

Chapter 6: Computations lifted to a functor context I

Filterable functors, their laws and structure

Sergei Winitzki

Academy by the Bay

2018-02-03

Computations within a functor context

- Example:

$$\sum_{x \in \mathbb{Z}; 0 \leq x \leq 100; \cos x > 0} \sqrt{\cos x} \approx 38.71$$

Scala code:

```
(0 to 100).map(math.cos(_)).filter(_ > 0).map(math.sqrt).sum
```

- Using Scala's `for`/`yield` syntax (“functor block”, “`for` comprehension”)

<pre>(for { x ← 0 to 100 y = math.cos(x) if y > 0 } yield { math.sqrt(y) }).sum</pre>	<pre>(0 to 100).map { x ⇒ math.cos(x) }.filter { y ⇒ y > 0 }.map { y ⇒ math.sqrt(y) }.sum</pre>
---	--

- “Functor block” is a syntax for manipulating data within a container
 - ▶ Container must be a functor (has `map` such that the laws hold)
- A **filterable functor** is a functor that has a `withFilter` method
- Functor block works if have `withFilter(p: A⇒Boolean): F[A] ⇒ F[A]`
 - ▶ What are the required laws for `withFilter`?
 - ▶ What data types are filterable functors?

Filterable functors: Intuitions I

Intuition: the `filter` call *may decrease* the number of data items held

- a filterable container can hold *more or fewer* data items of type T

Examples:

- $\text{Option}[T] \equiv 1 + T$
 - ▶ `Some(123).filter(_ > 0)` returns `Some(123)`
 - ▶ `Some(123).filter(_ == 1)` returns `None`
 - ▶ `Some(123).withFilter(_ == 1).map(identity)` returns `None`
- $\text{List}[T] \equiv 1 + T + T \times T + T \times T \times T + \dots$
 - ▶ `List(10, 20, 30).filter(_ > 10)` returns `List(20, 30)`
 - ▶ `List(10, 20, 30).filter(_ == 1)` returns `List()`

What we learn from these examples:

- The data type must contain a *disjunction* having different counts of T
- When the predicate `p` returns `false` on some T values, the remaining data goes to a part of the disjunction that has fewer T values
- Values `x` are *algebraically* replaced by 1 (a `Unit`) when `p(x) = false`
- The container can become “empty” as a result of filtering

Examples of filterable functors I

- Consider these business requirements:
 - ▶ An order can be placed on Tuesday and/or on Friday
 - ▶ An order is approved if requested amount is less than \$1,000, etc.

```
final case class Orders[A](tue: Option[A], fri: Option[A]) {  
  def withFilter(p: A ⇒ Boolean): Orders[A] =  
    Orders(tue.filter(p), fri.filter(p))  
}  
Orders(Some(500), Some(2000)).withFilter(_ < 1000)  
// returns Orders(Some(500), None)
```

- The functor type is $F^A = (1 + A) \times (1 + A)$
 - ▶ When a value does not pass the filter, the A is replaced by 1
- Filtering is applied to both parts of the product type independently
- What if additional business requirements were given:
 - ▶ (a) both orders must be approved, or else no orders can be placed
or
 - ▶ (b) both orders can be placed if at least one of them is approved
- Does this still make sense as “filtering”?
 - ▶ Need mathematical laws to decide this

Filterable functors: Intuitions II

- Intuition: computations in the functor block should “make sense”
 - ▶ we should be able to reason correctly by looking at the program text
- A schematic example of a functor block program using `map` and `filter`:

```
for { // computations lifted to the List functor
  x ← List(...) // the first line has “←”, other lines do not
  y = f(x) // will become a “map(f)” after compilation
  if p1(y) // will become a “withFilter(p1)”
  if p2(y)
  z = g(x, y)
  if q(x, y, z)
} yield // for all x in list, such that conditions hold, compute this:
k(x, y, z)
```

- What we intuitively expect to be true about such programs:
 - ① `y = f(x); if p(y);` is equivalent to `if p(f(x)); y = f(x);`
 - ② `if p1(y); if p2(y);` is equivalent to `if p1(y) && p2(y)`
 - ③ When a filter predicate `p(x)` returns `true` for *all* `x`, we can delete the line “`if p(x)`” from the program with no change to the results
 - ④ When a filter predicate `p(x)` returns `false` for some `x` then *that* `x` will be excluded from computations performed after “`if p(x)`”

