

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 (see code)

- “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:

<code>// Anywhere inside a for/yield, this...</code>	<code>// must be equivalent to...</code>
<code>i ← lift₁(p) // Any M₁ value.</code>	<code>pq = p.flatMap(q) // In M₁.</code>
<code>j ← lift₁(q(i)) // Any M₁ value.</code>	<code>j ← lift₁(pq) // Now lift it.</code>

`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 $\text{ftn}_M : M^{M^A} \Rightarrow M^A$, the law is $\text{lift}_1 \circ \text{fmap}_{\text{BigM}} \text{lift}_1 \circ \text{ftn}_{\text{BigM}} = \text{ftn}_{M_1} \circ \text{lift}_1$ – both sides are functions $M_1^{M^A} \Rightarrow \text{BigM}^A$
- In terms of Kleisli composition \diamond_M it becomes the **composition law**:
$$(b^{X \Rightarrow M_1^Y} \circ \text{lift}_1) \diamond_{\text{BigM}} (c^{Y \Rightarrow M_1^Z} \circ \text{lift}_1) = (b \diamond_{M_1} c) \circ \text{lift}_1$$
- 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
 - ▶ example: monad “interpreters” $M^A \Rightarrow N^A$ are monadic morphisms

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

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

Derivation of the functor naturality law for lift_1 :

- 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 $\text{lift}_L^M : M^A \rightsquigarrow T_L^{M,A}$
- “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: $\text{mrun}_L^M : (M^\bullet \rightsquigarrow N^\bullet) \Rightarrow T_L^{M,\bullet} \rightsquigarrow T_L^{N,\bullet}$ commuting with **lift** / **blift**
 - ★ 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 $T_L^{M,\bullet} \rightsquigarrow M^\bullet$; so $\text{brun}_L^M : (L^\bullet \rightsquigarrow \bullet) \Rightarrow T_L^{M,\bullet} \rightsquigarrow M^\bullet$, must commute with **lift** and **blift**

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` must be composed *inside* the foreign monad

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 *ad hoc* methods

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

- “Composed-inside”, base monad is inside foreign monad: $T_L^{M,A} = M^{L^A}$
 - ▶ Examples: the “linear-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 $S[A] = (A \Rightarrow Z) \Rightarrow A$
 - ▶ More generally: all “rigid” monads have “outside” transformers
- “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
 - ▶ $T_{\text{State}}^{M,A} = S \Rightarrow M^{S \times A}$; $T_{\text{Cont}}^{M,A} = (A \Rightarrow M^R) \Rightarrow M^R$; “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 known)
 - ▶ T_{Cod} , T_{Sel} , and T_{Cont} transformers have no `blift`, `brun`, or `mrn`

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 $\text{seq}^A : L^{M^A} \Rightarrow M^{L^A}$
 - ▶ $\text{pure}_T : A \Rightarrow M^{L^A}$ is defined as $\text{pure}_T = \text{pure}_M \circ \text{pure}_L^{\uparrow M}$
 - ▶ $\text{ftn}_T : T^{T^A} \Rightarrow T^A$ is defined as $\text{ftn}_T = \text{seq}^{\uparrow M} \circ \text{ftn}_L^{\uparrow M \uparrow M} \circ \text{ftn}_M$

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

- Monad laws must hold for T^A (must check this separately)
 - ▶ This depends on special properties of seq , e.g. $\text{pure}_L \circ \text{seq} = \text{pure}_L^{\uparrow M}$ (L -identity); $\text{pure}_M^{\uparrow L} \circ \text{seq} = \text{pure}_M$ (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, $\text{lift} : M^A \Rightarrow M^{L^A}$ is defined as $\text{lift} = \text{pure}_L^{\uparrow M}$
 - ▶ Base lifting, $\text{blift} : L^A \Rightarrow M^{L^A}$ is equal to pure_M
 - ▶ Runner, $\text{mrun} : (\forall B. M^B \Rightarrow N^B) \Rightarrow M^{L^A} \Rightarrow N^{L^A}$ is equal to id
 - ▶ Base runner, $\text{brun} : (\forall B. L^B \Rightarrow B) \Rightarrow M^{L^A} \Rightarrow M^A$ is equal to fmap_M

