More on monads

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP Dr. Lars Brünjes, IOHK 2018-02-05



Goals

- · Revisit monads
- · A few new monads
- Combining monads

Monads

The class hierarchy

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

```
class Functor f => Applicative f where
pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Laws

Functor:

```
fmap id x = x
fmap (f . g) x = (fmap f . fmap g) x
```

Laws

Functor:

```
fmap id x = x
fmap (f.g) x = (fmap f. fmap g) x
```

Applicative:

Laws (contd.)

Monad:

```
return x >>= f = f x
a >>= return = a
(a >>= f) >>= g = a >>= (\x -> f x >>= g)
```

Instances

```
instance Monad Maybe
instance Monad []
instance Monad (Either e)
instance Monad (State s)
instance Monad IO
instance Monad STM
instance Monad Gen
instance Monad (Parser t)
```

Extended interfaces

```
E.g. for State :
get :: State s s
put :: s -> State s ()
```

Extended interfaces

```
E.g. for State:
get :: State s s
put :: s -> State s ()
Or for Either:
throwError :: e -> Either e a
catchError ::
  Either e a -> (e -> Either e a) -> Either e a
```

A few new monads

Identity

Defined in Data.Functor.Identity:

```
newtype Identity a = Identity {runIdentity :: a}
```

An interesting special case.

Identity

Defined in **Data.Functor.Identity**:

```
newtype Identity a = Identity {runIdentity :: a}
```

An interesting special case.

```
instance Monad Identity where
  return = Identity
  (Identity x) >>= f = f x
```

Will become useful later today.

Reader

Defined in Control. Monad. Trans. Reader:

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

For distributing (read-only) state.

¹in a slightly different way, see later

Defined in Control. Monad. Trans. Reader:

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

For distributing (read-only) state.

```
instance Monad (Reader r) where
  return x = Reader (\ _ -> x)
  m >>= g =
    Reader (\ r -> runReader (g (runReader m r)) r)
```

¹in a slightly different way, see later

Extended interface for reader

Accessing the state:

```
ask :: Reader r r ask = Reader (\ r \rightarrow r)
```

Extended interface for reader

Accessing the state:

```
ask :: Reader r r
ask = Reader (\ r -> r)
```

Locally modifying the state:

```
local :: (r \rightarrow r) \rightarrow Reader \ r \ a \rightarrow Reader \ r \ a local f m = Reader (\ r \rightarrow runReader \ m \ (f \ r))
```

A simpler version of reader

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

```
Reader r a \approx r -> a Reader r \approx (->) r
```

A simpler version of reader

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

```
Reader r a \approx r -> a Reader r \approx (->) r
```

```
instance Monad ((->) r) where
return x = \ _ -> x
f >>= g = \ r -> g (f r) r
```

A simpler version of reader

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

```
Reader r a \approx r -> a Reader r \approx (->) r
```

```
instance Monad ((->) r) where
  return x = \ _ -> x
  f >>= g = \ r -> g (f r) r
```

Can we overload ask and local?

An interface for readers

We want ask and local to have at least the following types:

```
ask :: Reader r r
local :: (r -> r) -> Reader r a -> Reader r a
```

```
ask :: r \rightarrow r
local :: (r \rightarrow r) \rightarrow (r \rightarrow a) \rightarrow (r \rightarrow a)
```

First attempt

```
class Monad m => MonadReader m where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```

This is insufficient. Can you see why?

First attempt

```
class Monad m => MonadReader m where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```

This is insufficient. Can you see why?

```
instance MonadReader ((->) r) where  ask = \ r \rightarrow r \qquad -- type \ error   local = \ f \ m \ r \rightarrow m \ (f \ r) \ -- type \ error
```

The type variable \mathbf{r} in the class is different from the \mathbf{r} in the instance.

First attempt

```
class Monad m => MonadReader m where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```

This is insufficient. Can you see why?

