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

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

- The code will work if we "unify" all effects in a new, larger monad
- Need to compute the type of new monad that contains all given effects

Combining monadic effects I. Trial and error

There are several ways of combining two monads into a new monad:

- If M_1^A and M_2^A are monads then $M_1^A \times M_2^A$ is also a monad
  - But $M_1^A \times M_2^A$ describes two separate values with two separate effects
- If M_1^A and M_2^A are monads then $M_1^A + M_2^A$ is usually not a monad
  - If it worked, it would be a choice between two different values / effects
- If M_1^A and M_2^A are monads then one of $M_1^{M_2^A}$ or $M_2^{M_1^A}$ is often a monad
- Examples and counterexamples for functor composition:
  - Combine $Z \Rightarrow A$ and List^A as $Z \Rightarrow \text{List}^A$
  - Combine `Future[A]` and `Option[A]` as `Future[Option[A]]`
  - But `Either[Z, Future[A]]` and `Option[Z ⇒ A]` are not monads
  - Neither `Future[State[A]]` nor `State[Future[A]]` are monads
- The order of effects matters when composition works both ways:
  - Combine `Either` ($M_1^A = Z + A$) and `Writer` ($M_2^A = W \times A$)
    - ⋆ as $Z + W \times A$ – either compute result and write a message, or all fails
    - ⋆ as $(Z + A) \times W$ – message is always written, but computation may fail
- Find a general way of defining a new monad with combined effects
- Derive properties required for the new monad

Combining monadic effects II. Lifting into a larger monad

If a "big monad" `BigM[A]` *somehow* combines all the needed effects:

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

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

- Example 1: combining as `BigM[A] = Future[Option[A]]` with liftings:

```
def lift₁[A]: Option[A] ⇒ Future[Option[A]] = Future.successful(_)
def lift₂[A]: Future[A] ⇒ Future[Option[A]] = _.map(x ⇒ Some(x))
```

- Example 2: combining as `BigM[A] = List[Try[A]]` with liftings:

```
def lift₁[A]: Try[A] ⇒ List[Try[A]] = x ⇒ List(x)
def lift₂[A]: List[A] ⇒ List[Try[A]] = _.map(x ⇒ Success(x))
```

Remains to be understood:

- Finding suitable laws for the liftings; checking that the laws hold
- Building a "big monad" out of "smaller" ones, with lawful liftings
  - Is this always possible? Unique? Are there alternative solutions?
- Ways of reducing the complexity of code; make liftings automatic

Laws for monad liftings I. Identity laws

Whatever identities we expect to hold for monadic programs must continue to hold after lifting `M₁` or `M₂` values into the "big monad" `BigM`

- We assume that `M₁`, `M₂`, and `BigM` already satisfy all the monad laws

Consider the various functor block constructions containing the liftings:

- Left identity law after `lift₁`

```
// Anywhere inside a for/yield:          // Must be equivalent to...
i ← lift₁(M₁.pure(x))                    i = x
j ← bigM(i)  // Any BigM value.          j ← bigM(x)
```

`lift₁(M₁.pure(x)).flatMap(b) = b(x)` — in terms of Kleisli composition ($\diamond$):

$\left(\text{pure}_{M_1} ; \text{lift}_1\right)^{:X\Rightarrow\text{BigM}^X} \diamond b^{:X\Rightarrow\text{BigM}^Y} = b$ 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, namely $\text{pure}_{\text{BigM}}$
- The two-sided unit element is always unique: $u = u \diamond u' = u'$
- 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, this...
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 $\text{lift}_1 ⨾ \text{flm}_{\text{BigM}} \left(q ⨾ \text{lift}_1\right) = \text{flm}_{M_1} q ⨾ \text{lift}_1$ – 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 $\text{lift}_1 ⨾ \text{fmap}_{\text{BigM}} \text{lift}_1 ⨾ \text{ftn}_{\text{BigM}} = \text{ftn}_{M_1} ⨾ \text{lift}_1$ – both sides are functions $M_1^{M_1^A} \Rightarrow \text{BigM}^A$
- In terms of Kleisli composition $\diamond_M$ it becomes the **composition law**:

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

- 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 $\text{BigM}^\bullet$
  - example: monad "interpreters" $M^A \Rightarrow N^A$ are monadic morphisms

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

$$\text{lift}_1 \fatsemi \text{fmap}_{\text{BigM}} f = \text{fmap}_{M_1} f \fatsemi \text{lift}_1$$

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

Derivation of the functor naturality law for lift_1:

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

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 $T_L^{M,\bullet}$ the **monad transformer for** L
  - 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 transformers are compositions $L^{M^\bullet}$ or $M^{L^\bullet}$, others are not
  - It is not known whether all monads L 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

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`: $L^A \Rightarrow \left(A \Rightarrow F^B\right) \Rightarrow F^{L^B}$ – 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 ⇒ M[A]
type EitherT[Z, M[_], A] = M[Either[Z, A]]
type WriterT[W, M[_], A] = M[(W, A)]
```

- `ReaderT` composes with 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)
- Is a given monad's transformer unique? (No.)
- 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 ⇒ Z) ⇒ 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 $T_{L_1}^{M,A} \times T_{L_2}^{M,A}$
  - "Contrafunctor-choice" $H^A \Rightarrow A$ – 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 `mrun`

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

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

- Monad laws must hold for T^A (must check this separately)
  - This depends on special properties of seq, e.g. $\text{pure}_L \fatsemi \text{seq} = \text{pure}_L^{\uparrow M}$ (L-identity); $\text{pure}_M^{\uparrow L} \fatsemi \text{seq} = \text{pure}_M$ (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, $\text{mrun} : \left(\forall B. M^B \Rightarrow N^B\right) \Rightarrow M^{L^A} \Rightarrow N^{L^A}$ is equal to id
  - Base runner, $\text{brun} : \left(\forall B. L^B \Rightarrow B\right) \Rightarrow M^{L^A} \Rightarrow M^A$ 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 $\text{pure}_L ; \text{seq} = \text{pure}_L^{\uparrow M}$ and $\text{pure}_M^{\uparrow L} ; \text{seq} = \text{pure}_M$ hold
- $\text{pure}_T ; \text{ftn}_T = \text{id}$. Proof: $(\text{pure}_M ; \text{pure}_L^{\uparrow M}) ; (\text{seq}^{\uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M} ; \text{ftn}_M) =$ $\text{pure}_M ; (\text{pure}_L ; \text{seq})^{\uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M} ; \text{ftn}_M = \text{pure}_M ; \text{pure}_L^{\uparrow M \uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M} ; \text{ftn}_M = \text{id}$
