Functional programming, Seminar No. 8

Daniel Rogozin
Lomonosov Moscow State University,
Serokell OÜ
Higher School of Economics
The Department of Computer Science

Intro

On the previous seminar we

- studied such monads as `IO`, `Reader`, `Writer`, and `State`

Intro

On the previous seminar we

- studied such monads as `IO`, `Reader`, `Writer`, and `State`

Today we

- investigate monad transformers as an uniform method of the effect combining

Alternative and MonadPlus classes

Monoids

```haskell
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

Some of monads are also monoids, e.g., the list data type

```haskell
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

Two versions of the `Maybe` monoid

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Just memty
  Nothing `mappend` _ = Nothing
  _ `mappend` Nothing = Nothing
  (Just a) `mappend` (Just b) = Just (a `mappend` b)

instance Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m@(Just _) `mappend` _ = m
```

The `Alternative` class

The `Alternative` class is a generalisation of the idea above:

```
  class Applicative f => Alternative (f :: * -> *) where
    empty :: f a
    (<|>) :: f a -> f a -> f a
    some :: f a -> f [a]
    many :: f a -> f [a]
{-# MINIMAL empty, (<|>) #-}

infixl 3 <|>
```

The `MonadPlus` class

The `Alternative` class has an essential extension called `MonadPlus`:

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

This class should satisfy the following conditions:

```
mzero >>= f   ==   mzero
v >> mzero    ==   mzero
```

The MonadPlus uses

```haskell
mfilter :: (MonadPlus m) => (a -> Bool) -> m a -> m a
mfilter p ma = do
  a <- ma
  if p a then return a else mzero

guard :: Alternative f => Bool -> f ()
guard True = pure ()
guard False =  empty

when :: Applicative f => Bool -> f () -> f ()
when p s  = if p then s else pure ()
```

Monad transformers

How to compose Reader and Writer

```
foo :: RWS Int [Int] () Int
foo i = do
  baseCounter <- ask
  let newCounter = baseCounter + i
  put [baseCounter, newCounter]
  return newCounter

  foo :: State (Int, [Int]) Int
  foo i = do
    x <- gets fst
    let xi = x + i
    put (x, [x, xi])
    return xi
```

Maybe and IO

The example of a monad composition is the `MaybeIO` monad

```haskell
newtype MaybeIO a = MaybeIO { runMaybeIO :: IO (Maybe a) }

instance Monad MaybeIO where
  return x = MaybeIO (return (Just x))
  MaybeIO action >>= f = MaybeIO $ do
    result <- action
      case result of
        Nothing -> return Nothing
        Just x  -> runMaybeIO (f x)
```

The generalisation of the idea above

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance Monad m => Monad (MaybeT m) where
  return :: a -> MaybeT m a
  return x = MaybeT (return (Just x))

  (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
  MaybeT action >>= f = MaybeT $ do
    result <- action
    case result of
      Nothing -> return Nothing
      Just x  -> runMaybeT (f x)
```

The `MonadTrans` class

```haskell
class MonadTrans (t :: (* -> *) -> * -> *) where
  -- | Lift a computation from
  -- | the argument monad to the constructed monad.
  lift :: (Monad m) => m a -> t m a
```

This class has the following laws:

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

The MonadTrans instances

```haskell
transformToMaybeT :: Functor m => m a -> MaybeT m a
transformToMaybeT = error "homework"

instance MonadTrans MaybeT where
  lift :: Monad m => m a -> MaybeT m a
  lift = transformToMaybeT
```

The MaybeT example

```haskell
emailIsValid :: String -> Bool
emailIsValid email = '@' `elem` email

askEmail :: MaybeT IO String
askEmail = do
  lift $ putStrLn "Input your email, please:"
  email <- lift getLine
  guard $ emailIsValid email
  return email

main :: IO ()
main = do
  email <- askEmail
  case email of
    Nothing -> putStrLn "Wrong email."
    Just email' -> putStrLn email'
```

The ReaderT monad

```haskell
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

type LoggerIO a = ReaderT LoggerName IO a

logMessage :: Text -> LoggerIO ()

readFileWithLog :: FilePath -> LoggerIO Text
readFileWithLog path = do
  logMessage $ "Reading file: " <> T.pack (show path)
  lift $ readFile path
```

The ReaderT monad

```
writeFileWithLog :: FilePath -> Text -> LoggerIO ()
writeFileWithLog path content = do
    logMessage $ "Writing to file: " <> T.pack (show path)
    lift $ writeFile path content

prettifyFileContent :: FilePath -> LoggerIO ()
prettifyFileContent path = do
  content <- readFileWithLog path
  writeFileWithLog path (format content)

main :: IO ()
main =
  runReaderT
    (prettifyFileContent "foo.txt")
    (LoggerName "Application")
```

The MonadReader class

The class is defined the mtl-package

```
class Monad m => MonadReader r m | m -> r where
  ask    :: m r
  local  :: (r -> r) -> m a -> m a
  reader :: (r -> a) -> m a

instance MonadReader r m => MonadReader r (StateT s m) where
  ask    = lift ask
  local  = mapStateT . local
  reader = lift . reader
```

The MonadError class

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

newtype ExceptT e m a =
  ExceptT { runExceptT :: m (Either e a) }

runExceptT :: ExceptT e m a -> m (Either e a)

withExceptT ::
  Functor m => (e -> e') -> ExceptT e m a -> ExceptT e' m a
```

The MonadError class

```haskell
foo :: MonadError FooError m => ...
bar :: MonadError BarError m => ...
baz :: MonadError BazError m => ...

data BazError = BazFoo FooError | BazBar BarError

baz = do
  withExcept BazFoo foo
  withExcept BazBar ba
```