Kittenlab lecture notes

Owen Lynch

11/26/22

Table of contents

Introduction

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Lecture 1: Formal Math

Introduction

These lectures are meant to go from 0 to category theory as efficiently as possible. This means that we are going to optimize for being precise, for being clear, and for opening up new possibilities for you. We are not going to optimize for being entertaining or engaging; this is not a "monads as burritos" blog post or a popsci article about category theory. The answer to "why do we care about this" is often going to be "because it's important later on", and you are just going to have to trust me on that.

As we are going *from 0*, in this first lecture I plan to get you all aquainted with what it even means to "do math" at the level that category theory lives. This is a different kind of math than what you might have learned about in lower level math courses, and so sorting out from the beginning the mindset that you should have for the rest of the lectures is the most efficient use of this time. Generally, nobody tells you about this distinction and you have to work it out painfully over years of getting bad grades on university math homeworks; we don't have time for that.

However, we will have code examples for you to play with, because most of you are programmers and thus making a connection between math and code should speed the learning process.

Finally, everyone at some point in this lecture will be frustrated by how pedantic I'm being. Sorry. I'm erring on the side of pedantry because there's a "price is right" situation here: if I go too slow, we waste a bit of time, but if I go too fast we waste all of the time.

Definitions, Propositions, and Examples

Pure math consists of a series of *definitions*, *propositions*, and *examples*. In this document, we typeset these like

A **definition** introduces a new word, and I will always put that new word in bold. In normal speech, words have meanings given by context, by association, and only sometimes by explicit definition. In math, it is not like that. Every technical word has a single definition. That definition may not be written down explicitly; it may be agreed implicitly between mathematicians based on shared experience. However, in theory there is always a precise definition for every mathematical concept. It is expected that all participants in a mathematical conversation could be locked up in a cell, given paper, a pencil and a great deal of time, and then write down a fully rigorous formulation of each of the words they are using. Moreover, each of these formulations for each of the mathematicians might not be exactly the same, but they should be able to be proven equivalent.

Until you learn mathematical logic, which we will not cover here, this expectation cannot be realized because you don't know a precise definition of "fully rigorous". The level of rigor that will suffice for now is that you should be able to expand every definition to a level where it can be explained to a smart, patient human who knows no category theory, by going backwards and defining each of the terms used in that definition until you get to very basic concepts, like sets, functions, and equations.

The extremely important corollary to all of this is that if you feel like your understanding of a definition does *not* reach this level, YOU ARE CONFUSED. This is OK. It is good to be confused. It is far better to be confused and not yet wrong than it is to be unconfused and wrong.

What should you do when you are confused? First of all, GO BACKWARDS. Read the previous section of a textbook. If you are still confused, keep going backwards until you hit something that makes sense, and then work your way back up. Secondly, GO SIDEWAYS. Read another textbook that treats the same material in a different way. Then finally, if neither of those work, ASK AN EXPERT, and keep asking until you feel unconfused. The MOST IMPORTANT SKILL in math is to know when you are confused and don't continue until you are unconfused! If you continue on, you will get *hopelessly* confused; if you turn back immediately there is still hope.

Definitions are the most important part of higher math. Understanding the definitions is often half the battle, and it is most of the battle for category theory.

A **proposition** is an assertion that one logical statement (the conclusion) follows from several logical statements (the premises). Each proposition comes along with a **proof**. Just like definitions, it is expected that the participants in a mathematical conversation could expand a proof out to a fully rigorous level, even if the given proof is very vague. What you write down as a proof should be seen as a "hint" for the construction of the actual, fully rigorous proof; mathematicians come to cultural agreements for how much of a hint is needed in different circumstances.

As a mathematical learner, proofs are your window into the thought processes of subject experts. Thus, they are very good to study and understand. However, they are not as critical to understand as definitions. It is absolutely essential to fully understand definitions, but proofs can be "blackboxed" sometimes, and you can just remember the proposition without understanding fully the proof.

