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:

Manipulating items in container is preceded by a generator:

• After $x \leftarrow cont$, further computations will use all those x

```
\begin{array}{lll} x \leftarrow \text{cont} & & \text{y} \leftarrow \text{for } \{ \text{ x} \leftarrow \text{cont} \\ \text{y} \leftarrow \text{p(x)} & & \text{yy} \leftarrow \text{p(x)} \; \} \; \text{yield yy} \\ \text{z} \leftarrow \text{cont2(y)} & & \text{z} \leftarrow \text{cont2(y)} \end{array}
```

```
\texttt{cont.flatMap}(\texttt{x} \Rightarrow \texttt{p(x).flatMap}(\texttt{cont2})) = \texttt{cont.flatMap}(\texttt{p}).\texttt{flatMap}(\texttt{cont2})
```

Semimonad laws II: The laws for flatMap

To get a more concise notation, use flm instead of flatMap A semimonad S^A has $flm^{[S,A,B]}: (A \Rightarrow S^B) \Rightarrow S^A \Rightarrow S^B$ with 3 laws:

$$\begin{array}{c|c}
\operatorname{fimp} f^{A \Rightarrow B} & S^{B} & \operatorname{film} g^{B \Rightarrow S^{C}} \\
S^{A} & & \Longrightarrow S^{C} \\
& \operatorname{film} (f^{A \Rightarrow B} \circ g^{B \Rightarrow S^{C}})
\end{array}$$

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

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

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

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

$$S^{A} \xrightarrow{\text{flm } f^{A \Rightarrow S^{B}}} S^{B} \xrightarrow{\text{flm } g^{B \Rightarrow S^{C}}} S^{C}$$

Is there a shorter formulation of the laws?

Semimonad laws III: The laws for flatten

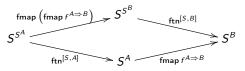
The methods flatten (denoted by ftn) and flatMap are equivalent:

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

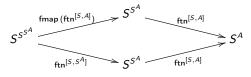
$$\mathsf{flm}\left(f^{A \Rightarrow S^B}\right) = \mathsf{fmap}\, f \circ \mathsf{ftn}$$

$$S^A \xrightarrow{\mathsf{flm}\left(f^{A \Rightarrow S^B}\right)} S^B$$

It turns out that flatten has only 2 laws:



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



Equivalence of a natural transformation and a "lifting"

- Equivalence of flm and ftn: ftn = flm (id); flm $f = \text{fmap } f \circ \text{ftn}$
- 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 $f = \operatorname{fmap} f \circ \operatorname{tr}$
- It follows that tr = ftr(id), and we have equivalence between tr and ftr:

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

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

$$f^A \xrightarrow{\operatorname{ftr}(f^{A \Rightarrow G^B})} F^B$$

- An automatic law for ftr ("naturality in A") follows from the definition: 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 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 o tr with that tr: fmap f o ftr (id) = ftr (f o id) = ftr f

Semimonad laws IV: Deriving the laws for flatten

Denote for brevity $q^{\uparrow} \equiv \operatorname{fmap}^{[S]} q$ for any function qExpress flm $f = f^{\uparrow} \circ$ ftn and substitute that into flm's 3 laws:

- flm $(f \circ g) = f^{\uparrow} \circ \text{flm } g \text{ gives } (f \circ g)^{\uparrow} \circ \text{ftn} = f^{\uparrow} \circ g^{\uparrow} \circ \text{ftn}$ - this law holds automatically due to functor composition law
- ② flm $(f \circ g^{\uparrow}) = \text{flm } f \circ g^{\uparrow} \text{ gives } (f \circ h)^{\uparrow} \circ \text{ftn} = f^{\uparrow} \circ \text{ftn} \circ h;$ using the functor composition law, we reduce this to $h^{\uparrow} \circ \text{ftn} = \text{ftn} \circ h - \text{this is the naturality law}$
- § $flm(f \circ flm g) = flm f \circ flm g$ with functor composition law gives $f^{\uparrow} \circ g^{\uparrow\uparrow} \circ \operatorname{ftn}^{\uparrow} \circ \operatorname{ftn} = f^{\uparrow} \circ \operatorname{ftn} \circ g^{\uparrow} \circ \operatorname{ftn}$; using ftn's naturality and omitting the common factor $f^{\uparrow} \circ g^{\uparrow\uparrow}$, we get $ftn^{\uparrow} \circ ftn = ftn \circ ftn - associativity law$
 - 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 : List List $\Rightarrow \text{List}^A$
 - ▶ 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 : $((((A \Rightarrow R) \Rightarrow R) \Rightarrow R) \Rightarrow (A \Rightarrow R) \Rightarrow R$
- 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:
 - $F^A \equiv A \times W \times W; \text{ ftn} ((a \times v_1 \times v_2) \times w_1 \times w_2) = a \times w_2 \times w_1$
 - $ightharpoonup F^A \equiv \operatorname{List}^A$, 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

pure :
$$A \Rightarrow M^A$$

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

```
\begin{array}{lll} & \text{$y$ = x$} \\ & \text{$z$ \leftarrow $cont(y)$} & \text{$z$ \leftarrow $cont(y)$} \\ & \text{pure(x).flatMap($y$ <math>\Rightarrow $cont(y)$) = $cont(x)$} & \text{pure of } \text{flm } f = f - \text{left identity} \end{array}
```

