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

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
    j ← 1 to n
    k ← 1 to n
  } yield f(i, j, k)
).sum
```

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

- `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() ◦ 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 "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:

```
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)
```

```
val result = {
  (1 to m).flatMap { i ⇒
    (1 to n).flatMap { j ⇒
      val x = f(i,j)
      (1 to p).map { k ⇒
        val y = g(i,j,k)
        h(x,y)  } } } }
```

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 $\geq$ 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 solved with "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.

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 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 `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 solved with pass/fail monads:

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

Examples: see code

1. Read values of Java properties, checking that they all exist
2. Obtain values from `Future` computations in sequence
3. Make arithmetic safe by returning error messages in `Either`
4. 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 solved with tree-like monads:

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

Examples: see code

1. Implement variable substitution for a simple arithmetic language
2. ???

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:

- Collect extra information about computations along the way
- Chain of computations with a nonstandard evaluation strategy

Examples: see code

1. Dependency injection with the `Reader` monad
2. Perform computations and log information about each step
3. Perform lazy or memoized computations in a sequence
4. A chain of asynchronous operations
5. 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 ← c`, the value of `x` will *go over items* held in container `c`
- Manipulating items in container is followed by a generator:

```
x ← cont1                          y ← cont1
y = f(x)                                .map(x ⇒ f(x))
z ← cont2(y)                       z ← cont2(y)
```

```
cont1.flatMap(x ⇒ cont2(f(x))) = cont1.map(f).flatMap(y ⇒ cont2(y))
```

- Manipulating items in container is preceded by a generator:

```
x ← cont1                          x ← cont1
y ← cont2(x)                       z ← cont2(x)
z = f(y)                                  .map(f)
```

```
cont1.flatMap(cont2).map(f) = cont1.flatMap(x ⇒ cont2(x).map(f))
```

- After `x ← c`, further computations will use *all those* `x`

```
x ← cont                           y ← for { x ← cont
y ← p(x)                                     yy ← p(x) } yield yy
z ← cont2(y)                       z ← cont2(y)
```

```
cont.flatMap(x ⇒ 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 $\text{flm}^{[S,A,B]} : (A \Rightarrow S^B) \Rightarrow S^A \Rightarrow S^B$ with 2 laws:

1. $\text{flm}\,(f^{A\Rightarrow B} \circ g^{B\Rightarrow S^C}) = \text{fmap}\,f \circ \text{flm}\,g$ (naturality in A)

$$\begin{tikzcd} & S^B \arrow[dr, "\text{flm}\,g^{B\Rightarrow S^C}"] & \\ S^A \arrow[ur, "\text{fmap}\,f^{A\Rightarrow B}"] \arrow[rr, "\text{flm}\,(f^{A\Rightarrow B}\circ g^{B\Rightarrow S^C})"'] & & S^C \end{tikzcd}$$

2. $\text{flm}\left(f^{A\Rightarrow S^B} \circ h^{S^B\Rightarrow S^C}\right) = \text{flm}\,f \circ h$ (naturality in B)

$$\begin{tikzcd} & S^B \arrow[dr, "h^{S^B\Rightarrow S^C}"] & \\ S^A \arrow[ur, "\text{flm}\,f^{A\Rightarrow S^B}"] \arrow[rr, "\text{flm}\,(f^{A\Rightarrow S^B}\circ h^{S^B\Rightarrow S^C})"'] & & S^C \end{tikzcd}$$

$\text{flm}\left(f^{A\Rightarrow S^B} \circ \text{flm}\,g^{B\Rightarrow S^C}\right) = \text{flm}\,f \circ \text{flm}\,g$ (composition) – this is law 2 with $h = \text{flm}\,g$

$$\begin{tikzcd} & S^B \arrow[dr, "\text{flm}\,g^{B\Rightarrow S^C}"] & \\ S^A \arrow[ur, "\text{flm}\,f^{A\Rightarrow S^B}"] \arrow[rr, "\text{flm}\left(f^{A\Rightarrow S^B}\circ \text{flm}\,g^{B\Rightarrow S^C}\right)"'] & & S^C \end{tikzcd}$$

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:

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

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

$$\begin{tikzcd}
 & S^{S^B} \arrow[dr, "\mathrm{ftn}"] & \\
S^A \arrow[ur, "\mathrm{fmap}\, f^{A\Rightarrow S^B}"] \arrow[rr, "\mathrm{flm}\left(f^{A\Rightarrow S^B}\right)"'] & & S^B
\end{tikzcd}$$

It turns out that `flatten` has *only one* law:

1. $\mathrm{fmap}\left(h^{S^A\Rightarrow S^B}\right) \circ \mathrm{ftn}^{[S,B]} = \mathrm{ftn}^{[S,A]} \circ \mathrm{fmap}\, f$ (naturality)

$$\begin{tikzcd}
 & S^{S^B} \arrow[dr, "\mathrm{ftn}^{[S,B]}"] & \\
S^{S^A} \arrow[ur, "\mathrm{fmap}\left(h^{S^A\Rightarrow S^B}\right)"] \arrow[dr, "\mathrm{ftn}^{[S,A]}"'] & & S^B \\
 & S^A \arrow[ur, "h^{S^A\Rightarrow S^B}"'] &
\end{tikzcd}$$

$\mathrm{fmap}\left(\mathrm{ftn}^{[S]}\right) \circ \mathrm{ftn}^{[S]} = \mathrm{ftn}^{\left[S^S\right]} \circ \mathrm{ftn}^{[S]}$ (associativity) – this is law 1 with $h = \mathrm{ftn}$

$$\begin{tikzcd}
 & S^{S^A} \arrow[dr, "\mathrm{ftn}^{[S]}"] & \\
S^{S^{S^A}} \arrow[ur, "\mathrm{fmap}\left(\mathrm{ftn}^{[S]}\right)"] \arrow[dr, "\mathrm{ftn}^{\left[S^S\right]}"'] & & S^A \\
 & S^{S^A} \arrow[ur, "\mathrm{ftn}^{[S]}"'] &
\end{tikzcd}$$

Semimonad laws III: Deriving the laws for `flatten`

Denote for brevity $f_{\uparrow} \equiv \text{fmap}^{[S]} f$

Express $\text{flm}\, f = f_{\uparrow} \circ \text{ftn}$ and substitute into flm's 2 laws:

1. $\text{flm}\,(f \circ g) = f_{\uparrow} \circ \text{flm}\, g$ 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. $\text{flm}\,(f \circ h) = \text{flm}\, f \circ h$ gives $(f \circ h)_{\uparrow} \circ \text{ftn} = f_{\uparrow} \circ \text{ftn} \circ h$;
   using the functor composition law, we reduce this to $h_{\uparrow} \circ \text{ftn} = \text{ftn} \circ h$
   – this is the naturality law for `flatten`

- `flatten` has the simplest type signature and just one law (naturality)
- It is usually easy to check naturality!

Semimonad laws IV: Checking naturality of `flatten`

Implement `flatten` for various standard monads and check naturality

- $F^A \equiv 1 + A$; $\text{ftn} : 1 + 1 + A \Rightarrow 1 + A$
- $F^A \equiv Z + A$; $\text{ftn} : Z + Z + A \Rightarrow Z + A$
- $F^A \equiv \text{List}^A$; $\text{ftn} : \text{List}^{\text{List}^A} \Rightarrow \text{List}^A$
- $F^A \equiv A \times W$; $\text{ftn} : A \times W \times W \Rightarrow A \times W$
- $F^A \equiv R \Rightarrow A$; $\text{ftn} : (R \Rightarrow R \Rightarrow A) \Rightarrow R \Rightarrow A$
- $F^A \equiv S \Rightarrow A \times S$; $\text{ftn} : (S \Rightarrow (S \Rightarrow A \times S) \times S) \Rightarrow S \Rightarrow A \times S$
- $F^A \equiv (A \Rightarrow R) \Rightarrow R$;
  $\text{ftn} : ((((A \Rightarrow R) \Rightarrow R) \Rightarrow R) \Rightarrow R) \Rightarrow (A \Rightarrow R) \Rightarrow R$

Naturality of all these `flatten` functions is easy to verify

- Code implementing `flatten` is fully parametric in A

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 $\text{fmap}\left(x^{1+A} \Rightarrow \text{filter}_{\text{Opt}}(p^{A\Rightarrow\text{Boolean}})(x)\right)$

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

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

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

```
def deflate[F[_],A](foa: F[Option[A]]): F[A] =
  foa.filter(_.nonEmpty).map(_.get) // _.get is 0 + x^A ⇒ 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 $\psi_p^{F^A \Rightarrow F^{\mathbf{1}+A}} \equiv \text{fmap}\,(\text{bop}\,p)$ for brevity, then $\text{filter}\,p = \psi_p \circ \text{deflate}$
  - Law 4 then becomes: $\psi_p \circ \text{deflate} \circ \text{fmap}\,f^{A\Rightarrow B} = \psi_p \circ \text{deflate} \circ \text{fmap}\,f_{|p}$

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

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

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

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

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

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

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

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

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

- `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`:

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

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

- 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]

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

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

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

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

- `fmapOpt` and `deflate` are *equivalent*: $\text{deflate}^{F,A} = \text{fmapOpt}^{F,\mathbb{1}+A,A}(\text{id}^{\mathbb{1}+A\Rightarrow \mathbb{1}+A})$
- 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

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

  - 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: $\text{id}_{\diamond_{\text{Opt}}}^{A\Rightarrow \mathbb{1}+A} \equiv x^A \Rightarrow \text{Some}\,(x)$
    - ★ Kleisli composition $\diamond_{\text{Opt}}$ is associative and respects the Kleisli identity!
    - ★ `fmapOpt` lifts a $\text{Kleisli}_{\text{Opt}}$ function $f^{A\Rightarrow \mathbb{1}+B}$ into the functor F

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

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

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

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

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

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

- 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 $f^{A\Rightarrow\mathbb{1}+B}$ 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`:

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

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

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

- 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 $k_f^{A \Rightarrow 1+B} = f \circ \text{id}_{\diamond_{\textbf{Opt}}}$; then we have $\text{fmapOpt}\,(k_f) = \text{fmap}\,k_f \circ \text{deflate} = \text{fmap}\,f \circ \text{fmap}\,\text{id}_{\diamond_{\textbf{Opt}}} \circ \text{deflate} = \text{fmap}\,f$
  - rewrite eq. (*) as $\text{fmapOpt}\,(k_f \diamond_{\textbf{Opt}} \text{bop}\,p) = \text{fmapOpt}\,(\text{bop}\,(f \circ p) \diamond_{\textbf{Opt}} k_f)$
  - it remains to show that $k_f \diamond_{\textbf{Opt}} \text{bop}\,p = \text{bop}\,(f \circ p) \diamond_{\textbf{Opt}} k_f$
  - use the properties $k_f \diamond_{\textbf{Opt}} q = f \circ q$ and $q \diamond_{\textbf{Opt}} k_f = q \circ \text{fmap}^{\text{Opt}} f$, 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 $\text{fmapOpt}\, f = \text{id}$)
   - 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: $f = \text{id}_{\diamond_{\text{Opt}}}$, 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: $f = \text{id}_{\diamond_{\textbf{Opt}}}$, so $f(a) = 0 + a$ and $\text{fmapOpt}_G f = \text{id}$
  - Hence we get $\text{fmapOpt}_F(\text{id}_{\diamond_{\textbf{Opt}}})(1 + a \times q) = 1 + a \times q$
- 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

(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: $\text{id}_{\diamond_{\textbf{Opt}}}(x) \neq 1 + 0$, so $\text{fmapOpt}_F(\text{id}_{\diamond_{\textbf{Opt}}})(q + a \times p) = q + a \times p$
- Composition law:
  $(\text{fmapOpt}_F(f_1) \circ \text{fmapOpt}_F(f_2))(q + a \times p) = \text{fmapOpt}_F(f_1 \diamond_{\textbf{Opt}} f_2)(q + a \times p)$
- 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 $(f_1 \diamond_{\textbf{Opt}} f_2)(a) = 1 + 0$; to show $\text{fmapOpt}'_F(f_2)(\text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \diamond_{\textbf{Opt}} f_2)(p)$, use the inductive assumption about $\text{fmapOpt}'_F$ on p
- If $f_1(a) = 0 + b$ and $f_2(b) = 1 + 0$ then $(f_1 \diamond_{\textbf{Opt}} f_2)(a) = 1 + 0$; to show $\text{fmapOpt}_F(f_2)(0 + b \times \text{fmapOpt}'_F(f_1)(p)) = \text{fmapOpt}'_F(f_1 \diamond_{\textbf{Opt}} f_2)(p)$, 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 $F^A = R_1 \left[G^A + R_2 \left[H^A \right] \right]$
  - 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.
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).