Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 8
Monad Transformers

CS4012

Topics in Functional Programming
Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

A problem with Monads

Having Monads as a program structuring idea is pretty good.

- **We have a consistent model for building program behaviours (certain kinds of behaviours, anyway)**
- **Because it's consistent we can build utilities (e.g. "mapM") for every kind of Monad.**

However, Monads have a big disadvantage which is that they do not *compose* naturally.

Given how much the functional style emphasises composition this is a big deal.

A problem with Monads

Composing?

- **What I mean is:**
  - **Monads give us a good way to hide all the plumbing and complexity involved in joining a series of actions together.**
  - **But the set of actions is *fixed* when the Monad is designed**
  - **The State Monad does State-handling things**
  - **The Error monad (“Maybe”) does error handling things**
  - **It feels pretty natural to want to do *both* things in one computation.**

A problem with Monads

Imagine a calculator program.

We might want a state monad to manage the calculator memory, letting us write things like:

```
do
  x <- get
  y <- divide 100 x
  put y
```

A problem with Monads

But what if we want to manage errors in the calculator as well?

Using the Maybe monad we could allow divide to return Nothing, but then we have to manage "Maybe" values everywhere.

That stuff is so conveniently handled in the Maybe monad instead of State!

A problem with Monads

We could even make a new Monad, like Maybe but which carried useful information about what had gone wrong!

```
data MonadOfExceptions e v = Error e | Success v
```

With all the usual monadic stuff (return, >>=), and maybe a function called handleError that acts as an exception handler

```
handleError :: (MonadOfExceptions e v) ->
               (MonadOfExceptions e v) ->
               (MonadOfExceptions e v)
```

A problem with Monads

But it’s not at all obvious how to combine the two monads in one calculation.

```
do
  x <- get
  y <- ( divide 100 x ) `handleError` ( return 0 )
  put y
```

There is a big problem with the types in the second action.

A problem with Monads

Instead we will have to make a new Monad that combines both state management and partiality.

It would be nice if we could just ask for the existing State and Maybe monads to be combined automatically to make:

```
newtype SM s a = SM (s -> Maybe (a, s))
```

Or do I mean:

```
newtype SM s a = SM (s -> (Maybe a, s))
```

Not obvious which is “right” (maybe they both are?)

A problem with Monads

Composition

Functors and Applicatives *do* compose naturally, so we can see what it would look like to do this automatically.

First, what do I mean by "compose"? I don't mean composing two functions (say "(fmap f) . (fmap g)"). I mean composing the *types*.

```
data Compose f g a = Compose (f (g a))
```

This "Compose" type captures the notion of two types being combined. Here are the two ways to compose List and Maybe, for example:

```
data ListMaybe a = ListMaybe (Compose [] Maybe a)
data MaybeList a = MaybeList (Compose Maybe [] a)
```

A problem with Monads

Composition

If we want one of our new composed type to be a Functor then all we need to do is write “fmap”.

```
data Compose f g a = Compose (f (g a))
```

```
data ListMaybe a = ListMaybe (Compose [] Maybe a)
```

```
fmapLM :: (a -> b) -> ListMaybe a -> ListMaybe b
fmapLM f (ListMaybe l) = ListMaybe (fmapC f l)

fmapC :: (a -> b) -> Compose f g a -> Compose f g b
fmapC f (Compose x) = Compose (fmap (fmap f) x)
```

A problem with Monads

Composition

If we want one of our new composed type to be a Functor then all we need to do is write "fmap".

```
data Compose f g a = Compose (f (g a))
```

```
data MaybeList a = MaybeList (Compose Maybe [] a)
```

```
fmapLM :: (a -> b) -> MaybeList a -> MaybeList b
fmapLM f (MaybeList l) = MaybeList (fmapC f c)

fmapC :: (a -> b) -> Compose f g a -> Compose f g b
fmapC f (Compose x) = Compose (fmap (fmap f) x)
```

A problem with Monads

Composition

In fact, this is *always* the composition of two Functors!

```
instance (Functor f, Functor g)
    => Functor (Compose f g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)
```

The composition of two Functors makes a new Functor

A problem with Monads

Composition

Applicatives compose this way as well:

```
instance (Applicative f, Applicative g)
    => Applicative (Compose f g) where

  pure x = Compose (pure (pure x))

  Compose fgf <*> Compose fgx =
    Compose (pure (<*>) <*> fgf <*> fgx)
```

A problem with Monads

Composition

Monads do *not* compose in this way.

```
instance (Monad f, Monad g)
    => Monad (Compose f g) where
 return x = Compose (return (return x)

 (Compose xs) >>= f = ????
```

A problem with Monads

Composition

That is, can we write a function with this type (using only the Applicative Functor and Monad laws):

```
bindC :: (Monad f, Monad g) =>
         Compose f g a -> (a -> Compose f g b) -> Compose f g b
```

The answer is no, you cannot.

