COMP4075: Lecture 11

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:

$$\mathbf{newtype}\ SE\ s\ a = SE\ (s \to Maybe\ (a, s))$$

Monad Transformers (2)

However:

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

$$\mathbf{newtype}\ SE\ s\ a = SE\ (s \to (Maybe\ a, s))$$

- 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.
- Monad transformer libraries can be developed, each transformer each adding a specific effect (state, error, . . .).
- A form of ***aspect-oriented programming***.
- MTL is one example of such a library.

Will consider the general idea of monad transformers first; specific libraries discussed later.

Monad Transformers in Haskell (1)

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

$$T :: (* \to *) \to (* \to *)$$

- Additionally, a monad transformer ***adds*** computational effects.
- A mapping $lift$ maps a computation in the underlying monad to one in the transformed monad:

$$lift :: M\ a \to T\ M\ a$$

Monad Transformers in Haskell (2)

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

$$\begin{array}{l}\mathbf{class}\ (Monad\ m, Monad\ (t\ m)) \\ \quad\quad \Rightarrow MonadTransformer\ t\ m\ \mathbf{where} \\ \quad lift :: m\ a \to t\ m\ a\end{array}$$

Classes for Specific Effects

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

$$\begin{array}{l}\mathbf{class}\ Monad\ m \Rightarrow E\ m\ \mathbf{where} \\ \quad eFail \quad\ :: m\ a \\ \quad eHandle :: m\ a \to m\ a \to m\ a \\ \mathbf{class}\ Monad\ m \Rightarrow S\ m\ s \mid m \to s\ \mathbf{where} \\ \quad sSet\ :: s \to m\ () \\ \quad sGet :: m\ s\end{array}$$

The Identity Monad

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

$$\begin{array}{l}\mathbf{newtype}\ I\ a = I\ a \\ unI\ (I\ a) = a \\ \mathbf{instance}\ Monad\ I\ \mathbf{where} \\ \quad return\ a = I\ a \\ \quad m \gg\!= f\ \ = f\ (unI\ m) \\ runI\ \ :: I\ a \to a \\ runI = unI\end{array}$$

The Error Monad Transformer (1)

$$\begin{array}{l}\mathbf{newtype}\ ET\ m\ a = ET\ (m\ (Maybe\ a)) \\ unET\ (ET\ m) = m\end{array}$$

The Error Monad Transformer (2)

Any monad transformed by ET is a monad:

```haskell
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 (3)

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"
```

(Note: To simplify use, we discarded information about the effect, but as a result, we get a partial function. Returning $Maybe\ a$ better in general.)

The Error Monad Transformer (4)

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 (5)

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
```


The Error Monad Transformer (6)

A state monad transformed by ET is a state monad:

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


Exercise 1: 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 1: Solution

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

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


The State Monad Transformer (1)

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

Any monad transformed by ST is a monad:

```haskell
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:

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

(We are again discarding information to keep things simple. Returning the final state along with result would be more general.)

The State Monad Transformer (3)

ST is a monad transformer:

$$\begin{array}{l} \textbf{instance}\ \mathit{Monad}\ m \Rightarrow \\ \qquad\qquad \mathit{MonadTransformer}\ (\mathit{ST}\ s)\ m\ \textbf{where} \\ \quad \mathit{lift}\ m = \mathit{ST}\ (\lambda s \to m \ggg\!= \lambda a \to \mathit{return}\ (a, s)) \end{array}$$

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

$$\begin{array}{l} \textbf{instance}\ \mathit{Monad}\ m \Rightarrow S\ (\mathit{ST}\ s\ m)\ s\ \textbf{where} \\ \quad \mathit{sSet}\ s = \mathit{ST}\ (\lambda\_ \to \mathit{return}\ ((), s)) \\ \quad \mathit{sGet} = \mathit{ST}\ (\lambda s \to \mathit{return}\ (s, s)) \end{array}$$

The State Monad Transformer (4)

An error monad transformed by ST is an error monad:

$$\begin{array}{l} \textbf{instance}\ E\ m \Rightarrow E\ (\mathit{ST}\ s\ m)\ \textbf{where} \\ \quad \mathit{eFail} = \mathit{lift}\ \mathit{eFail} \\ \quad \mathit{m1}\ `\mathit{eHandle}`\ \mathit{m2} = \mathit{ST}\ \$\ \lambda s \to \\ \qquad \mathit{unST}\ \mathit{m1}\ s\ `\mathit{eHandle}`\ \mathit{unST}\ \mathit{m2}\ s \end{array}$$

Exercise 2: Effect Ordering

Consider the code fragment

$$\begin{array}{l} \mathit{ex3a} :: (\mathit{ST}\ \mathit{Int}\ (\mathit{ET}\ I))\ \mathit{Int} \\ \mathit{ex3a} = (\mathit{sSet}\ 42 \gg \mathit{eFail})\ `\mathit{eHandle}`\ \mathit{sGet} \end{array}$$

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

$$\begin{array}{l} \mathit{ex3b} :: (\mathit{ET}\ (\mathit{ST}\ \mathit{Int}\ I))\ \mathit{Int} \\ \mathit{ex3b} = (\mathit{sSet}\ 42 \gg \mathit{eFail})\ `\mathit{eHandle}`\ \mathit{sGet} \end{array}$$

