Chapter 7: Computations lifted to a functor context II Part 1: Examples of monads and semimonads

Sergei Winitzki

Academy by the Bay

2018-02-10

Computations within a functor context: Semimonads

Intuitions behind adding more "generator arrows"

Example:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} f(i, j, k)$$

Using Scala's for/yield syntax ("functor block")

- map replaces the last left arrow, flatMap replaces other left arrows
 - ▶ When the functor is *also* filterable, we can use "if" as well
- Standard library defines flatMap() as equivalent of map() o flatten
 - (1 to n).flatMap(j \Rightarrow ...) is (1 to n).map(j \Rightarrow ...).flatten
- flatten: $F[F[A]] \Rightarrow F[A]$ can be expressed through flatMap as well:
 - ▶ (xss: Seq[Seq[A]]).flatten = xss.flatMap { (xs: Seq[A]) \Rightarrow xs }
- Functors having flatMap/flatten are "flattenable" or semimonads
 - Most of them also have method pure: A ⇒ F[A] and so are monads

What is flatMap doing with the data in a collection?

Consider this schematic code using Seq as the container type:

Computations are repeated for all i, for all j, etc., from each collection

- All collections must have the same container type
 - ► Each *generator line* finally computes a container of the same type
 - ▶ The total number of resulting data items is $\leq m * n * p$
 - ▶ All the resulting data items must fit within *the same* container type!
 - ▶ The set of *container capacity counts* must be closed under multiplication
- What container types have this property?
 - ► Seq, NonEmptyList can hold any number of elements ≥ min. count
 - ▶ Option, Either, Try, Future can hold 0 or 1 elements ("pass/fail")
 - ▶ "Tree-like" containers, e.g. can hold only 3, 6, 9, 12, ... elements
 - "Non-standard" containers: $F^A \equiv \text{String} \Rightarrow A$; $F^A \equiv (A \Rightarrow \text{Int}) \Rightarrow \text{Int}$

Working with list-like monads

Seq, NonEmptyList, Iterator, Stream

Typical tasks for "list-like" monads:

- Create a list of all combinations or all permutations of a sequence
- Traverse a "solution tree" with DFS and filter out incorrect solutions
 - ► Can use eager (Seq) or lazy (Iterator, Stream) evaluation strategies
 - Usually, list-like containers have many additional methods
 - ★ append, prepend, concat, fill, fold, scan, etc.

- All permutations of Seq("a", "b", "c")
- 2 All subsets of Set("a", "b", "c")
- 3 All subsequences of length 3 out of the sequence (1 to m)
- 4 All solutions of the "8 queens" problem
- **5** Generalize examples 1-3 to support arbitrary length n instead of 3
- Generalize example 4 to solve *n*-queens problem
- Transform Boolean formulas between CNF and DNF.

Intuitions for pass/fail monads

Option, Either, Try, Future

- Container F^A can hold n = 1 or n = 0 values of type A
- Such containers will have methods to create "pass" and "fail" values

Schematic example of a functor block program using the \mathtt{Try} functor:

```
val result: Try[A] = for { // computations in the Try functor
  x ← Try(...) // first computation; may fail
  y = f(x) // no possibility of failure in this line
  if p(y) // the entire expression will fail if this is false
  z ← Try(g(x, y)) // may fail here
  r ← Try(...) // may fail here as well
} yield r // r is of type A, so result is of type Try[A]
```

- Computations may yield a result (n = 1), or may fail (n = 0)
- The functor block chains several such computations sequentially
 - Computations are sequential even if using the Future functor!
 - ▶ Once any computation fails, the entire functor block fails (0 * n = 0)
 - Only if all computations succeed, the functor block returns one value
 - Filtering can also make the entire expression fail
- "Flat" functor block replaces a chain of nested if/else or match/case

Working with pass/fail monads

Typical tasks for pass/fail monads:

- Perform a linear sequence of computations that may fail
- Avoid crashing on failure, instead return an error value

- Read values of Java properties, checking that they all exist
- Obtain values from Future computations in sequence
- Make arithmetic safe by returning error messages in Either
- Fail less: allow up to 2 computations out of n to throw an exception
- **5** Generalize example 3 to support up to k failures instead of 2

Working with tree-like monads

Typical tasks for tree-like monads:

- Traverse a syntax tree, substitute subexpressions
- ???