- $\text{pure}_T^{\uparrow T} ; \text{ftn}_T = \text{id}$. Proof: $\text{pure}_T = \text{pure}_M ; \text{pure}_L^{\uparrow M} = \text{pure}_L ; \text{pure}_M$ (naturality); for all f: $f^{\uparrow T} = f^{\uparrow L \uparrow M}$ and $f ; \text{pure}_M = \text{pure}_M ; f^{\uparrow M}$ (naturality); so $\text{pure}_T^{\uparrow T} ; \text{ftn}_T$ is $(\text{pure}_L ; \text{pure}_M)^{\uparrow L \uparrow M} ; (\text{seq}^{\uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M} ; \text{ftn}_M) = \text{pure}_L^{\uparrow L \uparrow M} ; \text{pure}_M^{\uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M} ; \text{ftn}_M =$ $\text{pure}_M^{\uparrow M} ; (\text{pure}_L^{\uparrow L} ; \text{ftn}_L)^{\uparrow M \uparrow M} ; \text{ftn}_M = \text{id}$ where we used naturality with $f = \text{pure}_L^{\uparrow L}$
- Identity law for lift: $\text{pure}_M ; \text{lift} = \text{pure}_T$ (this *is* the definition of pure_T)
- Composition law: $\text{lift} ; \text{lift}^{\uparrow T} ; \text{ftn}_T = \text{ftn}_M ; \text{lift}$. Proof: $\text{ftn}_M ; \text{pure}_L^{\uparrow M} = \text{pure}_L^{\uparrow M \uparrow M} ; \text{ftn}_M$ and $\text{pure}_L^{\uparrow M} ; (\text{pure}_L^{\uparrow M \uparrow L \uparrow M} ; \text{seq}^{\uparrow M}) ; \text{ftn}_L^{\uparrow M \uparrow M} ; \text{ftn}_M =$ $(\text{pure}_L^{\uparrow M} ; \text{seq}^{\uparrow M}) ; (\text{pure}_L^{\uparrow L \uparrow M \uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M}) ; \text{ftn}_M = \text{pure}_L^{\uparrow M \uparrow M} ; \text{ftn}_M$
- Identity law for blift: $\text{pure}_L ; \text{blift} = \text{pure}_T$. $(\text{pure}_L ; \text{pure}_M = \text{pure}_M ; \text{pure}_L^{\uparrow M})$
- Composition law: $\text{blift} ; \text{blift}^{\uparrow T} ; \text{ftn}_T = \text{ftn}_L ; \text{blift}$. Proof: $\text{pure}_M ; \text{pure}_M^{\uparrow L \uparrow M} ; (\text{seq}^{\uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M} ; \text{ftn}_M) = \text{pure}_M ; (\text{pure}_M^{\uparrow M} ; \text{ftn}_L^{\uparrow M \uparrow M}) ; \text{ftn}_M =$ $\text{pure}_M ; (\text{ftn}_L^{\uparrow M} ; \text{pure}_M^{\uparrow M}) ; \text{ftn}_M = \text{ftn}_L ; \text{pure}_M ; (\text{pure}_M^{\uparrow M} ; \text{ftn}_M) = \text{ftn}_L ; \text{blift}$
- Runner laws follow from naturality of id and fmap

Rigid monads I: 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

$$\text{seq} = \text{pure}_M^{\uparrow R \uparrow M} ; \text{pure}_R ; \text{ftn}_T$$

$$\begin{tikzcd} M^{R^A} \arrow[r, "\text{fmap}_M(\text{fmap}_R(\text{pure}_M))"'] & M^{R^{M^A}} \arrow[r, "\text{pure}_R"'] & R^{M^{R^{M^A}}} = T^{T^A} \arrow[r, "\text{ftn}_T"'] & T^A \equiv R^{M^A} \end{tikzcd}$$

- 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`: $\left(A \Rightarrow R^B\right) \Rightarrow R^{A \Rightarrow B}$
  - 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` $h^{:R^{A \Rightarrow B}} = x^{:A} \Rightarrow \left(f^{:A \Rightarrow B} \Rightarrow f\, x\right)^{\uparrow R} h$
- Nondegeneracy law: `fuseOut(fuseIn(x)) == x` or $\text{fi} \fatsemi \text{fo} = \text{id}$
- `fi` must be natural in both type parameters
  - Naturality: $\text{fi}\left(f \fatsemi g^{\uparrow R}\right) = \left(q^{:A \Rightarrow B} \Rightarrow q \fatsemi g\right)^{\uparrow R} (\text{fi}\, f)$ for $\forall\ f^{:A \Rightarrow R^B}$, $g^{:B \Rightarrow C}$ and $\text{fi}\,(f \fatsemi g) = \left(q^{:B \Rightarrow C} \Rightarrow f \fatsemi q\right)^{\uparrow R} (\text{fi}\, g)$ for $\forall\ f^{:A \Rightarrow B}$, $g^{:B \Rightarrow R^C}$