Propositions are also known as **theorems**, **lemmas**, and **corollaries**. A theorem is an important proposition, a lemma is a small proposition mainly used to prove other propositions, and a corollary is a proposition whose proof is easy because of another proposition, for example a special case. Really, these are just vibes that mathematicians add to propositions.

Finally, an **example** is a definition or proposition that falls under one of three categories.

1. The purpose is mainly pedagogical; the example gives you intuition for another definition or proposition.
2. The purpose is mainly application-oriented; the example shows you how to apply something abstract in the real world.
3. It's a normal proposition or definition, except it's slightly less abstract than some previous proposition or definition. This is often the case in category theory; category theory is known as the subject where "the examples have examples."

A pure math document consists of a sequence of definitions, propositions, and examples, punctuated with interleaving prose that attempts to give intuition for *what* the definitions, propositions, and examples are saying, and *why* one should care about those definitions, propositions, and examples. Intuition is a very important part of math; it is what allows mathematicians to elaborate definitions, discover proofs, and most importantly, to figure out what is important to study in the first place. However, intuition is no substitute for rigor. Intuition allows you to leap off cliffs; rigor is what allows you to build a bridge underneath you before you hit the ground.

In the foundations of math, we also have two more types of statement: **axioms** and **undefined terms**. Definitions and propositions should always "bottom out" at axioms and undefined terms. However, most mathematicians do *not* do this, instead leaving it to the reader to choose a suitable foundations of math in which to fully formalize their theories. Surprisingly, most interesting math can be fully formalized in many foundations, so this normally works out fine.

For us, our "reality" will be what happens on the computer. So we will try to "bottom out" on concepts in the computer.

Takeaways

- Pure math consists of definitions, propositions, and examples.
- These are specified in enough detail so that participants in a mathematical conversation could independently come up with equivalent elaborations.
- If you feel you could *not* do this at any point, then you are in a perfectly normal situation and should not feel ashamed whatsoever. However, continuing on without first going back and understanding what you are confused about is a bad idea.

Finite Sets

We will now demonstrate the previous concepts by studying finite sets. We will not get to category theory today. Instead, we will revisit some things that should be familiar to you and treat them in the style that we will be using for the rest of the lectures.

Finite sets will be important for most of the applications of category theory that we will learn in the next lectures, and also most of the concepts of category theory are well-illustrated by finite sets. So a firm grasp of finite sets will be an immense aid in the coming weeks.

We start with a basic universe of discourse. It is traditional to be minimalistic with this universe of discourse, and say that everything is a set, or everything is a natural number. However, we choose to be untraditional.

A **primitive thing** is any possible value of a Julia variable.

`1`, `:a`, `[1.0 0; 0 1]` are all primitive things

A **finite set** is a list of primitive things. We typically write curly braces around this list.

$\{1, 2, 3, 4\}$, $\{\mathbf{a}, \mathbf{b}, 4, 2, 2, 6\}$ are both finite sets.

We might represent this in Julia with the following data structure.

```
abstract type   end

struct Vec  <:
  elems::Vector{Any}
end

Base.:( )(a, A::Vec ) = a   A.elems

Base.iterate(A::Vec ) = iterate(A.elems)
Base.iterate(A::Vec , k) = iterate(A.elems, k)
```

```
A = Vec ([:carrots, :peas])
B = Vec ([3, 7, 8])
```

Note that this is *not the only possible representation of a finite set.* Definitions in math always can be translated into code in many ways; the choice of a particular way is a delicate balancing act between simplicity, performance, clarity, and completeness.

Another possible representation of finite sets is

```
struct Int <:
  n::Int
end

Base.:( )(a, A::Int ) = 1 <= a <= A.n

Base.iterate(A::Int ) = iterate(1:A.n)
Base.iterate(A::Int , k) = iterate(1:A.n, k)
```

This represents the finite set $\{1, \ldots, n\}$. Here we have traded performance over completeness; we can only represent some finite sets, but we represent those finite sets more efficiently.

