

# 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  $x \leftarrow c$ , the value of  $x$  will *go over items* held in container  $c$
- Manipulating items in container is followed by a generator:

$x \leftarrow \text{cont1}$	$y \leftarrow \text{cont1}$
$y = f(x)$	$\text{.map}(x \Rightarrow f(x))$
$z \leftarrow \text{cont2}(y)$	$z \leftarrow \text{cont2}(y)$

$\text{cont1.flatMap}(x \Rightarrow \text{cont2}(f(x))) = \text{cont1.map}(f).\text{flatMap}(y \Rightarrow \text{cont2}(y))$

- Manipulating items in container is preceded by a generator:

$x \leftarrow \text{cont1}$	$x \leftarrow \text{cont1}$
$y \leftarrow \text{cont2}(x)$	$z \leftarrow \text{cont2}(x)$
$z = f(y)$	$\text{.map}(f)$

$\text{cont1.flatMap}(\text{cont2}).\text{map}(f) = \text{cont1.flatMap}(x \Rightarrow \text{cont2}(x).\text{map}(f))$

- Within a generator, `for {...} yield` can be inlined:

$x \leftarrow \text{cont}$	$yy \leftarrow \text{for } \{ x \leftarrow \text{cont}$
$y \leftarrow p(x)$	$y \leftarrow p(x) \} \text{ yield } y$
$z \leftarrow \text{cont2}(y)$	$z \leftarrow \text{cont2}(yy)$

$\text{cont.flatMap}(x \Rightarrow p(x).\text{flatMap}(\text{cont2})) = \text{cont.flatMap}(p).\text{flatMap}(\text{cont2})$

# Semimonad laws II: The laws for `flatMap`

For brevity, write `flm` instead of `flatMap`

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

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

$$\begin{array}{ccc} & S^B & \\ \text{fmap } f^{A \Rightarrow B} \nearrow & & \searrow \text{flm } g^{B \Rightarrow S^C} \\ S^A & \xRightarrow{\text{flm } (f^{A \Rightarrow B} \circ g^{B \Rightarrow S^C})} & S^C \end{array}$$

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

$$\begin{array}{ccc} & S^B & \\ \text{flm } f^{A \Rightarrow S^B} \nearrow & & \searrow \text{fmap } g^{B \Rightarrow C} \\ S^A & \xRightarrow{\text{flm } (f^{A \Rightarrow S^B} \circ \text{fmap } g^{B \Rightarrow C})} & S^C \end{array}$$

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

$$\begin{array}{ccc} & S^B & \\ \text{flm } f^{A \Rightarrow S^B} \nearrow & & \searrow \text{flm } g^{B \Rightarrow S^C} \\ S^A & \xRightarrow{\text{flm } (f^{A \Rightarrow S^B} \circ \text{flm } g^{B \Rightarrow S^C})} & S^C \end{array}$$

Is there a shorter and clearer formulation of these laws?

## Semimonad laws III: The laws for `flatten`

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

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

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

A commutative triangle diagram. The top-left node is  $S^A$ , the top-right node is  $S^{S^B}$ , and the bottom-right node is  $S^B$ . An arrow labeled  $\text{fmap } f^{A \Rightarrow S^B}$  points from  $S^A$  to  $S^{S^B}$ . An arrow labeled  $\text{ftn}$  points from  $S^{S^B}$  to  $S^B$ . A bottom arrow labeled  $\text{flm}(f^{A \Rightarrow S^B})$  points from  $S^A$  to  $S^B$ .

