Chapter 4: Functors. Part 1: Functor laws

How to recognize a functor

Sergei Winitzki

Academy by the Bay

December 28, 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:

1. How to avoid "information loss" in this function?
2. 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:

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

  and

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

  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

What it means to "be able to manipulate values in a container"

- Flip the two curried 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`:

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

- 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:

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

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:

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

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

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

$$\texttt{fmap}^{A,A}\left(\texttt{id}^{A}\right) = \texttt{id}^{F^{A}}$$

$$\texttt{fmap}\left(f \circ g\right) = \texttt{fmap}\left(f\right) \circ \texttt{fmap}\left(g\right)$$

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

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

Examples of functors I

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

- Specific functors will have methods for creating them, reading values out of them, adding / removing items, waiting for items to arrive, etc.
  - Common to all functors is the `map` function
- 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`)

Examples of non-functors that have a `map`:

- `Set[T]` – it works only when `T` has a well-behaved "`==`" operation
- `Map[K, V]` with respect to both `K` and `V`, because it is a `Set` w.r.t. `K`

See test code

Examples of functors II

Polynomial type constructors as functors

1. Short notation: $\text{QueryResult}^A = \text{String} \times \text{Int} \times A$

```
case class QueryResult[A](name: String, time: Int, data: A)
```

2. Short notation: $\text{Vec3}^A = A \times A \times A$

```
case class Vec3[A](x: A, y: A, z: A)
```

3. Short notation: $\text{QueryResult}^A = \text{String} + \text{String} \times \text{Int} \times A$

```
sealed trait QueryResult[A]
case class Error[A](message: String) extends QueryResult[A]
case class Success[A](name: String, time: Int, data: A)
                     extends QueryResult[A]
```

See test code

Examples of functors III: non-functors I

Type constructors that cannot have "`map`"

1. Type constructors that *consume* a value of the parameter type

$$\text{NotContainer}^A = (A \Rightarrow \text{Int}) \times A$$

```
case class NotContainer[A](x: A ⇒ Int, y: A)
```

2. Disjunction type constructors with non-parametric type values

```
sealed trait ServerAction[Res]
case class GetResult[Res](r: Long ⇒ Res) extends ServerAction[Res]
case class StoreId(x: Long, y: String) extends ServerAction[Long]
case class StoreName(name: String) extends ServerAction[String]
```

- The type constructor `ServerAction[Res]` is called a GADT ("generalized algebraic data type")
  - Not sure what the short notation should be for GADT types!

Examples of functors III: non-functors II

They could be functors, except for incorrect implementations of "`fmap`"

We need:

$$\text{fmap}\,(f^{A\Rightarrow B}) : \text{Container}^A \Rightarrow \text{Container}^B$$

1. `fmap(f)` ignores `f` – e.g. always returns `None` for `Option[B]`
2. `fmap(f)` reorders data items in a container:

   $$\text{Container}^A \equiv A\times A; \qquad \text{fmap}^{A,B}\,(f^{A\Rightarrow B})(x^A, y^A) = (f(y), f(x))$$

   or swap some elements in $A\times A\times A$:

   ```
   def fmap[A, B](f: A ⇒ B): Vec3[A] ⇒ Vec3[B] =
     { case Vec3(x, y, z) ⇒ Vec3(f(y), f(x), f(z)) }
   ```
3. Does a special computation if types are equal: if `A` and `B` are the same type, do `fmap[A, A](f) = identity`, otherwise $f(x)$ is applied
4. Does a special computation if type is equal to a specific type, e.g. if `A = B = Int` then do $f(f(x))$ else $f(x)$
5. Does a special computation if f is equal to some f_0, otherwise use $f(x)$

See test code

Recursive polynomial types as functors

- Example: List of pairs defined as a recursive type,

$$LP^A \equiv 1 + A \times A \times LP^A$$

```
sealed trait LP[A]
final case class LPempty[A]() extends LP[A]
final case class LPpair[A](x: A, y: A, tail: LP[A]) extends LP[A]
```

- We can implement `map` as a recursive function:

```
def fmap[A, B](f: A ⇒ B): LP[A] ⇒ LP[B] = {
  case LPempty() ⇒ LPempty[B]()
  case LPpair(x, y, tail) ⇒ LPpair[B](f(x), f(y), map(f)(tail))
}
```

- This is the only way to implement `map` that satisfies the functor laws!

See test code for checking the functor laws

Contrafunctors

- The type constructor $C^A \equiv A \Rightarrow \text{Int}$ is not a functor (impossible to implement `map`), but we can implement `contrafmap`:

$$\text{contrafmap}^{A,B} : (B \Rightarrow A) \Rightarrow C^A \Rightarrow C^B$$

- The contrafunctor laws are analogous to functor laws:

$$\texttt{contrafmap}^{A,A}(\texttt{id}^A) = \texttt{id}^{C^A}$$
$$\texttt{contrafmap}\,(g \circ f) = \texttt{contrafmap}\,(f) \circ \texttt{contrafmap}\,(g)$$

  The "contra-" reverses the arrow between A and B

- The type parameter A is to the left of the function arrow ("consumed")
- "Functors contain data; contrafunctors consume data"

Example of non-contrafunctor:

- The type $\text{NotContainer}^A = (A \Rightarrow \text{Int}) \times A$ is neither a functor nor a contrafunctor

Covariance, contravariance, and subtyping

- Example of subtyping:

```
sealed trait AtMostTwo
final case class Zero() extends AtMostTwo
final case class One(x: Int) extends AtMostTwo
final case class Two(x: Int, y: Int) extends AtMostTwo
```

