Is it possible to implement `(Applicative m) => Applicative (StateT s m)`?

I'm currently working on `Data.Fresh` and `Control.Monad.Trans.Fresh`, which resp. define an interface for generating fresh variables, and a monad transformer which implements this interface.

I initially thought it would be possible to implement the `Applicative` instance for my `FreshT v m` with the only requirement that `Applicative m` exists. However, I got stuck and it seemed like I need to require `Monad m`. Not trusting my Haskell-fu, I then turned to the transformers package, and was surprised by what I found in `Control.Monad.Trans.State.Lazy` and `.Strict`:

```
instance (Functor m, Monad m) => Applicative (StateT s m) where
    pure = return
    (<*>) = ap
```

So here is my question: is it possible to create an instance with equivalent semantics with the following instance head?

```
instance (Applicative m) => Applicative (StateT s m) where
```

haskell monad-transformers state-monad

asked Sep 7 '13 at 12:56

Rhymoid
5,607 2 20 47

3 Answers

Consider that you have two functions:

```
f :: s -> m (s, a -> b)
g :: s -> m (s, a)
```

And you want to create a function `h = StateT f <*> StateF g`

```
h :: s -> m (s, b)
```

From the above you have an `s` you can pass to `f` so you have:

```
f' :: m (s, a -> b)
g :: s -> m (s, a)
```

However to get `s` out of `f'` you need the Monad (whatever you'd do with applicative it would still be in form of `m s` so you would not be able to apply the value to `g`).

You can play with the definitions and use free monad but for the collapse of state you need `join`.

answered Sep 7 '13 at 13:40

Maciej Piechotka
4,058 2 19 47

I already thought so, since I was consistently confronted with some `m (m a)`. – Rhymoid Sep 7 '13 at 13:43

1 @Rhymoid: If you skip the `a -> b` part you would get `f' :: m s` and `g :: s -> m b`. Hence implementing the `<*>` for `StateT` would give an implementation of `>>=` for `m` assuming `s` can be arbitrary. – Maciej Piechotka Sep 7 '13 at 13:45

Surely this is wrong? StateT is a Monad, so by definition it is an Applicative, and that's why in the standard libraries they simply state `(<*>) = ap`. But what that means is that you MUST be able to implement `(<*>)` directly, using only `(<*>)` or `fmap` from the inner Applicative. Trouble is that I can't see how to do it. – Simon H Jan 12 '15 at 13:12

@SimonH Note that we assume that `m` is `Applicative` not `Monad`. Therefore `StateT m` is neither `Applicative` nor `Monad`. Had `m` be `Monad` you are right - `StateT` is `Monad` and therefore `Applicative`. – Maciej Piechotka Jan 12 '15 at 16:39

Although, as noted in the previous answer, this instance cannot be defined in general, it is worth noting that, when `f` is `Applicative` and `s` is a `Monoid`, `StateT s f` is also `Applicative`, since it can be regarded as a composition of applicative functors:

```
StateT s f = Reader s `Compose` f `Compose` Writer s
```

answered Sep 7 '13 at 14:06

Paolo Capriotti
3,289 11 20

Weaker variant of an `Applicative` transformer

Although it isn't possible to define an applicative transformer for `StateT`, It's possible to define a weaker variant that works. Instead of having `s -> m (a, s)`, where the state decides the next effect (therefore `m` must be a monad), we can use `m (s -> (a, s))`, or equivalently `m (State s a)`.

```
import Control.Applicative
import Control.Monad
import Control.Monad.State
import Control.Monad.Trans

newtype StateTA s m a = StateTA (m (State s a))
```

This is strictly weaker than `StateT`. Every `StateTA` can be made into `StateT` (but not vice versa):

```
toStateTA :: Applicative m => StateTA s m a -> StateT s m a
toStateTA (StateTA k) = StateT $ \s -> flip runState s <$> k
```

Defining `Functor` and `Applicative` is just the matter of lifting operations of `State` into the underlying `m`:

```
instance (Functor m) => Functor (StateTA s m) where
    fmap f (StateTA k) = StateTA $ liftM f <$> k
instance (Applicative m) => Applicative (StateTA s m) where
    pure = StateTA . pure . return
    (StateTA f) <*> (StateTA k) = StateTA $ ap <$> f <*> k
```

And we can define an applicative variant of `lift`:

```
lift :: (Applicative m) => m a -> StateTA s m a
lift = StateTA . fmap return
```

Update: Actually the above isn't necessary, as the composition of two applicative functors is always an applicative functor (unlike monads). Our `StateTA` is isomorphic to `Compose m (State s)`, which is automatically `Applicative`:

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

Therefore we could write just

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
import Control.Applicative
import Control.Monad.State
import Data.Functor.Compose

newtype StateTA s m a = StateTA (Compose m (State s) a)
    deriving (Functor, Applicative)
```

edited Sep 7 '13 at 18:11

answered Sep 7 '13 at 17:37

Petr Pudlák
36.5k 4 74 213

Very interesting. In my application, the state itself (a supply of fresh values) doesn't necessarily decide the next effect, just how often it is duplicated and extracted. One of my implementations can get around this problem, the other not yet; perhaps this answer can be of use. Thanks! – Rhymoid Sep 7 '13 at 22:51