MonadReader Transformer (12A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

MonadReader Transformer

https://carlo-hamalainen.net/2014/03/05/note-to-self-reader-monad-transformer/
https://github.com/carlohamalainen/playground/blob/master/haskell/transformers/MyOwnReaderT.lhs

Implementing ReaderT

Installing **mtl**

sudo apt-get install cabal-install

cabal update
cabal install mtl

ghci -package such-and-such
ghc-pkg list | grep such-and-such.
ghci -hide-package <package> flag on the command line
ghc-pkg hide <package> to hide the package by default
ghc-pkg --user hide <package> home directory packages

https://wiki.haskell.org/Monad_Transformers_Explained

Auto-lifting in **mtl** MonadReader

Each **monad** in the **mtl** is defined in terms of a type class.

Reader is an instance of **MonadReader**,
ReaderT is also an instance of **MonadReader**

anything that wraps a **MonadReader** is
also set up to be a **MonadReader**

asks and **local** functions will work without any (manual) lifting.
Other **mtl monads** behave in a similar way.

https://wiki.haskell.org/Monad_Transformers_Explained

MonadReader Class Definition

```
class Monad m => MonadReader r m | m -> r where

(ask | reader), local

ask :: m r
ask = reader id

local :: (r -> r) -> m a  -> m a

reader :: (r -> a) -> m a
reader f = do
    r <- ask
    return (f r)

asks :: MonadReader r m => (r -> a)  -> m a
asks = reader
```

See examples in **Control.Monad.Reader**.

Note, the partially applied function type **(->) r** is a simple **reader** monad.

cf)

```
instance (Monad m) => Monad (ReaderT r m) where
```

http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

MonadReader Class Methods

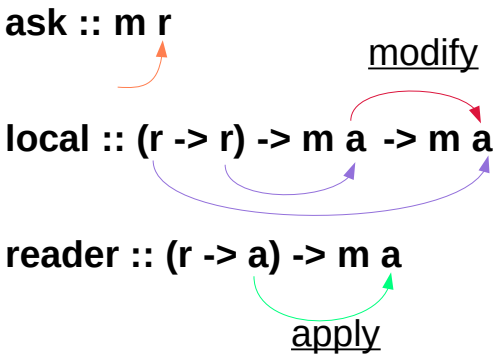
```
class Monad m => MonadReader r m | m -> r where


(ask | reader), local

ask :: m r          -- retrieves the monad environment.

local :: (r -> r)   -- the select function to modify the environment.
      -> m a        -- reader to run in the modified environment.
      -> m a        -- executes a computation in a modified environment.

reader :: (r -> a)  -- the selector function to apply to the environment.
       -> m a       -- retrieves a function of the current environment.
```



http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Reader.html

MonadReader Example

```
import Control.Monad.Reader

liftReaderT :: m a -> ReaderT r m a
liftReaderT m = ReaderT (const m)

eg2 :: ReaderT Int IO String
eg2 = do
          e <- ask :: ReaderT Int IO Int
           liftReaderT $ print $ "in eg2 the env  is: " ++ (show e)
           return $ "returned value: " ++ show e


*Main> runReaderT eg2 100
"in eg2 the env is: 100"
"returned value: 100"
```

https://gist.github.com/davidallsopp/9aaf8568349e6b8643d4

MonadReader Example

```
module ReaderMonad where
import Control.Monad.Reader

stuff :: Reader Int String
stuff = do
  s <- ask
  return (show s ++ " green bottles")

main :: IO ()
main = print $ runReader stuff 99

type IntRead = Reader Int

stuff2 :: IntRead String
stuff2 = asks show

-- stuff2 = do asks $ \s -> (show s ++ " green bottles")
```

https://gist.github.com/davidallsopp/9aaf8568349e6b8643d4

MonadReader Example

-- what's the point of **Reader**, since we could just pass
-- the parameters to the **stuff** function?

