

Chapter 9: Traversable functors and contrafunctors

Sergei Winitzki

Academy by the Bay

2018-08-08

Motivation for the `traverse` operation

- Consider data of type List^A and processing $f : A \Rightarrow \text{Future}^B$
- Typically, we want to wait until the entire data set is processed
- What we need is $\text{List}^A \Rightarrow (A \Rightarrow \text{Future}^B) \Rightarrow \text{Future}^{\text{List}^B}$
- Generalize: $L^A \Rightarrow (A \Rightarrow F^B) \Rightarrow F^{L^B}$ for some type constructors F, L
- This operation is called `traverse`
 - ▶ How to implement it: for example, a 3-element list is $A \times A \times A$
 - ▶ Consider $L^A \equiv A \times A \times A$, apply map f and get $F^B \times F^B \times F^B$
 - ▶ We will get $F^{L^B} \equiv F^{B \times B \times B}$ if we can apply `zip` as $F^B \times F^B \Rightarrow F^{B \times B}$
- So we need to assume that F is applicative
- In Scala, we have `Future.traverse()` that assumes L to be a sequence
 - ▶ This is the easy-to-remember example that fixes the requirements
- Questions:
 - ▶ Which functors L can have this operation?
 - ▶ Can we express `traverse` through a simpler operation?
 - ▶ What are the required laws for `traverse`?
 - ▶ What about contrafunctors or profunctors?

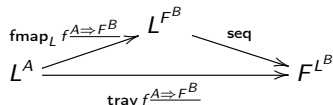
Deriving the `sequence` operation

- The type signature of `traverse` is a complicated “lifting”
 - ▶ A “lifting” is often equivalent to a simpler natural transformation
- To derive it, ask: what is missing from `fmap` to do the job of `traverse`?

$$\text{fmap}_L : (A \Rightarrow F^B) \Rightarrow L^A \Rightarrow L^{F^B}$$

- We need F^{L^B} , but the `traverse` operation gives us L^{F^B} instead
 - ▶ What’s missing is a natural transformation `sequence` : $L^{F^B} \Rightarrow F^{L^B}$
- The functions `traverse` and `sequence` are computationally equivalent:

$$\text{trav } f \xrightarrow{A \Rightarrow F^B} = \text{fmap}_L f \circ \text{seq}$$



Here F is an *arbitrary* applicative functor

- ▶ Keep in mind the example `Future.sequence` : $\text{List}^{\text{Future}^X} \Rightarrow \text{Future}^{\text{List}^X}$
- ▶ Examples: $L^A \equiv A \times A \times A$; $L^A = \text{List}^A$; finite trees
- ▶ Non-traversable: $L^A \equiv R \Rightarrow A$; lazy list (“infinite product”)

★ Note: We *cannot* have the opposite transformation $F^{L^B} \Rightarrow L^{F^B}$

Polynomial functors are traversable

- Generalize from the example $L^A \equiv A \times A \times A$ to other polynomials
- Polynomial functors have the form

$$L^A \equiv Z \times A \times \dots \times A + Y \times A \times \dots \times A + \dots + Q \times A + P$$

- To implement $\text{seq} : L^{F^B} \Rightarrow F^{L^B}$, consider monomial $L^A \equiv Z \times A \times \dots \times A$
- We have $L^{F^B} = Z \times F^B \times \dots \times F^B$; apply `zip` and get $Z \times F^{B \times \dots \times B}$
- Lift Z into the functor F using $Z \Rightarrow F^A \Rightarrow F^{Z \times A}$ (or with $F.\text{pure}$)
- The result is $F^{Z \times B \times \dots \times B} \equiv F^{L^B}$
 - ▶ For a polynomial L^A , do this to each monomial, then lift to F^{L^B}
 - ▶ Note that we could apply `zip` in various different orders
- The traversal order is arbitrary, may be application-specific
- Non-polynomial functors are not traversable (see Bird et al., 2013)
 - ▶ Example: $L^A \equiv E \Rightarrow A$; $F^A \equiv 1 + A$; can't have $\text{seq} : L^{F^B} \Rightarrow F^{L^B}$
- All polynomial functors are traversable, and usually in several ways
 - ▶ It is still useful to have a type class for traversable functors

Motivation for the laws of the `traverse` operation

- The “**law of traversals**” paper (2012) argues that `traverse` should “visit each element” of the container L^A exactly once, and evaluate each corresponding “effect” F^B exactly once; then they formulate the laws
- To derive the laws, use the “lifting” intuition for `traverse`,

$$\text{trav} : (A \Rightarrow F^B) \Rightarrow L^A \Rightarrow F^{L^B}$$

Look for “identity” and “composition” laws:

- ① “Identity” as `pure` : $A \Rightarrow F^A$ must be lifted to `pure` : $L^A \Rightarrow F^{L^A}$
- ② “Identity” as $\text{id}^{A \Rightarrow A}$ with $F^A \equiv A$ (identity functor) lifted to $\text{id}^{L^A \Rightarrow L^A}$
- ③ “Compose” $f : A \Rightarrow F^B$ and $g : B \Rightarrow G^C$ to get $h : A \Rightarrow F^{G^C}$, where F, G are applicative; a traversal with h maps L^A to $F^{G^{L^C}}$ and must be equal to the composition of traversals with f and then with $g^{F\uparrow}$

Questions:

- Are the laws for the `sequence` operation simpler?
- Are all these laws independent?
- What functors L satisfy these laws *for all* applicative functors F ?

Formulation of the laws for `traverse`

- Identity law: For any applicative functor F ,

$$\text{trav}(\text{pure}) = \text{pure}$$

$$L^A \xrightarrow[\text{trav}(\text{pure}^{A \Rightarrow F^A})]{\text{pure}^{L^A \Rightarrow F^{L^A}}} F^{L^A}$$

- Second identity law: $\text{trav}^{\text{Id}}(\text{id}^A) = \text{id}^{L^A}$ is a consequence with $F = \text{Id}$

★ So, we need only one identity law

- Composition law: For any $f^{A \Rightarrow F^B}$ and $g^{B \Rightarrow G^C}$, & applicative F and G ,

$$\text{trav } f \circ (\text{trav } g)^{F\uparrow} = \text{trav} (f \circ g^{F\uparrow})$$

$$\begin{array}{ccc} & F^{L^B} & \\ \text{trav}^F f^{A \Rightarrow F^B} \nearrow & & \searrow \text{fmap}_F(\text{trav}^G g)^{L^B \Rightarrow G^{L^C}} \\ L^A & \xrightarrow{\quad} & F^{G^{L^C}} \\ & \text{trav}^{FG} h^{A \Rightarrow F^{G^C}} \nearrow & \end{array}$$

where $h^{A \Rightarrow F^{G^C}} \equiv f \circ g^{F\uparrow}$. (Note: $H^A \equiv F^{G^A}$ is applicative!)

Derivation of the laws for [sequence](#)

Express $\text{trav } f = f^{L\uparrow} \circ \text{seq}$ and substitute into the laws for trav :

- Identity law: $\text{trav } (\text{pure}) = \text{pure}^{L\uparrow} \circ \text{seq} = \text{pure}$

$$\begin{array}{ccc}
 & L^{FA} & \\
 \text{fmap}_L \text{ pure}^A \nearrow & & \searrow \text{seq} \\
 L^A & \xRightarrow{\text{pure}^{LA}} & F^{LA}
 \end{array}$$

Naturality law: $\text{seq} \circ g^{F\uparrow L\uparrow} = g^{L\uparrow F\uparrow} \circ \text{seq}$ with $g^{A \Rightarrow B}$, mapping $L^{FA} \Rightarrow F^{LB}$

- Composition law:

$$\begin{aligned}
 \text{trav } f \circ (\text{trav } g)^{F\uparrow} &= f^{L\uparrow} \circ \text{seq} \circ (g^{L\uparrow} \circ \text{seq})^{F\uparrow} \\
 &= f^{L\uparrow} \circ \text{seq} \circ g^{L\uparrow F\uparrow} \circ \text{seq}^{F\uparrow} = f^{L\uparrow} \circ g^{F\uparrow L\uparrow} \circ \text{seq} \circ \text{seq}^{F\uparrow} \\
 \text{trav } (f \circ g^{F\uparrow}) &= (f \circ g^{F\uparrow})^{L\uparrow} \circ \text{seq} = f^{L\uparrow} \circ g^{F\uparrow L\uparrow} \circ \text{seq}
 \end{aligned}$$

Now omit the common prefix $f \cdots \circ g \cdots$ and obtain: $\text{seq} \circ \text{seq}^{F\uparrow} = \text{seq}$

$$\begin{array}{ccc}
 & F^{LG^C} & \\
 \text{seq}^F \nearrow & & \searrow (\text{seq}^G)^{F\uparrow} \\
 L^{FG^C} & \xRightarrow{\text{seq}^{FG}} & F^{GL^C}
 \end{array}$$

Constructions of traversable and bitraversable functors

Constructions of traversable functors:

- ① $L^A \equiv Z$ (constant functor) and $L^A \equiv A$ (identity functor)
 - ② $L^A \equiv G^A \times H^A$ for any traversable G^A and H^A
 - ③ $L^A \equiv G^A + H^A$ for any traversable G^A and H^A
 - ④ $L^A \equiv S^{A,L^A}$ (recursive) for a bitraversable bifunctor $S^{A,B}$
 - If L^A is infinite, laws will appear to hold but `seq` will not terminate
- A bifunctor $S^{A,B}$ is **bitraversable** if `bisequence` exists such that

$$\text{biseq} : S^{F^A, F^B} \Rightarrow F^{S^{A,B}}$$

for any applicative functor F ; the analogous laws must hold

Constructions of bitraversable bifunctors:

- ① $S^{A,B} \equiv Z$, $S^{A,B} \equiv A$, and $S^{A,B} = B$
- ② $S^{A,B} \equiv G^{A,B} \times H^{A,B}$ for any bitraversable G and H
- ③ $S^{A,B} \equiv G^{A,B} + H^{A,B}$ for any bitraversable G and H
- All polynomial bifunctors are bitraversable
- All polynomial functors, including recursive functors, are traversable

Foldable functors: traversing with respect to a monoid

- Take $F^A \equiv Z$ where Z is a monoid
 - ▶ The `zip` operation is the monoid operation \oplus
- The type signature of `traverse` becomes $(A \Rightarrow Z) \Rightarrow L^A \Rightarrow Z$
 - ▶ This method is called `foldMap`
- The type signature of `seq` becomes $L^Z \Rightarrow Z$
 - ▶ This is called `mconcat` – combines all values in L^Z with Z 's \oplus
- It is convenient to define the `Foldable` type class
 - ▶ But it has no laws any more
 - ▶ All traversable functors are also foldable
- The `foldLeft` method can be defined via `foldMap` with $Z \equiv (B \Rightarrow B)$:

$$\text{foldl} : (A \Rightarrow B \Rightarrow B) \Rightarrow L^A \Rightarrow B \Rightarrow B$$

Traversable contrafunctors and profunctors are not useful

Traversing profunctors with respect to functors F : effects of F are ignored

- All contrafunctors C^A are traversable: for any applicative functor F^A ,

$$\text{seq} : C^{F^A} \Rightarrow F^{C^A} \equiv \text{pure}^{C\downarrow} \circ \text{pure}$$

$$C^{F^A} \xrightarrow{\text{cmap}_C \text{pure}_F^A} C^A \xrightarrow{\text{pure}_F^{C^A}} F^{C^A}$$

- But not profunctors that are neither functors not contrafunctors
- Counterexample: $P^A \equiv A \Rightarrow A$; need $\text{seq} : (F^A \Rightarrow F^A) \Rightarrow F^{A \Rightarrow A}$; we can't get an $A \Rightarrow A$, so the only implementation is to return $\text{pure}_F(\text{id})$, which ignores its argument and so will fail the identity law

Traversing profunctors L with respect to profunctors F : effects are ignored

- Counterexample 1: contrafunctor $L^A \equiv R \Rightarrow A$ and contrafunctor $F^A \equiv S \Rightarrow A$, cannot have seq of type $L^{F^A} \Rightarrow F^{L^A}$
- Counterexample 2: contrafunctor $F^A \equiv (R \Rightarrow A) \Rightarrow S$ and functor $L^A \equiv 1 + A$, seq must use pure_F without arguments. So, the result is trivial and probably not useful. (Any applicative contrafunctor can be permuted with any other profunctor!)

Examples of usage

- ➊ Convert foldable data structures to list
- ➋ Fold a tree to aggregate data
- ➌ Decorate a tree with traversal order labels; DFS, BFS
- ➍ Implement scan on a list, as traverse with a state monad
- ➎ Use “reverse state monad” to change traversal direction

Naturality with respect to applicative functor as parameter

- The `traverse` method must be generic in the functor F :

$$\text{trav}^{F,A,B} : (A \Rightarrow F^B) \Rightarrow L^A \Rightarrow F^{L^B}$$

The code of `traverse` should only use `pure` and `zip` from F

- A functor F^A is “generic in A ”: have `fmap` : $(A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$
- “Generic in F ” means that $(F \Rightarrow G) \Rightarrow \text{trav}^F \Rightarrow \text{trav}^G$ in some way

- ① Show that any traversable functor L admits a method

$$\text{consume} : (L^A \Rightarrow B) \Rightarrow L^{F^A} \Rightarrow F^B$$

for any applicative functor F . Show that `traverse` and `consume` are equivalent.