# 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 $\text{List}^A$ 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 ($\diamond$):

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

- 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:

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

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

# Laws for monad liftings II. Simplifying the laws

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

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

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

```
// Anywhere inside a for/yield, this...        // 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 $\mathsf{flm}_M : (A \Rightarrow M^B) \Rightarrow M^A \Rightarrow M^B$ as
$\mathsf{lift}_1\,\mathring{,}\,\mathsf{flm}_{\mathsf{BigM}}\,(q\,\mathring{,}\,\mathsf{lift}_1) = \mathsf{flm}_{M_1}\,q\,\mathring{,}\,\mathsf{lift}_1$ – both sides are functions $M_1^A \Rightarrow \mathsf{BigM}^B$
- Rewritten equivalently through $\mathsf{ftn}_M : M^{M^A} \Rightarrow M^A$, the law is
$\mathsf{lift}_1\,\mathring{,}\,\mathsf{fmap}_{\mathsf{BigM}}\mathsf{lift}_1\,\mathring{,}\,\mathsf{ftn}_{\mathsf{BigM}} = \mathsf{ftn}_{M_1}\,\mathring{,}\,\mathsf{lift}_1$ – both sides are functions $M_1^{M_1^A} \Rightarrow \mathsf{BigM}^A$
- In terms of Kleisli composition $\diamond_M$ it becomes the **composition law**:
$$\left(b^{:X \Rightarrow M_1^Y}\,\mathring{,}\,\mathsf{lift}_1\right) \diamond_{\mathsf{BigM}} \left(c^{:Y \Rightarrow M_1^Z}\,\mathring{,}\,\mathsf{lift}_1\right) = \left(b \diamond_{M_1} c\right)\,\mathring{,}\,\mathsf{lift}_1$$
- Liftings $\mathsf{lift}_1$ and $\mathsf{lift}_2$ 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 $\text{pure}_{M_1}$ to $\text{pure}_{\text{BigM}}$ and $\text{flm}_{M_1}$ to $\text{flm}_{\text{BigM}}$
  - $\text{lift}_1$ is a **monadic morphism** between monads $M_1^\bullet$ and $\text{BigM}^\bullet$
  - example: monad "interpreters" $M^A \Rightarrow N^A$ are monadic morphisms

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

$$\text{lift}_1 \, \text{\fontsize{}{}\selectfont} \, \text{fmap}_{\text{BigM}} f = \text{fmap}_{M_1} f \, \text{\fontsize{}{}\selectfont} \, \text{lift}_1$$

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

Derivation of the functor naturality law for $\text{lift}_1$:

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

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 $\text{lift}_L^M : M^A \rightsquigarrow T_L^{M,A}$
- "Base lifting" – a monadic morphism $\text{blift} : L^A \rightsquigarrow T_L^{M,A}$
  - ▸ The "base lifting" could not possibly be natural in $L^\bullet$
- Transformed identity monad (Id) must become $L$, 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 $f : M^\bullet \rightsquigarrow N^\bullet$ we need to have a monadic morphism $T_L^{M,\bullet} \rightsquigarrow T_L^{N,\bullet}$ for the transformed monads: $\text{mrun}_L^M : (M^\bullet \rightsquigarrow N^\bullet) \Rightarrow T_L^{M,\bullet} \rightsquigarrow T_L^{N,\bullet}$
    - ★ 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 can be used for lifting a "runner" $M^A \rightsquigarrow A$ to $T_L^{M,\bullet} \rightsquigarrow T_L^{\text{Id},\bullet} = L^\bullet$
- "Base runner": lifts $L^A \rightsquigarrow A$ into a monadic morphism $T_L^{M,\bullet} \rightsquigarrow M^\bullet$; $\text{brun}_L^M : (L^\bullet \rightsquigarrow \bullet) \Rightarrow T_L^{M,\bullet} \rightsquigarrow M^\bullet$

