IA014: Advanced Functional Programming

7. Monad Transformers

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Motivation: combining monads

```haskell
data Res a = Bad Exception State | Good a State deriving Show
newtype MES a = MES { runMES :: State -> Res a }

instance Monad MES where
   return a = MES (\x -> Good a x)
   m >>= f  = MES $ \x ->
                  case runMES m x of
                      Good a s -> case runMES (f a) s of
                                      Good b s' -> Good b s'
                                      Bad  e s' -> Bad  e s'
                      Bad  e s -> Bad e s

raise_ES e = MES (\x -> Bad e x)
tick_ES    = MES (\s -> Good () (s+1))

evalMES (Div e1 e2) = do a <- evalMES e1;
                         b <- evalMES e2;
                         tick_ES;
                         if b == 0 then (raise_ES "division by 0")
                                   else return (a `div` b)
evalMES (Const i)   = return i
```

Adding state

Could not we just:

- take an existing monad `m`
- add state of type `s` to it
- so that the result would still be a monad?

```
newtype StateT s m a = s -> m (a, s)
```

- `StateT` has *kind* `* -> (* -> *) -> * -> *`
- if `m` is a monad, then `StateT s m` is a monad
- `StateT s` is a function which takes a monad, and returns a new, different monad
- such functions are called *monad transformers*

Monad transformers

- make a new monad out of an existing monad
- the old monad is *embedded* into the new one
- multiple monad transformers can be combined together
- we usually start with a base monad (e.g. `Identity`, `[]`, `IO`) and apply to it a *sequence* of monad transformers

```
StateT s Identity a = s -> Identity (a, s)
                    = s -> (a, s)
```

- applying a monad transformer to `Identity` we obtain the corresponding “simple” monad

```
type State s = StateT s Identity
```

- each monad transformer `FooT` should come with the `runFooT` operation to unwrap the transformer

```
newtype StateT s m a = StateT {runStateT :: s -> m (a, s)}
```

Onions have layers . . .

Monad transformers are like onions

Monad transformers are like onions. At first, they make you cry but then you learn to appreciate them. Like onions, they're also made of layers. Each layer is the functionality of a new monad, you lift monadic functions to get into the inner monads and you have transformerised functions to unwrap each layer.

https://www.haskell.org/haskellwiki/Monad_Transformers_Explained

The MonadTrans class

- in HASKELL, monad transformers are instances of the MonadTrans class

  ```
  class MonadTrans t where
      lift :: (Monad m) => m a -> t m a
  ```

- function lift lifts the computation from m to the constructed monad t m
- MonadTrans laws:

  ```
  lift . return = return
  lift (m >>= f) = lift m >>= (lift . f)
  ```

Example:

```
instance MonadTrans (StateT s) where
    lift m = StateT $ \ s -> do
        a <- m
        return (a, s)
```

The StateT transformer monad

```haskell
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

evalState :: State s a -> s  -> a
evalState m s = fst (runState m s)

evalStateT :: Monad m => StateT s m a -> s -> m a
evalStateT m s = liftM fst (runStateT m s)

instance Monad m => Monad (StateT s m) where
    return a = StateT (\ s -> return (a, s))
    m >>= k = StateT $ \ s -> do
        (a, s') <- runStateT m s
        runStateT (k a) s'

instance MonadTrans (StateT s) where
    lift m = StateT $ \ s -> do
        a <- m
        return (a, s)
```

StateT - modifying state

- monads modifying state can be instances of the `MonadState` class
- first benefit: easy access to the state

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

instance (Monad m) => MonadState s (StateT s m) where
    get   = StateT $ \s -> return (s, s)
    put s = StateT $ \_ -> return ((), s)
```

We also have helpful combined functions:

```haskell
modify :: (MonadState s m) => (s -> s) -> m ()
modify f = do
    s <- get
    put (f s)
```

Evaluator using StateT

```haskell
type MTS a = StateT Int Identity a

tick :: (Num s, MonadState s m) => m ()
tick = do st <- get;
          put (st+1)

evalMTS :: Exp -> MTS Int
evalMTS (Div e1 e2) = do a <- evalMTS e1;
                         b <- evalMTS e2;
                         tick;
                         return (a `div` b)
evalMTS (Const i)   = return i

runMTS s exp = runIdentity ((runStateT (evalMTS exp)) s)
```

Combining two states 1/2

