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 $\text{List}^A \Rightarrow \left(A \Rightarrow \text{Future}^B\right) \Rightarrow \text{Future}^{\text{List}^B}$
- Generalize: $L^A \Rightarrow \left(A \Rightarrow F^B\right) \Rightarrow F^{L^B}$ 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 iconic 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 always equivalent to a simpler natural transformation
- To derive it, ask: what is missing from `map` to do the job of `traverse`?

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

- 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:

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

$$\begin{tikzcd}
 & L^{F^B} \arrow[dr, "\text{seq}"] & \\
L^A \arrow[ur, "\text{fmap}\ f"] \arrow[rr, "\text{trav}\ (f^{\underline{A\Rightarrow F^B}})"'] & & F^{L^B}
\end{tikzcd}$$

Here F is an arbitrary applicative functor

The "liftings" show the types of category's morphisms

- The functions `fmap2` and `ap` are computationally equivalent:

$$\text{fmap}_2\, f^{A\Rightarrow B\Rightarrow Z} = \text{fmap}\, f \circ \text{ap}$$

$$\begin{tikzcd} & F^{B\Rightarrow Z} \arrow[dr, "\text{ap}"] & \\ F^A \arrow[ur, "\text{fmap}\, f"] \arrow[rr, "\text{fmap}_2\, (f^{A\Rightarrow B\Rightarrow Z})"'] & & (F^B \Rightarrow F^Z) \end{tikzcd}$$

  - Note the pattern: a natural transformation is equivalent to a lifting

Categorical overview of “regular” functor classes

The “liftings” show the types of category’s morphisms

Categorical overview of “regular” functor classes

The “liftings” show the types of category’s morphisms

Categorical overview of "regular" functor classes

The "liftings" show the types of category's morphisms

class name	**lifting's name and type signature**	**category's morphism**
functor	fmap : $(A \Rightarrow B) \Rightarrow F^A \Rightarrow F^B$	$A \Rightarrow B$
filterable	fmapOpt : $(A \Rightarrow 1 + B) \Rightarrow F^A \Rightarrow F^B$	$A \Rightarrow 1 + B$
monad	flm : $\left(A \Rightarrow F^B\right) \Rightarrow F^A \Rightarrow F^B$	$A \Rightarrow F^B$
applicative	ap : $F^{A \Rightarrow B} \Rightarrow F^A \Rightarrow F^B$	$F^{A \Rightarrow B}$
contrafunctor	contrafmap : $(B \Rightarrow A) \Rightarrow F^A \Rightarrow F^B$	$B \Rightarrow A$
profunctor	xmap : $(A \Rightarrow B) \times (B \Rightarrow A) \Rightarrow F^A \Rightarrow F^B$	$(A \Rightarrow B) \times (B \Rightarrow A)$
contra-filterable	contrafmapOpt : $(B \Rightarrow 1 + A) \Rightarrow F^A \Rightarrow F^B$	$B \Rightarrow 1 + A$
Not yet considered:		
comonad	coflm : $\left(F^A \Rightarrow B\right) \Rightarrow F^A \Rightarrow F^B$	$F^A \Rightarrow B$

Need to define each category's composition and identity morphism

Then impose the category laws, the naturality laws, and the functor laws

- Obtained a systematic picture of the "regular" type classes
- Some classes (e.g. contra-applicative) aren't covered by this scheme
- Some of the possibilities (e.g. "contramonad") don't actually work out

Exercises

1. Show that `pure` will be automatically a natural transformation when it is defined using `wu` as shown in the slides.
2. Use naturality of `pure` to show that $\text{pure}\, f \odot \text{pure}\, g = \text{pure}\, (f \circ g)$
3. Show that $F^A \equiv (A \Rightarrow Z) \Rightarrow (1 + A)$ is a functor but not applicative.
4. Show that P^S is a monoid if S is a monoid and P is any applicative functor, contrafunctor, or profunctor.
5. Implement an applicative instance for $F^A = 1 + \text{Int} \times A + A \times A \times A$.
6. Using applicative constructions, show without lengthy proofs that $F^A = G^A + H^{G^A}$ is applicative if G and H are applicative functors.
7. Explicitly implement contrafunctor construction 2 and prove the laws.
8. For any contrafunctor H^A, construction 5 says that $F^A \equiv H^A \Rightarrow A$ is applicative. Implement the code of `zip(fa, fb)` for this construction.
9. Show that the recursive functor $F^A \equiv 1 + G^{A \times F^A}$ is applicative if G^A is applicative and wu_F is defined recursively as $0 + \text{pure}_G\, (1 \times \text{wu}_F)$.
10. Explicitly implement profunctor construction 5 and prove the laws.
11. Prove rigorously that all exponential-polynomial type constructors are profunctors.
12. Implement profunctor and applicative instances for $P^A \equiv A + Z \times G^A$ where G^A is a given applicative profunctor and Z is a monoid.
13. Show that, for any profunctor P^A, one can implement a function of type $A \Rightarrow P^B \Rightarrow P^{A \times B}$ but not of type $A \Rightarrow P^B \Rightarrow P^A$.