Monad Transformers for the Easily Confused

•••

Github Repo:

https://github.com/TheWizardTower/monadTransformers

My friends have horrible senses of humor.

Ken Burnside Transformers! Monads in Disguise!

(You may not be old enough for that to be a horrifying earworm.)

TheWizardTower @TheWizardTower · Mar 7

More good news:

I'm speaking at LambdaConf 2018!! I'll be giving a talk on Monad Transformers. :-D

cc @snoyberg @LambdaMom @lambda_conf @mattoflambda

4 3 26

Miriam Snoyman
@LambdaMom

Following

Replying to @TheWizardTower @snoyberg and 2 others

TRANSFORMERS: MONADS IN DISGUISE

9:23 PM - 7 Mar 2018

4 Likes

1 4

Tweet your reply

TheWizardTower @TheWizardTower · Mar 7

Replying to @LambdaMom @snoyberg and 2 others

You are literally the second person to make this joke to me.

Today.

2 2

Miriam Snoyman @LambdaMom · Mar 7

So it's finally catching on? 7 years after I made the Haskellers shop?

shop.spreadshirt.com/haskellers/tra…

2

A word to begin

Don’t fret if this talk doesn’t make the lightbulb go off.

A word to begin

Don’t fret if this talk doesn’t make the lightbulb go off.

A lot of this is sitting with the concepts for a while and letting them stew.

It’s a lot like learning Algebra or Calculus for the first time. You have to give yourself time to make the cognitive leaps.

A word to begin

Be patient! Keep trying!

Talk Summary

Monad Transformers Transform Monads

…but how?

Talk Summary

Monad Transformers Transform Monads

We’re going to review Monoid, Functor, Applicative, and Monad laws, then see if we can figure out a way to get to what MTs do for us.

Monoids

"associative, binary operations with an identity."

Source: http://hackage.haskell.org/package/base-4.11.1.0/docs/Data-Monoid.html

Monoids -- Associative

Associative operatons mean that:

```
(a <op> (b <op> c)) == ((a <op> b) <op> c)
```

Monoids -- Binary

Binary operations are operations that take two values, and produce a result.

Monoids -- Binary

Binary operations are operations that take two values, and produce a result.

To illustrate, exponentiation and absolute value are examples of **unary** operations.

Monoids -- Binary

Binary operations are operations that take two values, and produce a result.

To illustrate, exponentiation and absolute value are examples of **unary** operations.

Binary operations include addition, subtraction, multiplication, division, list concatenation, and more.

Monoid -- With Identity

Monoidal operations must have an identity value, such that

```
a <op> id == a == id <op> a
```

Monoid -- With Identity

Monoidal operations must have an identity value, such that

```
a <op> id == a == id <op> a
```

Examples include 0 for addition, or the empty list for list concatenation.

Functors

Functors obey two laws

Functors

Functors obey two laws

```
fmap id = id
```

Functors

Functors obey two laws

```
fmap id = id
```

```
fmap g . fmap h == fmap (g . h)
```

Applicative

Applicative

```
pure id <*> v = v                                -- Identity
pure f <*> pure x = pure (f x)                    -- Homomorphism
u <*> pure y = pure ($ y) <*> u                   -- Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

(source: https://en.wikibooks.org/wiki/Haskell/Applicative_functors)

Applicative

Obviously, there’s a lot going on, so let’s unpack these one by one.

Applicative -- Identity

```
pure id <*> v = v                                    -- Identity
```

Much like the first functor law, this asserts that `pure id` does not distort the function in any way.

“Is your applicative instance doing sneaky things, Y/N?”

Applicative -- Homomorphism

```
pure f <*> pure x = pure (f x)                    -- Homomorphism
```

Applicative -- Homomorphism

```
pure f <*> pure x = pure (f x)                -- Homomorphism
```

Remember the second functor law?

Applicative -- Homomorphism

```
pure f <*> pure x = pure (f x)                    -- Homomorphism
```

Remember the second functor law?

