "x",
                          IF (LE (Var "x") (Const 1))
                             (Const 1)
                             (Times (Var "x") (App (Var "fact")
                     )
          let f10 = App fact (Const 10)
          print $ runReaderT (eval f10) []
-- Right (IntVal 3628800)
          let e0 = App f x
          let e1 = App x (Const 10)
          let e2 = Plus f (Const 10)
          print $ runReaderT (eval e0) []
-- Left (VariableNotFound "y")
          print $ runReaderT (eval e1) []
-- Left (NotAFun (IntVal 30))
          print $ runReaderT (eval e2) []
-- Left (NotAnInt (FVal (Nothing,"x",Plus (Var "y") (Const 5))
--                        []))
```