Chapter 7: Computations lifted to a functor context II. Monads

Part 2: Laws and structure of semimonads

Sergei Winitzki

Academy by the Bay

2018-03-11

Semimonad laws I: The intuitions

What properties of functor block programs do we expect to have?

- In `x ← c`, the value of `x` will *go over items* held in container `c`
- Manipulating items in container is followed by a generator:

```
x ← cont1                          y ← cont1
y = f(x)                             .map(x ⇒ f(x))
z ← cont2(y)                       z ← cont2(y)
```

```
cont1.flatMap(x ⇒ cont2(f(x))) = cont1.map(f).flatMap(y ⇒ cont2(y))
```

- Manipulating items in container is preceded by a generator:

```
x ← cont1                          x ← cont1
y ← cont2(x)                       z ← cont2(x)
z = f(y)                             .map(f)
```

```
cont1.flatMap(cont2).map(f) = cont1.flatMap(x ⇒ cont2(x).map(f))
```

- After `x ← cont`, further computations will use *all those* `x`

```
x ← cont                           y ← for { x ← cont
y ← p(x)                                      yy ← p(x) } yield yy
z ← cont2(y)                       z ← cont2(y)
```

```
cont.flatMap(x ⇒ p(x).flatMap(cont2)) = cont.flatMap(p).flatMap(cont2)
```

Semimonad laws II: The laws for `flatMap`

To get a more concise notation, use flm instead of `flatMap`

A **semimonad** S^A has $\text{flm}^{[S,A,B]} : (A \Rightarrow S^B) \Rightarrow S^A \Rightarrow S^B$ with 3 laws:

① $\text{flm}\,(f^{A \Rightarrow B} \circ g^{B \Rightarrow S^C}) = \text{fmap}\,f \circ \text{flm}\,g$ (naturality in A)

$$S^A \xrightarrow{\text{fmap}\,f^{A \Rightarrow B}} S^B \xrightarrow{\text{flm}\,g^{B \Rightarrow S^C}} S^C$$
$$S^A \xrightarrow{\text{flm}\,(f^{A \Rightarrow B} \circ g^{B \Rightarrow S^C})} S^C$$

② $\text{flm}\,(f^{A \Rightarrow S^B} \circ \text{fmap}\,g^{B \Rightarrow C}) = \text{flm}\,f \circ \text{fmap}\,g$ (naturality in B)

$$S^A \xrightarrow{\text{flm}\,f^{A \Rightarrow S^B}} S^B \xrightarrow{\text{fmap}\,g^{B \Rightarrow C}} S^C$$
$$S^A \xrightarrow{\text{flm}\,(f^{A \Rightarrow S^B} \circ \text{fmap}\,g^{B \Rightarrow C})} S^C$$

③ $\text{flm}\,(f^{A \Rightarrow S^B} \circ \text{flm}\,g^{B \Rightarrow S^C}) = \text{flm}\,f \circ \text{flm}\,g$ (associativity)

$$S^A \xrightarrow{\text{flm}\,f^{A \Rightarrow S^B}} S^B \xrightarrow{\text{flm}\,g^{B \Rightarrow S^C}} S^C$$
$$S^A \xrightarrow{\text{flm}\,\left(f^{A \Rightarrow S^B} \circ \text{flm}\,g^{B \Rightarrow S^C}\right)} S^C$$

Is there a shorter formulation of the laws?

Semimonad laws III: The laws for `flatten`

The methods `flatten` (denoted by ftn) and `flatMap` are equivalent:

$$\text{ftn}^{[S,A]} : S^{S^A} \Rightarrow S^A = \text{flm}^{[S,S^A,A]}(m^{S^A} \Rightarrow m)$$

$$\text{flm}\left(f^{A \Rightarrow S^B}\right) = \text{fmap } f \circ \text{ftn}$$

It turns out that `flatten` has only 2 laws:

1 $\text{fmap}\big(\text{fmap } f^{A \Rightarrow B}\big) \circ \text{ftn}^{[S,B]} = \text{ftn}^{[S,A]} \circ \text{fmap } f$ (naturality)

2 $\text{fmap}\left(\text{ftn}^{[S,A]}\right) \circ \text{ftn}^{[S,A]} = \text{ftn}^{[S,S^A]} \circ \text{ftn}^{[S,A]}$ (associativity)

