

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 $Z \Rightarrow A$ 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 \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(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 M_1 or M_2 values into the “big monad” BigM

- We assume that M_1 , M_2 , and BigM already satisfy all the monad laws

Consider the various functor block constructions containing the liftings:

- Left identity law after lift_1

// Anywhere inside a for/yield:	// Must be equivalent to...
$i \leftarrow \text{lift}_1(M_1.\text{pure}(x))$	$i = x$
$j \leftarrow \text{bigM}(i)$ // Any BigM value.	$j \leftarrow \text{bigM}(x)$

$\text{lift}_1(M_1.\text{pure}(x)).\text{flatMap}(b) = b(x)$ — in terms of Kleisli composition (\diamond):
 $(\text{pure}_{M_1} \circ \text{lift}_1)^{X \Rightarrow \text{BigM}^Y} \diamond b^{X \Rightarrow \text{BigM}^Y} = b$ with $f^{X \Rightarrow M^Y} \diamond g^{Y \Rightarrow M^Z} \equiv x \Rightarrow f(x).\text{flatMap}(g)$

- Right identity law after lift_1

// Anywhere inside a for/yield:	// Must be equivalent to...
$x \leftarrow \text{bigM}$ // Any BigM value.	$x \leftarrow \text{bigM}$
$i \leftarrow \text{lift}_1(M_1.\text{pure}(x))$	$i = x$

$b.\text{flatMap}(M_1.\text{pure} \text{ andThen } \text{lift}_1) = b$ — in terms of Kleisli composition:

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

- The same identity laws must hold for M_2 and lift_2 as well

Laws for monad liftings II. Simplifying the laws

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

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

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

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

// Anywhere inside a for/yield, this...	// must be equivalent to...
<code>i ← lift₁(p)</code> // Any M_1 value.	<code>pq = p.flatMap(q)</code> // In M_1 .
<code>j ← lift₁(q(i))</code> // Any M_1 value.	<code>j ← lift₁(pq)</code> // Now lift it.

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

- Rewritten equivalently through $\text{flm}_M : (A \Rightarrow M^B) \Rightarrow M^A \Rightarrow M^B$ as $\text{lift}_1 \circ \text{flm}_{\text{BigM}}(q \circ \text{lift}_1) = \text{flm}_{M_1} q \circ \text{lift}_1$ – both sides are functions $M_1^A \Rightarrow \text{BigM}^B$

- Rewritten equivalently through $\text{ftn}_M : M^{M^A} \Rightarrow M^A$ as

$\text{lift}_1 \circ \text{fmap}_{\text{BigM}} \text{lift}_1 \circ \text{ftn}_{\text{BigM}} = \text{ftn}_{M_1} \circ \text{lift}_1$ – both sides are functions $M_1^{M_1^A} \Rightarrow \text{BigM}^A$

- Rewritten equivalently in terms of Kleisli composition \diamond_M as

$$(b : X \Rightarrow M_1^Y \circ \text{lift}_1) \diamond_{\text{BigM}} (c : Y \Rightarrow M_1^Z \circ \text{lift}_1) = (b \diamond_{M_1} c) \circ \text{lift}_1$$

- Liftings `lift1` and `lift2` 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 BigM^\bullet

The (functor) naturality law: for any $f : X \Rightarrow Y$,

$$\begin{array}{ccc} M_1^X & \xrightarrow{\text{lift}_1} & \text{BigM}^X \\ \text{fmap}_{M_1} f : X \Rightarrow Y \downarrow & & \downarrow \text{fmap}_{\text{BigM}} f : X \Rightarrow Y \\ M_1^Y & \xrightarrow{\text{lift}_1} & \text{BigM}^Y \end{array}$$

Derivation of the naturality law:

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

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 $M_2^{M_1^\bullet}$
 - ★ Example: state monad $\text{St}_S^A \equiv S \Rightarrow A \times S$ does not compose
- 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**: $L^A \Rightarrow (A \Rightarrow F^B) \Rightarrow F^{L^B}$ – 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”: lifts $L^A \leadsto A$ into 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  $\Rightarrow$  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`)?
 - ▶ These monads do not compose (neither “inside” nor “outside” works)
- 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 the boilerplate around `lift`? (`mtl`-style transformers)