- Connection between monadic `flatMap` and applicative `ap` for monadic R:
  - $\text{flm} : \left(A \Rightarrow R^B\right) \Rightarrow R^A \Rightarrow R^B$
  - $\text{ap} : R^{A \Rightarrow B} \Rightarrow R^A \Rightarrow R^B$
  - The connection is $\text{flm} = \text{fi} \fatsemi \text{ap}$ and $\text{ap} = \text{fo} \fatsemi \text{flm}$
    - However, here we need to flip the order of R-effects in `ap`
  - Connection between ap and fo is $\text{fo}\, x\, a = \text{ap}\, x\, (\text{pure}\, a)$
- If $\text{flm} = \text{fi} \fatsemi \text{ap}$ then $\text{fi} \fatsemi \text{fo} = \text{id}$. Proof: set $x^{:R^{A \Rightarrow B}} = \text{fi}\, h^{:A \Rightarrow R^B}$ and get $\text{fo}\, x\, a = \text{ap}\ (\text{fi}\, h)\, (\text{pure}\, a) = \text{flm}\, h\, (\text{pure}\, a) = h\, a$, so $\text{fo}\, (\text{fi}\, h) = h$
- Conversely: If $\text{fi} \fatsemi \text{fo} = \text{id}$ and $\text{ap} = \text{fo} \fatsemi \text{flm}$ then $\text{flm} = \text{fi} \fatsemi \text{ap}$. Proof: $\text{fi} \fatsemi \text{ap} = \text{fi} \fatsemi \text{fo} \fatsemi \text{flm} = \text{flm}$

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`": $M^A \times \left(A \Rightarrow R^{M^B}\right) \Rightarrow R^{M^B}$
- Uptake monadic API: For a rigid functor $R^\bullet$, can implement a general refactoring function, `refactor`: $((A \Rightarrow B) \Rightarrow C) \Rightarrow \left(A \Rightarrow R^B\right) \Rightarrow R^C$, to transform a program $p\left(f^{A\Rightarrow B}\right) : C$ into $\tilde{p}\left(\tilde{f}^{:A\Rightarrow R^B}\right) : R^C$

Rigid monads II: Composed-outside transformers

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 $\text{pure}_T \equiv \text{pure}_M ; \text{pure}_R = \text{pure}_R ; \text{pure}_M^{\uparrow R}$
- $\text{ftn}_T : T^{T^A} \Rightarrow T^A$ 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 $\text{seq} = \text{fi} : \left(Z \Rightarrow R^A\right) \Rightarrow R^{Z \Rightarrow A}$
    - ★ 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_R
  - Base lifting, $\text{blift} : R^A \Rightarrow R^{M^A}$ is equal to $\text{pure}_M^{\uparrow R}$
  - Runner, $\text{mrun} : \left(\forall B. M^B \Rightarrow N^B\right) \Rightarrow R^{M^A} \Rightarrow R^{N^A}$ is equal to fmap_R
  - Base runner, $\text{brun} : \left(\forall B. R^B \Rightarrow B\right) \Rightarrow R^{M^A} \Rightarrow M^A$ 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?

* Rigid monads III: Some tricks for proving the laws

- Several classes of monads involve a higher-order function, e.g.:
  - $R^A \equiv H^A \Rightarrow A$, $R^A \equiv \left(A \Rightarrow P^Q\right) \Rightarrow P^A$, $R^A \equiv \forall B.\left(A \Rightarrow F^B\right) \Rightarrow F^B$
- 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 $A \Rightarrow R^B \equiv A \Rightarrow \left(B \Rightarrow P^Q\right) \Rightarrow P^B$, work with $\left(B \Rightarrow P^Q\right) \Rightarrow A \Rightarrow P^B$
  3. Use the Kleisli product for the nested monad: for example, to define