  - Here `Zero`, `One`, and `Two` are **subtypes** of `AtMostTwo`
- We can pass `Two(10, 20)` to a function that takes an `AtMostTwo`
- This is equivalent to an automatic type conversion `Two ⇒ AtMostTwo`
- A container `C[A]` is **covariant** if `C[Two]` is a subtype of `C[AtMostTwo]`
  - And then a type conversion function `C[Two] ⇒ C[AtMostTwo]` exists
- More generally, when `X` is a subtype of `Y` then we have `X ⇒ Y` and we need `C[X] ⇒ C[Y]`, which is guaranteed if we have a function of type

$$(A \Rightarrow B) \Rightarrow (C^A \Rightarrow C^B)$$

- Scala supports covariance annotations on types: `sealed trait C[+T]`

Functors are covariant, contrafunctors are contravariant

Worked examples

- Decide if a type constructor is a functor, a contrafunctor, or neither
- Implement a `map` or a `contramap` function that satisfies the laws

1. Define case classes for these type constructors, and implement `map`:
   1. $\text{Data}^A \equiv \text{String} + A \times \text{Int} + A \times A \times A$
   2. $\text{Data}^A \equiv 1 + A \times (\text{Int} \times \text{String} + A)$
   3. $\text{Data}^A \equiv (\text{String} \Rightarrow \text{Int} \Rightarrow A) \times A + (\text{Boolean} \Rightarrow \text{Double} \Rightarrow A) \times A$
2. Decide which of these type constructors are functors or contrafunctors, and implement `fmap` or `contrafmap` respectively:
   1. $\text{Data}^A \equiv (A \Rightarrow \text{Int}) + (A \Rightarrow A \Rightarrow \text{String})$
   2. $\text{Data}^{A,B} \equiv (A + B) \times ((A \Rightarrow \text{Int}) \Rightarrow B)$
3. Rewrite this code in the short notation; identify covariant and contravariant type usages; verify that with covariance annotations:

```
sealed trait Coi[A, B]
case class Pa[A, B](b: (A, B), c: B⇒Int)  extends Coi[A, B]
case class Re[A, B](d: A, e: B, c: Int)    extends Coi[A, B]
case class Ci[A, B](f: String⇒A, g: B⇒A) extends Coi[A, B]
```

Exercises

Define case classes for these type constructors, decide if they are covariant or contravariant, and implement `map` or `contramap` as needed:

1. $\text{Data}^A \equiv (1 + A) \times (1 + A) \times \text{String}$
2. $\text{Data}^A \equiv (A \Rightarrow \text{String}) \Rightarrow (A \times (\text{Int} + A))$
3. $\text{Data}^{A,B} \equiv (A \Rightarrow \text{String}) \times ((A + B) \Rightarrow \text{Int})$
4. $\text{Data}^A \equiv (1 + (A \Rightarrow \text{String})) \Rightarrow (1 + (A \Rightarrow \text{Int})) \Rightarrow \text{Int}$
5. $\text{Data}^B \equiv (B + (\text{Int} \Rightarrow B)) \times (B + (\text{String} \Rightarrow B))$
6. Rewrite this code in the short notation; identify covariant and contravariant type usages; verify that with covariance annotations:

```
sealed trait Result[A]
case class P[A](a: A, b: String, c: Int)    extends Result[A]
case class Q[A](d: Int⇒A, e: Int⇒String) extends Result[A]
case class R[A](f: A⇒A, g: A⇒String)      extends Result[A]
```

The structure of functor types I

How to build new functors out of old ones

Main question:

- Is any data type Z^A with A in covariant positions always a functor?
$$Z^{A,R} \equiv ((A \Rightarrow R) \Rightarrow R) \times A + (R \Rightarrow A + \text{Int}) + A \times A \times \text{Int} \times \text{Int}$$
- "Elementary" data types are built from parts:
    - Constant types 1, Int, String, etc.
    - Type parameters A, B, ..., Z, etc.
    - Previously defined type constructors F^A, G^A, etc.
    - Four operations: $F^A + G^A$, $F^A \times G^A$, $F^A \Rightarrow G^A$, F^{G^A} (composition)
    - Each time a type A is moved to the left of $\Rightarrow$, its covariance is reversed
        - ★ So $A \Rightarrow Z$ is contravariant in A, but $(A \Rightarrow Z) \Rightarrow Z$ is again covariant
    - If we exclude the operation $F^A \Rightarrow G^A$, the result is always covariant
        - ★ This yields polynomial type constructors = **polynomial functors**

To answer the question:

- Build `fmap` incrementally as we build up the type constructor
- Verify that the laws hold at every step

The structure of functor types II
The building blocks

- All our functors here will work with respect to the type parameter A
- Building blocks: creating functors from scratch
  - Constant functors $\text{Const}^{C,A} \equiv C$ with $\text{fmap}(f) = id$, and are at the same time contrafunctors with $\text{contrafmap}(f) = id$
  - Identity functor $\text{Id}^A = A$ with $\text{fmap}(f) = f$ (not a contrafunctor!)
- Operations: creating new functors out of previous ones
  - We have already seen how this works in examples
  - In each case, we already have the `fmap` implementations for F^A and G^A, and we assume that their functor laws were already checked
- $F^A + G^A$ – `fmap` is built by pattern-matching and preserving the sides
- $F^A \times G^A$ – `fmap` is built by tupling the two `fmap` results, in order
- $F^A \Rightarrow G^A$ – `fmap` is built by substituting the function argument
  - Here F^A must be a contrafunctor and G^A must be a functor
- F^{G^A} – `fmap` is built by composing the two `fmap`s
  - Check that the functor laws still hold after each operation
- Similar constructions hold for contrafunctors, *mutatis mutandis*