No matter how you try, you will get yourself in a knot. You can get most of the way there if you can assume you have a distributive operation:

```
distribute :: (Monad f, Monad g) => g (f a) -> f (g a)
```

But not every pair of Monads *have* such an operation that satisfies the monad laws; this is where you'll get stuck.
We can do it for *specific* cases (except when we can't!), but we can't do it in general.

In summary

Monads can be handy for capturing ideas like state, partiality, concurrency.

What happens if we want to combine those ideas in one computation?

Not obvious how to do it automatically, we might have to write the entire Monad from scratch.

Combining features from more than two monad sounds absolutely miserable!

A different approach is in order!

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 1

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Part 2
Monad Transformers

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Monad Transformers

It’s true that we cannot compose monads arbitrarily.

But there is a clever solution that allows us to construct almost any combination of monads we require!

A “Monad Transformer” is a type which can add effects to a monad by producing a new monad.

This isn’t Monad Composition, because the “transformer” is not actually a monad.

Monad Transformers

For example, let's say we have a State monad, and we want to add some way to manage errors. We have already seen that we cannot just take some "error" monad and magically compose them.

Instead, we define a type "MaybeT" which has an extra type parameter.

```
data maybeT m a = MaybeT m (Maybe a)
```

That first type parameter is going to be the monad we "transform" by adding Maybe-style partiality.

Monad Transformers

To be clear, "MaybeT" is not a monad.

But this is:

```
type MaybeState s = MaybeT (State s)
```

The final type of our monad is whatever "MaybeT" is, specialised to.

Just for completeness, the "real" definition of MaybeT is:

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

Monad Transformers

We can define suitable instances of maybeT to make sure it's a Functor, so long as the thing it's transforming is also a functor

```
instance (Functor m) => Functor (MaybeT m) where
    fmap f = MaybeT . fmap (fmap f) . runMaybeT
```

Remember the Functor instance for Compose with Maybe that we produced?

Monad Transformers

We can do the same thing for Applicative

```
instance (Applicative m) => Applicative (MaybeT m) where
  pure     = MaybeT . pure . pure

  (MaybeT ma) <*> (MaybeT mb) = MaybeT $ liftA2 (<*>) ma mb
```

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2 f x = (<*>) (fmap f x)
```

Again, this shouldn't surprise anyone, because we know that Applicatives can be composed in general

Monad Transformers

Now the clever bit. Because we have “fixed” one monad to be Maybe we can write an instance of Monad which captures the approach to distribution that is needed.

```
instance (Monad m) => Monad (MaybeT m) where

  a >>= f = MaybeT $ do
    a' <- runMaybeT a
    case a' of
      Nothing  -> return Nothing
      Just a'' -> runMaybeT $ f a''
```

Presto!

Monad Transformers

There's one detail we have to handle. A function like this still has some typing issues:

```
do
  x <- get
  y <- ( divide 100 x ) `handleError` ( return 0 )
  put y
```

Assuming this is an operation in the "MaybeState" monad, which is really the "MaybeT" type with state as one parameter, then those uses of "get" and "put are badly typed..

Monad Transformers

The “State” value is “stuck” inside the “MaybeT”. What we need is an operation that takes some operation in the State monad and “promotes” it to the “MaybeT” type.

```
lift :: (Monad m) => m a -> MaybeT m a

lift = MaybeT . fmap Just
```

```
do
  x <- lift get
  y <- ( divide 100 x ) `handleError` ( return 0 )
  lift $ put y
```

The ‘mtl’ library

The standard collection of Haskell monads are implemented in terms of transformers in the Control.Monad.Trans library. For example:

- MaybeT - add partiality to a computation
- ExceptT - add failure (with exception values) to a computation
- WriterT - Add writing to a stream of data (logging or assembling values) to a computation
- ReaderT - add reading an environment to a computation
- StateT - add reading and writing an environment

The ‘mtl’ library

You can get a “simple” monad from one of the transformers by using it to transform the Identity monad:

```
type State s = StateT (Identity s)
```

The identity monad is a monad that does not provide any computation, it simply applies the bound function without change. There’s really no reason to use it on it’s own.

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 2

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/