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")

```
(for { x ← 0 to 100
     y = math.cos(x)
     if y > 0
   } yield math.sqrt(y)
).sum
```

```
(0 to 100).map { x ⇒
   math.cos(x) }.filter { y ⇒
   y > 0 }.map { y ⇒
     math.sqrt(y)
  }.sum
```

  - "Functor block" is a syntax for manipulating data within a container
    - ⋆ Container must be a functor (has `map` such that the laws hold)
    - ⋆ Data changes but remains within the same container
- A **filterable functor** is a functor that has a `withFilter` method
- Can use "`if`" when `withFilter(p: A⇒Boolean): F[A] ⇒ F[A]` is defined
  - 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:

- `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`
- `List[T]` $\equiv 1 + T + T \times T + T \times T \times T + ...$
  - `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 under certain conditions (amount < $1000, 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) – see example code
```

- This functor type is written as $F^A = (1 + A) \times (1 + A)$
  - When a value does not pass the filter, the A is replaced by 1
- Filtering is applied independently to both parts of the product type
- 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 into 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) // – more conditions, etc.; see example code
} yield //  for all x in list, such that conditions hold, compute this:
  k(x, y, z) //  all the new values will stay within the container
```

- 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 II: 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. $\text{filter}\,\left(x^A \Rightarrow \text{true}\right) = \text{id}^{F^A\Rightarrow F^A}$
  4. $\text{filter}\, p \circ \text{fmap}\, f^{A\Rightarrow B} = \text{filter}\, p \circ \text{fmap}\,\left(f_{|p}\right)$ where $f_{|p}$ is the *partial function* defined as `{ case x if p(x) ⇒ f(x) }` – only works if $p(x)$ holds
- Can define a type class `Filterable`, method `withFilter`
- Check the laws for the `Orders` functor (see example code)
  - Laws hold for the `Orders` functor with / without business rule (a)
  - Another filterable functor: $F^A \equiv 1 + A\times A$ ("collapsible 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 e.g. for $A = \text{Int}$ – breaks law 1
  - $F^A \equiv 1 + A$ defining $\text{filter}\,(p)\,(x) \equiv 1 + 0$ breaks law 3
  - $F^A \equiv A$ – must define $\text{filter}\,\left(p^{A\Rightarrow \text{Boolean}}\right)\left(x^A\right) = x$, breaking law 4
  - $F^A \equiv A\times(1+A)$ – unable to remove the first A, breaking law 4

The equational laws 1–4 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 received 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. $F^A \equiv \text{Int} + \text{Int} \times A + \text{Int} \times A \times A + \text{Int} \times A \times A \times A$
5. $F^A = \text{NonEmptyList}^A$ defined recursively as $F^A \equiv A + A \times F^A$
6. $F^{Z,A} \equiv Z + \text{Int} \times Z \times A \times A$ (with respect to the type parameter A)
7. $F^{Z,A} \equiv 1 + Z + \text{Int} \times A \times \text{List}^A$ (w.r.t. the type parameter A)
8. * Show that $C^{Z,A} = A \Rightarrow 1 + Z$ is a filterable *contrafunctor* w.r.t. A (implement `withFilter` with the same type signature; no law checking)

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. $F^A \equiv \text{Int} + \text{String} \times A \times A \times A$
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 $F^A \equiv 1 + A \times F^A \times F^A$
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 $C^A \equiv A + A \times A \Rightarrow 1 + Z$ 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 $\text{inflate} : F^A \Rightarrow F^{1+A} = \text{fmap}\,(\text{Some}^{A\Rightarrow 1+A})$
  - Filter $F^{1+A} \Rightarrow F^{1+A}$ using $\text{fmap}\left(x^{1+A} \Rightarrow \text{filter}_{\text{Opt}}(p^{A\Rightarrow\text{Boolean}})(x)\right)$

$$\text{filter}\,p : \;\; F^A \xrightarrow[\text{inflate}]{} F^{1+A} \xrightarrow[\text{fmap}\left(\text{filter}_{\text{Opt}}\,p\right)]{} F^{1+A} \xrightarrow[\text{flatten}]{} F^A$$

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

$$\text{filter}\,p : \;\; F^A \xrightarrow[\text{fmap(bop}\,p)]{} F^{1+A} \xrightarrow[\text{flatten}]{} F^A$$

Filterable functors: Using `flatten`

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

$$\text{flatten}: \; F^{1+A} \xrightarrow[\text{filter(.nonEmpty)}]{} F^{1+A} \xrightarrow[\text{fmap(.get)}]{} F^{A}$$

```
def flatten[F[_],A](foa: F[Option[A]]): F[A] =
  foa.filter(_.nonEmpty).map(_.get) // _.get is 0 + x^A ⇒ x^A
// for F = Seq, this would be foa.collect { case Some(x) ⇒ x }
// for arbitrary functor F we need to use the partial function, _.get
```