# 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`)?
  - ▶ 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? (unknown; may be impossible for some monads)
- 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: the "single-value monads" `OptionT`, `WriterT`, `EitherT`
- "Composed-outside" – the 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`, `NonEmptyListT`, `FreeMonadT`
- 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 $M^{A+T_L^{M,A}}$
- "Irregular": none of the above constructions work, need something else
  - $T_{\mathbf{State}}^{M,A} = S \Rightarrow M^{S \times A}$; $T_{\mathbf{Cont}}^{M,A} = (A \Rightarrow M^R) \Rightarrow M^R$; "selector" $F^{A \Rightarrow P^Q} \Rightarrow P^A$
    – transformer $F^{A \Rightarrow T_P^{M,Q}} \Rightarrow T_P^{M,A}$; codensity $\forall R. (A \Rightarrow M^R) \Rightarrow M^R$
- Examples of monads $K^A$ for which no transformers exist? (not sure)
  - $K^A \equiv A + ((A \Rightarrow R) \Rightarrow R)$ and $K^A \equiv A + ((A \Rightarrow P^Q) \Rightarrow P^A)$

# Composed-inside transformers I

Base monad $L^\bullet$, foreign monad $M^\bullet$, transformer $T_L^{M,\bullet} \equiv T^\bullet \equiv M^{L^\bullet}$

- Monad instance: use the natural transformation $\mathrm{seq}_L^{M,A} : L^{M^A} \rightsquigarrow M^{L^A}$
  - $\mathrm{pure}_T : A \Rightarrow M^{L^A}$ is defined as $\mathrm{pure}_T = \mathrm{pure}_M \,\mathring{,}\, \mathrm{pure}_L^{\uparrow M}$
  - $\mathrm{ftn}_T : T^{T^A} \Rightarrow T^A$ is defined as $\mathrm{ftn}_T = \mathrm{seq}^{\uparrow M} \,\mathring{,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{,}\, \mathrm{ftn}_M$

$$T^{T^A} \equiv M^{L^{M^{L^A}}} \xrightarrow{\ \mathrm{fmap}_M \, \mathrm{seq}_L^{M,L^A}\ } M^{M^{L^{L^A}}} \xrightarrow{\ \mathrm{fmap}_M(\mathrm{fmap}_M \mathrm{ftn}_L)\ } M^{M^{L^A}} \xrightarrow{\ \ \mathrm{ftn}_M\ \ } M^{L^A} \equiv T^A$$

- Monad laws must hold for $T^A$ (must check this separately)
  - This depends on special properties of $\mathrm{seq}_L^{M,A}$ (denoted seq for brevity), e.g. $\mathrm{pure}_L \,\mathring{,}\, \mathrm{seq} = \mathrm{pure}_L^{\uparrow M}$ ($L$-identity); $\mathrm{pure}_M^{\uparrow L} \,\mathring{,}\, \mathrm{seq} = \mathrm{pure}_M$ ($M$-identity)
    - ★ See example code that verifies these properties for $L^A \equiv E + W \times A$
    - ★ It is not enough to have *any* traversable functor $L^\bullet$ here!