Examples of filterable functors I: Checking the laws

- Properties 1 – 4 are expressed as laws for `filter` $(p \Rightarrow \text{Boolean}) \Rightarrow F^A \Rightarrow F^A$:
 - ① $\text{fmap } f^{A \Rightarrow B} \circ \text{filter } p^{B \Rightarrow \text{Boolean}} = \text{filter } (f \circ p) \circ \text{fmap } f^{A \Rightarrow B}$
 - ② $\text{filter } p_1^{A \Rightarrow \text{Boolean}} \circ \text{filter } p_2^{A \Rightarrow \text{Boolean}} = \text{filter } (x \Rightarrow p_1(x) \wedge p_2(x))$
 - ③ $\text{filter } (x^A \Rightarrow \text{true}) = \text{id}^{F^A \Rightarrow F^A}$
 - ④ $\text{filter } p \circ \text{fmap } f^{A \Rightarrow B} = \text{filter } p \circ \text{fmap } (f|_p)$ where $f|_p$ is the *partial function* defined as `x => if (p(x)) f(x) else ???`
- Can define a type class `Filterable`, method `withFilter`
- Check the laws for Example I (see code)
 - ▶ “Orders” example with / without business rule (a) – laws hold
 - ▶ $F^A = 1 + A \times A$ (“stingy product”)
- Examples of functors that are *not* filterable:
 - ▶ “Orders” with additional business rule (b) – breaks law 2 for some $p_{1,2}$
 - ▶ F^A defining `filter` in a special way for $A = \text{Int}$ (breaks law 1)
 - ▶ $F^A = 1 + A$ defining $\text{filter } (p) (x) \equiv 1 + 0$ breaks law 3
 - ▶ $F^A \equiv A$ – must define $\text{filter } (p^{A \Rightarrow \text{Boolean}}) (x^A) = x$, breaking law 4
 - ▶ $F^A \equiv A \times (1 + A)$ – unable to remove the first A , breaking law 4

These mathematical laws specify *rigorously* what it means to “filter data”!

Worked examples I: Programming with filterables

- 1 John can have up to 3 coupons, and Jill up to 2. *All* of John's coupons must be valid on purchase day, while each of Jill's coupons is checked independently. Implement the filterable functor describing this setup.
- 2 A server receives a sequence of requests. Each request must be authenticated. Once a non-authenticated request is found, no further requests are accepted. Is this setup described by a filterable functor?

For each of these functors, determine whether they are filterable, and if so, implement `withFilter` via a type class:

- 3 `final case class P[T](first: Option[T], second: Option[(T, T)])`
- 4 $F^A = \text{Int} + \text{Int} \times A + \text{Int} \times A \times A + \text{Int} \times A \times A \times A$
- 5 $F^A = \text{NonEmptyList}^A$ defined recursively as $F^A = A + A \times F^A$
- 6 $F^{Z,A} = Z + \text{Int} \times Z \times A \times A$ (with respect to the type parameter A)
- 7 $F^{Z,A} = 1 + Z + \text{Int} \times Z \times A \times \text{List}^A$ (w.r.t. the type parameter A)
- 8 Show that $C^A = A \Rightarrow \text{Int}$ is a filterable *contrafunctor* (implement `withFilter` with the same type signature)

Exercises I

- 1 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?
- 2 Define `evenFilter(p)` on an `IndexedSeq[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 $F^A = \text{Int} + \text{String} \times A \times A \times A$
- 4 `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 $F^A = 1 + A \times F^A \times F^A$
- 6 `final case class R[A](x: Int, y: Int, z: A, data: List[A])`, where the standard functor `List` already has `withFilter` defined
- 7 Show that $C^A = (\text{Int} \Rightarrow A) \Rightarrow \text{Int}$ is a filterable contrafunctor

Filterable functors: The laws in depth I

Is there a more elegant formulation of the laws, easier to understand?