     $$f^{:\left(B\Rightarrow P^Q\right)\Rightarrow A\Rightarrow P^B} \diamond_R g^{:\left(C\Rightarrow P^Q\right)\Rightarrow B\Rightarrow P^C} : \left(C \Rightarrow P^Q\right) \Rightarrow A \Rightarrow P^C$$
     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 $A \Rightarrow P^C$
- Use "compatibility laws": for any monad M, and denoting $T^\bullet \equiv R^{M^\bullet}$,

$$\text{ftn}_R = \text{pure}_M^{\uparrow R} ; \text{ftn}_T \qquad \left(\text{ftn}_M\right)^{\uparrow R} = \text{pure}_R^{\uparrow T} ; \text{ftn}_T$$

$$\begin{tikzcd}
R^{R^{M^A}} \arrow[rr, "\text{ftn}_R"] \arrow[dr, "\text{pure}_M^{\uparrow R}"'] & & R^{M^A} \\
& R^{M^{R^{M^A}}} \arrow[ur, "\text{ftn}_T"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
R^{M^{M^A}} \arrow[rr, "\text{ftn}_M^{\uparrow R}"] \arrow[dr, "\text{pure}_R^{\uparrow T}"'] & & R^{M^A} \\
& R^{M^{R^{M^A}}} \arrow[ur, "\text{ftn}_T"'] &
\end{tikzcd}$$

Rigid monads IV: Open questions

- What properties of $\text{fi} : (A \Rightarrow R^B) \Rightarrow R^{A \Rightarrow B}$ define rigid monads?
  - The law $\text{fi} \fatsemi \text{fo} = \text{id}$ does not appear to be sufficient
  - Not clear if $\text{fi} \fatsemi \text{fo} = \text{id}$ follows from monadicity of $R^{M^\bullet}$
- 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: $\text{fi}\,(f \diamond_R g) = (p \times q \Rightarrow p \fatsemi q)^{\uparrow R}\,(\text{fi}\, f \bowtie \text{fi}\, g)$

$$\begin{tikzcd}
A \Rightarrow R^B \arrow[d, "\text{fi}"] & \times & B \Rightarrow R^C \arrow[d, "\text{fi}"] \arrow[rr, "\text{use } \diamond_R"'] & & A \Rightarrow R^C \arrow[d, "\text{fi}"] \\
R^{A \Rightarrow B} & \times & R^{B \Rightarrow C} \arrow[r, "\text{use } \bowtie"'] & R^{(A \Rightarrow B) \times (B \Rightarrow C)} \arrow[r, "\text{fmap}(\fatsemi)"'] & R^{A \Rightarrow C}
\end{tikzcd}$$

  - ★ not clear whether this holds

Define the rigid monad transformer using fi?

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

$$\begin{tikzcd}
\left(A \Rightarrow R^{M^B}\right) \arrow[d, "\text{fi}"] & \diamond_T & \left(B \Rightarrow R^{M^C}\right) \arrow[d, "\text{fi}"] \arrow[rr, "\text{define } \diamond_T \text{ as}"'] & & \left(A \Rightarrow R^{M^C}\right) \\
R^{A \Rightarrow M^B} & \bowtie_R & R^{B \Rightarrow M^C} \arrow[r] & R^{(A \Rightarrow M^B) \times (B \Rightarrow M^C)} \arrow[r, "\text{fmap}_R(\diamond_M)"'] & R^{A \Rightarrow M^C} \arrow[u, "\text{fo}"]
\end{tikzcd}$$

  - 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 $L^{M^\bullet}$: $F^A \equiv \forall B.\left(A \Rightarrow L^{M^B}\right) \Rightarrow L^{M^B}$ – 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 $\text{Id}^A \equiv A$ 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 $\text{lift}_1 ⨾ \text{fmap}_{\text{BigM}} \text{lift}_1 ⨾ \text{ftn}_{\text{BigM}} = \text{ftn}_{M_1} ⨾ \text{lift}_1$ from the flm-based law $\text{lift}_1 ⨾ \text{flm}_{\text{BigM}} \left(q ⨾ \text{lift}_1 \right) = \text{flm}_{M_1} q ⨾ \text{lift}_1$. 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 $\text{fo} \left(\text{pure}_P \left(f^{:A \Rightarrow B} \right) \right) = f ⨾ \text{pure}_P$ 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 $L_1 \times L_2$.