Tagging Monad Transformer Layers

Dan Piponi

February 2024 (original 2010)

A quick post extracted from some code I was writing at the weekend.

```
{-# LANGUAGE GADTs, ScopedTypeVariables, AllowAmbiguousTypes #-}

import Control.Monad.Trans
import Control.Monad.State
import Control.Monad.Writer
import Control.Monad.Identity
```

Monad transformers can get a little ugly. Here's a toy example that looks pretty bad:

```
test1 :: StateT Int
         (StateT Int(StateT Int (WriterT String Identity))) Int
test1 = do
   put 1
   lift $ put 2
   lift $ lift $ put 3
   a <- get
   b <- lift $ get
   c <- lift $ lift $ get
   lift $ lift $ lift $ tell $ show $ a+b+c
   return $ a*b*c

go1 = runIdentity (runWriterT
                  (runStateT
                  (runStateT (runStateT test1 0) 0) 0))
```

There are obvious ways to make it prettier, like the suggestions in Real World Haskell. But despite what it says there, the monad "layout" is still "hardwired" and the code is fragile if you decide to insert more layers into your transformer stack. It's no way to program.

So here's an alternative I came up with. First we make a bunch of tags:

```
data A = A
data B = B
data C = C
data D = D
```

We can now label each of the monad transformers with a tag:

```
test2 :: TStateT A Int
                   (TStateT B Int
                   (TStateT C Int
                   (TWriterT D String Identity))) Int
```

And now we can have everything lifted to the appropriate layer automatically:

```
test2 = do
   tput A 1
   tput B 2
   tput C 3
   a <- tget A
   b <- tget B
   c <- tget C
   ttell D $ show $ a+b+c
   return $ a*b*c

go2 = runIdentity (runTWriterT
                   (runTStateT
                   (runTStateT
                   (runTStateT test2 0) 0) 0))
```

Much more readable and much more robust. Change the order of the layers, or insert new ones, and the code still works.

I've tried to make this minimally invasive. It just introduces one new

monad transformer that can be used to tag any other. The definitions like `TStateT` and `tput` are just trivial wrapped versions of their originals.

Anyway, this is just the first thing that came to mind and I threw it together quickly. Surely nobody else likes all those lifts. So what other solutions already exist? I'd rather use someone else's well tested library than my hastily erected solution:

```
data T tag m a = T { runTag :: m a } deriving Show

instance Functor m => Functor (T tag m) where
   fmap f (T a) = T (fmap f a)

instance Applicative m => Applicative (T tag m) where
   pure a = T (pure a)
   T a <*> T b = T (a <*> b)

instance Monad m => Monad (T tag m) where
   T x >>= f = T $ x >>= (runTag . f)

instance MonadTrans (T tag) where
   lift m = T m

class TWith tag (m :: * -> *) (n :: * -> *) where
   taggedLift :: tag -> m a -> n a

instance {-# OVERLAPPING #-} (Monad m, m ~ n) =>
   TWith tag m (T tag n) where
     taggedLift _ x = lift x

instance {-# OVERLAPPING #-}
   (Monad m, Monad n, TWith tag m n, MonadTrans t) =>
   TWith tag m (t n) where
     taggedLift tag x = lift (taggedLift tag x)

type TStateT tag s m = T tag (StateT s m)
runTStateT = runStateT . runTag

tput tag x = taggedLift tag (put x)
tget tag = taggedLift tag get

type TWriterT tag w m = T tag (WriterT w m)
runTWriterT = runWriterT . runTag
```

```
ttell tag x = taggedLift tag (tell x)

main = do
  print go1
  print go2
```