composing Transformers

There exists a certain kind of monad transformers. I'm talking about all transformers that have lawful instances of `MonadTransControl` from the monad-control library. In particular `ReaderT`, `StateT` and `ExceptT` have lawful instances. Let's stick to `ReaderT` for now.

```
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
  deriving stock (Functor, Applicative, Monad)

instance MonadTrans ReaderT where
  lift = ReaderT . const

instance MonadTransControl ReaderT where
  type StT (ReaderT r) a = a
  liftWith f = ReaderT $ \r -> f $ \t -> runReaderT t r
  restoreT = ReaderT . const
```

Why is `ReaderT` of interest?

`ReaderT` in conjunction with an `IO`-based Monad `m` can be used to build stateful applications, when using something like stm's `TVar`s. `ReaderT` is special compared to other transformers, because it doesn't carry any monadic state `StT (ReaderT r) = a`, which is an associated type of the `MonadTransControl` class. This is also the reason, why it has an instance of `MonadUnliftIO`.

> **NOTE** `MonadBaseControl IO m` should be a superclass of `MonadUnliftIO m`.

So in the case of `ReaderT`, `withRunInIO` (from `MonadUnliftIO`) is just a special case of `liftBase` (from `MonadBaseControl`). This is a powerful function, which basically allows us to not only `lift` monadic values, but also provide monadic arguments. An interesting example of this is the `Application` type alias from wai.

```
type Application = Request -> (Response -> IO ResponseReceived) -> IO ResponseReceived
```

Instead of having `IO` directly in the `Application` type, you can use an arbitrary Monad `m` with the constraint `MonadUnliftIO m`. I did exactly this in a library called wai-control.

```
type ApplicationT m = Request -> (Response -> m ResponseReceived) -> m
ResponseReceived
```

Building an application

Now let's build an application using a `Configuration`, the ability to do logging and a database connection.

> **NOTE** You should be able to cover most of the required language extensions with `GHC2021`. You will also need `QuantifiedConstraints` and `UndecidableInstances`.

`Configuration` is just a simple record.

```
data Configuration = Configuration
  { databaseUrl :: T.Text
  , databasePassword :: T.Text
  , port :: Word8
  }
```

For logging the monad-logger library is used.

```
newtype LoggingT m a = LoggingT { runLoggingT :: (Loc -> LogSource -> LogLevel ->
LogStr -> IO ()) -> m a }
  deriving (Functor, Applicative, Monad)

-- and additional instances ...
```

When inspecting the type of `LoggingT`, one can see, that this is just a reader with access to the logging function.

For demonstration purposes let's add another transformer to our environment, that allows us to talk to a database. Some implementation details are left to your imagination.

```
newtype DatabaseT m a = DatabaseT { runDatabaseT :: ReaderT (TVar DatabaseConnection)
m a }
  deriving (Functor, Applicative, Monad)

-- and additional instances ...
```

And now we just need a type class to use as a constraint, when later on actually building the application logic.

```
class Monad m => MonadDatabase m where
  queryDatabase :: DatabaseQuery -> m DatabaseResponse
```

```
instance (MonadIO m) => MonadDatabase (DatabaseT m) where
  queryDatabase = undefined -- TODO: Implementation.
```

Naive AppT implementation

To hide away implementation details, it might be of interest to wrap your environment with a newtype AppT. We will also add MonadTrans and MonadTransControl instances, that have to be defined manually, but can help when implementing other instances like MonadIO or mtl type classes.

```
newtype AppT m a = AppT { unAppT :: DatabaseT (ReaderT Configuration (LoggingT m)) a }
  deriving newtype (Functor, Applicative, Monad)

instance MonadTrans AppT where
  lift = AppT . lift . lift . lift

instance MonadTransControl AppT where
  type StT AppT a = StT LoggingT (StT (ReaderT Configuration) (StT DatabaseT a))
  liftWith f = AppT $ liftWith $ \ run ->
                        liftWith $ \ run' ->
                          liftWith $ \ run'' ->
                            f $ run'' . run' . run . unAppT
  restoreT = AppT . restoreT . restoreT . restoreT
```

To use this you will also need a “runner” function. This function will initialize the environment for your application.

```
runAppT :: MonadIO m => Configuration -> AppT m a -> m a
runAppT config app = runStdOutLogging $ runReaderT (runDatabaseT config (unAppT app))
config
```

If you look closely, you might say that this is not all of the initialization. We still need to read the Configuration and connecting to the database was left to runDatabaseT. If you have a complicated environment you might be interested in the next approach, which addresses this non-trivial part of initialization.

Improving initialization

We can use our transformer stack to guide the initialization. The inner transformers should be initialized before the outer ones. In our example that would mean:

1. LoggingT
2. ReaderT Configuration
3. DatabaseT

This even allows us to use parts of the environment `AppT`, that were already set up beforehand. In the following example we use the logger while reading in the configuration and connecting to the database.

```
runApplication :: MonadIO m => AppT m a -> m a
runApplication app = do
  runStdOutLogging $ do -- run `LoggingT`
    config <- liftIO acquireConfiguration
    logInfoN $ "Acquired configuration: " <> T.pack (show config) -- we can log now
    (\ tma -> runReaderT tma config) $ do -- run `ReaderT Configuration`
      runDatabaseT config $ do -- run `DatabaseT`
        lift . lift $ logInfoN $ "Connected to database." -- we can also log here
        unAppT app

acquireConfiguration :: IO Configuration
acquireConfiguration = undefined -- TODO: Implementation.
```

If you just want something functional and are a proponent of simple Haskell you can stop here. This is already looking pretty good, but we can do even better.

Let me show you, where this can be improved.