* Composed-inside transformers II. Proofs

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 $\text{pure}_L \circ \text{seq} = \text{pure}_L^{\uparrow M}$ and $\text{pure}_M^{\uparrow L} \circ \text{seq} = \text{pure}_M$ hold
- $\text{pure}_T \circ \text{fth}_T = \text{id}$. Proof: $(\text{pure}_M \circ \text{pure}_L^{\uparrow M}) \circ (\text{seq}^{\uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M} \circ \text{fth}_M) = \text{pure}_M \circ (\text{pure}_L \circ \text{seq})^{\uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M} \circ \text{fth}_M = \text{pure}_M \circ \text{pure}_L^{\uparrow M \uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M} \circ \text{fth}_M = \text{id}$
- $\text{pure}_T^{\uparrow T} \circ \text{fth}_T = \text{id}$. Proof: $\text{pure}_T = \text{pure}_M \circ \text{pure}_L^{\uparrow M} = \text{pure}_L \circ \text{pure}_M$ (naturality); for all f : $f^{\uparrow T} = f^{\uparrow L \uparrow M}$ and $f \circ \text{pure}_M = \text{pure}_M \circ f^{\uparrow M}$ (naturality); so $\text{pure}_T^{\uparrow T} \circ \text{fth}_T$ is $(\text{pure}_L \circ \text{pure}_M)^{\uparrow L \uparrow M} \circ (\text{seq}^{\uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M} \circ \text{fth}_M) = \text{pure}_L^{\uparrow L \uparrow M} \circ \text{pure}_M^{\uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M} \circ \text{fth}_M = \text{pure}_M^{\uparrow M} \circ (\text{pure}_L^{\uparrow L} \circ \text{fth}_L)^{\uparrow M \uparrow M} \circ \text{fth}_M = \text{id}$ where we used naturality with $f = \text{pure}_L^{\uparrow L}$
- Identity law for lift: $\text{pure}_M \circ \text{lift} = \text{pure}_T$ (this is the definition of pure_T)
- Composition law: $\text{lift} \circ \text{lift}^{\uparrow T} \circ \text{fth}_T = \text{fth}_M \circ \text{lift}$. Proof: $\text{fth}_M \circ \text{pure}_L^{\uparrow M} = \text{pure}_L^{\uparrow M \uparrow M} \circ \text{fth}_M$ and $\text{pure}_L^{\uparrow M} \circ (\text{pure}_L^{\uparrow M \uparrow L \uparrow M} \circ \text{seq}^{\uparrow M}) \circ \text{fth}_L^{\uparrow M \uparrow M} \circ \text{fth}_M = (\text{pure}_L^{\uparrow M} \circ \text{seq}^{\uparrow M}) \circ (\text{pure}_L^{\uparrow L \uparrow M \uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M}) \circ \text{fth}_M = \text{pure}_L^{\uparrow M \uparrow M} \circ \text{fth}_M$
- Identity law for blift: $\text{pure}_L \circ \text{blift} = \text{pure}_T$. ($\text{pure}_L \circ \text{pure}_M = \text{pure}_M \circ \text{pure}_L^{\uparrow M}$)
- Composition law: $\text{blift} \circ \text{blift}^{\uparrow T} \circ \text{fth}_T = \text{fth}_L \circ \text{blift}$. Proof: $\text{pure}_M \circ \text{pure}_M^{\uparrow L \uparrow M} \circ (\text{seq}^{\uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M} \circ \text{fth}_M) = \text{pure}_M \circ (\text{pure}_M^{\uparrow M} \circ \text{fth}_L^{\uparrow M \uparrow M}) \circ \text{fth}_M = \text{pure}_M \circ (\text{fth}_L^{\uparrow M} \circ \text{pure}_M^{\uparrow M}) \circ \text{fth}_M = \text{fth}_L \circ \text{pure}_M \circ (\text{pure}_M^{\uparrow M} \circ \text{fth}_M) = \text{fth}_L \circ \text{blift}$
- Runner laws follow from naturality of id and fmap

