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?

Monad Transformers (1)

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

For example: State and Error/Partiality?

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:

$$\textbf{newtype}\ SE\ s\ a = SE\ (s \rightarrow Maybe\ (a, s))$$

Monad Transformers (2)

However:

Monad Transformers (2)

However:

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

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

Monad Transformers (2)

However:

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

$$\textbf{newtype}\ SE\ s\ a = SE\ (s \rightarrow (Maybe\ a, s))$$

- 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:

Monad Transformers (3)

Monad Transformers can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.

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, . . .).

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***.

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.

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:

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

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.

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:

$$\begin{aligned} &\textbf{class}\ (\mathit{Monad}\ m, \mathit{Monad}\ (t\ m)) \\ &\qquad \Rightarrow \mathit{MonadTransformer}\ t\ m\ \textbf{where} \\ &\quad \mathit{lift} :: m\ a \rightarrow t\ m\ a \end{aligned}$$

Classes for Specific Effects

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

$$\textbf{class } Monad\ m \Rightarrow E\ m \textbf{ where}$$
$$eFail \quad :: m\ a$$
$$eHandle :: m\ a \to m\ a \to m\ a$$

$$\textbf{class } Monad\ m \Rightarrow S\ m\ s \mid m \to s \textbf{ where}$$
$$sSet \ :: s \to m\ ()$$
$$sGet :: m\ s$$

The Identity Monad

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

$$\textbf{newtype}\ I\ a = I\ a$$

$$unI\ (I\ a) = a$$

$$\textbf{instance}\ Monad\ I\ \textbf{where}$$

$$\quad return\ a = I\ a$$

$$\quad m \ggg\!= f\ = f\ (unI\ m)$$

$$runI :: I\ a \rightarrow a$$

$$runI = unI$$

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:

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

The Error Monad Transformer (3)

We need the ability to run transformed monads:

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

The Error Monad Transformer (3)

We need the ability to run transformed monads:

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

(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:

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

The Error Monad Transformer (5)

Any monad transformed by ET is an instance of E:

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

The Error Monad Transformer (6)

A state monad transformed by ET is a state monad:

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

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

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

$$ex2result :: Int$$
$$ex2result = runI\ (runET\ ex2)$$

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:

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

The State Monad Transformer (2)

We need the ability to run transformed monads:

$$
\begin{aligned}
& \mathit{runST} :: \mathit{Monad}\ m \Rightarrow \mathit{ST}\ s\ m\ a \rightarrow s \rightarrow m\ a \\
& \mathit{runST}\ \mathit{stf}\ \mathit{s0} = \mathbf{do} \\
& \quad (a, \_) \leftarrow \mathit{unST}\ \mathit{stf}\ \mathit{s0} \\
& \quad \mathit{return}\ a
\end{aligned}
$$

The State Monad Transformer (2)

We need the ability to run transformed monads:

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

(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:

$$\begin{aligned} &\textbf{instance } Monad\ m \Rightarrow \\ &\qquad\qquad MonadTransformer\ (ST\ s)\ m \textbf{ where} \\ &\quad lift\ m = ST\ (\lambda s \rightarrow m \gg\!= \lambda a \rightarrow return\ (a, s)) \end{aligned}$$

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

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

The State Monad Transformer (4)

An error monad transformed by ST is an error monad:

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

Exercise 2: Effect Ordering

Consider the code fragment

$$ex3a :: (ST\ Int\ (ET\ I))\ Int$$
$$ex3a = (sSet\ 42 \gg eFail)\ `eHandle`\ sGet$$

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

$$ex3b :: (ET\ (ST\ Int\ I))\ Int$$
$$ex3b = (sSet\ 42 \gg eFail)\ `eHandle`\ sGet$$

What is

$$runI\ (runET\ (runST\ ex3a\ 0))$$
$$runI\ (runST\ (runET\ ex3b)\ 0)$$

Exercise 2: Solution

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

Why? Because:

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

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

$$\textbf{class}\ Monad\ m \Rightarrow MonadState\ s\ m \mid m \to s\ \textbf{where}$$

$$\begin{aligned} get &:: m\ s \\ put &:: s \to m\ () \\ state &:: (s \to (a, s)) \to m\ a \end{aligned}$$

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

Run functions:

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

MTL: Exception

$$\textbf{class } Monad\ m \Rightarrow$$
$$MonadError\ e\ m \mid m \rightarrow e \textbf{ where}$$
$$throwError :: e \rightarrow m\ a$$
$$catchError :: m\ a \rightarrow (e \rightarrow m\ a) \rightarrow m\ a$$

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

Run function:

$$runExcept :: Except\ e\ a \rightarrow Either\ e\ a$$

MTL: Reader

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

Transformer: $ReaderT$

Run function:

$$runReader :: Reader\ r\ a \rightarrow r \rightarrow a$$

MTL: Writer

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

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

Run function:

$$runWriter :: Writer\ w\ a \rightarrow (a, w)$$

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.