- Main intuition: When $p(x) = \text{false}$, replace $x: A$ by $1: \text{Unit}$ 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 the type $F[A]$ we had $F[\text{Option}[A]]$
 - ★ Map F^A to F^{1+A} using $\text{fmap}(\text{Some}^{A \Rightarrow 1+A}) : F^A \Rightarrow F^{1+A}$
 - ★ Now we can replace A by 1 in each item of type $1 + A$
 - ▶ Doing (2) means *defining* a function $\text{flatten} : F[\text{Option}[A]] \Rightarrow F[A]$
 - ★ standard library has $\text{flatten}[T] : \text{Seq}[\text{Option}[T]] \Rightarrow \text{Seq}[T]$
 - ▶ Express filter through flatten (see example code):
 - ★ Note: the Boolean type is isomorphic to $1 + 1$ or $\text{Option}[\text{Unit}]$
 - ★ $\text{filter}(p) = \text{fmap}(\text{optB}(p)) \circ \text{flatten}$, where we defined optB as

```
def optB[T](p: T => Option[Unit]): T => Option[T] =  
  x => p(x).map(_ => x)
```
- Express flatten through filter (using law 4):

```
def flatten[F[_],T](c: F[Option[T]]): F[T] =  
  c.filter(_.nonEmpty).map(_.get)  
// for F = Seq, this would be c.collect { case Some(x) => x }
```
- Law 4 is satisfied *automatically* if filter is defined via flatten !

* Filterable functors: The laws in depth II

Showing that law 4 is satisfied automatically if `filter` is defined via `flatten`

- Denote $\psi^{A \Rightarrow 1+A} \equiv \text{optB } (p^{A \Rightarrow 1+1}) = x^A \Rightarrow \text{fmap}^{\text{Opt}} (_ \Rightarrow x) (p(x))$
 - ▶ Have property: $f^{T \Rightarrow A} \circ \text{optB } (p^{A \Rightarrow 1+1}) = \text{optB } (f \circ p) \circ \text{fmap}^{\text{Opt}} f$
- Law 4: $\text{fmap } \psi \circ \text{flatten}^{F,T} \circ \text{fmap } f^{T \Rightarrow A} = \text{fmap } \psi \circ \text{flatten}^{F,T} \circ \text{fmap } f|_p$
 - ▶ We would like to interchange `flatten` and `fmap` here. Use Law 1?
- Reformulate Law 1 in terms of `flatten`:

$$\text{fmap } f^{T \Rightarrow A} \circ \text{fmap } \psi \circ \text{flatten}^{F,A} = \text{filter } (f \circ p) \circ \text{fmap } f$$

$$\text{fmap } (f^{T \Rightarrow A} \circ \text{optB } (p^{A \Rightarrow 1+A})) \circ \text{flatten}^{F,A} = \text{fmap } (\text{optB } (f \circ p)) \circ \text{flatten}^{F,T} \circ \text{fmap } f$$

$$\text{fmap}^F (\text{optB } (f \circ p)) \circ \text{fmap}^F (\text{fmap}^{\text{Opt}} f) = \text{fmap}^F (\text{optB } (f \circ p) \circ \text{fmap}^{\text{Opt}} f)$$

[remove common prefix $\text{fmap } (\text{optB } (f \circ p)) \circ \dots$ from both sides]

$$\text{fmap } (\text{fmap}^{\text{Opt}} f^{T \Rightarrow A}) \circ \text{flatten}^{F,A} = \text{flatten}^{F,T} \circ \text{fmap } f \quad \text{— law 1 for flatten}$$

- We can now interchange `flatten` and `fmap` in $\text{flatten}^{F,T} \circ \text{fmap } f|_p^{T \Rightarrow A}$:

$$\text{fmap } \psi \circ \text{flatten}^{F,T} \circ \text{fmap } f|_p = \text{fmap } \psi \circ \text{fmap } (\text{fmap}^{\text{Opt}} f|_p) \circ \text{flatten}^{F,A}$$

$$= \text{fmap } (\psi \circ \text{fmap}^{\text{Opt}} f) \circ \text{flatten}^{F,A} = \text{fmap } (\psi \circ \text{fmap}^{\text{Opt}} f|_p) \circ \text{flatten}^{F,A}$$

$$\psi \circ \text{fmap}^{\text{Opt}} f = \psi \circ \text{fmap}^{\text{Opt}} f|_p \quad \text{— check this by hand}$$

Filterable functors: The laws in depth III

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

$$\text{fmapOpt}^{F,A,B}(f^{A \Rightarrow 1+B}) : (A \Rightarrow 1+B) \Rightarrow F^A \Rightarrow F^B = \text{fmap } f \circ \text{flatten}^{F,B}$$