Rigid monads. Definitions

- A **rigid monad** R^\bullet has a composed-outside transformer, $T_R^{M,A} \equiv R^{M^A}$
 - ▶ Examples: $R^A \equiv A \times A$ and $R^A \equiv Z \Rightarrow A$ are rigid; $R^A \equiv 1 + A$ is not
 - ▶ For any monad M , we then have $\text{seq} : M^{R^A} \Rightarrow R^{M^A}$ defined by

$$\text{seq} = \text{pure}_M^{\uparrow R \uparrow M} \circ \text{pure}_{R^\circ} \circ \text{ftn}_T$$

$$M^{R^A} \xrightarrow{\text{fmap}_M(\text{fmap}_R(\text{pure}_M))} M^{R^{M^A}} \xrightarrow{\text{pure}_R} R^{M^{R^{M^A}}} = T^{T^A} \xrightarrow{\text{ftn}_T} T^A \equiv R^{M^A}$$

- Open question: is ftn_T definable via seq with some additional laws?

Examples and constructions of rigid monads:

- Rigid: **Id**, **Reader**, and $R^A \equiv H^A \Rightarrow A$ (where H is a contrafunctor)
 - ▶ The construction $R^A \equiv H^A \Rightarrow A$ covers $R^A \equiv 1$, $R^A \equiv A$, $R^A \equiv Z \Rightarrow A$
- The **selector monad** $S^A \equiv F^{A \Rightarrow R^Q} \Rightarrow R^A$ is rigid if R^\bullet is rigid
 - ▶ Simple example: search with failure, $S^A \equiv (A \Rightarrow \text{Bool}) \Rightarrow 1 + A$
- The composition of rigid monads, $R_1^{R_2^A}$, is a rigid monad
- The product of rigid monads, $R_1^A \times R_2^A$, is a rigid monad

Rigid functors, their laws and structure I

- A **rigid functor** R^\bullet has the method **fuseIn**: $(A \Rightarrow R^B) \Rightarrow R^{A \Rightarrow B}$
 - ▶ Rigid monads are rigid functors since **fi** = seq with $M^A \equiv Z \Rightarrow A$
 - ▶ Compare with **fuseOut**: $R^{A \Rightarrow B} \Rightarrow A \Rightarrow R^B$, which exists for *any* functor
 - ★ Implementation: **fo** $h^{R^{A \Rightarrow B}} = x^{A \Rightarrow} (f^{A \Rightarrow B} \Rightarrow f x)^{\uparrow R} h$
- Nondegeneracy law: **fuseOut**(**fuseIn**(**x**)) == **x** or **fi**;**fo** = id
- **fi** must be natural in both type parameters
 - ▶ Naturality: **fi** $(f \circ g^{\uparrow R}) = (q^{A \Rightarrow B} \Rightarrow q \circ g)^{\uparrow R} (\text{fi } f)$ for $\forall f^{A \Rightarrow R^B}, g^{B \Rightarrow C}$
and **fi** $(f \circ g) = (q^{B \Rightarrow C} \Rightarrow f \circ q)^{\uparrow R} (\text{fi } g)$ for $\forall f^{A \Rightarrow B}, g^{B \Rightarrow R^C}$
- Connection between monadic **flatMap** and applicative **ap** for monadic R :
 - ▶ **flm** : $(A \Rightarrow R^B) \Rightarrow R^A \Rightarrow R^B$
 - ▶ **ap** : $R^{A \Rightarrow B} \Rightarrow R^A \Rightarrow R^B$
 - ▶ The connection is **flm** = **fi**;**ap** and **ap** = **fo**;**flm**
 - ★ However, here we need to flip the order of R -effects in **ap**
 - ▶ Connection between **ap** and **fo** is **fo** $x \ a = \text{ap } x \ (\text{pure } a)$
- If **flm** = **fi**;**ap** then **fi**;**fo** = id. Proof: set $x^{R^{A \Rightarrow B}} = \text{fi } h^{A \Rightarrow R^B}$ and get **fo** $x \ a = \text{ap } (\text{fi } h) (\text{pure } a) = \text{flm } h (\text{pure } a) = h \ a$, so **fo** (**fi** h) = h
- Conversely: If **fi**;**fo** = id and **ap** = **fo**;**flm** then **flm** = **fi**;**ap**. Proof:
fi;**ap** = **fi**;**fo**;**flm** = **flm**

