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`)?

```scala
// 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??
```

```scala
// 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 ($\diamond$):

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

- 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:

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

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

Laws for monad liftings II. Simplifying the laws

$\left(\text{pure}_{M_1} ; \text{lift}_1\right)$ is a unit for the Kleisli composition $\diamond$ in the monad `BigM`

- But the monad `BigM` already has a unit element: $\text{pure}_{\text{BigM}}$
- The two-sided unit element is always unique: $\text{id} = \text{id} \diamond \text{id}' = \text{id}'$
- So the two identity laws for $\left(\text{pure}_{M_1} ; \text{lift}_1\right)$ can be reduced to one law:

$$\text{pure}_{M_1} ; \text{lift}_1 = \text{pure}_{\text{BigM}}$$

Refactoring a portion of a monadic program under `lift₁` gives another law:

```
// Anywhere inside a for/yield:
i ← lift₁(p) // Any M₁ value.
j ← lift₁(q(i)) // Any M₁ value.
```

```
// Must be equivalent to...
pq = p.flatMap(q) // In M₁.
j ← lift₁(pq) // Now lift it.
```

```
lift₁(p).flatMap(q andThen lift₁) = lift₁(p flatMap q)
```

- Rewritten equivalently through $\text{flm}_M : \left(A \Rightarrow M^B\right) \Rightarrow M^A \Rightarrow M^B$ as

$$\text{lift}_1 ; \text{flm}_{\text{BigM}} \left(q ; \text{lift}_1\right) = \text{flm}_{M_1} q ; \text{lift}_1$$

- Rewritten in terms of Kleisli composition:

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

- 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 $\text{lift}_1 : M_1^A \Rightarrow \text{BigM}^A$ is a natural transformation

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

The (functor) naturality law:

$$\text{lift}_1 ⨾ \text{fmap}_B f^{:X\Rightarrow Y} = \text{fmap}_{M_1} f^{:X\Rightarrow Y} ⨾ \text{lift}_1$$

$$\begin{tikzcd}
M_1^X \arrow[r, "\text{lift}_1"] \arrow[d, "\text{fmap}_{M_1} f^{:X\Rightarrow Y}"'] & \text{BigM}^X \arrow[d, "\text{fmap}_{\text{BigM}} f^{:X\Rightarrow Y}"] \\
M_1^Y \arrow[r, "\text{lift}_1"] & \text{BigM}^Y
\end{tikzcd}$$

Derivation of the naturality law:

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

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

Monad transformers I: Motivation

- 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 $M_2^{M_1^\bullet}$
- The trick: for a fixed **base** monad $L^\bullet$, let $M^\bullet$ (**foreign** monad) vary
- Call the desired result the "L's monad transformer", $T_L^{M,\bullet}$
  - In Scala: `LT[M[_]: Monad, A]` – e.g. `ReaderT`, `StateT`, etc.
- $T_L^{M,\bullet}$ is generic in M but not in L
  - No general formula for monad transformers seems to exist
  - For each base monad L, a different construction is needed
  - Some monads L do not seem to have a transformer!
- To combine 3 or more monads, compose the transformers: $T_{L_1}^{T_{L_2}^{M,\bullet}}$
  - Example in Scala: `StateT[S, ListT[Reader[R, ?], ?], A]`
- This is called a "monad stack" – but may not be functor composition
  - because e.g. `State[S, List[Reader[R, A]]]` is not a monad

Monad transformers II: The requirements

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 $\text{lift}_L^M : M^A \leadsto T_L^{M,A}$, natural in $M^\bullet$
- "Base lifting" – a monadic morphism $\text{blift} : L^A \leadsto T_L^{M,A}$
  - The "base lifting" could not possibly be natural in $L^\bullet$
- Transformed identity monad (Id) must be L, i.e. $T_L^{\text{Id},\bullet} \cong L^\bullet$
- $T_L^{M,\bullet}$ is **monadically natural** in $M^\bullet$ (but not in $L^\bullet$)
  - $T_L^{M,\bullet}$ is natural w.r.t. a monadic functor $M^\bullet$ as a type parameter
  - For any monad $N^\bullet$ and a monadic morphism $f : M^\bullet \leadsto N^\bullet$ we need to have a monadic morphism $T_L^{M,\bullet} \leadsto T_L^{N,\bullet}$ for the transformed monads
    - If we implement $T_L^{M,\bullet}$ only via M's monad methods, naturality will hold
  - Cf. `traverse`: $L^A \Rightarrow \left(A \Rightarrow F^B\right) \Rightarrow F^{L^B}$ – natural w.r.t. applicative $F^\bullet$
  - This is needed for lifting a "runner" $M^A \leadsto A$ to $T_L^{M,\bullet} \leadsto T_L^{\text{Id},\bullet} = L^\bullet$
- "Base runner": a monadic morphism $T_L^{M,\bullet} \leadsto M^\bullet$

Monad transformers III: First examples

Recall these monad constructions:

- If M^A is a monad then $R \Rightarrow M^A$ is also a monad (for a fixed type R)
- If M^A is a monad then $M^{Z+A\times W}$ is also a monad (for fixed W, Z)

This gives the monad transformers for base monads `Reader`, `Writer`, `Either`:

```
type ReaderT[R, M[_], A] = R ⇒ M[A]
type EitherT[Z, M[_], A] = M[Either[Z, A]]
type WriterT[W, M[_], A] = M[(W, A)]
```

- `ReaderT` wraps the foreign monad from the outside
- `EitherT` and `WriterT` require the foreign monad to wrap *them* outside

Remaining questions:

- What are transformers for other standard monads (`List`, `State`, `Cont`)?
  - ...in fact, these monads do not compose as either "inside" or "outside"!
- How to derive a monad transformer for an arbitrary given monad?
  - For monads obtained via known monad constructions?
  - For monads constructed via other monad transformers?
  - Is it always possible? (Probably not.)
- For a given monad, is the corresponding monad transformer unique?
- How to avoid lifting boilerplate? (mtl-style transformers)

Monad transformers IV: The zoology of monads

Need to select the correct monad transformer construction, per monad:

- "Inside" transformers: base monad inside foreign monad, $T_L^{M,A} = M^{L^A}$
  - Examples: `OptionT`, `WriterT`, `EitherT`
- "Outside" transformers: base monad is outside, $T_L^{M,A} = L^{M^A}$
  - Examples: `ReaderT`; `SearchT` for search monad `S[A] = (A ⇒ Z) ⇒ A`
  - More generally: all rigid monads have "outside" transformers
    - ★ Definition: a **rigid monad** has the method `fuseIn`: $(A \Rightarrow R^B) \Rightarrow R^{A \Rightarrow B}$
- "Recursive": interleaves the base monad and the foreign monad
  - Examples: `ListT`, `FreeMonadT`
- "Irregular": none of the above constructions apply
  - Examples: `StateT`, `ContT`, "codensity monads" (no natural transformers)
- Monad constructions: making a transformer for new monads
  - Product $M^A \times N^A$
  - Free pointed $A + M^A$ – has no transformers

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 $M_2^A \equiv Z \Rightarrow A$.