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

- ❶  $\text{fmap}(\text{fmap } f^{A \Rightarrow B}) \circ \text{ftn}^{[B]} = \text{ftn}^{[A]} \circ \text{fmap } f$  (naturality)

A commutative diagram with four nodes. The top-left node is  $S^{S^A}$ , the top-right node is  $S^{S^B}$ , the bottom-left node is  $S^A$ , and the bottom-right node is  $S^B$ . An arrow labeled  $\text{fmap}(\text{fmap } f^{A \Rightarrow B})$  points from  $S^{S^A}$  to  $S^{S^B}$ . An arrow labeled  $\text{ftn}^{[B]}$  points from  $S^{S^B}$  to  $S^B$ . An arrow labeled  $\text{ftn}^{[A]}$  points from  $S^{S^A}$  to  $S^A$ . An arrow labeled  $\text{fmap } f^{A \Rightarrow B}$  points from  $S^A$  to  $S^B$ .

- ❷  $\text{fmap}(\text{ftn}^{[A]}) \circ \text{ftn}^{[A]} = \text{ftn}^{[S^A]} \circ \text{ftn}^{[A]}$  (associativity)

A commutative diagram with four nodes. The top-left node is  $S^{S^{S^A}}$ , the top-right node is  $S^{S^A}$ , the bottom-left node is  $S^{S^A}$ , and the bottom-right node is  $S^A$ . An arrow labeled  $\text{fmap}(\text{ftn}^{[A]})$  points from  $S^{S^{S^A}}$  to  $S^{S^A}$ . An arrow labeled  $\text{ftn}^{[A]}$  points from  $S^{S^A}$  to  $S^A$ . An arrow labeled  $\text{ftn}^{[S^A]}$  points from  $S^{S^{S^A}}$  to  $S^{S^A}$ . An arrow labeled  $\text{ftn}^{[A]}$  points from  $S^{S^A}$  to  $S^A$ .

# Equivalence of a natural transformation and a “lifting”

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

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

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

$$\begin{array}{ccc} & F^{G^B} & \\ \text{fmap } f^{A \Rightarrow G^B} \nearrow & & \searrow \text{tr} \\ F^A & \xrightarrow{\text{ftr}(f^{A \Rightarrow G^B})} & F^B \end{array}$$

- An automatic law for  $\text{ftr}$  (“naturality in  $A$ ”) follows from the definition:  
 $\text{fmap } g \circ \text{ftr } f = \text{fmap } g \circ \text{fmap } f \circ \text{tr} = \text{fmap}(g \circ f) \circ \text{tr} = \text{ftr}(g \circ f)$ 
  - ▶ This is why  $\text{tr}$  always has *one law fewer* than  $\text{ftr}$
- To demonstrate equivalence in the direction  $\text{ftr} \rightarrow \text{tr}$ : Start with an arbitrary  $\text{ftr}$  satisfying “naturality in  $A$ ”, then obtain  $\text{tr} = \text{ftr}(\text{id})$  from it, then verify  $\text{ftr } f = \text{fmap } f \circ \text{tr}$  with that  $\text{tr}$ ;  $\text{fmap } f \circ \text{ftr}(\text{id}) = \text{ftr}(f \circ \text{id}) = \text{ftr } f$

## Semimonad laws IV: Deriving the laws for `flatten`

Denote for brevity  $q^\uparrow \equiv \text{fmap } q$  for any function  $q$  (“lifting”  $q^{A \Rightarrow B}$  to  $S$ )  
Express  $\text{flm } f = f^\uparrow \circ \text{ftn}$  and substitute that into  $\text{flm}$ ’s 3 laws:

- 1  $\text{flm } (f \circ g) = f^\uparrow \circ \text{flm } g$  gives  $(f \circ g)^\uparrow \circ \text{ftn} = f^\uparrow \circ g^\uparrow \circ \text{ftn}$   
– this law holds automatically due to functor composition law
  - 2  $\text{flm } (f \circ g^\uparrow) = \text{flm } f \circ g^\uparrow$  gives  $(f \circ h)^\uparrow \circ \text{ftn} = f^\uparrow \circ \text{ftn} \circ h$ ;  
using the functor composition law, we reduce this to  
 $h^\uparrow \circ \text{ftn} = \text{ftn} \circ h$  – this is the naturality law
  - 3  $\text{flm } (f \circ \text{flm } g) = \text{flm } f \circ \text{flm } g$  with functor composition law gives  
 $f^\uparrow \circ g^{\uparrow\uparrow} \circ \text{ftn}^\uparrow \circ \text{ftn} = f^\uparrow \circ \text{ftn} \circ g^\uparrow \circ \text{ftn}$ ; using  $\text{ftn}$ ’s naturality and omitting  
the common factor  $f^\uparrow \circ g^{\uparrow\uparrow}$ , we get  $\text{ftn}^\uparrow \circ \text{ftn} = \text{ftn} \circ \text{ftn}$  – 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:  $F^A \equiv 1 + A$ ;  $\text{ftn} : 1 + (1 + A) \Rightarrow 1 + A$
  - ▶ `Either` monad:  $F^A \equiv Z + A$ ;  $\text{ftn} : Z + (Z + A) \Rightarrow Z + A$
  - ▶ `List` monad:  $F^A \equiv \text{List}^A$ ;  $\text{ftn} : \text{List}^{\text{List}^A} \Rightarrow \text{List}^A$
  - ▶ `Writer` monad:  $F^A \equiv A \times W$ ;  $\text{ftn} : (A \times W) \times W \Rightarrow A \times W$
  - ▶ `Reader` monad:  $F^A \equiv R \Rightarrow A$ ;  $\text{ftn} : (R \Rightarrow (R \Rightarrow A)) \Rightarrow R \Rightarrow A$
  - ▶ `State`:  $F^A \equiv S \Rightarrow A \times S$ ;  $\text{ftn} : (S \Rightarrow (S \Rightarrow A \times S)) \times S \Rightarrow S \Rightarrow A \times S$
  - ▶ `Continuation` monad:  $F^A \equiv (A \Rightarrow R) \Rightarrow R$ ;  
 $\text{ftn} : (((A \Rightarrow R) \Rightarrow R) \Rightarrow R) \Rightarrow (A \Rightarrow R) \Rightarrow R$
- 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:
  - ▶  $F^A \equiv A \times V \times W$ ;  $\text{ftn}((a \times v_1 \times w_1) \times v_2 \times w_2) = a \times v_1 \times w_2$
- Examples of *non-associative* (i.e. wrong) implementations of `flatten`:
  - ▶  $F^A \equiv A \times W \times W$ ;  $\text{ftn}((a \times v_1 \times v_2) \times w_1 \times w_2) = a \times w_2 \times w_1$
  - ▶  $F^A \equiv \text{List}^A$ , 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:

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

Adding the empty context to another context should be 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`:  $f \circ \text{pure} = \text{pure} \circ \text{fmap } f$



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



- Right identity:  $\text{flm}(\text{pure}) = \text{fmap}(\text{pure}) \circ \text{ftn} = \text{id}^{S^A \Rightarrow S^A}$



# Formulating laws via Kleisli functions

- Recall: we formulated the laws of filterables via `fmapOpt`
  - type signature of `fmapOpt` :  $(A \Rightarrow 1 + B) \Rightarrow S^A \Rightarrow S^B$
  - and then we had to compose functions of types  $A \Rightarrow 1 + B$  via  $\diamond_{\text{Opt}}$
- 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**:  $f^{A \Rightarrow S^B} \diamond g^{B \Rightarrow S^C} \equiv f \circ \text{flm } g$
- Define **Kleisli identity**  $\text{id}_\diamond$  of type  $A \Rightarrow S^A$  as  $\text{id}_\diamond \equiv \text{pure}$
- Composition law:  $\text{flm } (f \diamond g) = \text{flm } f \circ \text{flm } g$  (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:  $\text{pure} \diamond f = f$  and  $f \diamond \text{pure} = f$
- Associativity law:  $(f \diamond g) \diamond h = f \diamond (g \diamond h)$ 
  - Follows from the `flm` law:  $f \circ \text{flm } (g \circ \text{flm } h) = f \circ \text{flm } g \circ \text{flm } h$