What is

$$\begin{array}{l} \mathit{runI}\ (\mathit{runET}\ (\mathit{runST}\ \mathit{ex3a}\ 0)) \\ \mathit{runI}\ (\mathit{runST}\ (\mathit{runET}\ \mathit{ex3b})\ 0) \end{array}$$

Exercise 2: Solution

$$\begin{array}{rcl} \mathit{runI}\ (\mathit{runET}\ (\mathit{runST}\ \mathit{ex3a}\ 0)) & = & 0 \\ \mathit{runI}\ (\mathit{runST}\ (\mathit{runET}\ \mathit{ex3b})\ 0) & = & 42 \end{array}$$

Why? Because:

$$\begin{array}{rcl} \mathit{ST}\ s\ (\mathit{ET}\ I)\ a & \cong & s \to (\mathit{ET}\ I)\ (a, s) \\ & \cong & s \to I\ (\mathit{Maybe}\ (a, s)) \\ & \cong & s \to \mathit{Maybe}\ (a, s) \\ \mathit{ET}\ (\mathit{ST}\ s\ I)\ a & \cong & (\mathit{ST}\ s\ I)\ (\mathit{Maybe}\ a) \\ & \cong & s \to I\ (\mathit{Maybe}\ a, s) \\ & \cong & s \to (\mathit{Maybe}\ a, s) \end{array}$$

MTL: Monad Transformer Library

Provides a number of standard monads, associated transformers, and all possible liftings in the style we have seen; e.g.:

- State ($\mathit{Control.Monad.State}$, lazy and strict)
- Exceptions ($\mathit{Control.Monad.Except}$)
- Lists ($\mathit{Control.Monad.List}$)
- Reader ($\mathit{Control.Monad.Reader}$)
- Writer ($\mathit{Control.Monad.Writer}$)
- Contiunations ($\mathit{Control.Monad.Cont}$)

MTL: State

$$\begin{array}{l} \textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \to s\ \textbf{where} \\ \quad \mathit{get}\ \ \ :: m\ s \\ \quad \mathit{put}\ \ \ :: s \to m\ () \\ \quad \mathit{state} :: (s \to (a, s)) \to m\ a \end{array}$$

Transformer: $\textbf{newtype}\ \mathit{StateT}\ s\ (m :: * \to *)\ a$

Run functions:

$$\begin{array}{l} \mathit{runState} :: \mathit{State}\ s\ a \to s \to (a, s) \\ \mathit{evalState} :: \mathit{State}\ s\ a \to s \to a \\ \mathit{execState} :: \mathit{State}\ s\ a \to s \to s \end{array}$$

MTL: Exception

$$\begin{array}{l} \textbf{class}\ \mathit{Monad}\ m \Rightarrow \\ \qquad \mathit{MonadError}\ e\ m \mid m \to e\ \textbf{where} \\ \quad \mathit{throwError} :: e \to m\ a \\ \quad \mathit{catchError}\ :: m\ a \to (e \to m\ a) \to m\ a \end{array}$$

Transformer: $\textbf{newtype}\ \mathit{ExceptT}\ e\ (m :: * \to *)\ a$

Run function:

$$\mathit{runExcept} :: \mathit{Except}\ e\ a \to \mathit{Either}\ e\ a$$

MTL: Reader

$$\begin{array}{l} \textbf{class}\ \mathit{Monad}\ m \Rightarrow \\ \qquad \mathit{MonadReader}\ r\ m \mid m \to r\ \textbf{where} \\ \quad \mathit{ask}\ \ \ \ :: m\ r \\ \quad \mathit{local}\ \ :: (r \to r) \to m\ a \to m\ a \\ \quad \mathit{reader} :: (r \to a) \to m\ a \end{array}$$

Transformer: $\mathit{ReaderT}$

Run function:

$$\mathit{runReader} :: \mathit{Reader}\ r\ a \to r \to a$$

MTL: Writer

$$\mathbf{class}\ (\mathit{Monoid}\ w, \mathit{Monad}\ m) \Rightarrow$$
$$\mathit{MonadWriter}\ w\ m \mid m \to w\ \mathbf{where}$$
$$\mathit{writer} :: (a, w) \to m\ a$$
$$\mathit{tell}\ \ :: w \to m\ ()$$
$$\mathit{listen}\ :: m\ a \to m\ (a, w)$$
$$\mathit{pass}\ \ :: m\ (a, w \to w) \to m\ a$$

Transformer: $\mathbf{newtype}\ \mathit{WriterT}\ w\ (m :: * \to *)\ a$

Run function:

$$\mathit{runWriter} :: \mathit{Writer}\ w\ a \to (a, w)$$

Problems with Monad Transformers

- With one transformer for each possible effect we get a quadratic number of combinations; each has to be instantiated explicitly.
- Jaskelioff (2008,2009) has proposed a possible, more extensible alternative:
  - Traditional approach: unsystematic lifting on case-by-case basis.
  - Jaskelioff: systematic lifting based on theoretical principles where each operation is paired with a type of its implementation allowing implementations to be transformed generically.

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.