- fmapOpt and flatten are *equivalent*: $\text{flatten}^{F,A} = \text{fmapOpt}^{F,1+A,A}(\text{id}^{1+A \Rightarrow 1+A})$
- Express laws 1 – 3 in terms of fmapOpt and $\psi^{A \Rightarrow 1+A} \equiv \text{optB}(p)$
 - ▶ Express filter through fmapOpt : $\text{filter}(p) = \text{fmapOpt}^{F,A,A}(\psi)$
 - ▶ Consider the expression needed for law 2: $x \Rightarrow p_1(x)$ and $p_2(x)$
 - ★ Written in terms of ψ_1 and ψ_2 , this is $x^A \Rightarrow \psi_1(x).\text{flatMap}(\psi_2)$
 - ▶ 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: $\text{id}_{\diamond_{\text{Opt}}}^{A \Rightarrow 1+A} \equiv x^A \Rightarrow \text{Some}(x)$
 - ★ Kleisli composition \diamond_{Opt} is associative and respects the Kleisli identity!
- fmapOpt lifts a $\text{Kleisli}_{\text{Opt}}$ function $f^{A \Rightarrow 1+B}$ into the functor F
- Only *two* laws are necessary for fmapOpt !
 - 1 **Identity law** (covers old law 3): $\text{fmapOpt}(\text{id}_{\diamond_{\text{Opt}}}^{A \Rightarrow 1+A}) = \text{id}^{F^A \Rightarrow F^A}$
 - 2 **Composition law** (covers old laws 1 and 2):
 $\text{fmapOpt}(f^{A \Rightarrow 1+B}) \circ \text{fmapOpt}(g^{B \Rightarrow 1+C}) = \text{fmapOpt}(f \diamond_{\text{Opt}} g)$
 - ▶ The two laws for fmapOpt are very similar to the two functor laws

* Filterable functors: The laws in depth IV

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`:

$$\text{filter}(x^A \Rightarrow 0 + 1) = \text{fmap}(x^A \Rightarrow 0 + x) \circ \text{flatten} = \text{fmapOpt}(\text{id}_{\diamond_{\text{Opt}}}) = \text{id}^{F^A \Rightarrow F^A}$$

- Derive old law 2: need to work with $\psi \equiv \text{optB}(p) : A \Rightarrow 1 + A$

- ▶ The Boolean conjunction $x \Rightarrow p_1(x) \wedge p_2(x)$ corresponds to $\psi_1 \diamond_{\text{Opt}} \psi_2$
- ▶ Apply the composition law to Kleisli functions of types $A \Rightarrow 1 + A$:

$$\begin{aligned}\text{filter}(p_1) \circ \text{filter}(p_2) &= \text{fmapOpt}(\psi_1) \circ \text{fmapOpt}(\psi_2) \\ &= \text{fmapOpt}(\psi_1 \diamond_{\text{Opt}} \psi_2) = \text{fmapOpt}(\text{optB}(x \Rightarrow p_1(x) \wedge p_2(x)))\end{aligned}$$

- Derive old law 1: express `filter` through `fmapOpt`, so law 1 becomes

- ▶ $\text{fmap } f \circ \text{fmapOpt}(\text{optB}(p)) = \text{fmapOpt}(\text{optB}(f \circ p)) \circ \text{fmap } f$ – eq. (*)
- ▶ denote $k_f^{A \Rightarrow 1+A} = x^A \Rightarrow 0 + f(x)$; that is, $k_f = f \circ \text{id}_{\diamond_{\text{Opt}}}$; then we have $\text{fmapOpt}(k_f) = \text{fmap } k_f \circ \text{flatten} = \text{fmap } f \circ \text{fmap } \text{id}_{\diamond_{\text{Opt}}} \circ \text{flatten} = \text{fmap } f$
- ▶ rewrite (*) as $\text{fmapOpt}(k_f \diamond_{\text{Opt}} \text{optB}(p)) = \text{fmapOpt}(\text{optB}(f \circ p) \diamond_{\text{Opt}} k_f)$
- ▶ it remains to show that $k_f \diamond_{\text{Opt}} \text{optB}(p) = \text{optB}(f \circ p) \diamond_{\text{Opt}} k_f$
- ▶ use the properties $k_f \diamond_{\text{Opt}} \psi = f \circ \psi$ and $\psi \diamond_{\text{Opt}} k_f = \psi \circ \text{fmap}^{\text{Opt}} f$, and $f \circ \text{optB}(p) = \text{optB}(f \circ p) \circ \text{fmap}^{\text{Opt}} f$ (property from slide 8)

