Monad Transformer (3I)

Young Won Lim

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

Monad Transformers

Using several monads at once for more **functionality**

a function could use both **I/O** and **Maybe exception handling**

While a nested type like **IO (Maybe a)** would work just fine,
it would force us to **do pattern matching**
within **IO do-blocks** to extract values,
something that the **Maybe monad** offers to remove

monad transformers:
special **types** that allow us to roll two **monads**
into a single one that shares the **behavior** of both.
(**functionality**)

https://wiki.haskell.org/Lifting

Monad Transformer Name Convetion

define a **monad transformer** that gives the **IO** **monad**
some characteristics (functionality) of the **Maybe** **monad**;
it is called **MaybeT**

Maybe → IO

monad transformers have a "**T**" appended
to the **name** of the **monad**
whose **characteristics** they provide.
(**functionality**, **behavior**)

https://wiki.haskell.org/Lifting

Packages for Monad Transformers

There are currently *several* ***packages***
that implement similar **interfaces** to **monad transformers**

besides an additional **package** with a similar goal
but different **interfaces** (API) named **MonadLib**

- **transformers** package
- **mtl** (**monad transformer library**) package
- **monads-fd** package
- **monads-tf** package

https://wiki.haskell.org/Monad_Transformers

The **transformers** package

classes:
MonadTrans
MonadIO,
concrete monad transformers (**instances**)
StateT, etc

multi-parameter type synonyms
The monad **State s a** is only
a type synonym for **StateT s Identity a**.

Thus both **State** and **StateT** can be accessed
by the same methods like **put** and **get**.

However, this only works if **StateT** is
the top-most transformer
in a monad transformer stack.

transformers package
MonadTrans class
MonadIO class
StateT instance

https://wiki.haskell.org/Monad_Transformers

The **transformers** package

A portable library of **functor** and **monad transformers**

the **monad transformer class** (in **Control.Monad.Trans.Class**)
concrete functor and **monad transformers**
each with associated operations and functions
to **lift** operations associated with other transformers.

The package can be used on its own in **portable** Haskell code,
in which case operations need to be **manually lifted**
through **transformer stacks**

Alternatively, it can be used with the **non-portabl**e monad classes
in the **mtl** or **monads-tf** packages,
which **automatically lift** operations introduced
by monad transformers through other transformers.

http://hackage.haskell.org/package/transformers

The **version 1 mtl** package

version 1 mtl : the first implementation, this version is now obsolete.

classes

- **MonadTrans**
- **MonadIO**

concrete monad transformers

- **StateT**, etc.

multi-parameter type classes with functional dependencies

- **MonadState**, etc.

Monads like **State** and their transformer counterparts like **StateT**
are distinct types and can be accessed uniformly
only through a type class abstraction like **MonadState**.

ver 1 mtl package
MonadTrans class
MonadIO class
StateT instance
MonadState class

https://wiki.haskell.org/Monad_Transformers

The **version 2 mtl** package

version 2 mtl :
<u>re-exports</u> the **classes** and **monad transformers**
of the **transformers** package,
and adds **<u>multi-parameter</u> type classes**
with <u>functional dependencies</u> such as **MonadState**.

classes
- **MonadTrans**
- **MonadIO**

concrete monad transformers
- **StateT**, etc.

} <u>re-exports</u> of the **transformers** package,

<u>multi-parameter</u> type classes with <u>functional dependencies</u>
- **MonadState**, etc.

ver 2 mtl package
MonadTrans class
MonadIO class
StateT instance
MonadState class

re-exports of transformer package
+
multi-parameter type classes

https://wiki.haskell.org/Monad_Transformers

The **transformers** vs **mtl** packages

transformers

Control.Monad.Signatures
Trans
- Control.Monad.Trans.Accum
- Control.Monad.Trans.Class
- Control.Monad.Trans.Cont
- Control.Monad.Trans.Error
- Control.Monad.Trans.Except
- Control.Monad.Trans.Identity
- Control.Monad.Trans.List
- Control.Monad.Trans.Maybe
- Control.Monad.Trans.RWS
  - Control.Monad.Trans.RWS.Lazy
  - Control.Monad.Trans.RWS.Strict
- Control.Monad.Trans.Reader
- Control.Monad.Trans.Select
- Control.Monad.Trans.State
  - Control.Monad.Trans.State.Lazy
  - Control.Monad.Trans.State.Strict
- Control.Monad.Trans.Writer
  - Control.Monad.Trans.Writer.Lazy