- Implement variable substitution for a simple arithmetic language
- ???

Single-value monads (non-standard containers)

Reader, Writer, Eval, Cont, State

- Container holds exactly 1 value, together with a "context"
- Usually, methods exist to insert a value and to work with the "context"

Typical tasks for single-value monads:

- Collecting extra information about computations along the way
- Chaining computations with a nonstandard evaluation strategy

- Dependency injection with the Reader monad
- Perform computations and log information about each step
- 3 Perform lazy or memoized computations in a sequence
- 4 A chain of asynchronous operations
- A sequence of steps that update state while returning results

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 cont1
                                                                v \leftarrow cont1
      y = f(x)
                                                                         .map(x \Rightarrow f(x))
                                                                z \leftarrow cont2(y)
      z \leftarrow cont2(y)
cont1.flatMap(x \Rightarrow cont2(f(x))) = cont1.map(f).flatMap(y \Rightarrow cont2(y))
```

Manipulating items in container is preceded by a generator:

```
x \leftarrow cont1
                                                       x \leftarrow cont1
      y \leftarrow cont2(x)
                                                       z \leftarrow cont2(x)
      z = f(v)
                                                                  .map(f)
cont1.flatMap(cont2).map(f) = cont1.flatMap(x \Rightarrow cont2(x).map(f))
```

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