## \* Motivation for categories and functors

Compare different “liftings” seen so far, and generalize

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

Category theory generalizes this situation

**Category:** a certain class of “twisted functions”  $A \rightsquigarrow B$  called **morphisms**

- For any two morphisms  $f^{A \rightsquigarrow B}$  and  $g^{B \rightsquigarrow C}$  the **composition** morphism  $f \diamond g$  of type  $A \rightsquigarrow C$  must exist
- For each type  $A$ , the **identity** morphism  $\text{id}_\diamond$  of type  $A \rightsquigarrow A$  must exist
- Composition respects identity:  $\text{id}_\diamond \diamond f = f$  and  $f \diamond \text{id}_\diamond = f$
- Composition is associative:  $(f \diamond g) \diamond h = f \diamond (g \diamond h)$

General **functor**: a map from one category to another

- A functor must **fmap** each morphism from one category to the other
- Functor laws: **fmap** must preserve identity and composition
  - ▶ What we call “functor” is called **endofunctor** in category theory
  - ▶ An endofunctor’s **fmap** goes from plain functions to  $F$ -lifted functions

## \* From Kleisli back to `flatMap`

The Kleisli functions,  $A \rightsquigarrow B \equiv A \Rightarrow S^B$ , form a category iff  $S$  is a monad

- `map` and `flatMap` are computationally equivalent to Kleisli composition:
  - ▶ Define `flatMap` through Kleisli:  $\text{flm } f^{A \Rightarrow S^B} \equiv \text{id}^{S^A \Rightarrow S^A} \diamond f$
  - ▶ Require two additional laws that connect  $\diamond$ , `fmap`, and  $\circ$ :
    - ★ Left naturality:  $f^{A \Rightarrow B} \circ g^{B \Rightarrow S^C} = (f \circ \text{pure}) \diamond g$
    - ★ Right naturality:  $f^{A \Rightarrow S^B} \circ \text{fmap } g^{B \Rightarrow C} = f \diamond (g \circ \text{pure})$
  - ▶ So, can define `fmap` through Kleisli:  $\text{fmap } g^{A \Rightarrow B} \equiv \text{id}^{S^A \Rightarrow S^A} \diamond (g \circ \text{pure})$

The laws for `pure` and `flatMap` then follow from category axioms for Kleisli:

- Left and right identity laws follow from  $\text{id} \diamond \text{pure} = \text{id}$  and  $\text{pure} \diamond f = f$
- Associativity for `flatMap` follows from  $(\text{id} \diamond f) \diamond g = \text{id} \diamond (f \diamond g)$
- Use “left naturality”, get:  $(f \circ g) \diamond h = (f \circ \text{pure}) \diamond g \diamond h = f \circ (g \diamond h)$
- Naturality for `pure`:  $\text{pure} \circ \text{fmap } f = \text{pure} \diamond (f \circ \text{pure}) = f \circ \text{pure}$
- Define `flatten`:  $\text{ftn} = \text{id}^{S^{S^A} \Rightarrow S^{S^A}} \diamond \text{id}^{S^A \Rightarrow S^A}$
- Naturality for `flatten`:  $\text{ftn} \circ \text{fmap } f = \text{id} \diamond \text{id} \diamond (f \circ \text{pure}) = \text{id} \diamond \text{fmap } f$   
and  $\text{fmap } (\text{fmap } f) \circ \text{ftn} = \text{id} \diamond ((\text{fmap } f) \circ \text{pure}) \circ \text{id} \diamond \text{id} = \text{id} \diamond \text{fmap } f$

