

Chapter 7: Computations lifted to a functor context II

Semimonads and monads

Sergei Winitzki

Academy by the Bay

2018-02-10

Computations within a functor context: Semimonads

Intuitions behind adding more “left 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”)

```
(for { i ← 1 to n          (1 to m).flatMap { i ⇒
    j ← 1 to n            (1 to n).flatMap { j ⇒
    k ← 1 to n            (1 to n).map { k ⇒
    } yield f(i, j, k)
    } yield f(i, j, k)
}).sum                      } } }.sum
```

- `map` replaces the last left arrow, `flatMap` replaces other left arrows
- Container must have both `map` (i.e. be a functor) and `flatMap`
 - ▶ When the functor is *also* filterable, we can use “`if`” as well
- Standard library defines `flatMap()` as equivalent of `map() ∘ flatten`
 - ▶ `(1 to n).flatMap(j ⇒ ...)` is `(1 to n).map(j ⇒ ...).flatten`
- `flatten: F[F[A]] ⇒ F[A]` can be expressed through `flatMap` as well:
 - ▶ `(xss: Seq[Seq[A]]).flatten = xss.flatMap { (xs: Seq[A]) ⇒ xs }`
- Functors having `flatMap`/`flatten` are “squashable” or **semimonads**

What is `flatMap` doing with the data in a collection?

Consider this schematic code using `Seq` as the container:

```
val result = for {  
  i ← 1 to m  
  j ← 1 to n  
  x = f(i, j)  
  k ← 1 to p  
  y = g(i,j,k)  
  ...  
} yield h(x,y)
```

Computations are repeated for all i , for all j , etc.

- The computation processes all elements from each collection
 - ▶ The number of resulting data items is $m * n * p$
 - ★ All the resulting data items must fit within *the same* container type!
 - ★ The set of *container capacity counts* is closed under multiplication
- What container types have this property?
 - ▶ `Seq`, `NonEmptyList` – can hold *any* number of elements \geq min. count
 - ▶ `Option`, `Either`, `Try`, `Future` – can hold 0 or 1 elements
 - ▶ “Tree-like” containers, e.g. binary tree (holds 1, 2, 4, 8, 16, ... 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” semimonads

`Seq`, `NonEmptyList`, `Iterator`, `Stream`

Typical tasks solved with “list-like” semimonads:

- 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

Examples: see code

- 1 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
- 6 Generalize example 4 to solve n -queens problem
- 7 Transform Boolean formulas between CNF and DNF

Intuitions for “single-value” semimonads

Option, Either, Try, Future

- The container can hold $n = 1$ or $n = 0$ values of type A

A schematic example of a functor block program using the `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`

Single-value semimonads: Examples I

- 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 `{ case x if p(x) => f(x) }` – only works if $p(x)$ holds
- Can define a type class `Filterable`, method `withFilter`

Single-value semimonads: Examples II

- 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 `{ case x if p(x) => f(x) }` – only works if $p(x)$ holds
- Can define a type class `Filterable`, method `withFilter`

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 $p(x) = \text{false}$, 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 F^A we had F^{1+A} i.e. $F[\text{Option}[A]]$
 - ▶ 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` ($x^{1+A} \Rightarrow \text{filter}_{\text{Opt}}(p^{A \Rightarrow \text{Boolean}})(x)$)

