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:

```haskell
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
  ```haskell
  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 in Haskell (1)

A monad transformer maps monads to monads. Represented by a type constructor \( T \) of the following kind:

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

Additionally, a monad transformer adds computational effects. A mapping \( \text{lift} \) from computations in the underlying monad to computations in the transformed monad is needed:

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

Monad Transformers in Haskell (2)

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

```haskell
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
```

Classes for Specific Effects

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

```haskell
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:

```haskell
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)

Any monad transformed by \( ET \) is a monad:

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

```haskell
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:

```haskell
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`:

```haskell
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
```

Exercise 2: Running Transf. Monads

Let

```haskell
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

```haskell
ex2 :: ET I Int
ex2 = eFail 'eHandle' return 1
ex2result :: Int
ex2result = runI (runET ex2)
```

Exercise 3: Effect Ordering

Consider the code fragment

```haskell
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:

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

What is

```haskell
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:

\[
\begin{align*}
ST \ s \ (ET \ I) \ a & \cong s \to (ET \ I) \ (a, s) \\
& \cong s \to I \ (Maybe \ (a, s)) \\
& \cong s \to Maybe \ (a, s) \\
ET \ (ST \ s \ I) \ a & \cong (ST \ s \ I) \ (Maybe \ a) \\
& \cong s \to I \ (Maybe \ a, s) \\
& \cong s \to (Maybe \ a, s)
\end{align*}
\]

Exercise 4: Alternative ST?

To think about.

Could \texttt{ST} have been defined in some other way, e.g.

\[
\text{newtype ST } s \ m \ a = ST \ (m \ (s \to (a, s)))
\]

or perhaps

\[
\text{newtype ST } s \ m \ a = ST \ (s \to (m \ a, s))
\]

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)


Reading (2)