Monad Transformers

Brisbane Functional Programming Group, June 2017

Tony Morris

The Plan

The outline of the journey[1]:

- Remind ourselves:
  - What is a functor?
  - What is a monad?
- What is a monad transformer?
- Why might I use a monad transformer?

[1] We will be approximating at times to make the point

Functor

A functor F is given by the function:

```
(a -> b) -> (F a -> F b)
```

lifts a unary function into an environment F

Functor Haskell

Functor using Haskell syntax

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Functor Examples

Examples of Functor instances

- The list functor maps a function on head of each cons cell:

```
(a -> b) -> ([] a -> [] b)
(a -> b) -> ([a] -> [b])
```

- The maybe functor maps a function on Just:

```
(a -> b) -> (Maybe a -> Maybe b)
```

Functor Examples in Haskell

Examples of Functor instances using Haskell syntax

```
instance Functor [] where
  fmap f =
    foldr ((:) . f) []

instance Functor Maybe where
  fmap f =
    maybe Nothing (Just . f)

instance Functor ((->) t) where
  fmap f g =
    \x -> f (g x)
```

Monad

A monad F is given by the function:

```
(a -> F b) -> (F a -> F b)
```

binds a function through an environment F

Functor vs Monad

Functor vs Monad

```
(a ->   b) -> (F a -> F b)
(a -> F b) -> (F a -> F b)
```

Monad Haskell

Monad using Haskell syntax

```
class Monad f where
  bind :: (a -> f b) -> (f a -> f b)
```

Monad Examples

Examples of Monad instances

- The list monad takes the cartesian product:

```
(a -> [] b) -> ([] a -> [] b)
(a -> [b]) -> ([a] -> [b])
```

- The maybe monad threads the possible Just value:

```
(a -> Maybe b) -> (Maybe a -> Maybe b)
```

Monad Examples in Haskell

Examples of Monad instances using Haskell syntax

```
instance Monad [] where
  bind f =
    foldr ((++) . f) []

instance Monad Maybe where
  bind f =
    maybe Nothing f

instance Monad ((->) t) where
  bind f g =
    \x -> f (g x) x
```

Combining Functors

Mapping on a [] of Maybe

Suppose I glue two functors, `[]` and `Maybe`, together:

```
value :: [Maybe a]
```

and I want to map a function `f :: a -> b`:

```
result :: [Maybe b]
result = fmap (fmap f) value
```

Functor Composition

Mapping on a (f of g)

```
\f -> fmap (fmap f) ::
  (Functor f, Functor g) =>
  (a -> b) -> f (g a) -> f (g b)
```

Functor Composition

Mapping on a (f of g)

In fact, I can do this for any two functors:

```
value :: f (g a)
```

to map a function `f :: a -> b`:

```
result :: f (g b)
result = fmap (fmap f) value
```

Functor Composition

In other words

If f and g are functors, then (f of g) is a functor:

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

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

Functors Compose

The composition of two arbitrary functors makes a new functor.
In brief, we say that *functors compose*.

Combining Monads

Binding on a [] of Maybe

Suppose I glue two monads, `[]` and `Maybe`, together:

```
value :: [Maybe a]
```

and I want to bind a function `f :: a -> [Maybe b]`:

```
result :: [Maybe b]
result = bind (maybe (unit Nothing) f) value
```

Combining Two Monads

Binding on a [] of Maybe

We called bind on a list

```
result = bind (maybe (unit Nothing) f) value
         ^        ^
```

but we destructured the Maybe using Maybe-specific calls.

Composing Monads

If f and g are monads, then is (f of g) a monad?

Can we generalise?

```
instance (Monad f, Monad g) =>
    Monad (Compose f g) where
  bind =
    error "???"
```

Composing Monads

In other words . . .

Can we write a function with this type?

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

Composing Monads

No. Try it.

There are several ways to tie your knickers in a knot, but they will always be tangled.

Composing Monads

However, we can bind on (f of Maybe) for any monad f.

```
result ::
  Monad f =>
  (a -> f (Maybe b))
  -> f (Maybe a)
  -> f (Maybe b)
result =
  bind (maybe (unit Nothing) f) value
```

Maybe Monad Transformer

And so the Maybe monad transformer comes to be.

```
data MaybeT f a = MaybeT {
  maybeT :: f (Maybe a)
}

instance Monad f => Monad (MaybeT f) where
  bind f (MaybeT x) =
    MaybeT
      (bind
        (maybe (unit Nothing)
        (maybeT . f)) x
      )
```

Maybe Monad Transformer

The Maybe monad transformer

provides the construction of the monad for (f of Maybe) for an arbitrary monad f. Its behaviour combines the individual monads of Maybe then f, in that order.

This transformer exists

precisely because *Monads do not compose in general*.

Maybe Monad Transformer

Example [] on Maybe

```
m1 :: MaybeT [] Integer
m1 = MaybeT [Just 1, Just 2, Just 30]

f1 :: Integer -> MaybeT [] Integer
f1 n =
  MaybeT
    [
      Just n
    , if n < 10 then Just (n * 50) else Nothing
    ]
```

```
> maybeT (bind f1 m1)
[Just 1,Just 50,Just 2,Just 100,Just 30,Nothing]
```

Maybe Monad Transformer

Example ((->) t) on Maybe

```
m2 :: MaybeT ((->) Integer) String
m2 = MaybeT (\x ->
       if even x
         then Just (show (x * 10))
         else Nothing)

f2 :: String -> MaybeT ((->) Integer) String
f2 s = MaybeT (\n ->
         if n < 100
           then Just (show n ++ s)
           else Nothing)
```

```
> map (maybeT (bind f2 m2)) [3, 4, 700]
[Nothing,Just "440",Nothing]
```

More Monad Transformers

- `MaybeT f a = f (Maybe a)`
- `EitherT f a b = f (Either a b)`
- `ReaderT f a b = a -> f b`
- `StateT f s a = a -> f (a, s)`

Each Monad Transformer exists

because *monads do not compose in general*

Transformers

Functor Transformers do not exist

because *functors compose* so what's the point?

Transformers

Actually, you'll find all these things compose:

- Functor
- Apply
- Applicative
- Alt
- Alternative
- Foldable
- Foldable1
- Traversable
- Traversable1

Transformers

These do not compose:

- Monad
- Bind
- Comonad
- Cobind

it's a useful exercise to try it anyway!

Questions

?