

Chapter 11: Computations in a functor context III. Monad transformers

Sergei Winitzki

Academy by the Bay

2019-01-05

Computations within a functor context: Combining monads

Programs often need to combine monadic effects

- “Effect” \equiv what else happens in $A \Rightarrow M^B$ besides computing B from A
- Examples of effects for some standard monads:
 - ▶ **Option** – computation will have no result or a single result
 - ▶ **List** – computation will have zero, one, or multiple results
 - ▶ **Either** – computation may fail to obtain its result, reports error
 - ▶ **Reader** – computation needs to read an external context value
 - ▶ **Writer** – some value will be appended to a (monoidal) accumulator
 - ▶ **Future** – computation will be scheduled to run later
- How to combine several effects in the same functor block (**for/yield**)?

```
// This is not valid Scala!           // This is not valid Scala!
val result = for { i ← 1 to n          (1 to n).flatMap { i ⇒
    j ← Future { q(i) }                Future(q(i)).flatMap { j ⇒
    k ← maybeError(j) : Try[Int]        maybeError(j).map { k ⇒
} yield f(k)                           f(k)
// What should be the type of result??   }}
```

- The code will work if we “unify” all effects in a new, larger monad
- Need to compute the type of new monad that contains all given effects

Combining monadic effects I. Trial and error

There are several ways of combining two monads into a new monad:

- If M_1^A and M_2^A are monads then $M_1^A \times M_2^A$ is also a monad
 - ▶ But $M_1^A \times M_2^A$ describes two separate values with two separate effects
- If M_1^A and M_2^A are monads then $M_1^A + M_2^A$ is usually not a monad
 - ▶ If it worked, it would be a choice between two different values / effects
- If M_1^A and M_2^A are monads then one of $M_1^{M_2^A}$ or $M_2^{M_1^A}$ is often a monad
- Examples and counterexamples for functor composition:
 - ▶ Combine `Future[A]` and `Option[A]` as `Future[Option[A]]`
 - ▶ Combine $Z \Rightarrow A$ and List^A as $Z \Rightarrow \text{List}^A$
 - ▶ But `Either[Z, Future[A]]` and `Option[Z \Rightarrow A]` are not monads
 - ▶ Neither `Future[State[A]]` nor `State[Future[A]]` are monads
- The order of effects matters when composition works both ways:
 - ▶ Combine `Either` ($M_1^A = Z + A$) and `Writer` ($M_2^A = W \times A$)
 - ★ as $Z + W \times A$ – either compute result and write a message, or all fails
 - ★ as $(Z + A) \times W$ – message is always written, but computation may fail
- Find a general way of defining a new monad with combined effects
- Derive properties required for the new monad

Combining monadic effects II. Lifting into a larger monad

If a “big monad” `BigM[A]` somehow combines all the needed effects:

```
// This could be valid Scala.      // If we define the various
val result: BigM[Int] = for {      // required “lifting” functions:
  i ← lift1(1 to n)                def lift1[A]: Seq[A] ⇒ BigM[A] = ???
  j ← lift2(Future{ q(i) })        def lift2[A]: Future[A] ⇒ BigM[A] = ???
  k ← lift3(maybeError(j))        def lift3[A]: Try[A] ⇒ BigM[A] = ???
} yield f(k)
```

- Example 1: combining as `BigM[A] = Future[Option[A]]` with liftings:

```
def lift1[A]: Option[A] ⇒ Future[Option[A]] = Future.successful(_)
def lift2[A]: Future[A] ⇒ Future[Option[A]] = _.map(Some.apply)
```

- Example 2: combining as `BigM[A] = List[Try[A]]` with liftings:

```
def lift1[A]: Try[A] ⇒ List[Try[A]] = x ⇒ List(x)
def lift2[A]: List[A] ⇒ List[Try[A]] = _.map(x ⇒ Success(x))
```

Remains to be understood:

- How to build a “big monad” out of “smaller” ones, providing liftings
 - ▶ Is this always possible? Are there alternatives?
- What are the required laws for liftings
- Ways of reducing the complexity of code; make liftings implicit

- 1 Show that $M_1^A + M_2^A$ is not a monad when $M_1^A \equiv 1 + A$ and $M_2^A \equiv Z \Rightarrow A$.