```
fmap g . fmap h = fmap (g . h)
```

Applicative -- Interchange

```
u <*> pure y = pure ($ y) <*> u                    -- Interchange
```

Applicative -- Interchange

```
u <*> pure y = pure ($ y) <*> u                -- Interchange
```

This becomes clearer with a concrete example.

Applicative -- Interchange

```
*Main> pure ($ 5) <*> Just (+10)

Just 15

*Main> Just (+10) <*> pure 5

Just 15

*Main> (pure ($ 5) <*> Just (+10)) == (Just (+10) <*> pure
5)

True
```

Applicative -- Interchange

(source: https://stackoverflow.com/questions/27285918/applicatives-interchange-law)

Applicative -- Interchange

Referencing the typeclassopedia, we find:

"Intuitively, this says that when evaluating the application of an effectful function to a pure argument, the order in which we evaluate the function and its argument doesn't matter."

(source: https://wiki.haskell.org/Typeclassopedia#Laws_2)

Bonus Round!

```
u <*> pure y = pure ($ y) <*> u                -- Interchange
```

Why `($ 5)` ?

Bonus Round!

```
u <*> pure y = pure ($ y) <*> u                    -- Interchange
```

Why `($ 5)` ?

Let's ask GHCi

Bonus Round!

```
*Main> :t ($ 5)

($ 5) :: Num a => (a -> b) -> b

*Main>
```

Bonus Round!

We can rewrite

```
($ 5)
```

Into:

```
(\f -> f 5)
```

Even GHC agrees!

Bonus Round!

```
*Main> :t (\f -> f 5)

(\f -> f 5) :: Num t1 => (t1 -> t2) -> t2

*Main>
```

Bonus Round!

```
*Main> :t (\f -> f 5)

(\f -> f 5) :: Num t1 =>
(t1 -> t2) -> t2

*Main>
```

```
*Main> :t ($ 5)

($ 5) :: Num a => (a -> b)
-> b

*Main>
```

Bonus Round!

TheWizardTower @TheWizardTower · May 28

Haskell Peeps!

Applicative laws. Specifically the interchange law.

u <*> pure y = pure ($ y) <*> u

Can someone explain why the $ is necessary in ($ y) ?

1

Harry Pan @xingzhipan · May 28

So that y is the second argument to $.

1

TheWizardTower @TheWizardTower · May 29

So,

Just ($ 5) <*> Just (+10)

would get rewritten as

Just $ +10 5

Right?

1

Michael Snoyman
@snoyberg

Following

Replying to @TheWizardTower @xingzhipan

Not quite, you need some parens. But ($ x) is equivalent to (\f -> f $ x), if that helps. It's called an operator section. And (f $ x) is the same as (f x)

12:04 AM - 29 May 2018

Applicative -- Composition

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

Applicative -- Composition

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

Again, concrete examples will make this clearer

Applicative -- Composition

```
*Main> pure (.) <*> Just (+10) <*> Just (+20) <*> Just 30

Just 60

*Main> Just (+10) <*> (Just (+20) <*> Just 30)

Just 60

*Main> (pure (.) <*> Just (+10) <*> Just (+20) <*> Just 30)
== (Just (+10) <*> (Just (+20) <*> Just 30))

True
```

Applicative -- Composition

Let’s walk through that again.

Applicative -- Composition

```
(pure (.) <*> Just (+10) <*> Just (+20)
```

becomes

```
Just (+20 . +10)
```

Applicative -- Composition

```
Just (+20 . +10)
```

Adding back the `<*> Just 30`:

```
Just (+20 . +10) <*> Just 30
```

Applicative -- Composition

```
Just (+20 . +10) <*> Just 30
```

is simplified down to

```
Just 60
```

Monad

Monad

- Must be burritos
- Must be delicious
- Can swap out burrito type with monadic burrito actions

Monad

(okay, okay)

Monad

```
return a >>= k  =  k a -- left identity

m >>= return  =  m       -- right identity

m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h -- associativity
```

Monad -- Left Identity

```
return a >>= k  =  k a
```

```
*Main> return 5 >>= (\x ->
Just (x + 5))