Rigid functors, their laws and structure II

Examples and constructions of rigid functors (see code):

- $R^A \equiv H^A \Rightarrow Q^A$ is a rigid functor (not monad) if Q^A is a rigid functor
- Not rigid: $R^A \equiv W \times A$, $R^A \equiv E + A$, List^A , Cont^A , State^A

Use cases for rigid functors:

- A rigid functor is pointed: a natural transformation $A \Rightarrow R^A$ exists
- A rigid functor has a single constructor because $R^1 \cong 1$
- Handle multiple M^\bullet effects at once: For a rigid functor R^\bullet and any monad M^\bullet , have “ R -valued M -flatMap”: $M^A \times (A \Rightarrow R^{M^B}) \Rightarrow R^{M^B}$
- Uptake monadic API: For a rigid functor R^\bullet , can implement a general refactoring function, `refactor`: $((A \Rightarrow B) \Rightarrow C) \Rightarrow (A \Rightarrow R^B) \Rightarrow R^C$, to transform a program $p(f^{A \Rightarrow B}) : C$ into $\tilde{p}(\tilde{f}^{A \Rightarrow R^B}) : R^C$

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 $A \Rightarrow R^{M^A}$
- $\text{pure}_T : A \Rightarrow R^{M^A}$ is defined by $\text{pure}_T \equiv \text{pure}_M \circ \text{pure}_R = \text{pure}_R \circ \text{pure}_M^{\uparrow R}$
- $\text{ftn}_T : T^{T^A} \Rightarrow T^A$ must be defined case by case for each construction
 - ▶ If R^{M^\bullet} is a monad then we can define $\text{seq} : M^{R^\bullet} \rightsquigarrow R^{M^\bullet}$
 - ▶ Choosing $M^A \equiv Z \Rightarrow A$, we get $\text{seq} = \text{fi} : (Z \Rightarrow R^A) \Rightarrow R^{Z \Rightarrow A}$
 - ★ Open question: Is a rigid monad always a rigid functor?

Define **rigid monads** via the existence of composed-outside transformers

- Monad transformer methods for $T_R^{M,\bullet} \equiv R^{M^\bullet}$:
 - ▶ Lifting, $\text{lift} : M^A \Rightarrow R^{M^A}$ is equal to pure_M
 - ▶ Base lifting, $\text{blift} : R^A \Rightarrow R^{M^A}$ is equal to $\text{pure}_M^{\uparrow R}$
 - ▶ Runner, $\text{mrun} : (\forall B. M^B \Rightarrow N^B) \Rightarrow R^{M^A} \Rightarrow R^{N^A}$ is equal to fmap_R
 - ▶ Base runner, $\text{brun} : (\forall B. R^B \Rightarrow B) \Rightarrow R^{M^A} \Rightarrow M^A$ is equal to id
- Checking the monad transformer laws, case by case
 - ▶ The laws hold for $R^A \equiv H^A \Rightarrow A$ and $R^A \equiv F^{A \Rightarrow P^Q} \Rightarrow P^A$
 - ▶ The laws hold for composition and product of rigid monads
 - ★ Any other constructions or examples?