Equivalence of a natural transformation and a "lifting"

- Equivalence of flm and ftn: ftn = flm (id); flm f = fmap $f \circ$ ftn
- We saw this before: deflate = fmapOpt (id); fmapOpt f = fmap $f \circ$ deflate
  - Is there a general pattern where two such functions are equivalent?
- Let tr : $F^{G^A} \Rightarrow F^A$ be a natural transformation (F and G are functors)
- Define ftr : $(A \Rightarrow G^B) \Rightarrow F^A \Rightarrow F^B$ by ftr f = fmap $f \circ$ tr
- It follows that tr = ftr (id), and we have equivalence between tr and ftr:

$$\text{tr} : F^{G^A} \Rightarrow F^A = \text{ftr}(m^{G^A} \Rightarrow m)$$

$$\text{ftr} \left(f^{A \Rightarrow G^B} \right) = \text{fmap } f \circ \text{tr}$$

$$F^A \xrightarrow[\text{ftr} \left(f^{A \Rightarrow G^B} \right)]{} F^B$$

$$F^A \xrightarrow{\text{fmap } f^{A \Rightarrow G^B}} F^{G^B} \xrightarrow{\text{tr}} F^B$$

- An automatic law for ftr ("naturality in A") follows from the definition:
  fmap $g \circ$ ftr f = fmap $g \circ$ fmap $f \circ$ tr = fmap $(g \circ f) \circ$ tr = ftr $(g \circ f)$
  - This is why tr has *one law fewer* than ftr
- To demonstrate equivalence in the direction ftr → tr: start with an arbitrary ftr satisfying "naturality in A", then obtain tr = ftr (id) from it, then verify ftr f = fmap $f \circ$ tr with that tr: fmap $f \circ$ ftr (id) = ftr $(f \circ id)$ = ftr f

Denote for brevity $q^\uparrow \equiv \mathsf{fmap}^{[S]}\, q$ for any function q

Express $\mathsf{flm}\, f = f^\uparrow \circ \mathsf{ftn}$ and substitute that into flm's 3 laws:

1. $\mathsf{flm}\,(f \circ g) = f^\uparrow \circ \mathsf{flm}\, g$ gives $(f \circ g)^\uparrow \circ \mathsf{ftn} = f^\uparrow \circ g^\uparrow \circ \mathsf{ftn}$
   – this law holds automatically due to functor composition law

2. $\mathsf{flm}\,(f \circ g^\uparrow) = \mathsf{flm}\, f \circ g^\uparrow$ gives $(f \circ h)^\uparrow \circ \mathsf{ftn} = f^\uparrow \circ \mathsf{ftn} \circ h$;
   using the functor composition law, we reduce this to
   $h^\uparrow \circ \mathsf{ftn} = \mathsf{ftn} \circ h$ – this is the naturality law

3. $\mathsf{flm}\,(f \circ \mathsf{flm}\, g) = \mathsf{flm}\, f \circ \mathsf{flm}\, g$ with functor composition law gives
   $f^\uparrow \circ g^{\uparrow\uparrow} \circ \mathsf{ftn}^\uparrow \circ \mathsf{ftn} = f^\uparrow \circ \mathsf{ftn} \circ g^\uparrow \circ \mathsf{ftn}$; using ftn's naturality and omitting
   the common factor $f^\uparrow \circ g^{\uparrow\uparrow}$, we get $\mathsf{ftn}^\uparrow \circ \mathsf{ftn} = \mathsf{ftn} \circ \mathsf{ftn}$ – associativity law

- `flatten` has the simplest type signature *and* the fewest laws
- It is usually easy to check naturality!
  - **Parametricity theorem**: Any *pure, fully parametric* code for a function of type $F^A \Rightarrow G^A$ will implement a natural transformation
- Checking `flatten`'s associativity needs *a lot* more work!