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

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

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

Filterable functors: Using `deflate`

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

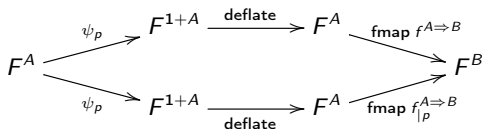
$$\text{deflate} : F^{1+A} \xrightarrow{\text{filter}(\cdot.\text{nonEmpty})} F^{1+A} \xrightarrow{\text{fmap}(\cdot.\text{get})} F^A$$

```
def deflate[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 `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 $1 \rightarrow A$
 - ▶ $F^A = \text{Int} \Rightarrow A$. Write $F^{1+A} = \text{Int} \Rightarrow 1 + A$
 - ★ type signature of `deflate` 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 $1 + A \rightarrow A$
- `deflate` is easier to implement and to reason about

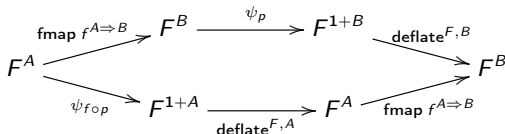
* Filterable functors: The laws in depth II

- 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 $\psi_p^{F^A \Rightarrow F^{1+A}} \equiv \text{fmap}(\text{bop } p)$ for brevity, then $\text{filter } p = \psi_p \circ \text{deflate}$
 - 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**:

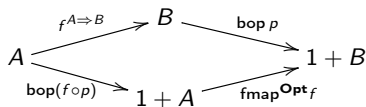
$$\text{fmap } f^{A \Rightarrow B} \circ \psi_p \circ \text{deflate}^{F, B} = \psi_{f \circ p} \circ \text{deflate}^{F, A} \circ \text{fmap } f^{A \Rightarrow B}$$



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

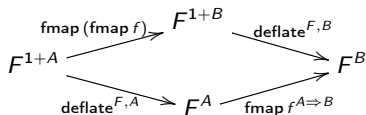


We can now rewrite Law 1 as

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

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

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



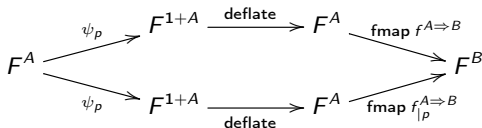
- deflate**: $F^{1+A} \Rightarrow F^A$ is a **natural transformation** (has naturality law)
 - Example: $F^A = 1 + A \times A$
 - $F^{1+A} = 1 + (1 + A) \times (1 + A) = 1 + 1 \times 1 + A \times 1 + 1 \times A + A \times A$
- natural transformations map containers $G^A \Rightarrow H^A$ by rearranging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

* Filterable functors: The laws in depth IV

- The naturality law for **deflate**:

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

Law 4 expressed via **deflate**:



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

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

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

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

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

`x \Rightarrow Some(x).filter(p).map(f)`

`x \Rightarrow Some(x).filter(p).map { x if p(x) \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

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

A commutative triangle diagram with F^A at the bottom-left vertex, F^{1+B} at the top vertex, and F^B at the bottom-right vertex. An arrow labeled $\text{fmap } f^{A \Rightarrow 1+B}$ points from F^A to F^{1+B} . An arrow labeled $\text{deflate}^{F,B}$ points from F^{1+B} to F^B . A direct arrow labeled $\text{fmapOpt } f^{A \Rightarrow 1+B}$ points from F^A to F^B .

- **fmapOpt** and **deflate** are *equivalent*: $\text{deflate}^{F,A} = \text{fmapOpt}^{F,1+A,A}(\text{id}^{1+A \Rightarrow 1+A})$
- Express laws 1 – 3 in terms of **fmapOpt**: do they get simpler?
 - ▶ Express **filter** through **fmapOpt**: $\text{filter } p = \text{fmapOpt}^{F,A,A}(\text{bop } p)$
 - ▶ 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_{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:
 $\diamond_M : (A \Rightarrow M^B) \Rightarrow (B \Rightarrow M^C) \Rightarrow (A \Rightarrow M^C)$; 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 Kleisli_{Opt} function $f^{A \Rightarrow 1+B}$ into the functor F

Filterable functors: The laws in depth VI

Simplifying down to two laws

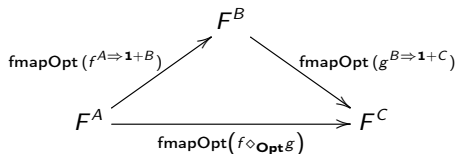
- Only *two* laws are necessary for `fmapOpt`!

- Identity law** (covers old law 3):

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

- 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
 - 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`:

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

- 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 $q_1 \diamond_{\text{Opt}} q_2$
- ▶ Apply the composition law to Kleisli functions of types $A \Rightarrow 1 + A$:

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

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

Summary: 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:
 - ▶ $\text{fmap} : (A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$ – 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 `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 $\text{Kleisli}_{\text{Opt}}$ 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

- 1 $F^A = Z$ (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
- 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: $f = \text{id}_{\diamond_{\text{Opt}}}$, 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:

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

- 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: $f = \text{id}_{\text{Opt}}$, so $f(a) = 0 + a$ and $\text{fmapOpt}_G f = \text{id}$
 - ▶ Hence we get $\text{fmapOpt}_F(\text{id}_{\text{Opt}})(1 + a \times q) = 1 + a \times q$
- 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 $(f_1 \diamond_{\text{Opt}} f_2)(a) = 0 + c$; then

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

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: $\text{id}_{\diamond_{\text{Opt}}}(x) \neq 1 + 0$, so $\text{fmapOpt}_F(\text{id}_{\diamond_{\text{Opt}}})(q + a \times p) = q + a \times p$
- Composition law:
 $(\text{fmapOpt}_F(f_1) \circ \text{fmapOpt}_F(f_2))(q + a \times p) = \text{fmapOpt}_F(f_1 \diamond_{\text{Opt}} f_2)(q + a \times p)$
- 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 $(f_1 \diamond_{\text{Opt}} f_2)(a) = 1 + 0$; to show $\text{fmapOpt}'_F(f_2)(\text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \diamond_{\text{Opt}} f_2)(p)$, use the inductive assumption about $\text{fmapOpt}'_F$ on p
- If $f_1(a) = 0 + b$ and $f_2(b) = 1 + 0$ then $(f_1 \diamond_{\text{Opt}} f_2)(a) = 1 + 0$; to show $\text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \diamond_{\text{Opt}} f_2)(p)$, 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 $F^A = R_1 [G^A + R_2 [H^A]]$
 - ▶ 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 (construction 4). Verify the laws rigorously (i.e. by calculations, not tests).
- 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 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.
- 7 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).

* Bonus slide I: Definition of filterable contrafunctors

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**: $(A \Rightarrow \text{Boolean}) \Rightarrow C^A \Rightarrow C^A$ – same as for functors
- **inflate**: $C^A \Rightarrow C^{1+A}$ and **contrafmapOpt**: $(B \Rightarrow 1 + A) \Rightarrow C^A \Rightarrow C^B$
- All three functions are computationally equivalent...
 - ▶ $\text{filter}(p^{A \Rightarrow \text{Boolean}}) = \text{inflate}^{C^A \Rightarrow C^{1+A}} \circ \text{contrafmap}(\text{bop } p)$
 - ▶ $\text{inflate}^{C^A \Rightarrow C^{1+A}} = \text{contrafmap}(0 + x^A \Rightarrow x) \circ \text{filter}(_ \Rightarrow \text{true})$
 - ▶ $\text{contrafmapOpt } f^{B \Rightarrow 1+A} = \text{inflate} \circ \text{contrafmap } f$
 - ▶ $\text{inflate} = \text{contrafmapOpt}(\text{id}^{1+A \Rightarrow 1+A})$
- 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**

* Bonus slide II: Structure of filterable contrafunctors

How to build up a filterable contrafunctor from parts?

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

Some 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