* Composed-outside transformers for rigid monads II. Proofs

- Several classes of monads contain a higher-order function, e.g.:
 - $R^A \equiv H^A \Rightarrow A$, $R^A \equiv (A \Rightarrow P^Q) \Rightarrow P^A$, $R^A \equiv \forall B. (A \Rightarrow F^B) \Rightarrow F^B$
- Proving laws for these monads is easier with these tricks:

- 1 Instead of defining flm_R or ftn_R directly, use the Kleisli functions and \diamond_R

$$f:A \Rightarrow R^B \diamond_R g:B \Rightarrow R^C : A \Rightarrow R^C$$

- 2 Flip the arguments of the Kleisli functions: for example, instead of $A \Rightarrow R^B \equiv A \Rightarrow (B \Rightarrow P^Q) \Rightarrow P^B$, work with $(B \Rightarrow P^Q) \Rightarrow A \Rightarrow P^B$

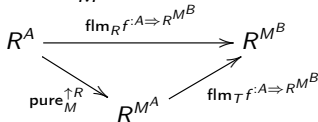
- 3 Use the Kleisli product for the nested monad: for example, to define

$$f:(B \Rightarrow P^Q) \Rightarrow A \Rightarrow P^B \diamond_R g:(C \Rightarrow P^Q) \Rightarrow B \Rightarrow P^C : (C \Rightarrow P^Q) \Rightarrow A \Rightarrow P^C$$

use the Kleisli product \diamond_P as $m:A \Rightarrow P^B \diamond_P n:B \Rightarrow P^C : A \Rightarrow P^C$ to obtain $B \Rightarrow P^Q$ from $B \Rightarrow P^C$ and $C \Rightarrow P^Q$, and then to get $A \Rightarrow P^C$

- A rigid monad R has the “compatibility law”: for any monad M ,

$$\text{flm}_R f:A \Rightarrow R^{M^B} = \text{pure}_M^{\uparrow R} \circ \text{flm}_T f:A \Rightarrow R^{M^B} \quad \text{where } T^\bullet \equiv R^{M^\bullet}$$



Rigid monads: Open questions

- What properties of $\text{fi} : (A \Rightarrow R^B) \Rightarrow R^{A \Rightarrow B}$ define rigid monads?
 - ▶ The law $\text{fi} \circ \text{fo} = \text{id}$ does not appear to be sufficient
 - ▶ Not clear if $\text{fi} \circ \text{fo} = \text{id}$ follows from monadicity of R^{M^*}
- A (generalized) functor from Kleisli category to “applicative” category?
 - ▶ Identity law: $\text{fi}(\text{pure}_R) = \text{pure}_R(\text{id})$ – this holds
 - ▶ Composition law: $\text{fi}(f \diamond_R g) = (p \times q \Rightarrow p \circ q)^{\uparrow R} (\text{fi } f \bowtie \text{fi } g)$

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

★ not clear whether this holds

Define the rigid monad transformer using fi ?

- Define \diamond_T by $f \diamond_T g \equiv \text{fo}((p \times q \Rightarrow p \diamond_M q)^{\uparrow R} (\text{fi } f \bowtie_R \text{fi } g))$

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

- ▶ not clear whether this holds

★ not clear whether associativity can be shown to hold in general

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 **lift** : $M^\bullet \rightsquigarrow L^\bullet \times M^\bullet$
- Making a monad out of functor composition or disjunction:
 - ▶ free monad over L^{M^\bullet} , $\text{Free}^{L^{M^\bullet}}$ – **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} : $F^A \equiv \forall B. (A \Rightarrow L^{M^B}) \Rightarrow L^{M^B}$ – no **lift**
- Codensity-L transformer: $\text{Cod}_L^{M,A} \equiv \forall B. (A \Rightarrow L^B) \Rightarrow L^{M^B}$ – no **lift**
 - ▶ applies the continuation transformer to $M^A \cong \forall B. (A \Rightarrow B) \Rightarrow M^B$
- 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 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 $\text{Id}^A \equiv A$ 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.
- 4 Show that the continuation monad is not rigid and does not compose with arbitrary other monads. Show that the list and state monads are not rigid.
- 5 Show that $\text{fo}(\text{pure}_P(f^{A \Rightarrow B})) = f \circ \text{pure}_P$ for any pointed functor P .
- 6 A rigid monad has a `pure` method because it is a monad, and also another `pure` method because it is a rigid functor. Show that these two `pure` methods must be the same.
- 7 Show that $T_{L_1}^{M,A} \times T_{L_2}^{M,A}$ is the transformer for the monad $L_1 \times L_2$.