Control.Monad.Trans.Writer.Strict

https://hackage.haskell.org/package/transformers

mtl

- Control.Monad.Cont
  - Control.Monad.Cont.Class
- Control.Monad.Error
  - Control.Monad.Error.Class
- Control.Monad.Except
- Control.Monad.Identity
- Control.Monad.List
- Control.Monad.RWS
  - Control.Monad.RWS.Class
  - Control.Monad.RWS.Lazy
  - Control.Monad.RWS.Strict
- Control.Monad.Reader
  - Control.Monad.Reader.Class
- Control.Monad.State
  - Control.Monad.State.Class
  - Control.Monad.State.Lazy
  - Control.Monad.State.Strict
- Control.Monad.Trans
- Control.Monad.Writer
  - Control.Monad.Writer.Class
  - Control.Monad.Writer.Lazy
  - Control.Monad.Writer.Strict

The **transformers** and **mtl** package

1 **MTL** and **transformers** use different module names,
but share common classes, type constructors and functions,
so they are fully compatible.

2 **Transformers** is Haskell 98 and thus more portable,
and doesn't tie you to functional dependencies.
But because it lacks the monad classes, you'll have
to **lift** operations manually to the composite monad yourself.

3 Many package using **MTL** can be ported to **transformers**
with only slight modifications.
Modules require the **Trans** infix, e.g.
For constructing you must use the function **state**
and instead of matching patterns you must call **runState**.

import **Control.Monad.State** ...
import **Control.Monad.Trans.State** ....

Since **State** is only a **type synonym**,
there is no longer a **constructor**
named **State**.

http://hackage.haskell.org/package/transformers

Automatic and Manual Lifting

The **transformers** package contains

- the **monad transformer** class (in Control.Monad.Trans.Class)
- concrete functor and monad transformers,
  each with associated operations and functions
  to lift operations associated with other transformers.

The **transformers** package can be used
on its own in *portable* Haskell code,
in which case operations need to be *manually lifted*
through **transformer stacks**

Alternatively, it can be used with the *non-portable* monad classes
in the **mtl** or **monads-tf** packages,
which *automatically lift* operations
introduced by monad transformers through other transformers.

transformers package
: manual lifting

mtl (monads-tf) package
: automatic lifting

https://hackage.haskell.org/package/transformers

Monad Transformer Class

Control.Monad.Trans.Class

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a



-- lifts a value from the inner monad m
-- to the transformed monad t m
-- could be called lift0

Laws
    lift . return = return
    lift (m >>= f) = lift m >>= (lift . f)
```

https://hackage.haskell.org/package/transformers-0.5.5.0/docs/Control-Monad-Trans-Class.html

Monad Transformer Instances

```
Control.Monad.Trans.Class

MonadTrans ListT
MonadTrans MaybeT
MonadTrans (ErrorT e)
MonadTrans (ExceptT e)
MonadTrans (IdentityT :: (* -> *) -> * -> *)
MonadTrans (SelectT r)
MonadTrans (StateT s)
MonadTrans (StateT s)
Monoid w => MonadTrans (WriterT w)
Monoid w => MonadTrans (AccumT w)
Monoid w => MonadTrans (WriterT w)
MonadTrans (ContT r)
MonadTrans (ReaderT r :: (* -> *) -> * -> *)
Monoid w => MonadTrans (RWST r w s)
Monoid w => MonadTrans (RWST r w s)
```

https://hackage.haskell.org/package/transformers-0.5.5.0/docs/Control-Monad-Trans-Class.html

Transformer Stacks

making a double, triple, quadruple, ... monad
by wrapping around existing monads
that provide wanted **functionality**.

the **innermost monad** is usually **Identity** or **IO**
but it can be any monad.
monad transformers wrap around this monad
to make bigger, better monads.

a ➡ M a ➡ N M a ➡ O N M a

To do stuff in an **inner monad** → cumbersome → **auto-lifting** mtl

lift $ lift $ lift $ foo

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

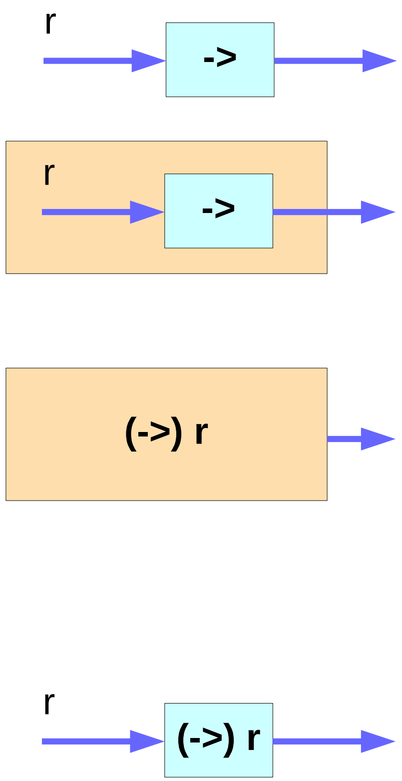
((->) r)

(+) 1 2 -- prefix function
(*) 3 4 -- prefix function

1 + 2 -- infix function
3 * 4 -- infix function

(->) r a

r -> a



https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

mtl MonadIO

class Monad m => MonadIO m where

Monads in which IO computations may be embedded.
Any monad built by applying a sequence of monad transformers
to the IO monad will be an instance of this class.

Instances should satisfy the following laws,
which state that liftIO is a transformer of monads:

```
liftIO . return = return
liftIO (m >>= f) = liftIO m >>= (liftIO . f)
```

liftIO Lift a computation from the IO monad.

```
liftIO :: IO a -> m a
```

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

Auto-lifting in **mtl** MonadReader

configuration that would be **global** (in an imperative program)
because client handling threads all need to query it.

data Config = Config Foo Bar Baz

to use currying and making all the client threads of type **Config -> IO ()**

Not good because any functions they call
have to be passed the **Config** parameter manually.

https://wiki.haskell.org/Monad_Transformers_Explained

Auto-lifting in **mtl** MonadReader

The **Reader** **monad** solves this problem

need to wrap **IO** in a **ReaderT**

The **type constructor** for **ReaderT** is

ReaderT r m a

r	the shared **environment** to read from
m	the **inner monad**
a	the **return** type.

https://wiki.haskell.org/Monad_Transformers_Explained

Auto-lifting in **mtl** MonadReader

client_func :: ReaderT Config IO ()

Config	the shared **environment**
IO	the **inner monad**
()	the **return** type.

We can then use the **ask**, **asks** and **local** functions
as if **Reader** was the only Monad:
(these examples are inside do blocks)

p <- asks port

https://wiki.haskell.org/Monad_Transformers_Explained

Auto-lifting in **mtl** MonadReader

This is all well and good, but the **client_func**
now has type **ReaderT Config IO ()**
and **forkIO** needs a function of type **IO ()**

The escape function for **Reader**

runReader :: Reader r a -> r -> a

Similarly, the escape function for **ReaderT**

runReaderT :: ReaderT r m a -> r -> m a

(Given some **c :: Config**
that's been assembled from config files or the like)

forkIO (runReaderT client_func c)

https://wiki.haskell.org/Monad_Transformers_Explained

Auto-lifting in **mtl** MonadIO

A type class called **MonadIO**
is used to implement a similar trick as above.

IO is an instance of **MonadIO**
any **mtl transformer** that wraps a **MonadIO** instance
also is an instance of **MonadIO**

This means that **IO** functions need only use **liftIO**
and not a big chain of **lifts**.

Note also that **IO** has no **transformer**
always be the **innermost monad**.

https://wiki.haskell.org/Monad_Transformers_Explained

Auto-lifting in **mtl** MonadIO

IO is an instance of **MonadIO**