-- **Reader** is used instead of **global state**, for "**constants**" etc
-- to avoid polluting every single function with **params**
-- (which it might only pass on to other functions,
-- and not even use itself)

-- You still have to modify all these functions to use **Reader**, though

-- Can use '**asks**' as well as '**ask**' to avoid all the **do-block** boilerplate
-- and may create an **alias** for the reader if it's used in lots of places

-- see http://lambdaman.blogspot.co.uk/2007/10/monadreader.html
-- See http://stackoverflow.com/questions/14178889/reader-monad-purpose

https://gist.github.com/davidallsopp/9aaf8568349e6b8643d4

MonadReader

```
data Reader env a

instance Monad (Reader env)                -- Reader is a monad

ask :: Reader env env                      -- get its environment

runReader :: Reader env a -> env -> a      -- to run the monad
```

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

MonadReader Purpose

```
data Reader env a
instance Monad (Reader env)          -- Reader is a monad
ask :: Reader env env                -- get its environment
runReader :: Reader env a -> env -> a   -- to run the monad
```

the **reader monad** is good for passing (implicit) **configuration information** through a **computation**.

Any time you have a "**constant**" in a **computation**
that you need at various points,
but really you would like to be able
to perform the same **computation** with different **values**,
then you should use a **reader monad**.

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

MonadReader Purpose

pricing an asset can do without any monads.
To deal with multiple currencies, on the fly conversion
between currencies is needed.

```
type CurrencyDict = Map CurrencyName Dollars
currencyDict :: CurrencyDict
```

You can then call this dictionary in your code....but that won't work!
The currency dictionary is immutable and so has to be the same
not only for the life of your program, but from the time it gets compiled!

```
computePrice :: Reader CurrencyDict Dollars
computePrice
   = do currencyDict <- ask
      -- insert computation here
```

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

MonadReader Purpose

```
type CurrencyDict = Map CurrencyName Dollars
currencyDict :: CurrencyDict
currencyDict :: Map CurrencyName Dollars              -- Map k e

computePrice :: Reader CurrencyDict Dollars
computePrice
   = do currencyDict <- ask
      -- insert computation here

(Ord k, Read k, Read e) => Read (Map k e)

computePrice :: Reader CurrencyDict Dollars
computePrice :: Reader Map CurrencyName Dollars Dollars
```

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

MonadReader Purpose

```
newtype Reader env a = Reader {runReader :: env -> a}
```

Reader is just a fancy name for **functions**!
We have already defined **runReader**
every **Monad** is also a **Functor**:

```
instance Functor (Reader env) where
  fmap f (Reader g) = Reader $ f . g

instance Monad (Reader env) where
  return x = Reader (\_ -> x)
  (Reader f) >>= g = Reader $ \x -> runReader (g (f x)) x

ask = Reader $ \x -> x

local f (Reader g) = Reader $ \x -> runReader g (f x)
```

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

MonadReader Purpose

Okay, so the reader monad is just a function.
Why have Reader at all? Good question. Actually, you don't need it!

```
instance Functor ((->) env) where
  fmap = (.)

instance Monad ((->) env) where
  return = const
  f >>= g = \x -> g (f x) x
```

These are even simpler. What is more, **ask** is just **id** and **local**
is just function composition in the other order!

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

MonadReader Purpose

Expression = a **Reader**
Free variables = uses of **ask**
Evaluation environment = **Reader** execution environment.
Binding constructs = **local**

https://stackoverflow.com/questions/14178889/what-is-the-purpose-of-the-reader-monad

MonadReader Purpose

```
example :: String
example = runReader computation "Hello"
   where
      computation :: Reader String String
      computation = do
         greeting <- ask
         return $ greeting ++ ", Haskell"

main = putStrLn example

   Hello, Haskell
```

https://passy.svbtle.com/dont-fear-the-reader

MonadReader Purpose

```
example1 :: String -> String
example1 context = runReader (computation “Tom”) context
   where
      computation :: String -> Reader String String
      computation name = do
         greeting <- ask
         return $ greeting ++ name

main :: IO ()
main = putStrLn example1 “Hello”

   Hello, Tom
```

https://passy.svbtle.com/dont-fear-the-reader

MonadReader Purpose

```
example2 :: String -> String
example2 context = runReader (greet “James” >== end) context
   where
      greet :: String -> Reader String String
      greet name = do
         greeting <- ask
         return $ greeting ++ “, “ ++ name

         end :: String -> Reader String String
         end input = do
            isHello <- asks (== “Hello”)
            return $ input ++ if isHello then “!” else “.”

main :: IO ()
main = putStrLn example2 “Hello”

   Hello, James
```

https://passy.svbtle.com/dont-fear-the-reader

MonadReader Purpose

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

instance Monad (Reader r) where
    return a = Reader $ \_ -> a
    m >>= k = Reader $ \r -> runReader (k $ runReader m r) r

asks :: (r -> a) -> Reader r a
asks f = Reader f

ask :: Reader a a
ask = Reader id
```

https://passy.svbtle.com/dont-fear-the-reader

Data Map (dictionary) Example