Just 10

*Main>
```

Monad -- Right Identity

```
m >>= return = m
```

```
*Main> Just 5 >>= return
Just 5
*Main>
```

Monad -- Associativity

```
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Monad -- Associativity

```
*Main> let f = (\x -> Just (5 - x))

*Main> return 5 >>= f

Just 0

*Main> return (-5) >>= f

Just 10
```

Monad -- Associativity

```
*Main> let g = (\x -> Just (10 - x))

*Main> return 5 >>= f >>= g

Just 10

*Main> return 5 >>= g >>= f

Just 0
```

Monad -- Associativity

```
*Main> Just 5 >>= (\x -> f x >>= g)

Just 10

*Main> (Just 5 >>= f) >>= g

Just 10

*Main>
```

Monad -- Remix

```
return >=> m  = m                              -- left identity

m >=> return  = m                              -- right identity

(f >=> g) >=> h = f >=> (g >=> h)    -- associativity
```

Monad -- Remix

```
return >=> m   = m                          -- left identity

m >=> return   = m                          -- right identity

(f >=> g) >=> h = f >=> (g >=> h)     -- associativity
```

(source: https://wiki.haskell.org/Monad_laws)

A Closer Look at >>=

Courtesy of Category Theory, we know that >>= can be expressed as

```
m >>= k = join $ fmap k m
```

A Closer Look at >>=

```
*Main> :t (>>=)

(>>=) :: Monad m => m a ->
(a -> m b) -> m b
```

A Closer Look at >>=

```
(>>=) :: Monad m =>

          m a

        -> (a -> m b)

        -> m b
```

A Closer Look at >>=

```
(>>=) :: Monad m =>
         m a
       -> (a -> m b)
       -> m b
```

```
*Main> :t fmap
fmap :: Functor f =>
          (a -> b)
       -> f a
       -> f b
```

A Closer Look at >>=

Remember that `m >>= k = join $ fmap k m`

```
*Main> fmap (\a -> Just (a + 10)) (Just (10))

Just (Just 20)
```

A Closer Look at >>=

```
*Main> join $ fmap (\a -> Just (a + 10)) (Just (10))

Just 20



*Main> (flip (>>=)) (\a -> Just (a + 10)) (Just (10))

Just 20
```

A Closer Look at >>=

```
*Main> join $ fmap (\a -> Just (a + 10)) (Just (10))

Just 20



*Main> (=<<) (\a -> Just (a + 10)) (Just (10))

Just 20
```

A Closer Look at >>=

```
*Main> :t join

join :: Monad m => m (m a) -> m a
```

A Closer Look at >>=

```
*Main> :t join

join :: Monad m => m (m a) -> m a
```

There's an important note here: The monadic contexts you're joining have to match!

A Closer Look at >>=

```
*Main> join $ Just $ Just 10

Just 10

*Main> join $ Just Nothing

Nothing

*Main> join $ Just $ Just Nothing

Just Nothing
```

A Closer Look at >>=

```
*Main Control.Monad> join [[1],[2],[3]]

[1,2,3]

*Main Control.Monad> join [[[1,2,3],[4,5,6],[7,8,9]]]

[[1,2,3],[4,5,6],[7,8,9]]
```

```
*Main> join [Just 10, Just 15, Just 20]



<interactive>:92:7: error:

• Couldn't match expected type '[a]'

with actual type 'Maybe Integer'

• In the expression: Just 10

In the first argument of 'join', namely

'[Just 10, Just 15, Just 20]'

In the expression: join [Just 10, Just 15, Just 20]

• Relevant bindings include it :: [a] (bound at <interactive>:92:1)
```

Let’s make $>>=$ for IdentityT!

Let's make $>>=$ for IdentityT!

First step, define our arguments.

Let’s make >>= for IdentityT!

```
(\a ->
  IdentityT (Just (a + 10))
```

Let’s make >>= for IdentityT!

```
(\a ->                                IdentityT (Just 10)

  IdentityT (Just (a + 10))
```

Let's make >>= for IdentityT!

```
*Main> fmap (\a -> IdentityT (Just (a + 10))) (IdentityT
(Just 10))



IdentityT (Just (IdentityT (Just 20)))
```

Let's make >>= for IdentityT!

```
IdentityT (Just (IdentityT (Just 20)))
```

`join` can't reduce any part of this -- none of the adjacent monads match.