- Monad transformer methods for $T_L^{M,\bullet} \equiv M^{L^\bullet}$:
  - Lifting, lift : $M^A \Rightarrow M^{L^A}$ is defined as $\mathrm{lift} = \mathrm{pure}_L^{\uparrow M}$
  - Base lifting, blift : $L^A \Rightarrow M^{L^A}$ is equal to $\mathrm{pure}_M$
  - Runner, mrun : $(\forall B.M^B \Rightarrow N^B) \Rightarrow M^{L^A} \Rightarrow N^{L^A}$ is equal to id
  - Base runner, brun : $(\forall B.L^B \Rightarrow B) \Rightarrow M^{L^A} \Rightarrow M^A$ is equal to $\mathrm{fmap}_M$

# Composed-inside transformers II

Base monad $L^\bullet$, foreign monad $M^\bullet$, transformer $T_L^{M,\bullet} \equiv T^\bullet \equiv M^{L^\bullet}$

- Identity laws for the monad $T^\bullet$ hold if they hold for $L^\bullet$ and $M^\bullet$ and if the properties $\mathrm{pure}_L \,\mathring{\,}\, \mathrm{seq} = \mathrm{pure}_L^{\uparrow M}$ and $\mathrm{pure}_M^{\uparrow L} \,\mathring{\,}\, \mathrm{seq} = \mathrm{pure}_M$ hold

- $\mathrm{pure}_T \,\mathring{\,}\, \mathrm{ftn}_T = \mathrm{id}$. Proof: $(\mathrm{pure}_M \,\mathring{\,}\, \mathrm{pure}_L^{\uparrow M}) \,\mathring{\,}\, (\mathrm{seq}^{\uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M) = \mathrm{pure}_M \,\mathring{\,}\, (\mathrm{pure}_L \,\mathring{\,}\, \mathrm{seq})^{\uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M = \mathrm{pure}_M \,\mathring{\,}\, \mathrm{pure}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M = \mathrm{id}$

- $\mathrm{pure}_T^{\uparrow T} \,\mathring{\,}\, \mathrm{ftn}_T = \mathrm{id}$. Proof: $\mathrm{pure}_T = \mathrm{pure}_M \,\mathring{\,}\, \mathrm{pure}_L^{\uparrow M} = \mathrm{pure}_L \,\mathring{\,}\, \mathrm{pure}_M$ (naturality); for all $f$: $f^{\uparrow T} = f^{\uparrow L \uparrow M}$ and $f \,\mathring{\,}\, \mathrm{pure}_M = \mathrm{pure}_M \,\mathring{\,}\, f^{\uparrow M}$ (naturality); so $\mathrm{pure}_T^{\uparrow T} \,\mathring{\,}\, \mathrm{ftn}_T$ is $(\mathrm{pure}_L \,\mathring{\,}\, \mathrm{pure}_M)^{\uparrow L \uparrow M} \,\mathring{\,}\, (\mathrm{seq}^{\uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M) = \mathrm{pure}_L^{\uparrow L \uparrow M} \,\mathring{\,}\, \mathrm{pure}_M^{\uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M = \mathrm{pure}_M^{\uparrow M} \,\mathring{\,}\, (\mathrm{pure}_L^{\uparrow L} \,\mathring{\,}\, \mathrm{ftn}_L)^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M = \mathrm{id}$ where we used naturality with $f = \mathrm{pure}_L^{\uparrow L}$

- Identity law for lift: $\mathrm{pure}_M \,\mathring{\,}\, \mathrm{lift} = \mathrm{pure}_T$ (this *is* the definition of $\mathrm{pure}_T$)

- Composition law: $\mathrm{lift} \,\mathring{\,}\, \mathrm{lift}^{\uparrow T} \,\mathring{\,}\, \mathrm{ftn}_T = \mathrm{ftn}_M \,\mathring{\,}\, \mathrm{lift}$. Proof: $\mathrm{ftn}_M \,\mathring{\,}\, \mathrm{pure}_L^{\uparrow M} = \mathrm{pure}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M$ and $\mathrm{pure}_L^{\uparrow M} \,\mathring{\,}\, (\mathrm{pure}_L^{\uparrow M \uparrow L \uparrow M} \,\mathring{\,}\, \mathrm{seq}^{\uparrow M}) \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M = (\mathrm{pure}_L^{\uparrow M} \,\mathring{\,}\, \mathrm{seq}^{\uparrow M}) \,\mathring{\,}\, (\mathrm{pure}_L^{\uparrow L \uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M}) \,\mathring{\,}\, \mathrm{ftn}_M = \mathrm{pure}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M$