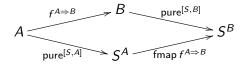
• Empty context is preceded by a generator:

```
x \leftarrow cont y \leftarrow pure(x) x \leftarrow cont y = x
```

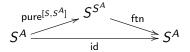
 $cont.flatMap(x \Rightarrow pure(x)) = cont$ flm(pure) = id - right identity

The monad laws formulated in terms of pure and flatten

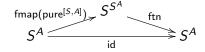
• Naturality law for pure: $f \circ pure = pure \circ f^{\uparrow}$



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



• Right identity: $flm(pure) = pure^{\uparrow} \circ ftn = id$



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_{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 id_{\diamond} of type $A \Rightarrow S^A$ as $id_{\diamond} \equiv pure$
- Composition law: $flm(f \diamond g) = flm f \circ flm g$ (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 ⋄ is used for composing Kleisli functions
- What are the properties of <?</p>
 - lacktriangle 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: pure $\diamond f = f$ and $f \diamond pure = f$
 - Associativity law: $(f \diamond g) \diamond h = f \diamond (g \diamond h)$
 - ▶ Follows from the flm law: $f \circ \text{flm}(g \circ \text{flm}h) = f \circ \text{flm} g \circ \text{flm} h$

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 <i>F</i>	$A \Rightarrow F^B$	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 $f^{A\Rightarrow S^B}\equiv \mathrm{id}^{S^A\Rightarrow S^A}\diamond f$
 - Define flatten through Kleisli: $ftn \equiv id^{S^{S^A} \Rightarrow S^{S^A}} \diamond id^{S^A \Rightarrow S^A}$
 - Express f_{map} through Kleisli: $f_{map} f \equiv (f_{map} id) \diamond (f \circ p_{ure})$
 - Need additional laws to connect ◊ and ○:
 - ▶ Left naturality: $f^{A\Rightarrow B} \circ g^{B\Rightarrow S^C} = (f \circ pure) \diamond g$
 - ▶ Right naturality: $f^{A\Rightarrow B} \circ \operatorname{fmap} g^{B\Rightarrow S^{C}} = f \diamond (g \circ \operatorname{pure})$
 - ★ 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:

- 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.
- **1** The function type $A \Rightarrow A$ is a monoid (for any type A)
 - ▶ The operation $f \circledast g$ is either $f \circ g$ or $g \circ f$
- ullet Any totally ordered type is a monoid, with \circledast defined as max or min
- **o** $S_1 \times S_2$ is a semigroup (monoid) if S_1 , S_2 are semigroups (monoids)
- **②** $S \times P$ 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 $a(s_1) \circ a(s_2) = a(s_1 \circledast s_2)$.
- - ullet 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:

- $F^A \equiv Z$ (constant functor) for a fixed type Z
 - For a full monad, need to choose Z=1
- $F^A \equiv A \times G^A$ for any functor G^A (a full monad only if $G^A \equiv 1$)
- - For a full monad, need W to be a monoid
- $F^A \equiv G^{Z+A\times W}$ 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
- \bullet $F^A \equiv A + G^A$ for any semimonad G^A
- § $F^A \equiv G^A + G^{F^A}$ (recursive) for any functor G^A (semimonad only!)
- $P^A \equiv R \rightarrow G^A$ for any (semi)monad G^A
- - ▶ For a full monad, need to set $G^A \equiv 1$