The type variable \mathbf{r} in the class is different from the \mathbf{r}' in the instance.

```
class MonadReader (m :: * -> * -> *) where
  ask :: m r r
  local :: (r -> r) -> m r a -> m r a
```

```
instance MonadReader (->) where
ask = \ r -> r
local = \ f m r -> m (f r)
```

```
class MonadReader (m :: * -> * -> *) where
   ask :: m r r
   local :: (r -> r) -> m r a -> m r a
```

```
instance MonadReader (->) where
ask = \ r -> r
local = \ f m r -> m (f r)
```

```
instance MonadReader Reader where
  ask = Reader (\r -> r)
  local f m = Reader (\r -> runReader m (f r))
```

```
class MonadReader (m :: * -> * -> *) where
  ask :: m r r
  local :: (r -> r) -> m r a -> m r a
```

```
instance MonadReader (->) where
ask = \ r -> r
local = \ f m r -> m (f r)
```

```
instance MonadReader Reader where
ask = Reader (\r -> r)
local f m = Reader (\r -> runReader m (f r))
```

We can no longer easily impose a superclass constraint.

Solution 2: multi-parameter type classes

```
class Monad m => MonadReader r m where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```

Solution 2: multi-parameter type classes

```
class Monad m => MonadReader r m where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```

```
instance MonadReader r ((->) r) where
ask = \ r \rightarrow r
local = \ f m r \rightarrow m (f r)
```

Solution 2: multi-parameter type classes

```
class Monad m => MonadReader r m where
ask :: m r
local :: (r -> r) -> m a -> m a
```

```
instance MonadReader r ((->) r) where ask = \ r \rightarrow r local = \ f m r \rightarrow m (f r)
```

```
instance MonadReader r (Reader r) where
ask = Reader (\r -> r)
local f m = Reader (\r -> runReader m (f r))
```

Language extensions

This solution requires the MultiParamTypeClasses extensions.

In general, code that uses multi-parameter type classes will also require the FlexibleInstances and FlexibleContexts extensions.

Solution 2 is somewhat more flexible

```
type Env = Map String Int
newtype EnvErr a =
  EnvErr {runEnvErr :: Env -> Either String a}
```

Solution 2 is somewhat more flexible

```
type Env = Map String Int
newtype EnvErr a =
   EnvErr {runEnvErr :: Env -> Either String a}

instance Monad EnvErr where
  return x = EnvErr $ \ env -> return x
  EnvErr f >>= g = EnvErr $ \ env ->
    f env >>= \ a -> runEnvErr (g a) env
```

Solution 2 is somewhat more flexible

```
type Env = Map String Int
newtype EnvErr a =
   EnvErr {runEnvErr :: Env -> Either String a}
```

```
instance Monad EnvErr where
  return x = EnvErr $ \ env -> return x
  EnvErr f >>= g = EnvErr $ \ env ->
    f env >>= \ a -> runEnvErr (g a) env
```

```
instance MonadReader Env EnvErr where
  ask = EnvErr $ \ env -> return env
  local f m = EnvErr $ \ env -> runEnvErr m (f env)
```

Solution 2 also creates new problems

Let's try to define the following abstraction:

```
withDouble m = local (* 2) m -- type error
```

Solution 2 also creates new problems

Let's try to define the following abstraction:

```
withDouble m = local (* 2) m -- type error
```

Yields a type error similar to this:

```
Could not deduce (Num r0)

from the context: (Num r, MonadReader r m)

bound by the inferred type for 'withDouble':

(Num r, MonadReader r m) => m a -> m a

The type variable 'r0' is ambiguous
```

Recall: "ambiguous" type variable usually mean there's not sufficient contextual info to resolve the overloading.