`AppT`'s instances

To use the environment you will have to provide a few instances.

```
instance (MonadIO m) => MonadLogger (AppT m) where
  monadLoggerLog loc src level msg = AppT . lift . lift $ monadLoggerLog loc src level
msg

instance (Monad m) => MonadReader Configuration (AppT m) where
  ask = AppT $ lift ask
  local f ma = AppT $ liftWith $ \ run -> local f $ run $ unAppT ma

instance (Monad m) => MonadDatabase (AppT m) where
  queryDatabase = AppT . queryDatabase
```

This is quite annoying. If you add another transformer to the stack, you will have to manually add the `lift`ing to each method. Only instances of the outer most transformer can be used for deriving (`DatabaseT` in this case).

Using methods during initialization

We were able to use `logInfoN` during the initialization. Unfortunately we still have to remember to `lift` the method call, unless each transformer in our stack provides a `MonadLogger` instance.

For a more complicated setup it might become hard to track all the `lift`s and sometimes we might even need to use `liftWith` from `MonadTransControl`.

It would be nice to also have a `MonadLogger m` constraint on `runDatabaseT`. So we basically want to be able to use the full power of each transformer, right after we set it up.

Actually composing transformers

Until now, we have applied transformers on monads to generate a new monad from an existing one. We can also compose two transformers and generate a new transformer with `ComposeT`.

```
newtype ComposeT
  (t1 :: (Type -> Type) -> Type -> Type)
  (t2 :: (Type -> Type) -> Type -> Type)
  (m :: Type -> Type)
  (a :: Type)
    = ComposeT { unComposeT :: t1 (t2 m) a }
  deriving newtype (Applicative, Functor, Monad)
```

Now we have to be clever about adding some instances to `ComposeT`.

Some canonical instances would include `MonadTrans`, `MonadTransControl`, `MonadIO`, `MonadBase`, `MonadBaseControl` and maybe a few more like `MonadThrow` and `MonadCatch`. All of these canonical instances can be implemented, as long as `t1` and `t2` implement `MonadTransControl`. These instances just lift into the base monad `m`.

> **TIP** You can find those canonical implementations here for example.

Then there are also our own semantically important instances, which we have to be especially careful with. Let's look at the example of `MonadLogger`:

```
-- | Default instance.
instance {-# OVERLAPPABLE #-} (Monad (t1 (t2 m)), MonadTrans t1, MonadLogger (t2 m))
=> MonadLogger (ComposeT t1 t2 m) where
  monadLoggerLog loc logSource logLevel = ComposeT . lift . monadLoggerLog loc
logSource logLevel

-- | Override the default instance, whenever `LoggingT` is used in a transformer
stack.
instance {-# OVERLAPPING #-} MonadIO (t2 m) => MonadLogger (ComposeT LoggingT t2 m)
where
  monadLoggerLog loc logSource logLevel = ComposeT . monadLoggerLog loc logSource
logLevel
```

With this setup we can `lift` instances through our entire transformer stack, from the point they are initialized at.

The same overlapping style, using `MonadTrans`/`MonadTransControl` should be used for `MonadReader Configuration` and `MonadDatabase`

This recursive instance lookup will be useful to us, because now we don't have to keep track of `lift`/`liftWith` throughout our transformer stack anymore.

Deriving to the rescue

We did all of this with the premise, that deriving would improve. After we have set up our `ComposeT`, we can derive everything we want for `AppT`. And now we can easily add another layer to our transformer stack without changing any of the other instances.

We can also leave out some instances like `MonadIO` for example, that we needed during initialization, but don't want as part of our environment.

> **NOTE** I am not a huge fan of `MonadIO`, because `MonadBase IO` does the job as well.

```
type (|.) = ComposeT

newtype AppT m a = AppT { unAppT :: (DatabaseT |. ReaderT Configuration |. LoggingT |.
IdentityT) m a }
  deriving newtype (Applicative, Functor, Monad)
  deriving newtype (MonadBase b, MonadBaseControl b)
  deriving newtype (MonadTrans, MonadTransControl)
  deriving newtype (MonadLogger)
  deriving newtype (MonadReader Configuration)
  deriving newtype (MonadDatabase)
```

We need `IdentityT` at the end of our transformer stack, so that our "non-default" instance of `LoggingT` is inferred.

Initializing in style

Now we can finally use any class, as soon as we want. Let's reimplement our initialization.

```
(|.) :: (t1 (t2 m) a -> t2 m a)
     -> (t2 m a -> m a)
     -> ((t1 |. t2) m a -> m a)
(|.) runT1 runT2 = runT2 . runT1 . unComposeT

runApplication :: (MonadIO m, MonadBaseControl IO m) => AppT m a -> m a
runApplication app = do

  let

    runConfigured tma = do
      logInfoN "Reading configuration."
      config <- liftIO acquireConfiguration
      logInfoN $ "Acquired configuration: " <> T.pack (show config)
      runReaderT tma config
```

```
    runDatabaseT' tma = do
      config <- ask
      logInfoN "Connect to the database."
      -- Now we can even have a `MonadLogger m` constraint on `runDatabaseT`.
      runDatabaseT config tma

  runDatabaseT' |. runConfigured |. runStdOutLogging |. runIdentityT $ unAppT app
```

We finally arrived at a solution, that allows us to easily compose each step of initialization and also comfortably derives our instances for us.

References

I personally use this kind of transformer stack for my homepage.

Since `ComposeT` has quite a few canonical instances, it would be sensible to add `ComposeT` to the transformers library.

> **CAUTION** mmorph also implements `ComposeT`, but the instances are a bit different!

I am also using a standalone module just for `ComposeT`. For the project specific instances I then use a newtype `(|.)`. I try to keep class definitions separated from the rest. And then finally I can spin up my application.