- Identity law for blift: $\mathrm{pure}_L \,\mathring{\,}\, \mathrm{blift} = \mathrm{pure}_T$. ($\mathrm{pure}_L \,\mathring{\,}\, \mathrm{pure}_M = \mathrm{pure}_M \,\mathring{\,}\, \mathrm{pure}_L^{\uparrow M}$)

- Composition law: $\mathrm{blift} \,\mathring{\,}\, \mathrm{blift}^{\uparrow T} \,\mathring{\,}\, \mathrm{ftn}_T = \mathrm{ftn}_L \,\mathring{\,}\, \mathrm{blift}$. Proof: $\mathrm{pure}_M \,\mathring{\,}\, \mathrm{pure}_M^{\uparrow L \uparrow M} \,\mathring{\,}\, (\mathrm{seq}^{\uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M) = \mathrm{pure}_M \,\mathring{\,}\, (\mathrm{pure}_L^{\uparrow M} \,\mathring{\,}\, \mathrm{ftn}_L^{\uparrow M \uparrow M}) \,\mathring{\,}\, \mathrm{ftn}_M = \mathrm{pure}_M \,\mathring{\,}\, (\mathrm{ftn}_L^{\uparrow M} \,\mathring{\,}\, \mathrm{pure}_M^{\uparrow M}) \,\mathring{\,}\, \mathrm{ftn}_M = \mathrm{ftn}_L \,\mathring{\,}\, \mathrm{pure}_M \,\mathring{\,}\, (\mathrm{pure}_M^{\uparrow M} \,\mathring{\,}\, \mathrm{ftn}_M) = \mathrm{ftn}_L \,\mathring{\,}\, \mathrm{blift}$

- Runner laws follow from naturality of $\mathrm{id}$ and $\mathrm{fmap}$

# Rigid monads, their laws and structure I

- A **rigid functor** $R^\bullet$ has the method `fuseIn`: $\left(A \Rightarrow R^B\right) \Rightarrow R^{A \Rightarrow B}$
  - 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: fo $h^{:R^{A \Rightarrow B}} = x^{:A} \Rightarrow \left(f^{:A \Rightarrow B} \Rightarrow f\, x\right)^{\uparrow R} h$

Laws: the `fuseIn` method (`fi`) must be "compatible with the monad $R$"

- `fi` must be a natural lifting from $A \Rightarrow R^B$ to $R^{A \Rightarrow B}$
- Naturality: fi $\left(f \,\overset{\circ}{,}\, g^{\uparrow R}\right) =$ fi $f \,\overset{\circ}{,}\, \left(q^{:A \Rightarrow B} \Rightarrow q \,\overset{\circ}{,}\, g\right)^{\uparrow R}$ for $\forall\, f^{:A \Rightarrow R^B}$, $g^{:B \Rightarrow C}$
- A (generalized) functor from Kleisli category to "applicative" category
  - identity law: fi $\left(\text{pure}_R\right) = \text{pure}_R\left(\text{id}\right)$
  - composition law: fi $\left(f \diamond_R g\right) = \left(p \times q \Rightarrow p \,\overset{\circ}{,}\, q\right)^{\uparrow R}\left(\text{fi}\, f \bowtie \text{fi}\, g\right)$

$$
\begin{array}{ccccc}
A \Rightarrow R^B & \times & B \Rightarrow R^C & \xrightarrow{\;\text{use } \diamond_R\;} & A \Rightarrow R^C \\
\downarrow{\scriptstyle\text{fi}} & & \downarrow{\scriptstyle\text{fi}} & & \downarrow{\scriptstyle\text{fi}} \\
R^{A \Rightarrow B} & \times & R^{B \Rightarrow C} & \underset{\text{use } \bowtie}{\longrightarrow} R^{(A \Rightarrow B) \times (B \Rightarrow C)} \xrightarrow[\text{fmap}(\overset{\circ}{,})]{} & R^{A \Rightarrow C}
\end{array}
$$