Ambiguity

We can annotate the type of the state locally:

```
withDouble m = local (* (2 :: Int)) m
```

Ambiguity

We can annotate the type of the state locally:

```
withDouble m = local (* (2 :: Int)) m
```

But we cannot just provide a type signature:

```
withDouble :: MonadReader m Int => m a -> m a
withDouble m = local (* 2) m -- type error
```

Ambiguity

We can annotate the type of the state locally:

```
withDouble m = local (* (2 :: Int)) m
```

But we cannot just provide a type signature:

```
withDouble :: MonadReader m Int => m a -> m a
withDouble m = local (* 2) m -- type error
```

And we cannot easily leave the state type polymorphic:

```
withDouble :: forall r m a .
   (MonadReader r m, Num r) => m a -> m a
withDouble m = local (* (2 :: r)) m -- type error
This causes an error despite scoped type variables.
```

Ambiguous types

```
withDouble :: forall r m a .
  (MonadReader r m, Num r) => m a -> m a
withDouble m = local (* (2 :: r)) m -- type error
```

Our example is much like

```
readShow ::
   forall a . (Show a, Read a) => String -> String
readShow x = show (read x :: a) -- type error
```

Both have ambiguous types. There are type variables appearing in the class context that do not occur in the type itself. There is no way for type inference to ever resolve the constraint from contextual information.

Is there true ambiguity?

```
withDouble :: forall r m a .
  (MonadReader r m, Num r) => m a -> m a
withDouble m = local (* (2 :: r)) m -- type error
```

In our case, there should not really be any ambiguity, because the type ${\bf r}$ should be computable from ${\bf m}$.

Consider m to be one of our MonadReader examples so far:

```
Reader r -- state type must be r
(->) r -- state type must be r
EnvErr -- state type must be Env
```

In all three cases, the state type is determined by **m**. But GHC does not know that.

Solution 3a: use a type family

Requires the **TypeFamilies** extension:

```
type family EnvType (m :: * -> *) :: *
class Monad m => MonadReader m where
  ask :: m (EnvType m)
  local :: (EnvType m -> EnvType m) -> m a -> m a
```

A type family is an open, type-level function.

The type family **EnvType** maps types of kind * -> * to types of kind *.

Whenever we provide an instance declaration for **MonadReader**, we can (and must) extend the type family as well.

Solution 3a (contd.)

```
type instance EnvType (Reader r) = r
instance MonadReader (Reader r) where
ask = Reader (\r -> r)
local f m = Reader (\r -> runReader m (f r))
```

Solution 3a (contd.)

```
type instance EnvType (Reader r) = r
instance MonadReader (Reader r) where
ask = Reader (\r -> r)
local f m = Reader (\r -> runReader m (f r))
```

```
type instance EnvType EnvErr = Env
instance MonadReader EnvErr where
  ask = EnvErr $ \ env -> Right env
  local f m = EnvErr $ \ env -> runEnvErr m (f env)
```

Solution 3a (contd.)

The problem with withDouble disappears now:

```
withDouble ::
   (MonadReader m, Num (EnvType m)) => m a -> m a
withDouble m = local (* 2) m
```

The type no longer mentions r , and $EnvType\ m$ is clearly computable from m .

Solution 3b: use a functional dependency

Requires the Functional Dependencies extension:

```
class Monad m => MonadReader r m | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a
```

A functional dependency restricts the instances we can define for the class. For example, declaring

```
instance MonadReader Env EnvErr
instance MonadReader Int EnvErr
```

would be rejected as violating the functional dependency.

In turn, GHC now treats \mathbf{r} as determined by \mathbf{m} .

Solution 3b (contd.)

Again, the problem with withDouble disappears:

```
withDouble ::
   (MonadReader r m, Num r) => m a -> m a
withDouble m = local (* 2) m
```

While the type looks as before, ${\bf r}$ is now treated as determined by ${\bf m}$, so the type is no longer considered to be ambiguous.

Comparison

- · Both solutions are viable.
- Both functional dependencies and type families are widely applicable for a large number of programming problems that go far beyond this particular scenario.
- While in general, type families are often preferred over functional dependencies (their implementation in GHC is somewhat more principled), historically, for the use in the MonadReader class and similar classes, functional dependencies still dominate.
- Both implementations exist on Hackage: mtl implements the version with functional dependencies, and e.g.
 monads-tf implements a version with type families.

