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:

$$\mathit{lift} :: M\ a \to T\ M\ a$$

Monad Transformers in Haskell (2)

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

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

Classes for Specific Effects

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

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

The Identity Monad

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

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

The Error Monad Transformer (1)

$$\textbf{newtype}\ ET\ m\ a = ET\ (m\ (Maybe\ a))$$
$$unET\ (ET\ m) = m$$

The Error Monad Transformer (2)

Any monad transformed by ET is a monad:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow Monad\ (ET\ m)\ \textbf{where} \\
\quad return\ a = ET\ (return\ (Just\ a)) \\
\quad m \gg= f = ET\ \$\ \textbf{do} \\
\quad\quad ma \leftarrow unET\ m \\
\quad\quad \textbf{case}\ ma\ \textbf{of} \\
\quad\quad\quad Nothing \rightarrow return\ Nothing \\
\quad\quad\quad Just\ a\ \ \ \ \rightarrow unET\ (f\ a)
\end{array}
$$

The Error Monad Transformer (3)

We need the ability to run transformed monads:

$$
\begin{array}{l}
runET :: Monad\ m \Rightarrow ET\ m\ a \rightarrow m\ a \\
runET\ etm = \textbf{do} \\
\quad ma \leftarrow unET\ etm \\
\quad \textbf{case}\ ma\ \textbf{of} \\
\quad\quad Just\ a\ \ \ \ \rightarrow return\ a \\
\quad\quad Nothing \rightarrow error\ \texttt{"Should not happen"}
\end{array}
$$

(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:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow \\
\qquad\qquad\qquad MonadTransformer\ ET\ m\ \textbf{where} \\
\quad lift\ m = ET\ (m \gg= \lambda a \rightarrow return\ (Just\ a))
\end{array}
$$

The Error Monad Transformer (5)

Any monad transformed by ET is an instance of E:

$$
\begin{array}{l}
\textbf{instance } Monad\ m \Rightarrow E\ (ET\ m)\ \textbf{where} \\
\quad eFail = ET\ (return\ Nothing) \\
\quad m1\ `eHandle`\ m2 = ET\ \$\ \textbf{do} \\
\qquad ma \leftarrow unET\ m1 \\
\qquad \textbf{case}\ ma\ \textbf{of} \\
\qquad\quad Nothing \rightarrow unET\ m2 \\
\qquad\quad Just\ \_ \quad \rightarrow return\ ma
\end{array}
$$

The Error Monad Transformer (6)

A state monad transformed by ET is a state monad:

$$
\begin{array}{l}
\textbf{instance } S\ m\ s \Rightarrow S\ (ET\ m)\ s\ \textbf{where} \\
\quad sSet\ s = lift\ (sSet\ s) \\
\quad sGet \quad = lift\ sGet
\end{array}
$$

Exercise 1: 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 1: Solution

$$
\begin{array}{l}
ex2 :: ET\ I\ Int \\
ex2 = eFail\ `eHandle`\ return\ 1 \\
ex2result :: Int \\
ex2result = runI\ (runET\ ex2)
\end{array}
$$

The State Monad Transformer (1)

$$\textbf{newtype}\ ST\ s\ m\ a = ST\ (s \to m\ (a, s))$$
$$unST\ (ST\ m) = m$$

Any monad transformed by ST is a monad:

$$\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow Monad\ (ST\ s\ m)\ \textbf{where} \\
\quad return\ a = ST\ (\lambda s \to return\ (a, s)) \\
\quad m \gg= f = ST\ \$\ \lambda s \to \textbf{do} \\
\qquad (a, s') \leftarrow unST\ m\ s \\
\qquad unST\ (f\ a)\ s'
\end{array}$$

The State Monad Transformer (2)

We need the ability to run transformed monads:

$$\begin{array}{l}
runST :: Monad\ m \Rightarrow ST\ s\ m\ a \to s \to m\ a \\
runST\ stf\ s0 = \textbf{do} \\
\quad (a, _) \leftarrow unST\ stf\ s0 \\
\quad return\ a
\end{array}$$

(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:

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

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

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

The State Monad Transformer (4)

An error monad transformed by ST is an error monad:

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

Exercise 2: Effect Ordering

Consider the code fragment

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

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

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

What is

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

Exercise 2: Solution

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

Why? Because:

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

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

MTL: State

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

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

Run functions:

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

MTL: Exception

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

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

Run function:

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

MTL: Reader

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

Transformer: $\mathit{ReaderT}$

Run function:

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

MTL: Writer

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Monoid}\ w, \mathit{Monad}\ m) \Rightarrow\\
&\qquad \mathit{MonadWriter}\ w\ m \mid m \to w\ \textbf{where}\\
&\quad \mathit{writer} :: (a, w) \to m\ a\\
&\quad \mathit{tell} \;\;\; :: w \to m\ ()\\
&\quad \mathit{listen} \; :: m\ a \to m\ (a, w)\\
&\quad \mathit{pass} \;\;\; :: m\ (a, w \to w) \to m\ a
\end{aligned}
$$

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

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.