any **mtl transformer** that wraps a **MonadIO** instance
also is an instance of **MonadIO**

thus, **IO** functions need only use **liftIO**
and not a big chain of lift's.

(given **h** :: Handle, the client's handle)
liftIO $ **hPutStrLn** **h** "You win"
liftIO $ **hFlush** **h**

Note also that **IO** has no transformer
must therefore always be the **innermost** monad.

https://wiki.haskell.org/Monad_Transformers_Explained

Monad Transformers

Precursor	Transformer	Original Type	Combined Type
Writer	WriterT	(a, w)	m (a, w)
Reader	ReaderT	r -> a	r -> m a
State	StateT	s -> (a, s)	s -> m (a, s)
Cont	ContT	(a -> r) -> r	(a -> m r) -> m r

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

MaybeT

Define a **monad transformer** that gives the **IO** monad
some characteristics of the **Maybe** monad;
Call it **MaybeT**.

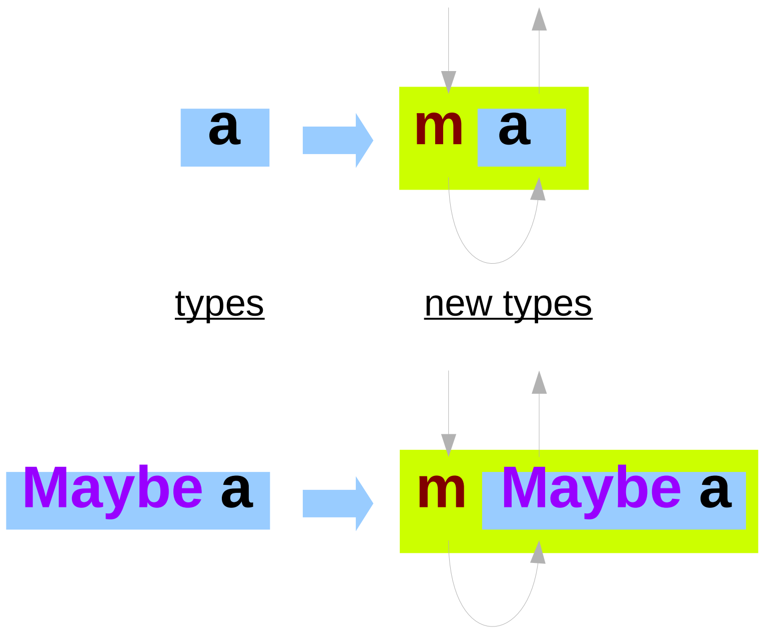
MaybeT is a wrapper around **m (Maybe a)**,
where **m** can be <u>any</u> <u>monad</u> (**IO** in our example):

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

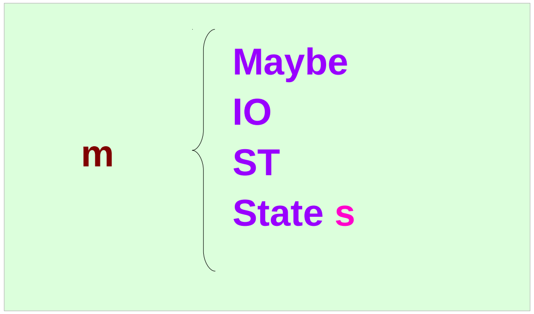
m {
- **Maybe**
- **IO**
- **ST**
- **State s**

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT



types new types

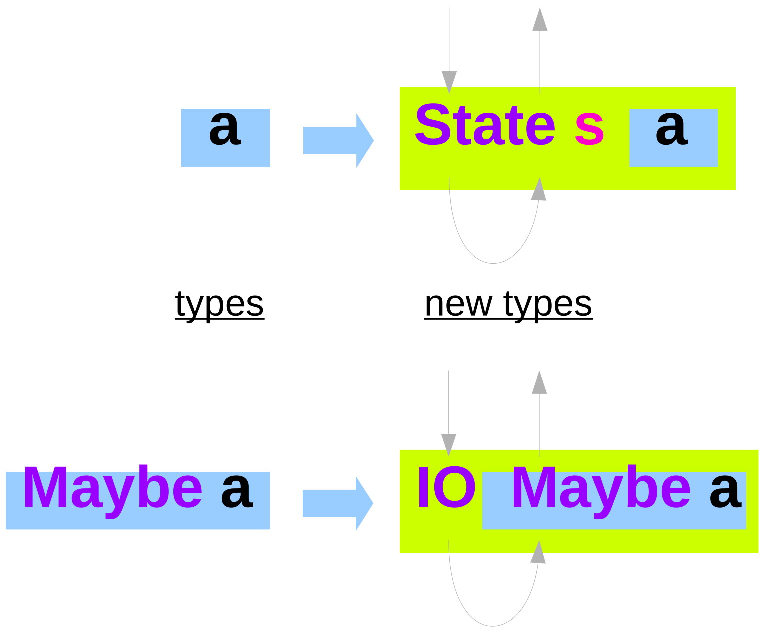


newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

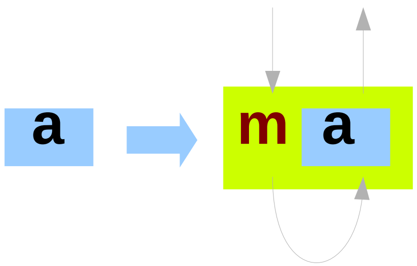
https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

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



newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }



https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

This data type definition specifies

- a type constructor **MaybeT**
- a parameter **m**,
- a term (value) constructor **MaybeT**,
- an accessor function **runMaybeT**,

The whole point of **monad transformers** is
that they transform monads into monads;
and so we need to make **MaybeT m**
an instance of the Monad class:

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }



instance Monad m => Monad (MaybeT m) where
  return  = MaybeT . return . Just

  -- The signature of (>>=), specialized to MaybeT m:
  -- (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
  x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                    case maybe_value of
                         Nothing    -> return Nothing
                         Just value -> runMaybeT $ f value
```