Let's make >>= for IdentityT!

```
IdentityT (Just (IdentityT (Just 20)))
```

`join` can't reduce any part of this -- none of the adjacent monads match.

There is a function that can help, though.

Let’s make >>= for IdentityT!

```
*Main> :t runIdentityT

runIdentityT :: IdentityT f a -> f a
```

Let's make >>= for IdentityT!

```
*Main> fmap (\a -> IdentityT (Just (a + 10))) (IdentityT
(Just 10))


IdentityT (Just (IdentityT (Just 20)))
```

Let’s make >>= for IdentityT!

```
*Main> fmap (runIdentityT . (\a -> IdentityT (Just (a +
10)))) (IdentityT (Just 10))


IdentityT (Just (Just 20))
```

Let’s make >>= for IdentityT!

```
*Main> fmap (runIdentityT . (\a -> IdentityT (Just (a +
10)))) (IdentityT (Just 10))
```

```
IdentityT (Just (Just 20))
```

Progress! Still not correct, but progress all the same!

Let’s make >>= for IdentityT!

```
*Main> runIdentityT $ fmap (runIdentityT . (\a -> IdentityT
(Just (a + 10)))) (IdentityT (Just 10))



(Just (Just 20))
```

Let’s make >>= for IdentityT!

```
*Main> join $ runIdentityT $ fmap (runIdentityT . (\a ->
IdentityT (Just (a + 10)))) (IdentityT (Just 10))
```

```
Just 20
```

Let’s make >>= for IdentityT!

```
*Main> IdentityT $ join $ runIdentityT $ fmap (runIdentityT
. (\a -> IdentityT (Just (a + 10)))) (IdentityT (Just 10))


IdentityT Just 20
```

Let’s make >>= for IdentityT!

```
*Main> IdentityT $ join $ runIdentityT $ fmap (runIdentityT
. (\a -> IdentityT (Just (a + 10)))) (IdentityT (Just 10))
```

```
IdentityT Just 20
```

We did it!

Let’s make >>= for IdentityT!

However, we did it the *really really ugly* way.

Let’s make >>= for IdentityT!

```
iTBind :: Monad f =>

        IdentityT f a

     -> (a -> IdentityT f b)

     -> IdentityT f b

iTBind (IdentityT fa) ab = IdentityT $ join $ fmap
(runIdentityT . ab) fa
```

Let’s make >>= for IdentityT!

We can also look at the monad instance for IdentityT in the transformers library!

```
instance (Monad m) => Monad (IdentityT m) where
#if !(MIN_VERSION_base(4,8,0))
    return = IdentityT . return
    {-# INLINE return #-}
#endif
    m >>= k = IdentityT $ runIdentityT . k =<< runIdentityT m
    {-# INLINE (>>=) #-}
    fail msg = IdentityT $ fail msg
    {-# INLINE fail #-}
```

Let’s make >>= for IdentityT!

```
m >>= k = IdentityT $ runIdentityT . k =<< runIdentityT m
```

Let's make >>= for IdentityT!

Reminder:

```
*Main> :t (>>=)

(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Let's make >>= for IdentityT!

Reminder V2: We've seen =<< before!

```
*Main> :t (=<<)

(=<<) :: Monad m => (a -> m b) -> m a -> m b

*Main> :t (flip (>>=))

(flip (>>=)) :: Monad m => (a -> m b) -> m a -> m b
```

Let’s make >>= for IdentityT!

```
(>>=) :: Monad m =>
         m a
      -> (a -> m b)
      -> m b
```

```
runIdentityT . k
```

Let’s make >>= for IdentityT!

```
(>>=) :: Monad m =>
         m a
      -> (a -> m b)
      -> m b
```

```
runIdentityT . k :: (a -> m b)
```

Let’s make >>= for IdentityT!

