Chapter 4: Functors

The Curry-Howard correspondence

Sergei Winitzki

Academy by the Bay

December 24, 2017

"Container-like" type constructors

- Visualize `Seq[T]` as a container with some items of type `T`
  - How to formalize this idea as a property of `Seq`?
- Another example of a container: `Future[T]`
  - a value of type `T` will be available later, or may fail to arrive

Let us separate the "bare container" functionality from other functionality

- A "bare container" will allow us to:
  - manipulate items held within the container
    - In FP, to "manipulate items" means to apply functions to values
- "Container holds items" = we can apply a function to the items
  - but the new items *remain* within the same container!
  - need `map: Container[A] ⇒ (A ⇒ B) ⇒ Container[B]`
- A "bare container" will *not* allow us to:
  - make a new container out of a given set of items
  - read values out of the container
  - add more items into container, delete items from container
  - wait until items are available in container, etc.

`Option[T]` as a container I

In the short notation: $\text{Option}^A = 1 + A$

The `map` function is required to have the type

$$\text{map}^{A,B} : 1 + A \Rightarrow (A \Rightarrow B) \Rightarrow 1 + B$$

Main questions:

- How to avoid "information loss" in this function?
- Does this `map` allow us to "manipulate values within the container"?

`Option[T]` as a container II

Avoiding "information loss" means:

- `map[A,A](opt)(x⇒x) == opt` – "**identity law**" for `map`
- We have two implementations of the type:

$$\mathrm{map}^{[A,B]} = (1 + a^A) \Rightarrow (f^{A \Rightarrow B}) = 1 + f(a)$$

  and

$$\mathrm{map}^{[A,B]} = (1 + a^A) \Rightarrow (f^{A \Rightarrow B}) = 1 + 0^B$$

  The second implementation has "information loss"!
- Short notation for code (type annotations are optional):

Short notation	**Scala code**
a^A	`val a: A`
$f^{[A]\,B \Rightarrow C}$	`def f[A]: B ⇒ C ...`
$a^A + b^B$	`x: Either[A, B] match {...}`
$a^A + 0^B$	`Left(a): Either[A, B]`
1	`()`

`Option[T]` as a container III

- Flip the two arguments in the type signature of `map`:

$$\text{fmap}^{[A,B]} : (A \Rightarrow B) \Rightarrow \text{Option}^A \Rightarrow \text{Option}^B$$

- A function is "**lifted**" from $A \Rightarrow B$ to $\text{Option}^A \Rightarrow \text{Option}^B$ by `fmap`:

$$\text{fmap}\left(f^{A \Rightarrow B}\right) : \text{Option}^A \Rightarrow \text{Option}^B$$

- Being able to manipulate values means that functions behave normally when lifted, i.e. when applied within the container
- The standard properties of function composition are

$$f^{A \Rightarrow B} \circ id^{B \Rightarrow B} = f^{A \Rightarrow B}$$

$$id^{A \Rightarrow A} \circ f^{A \Rightarrow B} = f^{A \Rightarrow B}$$

$$f^{A \Rightarrow B} \circ (g^{B \Rightarrow C} \circ h^{C \Rightarrow D}) = (f^{A \Rightarrow B} \circ g^{B \Rightarrow C}) \circ h^{C \Rightarrow D}$$

and should hold for the "lifted" functions as well!
- The "identity law" already requires that $\text{fmap}(\text{id}^{A \Rightarrow A}) = \text{id}^{\text{Option}^A \Rightarrow \text{Option}^A}$
- It remains to require that `fmap` should preserve function composition:

$$\text{fmap}\left(f^{A \Rightarrow B} \circ g^{B \Rightarrow C}\right) = \text{fmap}\left(f^{A \Rightarrow B}\right) \circ \text{fmap}\left(g^{B \Rightarrow C}\right)$$

Functor: the definition

An abstraction for "bare container" functionality

A **functor** is:

- a type constructor with a type parameter, e.g. `MyType[T]`
- such that a function `map` or, equivalently, `fmap` is available:

$$\mathrm{map}^{[A,B]} : \mathrm{MyType}^{A} \Rightarrow (A \Rightarrow B) \Rightarrow \mathrm{MyType}^{B}$$

$$\mathrm{fmap}^{[A,B]} : (A \Rightarrow B) \Rightarrow \mathrm{MyType}^{A} \Rightarrow \mathrm{MyType}^{B}$$

- such that the identity law and the composition law hold for any type `T`
  - The laws are easier to formulate in terms of `fmap`:

$$\mathtt{fmap}\,(\mathtt{id}) = \mathtt{id}$$

$$\mathtt{fmap}\,(f \circ g) = \mathtt{fmap}\,(f) \circ \mathtt{fmap}\,(g)$$

- Verify the laws for `Option[A]`: see test code

```
def fmap[A,B]: (A ⇒ B) ⇒ Option[A] ⇒ Option[B] = f ⇒ {
  case Some(a) ⇒ Some(f(a))
  case None ⇒ None
}
```

Functor: examples

(Almost) everything that has a "`map`" is a functor

- Need to verify the laws!

Examples of functors in the Scala standard library:

- `Option[T]`
- `Either[L, R] with respect to R`
- `Seq[T] and Iterator[T]`
- `the many subtypes of Seq (Range, List, Vector, IndexedSeq, etc.)`
- `Future[T]`
- `Try[T]`
- `Map[K, V] with respect to V (using mapValues)`

Example of non-functor that has a `map` in the standard library:

- `Set[T]`

See example code

What do we need to know about functors?

Specific functors will have methods for creating them, reading values out of them, adding / removing items, waiting for items to arrive, etc.
Main questions:

- Given a type, how to recognize whether it is a functor?
- If so, how to implement the `map` function that satisfies the laws?
- Can we build new functors out of given ones?

Other topics:

- Contrafunctors, profunctors, and type constructors that are neither
- Implementing Functor instance using Cats
- Implementing Functor instance for recursive types
- Functor typeclass derivation using Shapeless
- Functions that are parameterized by a Functor type constructor
- Examples of APIs that consume a functor, with type class constraint

Worked examples

1. Define

Exercises

1. Define type