The `cats` library has a `FlatMap` type class, defining `flatten` via `flatMap`

Checking the associativity law for standard monads

- Implement `flatten` for these functors and check the laws (see code):
  - `Option` monad: $F^A \equiv 1 + A$; ftn $: 1 + (1 + A) \Rightarrow 1 + A$
  - `Either` monad: $F^A \equiv Z + A$; ftn $: Z + (Z + A) \Rightarrow Z + A$
  - `List` monad: $F^A \equiv \text{List}^A$; ftn $: \text{List}^{\text{List}^A} \Rightarrow \text{List}^A$
  - Writer monad: $F^A \equiv A \times W$; ftn $: (A \times W) \times W \Rightarrow A \times W$
  - Reader monad: $F^A \equiv R \Rightarrow A$; ftn $: (R \Rightarrow (R \Rightarrow A)) \Rightarrow R \Rightarrow A$
  - State: $F^A \equiv S \Rightarrow A \times S$; ftn $: (S \Rightarrow (S \Rightarrow A \times S) \times S) \Rightarrow S \Rightarrow A \times S$
  - Continuation monad: $F^A \equiv (A \Rightarrow R) \Rightarrow R$;
    ftn $: ((((A \Rightarrow R) \Rightarrow R) \Rightarrow R) \Rightarrow R) \Rightarrow (A \Rightarrow R) \Rightarrow R$
- Code implementing these `flatten` functions is *fully parametric* in A
  - Naturality of these functions follows from parametricity theorem
  - Associativity needs to be checked for each monad!
- Example of a useful semimonad that is *not* a full monad:
  - $F^A \equiv A \times V \times W$; ftn $((a \times v_1 \times w_1) \times v_2 \times w_2) = a \times v_1 \times w_1$
- Examples of *non-associative* (i.e. wrong) implementations of `flatten`:
  - $F^A \equiv A \times W \times W$; ftn $((a \times v_1 \times v_2) \times w_1 \times w_2) = a \times w_2 \times w_1$
  - $F^A \equiv \text{List}^A$, but `flatten` concatenates the nested lists in reverse order

Motivation for monads

- Monads represent values with a "special computational context"
- Specific monads will have methods to create various contexts
- Monadic composition will "combine" the contexts associatively
- It is generally useful to have an "empty context" available

$$\text{pure} : A \Rightarrow M^A$$

- Combining empty context with another context works as a no-op
- Empty context is followed by a generator:

```
y ← pure(x)                          y = x
z ← cont(y)                          z ← cont(y)
pure(x).flatMap(y ⇒ cont(y)) = cont(x)      pure ∘ flm f = f − left identity
```

- Empty context is preceded by a generator:

```
x ← cont                             x ← cont
y ← pure(x)                          y = x
cont.flatMap(x ⇒ pure(x)) = cont        flm (pure) = id − right identity
```
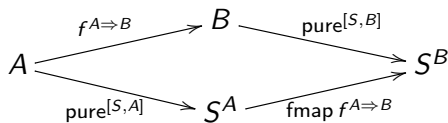
The monad laws formulated in terms of `pure` and `flatten`

- Naturality law for `pure`: $f \circ \text{pure} = \text{pure} \circ f^\uparrow$

$$
\begin{array}{ccc}
 & \overset{f^{A \Rightarrow B}}{\nearrow} & B \overset{\text{pure}^{[S,B]}}{\searrow} \\
A & & S^B \\
 & \underset{\text{pure}^{[S,A]}}{\searrow} & S^A \underset{\text{fmap } f^{A \Rightarrow B}}{\nearrow}
\end{array}
$$

- Left identity: $\text{pure} \circ \text{flm } f = \text{pure} \circ f^\uparrow \circ \text{ftn} = f \circ \text{pure} \circ \text{ftn} = f$
  requires that $\text{pure} \circ \text{ftn} = \text{id}$ (both sides applied to S^A)

$$
\begin{array}{ccc}
 & \overset{\text{pure}^{[S,S^A]}}{\nearrow} S^{S^A} \overset{\text{ftn}}{\searrow} \\
S^A & \underset{\text{id}}{\longrightarrow} & S^A
\end{array}
$$
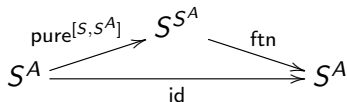
- Right identity: $\text{flm}(\text{pure}) = \text{pure}^\uparrow \circ \text{ftn} = \text{id}$

$$
\begin{array}{ccc}
 & \overset{\text{fmap}(\text{pure}^{[S,A]})}{\nearrow} S^{S^A} \overset{\text{ftn}}{\searrow} \\
S^A & \underset{\text{id}}{\longrightarrow} & S^A
\end{array}
$$
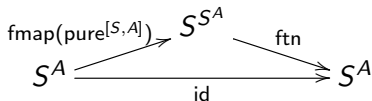
Formulating laws via Kleisli functions