Summary so far

Filterable functors can be defined via `filter`, `flatten`, or `fmapOpt`

- All three are computationally *equivalent* but have different roles:
 - ▶ The easiest to use in program code is `filter` / `withFilter`
 - ▶ The easiest type signature to implement is `flatten`
 - ▶ The easiest to use for checking laws is `fmapOpt`
- The easiest way to derive the laws is to *begin* with simpler laws
- * The 2 laws for `fmapOpt` are functor laws with a Kleisli “twist”
- Category theory accommodates this via a generalized definition of functors as liftings between “twisted” function types. Compare:
 - ▶ $\text{fmap} : (A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$ – ordinary container (“endofunctor”)
 - ▶ $\text{contrafmap} : (B \Rightarrow A) \Rightarrow F^A \Rightarrow F^B$ – lifting from reversed functions
 - ▶ $\text{fmapOpt} : (A \Rightarrow 1 + B) \Rightarrow F^A \Rightarrow F^B$ – lifting from $\text{Kleisli}_{\text{Opt}}$ -functions
- CT gives us an intuition: prefer type signatures that resemble “lifting”
 - ▶ but CT is abstract, does not directly deliver a good formulation of laws
 - ▶ CT gives us no help with derivations when we struggle with the laws

Structure of filterable functors

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

- “reshuffling” means reusing different parts of a disjunction

Construction of exponential-polynomial filterable functors

- 1 $F^A = Z$ (constant functor) for any type Z (define $\text{fmapOpt } f = \text{id}$)
 - 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
- 3 $F^A \equiv G^A + H^A$ for any filterable functors G^A and H^A
- 4 $F^A \equiv G^{H^A}$ for *any* functor G^A and filterable functor H^A
- 5 $F^A \equiv 1 + A \times G^A$ 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 $F^A \equiv P + A \times A \times \dots \times A \times G^A$ similarly
- 6 $F^A \equiv G^A + A \times F^A$ (recursive) for a filterable functor G^A
- 7 $F^A \equiv G^A \Rightarrow H^A$ 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 $\text{fmapOpt}_F(f) : F^A \Rightarrow F^B$ and $\text{fmapOpt}_G(f) : G^A \Rightarrow G^B$
- Define $\text{fmapOpt}_{F \times G} f \equiv p^{F^A} \times q^{G^A} \Rightarrow \text{fmapOpt}_F(f)(p) \times \text{fmapOpt}_G(f)(q)$
- Identity law: $f = \text{id}_\diamond$, so $\text{fmapOpt}_F f = \text{id}$ and $\text{fmapOpt}_G f = \text{id}$
 - ▶ Hence we get $\text{fmapOpt}_{F+G}(f)(p \times q) = \text{id}(p) \times \text{id}(q) = p \times q$
- Composition law:

$$\begin{aligned} & (\text{fmapOpt}_{F \times G} f_1 \circ \text{fmapOpt}_{F \times G} f_2)(p \times q) \\ &= \text{fmapOpt}_{F \times G}(f_2)(\text{fmapOpt}_F(f_1)(p) \times \text{fmapOpt}_G(f_1)(q)) \\ &= (\text{fmapOpt}_F f_1 \circ \text{fmapOpt}_F f_2)(p) \times (\text{fmapOpt}_G f_1 \circ \text{fmapOpt}_G f_2)(q) \\ &= \text{fmapOpt}_F(f_1 \diamond f_2)(p) \times \text{fmapOpt}_G(f_1 \diamond f_2)(q) \\ &= \text{fmapOpt}_{F \times G}(f_1 \diamond f_2)(p \times q) \end{aligned}$$