```
(>>=) :: Monad m =>
         m a
      -> (a -> m b)
      -> m b
```

```
runIdentityT . k :: (a -> m b)
```

```
runIdentityT m
```

Let's make >>= for IdentityT!

```
(>>=) :: Monad m =>
         m a
      -> (a -> m b)
      -> m b
```

```
runIdentityT . k :: (a -> m b)

=<<

runIdentityT m :: m a
```

Let’s make >>= for IdentityT!

```
(>>=) :: Monad m =>
         m a
      -> (a -> m b)
      -> m b
```

```
runIdentityT . k :: (a -> m b)

=<<

runIdentityT m :: m a



:: m b
```

Let's make >>= for IdentityT!

```
(>>=) :: Monad m =>
         m a
      -> (a -> m b)
      -> m b
```

```
IdentityT $

runIdentityT . k :: (a -> m b)

=<<

runIdentityT m :: m a


:: IdentityT m b
```

That’s great, but how do we use it?

That’s great, but how do we use it?

Glad you asked. Let’s build a basic MaybeT computation.

That's great, but how do we use it?

```
myPrompt :: String
         -> IO String
myPrompt prompt = do
  putStr prompt
  result <- getLine
  return result
```

That's great, but how do we use it?

```
getName :: MaybeT IO String
getName = do
  input <- myPrompt "Name? "
  if input == ""
    then MaybeT $ return Nothing
    else MaybeT $ return $ Just $ "Name: " ++  input ++ "\n"
```

This Does Not Typecheck!

```
getName :: MaybeT IO String
getName = do
  input <- myPrompt "Name? "
  if input == ""
    then MaybeT $ return Nothing
    else MaybeT $ return $ Just $ "Name: " ++  input ++ "\n"
```

Let’s imagine the function we want.

We want a function that has this type signature:

```
IO String -> MaybeT IO String
```

Let’s imagine the function we want.

```
IO String -> MaybeT IO String
```

Let’s imagine the function we want.

```
IO a -> MaybeT IO a
```

Let’s imagine the function we want.

```
Monad m => m a -> MaybeT m a
```

Let’s imagine the function we want.

```
(Monad m, Control.Monad.Trans.Class.MonadTrans t) =>
        m a
   -> t m a
```

SURPRISE! This is called `lift`

```
(Monad m, Control.Monad.Trans.Class.MonadTrans t) =>

        m a

   -> t m a
```

SURPRISE! This is called `lift`

```
*Main> :t lift

lift

  :: (Monad m, Control.Monad.Trans.Class.MonadTrans t) =>

   m a -> t m a
```

This typechecks now!

```
getName :: MaybeT IO String
getName = do
  input <- lift $ myPrompt "Name? "
  if input == ""
    then MaybeT $ return Nothing
    else MaybeT $ return $ Just $ "Name: " ++  input ++ "\n"
```

Let's make a few more.

```
getNumber :: String -> MaybeT IO String
getNumber str = do
  input <- lift $ myPrompt "Phone number? "
  if (length input) /= 7
    then MaybeT $ return Nothing
    else MaybeT $ return $ Just $ str ++ "Phone Number: " ++ 
input ++ "\n"
```

Let’s make a few more.

```
getStreetName :: String -> MaybeT IO String

getStreetName str = do

  input <- lift $ myPrompt "Street Name? "

  if input == ""

    then MaybeT $ return Nothing

    else MaybeT $ return $ Just $ str ++ "Street Name: " ++
input ++ "\n"
```

Let’s make a few more.

```
compositionMethod :: MaybeT IO (String)

compositionMethod = getName >>=

                    getNumber >>=

                    getStreetName
```

(Just to prove you can still do this with do-notation)

```
compAlt :: MaybeT IO (String)

compAlt = do

  a <- getName

  b <- getNumber a

  c <- getStreetName b

  return c
```

That's great, but how do you run it? It's not an IO action, it's a MaybeT IO action.

You use `runMaybeT`!

```haskell
allTogetherNow :: IO ()
allTogetherNow = do
  result <- runMaybeT compositionMethod
  case result of
    Just str -> putStrLn str
    Nothing -> putStrLn "Failure!"
```