- This means `flatten` and `filter` are **computationally equivalent**
  - We could specify filterable functors by defining `flatten`
    - ⋆ The implementation of `filter` would then be derived by library
- Use `flatten` 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 `flatten` would be $(\text{Int} \Rightarrow 1 + A) \Rightarrow \text{Int} \Rightarrow A$
    - ⋆ cannot map $F^{1+A} \Rightarrow F^A$ because we do not have $1 + A \to A$

Filterable functors: The laws in depth II

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

$$
\begin{tikzcd}
 & F^{\mathbf{1}+A} \arrow[r, "\text{flatten}"] & F^A \arrow[dr, "\text{fmap}\, f^{A\Rightarrow B}"] & \\
F^A \arrow[ur, "\psi_p"] \arrow[dr, "\psi_p"'] & & & F^B \\
 & F^{\mathbf{1}+A} \arrow[r, "\text{flatten}"'] & F^A \arrow[ur, "\text{fmap}\, f_{|p}^{A\Rightarrow B}"'] &
\end{tikzcd}
$$

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

$$
\text{fmap}\, f^{A \Rightarrow B} \circ \psi_p \circ \text{flatten}^{F,B} = \psi_{f \circ p} \circ \text{flatten}^{F,A} \circ \text{fmap}\, f^{A \Rightarrow B}
$$

$$
\begin{tikzcd}
 & F^B \arrow[r, "\psi_p"] & F^{\mathbf{1}+B} \arrow[dr, "\text{flatten}^{F,B}"] & \\
F^A \arrow[ur, "\text{fmap}\, f^{A\Rightarrow B}"] \arrow[dr, "\psi_{f \circ p}"'] & & & F^B \\
 & F^{\mathbf{1}+A} \arrow[r, "\text{flatten}^{F,A}"'] & F^A \arrow[ur, "\text{fmap}\, f^{A\Rightarrow B}"'] &
\end{tikzcd}
$$

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

* Filterable functors: The laws in depth III

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

$$\begin{tikzcd} & B \arrow[dr, "\text{bop}\,p"] & \\ A \arrow[ur, "f^{A\Rightarrow B}"] \arrow[dr, "\text{bop}(f\circ p)"'] & & 1+B \\ & 1+A \arrow[ur, "\text{fmap}^{\text{Opt}} f"'] & \end{tikzcd}$$

We can now rewrite Law 1 as

$\text{fmap}\,(\text{bop}\,(f \circ p)) \circ \text{fmap}\,(\text{fmap}^{\text{Opt}} f) \circ \text{flatten} = \text{fmap}\,(\text{bop}\,(f \circ p)) \circ \text{flatten} \circ \text{fmap}\, f$

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

$\text{fmap}\,(\text{fmap}^{\text{Opt}} f^{A\Rightarrow B}) \circ \text{flatten}^{F,B} = \text{flatten}^{F,A} \circ \text{fmap}\, f^{A\Rightarrow B}$ – **law 1 for flatten**

$$\begin{tikzcd} & F^{1+B} \arrow[dr, "\text{flatten}^{F,B}"] & \\ F^{1+A} \arrow[ur, "\text{fmap}\,(\text{fmap}\,f)"] \arrow[dr, "\text{flatten}^{F,A}"'] & & F^B \\ & F^A \arrow[ur, "\text{fmap}\,f^{A\Rightarrow B}"'] & \end{tikzcd}$$

- `flatten`: $F^{1+A} \Rightarrow F^A$ 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 `flatten`:

$$\text{fmap}\,(\text{fmap}^{\text{Opt}} f^{A\Rightarrow B}) \circ \text{flatten}^{F,B} = \text{flatten}^{F,A} \circ \text{fmap}\, f^{A\Rightarrow B}$$

  Law 4 expressed via `flatten`:

$$\begin{tikzcd}
 & F^{1+A} \arrow[r, "\text{flatten}"] & F^A \arrow[dr, "\text{fmap}\, f^{A\Rightarrow B}"] & \\
F^A \arrow[ur, "\psi_p"] \arrow[dr, "\psi_p"'] & & & F^B \\
 & F^{1+A} \arrow[r, "\text{flatten}"'] & F^A \arrow[ur, "\text{fmap}\, f_{|p}^{A\Rightarrow B}"'] &
\end{tikzcd}$$

$$\psi_p \circ \text{flatten}^{F,A} \circ \text{fmap}\, f^{A\Rightarrow B} = \psi_p \circ \text{flatten}^{F,A} \circ \text{fmap}\, f_{|p}$$

