3

Categories Great and Small

You can get real appreciation for categories by studying a variety of examples. Categories come in all shapes and sizes and often pop up in unexpected places. We'll start with something really simple.

3.1 No Objects

The most trivial category is one with zero objects and, consequently, zero morphisms. It's a very sad category by itself, but it may be important in the context of other categories, for instance, in the category of all categories (yes, there is one). If you think that an empty set makes sense, then why not an empty category?

3.2 Simple Graphs

You can build categories just by connecting objects with arrows. You can imagine starting with any directed graph and making it into a category by simply adding more arrows. First, add an identity arrow at each node. Then, for any two arrows such that the end of one coincides with the beginning of the other (in other words, any two *composable* arrows), add a new arrow to serve as their composition. Every time you add a new arrow, you have to also consider its composition with any other arrow (except for the identity arrows) and itself. You usually end up with infinitely many arrows, but that's okay.

Another way of looking at this process is that you're creating a category, which has an object for every node in the graph, and all possible *chains* of composable graph edges as morphisms. (You may even consider identity morphisms as special cases of chains of length zero.)

Such a category is called a *free category* generated by a given graph. It's an example of a free construction, a process of completing a given structure by extending it with a minimum number of items to satisfy its laws (here, the laws of a category). We'll see more examples of it in the future.

3.3 Orders

And now for something completely different! A category where a morphism is a relation between objects: the relation of being less than or equal. Let's check if it indeed is a category. Do we have identity morphisms? Every object is less than or equal to itself: check! Do we have composition? If $a \leqslant b$ and $b \leqslant c$ then $a \leqslant c$: check! Is composition as-

sociative? Check! A set with a relation like this is called a *preorder*, so a preorder is indeed a category.

You can also have a stronger relation, that satisfies an additional condition that, if $a \leqslant b$ and $b \leqslant a$ then a must be the same as b. That's called a *partial order*.

Finally, you can impose the condition that any two objects are in a relation with each other, one way or another; and that gives you a *linear order* or *total order*.

Let's characterize these ordered sets as categories. A preorder is a category where there is at most one morphism going from any object a to any object b. Another name for such a category is "thin." A preorder is a thin category.

A set of morphisms from object a to object b in a category $\mathbf{C}$ is called a *hom-set* and is written as $\mathbf{C}(a, b)$ (or, sometimes, $\mathbf{Hom}_{\mathbf{C}}(a, b)$). So every hom-set in a preorder is either empty or a singleton. That includes the hom-set $\mathbf{C}(a, a)$, the set of morphisms from a to a, which must be a singleton, containing only the identity, in any preorder. You may, however, have cycles in a preorder. Cycles are forbidden in a partial order.

It's very important to be able to recognize preorders, partial orders, and total orders because of sorting. Sorting algorithms, such as quicksort, bubble sort, merge sort, etc., can only work correctly on total orders. Partial orders can be sorted using topological sort.

3.4 Monoid as Set

Monoid is an embarrassingly simple but amazingly powerful concept. It's the concept behind basic arithmetics: Both addition and multiplication form a monoid. Monoids are ubiquitous in programming. They

show up as strings, lists, foldable data structures, futures in concurrent programming, events in functional reactive programming, and so on.

Traditionally, a monoid is defined as a set with a binary operation. All that's required from this operation is that it's associative, and that there is one special element that behaves like a unit with respect to it.

For instance, natural numbers with zero form a monoid under addition. Associativity means that:

$$(a + b) + c = a + (b + c)$$

(In other words, we can skip parentheses when adding numbers.)

The neutral element is zero, because:

$$0 + a = a$$

and

$$a + 0 = a$$

The second equation is redundant, because addition is commutative ($a + b = b + a$), but commutativity is not part of the definition of a monoid. For instance, string concatenation is not commutative and yet it forms a monoid. The neutral element for string concatenation, by the way, is an empty string, which can be attached to either side of a string without changing it.

In Haskell we can define a type class for monoids — a type for which there is a neutral element called **`mempty`** and a binary operation called **`mappend`**:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m
```

The type signature for a two-argument function, `m -> m -> m`, might look strange at first, but it will make perfect sense after we talk about currying. You may interpret a signature with multiple arrows in two basic ways: as a function of multiple arguments, with the rightmost type being the return type; or as a function of one argument (the leftmost one), returning a function. The latter interpretation may be emphasized by adding parentheses (which are redundant, because the arrow is right-associative), as in: `m -> (m -> m)`. We'll come back to this interpretation in a moment.

Notice that, in Haskell, there is no way to express the monoidal properties of `mempty` and `mappend` (i.e., the fact that `mempty` is neutral and that `mappend` is associative). It's the responsibility of the programmer to make sure they are satisfied.

Haskell classes are not as intrusive as C++ classes. When you're defining a new type, you don't have to specify its class up front. You are free to procrastinate and declare a given type to be an instance of some class much later. As an example, let's declare `String` to be a monoid by providing the implementation of `mempty` and `mappend` (this is, in fact, done for you in the standard Prelude):

```
instance Monoid String where
    mempty = ""
    mappend = (++)
```

Here, we have reused the list concatenation operator `(++)`, because a `String` is just a list of characters.

A word about Haskell syntax: Any infix operator can be turned into a two-argument function by surrounding it with parentheses. Given two strings, you can concatenate them by inserting `++` between them:

```
"Hello " ++ "world!"
```

or by passing them as two arguments to the parenthesized **(++)**:

```
(++) "Hello " "world!"
```

Notice that arguments to a function are not separated by commas or surrounded by parentheses. (This is probably the hardest thing to get used to when learning Haskell.)

It's worth emphasizing that Haskell lets you express equality of functions, as in:

```
mappend = (++)
```

Conceptually, this is different than expressing the equality of values produced by functions, as in:

```
mappend s1 s2 = (++) s1 s2
```

The former translates into equality of morphisms in the category **Hask** (or **Set**, if we ignore bottoms, which is the name for never-ending calculations). Such equations are not only more succinct, but can often be generalized to other categories. The latter is called *extensional* equality, and states the fact that for any two input strings, the outputs of **mappend** and **(++)** are the same. Since the values of arguments are sometimes called *points* (as in: the value of f at point x), this is called pointwise equality. Function equality without specifying the arguments is described as *point-free*. (Incidentally, point-free equations often involve composition of functions, which is symbolized by a point, so this might be a little confusing to the beginner.)

The closest one can get to declaring a monoid in C++ would be to use the C++20 standard's concept feature.

```
template<class T>
struct mempty;

template<class T>
T mappend(T, T) = delete;

template<class M>
concept Monoid = requires (M m) {
    { mempty<M>::value() } -> std::same_as<M>;
    { mappend(m, m) } -> std::same_as<M>;
};
```

The first definition is a structure meant to hold the neutral element for each specialization.

The keyword **delete** means that there is no default value defined: It will have to be specified on a case-by-case basis. Similarly, there is no default for **mappend**.

The concept **Monoid** tests whether there exist appropriate definitions of **mempty** and **mappend** for a given type **M**.

An instantiation of the Monoid concept can be accomplished by providing appropriate specializations and overloads:

```
template<>
struct mempty<std::string> {
    static std::string value() { return ""; }
};

template<>
std::string mappend(std::string s1, std::string s2) {
    return s1 + s2;
}
```

3.5 Monoid as Category

That was the "familiar" definition of the monoid in terms of elements of a set. But as you know, in category theory we try to get away from sets and their elements, and instead talk about objects and morphisms. So let's change our perspective a bit and think of the application of the binary operator as "moving" or "shifting" things around the set.

For instance, there is the operation of adding 5 to every natural number. It maps 0 to 5, 1 to 6, 2 to 7, and so on. That's a function defined on the set of natural numbers. That's good: we have a function and a set. In general, for any number n there is a function of adding n — the "adder" of n.

How do adders compose? The composition of the function that adds 5 with the function that adds 7 is a function that adds 12. So the composition of adders can be made equivalent to the rules of addition. That's good too: we can replace addition with function composition.

But wait, there's more: There is also the adder for the neutral element, zero. Adding zero doesn't move things around, so it's the identity function in the set of natural numbers.

Instead of giving you the traditional rules of addition, I could as well give you the rules of composing adders, without any loss of information. Notice that the composition of adders is associative, because the composition of functions is associative; and we have the zero adder corresponding to the identity function.

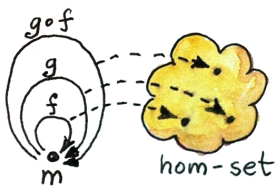
An astute reader might have noticed that the mapping from integers to adders follows from the second interpretation of the type signature of `mappend` as `m -> (m -> m)`. It tells us that `mappend` maps an element of a monoid set to a function acting on that set.

Now I want you to forget that you are dealing with the set of natural numbers and just think of it as a single object, a blob with a bunch of morphisms — the adders. A monoid is a single object category. In fact the name monoid comes from Greek *mono*, which means single. Every monoid can be described as a single object category with a set of morphisms that follow appropriate rules of composition.

String concatenation is an interesting case, because we have a choice of defining right appenders and left appenders (or *prependers*, if you will). The composition tables of the two models are a mirror reverse of each other. You can easily convince yourself that appending "bar" after "foo" corresponds to prepending "foo" after prepending "bar".

You might ask the question whether every categorical monoid — a one-object category — defines a unique set-with-binary-operator monoid. It turns out that we can always extract a set from a single-object category. This set is the set of morphisms — the adders in our example. In other words, we have the hom-set $\mathbf{M}(m, m)$ of the single object m in the category $\mathbf{M}$. We can easily define a binary operator in this set: The monoidal product of two set-elements is the element corresponding to

the composition of the corresponding morphisms. If you give me two elements of $\mathbf{M}(m, m)$ corresponding to f and g, their product will correspond to the composition $f \circ g$. The composition always exists, because the source and the target for these morphisms are the same object. And it's associative by the rules of category. The identity morphism is the neutral element of this product. So we can always recover a set monoid from a category monoid. For all intents and purposes they are one and the same.



Monoid hom-set seen as morphisms and as points in a set.

There is just one little nit for mathematicians to pick: morphisms don't have to form a set. In the world of categories there are things larger than sets. A category in which morphisms between any two objects form a set is called locally small. As promised, I will be mostly ignoring such subtleties, but I thought I should mention them for the record.

A lot of interesting phenomena in category theory have their root in the fact that elements of a hom-set can be seen both as morphisms, which follow the rules of composition, and as points in a set. Here, composition of morphisms in $\mathbf{M}$ translates into monoidal product in the set $\mathbf{M}(m, m)$.

3.6 Challenges

1. Generate a free category from:

   (a) A graph with one node and no edges
   (b) A graph with one node and one (directed) edge (hint: this edge can be composed with itself)
   (c) A graph with two nodes and a single arrow between them
   (d) A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c … z.

2. What kind of order is this?

   (a) A set of sets with the inclusion relation: A is included in B if every element of A is also an element of B.
   (b) C++ types with the following subtyping relation: `T1` is a subtype of `T2` if a pointer to `T1` can be passed to a function that expects a pointer to `T2` without triggering a compilation error.

3. Considering that `Bool` is a set of two values `True` and `False`, show that it forms two (set-theoretical) monoids with respect to, respectively, operator `&&` (AND) and `||` (OR).
4. Represent the `Bool` monoid with the AND operator as a category: List the morphisms and their rules of composition.
5. Represent addition modulo 3 as a monoid category.