  - Alternative formulation: flm $=$ fi $\overset{\circ}{,}$ pa where pa $: R^{A \Rightarrow B} \Rightarrow R^A \Rightarrow R^B$
  - Then fi $\overset{\circ}{,}$ fo $=$ id. Proof: fo $x\, a = $ pa $x$ (pure $a$); set $x^{:R^{A \Rightarrow B}} = $ fi $h^{:A \Rightarrow R^B}$
    and get fo $x\, a = $ pa (fi $h$) (pure $a$) $= $ flm $h$ (pure $a$) $= h\, a$, so fo (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: `Id`, `Reader`, and $R^A \equiv H^A \Rightarrow A$ (where $H^\bullet$ is a contrafunctor)
  - The construction $R^A \equiv H^A \Rightarrow A$ covers $R^A \equiv 1$, $R^A = A$, $R^A = Z \Rightarrow A$
- Not rigid: $R^A \equiv W \times A$, $R^A \equiv E + A$, $\text{List}^A$, $\text{Cont}^A$, $\text{State}^A$
- 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 $S^A \equiv \left(A \Rightarrow R^Q\right) \Rightarrow R^A$ is rigid if $R^A$ is rigid

Use cases for rigid functors and rigid monads:

- A rigid functor is pointed: a method $A \Rightarrow R^A$ can be defined
- A rigid functor has a single constructor because $R^1 \cong 1$
- Handle multiple $M^\bullet$ effects at once: For a rigid monad $R^\bullet$ and any monad $M^\bullet$, have "$R$-valued `flatMap`": $M^A \times \left(A \Rightarrow R^{M^B}\right) \Rightarrow R^{M^B}$
- Uptake monadic API: For a rigid monad $R^\bullet$, can implement a general refactoring function, `monadify`: $((A \Rightarrow B) \Rightarrow C) \Rightarrow \left(A \Rightarrow R^B\right) \Rightarrow R^C$

# Composed-outside transformers for rigid monads I

Base rigid monad $R^\bullet$, foreign monad $M^\bullet$, transformer $T_R^{M,\bullet} \equiv T^\bullet \equiv R^{M^\bullet}$

- Monad instance: define the Kleisli category with morphisms $A \Rightarrow R^{M^A}$
- $\mathsf{pure}_T : A \Rightarrow R^{M^A}$ is defined as
  $\mathsf{pure}_T = \mathsf{pure}_M \,\hat{\circ}\, \mathsf{pure}_R = \mathsf{pure}_R \,\hat{\circ}\, \mathsf{pure}_M^{\uparrow R}$
- $\diamond_T$ is defined by $f \diamond_T g = \mathsf{fo}\big( (p \times q \Rightarrow p \diamond_M q)^{\uparrow R} (\mathsf{fi}\, f \bowtie_R \mathsf{fi}\, g) \big)$

$$
\begin{array}{ccccc}
(A \Rightarrow R^{M^B}) & \diamond_T & (B \Rightarrow R^{M^C}) & \xrightarrow{\quad\text{define } \diamond_T \text{ as}\quad} & A \Rightarrow R^{M^C} \\
\downarrow{\scriptstyle\mathsf{fi}} & & \downarrow{\scriptstyle\mathsf{fi}} & & \uparrow{\scriptstyle\mathsf{fo}} \\
R^{A \Rightarrow M^B} & \bowtie_R & R^{B \Rightarrow M^C} \longrightarrow R^{(A \Rightarrow M^B) \times (B \Rightarrow M^C)} & \xrightarrow{\mathsf{fmap}_R(\diamond_M)} & R^{A \Rightarrow M^C}
\end{array}
$$

- Monad laws hold for $T^A$ as Kleisli category laws
- Identity: $\mathsf{pure}_T$ is $\mathsf{id}_{\diamond_T}$ due to $\mathsf{fi}\,(\mathsf{pure}_T) = \mathsf{pure}_R\,(\mathsf{pure}_M)$
- Associativity of $\diamond_T$ follows from associativity of $\diamond_M$ and $\bowtie_R$
- Monad transformer methods for $T_R^{M,\bullet} \equiv R^{M^\bullet}$:
  - Lifting, $\mathsf{lift} : M^A \Rightarrow R^{M^A}$ is equal to $\mathsf{pure}_M$
  - Base lifting, $\mathsf{blift} : R^A \Rightarrow R^{M^A}$ is equal to $\mathsf{pure}_M^{\uparrow R}$
