• operations added by Transform1 and
• operations added by Transform2 and
• operations added by Transform3 ...


"Exceptions"
  - Either, throw, catch

"Global State"
  - ST s → (a, s)
  - put, get

Mixing Monads with Transformers

• Step 1: Specifying Monads with Extra Features
• Step 2: Implementing Monads with Extra Features
Specifying Monads with Extra Features

First, instead of using concrete monads

- e.g. Profile or Either

We will use type-classes to abstractly specify a monad’s capabilities
A Class for State-Transformers Monads

The class MonadState s m defined in the Control.Monad.State (http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html) says

- m is a State-Transformer monad with state type s

class Monad m => MonadState s m where
  get :: m s
  put :: s -> m ()

That is to say, m implements
- \texttt{>>=} and \texttt{return} operations specified by \texttt{Monad and}
- \texttt{get and put} operations specified by \texttt{MonadState}!

**Generalize Types to use Classes**

So we can \texttt{generalize} the type of \texttt{count} to use \texttt{MonadState Int m}

\begin{verbatim}
count :: (MonadState Int m) => m ()
count = do
    n <- get
    put (n+1)
\end{verbatim}
A Class for Exception Handling Monads

The class `MonadError e m` defined in `[Control.Monad.Except][6]` says

- `m` is a `Exception-Handling` monad with exception type `e`

```haskell
class Monad m => MonadError e m where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

That is to say, `m` implements

- `>>=` and `return` operations specified by `Monad` and
- `throwError` and `catchError` operations specified by `MonadError`!

Generalize Types to use Classes

So we can `generalize` the type of `tryCatch` to use `MonadError e m`

```haskell
tryCatch :: (MonadError e m) => m a -> a -> m a
tryCatch m def = catchError m (\_ -> return def)
```
Generalize `eval` to use `Constraints`

We can now specify that `eval` uses a monad `m` that implements

- `MonadState Int` and `MonadError Expr`
eval :: (MonadState Int m, MonadError Expr m) => Expr -> m Int
eval (Number n)  = return n
eval (Plus  e1 e2) = do n1 <- eval e1
                        n2 <- eval e2
                        count
                        return (n1 + n2)
eval (Div   e1 e2) = do n1 <- eval e1
                        n2 <- eval e2
                        count
                        if (n2 /= 0)
                        then return (n1 'div' n2)
                        else throwError e2
 eval (Try e n)    = tryCatch (eval e) n

Lets try to run it!

>>> e1

>>> evalMix e1
... GHC yells "please IMPLEMENT this MAGIC monad that implements BOTH features"
Mixing Monads with Transformers

- Step 1: Specifying Monads with Extra Features
- Step 2: Implementing Monads with Extra Features
Implementing Monads with Extra Features

Transform2 (Transform1 m) implements

- m operations and
- operations added by Transform1 and
- operations added by Transform2
We require

- A basic monad \( m \)
- A Transform\(^1\) that adds State capabilities
- A Transform\(^2\) that adds Exception capabilities

A Basic Monad

First, let's make a basic monad

- only implements \texttt{>>=} and \texttt{return}
**data** Identity a = Id a

**instance** Monad Identity where

\[
\begin{align*}
\text{return } a &\quad = \text{Id } a \\
(\text{Id } a) \gg= f &\quad = f a
\end{align*}
\]

A very *basic* monad: just a *wrapper* (Id) around the value (a)

- No extra features
A Transform that adds **State** Capabilities

The transformer `StateT s m` defined in the `Control.Monad.State` module (http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html) - *takes* as input monad `m` and

- *transforms* it into a new monad `m'`

such that `m'` implements

- all the operations that `m` implements

- *and adds* State-transformer capabilities

`StateT s m` satisfies the constraint `(MonadState s (StateT s m))

A State-transformer over **Int** states

```
type Prof = StateT Int Identity
```

\[(\text{StateT} \text{Id})\]

We can go back and give `evalProf` the type
evalProf :: Expr -> Prof Int

\[\text{A Transform that adds \textbf{Exception Capabilities}}\]

The transformer \(\text{ExceptT } e \cdot m\) implements \textbf{Monad Error}

- takes as \textit{input} a monad \(m\) and
- \textit{transforms} it into a new monad \(m'\)

such that \(m'\) implements

- all the operations that \(m\) implements
- \textit{and adds} Exception-handling capabilities

\(\text{ExceptT } e \cdot m\) satisfies the constraint \((\text{MonadError } e (\text{ExceptT } e \cdot m))\)
An Exception Handler Monad with \texttt{Expr}-typed exceptions

\textbf{type} \hspace{1em} \texttt{Exn} = \texttt{ExceptT Expr Identity}

We can go back and give \texttt{evalThrowCatch} the type

\texttt{evalThrowCatch :: Expr -> Exn Int}
Composing Transformers

We can use both transformers to get both powers!

```haskell
type ExnProf a = ExceptT Expr (StateT Int (Identity)) a
```

ExnProf implements State-transformer-over Int and Exception-handling-over Expr
EXERCISE: Executing the Combined Transformer

Recall that

```plaintext
type ExnProf a = ExceptT Expr (StateT Int (Identity)) a
```

Lets write a function

```plaintext
runExnProf :: (Show a) => ExnProf a -> String
runExnProf epm = ???
```

such that

```plaintext
>>> runExnProf (eval e1)
"value: 1, count: 2"

>>> runExnProf (eval e2)
"Plus (Number 5) (Number (-5)) after 2 operations"
```
TRY AT HOME: Combining in a Different Order

We can also combine the transformers in a different order

```haskell
type ProfExn a = StateT Int (ExceptT Expr (Identity)) a
```
ExnProf implements \textit{State-transformer-over Int} and \textit{Exception-handling-over Expr}

Can you implement the function

\[ \text{runProfExn :: (Show a) => ProfExn a -> String} \]

such that when you are done, we can get the following behavior?

\[
\begin{align*}
\text{>>> runProfExn (eval e1)} \\
&\quad \text{"value: 1, count: 2"}
\end{align*}
\]

\[
\begin{align*}
\text{>>> runProfExn (eval e2)} \\
&\quad \text{"Left (Plus (Number 5) (Number (-5)))"}
\end{align*}
\]
Summary: Mixing Monads with Many Features

1. Transformers add capabilities to Monads

Transform2 (Transform1 m) implements

- m operations and
• operations added by Transform1 and
• operations added by Transform2

2. StateT and ExceptT add State and Exceptions

• Start with a basic monad Identity
• Use StateT Int to add global- Int state-update capabilities
• Use ExceptT Expr to add exception-handling capabilities

Play around with this in your homework assignment!


Generated by Hakyll (http://jaspervdj.be/hakyll), template by Armin Ronacher (http://lucumr.pocoo.org),
suggest improvements here (https://github.com/ucsd-progsys/liquidhaskell-blog/).