- Use naturality to interchange `flatten` and `fmap` in both sides of law 4:

$$\psi_p \circ \text{fmap}\,(\text{fmap}^{\text{Opt}} f) \circ \text{flatten}^{F,B} = \psi_p \circ \text{fmap}\,(\text{fmap}^{\text{Opt}} f_{|p}) \circ \text{flatten}^{F,B}$$

$$[\text{omit flatten}^{F,B} \text{ from both sides; expand } \psi_p]$$

$$\text{bop}\, p \circ \text{fmap}^{\text{Opt}} f = \text{bop}\, p \circ \text{fmap}^{\text{Opt}} f_{|p} \quad - \quad \text{check this by hand:}$$

```
x ⇒ Some(x).filter(p).map(f)
x ⇒ Some(x).filter(p).map { x if p(x) ⇒ f(x) }
```

- These functions are equivalent because law 4 holds for `Option`

Filterable functors: The laws in depth V

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

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

$$\begin{tikzcd} & F^{1+B} \arrow[dr, "\text{flatten}^{F,B}"] & \\ F^A \arrow[ur, "\text{fmap}\, f^{A\Rightarrow 1+B}"] \arrow[rr, "\text{fmapOpt}^{F,A,B}\, f^{A\Rightarrow 1+B}"'] & & F^B \end{tikzcd}$$

- `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`: do they get simpler?
  - Express `filter` through `fmapOpt`: $\text{filter}\, p = \text{fmapOpt}^{F,A,A}\,(\text{bop}\, p)$
  - Consider the expression needed for law 2: $x \Rightarrow p_1(x) \wedge p_2(x)$
  - $\text{bop}\,(x \Rightarrow p_1(x) \wedge p_2(x)) = x^A \Rightarrow (\text{bop}\, p_1)\,(x).\text{flatMap}\,(\text{bop}\, p_2)$ – see code
    - Denote this computation by $\diamond_{\text{Opt}}$ and write

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

  - Similar to composition of functions, except the types are $A \Rightarrow 1 + B$
    - This is a particular case of **Kleisli composition**; the general case: $\diamond_M : \left(A \Rightarrow M^B\right) \Rightarrow \left(B \Rightarrow M^C\right) \Rightarrow \left(A \Rightarrow M^C\right)$; 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 $\diamond_{\text{Opt}}$ is associative and respects the Kleisli identity!
    - `fmapOpt` lifts a $\text{Kleisli}_{\text{Opt}}$ function $f^{A\Rightarrow 1+B}$ into the functor F

Filterable functors: The laws in depth VI

Simplifying down to two laws

- Only *two* laws are necessary for `fmapOpt`!

1. **Identity law** (covers old law 3):

$$\text{fmapOpt}\,(\text{id}_{\diamond_{\mathbf{Opt}}}^{A\Rightarrow\mathbf{1}+A}) = \text{id}^{F^A\Rightarrow F^A}$$

2. **Composition law** (covers old laws 1 and 2):

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

$$\begin{tikzcd}
 & F^B \arrow[dr, "\text{fmapOpt}\,(g^{B\Rightarrow\mathbf{1}+C})"] & \\
F^A \arrow[ur, "\text{fmapOpt}\,(f^{A\Rightarrow\mathbf{1}+B})"] \arrow[rr, "\text{fmapOpt}\left(f\diamond_{\mathbf{Opt}}g\right)"'] & & F^C
\end{tikzcd}$$

- 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 $F^A \Rightarrow F^B$)

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

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

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

$$\text{filter}\,(p_1) \circ \text{filter}\,(p_2) = \text{fmapOpt}\,(q_1) \circ \text{fmapOpt}\,(q_2)$$

$$= \text{fmapOpt}\,(q_1 \diamond_{\mathbf{Opt}} q_2) = \text{fmapOpt}\,(\text{bop}\,(x \Rightarrow p_1(x) \wedge p_2(x)))$$

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

$$\text{fmap}\, f \circ \text{fmapOpt}\,(\text{bop}\, p) = \text{fmapOpt}\,(\text{bop}\,(f \circ p)) \circ \text{fmap}\, f \quad - \text{eq. } (*)$$

