Monad Error

Daniel Reigada

Codacy

August 16, 2018

Monad error

A typeclass defined that allows one to abstract over error-handling monads.

Monad error

A typeclass defined that allows one to abstract over error-handling monads.

But what does that mean?

Monad error

It allows the creation of generic code, that would otherwise return errors wrapped in a specific monad.

Monad error

It allows the creation of generic code, that would otherwise return errors wrapped in a specific monad.

OKAY! But what does that mean?!?

Monad error - Abstracting the monad

```scala
def divide(num: Int, denom: Int): Int = num / denom
```

Monad error - Abstracting the monad

What if *denom is* 0 ?

```
def divide(num: Int, denom: Int): Int = num / denom
```

Monad error - Abstracting the monad

We can improve this using Option...

```
def divide(num: Int , denom: Int ): Option [Int ] =
  if(denom == 0) None else Some(num / denom)
```

Monad error - Abstracting the monad

...or Try...

```
def divideTry(num: Int , denom: Int ): Try [Int ] =
  if (denom == 0) Failure(new Throwable("Division by 0"))
  else Success(num / denom)
```

Monad error - Abstracting the monad

...or Future...

```
def divideFuture(num: Int , denom: Int ): Future [Int ] =
  if (denom == 0) Future.failed(new Throwable("Division by 0"))
  else Future.successful(num / denom)
```

Monad error - Abstracting the monad

...or Either...

```scala
def divideEither(num: Int, denom: Int): Either[String, Int] =
  if (denom == 0) Left("Division by 0")
  else Right(num / denom)
```

Monad error - Abstracting the monad

...or a custom result type (i.e. foundation Response)...

```
def divideEither(num: Int , denom: Int ): Result [Int ] =
  if (denom == 0) Result.error("Division by 0")
  else Result.success(num / denom)
```

Monad error - Abstracting the monad

... you get the idea.

What if you are trying to write generic code (i.e. library)?
We should be able to abstract it as much as possible.

```
def divide???(num: Int , denom: Int ): F [Int ] =
  if (denom == 0) ERROR_CASE
  else SUCCESS_CASE
```

Monad error - Abstracting the monad

By the power of the Monad!!!

Monad error - Abstracting the monad

```scala
def divideF[F[_]](num: Int, denom: Int)(
    implicit M: MonadError[F, Throwable]): F[Int] = {
  if (denom == 0) M.raiseError(new Throwable("Division by 0"))
  else M.pure(num / denom)
}
```

Monad error - Abstracting the monad

```
def getNum: Try[Int] = ???
def getDenom: Try[Int] = ???

for {
  num <- getNum
  denom <- getDenom
  result <- divideF[Try](num, denom)
} yield result
```

Monad error - Abstracting the monad

Monad error - Useful methods

```
// pure
MonadError[Try, Throwable].pure(1)
// scala.util.Try[Int] = Success(1)

// raiseError
MonadError[Try, Throwable].raiseError(new Throwable("error"))
// scala.util.Try[Nothing] = Failure(java.lang.Throwable: error)
```

Monad error - Useful methods

```scala
// fromEither / fromTry / fromOption / fromValidated

MonadError[Try, Throwable].fromEither(Right(123))
// scala.util.Try[Int] = Success(123)

MonadError[Try, Throwable].fromOption(None, new Throwable("empty"))
// scala.util.Try[Nothing] = Failure(java.lang.Throwable: empty)
```

Monad error - Useful methods

```
// catchNonFatal
MonadError[Try, Throwable].catchNonFatal(1 / 0)
// scala.util.Try[Int] =
//    Failure(java.lang.ArithmeticException: / by zero)
```

Monad error - Useful methods

```scala
// handleError / handleErrorWith / recover / recoverWith
MonadError[Try, Throwable]
  .catchNonFatal(1 / 0)
  .recover {
    case _ : ArithmeticException => 0
  }
// scala.util.Try[Int] = Success(0)
```

Monad error - Useful methods

```scala
MonadError[Try, Throwable]
  .pure(123)
  .attempt
// scala.util.Try[Either[Throwable,Int]] =
//    Success(Right(123))

MonadError[Try, Throwable]
  .raiseError(new Throwable("error"))
  .attempt
// scala.util.Try[Either[Throwable,Nothing]] =
//    Success(Left(java.lang.Throwable: error))
```

Monad error - Abstracting the error

Our code is still not totally generic.
We are restricted by the error type.

Monad error - Abstracting the error

```scala
trait UIError[A] {
  def errorFromString(str: String): A

  def errorFromThrowable(thr: Throwable): A
}

object UIError {
  implicit val throwableInstance: UIError[Throwable] = ??
  implicit val stringInstance: UIError[String] = ???
}
```

Monad error - Abstracting the error

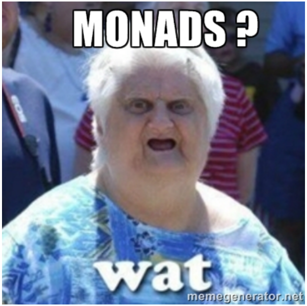
```scala
def divideF[F[_], E](num: Int, denom: Int)(
  implicit M: MonadError[F, E], Err: UIError[E]): F[Int] = {
  if (denom == 0) M.raiseError(Err.errorFromString("Division by 0"))
  else M.pure(num / denom)
}

divideF[Try, Throwable](1, 0)
// scala.util.Try[Int] = Failure(java.lang.Throwable: Division by 0)

divideF[Either[String, ?], String](1, 0)
// scala.util.Either[String,Int] = Left(Division by 0)
```

Questionary time

Which one do you feel like?



(a)

(b)

Further reading

- Documentation - https://typelevel.org/cats/api/cats/MonadError.html
- Haskell docs (if you are brave enough) - http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Error.html
- Rethinking MonadError - https://typelevel.org/blog/2018/04/13/rethinking-monaderror.html

Q & A

or invoice-manager demo if we have time