* Worked examples II: Constructions of filterable functors I

- (2) The fmapOpt laws hold for $F^A \times G^A$ if they hold for F^A and G^A
 - For $f^{A\Rightarrow 1+B}$, get fmapOpt_E $(f): F^A \Rightarrow F^B$ and fmapOpt_G $(f): G^A \Rightarrow G^B$
 - Define fmapOpt_{F\colored} $f \equiv p^{F^A} \times q^{G^A} \Rightarrow \text{fmapOpt}_F(f)(p) \times \text{fmapOpt}_G(f)(q)$
 - Identity law: $f = id_{\Diamond_{Opt}}$, so fmapOpt_F f = id and fmapOpt_G f = id
 - ▶ Hence we get fmapOpt_{F+G} $(f)(p \times q) = id(p) \times id(q) = p \times q$
 - Composition law:

```
(fmapOpt_{F\times G} f_1 \circ fmapOpt_{F\perp G} f_2)(p\times q)
= fmapOpt<sub>F×G</sub>(f_2) (fmapOpt<sub>F</sub>(f_1)(p) × fmapOpt<sub>G</sub>(f_1)(q))
= (fmapOpt_{\varepsilon} f_1 \circ fmapOpt_{\varepsilon} f_2)(p) \times (fmapOpt_{\varepsilon} f_1 \circ fmapOpt_{\varepsilon} f_2)(q)
= fmapOpt<sub>E</sub>(f_1 \diamond_{Opt} f_2)(p) \times fmapOpt_C(f_1 \diamond f_2)(q)
= fmapOpt<sub>F \ C</sub> (f_1 \diamond_{Opt} f_2)(p \times q)
```

- Exactly the same proof as that for functor property for $F^A \times G^A$
 - ▶ this is because fmapOpt corresponds to a generalized functor
- New proofs are necessary only when using non-filterable functors
 - ▶ these are used in constructions 4 6

* Worked examples II: Constructions of filterable functors II

- (5) The fmapOpt laws hold for $F^A \equiv 1 + A \times G^A$ if they hold for G^A
 - For $f^{A\Rightarrow 1+B}$, get fmapOpt_G $(f): G^A \Rightarrow G^B$
 - Define fmapOpt_F(f)(1 + $a^A \times q^{G^A}$) by returning 0 + $b \times$ fmapOpt_G(f)(q) if the argument is 0 + $a \times q$ and f(a) = 0 + b, and returning 1 + 0 otherwise
 - Identity law: $f = id_{\diamond_{Ont}}$, so f(a) = 0 + a and fmapOpt_Gf = id
 - ▶ Hence we get fmapOpt_F(id_{Opt}) $(1 + a \times q) = 1 + a \times q$
 - Composition law: need only to check for arguments $0 + a \times q$, and only when $f_1(a) = 0 + b$ and $f_2(b) = 0 + c$, in which case $(f_1 \diamond_{\mathsf{Opt}} f_2)(a) = 0 + c$; then

$$\begin{split} &(\mathsf{fmapOpt}_F \, f_1 \circ \mathsf{fmapOpt}_F \, f_2)(0 + a \times q) \\ &= \mathsf{fmapOpt}_F(f_2) \, (\mathsf{fmapOpt}_F(f_1)(0 + a \times q)) \\ &= \mathsf{fmapOpt}_F(f_2) \, (0 + b \times \mathsf{fmapOpt}_G(f_1)(q)) \\ &= 0 + c \times (\mathsf{fmapOpt}_G \, f_1 \circ \mathsf{fmapOpt}_G \, f_2)(q) \\ &= 0 + c \times \mathsf{fmapOpt}_G(f_1 \diamond_{\mathsf{Opt}} \, f_2)(q) \\ &= \mathsf{fmapOpt}_F(f_1 \diamond_{\mathsf{Opt}} \, f_2)(0 + a \times q) \end{split}$$

This is a "greedy filter": if f(a) is empty, will delete all data in G^A

* Worked examples II: Constructions of filterable functors III

- (6) The fmapOpt laws hold for $F^A \equiv G^A + A \times F^A$ if they hold for G^A
 - For $f^{A\Rightarrow 1+B}$, we have fmapOpt_G(f): $G^A \Rightarrow G^B$ and fmapOpt'_F(f): $F^A \Rightarrow F^B$ (for use in recursive arguments as the inductive assumption)
 - Define fmapOpt_F(f)($q^{G^A} + a^A \times p^{F^A}$) by returning $0 + \text{fmapOpt}'_F(f)(p)$ if f(a) = 1 + 0, and fmapOpt_G(f)(q) + $b \times \text{fmapOpt}'_F(f)(p)$ otherwise
 - Identity law: $id_{\diamond_{\mathbf{Opt}}}(x) \neq 1 + 0$, so $fmapOpt_F(id_{\diamond_{\mathbf{Opt}}})(q + a \times p) = q + a \times p$
 - Composition law:
 - $(\mathsf{fmapOpt}_F(f_1) \circ \mathsf{fmapOpt}_F(f_2))(q + a \times p) = \mathsf{fmapOpt}_F(f_1 \diamond_{\mathsf{Opt}} f_2)(q + a \times p)$
 - For arguments q+0, the laws for G^A hold; so assume arguments $0+a\times p$. When $f_1(a)=0+b$ and $f_2(b)=0+c$, the proof of the previous example will go through. So we need to consider the two cases $f_1(a)=1+0$ and $f_1(a)=0+b$, $f_2(b)=1+0$
 - If $f_1(a) = 1 + 0$ then $(f_1 \diamond_{\mathsf{Opt}} f_2)(a) = 1 + 0$; to show $\mathsf{fmapOpt}_F'(f_2)(\mathsf{fmapOpt}_F'(f_1)(p)) = \mathsf{fmapOpt}_F'(f_1 \diamond_{\mathsf{Opt}} f_2)(p)$, use the inductive assumption about $\mathsf{fmapOpt}_F'$ on p
 - If $f_1(a) = 0 + b$ and $f_2(b) = 1 + 0$ then $(f_1 \diamond_{\mathsf{Opt}} f_2)(a) = 1 + 0$; to show $\mathsf{fmapOpt}_F(f_2)(0 + b \times \mathsf{fmapOpt}_F'(f_1)(p)) = \mathsf{fmapOpt}_F'(f_1 \diamond_{\mathsf{Opt}} f_2)(p)$, rewrite $\mathsf{fmapOpt}_F(f_2)(0 + b \times \mathsf{fmapOpt}_F'(f_1)(p)) = \mathsf{fmapOpt}_F'(f_2)(\mathsf{fmapOpt}_F'(f_1)(p))$ and again use the inductive assumption about $\mathsf{fmapOpt}_F'$ on p