# 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 (see code):

- 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 (or  $S \equiv 0$ )
  - 3  $\text{List}^A$  is a monoid (for any type  $A$ ), also  $\text{Seq}^A$  etc.
  - 4 The function type  $A \Rightarrow A$  is a monoid (for any type  $A$ )
    - ▶ The operation  $f \circledast g$  can be 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  $S_1 + S_2$  is a semigroup (monoid) if  $S_1, S_2$  are semigroups (monoids)
  - 8  $M^S$  is a monoid if  $S$  is a monoid and  $M$  is a monad
  - 9  $S \times P$  is a semigroup if  $S$  is a semigroup that has an **action on**  $P$ .
    - ▶ The “action” is  $\alpha : S \Rightarrow P \Rightarrow P$  such that  $\alpha(s_2) \circ \alpha(s_1) = \alpha(s_1 \circledast s_2)$ .
    - ▶  $S \times P$  is a “twisted product.” Examples:  $(A \Rightarrow A) \times A$ ;  $\text{Bool} \times (1 + A)$ .
- Other examples of monoids:  $\text{Int}$  (many),  $\text{String}$ ,  $\text{Set}^A$ , Akka's [Route](#)

# 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 (semi)monads:

- 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  $F^A \equiv G^{Z+A \times W}$  is a (semi)monad 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  $F^A \equiv A + G^A$  is a monad for a semimonad  $G^A$  (**free pointed** over  $G$ )
- 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  $F^A \equiv R \Rightarrow G^A$  is a (semi)monad 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  $G^A \equiv 1$ , i.e.  $F^A \equiv H^A \Rightarrow A$

## Exercises II

- 1 Show that  $M[P]$  is a monoid if  $M[_]$  is a monad and  $P$  is a monoid.
- 2 A framework implements a “route” type  $R$  as  $R \equiv Q \Rightarrow (E + S)$ , where  $Q$  is a query,  $E$  is an error response, and  $S$  is a success response. A server is defined as a “sum” of several routes. For a given query  $Q$ , the response is the first route (if it exists) that yields a success. Implement the route “summation” operation and show that it makes  $R$  into a semigroup. What would be necessary to make  $R$  into a monoid?
- 3 Verify the associativity law for the semimonad  $F^A \equiv Z + \text{Bool} \times A$ .
- 4 Show that the functor  $F^A \equiv \text{Boolean} \times M^A$  (where  $M^A$  is an arbitrary monad) can be made into a semimonad but not into a monad.
- 5 If  $W$  and  $R$  are arbitrary fixed types, which of the functors can be made into a semimonad:  $F^A \equiv W \times (R \Rightarrow A)$ ,  $G^A \equiv R \Rightarrow (W \times A)$ ?
- 6 Show that  $F^A \equiv (P \Rightarrow A) + (Q \Rightarrow A)$  is not a semimonad (cannot define `flatMap`) when  $P$  and  $Q$  are arbitrary, different types.
- 7 Implement the `flatten` and `pure` methods for  $D^A \equiv 1 + A \times A$  (`type D[A] = Option[(A, A)]`) in at least two significantly different ways, and show that the monad laws always fail to hold. ( $D^A$  is not a monad!)

## Exercises II (continued)

- 8 Implement `flatten` and `pure` for  $F^A \equiv A + (R \Rightarrow A)$ , where  $R$  is a fixed type, and show that all the monad laws hold.
- 9 Check the identity laws for monad construction 6,  $F^A \equiv A + G^A$ , when  $\text{pure}_F$  is defined as  $\text{id}^{A \Rightarrow A} + 0$  (given that  $G^A$  is a monad). Show that the identity laws fail if  $\text{pure}_F$  were defined as  $0 + \text{pure}_G$ .
- 10 Implement the monad methods for  $F^A \equiv (Z \Rightarrow 1 + A) \times \text{List}^A$  using the known monad constructions (no need to check the laws).
- 11 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  $\text{ftn} = \text{ex}^{[F^A]}$  or  $\text{ftn} = \text{fmap ex}$ ?
- 12 A programmer implemented the `fmap` method for  $F^A \equiv A \times (A \Rightarrow Z)$  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.