```
test1 = do a <- get                      test2 = do a <- get
           modify (+1)                              modify (++"1")
           b <- get                                 b <- get
           return (a,b)                             return (a,b)

go1 = evalState test1 0
go2 = evalState test2 "0"
```

What if I want to have a combined state?

Solution 1: the new state is a pair `(Int, String)`

```
test3 = do (a1,a2) <- get
           modify (\x -> ((fst x)+1, (snd x)++"1"))
           (b1,b2) <- get
           return ((a1,b1),(a2,b2))

go3 = evalState test3 (0,"0")
```

Combining two states 2/2

Solution 2: Use *two* `StateT` transformers applied to `Identity`

```
test4 = do modify (+ 1)
           lift $ modify (++ "1")
           a <- get
           b <- lift get
           return (a,b)

go4 = runIdentity $ evalStateT (evalStateT test4 0) "0"
```

"Heavy lifting"

- ordinary monad “commands” talk to the outer monad
- `lift` “sends” command one layer inwards
- multiple `lift` calls can be chained together

Choosing the “base” monad

- `Identity` does not have to be the innermost monad
- another version of *Solution 2*:

```
test5 = do modify (+ 1)
           lift $ modify (++ "1")
           a <- get
           b <- lift get
           return (a,b)

go5 = evalState (evalStateT test5 0) "0"
```

- here the `StateT` transformer is applied to `State`

Combining `State` and `IO`

- `IO` is the (only) “magic” monad
- therefore `IO` must be the *innermost* monad

```haskell
test6 = do modify (+ 1)
           a <- get
           lift (print a)
           modify (+ 1)
           b <- get
           lift (print b)

go6 = evalStateT test6 0
```

```
*Main> :t test6
test6 :: StateT Integer IO ()
```

Transformer versions of standard monads

Standard Monad	Transformer Version	Original Type	Combined Type
`Error`	`ErrorT`	`Either e a`	`m (Either e a)`
`State`	`StateT`	`s -> (a,s)`	`s -> m (a,s)`
`Reader`	`ReaderT`	`r -> a`	`r -> m a`
`Writer`	`WriterT`	`(a,w)`	`m (a,w)`
`Cont`	`ContT`	`(a -> r) -> r`	`(a -> m r) -> m r`

Evaluator using ErrorT

```
type MTE a = ErrorT String Identity a

evalMTE :: Exp -> MTE Int
evalMTE (Div e1 e2) = do a <- evalMTE e1;
                         b <- evalMTE e2;
                         if b == 0 then (throwError "division by 0")
                                   else return (a `div` b)
evalMTE (Const i)   = return i

runMTE exp = runIdentity (runErrorT (evalMTE exp))
```

Evaluator with state and error handling

```haskell
type MTES a = ErrorT String (StateT Int Identity) a

tick :: (Num s, MonadState s m) => m ()
tick = do st <- get;
          put (st+1)

evalMTES :: Exp -> MTES Int
evalMTES (Div e1 e2) = do a <- evalMTES e1;
                          b <- evalMTES e2;
                          tick;
                          if b == 0 then (throwError "division by 0")
                                    else return (a `div` b)
evalMTES (Const i) = return i

runMTES s exp = runIdentity (runStateT (runErrorT (evalMTES exp)) s)
```

Where are the `lift` functions?

"Missing" `lift` functions

- `StateT` has `put` and `get` defined, because it is an instance of the `MonadState` class

```
instance (Monad m) => MonadState s (StateT s m) where
    get   = StateT $ \s -> return (s, s)
    put s = StateT $ \_ -> return ((), s)
```

- but so is `ErrorT`, if the inner monad is an instance of `MonadState:`

```
instance (Error e, MonadState s m) => MonadState s (ErrorT e m)
                                                              where
    get = lift get
    put = lift . put
```

- the HASKELL `mtl` library defines this “invisible lifting” for the standard transformers (and their respective classes)

Order matters

- we want *both* error handling and state
- combined monad which is an instance of both `MonadState` and `MonadError`

possibilities:

1. apply `StateT` to `Error`
   - result: state transformer function `s -> Error (a, s)`
   - error means no state can be produced
2. apply `ErrorT` to `State`
   - result: state transformer function `s -> (Error e a, s)`
   - error means no value can be produced
   - state remains valid

Reading

- S. Liang, P. Hudak, M. Jones: *Monad Transformers and Modular Interpreters.* POPL'96.
- M. Grabmüller: *Monad Transformers Step by Step.* `http://www.grabmueller.de/martin/www/pub/Transformers.en.html`