Monad transformers IV: The zoology of monads

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

- “Composed-inside”, base monad is inside foreign monad: $T_L^{M,A} = M^{L^A}$
 - ▶ Examples: `OptionT`, `WriterT`, `EitherT`
- “Composed-outside”, base monad is outside: $T_L^{M,A} = L^{M^A}$
 - ▶ Examples: `ReaderT`; `SearchT` for search monad $S[A] = (A \Rightarrow Z) \Rightarrow A$
 - ▶ 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`, `NonEmptyListT`, `FreeMonadT`
- “Irregular”: none of the above constructions work
 - ▶ Examples: `StateT`, `ContT`, “codensity monads”
- Monad constructions: defining a transformer for new monads
 - ▶ Product monads $L_1^A \times L_2^A$ – “product transformer” $T_{L_1}^{M,A} \times T_{L_2}^{M,A}$
 - ▶ Consumer choice monads $H^A \Rightarrow A$ – “composed-outside” transformer
 - ▶ Free pointed monads $A + L^A$ – transformer $M^{A+T_L^{M,A}}$
 - ▶ “Selectors” $(A \Rightarrow P^Q) \Rightarrow P^A$ – transformer $(M^A \Rightarrow T_P^{M,Q}) \Rightarrow T_P^{M,A}$
- Examples of monads for which no transformers exist? (unknown)

Rigid monads, their laws and structure I

- A **rigid monad** R^\bullet has the method **fuseIn**: $(A \Rightarrow R^B) \Rightarrow R^{A \Rightarrow B}$
 - ▶ Examples: $R^A \equiv A \times A$ and $R^A \equiv Z \Rightarrow A$ are rigid; $R^A \equiv 1 + A$ is not
 - ▶ Compare with **fuseOut**: $R^{A \Rightarrow B} \Rightarrow A \Rightarrow R^B$, which exists for any functor
 - ▶ Implementation: $\text{fo } h : R^{A \Rightarrow B} = x : A \Rightarrow (f : A \Rightarrow B \Rightarrow f x)^{\uparrow R} h$
- Laws: the **fuseIn** method (**fi**) must be “compatible with the monad”
 - ▶ **fi** must be a lawful lifting from $A \Rightarrow R^B$ to $R^{A \Rightarrow B}$
- That is, a functor from Kleisli category to Applicative category
 - ▶ identity law: $\text{fi}(\text{pure}_R) = \text{pure}_R(\text{id})$
 - ▶ composition law: $\text{fi}(f \diamond_R g) = (p \times q \Rightarrow p \circ q)^{\uparrow R} (\text{fi } f \boxtimes \text{fi } g)$

$$\begin{array}{ccccc}
 A \Rightarrow R^B & \times & B \Rightarrow R^C & \xrightarrow{\text{use } \diamond_R} & A \Rightarrow R^C \\
 \downarrow \text{fi} & & \downarrow \text{fi} & & \downarrow \text{fi} \\
 R^{A \Rightarrow B} & \times & R^{B \Rightarrow C} & \xrightarrow{\text{use } \boxtimes} R^{(A \Rightarrow B) \times (B \Rightarrow C)} & \xrightarrow{\text{fmap}(\circ)} R^{A \Rightarrow C} \\
 & & & & \downarrow \text{fi}
 \end{array}$$

 - ▶ Alternative formulation: $\text{flm} = \text{fi} \circ \text{pa}$ where $\text{pa} : R^{A \Rightarrow B} \Rightarrow R^A \Rightarrow R^B$
 - ▶ Then $\text{fi} \circ \text{fo} = \text{id}$. Proof: $\text{fo } x \ a = \text{pa } x \ (\text{pure } a)$; set $x : R^{A \Rightarrow B} = \text{fi } h : A \Rightarrow R^B$ and get $\text{fo } x \ a = \text{pa} \ (\text{fi } h) \ (\text{pure } a) = \text{flm } h \ (\text{pure } a) = h \ a$, so $\text{fo} \ (\text{fi } h) = h$
- Rigid monads R^\bullet have “composed-outside” transformers, $T_R^{M,A} \equiv R^{M^A}$

Rigid monads, their laws and structure II

Examples and constructions of rigid and non-rigid monads:

- Rigid: $R^A \equiv A$, $R^A \equiv Z \Rightarrow A$, $R^A \equiv H^A \Rightarrow A$ (H^\bullet is a contrafunctor)
- Not rigid: $R^A \equiv 1$, $R^A \equiv W \times A$, $R^A \equiv E + A$, List^A , Cont^A , State^A
- The composition of rigid monads is rigid: $R_1^{R_2^A}$
- The product of rigid monads is rigid: $R_1^A \times R_2^A$
- The selector monad $S^A \equiv (A \Rightarrow R^Q) \Rightarrow R^A$ is rigid if R^A is rigid
- Any rigid functor is pointed: $A \Rightarrow R^A$

Use cases for rigid monads:

- For a rigid monad R^\bullet and any monad M^\bullet , have “ R -valued `flatMap`”:
 $M^A \times (A \Rightarrow R^{M^B}) \Rightarrow R^{M^B}$ – handles multiple M^\bullet effects at once
- For a rigid monad R^\bullet , can implement a general refactoring function,
`monadify`: $((A \Rightarrow B) \Rightarrow C) \Rightarrow (A \Rightarrow R^B) \Rightarrow R^C$ – uptake monadic API

Invalid attempts to create a general monad transformer

General recipes for combining two functors L^\bullet and M^\bullet all fail

- “Fake” transformers: $T_L^{M,A} \equiv L^A$; or $T_L^{M,A} \equiv M^A$; or just $T_L^{M,A} \equiv 1$
 - ▶ no **lift** and/or no base runner and/or $T_L^{\text{Id},A} \not\equiv L^A$
- Functor composition: $L^{M^\bullet}, M^{L^\bullet}$ – not a monad for some L^\bullet, M^\bullet
- Making a monad out of functor composition:
 - ▶ free monad over L^{M^\bullet} , Free^{L^M} – **lift** violates lifting laws
 - ▶ free monad over $L^\bullet + M^\bullet$, $\text{Free}^{L^\bullet + M^\bullet}$ – **lift** violates lifting laws
 - ★ However, laws will hold after interpreting the free monad!
 - ▶ codensity monad over L^{M^\bullet} : $F^A \equiv \forall B. (A \Rightarrow L^{M^B}) \Rightarrow L^{M^B}$ – no **lift**
- Codensity- L transformer: $\text{Cod}_L^{M,A} \equiv \forall B. (A \Rightarrow L^B) \Rightarrow L^{M^B}$ – no **lift**
 - ▶ uses the continuation transformer on $M^A \cong \forall B. (A \Rightarrow B) \Rightarrow M^B$
- Codensity composition: $F^A \equiv \forall B. (M^A \Rightarrow L^B) \Rightarrow L^B$ – not a monad
 - ▶ Counterexample: $M^A \equiv R \Rightarrow A$ and $L^A \equiv S \Rightarrow A$
- “Monoidal” convolution: $(L \star M)^A \equiv \exists P \exists Q. (P \times Q \Rightarrow A) \times L^P \times M^Q$
 - ▶ combines $L^A \cong \exists P. L^P \times (P \Rightarrow A)$ with $M^A \cong \exists Q. M^Q \times (Q \Rightarrow A)$
 - ▶ $L \star M$ is not a monad for some L^\bullet, M^\bullet

- 1 Show that the method `pure`: $A \Rightarrow M^A$ is a monadic morphism between monads $\text{Id}^A \equiv A$ and M^A . Show that $1 \Rightarrow 1 + A$ is not a monadic morphism.
- 2 Show that $M_1^A + M_2^A$ is *not* a monad when $M_1^A \equiv 1 + A$ and $M_2^A \equiv Z \Rightarrow A$.
- 3 Derive the composition law for `lift` written using `ftn` as $\text{lift}_1 \circ \text{fmap}_{\text{BigM}} \text{lift}_1 \circ \text{ftn}_{\text{BigM}} = \text{ftn}_{M_1} \circ \text{lift}_1$ from the `flm`-based law $\text{lift}_1 \circ \text{flm}_{\text{BigM}} (q \circ \text{lift}_1) = \text{flm}_{M_1} q \circ \text{lift}_1$. Draw type diagrams for both laws.