  - lift $f^{A \Rightarrow B}$ to $\text{Kleisli}_{\text{Opt}}$ by defining $k_f^{A \Rightarrow 1+B} = f \circ \text{id}_{\diamond_{\mathbf{Opt}}}$; then we have $\text{fmapOpt}\,(k_f) = \text{fmap}\, k_f \circ \text{flatten} = \text{fmap}\, f \circ \text{fmap}\, \text{id}_{\diamond_{\mathbf{Opt}}} \circ \text{flatten} = \text{fmap}\, f$
  - rewrite eq. (*) as $\text{fmapOpt}\,(k_f \diamond_{\mathbf{Opt}} \text{bop}\, p) = \text{fmapOpt}\,(\text{bop}\,(f \circ p) \diamond_{\mathbf{Opt}} k_f)$
  - it remains to show that $k_f \diamond_{\mathbf{Opt}} \text{bop}\, p = \text{bop}\,(f \circ p) \diamond_{\mathbf{Opt}} k_f$
  - use the properties $k_f \diamond_{\mathbf{Opt}} q = f \circ q$ and $q \diamond_{\mathbf{Opt}} k_f = q \circ \text{fmap}^{\text{Opt}} f$, and $f \circ \text{bop}\, p = \text{bop}\,(f \circ p) \circ \text{fmap}^{\text{Opt}}\, f$ (property from slide 11)

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 and reason about is `flatten`
  - 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:
  - $\text{fmap} : (A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$ – 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 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 `flatten` in any CT book
  - CT is abstract, only gives hints and intuitions about further directions

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 a fixed 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 $F^A \equiv P + A \times A \times ... \times A \times G^A$ 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:

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

- 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, 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 $\text{fmapOpt}_G(f) : G^A \Rightarrow G^B$ 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 again use the inductive assumption about $\text{fmapOpt}'_F$ 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
$F^A \equiv (\text{Int} \times \text{String}) \Rightarrow (1 + \text{Int} \times A + A \times (1 + A) + (\text{Int} \Rightarrow 1 + A + A \times A \times \text{String}))$
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 $F^A = R_1 \left[G^A + R_2 \left[H^A \right] \right]$
  - 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 $R_1 \left[G^A + R_2 \left[H^A \right] \right]$ is filterable

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

* Exercises II

1. 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).
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.
5. 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 $F^A = A + (\text{Int} \Rightarrow A)$ is not filterable.

* Filterable contrafunctors I: Their definition

Bonus slide 1: When is a contrafunctor filterable?

When a contrafunctor C^A with $\text{contrafmap} : (B \Rightarrow A) \Rightarrow C^A \Rightarrow C^B$ has also

- `filter`/`withFilter`: $(A \Rightarrow \text{Boolean}) \Rightarrow C^A \Rightarrow C^A$ just like for functors
- `inflate`: $C^A \Rightarrow C^{\mathbf{1}+A}$ and `contrafmapOpt`: $(B \Rightarrow 1 + A) \Rightarrow C^A \Rightarrow C^B$
- All three functions are computationally equivalent...
  - $\text{filter}(p^{A\Rightarrow\text{Boolean}}) = \text{inflate}^{C^A \Rightarrow C^{\mathbf{1}+A}} \circ \text{contrafmap}(\text{bop}\, p)$
  - $\text{inflate}^{C^A \Rightarrow C^{\mathbf{1}+A}} = \text{contrafmap}\left(0 + x^A \Rightarrow x\right) \circ \text{filter}\left(\_ \Rightarrow \text{true}\right)$
  - $\text{contrafmapOpt}\, f^{B\Rightarrow\mathbf{1}+A} = \text{inflate} \circ \text{contrafmap}\, f$
  - $\text{inflate} = \text{contrafmapOpt}\,(\text{id}^{\mathbf{1}+A\Rightarrow\mathbf{1}+A})$
- but have different laws
  - 4 laws (naturality, conjunction, identity, partial function) for `filter`
  - 3 laws (naturality, conjunction, identity) for `inflate`
  - 2 laws (identity, contracomposition) for `contrafmapOpt`
    - ★ as before, `contrafmapOpt` is a "twisted" version of `contrafmap`
- Examples of filterable contrafunctors
  - $C^A \equiv A \Rightarrow 1 + Z$ where Z is a fixed type
  - $C^A \equiv 1 + A \Rightarrow Z$
- Examples of non-filterable contrafunctors
  - $C^A \equiv A \times F^A \Rightarrow Z$ – cannot implement `inflate`

- Filterable contrafunctors "can consume fewer data items"
- The easiest function to consider first is `inflate`

Constructions of filterable contrafunctors:

1. $C^A = Z$ (constant contrafunctor)
   Functor constructions (no need to check laws for these):
2. $F^A \equiv G^A \times H^A$ for any filterable contrafunctor G^A and H^A
3. $F^A \equiv G^A + H^A$ for any filterable contrafunctor G^A and H^A
4. $F^A \equiv G^{H^A}$ for H^A a filterable (contra)functor and G^A any (contra)functor – various combinations possible here
5. $F^A \equiv G^A \Rightarrow H^A$ if functor G^A and contrafunctor H^A *both filterable*
   Special constructions:
6. $F^A \equiv 1 + A \times G^A \Rightarrow H^A$ where G^A and H^A are filterable
7. $F^A \equiv A \times G^A \Rightarrow 1 + H^A$ if G^A and H^A are filterable