```
x \leftarrow cont
                                                              y \leftarrow for \{ x \leftarrow cont \}
y \leftarrow p(x)
                                                                                 yy \leftarrow p(x) } yield yy
z \leftarrow cont2(y)
                                                              z \leftarrow cont2(v)
```

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

Semimonad laws II: The laws for flatMap

To use the concise notation, denote flatMap by flm A semimonad S^A has $flm^{[S,A,B]}: (A \Rightarrow S^B) \Rightarrow S^A \Rightarrow S^B$ with 3 laws:

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

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

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

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

Is there a shorter formulation of the laws?

Semimonad laws III: The laws for flatten

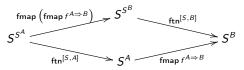
The methods flatten (denoted by ftn) and flatMap are equivalent:

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

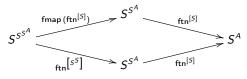
$$\operatorname{flm}(f^{A \Rightarrow S^B}) = \operatorname{fmap} f \circ \operatorname{ftn}$$

$$S^A \xrightarrow{\operatorname{flm}(f^{A \Rightarrow S^B})} S^{S^B}$$

It turns out that flatten has only 2 laws:



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



Semimonad laws III: Deriving the laws for flatten

Denote for brevity $q_{\uparrow} \equiv \operatorname{fmap}^{[S]} q$ for any function qExpress flm $f = f_{\uparrow} \circ \text{ftn}$ 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 $\mathsf{flm}\,(f\circ g_\uparrow)=\mathsf{flm}\,f\circ g_\uparrow \;\mathsf{gives}\;(f\circ h)_\uparrow\circ\mathsf{ftn}=f_\uparrow\circ\mathsf{ftn}\circ h;$ using the functor composition law, we reduce this to $h_{\uparrow} \circ \text{ftn} = \text{ftn} \circ h$ - this is the naturality law for flatten
- § $flm(f \circ flm g) = flm f \circ flm g$ 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 ftn's naturality and omitting the common factor $f_{\uparrow} \circ g_{\uparrow\uparrow}$, we get ftn's associativity: $ftn_{\uparrow} \circ ftn = ftn \circ ftn$
 - flatten has the simplest type signature and the fewest laws
 - It is usually easy to check naturality!
 - ▶ Parametricity theorem: Any fully parametric code for a function of type $F^A \Rightarrow G^A$ implements a natural transformation $F \rightsquigarrow G$
 - Checking flatten's associativity needs more work

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

Semimonad laws IV: Checking the laws of flatten

- 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

Exercises I

- Confucius gave wisdom on each of the 7 days of a week. Sometimes the wise proverbs were hard to remember. If Confucius forgets what he said on a given day, he also forgets what he said on all the previous days of the week. Is this setup described by a filterable functor?
- Define evenFilter(p) on an IndexedSeg[T] such that a value x: T is retained if p(x)=true and only if the sequence has an even number of elements y for which p(y)=false. Does this define a filterable functor?

Implement filter for these functors if possible (law checking optional):

- 3 $F^A \equiv Int + String \times A \times A \times A$
- final case class Q[A, Z](id: Long, user1: Option[(A, Z)], user2: Option[(A, Z)]) – with respect to the type parameter A
- **5** $F^A = \text{MyTree}^A$ defined recursively as $F^A \equiv 1 + A \times F^A \times F^A$
- final case class R[A](x: Int, y: Int, z: A, data: List[A]), where the standard functor List already has withFilter defined
- Show that $C^A \equiv A + A \times A \Rightarrow 1 + Z$ is a filterable contrafunctor

Filterable functors: The laws in depth I

Is there a shorter formulation of the laws that is easier to remember?

- Intuition: When p(x) = false, replace x: A by 1: Unit in F[A]
 - ▶ (1) How to replace x by 1 in F[A] without breaking the types?
 - ▶ (2) How to transform the resulting type back to F[A]?
- We could do (1) if instead of F^A we had F^{1+A} i.e. F[Option[A]]
 - ▶ Now use filter to replace A by 1 in each item of type 1 + A
 - ▶ Get F^{1+A} from F^A using inflate : $F^A \Rightarrow F^{1+A} = \text{fmap} (\text{Some}^{A \Rightarrow 1+A})$
 - ► Filter $F^{1+A} \Rightarrow F^{1+A}$ using fmap $(x^{1+A} \Rightarrow \text{filter}_{Opt}(p^{A \Rightarrow Boolean})(x))$

filter
