Showcase: `deriving-trans`

When creating monad transformer stacks, you usually have to implement some instances manually. Here is a way how this can mostly be avoided. After writing several blog articles on the topic I finally decided to create a library on Hackage.

The library consists of two concepts.

- `Elevator`
- `ComposeT`

`Elevator`

`Elevator` is a newtype wrapper for monad transformers. It can take three arguments.

1. monad transformer `t :: (Type → Type) → Type → Type`
2. monad `m :: Type → Type`
3. value `a :: Type`

```
newtype Elevator t m a = Ascend { descend :: t m a }
  deriving newtype (Functor, Applicative, Monad)
  deriving newtype (MonadTrans, MonadTransControl)
```

We can use it to derive instances for monad transformers.

NOTE	mtl adds an instance of each type class to each transformer. With *n* type classes and *n* transformers this results in n^2 instances. This is particularly annoying when you add your own transformers and type classes.

Example

Let's use mtl's `Reader` as an example.

Type class example "`MonadReader`"

You are probably familiar with the `MonadReader` class.

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

Any transformer, that implements `MonadTransControl` from monad-control can be stacked on top of a `Reader` and the `MonadReader` instance can be passed through. We can observe exactly this with the `Elevator` instance below.

`MonadReader` *instance for* `Elevator`

```
instance (Monad (t m), MonadTransControl t, MonadReader r m) =>
  MonadReader r (Elevator t m) where
    ask = lift ask
    local f tma = (restoreT . pure =<<) $ liftWith $ \ runT ->
      local f $ runT tma
```

Monad transformer example "`ReaderT`"

`ReaderT` is the canonical transformer, that implements a `MonadReader` instance.

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

Here we don't have to do anything special aside from implementing the regular `MonadReader` instance.

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

Usage

Now we can use `Elevator` to access a specific `MonadReader` instance in a transformer stack.

```
newtype CustomT m a = CustomT { unCustomT :: ReaderT Bool (ReaderT Char m) a }
  deriving newtype (Functor, Applicative, Monad)
  deriving (MonadReader Char) via Elevator (ReaderT Bool) (ReaderT Char m) a
```

Usually we would have to define this `MonadReader Char` instance manually, but now we can use *DerivingVia* to generate it for us.

`ComposeT`

`ComposeT` is also a newtype wrapper, but it can take two monad transformers as arguments. This allows us to actually compose two transformers (`t1` and `t2`) into a new transformer (`ComposeT t1 t2`).

1. outer monad transformer `t1 :: (Type → Type) → Type → Type`
2. inner monad transformer `t2 :: (Type → Type) → Type → Type`

3. monad `m :: Type → Type`
4. value `a :: Type`

```
newtype ComposeT t1 t2 m a = ComposeT { deComposeT :: t1 (t2 m) a }
  deriving newtype (Applicative, Functor, Monad)
  -- manually implemented (MonadTrans, MonadTransControl)
```

We can use it to derive instances for monad transformer stacks. For each type class we will need one recursive instance, that can be implemented with `Elevator`. Each transformer, that is supposed to provide an instance for a transformer stack, will require an additional instance.

Example

We are sticking to our `Reader` example.

TIP The recursive instance can always be derived using `Elevator`.

recursive `MonadReader` instance for `ComposeT`

```
deriving via Elevator t1 (t2 (m :: * -> *))
  instance {-# OVERLAPPABLE #-}
    ( Monad (t1 (t2 m))
    , MonadTransControl t1
    , MonadReader r (t2 m)
    ) => MonadReader r (ComposeT t1 t2 m)
```

Additionaly we need the base case for our recursion, which is `ReaderT` in this case.

WARNING The recursive instance is using the `OVERLAPPABLE` pragma, because whenever a `ReaderT` is encountered in a transformer stack, we want to use the following instance.

`ReaderT`'s `MonadReader` instance for `ComposeT`

```
deriving via ReaderT r (t2 (m :: * -> *))
  instance Monad (t2 m) => MonadReader r (ComposeT (ReaderT r) t2 m)
```

Usage

Monad transformer stacks can be useful if you want to combine multiple transformers. The instances I just introduced will look very similar for any type class and transformer.

Now let's get to a use case.

NOTE We will be using a handy infix type operator.

```
type (|.) = ComposeT
```

```
type StackT = StateT Int |. CustomT |. ReaderT Char |. IdentityT
newtype FinalT m a = FinalT { unFinalT :: StackT m a }
  deriving newtype (Functor, Applicative, Monad)
  deriving newtype (MonadTrans, MonadTransControl)
  deriving newtype (MonadBase b, MonadBaseControl b)
  deriving newtype (MonadReader Char)
  deriving newtype (MonadCustom)
  deriving newtype (MonadState Int)
  deriving (MonadError e) via Elevator StackT m
```

CAUTION We add `IdentityT` at the end, because the “base-case” instances only cover `t1` (`ComposeT`'s first argument).

Now we are able to derive a whole lot of instances.

NOTE One big advantage of this method is, that when you change the transformer stack, the instances will still keep working. Especially manually using `lift`/`liftWith` would be cumbersome and even error prone.

We also need a runner function for `FinalT`. We can now implement this incrementally, which is very clean and might be a good way to refactor your huge initialization function, that lived in `IO` until now.

```
runFinalT :: MonadBaseControl IO m => FinalT m a -> m (StT FinalT a)
runFinalT final =
  runStateTFinal |.
    runCustomT |.
      runReaderTFinal |.
        runIdentityT $ unFinalT final
  where
    runReaderTFinal :: MonadBase IO n => ReaderT Char n b -> n b
    runReaderTFinal tma = do
      content <- liftBase $ readFile "config.json"
      case content of
        [] -> error "empty file"
        char : _ -> runReaderT tma char

    runStateTFinal :: MonadReader Char n => StateT Int n b -> n (b, Int)
    runStateTFinal tma = do
      number <- fromEnum <$> ask
      runStateT tma number
```

Now every transformer represents an initialization step.

NOTE

We are using another infix operator here, that allows us to combine transformer runners.

```
(|.) :: (forall a. t1 (t2 m) a -> t2 m (StT t1 a))
     -> (forall a. t2 m a -> m (StT t2 a))
     -> (forall a. (t1 |. t2) m a -> m (StT t2 (StT t1 a)))
(|.) = runComposeT
infixr 1 |.
```

Summary

1. Use `Elevator` to access instances, that are shadowed by transformers stacked on top.
2. Use `ComposeT` to implement large monad transformer stacks.

There are some caveats

- You will need quite a few language extensions *(and I'm too lazy to look them all up)*.
- Be careful with `MonadTransControl`, when implementing `Elevator` instances.
- *DerivingVia* sometimes needs a little help with kind inference.
- Watch out for mistakes with overlapping instances.
- Append `IdentityT` to your `ComposeT` transformer stack, to keep all instances.

I am using this library myself for my homepage. If you notice any problem, I will be happy to hear from you!