Other similar interfaces

```
class Monad m => MonadError e m | m -> e where
 throwError :: e -> m a
 catchError :: m a -> (e -> m a) -> m a
class Monad m => MonadState s m | m -> s where
 get :: m s
 put :: s -> m ()
class Monad m => MonadIO m where
 liftIO :: IO a -> m a
```

Combining monads

Combined monads

```
Reader Env and Either String:
newtype EnvErr a =
EnvErr {runEnvErr :: Env -> Either String a}
```

Combined monads

ErrorState

Reader Env and Either String:

```
newtype EnvErr a =
   EnvErr {runEnvErr :: Env -> Either String a}

State s and Either String (from the exercises):
newtype ErrorState s a =
```

{runErrorState :: s -> Either String (a, s)}

Combined monads

```
Reader Env and Either String:
newtype EnvErr a =
 EnvErr {runEnvErr :: Env -> Either String a}
State s and Either String (from the exercises):
newtype ErrorState s a =
 ErrorState
   {runErrorState :: s -> Either String (a, s)}
State [t] and [] (from the lecture on parsers):
newtype Parser t a =
 MkParser {runParser :: [t] -> [(a, [t])]}
```

Combining systematically

Rather than defining all these combinations by hand, can we just combine the individual monads into larger ones?

Combining systematically

Rather than defining all these combinations by hand, can we just combine the individual monads into larger ones?

Unfortunately, the answer is not a simple yes.

Excursion: composition of functors

Defined in **Data.Functor.Compose**:

```
newtype Compose f g a =
Compose {getCompose :: f (g a)}
```

Excursion: composition of functors

Defined in Data.Functor.Compose:

```
newtype Compose f g a =
   Compose {getCompose :: f (g a)}

If f and g are functors, then Compose f g is a functor:

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

Examples

Composition of applicative functors

Applicative functors are closed under composition as well:

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

Compose f <*> Compose x =
    Compose (pure (<*>) <*> f <*> x)
```

Example

```
GHCi> getCompose
  pure (+)
  <*> Compose [Just 1, Just 10]
  <*> Compose [Nothing, Just 100]
[Nothing, Just 101, Nothing, Just 110]
```

And for monads?

Unfortunately, we cannot make $\begin{array}{c} \textbf{Compose} \\ \textbf{Monad} \end{array}$ an instance of

And for monads?

Unfortunately, we cannot make **Compose** an instance of **Monad**.

The story for monads is more complicated and less general:

- · For some monads, we can define monad transformers.
- A monad transformer takes an existing monad to a new monad.
- The order in which we apply monad transformers matters.
- · Not all monads have associated transformers.

Monad transformers

The reader transformer

Defined in **Control.Monad.Trans.Reader** in package transformers:

```
newtype ReaderT r m a =
ReaderT {runReaderT :: r -> m a}
```

The **ReaderT** type is parameterized over the "underlying" monad – compare to the "old" type:

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

Note that the kind of **ReaderT** is

```
* -> (* -> *) -> (* -> *)
```

A monad transformer

The ReaderT type indeed takes monads to monads:

```
instance Monad m => Monad (ReaderT r m) where
  return x = ReaderT (\ _ -> return x)
  m >>= g =
    ReaderT $ \ r ->
    runReaderT m r >>= \ a -> runReaderT (g a) r
```

Compare to the old **Reader** instance:

```
instance Monad (Reader r) where
  return x = Reader (\ _ -> x)
  m >>= g =
    Reader (\ r -> runReader (g (runReader m r)) r)
```

Functors and applicatives

We still need to handle the superclasses – we do not use the "standard" instances here because we can do with weaker constraints on \mathbf{m} :

```
instance Functor m => Functor (ReaderT r m) where
fmap f m =
   ReaderT (\ r -> fmap f (runReaderT m r))
```

```
instance Applicative m
    => Applicative (ReaderT r m) where
    pure x = ReaderT (\ _ -> pure x)
    f <*> x =
        ReaderT $ \ r ->
        runReaderT f r <*> runReaderT x r
```

We obtain a reader

```
instance Monad m
  => MonadReader r (ReaderT r m) where
  ask = ReaderT (\ r -> return r)
  local f m =
    ReaderT $\ env -> runReaderT m (f env)
```

(More or less the same instance we used for EnvErr.)