$$p: F^A \xrightarrow{\text{inflate}} F^{1+A} \xrightarrow{\text{fmap(filter}_{Opt}p)} F^{1+A} \xrightarrow{\text{deflate}} F^A$$

- Doing (2) means defining a function deflate: F[Option[A]] ⇒ F[A] ▶ standard library already has flatten[T]: Seq[Option[T]] ⇒ Seq[T]
- Simplify fmap(Some^{$A\Rightarrow 1+A$}) \circ fmap (filter_{Opt}p) = fmap (bop (p)) where we
- defined bop(p): $(A \Rightarrow 1 + A) \equiv x \Rightarrow Some(x)$.filter(p)
- In this way, express filter through deflate (see example code)
 - filter $p = \text{fmap}(\text{bop } p) \circ \text{deflate.} \text{Notation: bop } p \text{ is bop } (p)$, like $\cos x$ filter $p: F^A \xrightarrow{\text{fmap}(\text{bop } p)} F^{1+A} \xrightarrow{\text{deflate}} F^A$

Filterable functors: Using deflate

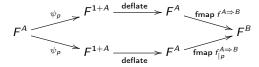
- So far we have expressed filter through deflate
- We can also express deflate through filter (assuming law 4 holds):

deflate:
$$F^{1+A} \xrightarrow{\text{filter(.nonEmpty)}} F^{1+A} \xrightarrow{\text{fmap(.get)}} F^A$$
def deflate[F[_],A](foa: F[Option[A]]): F[A] =
foa.filter(_.nonEmpty).map(_.get) // _.get is $0 + x^A \Rightarrow x^A$
// for F = Seq, this would be foa.collect { case Some(x) \Rightarrow x }
// for arbitrary functor F we need to use the partial function, _.get

- This means deflate and filter are computationally equivalent
 - ► We could specify filterable functors by implementing deflate
 - ★ The implementation of filter would then be derived by library
- Use deflate to verify that some functors are certainly not filterable:
 - $F^A = A + A \times A$. Write $F^{1+A} = 1 + A + (1+A) \times (1+A)$
 - **★** cannot map $F^{1+A} \Rightarrow F^A$ because we do not have $1 \to A$
 - ► $F^A = \text{Int} \Rightarrow A$. Write $F^{1+A} = \text{Int} \Rightarrow 1 + A$
 - * type signature of deflate would be (Int $\Rightarrow 1 + A$) \Rightarrow Int $\Rightarrow A$
 - **★** cannot map $F^{1+A} \Rightarrow F^A$ because we do not have $1 + A \rightarrow A$
- deflate is easier to implement and to reason about

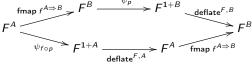
* Filterable functors: The laws in depth II

- We were able to define deflate only by assuming that law 4 holds
- Now, law 4 is satisfied automatically if filter is defined via deflate!
 - ▶ Denote $\psi_p^{F^A \Rightarrow F^{1+A}} \equiv \text{fmap (bop } p)$ for brevity, then filter $p = \psi_p \circ \text{deflate}$
 - ▶ Law 4 then becomes: $\psi_p \circ \text{deflate} \circ \text{fmap } f^{A \Rightarrow B} = \psi_p \circ \text{deflate} \circ \text{fmap } f_{|p|}$



- We would like to interchange deflate and fmap in both sides
 - ▶ We need a *naturality* law; let's express law 1 through deflate:

fmap
$$f^{A\Rightarrow B}\circ\psi_{p}\circ \mathsf{deflate}^{F,B}=\psi_{f\circ p}\circ \mathsf{deflate}^{F,A}\circ \mathsf{fmap}\ f^{A\Rightarrow B}$$



Can we simplify fmap $f \circ \psi_p = \text{fmap } f \circ \text{fmap (bop } p) = \text{fmap } (f \circ \text{bop } p)$?

* Filterable functors: The laws in depth III

• Have property: $f^{A\Rightarrow B} \circ \text{bop}(p^{B\Rightarrow \text{Boolean}}) = \text{bop}(f \circ p) \circ \text{fmap}^{\text{Opt}} f$ (see code)

$$A \xrightarrow{f^{A \Rightarrow B}} B \xrightarrow{\text{bop } p} 1 + B$$

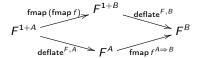
$$1 + A \xrightarrow{\text{fmap}^{Opt}_f} B$$

We can now rewrite Law 1 as

 $fmap(bop(f \circ p)) \circ fmap(fmap^{Opt}f) \circ deflate = fmap(bop(f \circ p)) \circ deflate \circ fmapf$

Remove common prefix fmap $(bop (f \circ p)) \circ ...$ from both sides:

 $fmap(fmap^{Opt}f^{A\Rightarrow B}) \circ deflate^{F,B} = deflate^{F,A} \circ fmap f^{A\Rightarrow B} - law 1 for deflate$

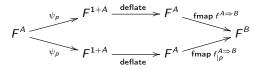


- deflate: $F^{1+A} \Rightarrow F^A$ is a natural transformation (has naturality law)
 - Example: $F^A = 1 + A \times A$
 - $F^{1+A} = 1 + (1+A) \times (1+A) = 1 + 1 \times 1 + A \times 1 + 1 \times A + A \times A$
- natural transformations map containers $G^A \Rightarrow H^A$ by rearranging data in them

* Filterable functors: The laws in depth IV

• The naturality law for deflate:

$$\mathsf{fmap}\,(\mathsf{fmap}^{\mathsf{Opt}}f^{A\Rightarrow B})\circ\mathsf{deflate}^{F,B}=\mathsf{deflate}^{F,A}\circ\mathsf{fmap}\,f^{A\Rightarrow B}$$
 Law 4 expressed via $\mathsf{deflate}$:



$$\psi_{P} \circ \mathsf{deflate}^{F,A} \circ \mathsf{fmap} \ f^{A \Rightarrow B} = \psi_{P} \circ \mathsf{deflate}^{F,A} \circ \mathsf{fmap} \ f_{|P}$$

Use naturality to interchange deflate and fmap in both sides of law 4:

```
\psi_p \circ \mathsf{fmap}(\mathsf{fmap}^{\mathsf{Opt}} f) \circ \mathsf{deflate}^{F,B} = \psi_p \circ \mathsf{fmap}(\mathsf{fmap}^{\mathsf{Opt}} f_{|p}) \circ \mathsf{deflate}^{F,B}
                           [omit deflate<sup>F,B</sup> from both sides; expand \psi_p]
                                    bop p \circ \operatorname{fmap}^{\operatorname{Opt}} f = \operatorname{bop} p \circ \operatorname{fmap}^{\operatorname{Opt}} f_{|p|} - check this by hand:
```

```
x \Rightarrow Some(x).filter(p).map(f)
x \Rightarrow Some(x).filter(p).map { x if p(x) <math>\Rightarrow f(x) }
```

• These functions are equivalent because law 4 holds for Option

Filterable functors: The laws in depth V

Maybe $\psi_p \circ \text{deflate}$ is easier to handle than deflate? Let us define

$$\begin{array}{c} \mathsf{fmapOpt}^{F,A,B}(f^{A\Rightarrow 1+B}): F^A \Rightarrow F^B = \mathsf{fmap}\ f \circ \mathsf{deflate}^{F,B} \\ \\ f^{\mathsf{fmap}\ f^{A\Rightarrow 1+B}} F^{1+B} & \overset{\mathsf{deflate}^{F,B}}{\longrightarrow} F^B \end{array}$$

- fmapOpt and deflate are equivalent: deflate $^{F,A} = \text{fmapOpt}^{F,1+A,A}(\text{id}^{1+A\Rightarrow 1+A})$
- Express laws 1 3 in terms of fmapOpt: do they get simpler?
 - ► Express filter through fmapOpt: filter $p = \text{fmapOpt}^{F,A,A}$ (bop p)
 - ▶ Consider the expression needed for law 2: $x \Rightarrow p_1(x) \land p_2(x)$
 - ▶ bop $(x \Rightarrow p_1(x) \land p_2(x)) = x^A \Rightarrow (bop p_1)(x)$.flatMap $(bop p_2)$ see code
 - ★ Denote this computation by ⋄_{Opt} and write

$$q_1^{A\Rightarrow 1+B}\diamond_{\mathsf{Opt}}q_2^{B\Rightarrow 1+C}\equiv x^A\Rightarrow q_1(x).\mathsf{flatMap}\left(q_2
ight)$$

- ▶ Similar to composition of functions, except the types are $A \Rightarrow 1 + B$
 - ★ This is a particular case of **Kleisli composition**; the general case: $\diamond_M: (A \Rightarrow M^B) \Rightarrow (B \Rightarrow M^C) \Rightarrow (A \Rightarrow M^C)$; we set $M^A \equiv 1 + A$
 - **★** The **Kleisli identity** function: $id_{\diamond_{\mathbf{Ont}}}^{A\Rightarrow 1+A} \equiv x^{A} \Rightarrow \mathsf{Some}(x)$
 - ★ Kleisli composition ⋄_{Opt} is associative and respects the Kleisli identity!
 - * fmapOpt lifts a Kleisliopt function $f^{A\Rightarrow 1+B}$ into the functor F

Filterable functors: The laws in depth VI

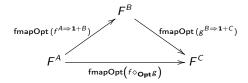
Simplifying down to two laws

- Only two laws are necessary for fmapOpt!
- Identity law (covers old law 3):

$$\mathsf{fmapOpt}\left(\mathsf{id}_{\diamond_{\mathbf{Opt}}}^{A\Rightarrow 1+A}\right) = \mathsf{id}^{F^A\Rightarrow F^A}$$

Composition law (covers old laws 1 and 2):

$$\mathsf{fmapOpt}\,(f^{A\Rightarrow 1+B}) \circ \mathsf{fmapOpt}\,(g^{B\Rightarrow 1+\mathcal{C}}) = \mathsf{fmapOpt}\,(f \diamond_{\mathsf{Opt}} g)$$



- The two laws for fmapOpt are very similar to the two functor laws
 - ▶ Both of them use more complicated types than the old laws
 - Conceptually, the new laws are simpler (lift $f^{A\Rightarrow 1+B}$ into $F^A\Rightarrow F^B$)

* Filterable functors: The laws in depth VII

Showing that old laws 1-3 follow from the identity and composition laws for fmapOpt

• Old law 3 is *equivalent* to the identity law for fmapOpt:

$$\mathsf{filter}\,(x^A\Rightarrow\mathsf{true})=\mathsf{fmap}\,(x^A\Rightarrow\mathsf{0}+x)\circ\mathsf{deflate}=\mathsf{fmapOpt}\,(\mathsf{id}_{\diamond_{\mathbf{Opt}}})=\mathsf{id}^{F^A\Rightarrow F^A}$$

- Derive old law 2: need to work with $q_{1,2} \equiv bop(p_{1,2}) : A \Rightarrow 1 + A$
 - ▶ The Boolean conjunction $x \Rightarrow p_1(x) \land p_2(x)$ corresponds to $q_1 \diamond_{\mathsf{Opt}} q_2$
 - ▶ Apply the composition law to Kleisli functions of types $A \Rightarrow 1 + A$:

$$\begin{aligned} & \text{filter } p_1 \circ \text{filter } p_2 = \text{fmapOpt } q_1 \circ \text{fmapOpt } q_2 \\ &= \text{fmapOpt } (q_1 \diamond_{\mathsf{Opt}} q_2) = \text{fmapOpt } (\mathsf{bop} \, (x \Rightarrow p_1(x) \land p_2(x))) \end{aligned}$$

- Derive old law 1:
 - ▶ express filter through fmapOpt; old law 1 becomes fmap $f \circ \text{fmapOpt} (\text{bop } p) = \text{fmapOpt} (\text{bop} (f \circ p)) \circ \text{fmap } f \text{eq. (*)}$
 - ▶ lift $f^{A\Rightarrow B}$ to Kleisli_{Opt} by defining $k_f^{A\Rightarrow 1+B} = f \circ \mathrm{id}_{\diamond_{\mathrm{Opt}}}$; then we have fmapOpt (k_f) = fmap $k_f \circ \mathrm{deflate} = \mathrm{fmap} f \circ \mathrm{fmap} \, \mathrm{id}_{\diamond_{\mathrm{Opt}}} \circ \mathrm{deflate} = \mathrm{fmap} f$
 - rewrite eq. (*) as fmapOpt $(k_f \diamond_{\mathsf{Opt}} \mathsf{bop}\, p) = \mathsf{fmapOpt}\, (\mathsf{bop}\, (f \circ p) \diamond_{\mathsf{Opt}} k_f)$
 - ▶ it remains to show that $k_f \diamond_{\mathsf{Opt}} \mathsf{bop} \, p = \mathsf{bop} \, (f \circ p) \diamond_{\mathsf{Opt}} k_f$
 - ▶ use the properties $k_f \diamond_{\mathsf{Opt}} q = f \circ q$ and $q \diamond_{\mathsf{Opt}} k_f = q \circ \mathsf{fmap}^{\mathsf{Opt}} f$, and $f \circ \mathsf{bop} p = \mathsf{bop} (f \circ p) \circ \mathsf{fmap}^{\mathsf{Opt}} f$ (property from slide 11)

Summary: The methods and the laws

Filterable functors can be defined via filter, deflate, or fmapOpt

- All three methods are equivalent but have different roles:
 - ► The easiest to use in program code is filter / withFilter
 - ► The easiest type signature to implement and reason about is deflate
 - Conceptually, the laws are easiest to remember with fmapOpt
- * The 2 laws for fmapOpt are the 2 functor laws with a Kleisli "twist"
- * Category theory accommodates this via a generalized definition of functors as liftings between "twisted" types. Compare:
 - fmap : $(A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$ ordinary container ("endofunctor")
 - ▶ contrafmap : $(B \Rightarrow A) \Rightarrow F^A \Rightarrow F^B$ lifting from reversed functions
 - ▶ fmapOpt : $(A \Rightarrow 1 + B) \Rightarrow F^A \Rightarrow F^B$ lifting from Kleisli_{Opt}-functions
- CT gives us some *intuitions* about how to derive better laws:
 - look for type signatures that resemble a generalized sort of "lifting"
 - look for natural transformations and use the naturality law
- However, CT does not directly provide any derivations for the laws
 - you will not find the laws for filter or deflate in any CT book
 - ▶ CT is abstract, only gives hints about possible further directions
 - ★ investigate functors having "liftings" with different type signatures
 - ★ replace Option in the Kleisli_{Opt} construction by another functor

