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!
val result = for { i ← 1 to n
   j ← Future { q(i) }
   k ← maybeError(j) : Try[Int]
} yield f(k)
// What should be the type of result??
```

```
// This is not valid Scala!
(1 to n).flatMap { i ⇒
   Future(q(i)).flatMap { j ⇒
     maybeError(j).map { k ⇒
       f(k)
         }}}
```

- 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:

```
// This could be valid Scala...
val result: BigM[Int] = for {
  i ← lift₁(1 to n)
  j ← lift₂(Future{ q(i) })
  k ← lift₃(maybeError(j))
} yield f(k)
```

```
// If we define the various
// required "lifting" functions:
def lift₁[A]: Seq[A] ⇒ BigM[A] = ???
def lift₂[A]: Future[A] ⇒ BigM[A] = ???
def lift₃[A]: Try[A] ⇒ BigM[A] = ???
```

- 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(x ⇒ Some(x))
```

- 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:

- 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 ($\diamond$):

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

- 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:

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

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

Exercises

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