LiU-FP2016: Lecture 10

Monad Transformers

Henrik Nilsson

University of Nottingham, UK

Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

We could implement a suitable monad from scratch:

```
newtype SE s a = SE (s -> Maybe (a, s))
```


Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

  ```
  newtype SE s a = SE (s -> (Maybe a, s))
  ```

- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, . . .), allowing the programmer to mix and match.
- A form of ***aspect-oriented programming***.

Caveat: Will consider the idea of monad transformers, not any specific library like e.g. MTL.

Monad Transformers in Haskell (1)

- A ***monad transformer*** maps monads to monads. Represented by a type constructor `T` of the following kind:

  ```
  T :: (* -> *) -> (* -> *)
  ```

- Additionally, a monad transformer ***adds*** computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

  ```
  lift :: M a -> T M a
  ```


Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:

  ```
  class (Monad m, Monad (t m))
        => MonadTransformer t m where
      lift :: m a -> t m a
  ```


Classes for Specific Effects

A monad transformer adds specific effects to ***any*** monad. Thus the effect-specific operations needs to be overloaded. For example:

```
class Monad m => E m where
    eFail   :: m a
    eHandle :: m a -> m a -> m a

class Monad m => S m s | m -> s where
    sSet :: s -> m ()
    sGet :: m s
```


The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

```
newtype I a = I a
unI (I a) = a

instance Monad I where
    return a = I a
    m >>= f  = f (unI m)

runI :: I a -> a
runI = unI
```


The Error Monad Transformer (1)

```
newtype ET m a = ET (m (Maybe a))
unET (ET m) = m
```

Any monad transformed by `ET` is a monad:

```
instance Monad m => Monad (ET m) where
    return a = ET (return (Just a))

    m >>= f = ET $ do
        ma <- unET m
        case ma of
            Nothing -> return Nothing
            Just a  -> unET (f a)
```

The Error Monad Transformer (2)

We need the ability to run transformed monads:

```
runET :: Monad m => ET m a -> m a
runET etm = do
    ma <- unET etm
    case ma of
        Just a  -> return a
        Nothing -> error "Should not happen"
```

ET is a monad transformer:

```
instance Monad m =>
         MonadTransformer ET m where
    lift m = ET (m >>= \a -> return (Just a))
```


The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

```
instance Monad m => E (ET m) where
    eFail = ET (return Nothing)
    m1 `eHandle` m2 = ET $ do
        ma <- unET m1
        case ma of
            Nothing -> unET m2
            Just _  -> return ma
```


The Error Monad Transformer (4)

A state monad transformed by ET is a state monad:

```
instance S m s => S (ET m) s where
    sSet s = lift (sSet s)
    sGet = lift sGet
```


Exercise 2: Running Transf. Monads

Let

```
ex2 = eFail `eHandle` return 1
```

1. Suggest a possible type for ex2. (Assume `1 :: Int`.)
2. Given your type, use the appropriate combination of “run functions” to run ex2.

Exercise 2: Solution

```
ex2 :: ET I Int
ex2 = eFail `eHandle` return 1

ex2result :: Int
ex2result = runI (runET ex2)
```


The State Monad Transformer (1)

```
newtype ST s m a = ST (s -> m (a, s))
unST (ST m) = m
```

Any monad transformed by ST is a monad:

```
instance Monad m => Monad (ST s m) where
    return a = ST (\s -> return (a, s))

    m >>= f = ST $ \s -> do
        (a, s') <- unST m s
        unST (f a) s'
```


The State Monad Transformer (2)

We need the ability to run transformed monads:

```
runST :: Monad m => ST s m a -> s -> m a
runST stf s0 = do
    (a, _) <- unST stf s0
    return a
```

ST is a monad transformer:

```
instance Monad m =>
         MonadTransformer (ST s) m where
    lift m = ST (\s -> m >>= \a ->
                    return (a, s))
```


The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

```
instance Monad m => S (ST s m) s where
    sSet s = ST (\_ -> return ((), s))
    sGet   = ST (\s -> return (s, s))
```

An error monad transformed by ST is an error monad:

```
instance E m => E (ST s m) where
    eFail = lift eFail
    m1 `eHandle` m2 = ST $ \s ->
        unST m1 s `eHandle` unST m2 s
```


Exercise 3: Effect Ordering

Consider the code fragment

```
ex3a :: (ST Int (ET I)) Int
ex3a = (sSet 42 >> eFail) `eHandle` sGet
```

Note that the exact same code fragment also can be typed as follows:

```
ex3b :: (ET (ST Int I)) Int
ex3b = (sSet 42 >> eFail) `eHandle` sGet
```

What is

```
runI (runET (runST ex3a 0))
runI (runST (runET ex3b) 0)
```

Exercise 3: Solution

```
runI (runET (runST ex3a 0)) = 0
runI (runST (runET ex3b) 0) = 42
```

Why? Because:

```
ST s (ET I) a ≅ s -> (ET I) (a, s)
              ≅ s -> I (Maybe (a, s))
              ≅ s -> Maybe (a, s)
ET (ST s I) a ≅ (ST s I) (Maybe a)
              ≅ s -> I (Maybe a, s)
              ≅ s -> (Maybe a, s)
```


Exercise 4: Alternative ST?

To think about.

Could ST have been defined in some other way, e.g.

```
newtype ST s m a = ST (m (s -> (a, s)))
```

or perhaps

```
newtype ST s m a = ST (s -> (m a, s))
```


Problems with Monad Transformers

- With one transformer for each possible effect, we get a lot of combinations: the number grows quadratically; each has to be instantiated explicitly.
- Jaskelioff (2008,2009) has proposed a possible, more extensible alternative.

Reading (1)

- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, January 1995, San Francisco, California

Reading (2)

- Mauro Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *Implementation of Functional Languages (IFL'08)*, 2008.
- Mauro Jaskelioff. Modular Monad Transformers. In *European Symposium on Programming (ESOP,09)*, 2009.