- 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 $\text{fmapOpt}_G(f) : G^A \Rightarrow G^B$
- Define $\text{fmapOpt}_F(f)(1 + a^A \times q^{G^A})$ by returning $0 + b \times \text{fmapOpt}_G(f)(q)$ if the argument is $0 + a \times q$ and $f(a) = 0 + b$, and returning $1 + 0$ otherwise
- Identity law: $f = \text{id}_\diamond$, so $f(a) = 0 + a$ and $\text{fmapOpt}_G f = \text{id}$
 - ▶ Hence we get $\text{fmapOpt}_F(\text{id}_\diamond)(1 + a \times q) = 1 + a \times q$
- 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 $(f_1 \diamond f_2)(a) = 0 + c$; then

$$\begin{aligned} & (\text{fmapOpt}_F f_1 \circ \text{fmapOpt}_F f_2)(0 + a \times q) \\ &= \text{fmapOpt}_F(f_2)(\text{fmapOpt}_F(f_1)(0 + a \times q)) \\ &= \text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}_G(f_1)(q)) \\ &= 0 + c \times (\text{fmapOpt}_G f_1 \circ \text{fmapOpt}_G f_2)(q) \\ &= 0 + c \times \text{fmapOpt}_G(f_1 \diamond f_2)(q) \\ &= \text{fmapOpt}_F(f_1 \diamond f_2)(0 + a \times q) \end{aligned}$$

This is a “greedy filter”: if $f(a)$ is empty, deletes all G^A data

* 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 $f^{A \Rightarrow 1+B}$, get $\text{fmapOpt}_G(f) : G^A \Rightarrow G^B$ and $\text{fmapOpt}'_F(f) : F^A \Rightarrow F^B$ (for use in recursive arguments as the inductive assumption)
- Define $\text{fmapOpt}_F(f)(q^{G^A} + a^A \times p^{F^A})$ by returning $0 + \text{fmapOpt}'_F(f)(p)$ if $f(a) = 1 + 0$, and $\text{fmapOpt}_G(f)(q) + b \times \text{fmapOpt}'_F(f)(p)$ otherwise
- Identity law: $f(a) = \text{id}_\diamond(a) \neq 1 + 0$, so $\text{fmapOpt}_F(\text{id}_\diamond)(q + a \times p) = q + a \times p$
- Composition law:
$$(\text{fmapOpt}_F(f_1) \circ \text{fmapOpt}_F(f_2))(q + a \times p) = \text{fmapOpt}_F(f_1 \diamond f_2)(q + a \times p)$$
- 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 $(f_1 \diamond f_2)(a) = 1 + 0$; to show $\text{fmapOpt}'_F(f_2)(\text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \diamond f_2)(p)$, use the inductive assumption about $\text{fmapOpt}'_F$ on p
- If $f_1(a) = 0 + b$ and $f_2(b) = 1 + 0$ then $(f_1 \diamond f_2)(a) = 1 + 0$; to show $\text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \diamond f_2)(p)$, rewrite $\text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_2)(\text{fmapOpt}'_F(f_1)(p))$ and use the inductive assumption about $\text{fmapOpt}'_F$ on p

This is a “list-like filter”: if $f(a)$ is empty, recurses into nested F^A data

Worked examples II: Constructions of filterable functors IV

Use known filterable constructions to show that

$F^A \equiv (\text{Int} \times \text{String}) \Rightarrow (1 + \text{Int} \times A + A \times (1 + A) + (\text{Int} \Rightarrow 1 + A + A \times A \times \text{String}))$
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 ,
 - ▶ $R_1^A = (\text{Int} \times \text{String}) \Rightarrow A$ and $R_2^A = \text{Int} \Rightarrow A$
 - ▶ $G^A = 1 + \text{Int} \times A + A \times (1 + A)$ and $H^A = 1 + A + A \times A \times \text{String}$
- Now we can rewrite $F^A = R_1 [G^A + R_2 [H^A]]$
 - ▶ G^A is filterable by construction 5 because it is of the form $G^A = 1 + A \times K^A$ with filterable functor $K^A = 1 + \text{Int} + A$
 - ▶ 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 $1 + A \times \text{String}$ is filterable by constructions 5 and 1
- Constructions 3 and 4 show that $R_1 [G^A + R_2 [H^A]]$ is filterable

Note that there are more than one way of implementing `Filterable` here

* Exercises II

- 1 Implement a `Filterable` instance for `type F[T] = G[H[T]]` assuming that the functor `H[T]` already has a `Filterable` instance. Verify the laws rigorously.
- 2 For `type F[T] = Option[Int \Rightarrow Option[(T, T)]]`, implement a `Filterable` instance. Show that the filterable laws hold by using known filterable constructions (avoiding explicit proofs).
- 3 Implement a `Filterable` instance for $F^A \equiv G^A + \text{Int} \times A \times A \times F^A$ (recursive) for a filterable functor G^A . Verify the laws rigorously.
- 4 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.