Other transformers

```
newtype StateT s m a =
   StateT {runStateT :: s -> m (a, s)}

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

newtype ExceptT e m a =
   ExceptT {runExceptT :: m (Either e a)}
```

Identity as a base case

```
StateT s Identity \approx State s ReaderT r Identity \approx Reader r MaybeT Identity \approx Maybe ExceptT e Identity \approx Either e
```

Identity as a base case

```
StateT s Identity \approx State s

ReaderT r Identity \approx Reader r

MaybeT Identity \approx Maybe

ExceptT e Identity \approx Either e
```

There is also an **IdentityT**, but it's rarely useful, as it adds no extra functionality.

Building combinations

```
newtype EnvErr a =
   EnvErr {runEnvErr :: Env -> Either String a}
```

Building combinations

```
newtype EnvErr a =
EnvErr {runEnvErr :: Env -> Either String a}
```

```
type EnvErr =
  ReaderT Env (ExceptT String Identity)
```

```
newtype ErrorState s a =
ErrorState
{runErrorState :: s -> Either String (a, s)}
```

```
newtype ErrorState s a =
ErrorState
{runErrorState :: s -> Either String (a, s)}
```

```
type ErrorState s =
  StateT s (ExceptT String Identity)
```

```
newtype Parser t a =
MkParser {runParser :: [t] -> [(a, [t])]}
```

```
newtype Parser t a =
MkParser {runParser :: [t] -> [(a, [t])]}
```

```
type Parser t =
  StateT [t] []
```

```
newtype Parser t a =
MkParser {runParser :: [t] -> [(a, [t])]}
```

```
type Parser t =
  StateT [t] []
```

Note: There is a ListT in the libraries, but it should be used with extreme care, as it often does not produce valid monads.

IO as a base case

We can also build monads on top of IO:

```
type Example =
  ReaderT Env (ExceptT Message IO)
```

IO as a base case

We can also build monads on top of IO:

```
type Example =
  ReaderT Env (ExceptT Message IO)
```

There is no transformer version of **IO**.

Are we done?

Not quite – consider again:

```
type Example =
  ReaderT Env (ExceptT Message IO)
```

- Example is an instance of MonadReader Env.
- Similarly, ExceptT Message IO is an instance of MonadError Message.
- But shouldn't Example also be an instance of MonadError Message ?

Lifting MonadError through ReaderT

This instance requires the **UndecidableInstances** extension (GHC is *extremely* conservative without it in trying to guarantee the termination of instance search; the extension is more harmless than it sounds.)

Unfortunately, we need lifting instances for *all combinations* of monad transformers:

```
instance MonadState s m
  => MonadState s (ReaderT r m) where
  get = ReaderT $ return get
  put s = ReaderT $ return (put s)
```

This makes monad transformers less modular and less extensible than would be desirable.

Lifting generically

Fortunately, most liftings are very simple:

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

Lifting generically

Fortunately, most liftings are very simple:

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

Example instances:

```
instance MonadTrans (ReaderT r) where
lift m = ReaderT $ return m
```

```
instance MonadTrans (StateT s) where
lift m = StateT $ \ s -> fmap (\ a -> (a, s)) m
```

```
instance MonadTrans (ExceptT e) where
lift m = ExceptT $ fmap return m
```

Monad transformer laws

```
lift (return x) = return x
lift (m >>= f) = lift m >>= \ a -> lift (f a)
```

Lifting generically (contd.)

Some lifting instances get near trivial.

```
instance MonadState s m
  => MonadState s (ReaderT r m) where
  get = lift get
  put s = lift (put s)
```

Others (e.g. the catchError definition) are less straight-forward.

Monad transformers summary

- Monad transformers provide a way to build monads out of smaller building blocks and to combine effects as needed.
- Unfortunately, the underlying theory is not extremely beautiful, leading to several restrictions and complications.
- A few approaches have been tried to address some of the shortcomings, but mtl remains dominant.