|  |  |
| --- | --- |
| **6: Functors** |  |

A *functor* is a generic abstraction which includes a function to transform or map its properties/data into another functor while preserving its structure. For simplicity you can think of it as a mapping from the elements in some kind of a container to another set of elements (either of the same type or not) of the same kind of container. The function that transforms the source element to the target element is provided by the mapping operation. Functors are usually applied to map instances of generic data structures in a way which preserves their inner structure. For collections like **List** we would implement the mapping function by just applying it to every element.

**6.1 Functor abstraction**

Programming is about abstractions. Some abstractions, such as the *functor*, are well known from a branch of mathematics known as *category theory*, which has proven to be fertile ground for functional programming. Many of these concepts are, however, equally applicable to object oriented programming.

The aim isn't to give a comprehensive introduction to the category theory of functors. Rather, the purpose is to give an opportunity to learn how it translates to object oriented code in Kotlin.

A functor is a container that can be mapped over by a function. Perhaps the best known of all functors is the **List**class with its **map**member function.. The **List**class is a wrapper around none or more values.

The functor is an abstraction that allows us to write generic code that can be used for **List**s, **Option**s, **Either**s, or any other mappable type. This list is not exhaustive and many more functors exist. Some types that are functors are not even containers and consequently the functional community use the term *context*in preference to container to describe the wrapper around a value or values. The Java concurrency interface **Callable**can be defined as functor but is certainly not a container.

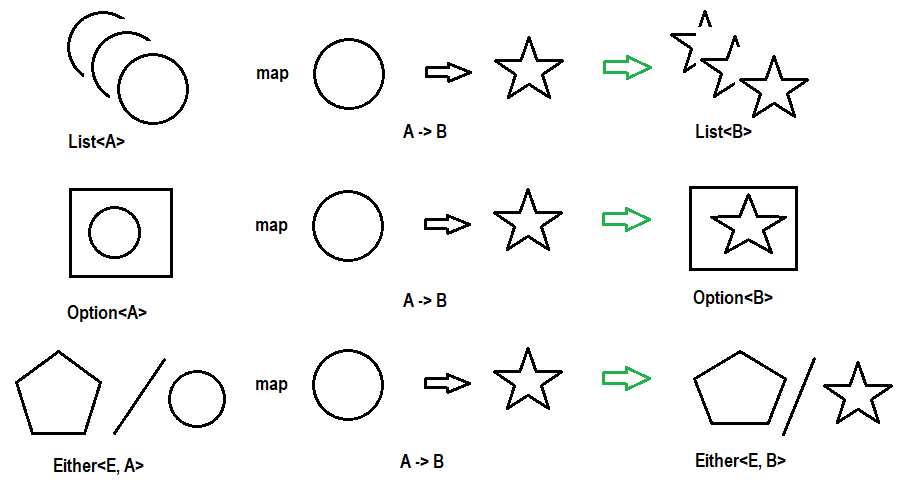
Informally, a functor is anything with a **map** method. There are many types that have this: **List**, **Option** and **Either**, to name a few.

We typically first encounter **map** when iterating over **List**s. However, to understand functors we need to think of the **map** function in another way. Rather than traversing the list, we should think of it as transforming all of the values inside in one go. We specify the function to apply, and **map** ensures it is applied to every item. The values change but the structure of the list remains the same:

ListF.of(1, 2, 3, 4).map { n -> n + 1 } 🡺 ListF.of(2, 3, 4, 5)

Similarly, when we map over an **Option**, we transform the contents but leave the **Some** or **None** context unchanged. The same principle applies to **Either** with its **Left** and **Right** contexts. This general notion of transformation, along with the common pattern of type signatures shown in Figure 6.1, is what connects the behaviour of **map** across different data types.

**Figure 6.1**: *Mapping over List, Option and Either*



**6.2 Kotlin functor**

We are familiar with the **map**function from the **List**class which applies a transformer function to each element in the **List** and delivers a new **List**of transformed values. Because **List**, **Option**, etc all share this behavior, we can document it this way as an extension function, in which **F**is a place-marker for the actual types for which it applies:

fun <A, B> **F**<A>.fmap(f: (A) -> B): **F**<B>

This is called a *functor*and abstracts the capability of mapping over a data type with a regular function. We emphasise the functions source by calling it **fmap**. If **F**is **Option**then we expect the **Option**extension function:

fun <A, B> Option<A>.fmap(f: (A) -> B): Option<B>

as shown above. In the Dogs library the data types **Option**, **Either**, **List**, **Map** and others are functors and have their implementation for function **fmap**.

We might consider introducing the **Option**, **List**, etc. types as functors by first defining a **Functor** interface then subtyping **Option**, **List**, etc from this. A first attempt for the **Functor** interface might look like:

interface Functor<F> {

fun <A, B> F<A>.fmap(f: (A) -> B): F<B>

} // Functor

However, the declaration for function **fmap** reveals the problem. Kotlin, like Java, does not support constructs such as F<A> where both F and A are generic types. The construct F<A> is an example of a *higher-kinded* type not supported in Kotlin. Hence our approach is to augment a type with the **fmap** extension function. It will reduce our capabilities since we could not define a function fun <F> process(f: Functor(F>) then call it with process(myList) or process(myOption) where **List** and **Option** subtype **Functor**.

**6.3 Option functor**

The Dogs library provides the **Option** type which is used as a replacement for working with **null**. The sealed class **Option** has two sub-types: the object **None** is used to represent the absence of a value, while the class **Some** represents the presence of a value wrapped as its property. We could use **Option** for a definition of function **mean** that computes the mean of a list, which is undefined if the list is empty. In Example 01a the functions **none** and **some** are the factory constructors for the **Option** class.

**Example 01a**: *Option type*

package example01a

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.option.Option

import com.adt.kotlin.dogs.data.immutable.option.OptionF.none

import com.adt.kotlin.dogs.data.immutable.option.OptionF.some

import kotlin.test.\*

fun mean(xs: List<Double>): Option<Double> =

if (xs.isEmpty())

none()

else

some(xs.sum() / xs.size())

fun main(args: Array<String>) {

assertEquals(none(), mean(ListF.empty()))

assertEquals(some(2.0), mean(ListF.of(1.0, 2.0, 3.0)))

}

Example 01b demonstrates using the **Option** functor. The infix **dollar** function is an alias for **fmap** with its arguments flipped..

**Example 01b**: *Option functor*

package example01b

import com.adt.kotlin.dogs.data.immutable.option.OptionF.none

import com.adt.kotlin.dogs.data.immutable.option.OptionF.some

import com.adt.kotlin.dogs.data.immutable.option.dollar

import com.adt.kotlin.dogs.data.immutable.option.fmap

import kotlin.test.\*

fun main(args: Array<String>) {

assertEquals(none(), none<Int>().fmap{n: Int -> n \* n})

assertEquals(some(25), some(5).fmap{n: Int -> n \* n})

assertEquals(some(25), some(5).fmap{n -> n \* n}) // types inferred

assertEquals(none(), {n: Int -> n \* n} dollar none())

assertEquals(some(25), {n: Int -> n \* n} dollar some(5))

}

We can think of **fmap** as proceeding with a computation on the assumption that an error has not occurred. In effect, we are deferring the error handling until later. Example 01c demonstrates this while looking up a name from a **Map** and obtaining the **Employee** detail.

**Example 01c**: *Error handling*

package example01c

import com.adt.kotlin.dogs.data.immutable.map.Map

import com.adt.kotlin.dogs.data.immutable.map.MapF

import com.adt.kotlin.dogs.data.immutable.option.Option

import com.adt.kotlin.dogs.data.immutable.option.OptionF.none

import com.adt.kotlin.dogs.data.immutable.option.OptionF.some

import com.adt.kotlin.dogs.data.immutable.option.fmap

import com.adt.kotlin.dogs.data.immutable.option.getOrElse

import kotlin.test.\*

class Employee(val name: String, val salary: Int)

val employees: Map<String, Employee> = MapF.of(

"Ken" to Employee("Ken", 25000),

"John" to Employee("John", 22000),

"Jessie" to Employee("Jessie", 32000)

)

fun main(args: Array<String>) {

fun lookup(name: String): Option<Employee> = employees.lookUpKey(name)

assertEquals(some(25000), lookup("Ken").fmap{emp: Employee -> emp.salary})

assertEquals(none(), lookup("Dawn").fmap{emp: Employee -> emp.salary})

assertEquals(25000, lookup("Ken").fmap{emp: Employee -> emp.salary}.getOrElse(10000))

assertEquals(10000, lookup("Dawn").fmap{emp: Employee -> emp.salary}.getOrElse(10000))

assertEquals(some(25000), lookup("Ken").fmap{emp: Employee -> emp.salary}.filter{sal: Int -> (sal > 24000)})

assertEquals(none(), lookup("Dawn").fmap{emp: Employee -> emp.salary}.filter{sal: Int -> (sal > 24000)})

}

Here, lookup("Ken") returns an Option<Employee> which we transform using **fmap** to pull out the Option<Int> representing the salary. We do not need to explicitly check the result of lookup("Ken"); we simply continue the computation as if no error occurred. Since lookup("Dawn") returns **None** this aborts the computation and **fmap** will not access the salary property. The final two asserts demonstrates how we construct a computation with multiple stages, any of which can fail. These two examples have a complete absence of a conditional checking for **null** as might happen with code not employing class **Option**.

**6.3.1 Lifting**

Rearranging the ordering of the **fmap** parameters and result type we can view it as a function that lifts a regular unary function to the functor context so that it can be applied over values of the implementing data type. Using the compact introductory notation we have:

fun <A, B> lift(f: (A) -> B): (Option<A>) -> Option<B> =

{ oa: Option<A> ->

oa.fmap(f)

}

The single parameter to **lift** transforms a value of type A to a value of type B. Function **lift** returns a function that transforms a value of type Option<A> to a value of type Option<B>. Example 01d is a short illustration.

**Example 01d**: *Lifting*

package example01d

import com.adt.kotlin.dogs.data.immutable.option.Option

import com.adt.kotlin.dogs.data.immutable.option.OptionF

import com.adt.kotlin.dogs.data.immutable.option.OptionF.none

import com.adt.kotlin.dogs.data.immutable.option.OptionF.some

import com.adt.kotlin.dogs.fp.FunctionF.isEven

import kotlin.test.\*

fun main(args: Array<String>) {

val lifted: (Option<Int>) -> Option<Boolean> = OptionF.lift{ n -> isEven(n) }

assertEquals(none(), lifted(none()))

assertEquals(some(false), lifted(some(25)))

}

**6.3.2 Functors for effect management**

The **F** in functor is often referred to as an *effect* or *computational context*. Different effects will abstract away different behaviours with respect to fundamental functions like **fmap**. For instance, **Option**'s effect abstracts away potentially missing values, where **fmap** applies the function only in the **Some** case but otherwise threads the **None** through. Taking this view, we can view functors as the ability to work with a single effect - we can apply a pure function to a single effectful value without needing to leave the effect. We saw this in Example 01c. We will have more to say on *effectful computations* in later chapters.

**6.4 List functor**

Like the **Option** type, the **List** type is a functor. The following example demonstrates using function **fmap** on a **List**. Function **fmap** is defined in terms of the member function **map** from the **List** class itself.

**Example 02a**: *List functor*

package example02a

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.list.fmap

import com.adt.kotlin.dogs.fp.FunctionF.isEven

import kotlin.test.\*

fun main(args: Array<String>) {

assertEquals(ListF.of(2, 4, 6), ListF.of(1, 2, 3).fmap { n -> 2 \* n })

assertEquals(ListF.of(3, 4, 6), ListF.of("Ken", "John", "Jessie").fmap { str -> str.length })

assertEquals(ListF.of(true, false, true), ListF.of(1, 2, 3).fmap { n -> n + 1 }.fmap { n -> isEven(n) })

}

Function **fmap** is accompanied with the infix equivalent **dollar**. This allows us to use the transformation function as the first parameter.

**Example 02b**: *Dollar function*

package example02b

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.list.dollar

import com.adt.kotlin.dogs.fp.FunctionF.isEven

import kotlin.test.\*

fun main(args: Array<String>) {

assertEquals(ListF.of(2, 3, 4, 5), {n: Int -> n + 1} dollar ListF.of(1, 2, 3, 4))

assertEquals(

ListF.of(true, false, true, false),

{n: Int -> isEven(n)} dollar ({n: Int -> n + 1} dollar ListF.of(1, 2, 3, 4))

)

}

The following example defines the function **replicate** that creates a **List** of **n** occurrences of some value **a**. In the assertion we *partially apply* the **replicate** function providing a set value 3 for the parameter **n**. This partial application of **replicate** produces a function with the signature (A) -> List<A> with the list having 3 elements.

**Example 02c**: Partial application

package example02c

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.list.fmap

import com.adt.kotlin.dogs.fp.FunctionF.partial

import kotlin.test.\*

fun <A : Any> replicate(n: Int, a: A): List<A> =

if (n <= 0)

ListF.empty()

else

ListF.cons(a, replicate(n - 1, a))

fun main(args: Array<String>) {

assertEquals(

ListF.of(ListF.of(1, 1, 1), ListF.of(2, 2, 2), ListF.of(3, 3, 3), ListF.of(4, 4, 4)),

ListF.of(1, 2, 3, 4).fmap(partial(3, ::replicate))

)

}

**6.5 Map functor**

If we wish to make a type a functor it must have exactly one type parameter. We documented this with an extension function, in which **F** is a place-marker for the actual types for which it applies:

fun <A, B> **F**<A>.fmap(f: (A) -> B): **F**<B>

If a type takes two type parameters, such as the **Map** type, then we effectively partially apply the **Map** type so that it only takes one type parameter. In effect we fix the type parameter for the key type in Map<**K**, ...> so we can make Map<**K**, ...> a functor. The extension function **fmap** for the **Map** type is then:

fun <K, V, W> Map<K, V>(f: (V) -> W): Map<K, W>

and note that the transformation is applied to the values in the **Map**.

In the following example a **Map** is used to match a country with its capital city. In the first assert we use function **fmap** to capitalize the name of the capital cities. In the second assert we match the name of the country with the number of characters in the name of its capital city.

**Example 03a**: *Map functor*

package example03a

import com.adt.kotlin.dogs.data.immutable.map.MapF

import com.adt.kotlin.dogs.data.immutable.map.fmap

import java.util.\*

import kotlin.test.\*

fun main(args: Array<String>) {

val countries = MapF.of("UK" to "London", "France" to "Paris", "Spain" to "Madrid")

assertEquals(

MapF.of("UK" to "LONDON", "France" to "PARIS", "Spain" to "MADRID"),

countries.fmap{ capital -> capital.uppercase(Locale.getDefault()) }

)

assertEquals(

MapF.of("UK" to 6, "France" to 5, "Spain" to 6),

countries.fmap{ capital -> capital.length }

)

}

In this next simple example we match the names of solar system planets with lists of the diameters of its moons. Function **fmap** is used to pair the planetary names with the average moon diameter.

**Example 03b**: *Solar moons*

package example03b

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.map.MapF

import com.adt.kotlin.dogs.data.immutable.map.fmap

import kotlin.test.\*

fun main(args: Array<String>) {

val moons = MapF.of(

"Earth" to ListF.of(3474.8),

"Mars" to ListF.of(22.2, 12.6),

"Neptune" to ListF.of(60.4, 81.4, 156.0, 174.8, 194.0, 34.8, 420.0,

2705.2, 340.0, 62.0, 44.0, 42.0, 40.0, 60.0)

)

assertEquals(

MapF.of("Earth" to 3474.8, "Mars" to 17.4, "Neptune" to 315.3285714285715),

moons.fmap{diameters ->

diameters.sum() / diameters.size()

}

)

}

**6.6 Functor laws**

The **Map** (and **List**, **Option**, etc.) as a functor requires that we should ensure it satisfies the two functor laws: the *identity law* and the *composition law*. If we follow the patterns for the **SemigroupLaws** and the **MonoidLaws** from the previous chapter we can readily derive the **MapFunctorLaws** which we can show are satisfied.

**Example 04**: *Map functor laws*

package example04

import com.adt.kotlin.dogs.data.immutable.map.Map

import com.adt.kotlin.dogs.data.immutable.map.MapF.lift

import com.adt.kotlin.dogs.data.immutable.map.fmap

import com.adt.kotlin.dogs.fp.FunctionF.compose

import com.adt.kotlin.kwikcheck.generator.CogenF

import com.adt.kotlin.kwikcheck.generator.Gen

import com.adt.kotlin.kwikcheck.generator.GenF

import com.adt.kotlin.kwikcheck.property.PropertyF.forAll

import com.adt.kotlin.kwikcheck.property.PropertyF.prop

import org.junit.Test

import kotlin.test.\*

/\*\*

\* Functors must preserve identity and composition laws. When performing the

\* mapping operation, if the values in the functor are mapped to themselves,

\* the result will be an unmodified functor. If two sequential mapping

\* operations are performed one after the other using two functions, the

\* result should be the same as a single mapping operation with one

\* function that is equivalent to applying the first function to the result

\* of the second.

\*/

object MapFunctorLaws {

/\*\*

\* The first functor law states that if we map the identity function id

\* over a functor, the functor that we get back should be the same as

\* the original functor. It means that fmap id = id. So essentially,

\* this says that if we do fmap id over a functor, it should be the

\* same as just calling id on the functor.

\*/

fun <K, V : Any> identityLaw(map: Map<K, V>): Boolean where K : Any, K : Comparable<K> {

val id: (V) -> V = { v: V -> v }

return (map.fmap(id) == map)

} // identityLaw

/\*\*

\* The second law says that composing two functions and then mapping the

\* resulting function over a functor should be the same as first mapping

\* one function over the functor and then mapping the other one. Formally

\* written, this means that fmap (f . g) = fmap f . fmap g.

\*/

fun <K, V : Any, C : Any, D : Any> compositionLaw(map: Map<K, V>, f: (C) -> D, g: (V) -> C): Boolean where K : Any, K : Comparable<K> {

return (map.fmap(compose(f, g)) == compose(lift<K, C, D>(f), lift<K, V, C>(g))(map))

} // compositionLaw

} // MapFunctorLaws

class Example04 {

@Test

fun mapFunctorOperation() {

val functorLaws = MapFunctorLaws

val stringToInt: Gen<(String) -> Int> = GenF.genF(CogenF.cogenString, GenF.genInt)

val intToBoolean: Gen<(Int) -> Boolean> = GenF.genF(CogenF.cogenInt, GenF.genBoolean)

val property =

forAll(GenF.genMap(GenF.genInt, GenF.genString), stringToInt, intToBoolean){map, strToInt, intToBool ->

val identity: Boolean = functorLaws.identityLaw(map)

val composition: Boolean = functorLaws.compositionLaw(map, intToBool, strToInt)

prop(identity && composition)

}

val checkResult = property.check()

assertTrue(checkResult.isPassed())

}

}

In Chapter 5 we tested the **Semigroup** and **Monoid** laws (respectively Example 01e and Example 02h). In this last example we also need to generate arbitrary functions to test the composition law, The fundamental trick used by KwikCheck is to transform a function of type (A) -> Gen<B> into Gen<(A) -> B>. This is relatively straightforward. The real problem is how to get the function (A) -> Gen<B> in the first place. KwikCheck provides this through the class **Cogen** with the object **CogenF** providing instances for a number of basic types. The object declaration **GenF** supports the function:

fun <A, B> genF(cogenA: Cogen<A>, genB: Gen<B>): Gen<(A) -> B>

where **cogenA** provides the values for the function domain, and **genB** is the generator for the function codomain. In Example 04 **stringToInt** is a generator for a function from a **String** to an **Int**.