A **function** f from a finite set A to a finite set B is something that associates a value $f(a)$ in B to every thing a in A. A is called the **domain** of f, and B is called the **codomain** of f. We write f along with its domain and codomain as $f\colon A \to B$.

It is important to note that even if a is listed multiple times in A, $f(a)$ only has one value.

We might represent a function between s, also known as a morphism of finite sets, with the following data structure.

```
struct  Mor
  dom::
  codom::
  vals::Dict{Any,Any}
end

(f:: Mor)(x) = f.vals[x]
```

```
f =  Mor(A, B, Dict(:carrots => 3, :peas => 3))
```

However, not all instances of this data structure represent functions. The following Julia function determines whether a morphism of finite sets is valid.

```
isvalid(f:: Mor) =
  all(x   keys(f.vals) && f(x)   f.codom for x in f.dom)
```

```
[
  isvalid(f),
  isvalid( Mor(A, B, Dict(:peas => 3))),
  isvalid( Mor(A, B, Dict(:peas => 5, :carrots => 3)))
]
```

```
3-element Vector{Bool}:
 1
 0
 0
```

Frequently we will write down Julia types representing mathematical concepts where not all values of that type are valid representations of that mathematical concept. This is unavoidable because Julia types do not have the specificity to narrow down the space of values enough. There are languages where the types can narrow down the space of values sufficiently, but none of those languages have well-maintained BLAS/LAPACK bindings.

For finite sets implemented by `Int` , we can give a more efficient encoding of morphism.

```
struct Int Mor
  dom::Int
  codom::Int
  vals::Vector{Int}
end

(f::Int Mor)(i) = f.vals[i]
```

with corresponding validation function

```
isvalid(f::Int Mor) = length(f.vals) == f.dom.n &&
  all(f(x)   f.codom for x in f.dom)
```

```
isvalid(Int Mor(Int (3), Int (2), [1,2,2]))
```

```
true
```

From now on, we will work with only `Vec` s and `Mor`s, and leave the implementation of more efficient code to the reader.

Takeaways

- A finite set is a list of things
- A morphism of finite sets from A to B sends each unique element of A to an element of B
- There are multiple ways of implementing representations of finite sets and morphisms between them on the computer

Composition and Isomorphisms

We now introduce the operation at the core of all category theory: composition!

Given finite sets A, B, C and morphisms $f\colon A \to B$ and $g\colon B \to C$, their **composite** $g \circ f$ is a morphism from A to C defined by

$$(g \circ f)(a) = g(f(a))$$

for all a in A. We call $\circ$ the operation of **composition**, and we say that we **compose** f and g to form $g \circ f$.

```
function compose(f::Mor, g::Mor)
  @assert f.codom == g.dom
   Mor(f.dom, g.codom, Dict(a => g(f(a)) for a in f.dom))
end
```

```
A,B,C = Vec([1,2,3]), Vec([:a,:b]), Vec(["s", "t"])
f = Mor(A,B,Dict(1=>:a,2=>:a,3=>:b))
g = Mor(B,C,Dict(:a=>"t",:b=>"s"))
compose(f,g).vals
```

```
Dict{Any, Any} with 3 entries:
  2 => "t"
  3 => "s"
  1 => "t"
```

We now come to an issue that is *everywhere* in category theory: equality. If you have seen any set theory before, you might think that $\{1, 2, 3, 3\}$ and $\{3, 2, 1\}$ are "the same" set. Note however that when I defined finite set, I just said that a finite set was a list of things surrounded by curly braces. Additionally,

```
A, A = Vec([1,2,3,3]), Vec([3,2,1])
A == A
```

```
false
```

So it seems like something is wrong with our definition. One way one might try to fix this would be by using `Set{Any}` instead of `Vector{Any}`. This certainly makes more of our Julia values equal.

```
struct Set
  elems::Set{Any}
end

Base.:( )(a, A::Vec ) = a   A.elems

Base.iterate(A::Vec ) = iterate(A.elems)
Base.iterate(A::Vec , k) = iterate(A.elems, k)
```