Structure of filterable functors

How to recognize a filterable functor by its type?

Intuition from deflate: reshuffle data in F^A after replacing some A's by 1

- "reshuffling" usually means reusing different parts of a disjunction Some constructions of exponential-polynomial filterable functors
 - $F^A = Z$ (constant functor) for a fixed type Z (define fmapOpt f = id)
 - Note: $F^A = A$ (identity functor) is *not* filterable
 - ② $F^A \equiv G^A \times H^A$ for any filterable functors G^A and H^A

 - $F^A \equiv G^{H^A}$ for any functor G^A and filterable functor H^A
 - $F^A \equiv 1 + A \times G^A$ for a filterable functor G^A
 - Note: pointed types P are isomorphic to 1 + Z for some type Z
 - **★** Example of non-trivial pointed type: $A \Rightarrow A$
 - ***** Example of non-pointed type: $A \Rightarrow B$ when A is different from B
 - So $F^A \equiv P + A \times G^A$ where P is a pointed type and G^A is filterable
 - ▶ Also have $F^A \equiv P + A \times A \times ... \times A \times G^A$ similarly
 - **6** $F^A \equiv G^A + A \times F^A$ (recursive) for a filterable functor G^A
 - $F^A \equiv G^A \Rightarrow H^A$ if contrafunctor G^A and functor H^A both filterable
 - ▶ Note: the functor $F^A \equiv G^A \Rightarrow A$ is not filterable

* 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:

$$\begin{split} &(\mathsf{fmapOpt}_{F \times G} \, f_1 \circ \mathsf{fmapOpt}_{F + G} \, f_2)(p \times q) \\ &= \mathsf{fmapOpt}_{F \times G}(f_2) \, (\mathsf{fmapOpt}_F(f_1)(p) \times \mathsf{fmapOpt}_G(f_1)(q)) \\ &= (\mathsf{fmapOpt}_F \, f_1 \circ \mathsf{fmapOpt}_F \, f_2)(p) \times (\mathsf{fmapOpt}_G \, f_1 \circ \mathsf{fmapOpt}_G \, f_2) \, (q) \\ &= \mathsf{fmapOpt}_F(f_1 \diamond_{\mathsf{Opt}} \, f_2)(p) \times \mathsf{fmapOpt}_G(f_1 \diamond f_2)(q) \\ &= \mathsf{fmapOpt}_{F \times G}(f_1 \diamond_{\mathsf{Opt}} \, f_2)(p \times q) \end{split}$$

- 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 $K^A = 1 + \text{Int} + A$
 - \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

- Implement a Filterable instance for type F[T] = G[H[T]] assuming that the functor H[T] already has a Filterable instance (construction 4). Verify the laws rigorously (i.e. by calculations, not tests).
- ② For type F[T] = Option[Int ⇒ Option[(T, T)]], implement a Filterable instance. Show that the filterable laws hold by using known filterable constructions (avoiding explicit proofs or tests).
- Implement a Filterable instance for $F^A \equiv G^A + \operatorname{Int} \times A \times A \times F^A$ (recursive) for a filterable functor G^A . Verify the laws rigorously.
- **3** Show that $F^A = 1 + A \times G^A$ is in general *not* filterable if G^A is an arbitrary (non-filterable) functor; it is enough to give an example.
- Show that $F^A = 1 + G^A + H^A$ is filterable if $1 + G^A$ and $1 + H^A$ are filterable (even when G^A and H^A are by themselves not filterable).
- **6** Show that the functor $F^A = A + (Int \Rightarrow A)$ is not filterable.
- **②** Show that one can define deflate: $C^{1+A} \Rightarrow C^A$ for any contrafunctor C^A (not necessarily filterable), similarly to how one can define inflate: $F^A \Rightarrow F^{1+A}$ for any functor F^A (not necessarily filterable).