Relationships Between Categories

So far we've been looking at individual categories as if they were isolated galaxies – interesting structures inside, no connections between them. In this chapter we will investigate how categories relate to each other, so that we can actually have a category of all categories, a thing that is impossible for sets, for instance.

To build a category of categories, we need to introduce functions between categories, their composition, and identity functions on categories (this one is the easiest).

Let's start with two predecessors of categories – monoids and graphs.

In **Grph**, a category of all graphs, functions consist of mapping nodes and mapping edges, so that sources map to sources and targets map to targets.

In **Mon**, a category of all monoids, functions (that is, elements of monoids) map so that composition and neutral elements are preserved.

Since a category is both a graph and a generalization of monoid, it would be natural to define a function from category **A** to category **B** as a couple of mappings, objects to objects and functions to functions, so that identities and compositions are preserved.

Functor

Definition. Given two categories, **A** and **B**, a *functor* F: **A** → **B** consists of the following components:

- F_0 that maps objects of **A** to objects of **B**;
- F_1 that maps functions of **A** to functions of **B**

Since these two components, F_0 and F_1, act on collections of totally different nature, we can safely (and for simplicity) omit the subscripts and denote both with just one letter F. This is what I plan to do further on. Also, application of functor, according to a tradition of certain programming languages, will be denoted not as $F(X)$ but as $F[X]$; you will see that, although unusual, it does make expressions more readable.

These two mappings should have the following two properties:

- $F[id_X] = id_{F[X]}$
- $F[f \circ g] = F[f] \circ F[g]$

Traditionally, the definition of functor also requires that for a function $f:X \to Y$ we should have $F[f]: F[X] \to F[Y]$. But this actually just follows from our definition above: $F[f] = F[f \circ id_X] = F[f] \circ id_{F[X]}$ — this means that $id_{F[X]}$ can be followed by $F[f]$, which is possible only if the domain of $F[f]$ is $F[X]$.

You may have heard the term “functor” in a variety of confusing contexts. Some programming books use the word to denote any functions that acts on other functions. This is wrong. A mapping from one function to another is called an *operator* in mathematics (e.g. derivative); a mapping from a function to a scalar value is called a *functional* (e.g. an integral, or map/reduce).

The simplest example of a functor is an identity functor. Map everything to itself, and you have a functor, actually, an *endofunctor*, since its domain and codomain are equal.

Before showing examples, let’s wrap it up with showing that we have a category. A category of categories, called **Cat**. Categories are objects of Cat, and functors are functions. Composition of functors is obvious: apply one, then apply another. It is associative just because for any X we have $(F \circ G \circ H)[X] = F[G[H[X]]]$.

Note that **Cat** is so huge that it contains itself as an object. It’s okay, we are not dealing with sets, we do not have set-theoretic axioms, specifically, comprehension axiom that would allow us to build a category of all barbers that don’t shave themselves.

Examples of Functors

Example 1. List[+T]

This is the most popular functor in programming. We will not discuss its features, just look at it from a categorical point of view. In an unspecified programming language (as long as it is Scala) we have its types as objects of the category, and single-parameter functions as functions of the category. Now, given a type T, we can produce a type $List[T]$; so we have a functor defined on objects. How about functions, given $f: T \to U$, what would serve as $List[f]: List[T] \to List[U]$? We don’t have much of a choice; it is List.map function. List.map(f) is the function from $List[T]$ to $List[U]$ that we were looking for.

If we limit ourselves to the category where only subtypings are allowed as functions, we don’t have to introduce map function; the language can provide us with a feature that from $isSubtype:T \to U$ we have $isSubtype:List[T] \to List[U]$. This feature, in Scala, is denoted by having the + sign, like in the title of this example. More on this later.

(Counter-)example 2. Set[T]

In Scala, as well as in Java, and probably in some other languages, Set[T] is a parameterized type impersonating "a set of values of type T". But mapping just types is not enough; we need to map functions, and here we have a problem.
The obvious candidate is the traditional Set[T].map(f:T=>U), which creates a new set out of the values of f on elements of the original set, efficiently building an image. This is not very efficient, since the new set should be materialized right away. If it remained virtual (lazy), like in the case of list, we would have for every call of contains method scan through the whole original set and compare the result of function application with the value provided.
Another solution exists; but it requires some new notions, to be introduced later on.

Example 3. A functor from 1
1 is a category consisting of one object and its identity function. How does a functor from 1 to a category **C**? We select an object in **C**, that's all that we need. For any object x there's a functor x: **1** → **C**.

Example 4. A functor from 2 to Set
If you remember, **2** is a category consisting of two objects, 0 and 1, a function, let's call it 01, from 0 to 1 (and a couple of identity functions). **Set** is a category of sets. To define a functor F from **2** to **Set**, we will need three items: a set F[0], a set F[1] and a function F[01] from F[0] to F[1].

$$F[0] \xrightarrow{F[01]} F[1]$$

For artistic reasons, let's rename F[0] to F_0, F[1] to F_1, and F[01] to F_{01}. We see that every functor from **2** to **Set** is just a function F_{01}: $F_0 \to F_1$ in sets; and every function f:A→B can be thought of as a functor from **2** to **Set** where F_0=A, F_1=B F_{01}=f.

Actually, the fact that we are dealing with category **Set** is irrelevant. The same argument would work for any category **C**.

Example 5. A functor from Set to 2
Now let's try to figure out what we can have here. Two obvious functors are constants: map all objects of **Set** to object 0 of **2**, and all functions to id_0, and similarly, map all objects of **Set** to object 1 of **2**, and all functions to id_1.

Except these two obvious functors, there must be others that cover both 0 and 1. Note that empty set ∅ is a subset of every set, so if a functor F maps it to 1, every other set should map to 1. Meaning, we will have the constant functor we talked about above. Similarly with singletons, which are terminal objects, either they map to 1 or every set maps to 0. We have mappings for ∅ and singletons. Every nonempty set S has an element, and so there is a function from singleton to S; so S should also map to 1. As you see, all nonempty sets should map to 1. We have exactly three functors from **Set** to **2**.

Example 6. Product with an object
Given a category **C** that has products, and an object A in **C**, we can produce a functor that consists of multiplying by A, that is, A×- : **C** → **C**. The functor maps each object X to A×X, and for

a function $f:X \to Y$ it provides $A \times f: A \times X \to A \times Y$; you have probably figured out already how it does it.

Example 7. Set Exponentiation
In the category of sets, given a set A, we can always build, for any set X, a set X^A, which consists of functions from A to X. This is a functor, $-^A$: **Set** → **Set**.

Example 8. Monoids
A monoid can be represented as a category with one object. So, if we have two monoids, a monoidal function from one to another is the same as a functor: we preserve multiplication and identity.

Example 9. Partial Order
A partial order is also a category; and a functor between two such categories is a partial order function that preserves order (it's called *monotone*).

Building New Categories

Now that we know that categories form a category, we can try to figure out how to build unions, products, pullbacks, equalizers in **Cat**, and whether it has initial and terminal objects. Let's walk through all these.

Initial Category

This is a category that has a unique functor to any (other) category. Of course such a category cannot have objects; if it did, we could apply constant functor to it, for each object in a target category, and have more than one of such functors, generally speaking. So the only choice is the empty category, **0**. Feel free to define a functor from **0** to any category **C**.

Terminal Category

For a terminal category each category **C** has a unique functor ending in it. If we take category **1**, for each category **C** there can be exactly one functor **C**→**1**. So we have a terminal object in **Cat**.

Product of Two Categories

This structure can be built similar to what we have in **Set**. Given two categories, **C** and **D**, take as objects of **C×D** all pairs of objects **(**x,y**)** where x is an object of **C** and y is an object of **D**. The

very fact that math allows us to form such pairs is beyond the scope of this text, of course; this can be done internally if we are within a certain domain. As functions, take all pairs of functions **(**f,g**)** where f is a function in **C** and g is a function in **D**. composition is defined component-wise, so
(f1,g1) ◦(f2,g2)=(f1◦f2,g1◦g2).

To see that we have a category, we provide identities $id_{(X,Y)} = (id_X, id_Y)$, and verify that they are neutral re: composition; and that composition is associative.

Does this category satisfy the universal property in **Cat**? It can be proved component-wise that it does.

Sum of Two Categories

Given two categories, **C** and **D**, and assuming that we can build a category consisting of objects of **C** and objects of **D**, and functions from these two categories, we get a new category, **C**+**D**. We already saw examples of this: the sum of n instances of **1**, that is, **1**+**1**+…+**1**, is a discrete category consisting of n objects and only identity functions.

Equalizer? Pullback? Pushout?

Generally speaking, these constructions are not available in Cat, for many reasons, one of them being that equality for objects is not defined in categories, only isomorphisms; so we would have to define everything up to an isomorphism, which gets us into higher-order categories. So let's not count on these.

Reversing the Arrows

Remember that functions in a category have, in general, nothing to do with something that takes an argument and returns a value; they are just formal generalizations. So, given a category C, nothing can stop us from producing another category out of it, by reversing the direction of all functions.

Definition. Given a category **C**, its *opposite*, or *dual*, $\mathbf{C}^{op}$, is a category with the same object and the same functions, but the direction of functions reverted. Note that having any knowledge about function makes no sense here; these are just symbols. Composition is defined in the

opposite direction too, so $(f \circ g)^{op} = g^{op} \circ f^{op}$. The fact that it is a category can be easily proven (you can do it as an exercise).

For some categories opposite is the same as the original category (e.g. **1**, **2**, **3**…); even **Rel**, a category of sets and their binary relationships, is symmetrical relative to this operation; for others it is not trivial at all. For instance, $\mathbf{Set}^{op}$ is the category of *Complete Atomic Boolean Algebras.*

Omitting the exact definition of this, we can intuitively look into it like this: given a set, we have its characteristic function, a predicate that is true only on members of the set. Now, if we have a function $f:X \to Y$ on sets, and for set X we have a predicate p_X, and for set Y we have a predicate p_Y, we can map p_Y to a predicate on X by defining $f(p_Y)(x) = p_Y(f(x))$. This way, for each map between sets we have a map between predicates. We can view sets of such predicates for each given set, as objects of the category; and we, under certain assumptions, may think of such predicates as being the same as the underlying sets.

In programming languages this operation is equivalent to defining sets via its 'contains' predicate. Of course this is not enough; we also need to make sure that every such predicate can be represented as a disjunction of atomic predicates, which correspond to elements of the set.

Contravariant Functor

Frequently the functors we've been discussing so far are called *covariant* functors, due to their actions on functions that map domain to domain and codomain to codomain. Another kind of functor, the one that maps domain of a function to codomain, and codomain to domain, is called *contravariant*.

Strictly speaking, we do not need a special term, because a contravariant functor can be always thought of as a (covariant) functor $\mathbf{C}^{op} \to \mathbf{D}$ (or $\mathbf{C} \to \mathbf{D}^{op}$). But since variance plays an important role in computer science, we have to spend some time discussing it.

Example 1. Map[_,T]
If, in Scala, we fix the second argument of the parameterized type `Map`, we have this feature that for a function $f:X \to Y$, we can produce a function $Map[Y,T] \to Map[X,T]$. This mapping preserves identities and composition; so we have a contravariant functor. In Scala, contravariance is denoted using minus: `Map[-X,+Y]` is the signature of this type.

Variance in Programming Languages

Usually, in languages allowing subtyping (e.g. in Scala) parameterized classes, if they happen to be functors, get their variance marker not because they behave covariantly or contravariantly on arbitrary functions, but only on inclusions (“subtyping”) of types into other types. So that, e.g., if A<:B and X<:Y (this is a notation for the compiler’s ability of subtype one into another), we have Map[B,X]<:Map[A,X] and Map[A,X]<:Map[A,Y]. We are obviously dealing with a category where types are objects, and the relationships of subtyping are functions. This is a partial order, so things are easier than with generic functions.