  - Runner, $\mathsf{mrun} : \big(\forall B.M^B \Rightarrow N^B\big) \Rightarrow R^{M^A} \Rightarrow R^{N^A}$ is equal to $\mathsf{fmap}_R$
  - Base runner, $\mathsf{brun} : \big(\forall B.R^B \Rightarrow B\big) \Rightarrow R^{M^A} \Rightarrow M^A$ is equal to $\mathsf{id}$

Base rigid monad $R^\bullet$, foreign monad $M^\bullet$, transformer $T_R^{M,\bullet} \equiv T^\bullet \equiv R^{M^\bullet}$

- Monad instance: define the Kleisli category with morphisms $A \Rightarrow R^{M^A}$
- $\text{pure}_T : A \Rightarrow R^{M^A}$ is defined as
  $\text{pure}_T = \text{pure}_M \, \mathring{;} \, \text{pure}_R = \text{pure}_R \, \mathring{;} \, \text{pure}_M^{\uparrow R}$
- $\diamond_T$ is define

# Codensity monads

**Codensity monad** over a functor $F^\bullet$ is $\mathrm{Cod}^{F,A} \equiv \forall B.\, (A \Rightarrow F^B) \Rightarrow F^B$
Properties:

- $\mathrm{Cod}^{F,\bullet}$ is a monad for any functor $F^\bullet$
- If $F^\bullet$ is itself a monad then we have monadic morphisms
  $\mathrm{inC} : F^\bullet \rightsquigarrow \mathrm{Cod}^{F,\bullet}$ and $\mathrm{outC} : \mathrm{Cod}^{F,\bullet} \rightsquigarrow F^\bullet$ such that $\mathrm{inC} \,\raise{1pt}{\mathbf{;}}\, \mathrm{outC} = \mathrm{id}$

# 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, disjunction, or product: $L^{M^\bullet}$, $M^{L^\bullet}$, $L^\bullet + M^\bullet$ – not a monad in general; $L^\bullet \times M^\bullet$ – no lifting $M^\bullet \rightsquigarrow L^\bullet \times M^\bullet$
- Making a monad out of functor composition:
  - free monad over $L^{M^\bullet}$, $\text{Free}^{L^M}$ – `lift` violates lifting laws
  - free monad over $L^\bullet + M^\bullet$, $\text{Free}^{L^\bullet + M^\bullet}$ – `lift` violates lifting laws
    - ★ Laws will hold after interpreting the free monad into a concrete monad
  - codensity monad over $L^{M^\bullet}$: $F^A \equiv \forall B. \left( A \Rightarrow L^{M^B} \right) \Rightarrow L^{M^B}$ – no `lift`
- Codensity-$L$ transformer: $\text{Cod}_L^{M,A} \equiv \forall B. \left( A \Rightarrow L^B \right) \Rightarrow L^{M^B}$ – no `lift`
  - applies the continuation transformer to $M^A \cong \forall B. (A \Rightarrow B) \Rightarrow M^B$
- Codensity composition: $F^A \equiv \forall B. \left( M^A \Rightarrow L^B \right) \Rightarrow L^B$ – 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 e.g. $L^A \equiv 1 + A$ and $M^A \equiv R \Rightarrow A$

# Exercises

1. Show that the method `pure`: $A \Rightarrow M^A$ is a monadic morphism between monads $\mathrm{Id}^A \equiv A$ 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 $\mathrm{lift}_1 \,\mathring{,}\, \mathrm{fmap}_{\mathrm{BigM}} \mathrm{lift}_1 \,\mathring{,}\, \mathrm{ftn}_{\mathrm{BigM}} = \mathrm{ftn}_{M_1} \,\mathring{,}\, \mathrm{lift}_1$ from the flm-based law $\mathrm{lift}_1 \,\mathring{,}\, \mathrm{flm}_{\mathrm{BigM}} (q \,\mathring{,}\, \mathrm{lift}_1) = \mathrm{flm}_{M_1} q \,\mathring{,}\, \mathrm{lift}_1$. Draw type diagrams for both laws.

4. Show that the continuation monad is not rigid and does not compose with arbitrary other monads.