This is a "list-like filter": if f(a) is empty, will recurse into nested F^A data

Worked examples II: Constructions of filterable functors IV

Use known filterable constructions to show that

$$F^A \equiv (Int \times String) \Rightarrow (1 + Int \times A + A \times (1 + A) + (Int \Rightarrow 1 + A + A \times A \times String))$$
 is a filterable functor

- Instead of implementing Filterable and verifying laws by hand, we analyze the structure of this data type and use known constructions
- Define some auxiliary functors that are parts of the structure of F^A ,
 - $ightharpoonup R_1^A = (Int \times String) \Rightarrow A \text{ and } R_2^A = Int \Rightarrow A$
 - $G^A = 1 + \text{Int} \times A + A \times (1 + A)$ and $H^A = 1 + A + A \times A \times \text{String}$
- Now we can rewrite $F^A = R_1 [G^A + R_2 [H^A]]$
 - \triangleright G^A is filterable by construction 5 because it is of the form $G^A = 1 + A \times K^A$ with filterable functor $K^A = 1 + Int + A$
 - \triangleright K^A is of the form 1+A+X with constant type X, so it is filterable by constructions 1 and 3 with the Option functor 1 + A
 - ▶ H^A is filterable by construction 5 with $H^A = 1 + A \times (1 + A \times \text{String})$, while $1 + A \times String$ is filterable by constructions 5 and 1
- Constructions 3 and 4 show that $R_1 \left[G^A + R_2 \left[H^A \right] \right]$ is filterable Note that there are more than one way of implementing Filterable here

Exercises II

- 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: $F^A \equiv W \times (R \Rightarrow A)$, $G^A = R \Rightarrow (W \times A)$?
- Suppose a functor F^A has a natural transformation $ex^{[A]}: F^A \Rightarrow A$ that "extracts the value" from F^A . Would F^A be a semimonad if we defined flatten as $ftn = ex^{[F^A]}$ or $ftn = fmap \, ex$?
- **3** 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\RightarrowB): ((A, A\RightarrowZ)) \Rightarrow (B, B\RightarrowZ) = { case(a, az) \Rightarrow (f(a), (_: B) \Rightarrow az(a)) }
```

Show that this implementation fails to satisfy the functor laws.

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

Exercises II

(continued from the previous slide)

- 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.
- **1** If W and R are arbitrary fixed types, which of the functors can be made into a semimonad: $F^A \equiv W \times (R \Rightarrow A)$, $G^A = R \Rightarrow (W \times A)$?
- Suppose a functor F^A has a natural transformation $ex^{[A]}: F^A \Rightarrow A$ that "extracts the value" from F^A . Would it be correct to define F^A as a semimonad if we defined flatten as $ftn = ex^{[F^A]}$ or $ftn = fmap \, ex$?
- **Q** 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\RightarrowB): ((A, A\RightarrowZ)) \Rightarrow (B, B\RightarrowZ) = { case(a, az) \Rightarrow (f(a), (_: B) \Rightarrow a2z(a)) }
```

Show that this implementation fails to satisfy the functor laws.

① A programmer implemented the flatMap method for the type constructor $F^A \equiv 1 + A \times A$ as

```
def flatMap[A,B](f: A \Rightarrow F[B]): ((A,A\RightarrowZ)) \Rightarrow (B,B\RightarrowZ) = { case (a, az) \Rightarrow (f(a), (_: B) \Rightarrow az(a)) }
```