```
import Data.Map (Map, (!))
import qualified Data.Map as Map

main = do
   let m0  = Map.empty
   let m1  = Map.insert "k1" 7 m0
   let m   = Map.insert "k2" 13 m1
   putStrLn $ "map: " ++ show m

   let v1  = m ! "k1"
   putStrLn $ "v1: " ++ show v1
   putStrLn $ "len: " ++ show (Map.size m)
   let m'  = Map.delete "k2" m
   putStrLn $ "map: " ++ show m'
   let prs = Map.lookup "k2" m'
   putStrLn $ "prs: " ++ show prs
   let n   = Map.fromList [("foo", 1), ("bar", 2)]
   putStrLn $ "map: " ++ show n
```

```
$ runhaskell maps.hs
map: fromList [("k1",7),("k2",13)]
v1: 7
len: 2
map: fromList [("k1",7)]
prs: Nothing
map: fromList [("bar",2),("foo",1)]
```

https://lotz84.github.io/haskellbyexample/ex/maps

Data Map (dictionary) Example

```
import Prelude hiding (lookup)
import Data.Map

employeeDept        = fromList([("John","Sales"), ("Bob","IT")])
deptCountry         = fromList([("IT","USA"), ("Sales","France")])
countryCurrency     = fromList([("USA", "Dollar"), ("France", "Euro")])

employeeCurrency :: String -> Maybe String
employeeCurrency name = do
   dept <- lookup name employeeDept
   country <- lookup dept deptCountry
   lookup country countryCurrency

main = do
   putStrLn $ "John's currency: " ++ (show (employeeCurrency "John"))
   putStrLn $ "Pete's currency: " ++ (show (employeeCurrency "Pete"))
```

```
John's currency: Just "Euro"
Pete's currency: Nothing
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Data Map (dictionary) Example

```
type Bindings = Map String Int;

-- Returns True if the "count" variable contains correct bindings size.
isCountCorrect :: Bindings -> Bool
isCountCorrect bindings = runReader calc_isCountCorrect bindings

-- The Reader monad, which implements this complicated check.
calc_isCountCorrect :: Reader Bindings Bool
calc_isCountCorrect = do
    count <- asks (lookupVar "count")
    bindings <- ask
    return (count == (Map.size bindings))

-- The selector function to  use with 'asks'.
-- Returns value of the variable with specified name.
lookupVar :: String -> Bindings -> Int
lookupVar name bindings = maybe 0 id (Map.lookup name bindings)
```

```
sampleBindings = Map.fromList [("count",3), ("1",1), ("b",2)]

main = do
    putStr $ "Count is correct for bindings " ++ (show sampleBindings) ++ ": ";
    putStrLn $ show (isCountCorrect sampleBindings);
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Data Map (dictionary) Example

```
calculateContentLen :: Reader String Int
calculateContentLen = do
    content <- ask
    return (length content);

-- Calls calculateContentLen after adding a prefix to the Reader content.
calculateModifiedContentLen :: Reader String Int
calculateModifiedContentLen = local ("Prefix " ++) calculateContentLen

main = do
    let s = "12345";
    let modifiedLen = runReader calculateModifiedContentLen s
    let len = runReader calculateContentLen s
    putStrLn $ "Modified 's' length: " ++ (show modifiedLen)
    putStrLn $ "Original 's' length: " ++ (show len)
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Data Map (dictionary) Example

```
-- The Reader/IO combined monad, where Reader stores a string.
printReaderContent :: ReaderT String IO ()
printReaderContent = do
    content <- ask
    liftIO $ putStrLn ("The Reader Content: " ++ content)

main = do
    runReaderT printReaderContent "Some Content"
```

https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf