Category Theory & Functional Data Abstraction

Brandon Shapiro
Math 100b

1. Introduction

Throughout mathematics, particularly algebra, there are numerous commonalities between the studies of various objects and ideas. For instance, the topics of sets, groups, rings, modules, and fields, among others, all expand on just a few basic ideas. In each case there is a notion of a substructure (e.g. subset, subgroup), as well as functions between objects (e.g. homomorphisms). Category theory can be thought of as a framework for analyzing the relationships between some of these ideas. It allows expansive classes of mathematical objects to be considered in the same light, through basic principles general enough to be applied to a broad selection of mathematical objects across multiple fields of study. From category theory comes a notion of "natural transformations," giving precise meaning to many ideas otherwise only clear through intuition. However, here the focus will be on how certain examples of category theory relate to interesting forms of computationally representable data. These relationships between data types and mathematical objects allow for incredibly powerful computational techniques. In fact, from one categorical concept in particular arises an entire programming idiom incredibly useful for working within specialized forms of data. While it would be nearly impossible to properly describe the full context of these connections in a single paper, category theory provides an abstract mathematical lens through which these computational ideas can be expressed.

2. Categories

A category is a collection of objects with arrows between them (the arrows are often called morphisms), with the following properties:

- Let $Hom_{\mathbf{C}}(X,Y)$ be the set of all morphisms in category $\mathbf{C}$ from object X to object Y (written as $Hom(X,Y)$ when the category is clear from context). Similarly to functions, morphisms can be composed. That is, if $\mathbf{C}$ is a category and X, Y, and Z are objects in $\mathbf{C}$ with $f \in Hom_{\mathbf{C}}(X,Y)$ and $g \in Hom_{\mathbf{C}}(Y,Z)$, then there exists a morphism $f \circ g$, or simply fg, in $Hom_{\mathbf{C}}(X,Z)$.
- The sets of morphisms between two distinct pairs of objects are disjoint.
- Morphism composition is associative.
- For every object X in $\mathbf{C}$, there is an identity morphism $1_X \in Hom_{\mathbf{C}}(X,X)$. For any $f \in Hom_{\mathbf{C}}(X,Y)$ and $g \in Hom_{\mathbf{C}}(Y,X)$, $f \circ 1_X = f$ and $1_X \circ g = g$.

Many examples of categories come from familiar algebraic objects.

1. Let **Set** denote the category of all sets, with functions between sets as the morphisms. That is, for sets A and B, $Hom(A, B)$ is the set of all functions from A to B, and composition of morphisms is simply composition of functions (which is associative). Every set A has an identity morphism 1_A in the identity function from A to A.

2. All groups also form a category, **Grp**, with group homomorphisms as its morphisms. Just like functions in **Set**, group homomorphisms are composable and every group has an identity homomorphism to itself.

3. Similarly, categories of rings (**Ring**) and R-modules (R-**mod** for some ring R) can be formed with ring and module homomorphisms as morphisms (respectively).

4. A subcategory of category **C** is a category with all of its objects and morphisms contained in **C**. Finite sets and the functions between them form a subcategory of **Set**, and abelian groups are a subcategory of **Grp**. Fields form a subcategory of the category of commutative rings, which is itself a subcategory of **Ring**.

3. Functors

A functor can be thought of as a structure preserving map between categories. More precisely, given categories **C** and **D**, a covariant functor is a function $\mathcal{F} : \mathbf{C} \to \mathbf{D}$ that sends the objects of **C** to objects in **D**, and sends the morphisms in **C** to morphisms in **D** such that if $f \in Hom_{\mathbf{C}}(X, Y)$, $\mathcal{F}(f) \in Hom_{\mathbf{D}}(\mathcal{F}(X), \mathcal{F}(Y))$. A functor must also have the following properties:

- $\mathcal{F}(1_X) = 1_{\mathcal{F}(X)}$
- $\mathcal{F}(f \circ g) = \mathcal{F}(f) \circ \mathcal{F}(g)$

A contravariant functor is defined similarly, but while covariant functors preserve the structure of arrows in **C**, contravariant functors reverse those arrows. Specifically, if $f \in Hom_{\mathbf{C}}(X, Y)$, then $\mathcal{F}(f) \in Hom_{\mathbf{D}}(\mathcal{F}(Y), \mathcal{F}(X))$, and $\mathcal{F}(f \circ g) = \mathcal{F}(g) \circ \mathcal{F}(f)$.

Some basic functors come from the previously discussed categories of algebraic objects. While it is not explicitly done here, verifying that these examples satisfy the functor laws is trivial.

1. The identity functor from **C** to **C** sends every object and morphism in **C** to itself.

2. Let $\mathcal{F}$ be a map from **Grp** to **Set** sending groups (which are of course sets) and homomorphisms (which are functions) in **Grp** to themselves in **Set**. $\mathcal{F}$ is a functor from **Grp** to **Set** called the 'forgetful functor', as it sends groups and homomorphisms, which have structure, to sets and functions without that structure (so some information is lost, or forgotten). Similarly, forgetful functors exist from **Ring** and R-**mod** to **Grp** and to **Set**.

A functor from a category to itself is called an endofunctor. An obvious example of an endofunctor is the identity functor. The following sections will introduce some less trivial endofunctors of **Set** that arise very frequently in computer science.

4. Data Types & Maybe

In computer programming languages, a data type is a set of elements that can be represented by a computer (finitely in binary) in the same way. Two of the most common data types are integers ($\mathbb{Z}$) and real numbers ($\mathbb{R}$), which have straightforward binary representations. In real-world computing, memory is finite, so only numbers up to a certain size can be handled (and in the case of the reals up to a certain decimal precision), but these limitations are usually not important in an abstract discussion of data types. Mathematically, a data type can be treated just as a set (with the specific sets considered often chosen to correspond with commonly used data types in computer science). Categorical results about sets in many cases translate to powerful ways of reasoning about computational data types.

Consider then the category **Set**, with all sets as objects and functions between those sets as morphisms. Define the function

$$\mathcal{M}aybe : \mathbf{Set} \to \mathbf{Set}$$
$$\mathcal{M}aybe(A) = A \cup \{Nothing\}$$

Here $Nothing$ is simply a single additional set element. To better understand why the name $\mathcal{M}aybe$ is used, consider $\mathcal{M}aybe(\mathbb{Z})$. If $x \in \mathcal{M}aybe(\mathbb{Z})$, then x is either an integer or $Nothing$, so x is maybe an integer. $\mathcal{M}aybe$ is particularly useful for defining ‘safe’ versions of partial functions, which are functions that have no image for certain elements of the domain set. For example, the reciprocal function is from the real numbers to the real numbers, but the reciprocal of zero is undefined. This function can be expressed as a ‘total’ function (one that is not partial) using the maybe real numbers as follows.

$$f : \mathbb{R} \to \mathcal{M}aybe(\mathbb{R})$$
$$f(0) = Nothing$$
$$f(x) = 1/x \ (x \neq 0)$$

Instead of returning a real number for only a subset of the domain, this reciprocal function always returns a maybe real number. If the input is nonzero, the output is real; otherwise it returns $Nothing$. Redefining partial functions in this manner is very useful in computing, as instead of generating an error when the function is called on an inconvenient input (such as 0 in the case of reciprocal), the function still returns a value, $Nothing$, which can then be handled by a programmer however he or she chooses. More complicated computations involving $\mathcal{M}aybe$ types generally involve multiple functions and interaction between different $\mathcal{M}aybe$ sets, which motivates a categorical perspective on $\mathcal{M}aybe$.

The $\mathcal{M}aybe$ function can in fact define a functor (more specifically an endofunctor) from **Set** to **Set** that maps every set A to the set $\mathcal{M}aybe(A)$, given a suitable mapping for the morphisms of **Set** (functions). In order to differentiate between the functorial mapping of functions as morphisms in the category **Set** and the functorial mapping of functions as members of some set of functions (which would of course be an object in **Set**), the function applying the functor to morphisms shall be called $\mathcal{M}map$ instead of $\mathcal{M}aybe$. $\mathcal{M}map$ needs to send functions from one set (A) to another (B) to functions from $\mathcal{M}aybe(A)$ to $\mathcal{M}aybe(B)$ such that the (covariant) functor laws are satisfied. It can be shown that, for unspecified sets A and B, the only such definition of $\mathcal{M}map$ is as follows.

$$\mathcal{M}map : Hom(A,B) \rightarrow Hom(\mathcal{M}aybe(A), \mathcal{M}aybe(B))$$
$$\mathcal{M}map(f)(Nothing) = Nothing$$
$$\mathcal{M}map(f)(x) = f(x)\ (x \neq Nothing)$$

$\mathcal{M}map$ essentially extends a function from A to B to send $Nothing$ to $Nothing$, resulting in a function from $\mathcal{M}aybe(A)$ to $\mathcal{M}aybe(B)$. It should be clear then that $\mathcal{M}map(1_A) = 1_{\mathcal{M}aybe(A)}$, as $\mathcal{M}map(1_A)$ is by definition the identity when acting on elements of A, and the only other element of $\mathcal{M}aybe(A)$ is $Nothing$, which is sent to itself. It is easy to show that the composition law for functors is also satisfied. $\mathcal{M}aybe$ and $\mathcal{M}map$ together then define a covariant endofunctor on **Set**.

5. List

Another very useful endofunctor on **Set** sends each set to the set of all possible 'lists' of elements in the set, defined recursively as follows.

$$\mathcal{L}ist : \mathbf{Set} \rightarrow \mathbf{Set}$$
$$\mathcal{L}ist(A) = \{()\} \cup \{(x, xlist) | x \in A, xlist \in \mathcal{L}ist(A)\}$$

Elements of $\mathcal{L}ist(A)$ are called lists (of elements in A), and the element () is called the empty list. Every element of $\mathcal{L}ist(A)$ is either the empty list or a pair of an element of A and a list of elements in A (which may or may not be empty). Below are some examples of lists.

$$(1,(2,(3,(4,()))))\in \mathcal{L}ist(\mathbb{Z})$$
$$(1/2,(Nothing,(1/4,()))) \in \mathcal{L}ist(\mathcal{M}aybe(\mathbb{Q}))$$
$$(1,2,3,4) \in \mathcal{L}ist(\mathbb{Z}) \text{ (This is a more convenient notation for the first list)}$$

Lists are fundamental structures in computer science, used for everything from computational puzzle solving to automatically translating written code into machine language. Functions that filter out certain elements in a list, reduce a list of elements to a single one (e.g. the cumulative sum or product of a list of numbers), or merge together multiple lists are among the numerous means of using lists for a variety of powerful computations. One of the most useful functions on lists is that used to define the $\mathcal{L}ist$ functor.

Just as with the $\mathcal{M}aybe$ functor, a functorial definition of $\mathcal{L}ist$ requires a means of sending functions between sets to functions between the corresponding sets of lists. This can be accomplished using the $\mathcal{L}map$ function (named distinctly from $\mathcal{L}ist$ to distinguish from dealing with lists of functions), defined recursively as follows.

$$\mathcal{L}map : Hom(A,B) \to Hom(\mathcal{L}ist(A), \mathcal{L}ist(B))$$
$$\mathcal{L}map(f)(()) = ()$$
$$\mathcal{L}map(f)((x, xlist)) = (f(x), \mathcal{L}map(f)(xlist))$$

$$\text{For } f(x) = x^2,\ \mathcal{L}map(f)((1,2,3,4)) = (1,4,9,16)$$

$\mathcal{L}map$ sends functions from set A to set B to functions from lists of elements in A to lists in elements of B by applying the input function to each successive element in a list. The computational power of $\mathcal{L}map$ is not difficult to see, as it allows for easy generation of interesting lists (such as the first 4 squares as shown above) from simpler ones. In programming languages that can handle infinite lists (such as Haskell), $\mathcal{L}map$ can be used to generate lists of all squares, all powers of 2, all digits of π, and more (though such a list could not be fully displayed in finite time).

$\mathcal{L}map(id)$ is the identity function on lists, as each element of the list is sent to itself, so the first functor law is satisfied. Similarly, $\mathcal{L}map(f \circ g) = \mathcal{L}map(f) \circ \mathcal{L}map(g)$, as composing $\mathcal{L}map(f)$ with $\mathcal{L}map(g)$ sends each element x in the list to $f(g(x))$, which is exactly what $\mathcal{L}map(f \circ g)$ does. $\mathcal{L}ist$ (along with $\mathcal{L}map$) is therefore an endofunctor on the category **Set**.

6. Applicative Functors

In each of the two previous sections, the implications of applying an endofunctor of **Set** to a set of functions was carefully avoided. However, the image of a set of a set of functions under an endofunctor such as $\mathcal{L}ist$ or $\mathcal{M}aybe$ can have very rich interactions with the images of the domain and codomain sets of those functions. As has been the theme, these interactions have powerful uses in computation.

For the purpose of simplicity, only the case of endofunctors of **Set** shall be discussed. An applicative functor is a functor for which the following function can be defined.

$$\mathcal{F}splat : \mathcal{F}(Hom(A,B)) \to Hom(\mathcal{F}(A), \mathcal{F}(B))$$

$\mathcal{F}splat$ takes elements of image under $\mathcal{F}$ of the set of functions from A to B to functions from the sets $\mathcal{F}(A)$ to $\mathcal{F}(B)$ (the name $\mathcal{F}splat$ is derived from what the analogous function is referred to in the programming language Haskell, which captures the spirit of the function rather well). Additionally, an applicative functor must satisfy certain properties involving identity, composition, and application, but these properties involve mappings from elements of $Hom(A,B)$ to $\mathcal{F}(Hom(A,B))$, which would be unnecessary and tedious to go into. It shall be noted in the examples that follow how $\mathcal{F}splat$ can generally be defined in any number of ways, and taken on faith (as well as justified intuition) that the definition(s) chosen to best preserve information do in fact obey the rules.

Consider the set $\mathcal{M}aybe(Hom(A,B)) = Hom(A,B) \cup \{Nothing\}$. An elements of this set is *maybe* a function from set A to set B, in that it is either a function from A to B or $Nothing$.

Defining $\mathcal{F}splat$ for $\mathcal{M}aybe$ (which will be called $\mathcal{M}splat$) requires deciding how to apply a maybe function to a maybe argument. As with $\mathcal{M}map$, there is only one such valid (under the unstated applicative functor laws) definition. For $f \neq Nothing$ & $x \neq Nothing$:

$$\mathcal{M}splat : \mathcal{M}aybe(Hom(A, B)) \to Hom(\mathcal{M}aybe(A), \mathcal{M}aybe(B))$$
$$\mathcal{M}splat(Nothing)(\_) = \mathcal{M}splat(\_)(Nothing) = Nothing$$
$$\text{where } \_ \text{ indicates any argument}$$
$$\mathcal{M}splat(f)(x) = f(x)$$

In other words, $\mathcal{M}splat(f)(x)$ is just $f(x)$ if both f and x are not $Nothing$, otherwise it is $Nothing$. $\mathcal{M}splat$ can alternatively be thought of as a function of two arguments, a maybe function f and an element x of $\mathcal{M}aybe(A)$, that returns an element of $\mathcal{M}aybe(B)$. If either argument is $Nothing$, $\mathcal{M}splat$ returns $Nothing$, otherwise it returns f applied to x, so $\mathcal{M}splat$ acts like a $Nothing$ preserving version of function application. $\mathcal{M}splat$ can be used to define multivariable functions on $\mathcal{M}aybe$ sets. Consider the following function.

$$addTo : \mathbb{Z} \to Hom(\mathbb{Z}, \mathbb{Z})$$
$$addTo(x) = f \text{ where } f(y) = x + y$$

Recall $\mathcal{M}map$ from section 4, which sends functions between sets to functions between the respective $\mathcal{M}aybe$ sets. Applying it to $addTo$ gives the following signature.

$$\mathcal{M}map(addTo) : \mathcal{M}aybe(\mathbb{Z}) \to \mathcal{M}aybe(Hom(\mathbb{Z}, \mathbb{Z}))$$

$\mathcal{M}map(addTo)(x)$ is then a maybe function from $\mathbb{Z}$ to $\mathbb{Z}$, equal to $Nothing$ if x is $Nothing$, otherwise equal to $addTo(x)$. Applying $\mathcal{M}splat$ to this maybe function gives

$$\mathcal{M}splat(\mathcal{M}map(addTo)(x)) : \mathcal{M}aybe(\mathbb{Z}) \to \mathcal{M}aybe(\mathbb{Z})$$
$$\mathcal{M}splat(\mathcal{M}map(addTo)(x))(y) = x + y \text{ if } x \neq Nothing \text{ \& } y \neq Nothing$$
$$\text{Otherwise, } \mathcal{M}splat(\mathcal{M}map(addTo)(x))(y) = Nothing$$

This is a $Nothing$ preserving addition of maybe integers! Traditional mathematical function notation makes it look very clunky, but infix operators and a slightly less familiar paradigm for thinking about multivariable functions allow the same function as defined above to be expressed in the functional programming language Haskell as follows.

```
addMaybe :: Maybe Integer -> Maybe Integer -> Maybe Integer
addMaybe x y = (+) <$> x <*> y
```

While it would take far too long to explain the syntax above in its entirety, the key points are that addMaybe sends two maybe integers to another maybe integer using either addition or propogation of $Nothing$, <$> is an infix version of $\mathcal{M}map$, and <*> is an infix version of $\mathcal{M}splat$. Both $\mathcal{M}map$ and $\mathcal{M}splat$ are used here in the two argument sense described above for $\mathcal{M}splat$

(however in Haskell there is no distinction). It is noteworthy that the '(+)' could easily be replaced with any other binary operation on integers to make a *Nothing* preserving version of it, just as the process of defining the function mathematically above could be replicated for any binary function.

$\mathcal{L}ist$ is also an applicative functor. Defining $\mathcal{F}splat$ for lists ($\mathcal{L}splat$) requires deciding how to apply a list of functions to a list of arguments. One could easily find many such functions, but consider two in particular (which conveniently satisfy the mysterious applicative functor laws).

$$\mathcal{L}splat_1 : \mathcal{L}ist(Hom(A,B)) \to Hom(\mathcal{L}ist(A), \mathcal{L}ist(B))$$
$$\mathcal{L}splat_1(())(\_) = \mathcal{L}splat_1(\_)(()) = ()$$
$$\mathcal{L}splat_1((f, flist))((x, xlist)) = (f(x), \mathcal{L}splat_1(flist)(xlist))$$

$\mathcal{L}splat_1$ of a list of functions applied to a list of arguments essentially zips the two together, applying each function to its corresponding element in the argument list, until either list reaches its end (the empty list), at which point it cuts off.

$$\mathcal{L}splat_1((sin, cos, tan))((\pi/2, \pi/3, 0, 2, 5, 3)) = (1, 1/2, 0)$$
$$\mathcal{L}splat_1((x^2, x^3, x^4, x^5))((4, 3, 2)) = (16, 27, 16)$$

While this definition can be useful for applying different functions to different entries in a list all at once, there is nothing sacred about only applying the first function to the first argument and so on. This definition also results in functions or elements getting left out when the lists of functions and arguments are not the same length. While it would be difficult to write out concisely, a different function $\mathcal{L}splat_2$, with the same signature, can be defined so that $\mathcal{L}splat_2(flist)(xlist)$ takes the Cartesian product of the functions of $flist$ with the arguments of $xlist$, applying each function to each argument and concatenating the results into a list. The resulting list contains every possible value obtained by applying some function in $flist$ to some argument in $xlist$, so no information at all is lost. This definition is generally considered more powerful than the first. $\mathcal{L}ist$ can also support the 'next level' of categorical structure used in programming with endofunctors of **Set**, called monads, but only by using an analog of $\mathcal{L}splat_2$ ($\mathcal{L}splat_1$ does not make $\mathcal{L}ist$ a monad).

7. Conclusion

Category theory is an exciting lens through which to consider broad topics in mathematics, as the patterns and common ideas between different objects and even different fields of study can be rigorously defined and examined. It also provides a mathematical perspective on advanced programming methods used in typed functional programming languages (such as Haskell) to do incredible things with functions and data structures by assigning categorical meaning to types of computationally representable data (by considering them as objects in **Set**). This intersection of mathematics and computer science allows both subjects to learn from each other's ideas and techniques in exciting ways.

8. Sources

Abstract Algebra by Dummit and Foote
http://en.wikibooks.org/wiki/Haskell/Category_theory
Lectures by and conversations with Kenny Foner