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

return = MaybeT . return . Just

It would also have been possible (though arguably less readable) to write the return function as: **return = MaybeT . return . return**

x >>= f = MaybeT $ do maybe_value <- runMaybeT x

First, the **runMaybeT** accessor unwraps **x** into an **m** (**Maybe a**) computation. That shows us that the whole **d**o block is in **m**. Still in the first line, **<-** extracts a **Maybe a** value from the unwrapped computation.

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

```
case maybe_value of
   Nothing    -> return Nothing
   Just value -> runMaybeT $ f value
```

The case statement tests **maybe_value**:

With **Nothing**, we return **Nothing** into m;

With **Just**, we apply **f** to the value from the **f** .

Since **f** has **MaybeT m b** as result type,

we need an extra **runMaybeT**

to put the result back into the **m** monad.

Finally, the do block as a whole has **m** (**Maybe b**) type;
so it is wrapped with the **MaybeT** constructor.

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

It may look a bit complicated;
but aside from the copious amounts of wrapping and unwrapping,
the implementation does the same
as the familiar bind operator of Maybe:

```
-- (>>=) for the Maybe monad
maybe_value >>= f = case maybe_value of
                        Nothing -> Nothing
                        Just value -> f value
```

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

Why use the **MaybeT** constructor before the do block
while we have the accessor **runMaybeT** within do?
Well, the do block must be in the **m** monad, not in **MaybeT m**
(which lacks a defined bind operator at this point).

Technically, this is all we need; however,
it is convenient to make **MaybeT m** an instance of a few other classes:

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

```
instance Monad m => Alternative (MaybeT m) where
    empty   = MaybeT $ return Nothing
    x <|> y = MaybeT $ do maybe_value <- runMaybeT x
                          case maybe_value of
                               Nothing    -> runMaybeT y
                               Just _     -> return maybe_value

instance Monad m => MonadPlus (MaybeT m) where
    mzero = empty
    mplus = (<|>)

instance MonadTrans MaybeT where
    lift = MaybeT . (liftM Just)
```

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

MonadTrans implements the lift function,
so we can take functions from the m monad and
bring them into the MaybeT m monad
in order to use them in do blocks.
As for Alternative and MonadPlus,
since Maybe is an instance of those class it makes sense
to make the MaybeT m an instance too.

https://wiki.haskell.org/Monad_Transformers_Explained

MaybeT

```
getPassphrase :: IO (Maybe String)
getPassphrase = do s <- getLine
                   if isValid s then return $ Just s
                               else return Nothing

-- The validation test could be anything we want it to be.
isValid :: String -> Bool
isValid s = length s >= 8
          && any isAlpha s
          && any isNumber s
          && any isPunctuation s
```

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

MaybeT

```
askPassphrase :: IO ()
askPassphrase = do putStrLn "Insert your new passphrase:"
                   maybe_value <- getPassphrase
                   case maybe_value of
                       Just value -> do putStrLn "Storing in database..."  -- do stuff
                       Nothing -> putStrLn "Passphrase invalid."
```

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

MaybeT

```
askPassphrase :: IO ()
askPassphrase = do putStrLn "Insert your new passphrase:"
                   maybe_value <- getPassphrasegetPassphrase :: MaybeT IO String
getPassphrase = do s <- lift getLine
                   guard (isValid s) -- Alternative provides guard.
                   return s

askPassphrase :: MaybeT IO ()
askPassphrase = do lift $ putStrLn "Insert your new passphrase:"
                   value <- getPassphrase
                   lift $ putStrLn "Storing in database..."

                   case maybe_value of
                     Just value -> do putStrLn "Storing in database..."  -- do stuff
                     Nothing -> putStrLn "Passphrase invalid."
```

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

MaybeT

```
askPassphrase :: MaybeT IO ()
askPassphrase = do lift $ putStrLn "Insert your new passphrase:"
                   value <- msum $ repeat getPassphrase
                   lift $ putStrLn "Storing in database..."
```

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

References

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