- Recall: we formulated the laws of filterables via `fmapOpt`
  - fmapOpt : $(A \Rightarrow 1 + B) \Rightarrow S^A \Rightarrow S^B$
- And then we had to compose functions of types $A \Rightarrow 1 + B$ with $\diamond_{\mathsf{Opt}}$
- Here we have flm : $(A \Rightarrow S^B) \Rightarrow S^A \Rightarrow S^B$ instead of `fmapOpt`
- Can we compose **Kleisli functions** with "twisted" types $A \Rightarrow S^B$?
- Use flm to define **Kleisli composition**: $f^{A \Rightarrow S^B} \diamond g^{B \Rightarrow S^C} \equiv f \circ \mathsf{flm}\, g$
- Define **Kleisli identity** $\mathsf{id}_\diamond$ of type $A \Rightarrow S^A$ as $\mathsf{id}_\diamond \equiv \mathsf{pure}$
- Composition law: $\mathsf{flm}\,(f \diamond g) = \mathsf{flm}\, f \circ \mathsf{flm}\, g$ (same as for `fmapOpt`)
  - Shows that `flatMap` is a "lifting" of $A \Rightarrow S^B$ to $S^A \Rightarrow S^B$
- These laws are similar to functor "lifting" laws...
  - except that $\diamond$ is used for composing Kleisli functions
- What are the properties of $\diamond$?
  - Exactly similar to the properties of function composition $f \circ g$

Reformulate flm's laws in terms of the $\diamond$ operation:

- flm's left and right identity laws: $\mathsf{pure} \diamond f = f$ and $f \diamond \mathsf{pure} = f$
- Associativity law: $(f \diamond g) \diamond h = f \diamond (g \diamond h)$
  - Follows from the flm law: $f \circ \mathsf{flm}\,(g \circ \mathsf{flm}\,h) = f \circ \mathsf{flm}\, g \circ \mathsf{flm}\, h$

From Kleisli back to `flatMap`

Compare different "liftings" seen so far:

Category	Function type	Identity	Composition
plain functions	$A \Rightarrow B$	id : $A \Rightarrow A$	$f^{A \Rightarrow B} \circ g^{B \Rightarrow C}$
lifted to F	$F^A \Rightarrow F^B$	id : $F^A \Rightarrow F^A$	$f^{F^A \Rightarrow F^B} \circ g^{F^B \Rightarrow F^C}$
Kleisli over F	$A \Rightarrow F^B$	pure : $A \Rightarrow F^A$	$f^{A \Rightarrow F^B} \diamond g^{B \Rightarrow F^C}$

Category axioms: identity and associativity for composition

General **functor**: a "lifting" maps functions from one category to another

- Functor laws: "lifting" must preserve identity and composition

Reformulate `map` and `flatMap` in terms of the $\diamond$ operation:

- Define `flatMap` through Kleisli composition: flm $f^{A \Rightarrow S^B} \equiv \text{id}^{S^A \Rightarrow S^A} \diamond f$
- Define `flatten` through Kleisli: ftn $\equiv \text{id}^{S^{S^A} \Rightarrow S^{S^A}} \diamond \text{id}^{S^A \Rightarrow S^A}$
- Express `fmap` through Kleisli: fmap $f \equiv (\text{fmap id}) \diamond (f \circ \text{pure})$
- Need two additional laws to connect $\diamond$ and $\circ$:
  - Left naturality: $f^{A \Rightarrow B} \circ g^{B \Rightarrow S^C} = (f \circ \text{pure}) \diamond g$
  - Right naturality: $f^{A \Rightarrow B} \circ \text{fmap } g^{B \Rightarrow S^C} = f \diamond (g \circ \text{pure})$
    - ⋆ With these laws, monad laws follow from category axioms for Kleisli

Structure of semigroups and monoids

- Semimonad contexts are combined associatively, as in a semigroup
- A full monad includes an "empty" context, i.e. the identity element
- Semigroup with an identity element is a monoid

Some constructions of semigroups and monoids:

1. Any type Z is a semigroup with operation $z_1 \circledast z_2 = z_1$ (or z_2)
2. $1 + S$ is a monoid if S is (at least) a semigroup
3. List^A is a monoid (for any type A), also Seq^A etc.
4. The function type $A \Rightarrow A$ is a monoid (for any type A)
   - The operation $f \circledast g$ is either $f \circ g$ or $g \circ f$
5. Any totally ordered type is a monoid, with $\circledast$ defined as max or min
6. $S_1 \times S_2$ is a semigroup (monoid) if S_1, S_2 are semigroups (monoids)
7. $S \times P$ is a semigroup (monoid) if S is a semigroup (monoid) such that S acts on P. ("Twisted product.") Example: $(A \Rightarrow A) \times A$
   - The "action" is $a : S \Rightarrow P \Rightarrow P$ such that $a(s_1) \circ a(s_2) = a(s_1 \circledast s_2)$.
8. $Z \Rightarrow S$ is a semigroup/monoid, for any Z, if S is a semigroup/monoid

- There are other examples: Int, String, Set^A, Akka routes, ...
- Non-examples: trees; $S_1 + S_2$ where $S_{1,2}$ are different monoids

Structure of (semi)monads

How to recognize a (semi)monad by its type? Open question!

Intuition from `flatten`: reshuffle data in F^{F^A} to fit into F^A

Some constructions of exponential-polynomial semimonads:

1. $F^A \equiv Z$ (constant functor) for a fixed type Z
   - For a full monad, need to choose $Z = 1$
2. $F^A \equiv A \times G^A$ for any functor G^A (a full monad only if $G^A \equiv 1$)
3. $F^A \equiv Z + A \times W$ for a fixed type Z and a semigroup W
   - For a full monad, need W to be a monoid
4. $F^A \equiv G^{Z+A \times W}$ if $Z + A \times W$ is a (semi)monad
5. $F^A \equiv G^A \times H^A$ for any (semi)monads G^A and H^A
6. $F^A \equiv A + G^A$ for any semimonad G^A
7. $F^A \equiv A + G^{F^A}$ (recursive) for any functor G^A (**free monad** over G)
8. $F^A \equiv G^A + G^{F^A}$ (recursive) for any functor G^A (semimonad only!)
9. $F^A \equiv R \to G^A$ for any (semi)monad G^A
10. $F^A \equiv H^A \Rightarrow A \times G^A$ for any contrafunctor H^A and functor G^A
    - For a full monad, need to set $G^A \equiv 1$

z

Exercises II

1. For an arbitrary monad M^A, show that the functor
   $F^A \equiv \text{Boolean} \times M^A$ can be defined as a semimonad but not a monad.

2. If W and R are arbitrary fixed types, which of the functors can be made into a semimonad: $F^A \equiv W \times (R \Rightarrow A)$, $G^A = R \Rightarrow (W \times A)$?

3. Suppose a functor F^A has a natural transformation $\text{ex}^{[A]} : F^A \Rightarrow A$ that "extracts the value" from F^A. Would F^A be a semimonad if we defined `flatten` as ftn = $\text{ex}^{[F^A]}$ or ftn = fmap ex?

4. A programmer implemented the `fmap` method for the type constructor $F^A \equiv A \times (A \Rightarrow Z)$ as
   ```
   def fmap[A,B](f: A⇒B): ((A, A⇒Z)) ⇒ (B, B⇒Z) =
     { case(a, az) ⇒ (f(a), (_: B) ⇒ az(a)) }
   ```
   Show that this implementation fails to satisfy the functor laws.

5. Implement the `flatten` and `pure` methods for the type constructor $F^A \equiv 1 + A \times A$ (`type F[A] = Option[(A, A)]`) in at least two different ways, and show that the monad laws always fail.

6. Implement the monad methods for $F^A \equiv (Z \Rightarrow 1 + A) \times \text{List}^A$ using the known monad constructions.

(continued from the previous slide)

7. Check the identity laws for monad construction 6, $F^A \equiv A + G^A$, when pure_F is defined as $\text{id}^{[A]} + 0$ (given that G^A is a monad). Show that the identity laws fail if pure_F is defined as $0 + \text{pure}_G$.

8. Show that $F^A = (P \Rightarrow A) + (Q \Rightarrow A)$ is not a semimonad when P and Q are arbitrary, different types.