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" ≡ 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`)?

```scala
// 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 $Z \Rightarrow A$ and List^A as $Z \Rightarrow \text{List}^A$
  - Combine `Future[A]` and `Option[A]` as `Future[Option[A]]`
  - But `Either[Z, Future[A]]` and `Option[Z ⇒ 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:

```scala
// 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:
  ```scala
  def lift₁[A]: Option[A] ⇒ Future[Option[A]] = Future.successful(_)
  def lift₂[A]: Future[A] ⇒ Future[Option[A]] = _.map(x ⇒ Some(x))
  ```
- Example 2: combining as `BigM[A] = List[Try[A]]` with liftings:
  ```scala
  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:

- Finding suitable laws for the liftings; checking that the laws hold
- Building a "big monad" out of "smaller" ones, with lawful liftings
  - ▶ Is this always possible? Unique? Are there alternative solutions?
- Ways of reducing the complexity of code; make liftings automatic

Laws for monad liftings I. Identity laws

Whatever identities we expect to hold for monadic programs must continue to hold after lifting `M₁` or `M₂` values into the "big monad" `BigM`

- We assume that `M₁`, `M₂`, and `BigM` already satisfy all the monad laws

Consider the various functor block constructions containing the liftings:

- Left identity law after `lift₁`

```
// Anywhere inside a for/yield:          // Must be equivalent to...
i ← lift₁(M₁.pure(x))                    i = x
j ← bigM(i) // Any BigM value.           j ← bigM(x)
```

`lift₁(M₁.pure(x)).flatMap(b) = b(x)` — in terms of Kleisli composition ($\diamond$):

$$\left(\mathsf{pure}_{M_1} \,\mathring{,}\, \mathsf{lift}_1\right)^{:X \Rightarrow \mathsf{BigM}^X} \diamond b^{:X \Rightarrow \mathsf{BigM}^Y} = b \text{ with } f^{:X \Rightarrow M^Y} \diamond g^{:Y \Rightarrow M^Z} \equiv x \Rightarrow f(x).\mathsf{flatMap}(g)$$

- Right identity law after `lift₁`

```
// Anywhere inside a for/yield:          // Must be equivalent to...
x ← bigM // Any BigM value.              x ← bigM
i ← lift₁(M₁.pure(x))                    i = x
```

`b.flatMap(M₁.pure andThen lift₁) = b` — in terms of Kleisli composition:

$$b^{:X \Rightarrow \mathsf{BigM}^Y} \diamond \left(\mathsf{pure}_{M_1} \,\mathring{,}\, \mathsf{lift}_1\right)^{:Y \Rightarrow \mathsf{BigM}^Y} = b$$

- The same identity laws must hold for `M₂` and `lift₂` as well

Laws for monad liftings II. Simplifying the laws

$\big(\mathsf{pure}_{M_1}\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1\big)$ is a unit for the Kleisli composition $\diamond$ in the monad `BigM`

- But the monad `BigM` already has a unit element: $\mathsf{pure}_{\mathsf{BigM}}$
- The two-sided unit element is always unique: $\mathsf{id} = \mathsf{id} \diamond \mathsf{id}' = \mathsf{id}'$
- So the two identity laws for $\big(\mathsf{pure}_{M_1}\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1\big)$ can be reduced to one law:

$$\mathsf{pure}_{M_1}\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1 = \mathsf{pure}_{\mathsf{BigM}}$$

Refactoring a portion of a monadic program under `lift`₁ gives another law:

```
// Anywhere inside a for/yield:        // Must be equivalent to...
i ← lift₁(p) // Any M₁ value.          pq = p.flatMap(q) // In M₁.
j ← lift₁(q(i)) // Any M₁ value.       j ← lift₁(pq) // Now lift it.
```
```
lift₁(p).flatMap(q andThen lift₁) = lift₁(p flatMap q)
```

- Rewritten equivalently through $\mathsf{flm}_M : \big(A \Rightarrow M^B\big) \Rightarrow M^A \Rightarrow M^B$ as

$$\mathsf{lift}_1\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{flm}_{\mathsf{BigM}}\,(q\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1) = \mathsf{flm}_{M_1}\,q\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1$$

- Rewritten in terms of Kleisli composition:

$$\big(b^{:X \Rightarrow M_1^Y}\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1\big) \diamond \big(c^{:Y \Rightarrow M_1^Z}\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1\big) = (b \diamond c)\,\mathbin{\raise1pt\hbox{$\circ$}}\,\mathsf{lift}_1$$

- Liftings lift_1 and lift_2 must obey an identity law and a composition law
- The laws say that the liftings **commute with** the monads' operations

Laws for monad liftings III. The naturality law

Show that $\mathrm{lift}_1 : M_1^A \Rightarrow \mathrm{BigM}^A$ is a natural transformation

- It maps pure_{M_1} to $\mathrm{pure}_{\mathrm{BigM}}$ and flm_{M_1} to $\mathrm{flm}_{\mathrm{BigM}}$
  - lift_1 is a **monadic morphism** between monads $M_1^\bullet$ and $\mathrm{BigM}^\bullet$

The (functor) naturality law:

$$\mathrm{lift}_1 \,\mathring{\,}\, \mathrm{fmap}_B\, f^{:X \Rightarrow Y} = \mathrm{fmap}_{M_1}\, f^{:X \Rightarrow Y} \,\mathring{\,}\, \mathrm{lift}_1$$

$$
\begin{array}{ccc}
M_1^X & \xrightarrow{\ \ \mathrm{lift_1}\ \ } & \mathrm{BigM}^X \\
{\scriptstyle \mathrm{fmap}_{M_1}\ f^{:X \Rightarrow Y}} \downarrow & & \downarrow {\scriptstyle \mathrm{fmap}_{\mathbf{BigM}}\ f^{:X \Rightarrow Y}} \\
M_1^Y & \xrightarrow{\ \ \mathrm{lift_1}\ \ } & \mathrm{BigM}^Y
\end{array}
$$

Derivation of the naturality law:

- Express fmap as $\mathrm{fmap}_M\, f = \mathrm{flm}_M\, (f \,\mathring{\,}\, \mathrm{pure}_M)$ for both monads
- Given $f^{:X \Rightarrow Y}$, use the law $\mathrm{flm}_{M_1}\, q \,\mathring{\,}\, \mathrm{lift}_1 = \mathrm{lift}_1 \,\mathring{\,}\, \mathrm{flm}_{\mathrm{BigM}}\, (q \,\mathring{\,}\, \mathrm{lift}_1)$ to compute
  $\mathrm{flm}_{M_1}\, (f \,\mathring{\,}\, \mathrm{pure}_{M_1}) \,\mathring{\,}\, \mathrm{lift}_1 = \mathrm{lift}_1 \,\mathring{\,}\, \mathrm{flm}\, (f \,\mathring{\,}\, \mathrm{pure}_{M_1} \,\mathring{\,}\, \mathrm{lift}_1) = \mathrm{lift}_1 \,\mathring{\,}\, \mathrm{flm}\, (f \,\mathring{\,}\, \mathrm{pure}_{\mathrm{BigM}}) =$
  $\mathrm{lift}_1 \,\mathring{\,}\, \mathrm{fmap}_{\mathrm{BigM}}\, f$

A monadic morphism is always also a natural transformation of the functors

Monad transformers I

- Combine $Z \Rightarrow A$ and $1 + A$: only $Z \Rightarrow 1 + A$ works, not $1 + (Z \Rightarrow A)$
  - It is not possible to combine monads via a natural bifunctor B^{M_1, M_2}
  - It is not possible to combine arbitrary monads as $M_1^{M_2^\bullet}$ or $M_2^{M_1^\bullet}$
- But we can fix $M_1^\bullet$ (base monad) and to let $M_2^\bullet$ (foreign monad) vary
- The result is a monad transformer $T_{M_1}^{M_2, A}$ – a natural functor in M_2

A **monad transformer** for a **base** monad $L^\bullet$ is a type constructor $T_L^{M, \bullet}$ parameterized by a monad $M^\bullet$, such that for all monads $M^\bullet$

- $T_L^{M, \bullet}$ is a monad (the monad M **transformed with** T_L)
- "Lifting" – a monadic morphism $\text{lift}_L^M : M^A \rightsquigarrow T_L^{M, A}$, natural in $M^\bullet$
- "Injection" – a monadic morphism $\text{inj} : L^A \rightsquigarrow T_L^{M, A}$
- $T_L^{M, \bullet}$ is **monadically natural** in $M^\bullet$
  - $T_L^{M, \bullet}$ is natural w.r.t. a monadic functor $M^\bullet$ as a type parameter
  - For any monad $N^\bullet$ and a monadic morphism $f : M^\bullet \rightsquigarrow N^\bullet$ we need to have a monadic morphism $T_L^{M, \bullet} \rightsquigarrow T_L^{N, \bullet}$ for the transformed monads
  - Naturality will hold if we implement $T_L^{M, \bullet}$ only via M's monad methods
  - Cf. `traverse`: $L^A \Rightarrow (A \Rightarrow F^B) \Rightarrow F^{L^B}$ – natural w.r.t. applicative $F^\bullet$

Exercises

1. Show that the method `pure`: $A \Rightarrow M^A$ is a monadic morphism between monads $\text{Id}^A \equiv A$ and M^A.

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