But let's think about why we typically choose to make $\{1,2,3,3\}$ and $\{3,2,1\}$ "the same" set. One good reason is that any morphism out of $\{1,2,3,3\}$ can be seen as a morphism out of $\{3,2,1\}$, and the same goes for incoming morphisms.

```
convertAA (f:: Mor) =  Mor(A ,f.codom,f.vals)
convertA A(f:: Mor) =  Mor(A,f.codom,f.vals)
```

If `f` is a valid `Mor` with `f.dom == A` then `f  = convertAA (f)` is a valid `Mor` with `f .dom == A` , and the same the other way around. Moreover, `convertA A(convertAA (f)) == f` and `convertAA (convertA A(f )) == f` . We could define similar functions for morphisms into `A` and `A` .

Therefore, from a "morphism's-eye" perspective, `A` and `A` behave the exact same way; defining a morphism in or out of `A` is equivalent to defining a morphism in or out of `A` .

But if we take this to its logical conclusion, we find that this is true not only of $\{1,2,3,3\}$ and of $\{3,2,1\}$, but also of $\{1,2,3\}$ and $\{\mathbf{a},\mathbf{b},\mathbf{c}\}$! Specifically, we can do the following.

```
B, B  = Vec ([1,2,3]), Vec ([:a,:b,:c])
f =  Mor(B, B , Dict(1 => :a, 2 => :b, 3 => :c))
g =  Mor(B , B, Dict(:a => 1, :b => 2, :c => 3))
convertBB (h:: Mor) = compose(g,h)
convertB B(h:: Mor) = compose(f,h)
```

We can use `f` and `g` to convert freely between morphisms out of `B` and morphisms out of `B` . Thus, `B` and `B` also are equivalent in some sense. But there are many ways of "forming" this equivalence; we could use `f.vals = Dict(1 => :b, 2 => :a, 3 => :c)`, for instance.

All of this motivates the next few definitions.

The **identity function** 1_A on a finite set A has domain and codomain A and is defined by $1_A(a) = a$.

```
identity(A::) = Mor(A, A, Dict(a => a for a in A))
```

We say that two finite sets A and A' are **isomorphic** if there exists morphisms $f\colon A \to A'$ and $g\colon A' \to A$ such that $g \circ f = 1_A$ and $f \circ g = 1_{A'}$. Moreover, if this is the case we call f an **isomorphism** from A to A', and g an isomorphism from A' to A, and we say that g is the **inverse** of f.

You can remember the reasoning about why we care isomorphism means by thinking "isomorphic = same morphisms". That is, if we have an isomorphism between A and A', then there are "the same morphisms" out of A and out of A'.

However, as noted before, there might be several distinct isomorphisms between A and A'. Thus, one must be careful to specify *which* isomorphism when you are talking about isomorphic finite sets.

Anyways, this is why it's not a big problem to use a vector to represent a finite set. There are only rare cases where you can find a representation of your mathematical objects such that two representations are equal if and only if the mathematical objects are isomorphic. If you are lucky enough to find this, it's called a "canonical form" and it's a big deal. As a practical matter, we might use `Set` instead of `Vector` because it gets a bit *closer* to a canonical form, but I wanted to start with `Vector` to make the point that the *representation* of your mathematical object on a computer in general *will not* be canonical.

We now prove a theorem about isomorphisms of finite sets.

The **cardinality** of a finite set is the number of unique elements listed in that finite set.

If two sets have the same cardinality, then they are isomorphic.

We prove this with a program.

```
function find_isomorphism(A::, B::)
  A_vec, B_vec = unique!.([Any[A...], Any[B...]])

  @assert length(A_vec) == length(B_vec)
  n = length(A_vec)

   Mor(A, B, Dict(A_vec[i] => B_vec[i] for i in 1:n))
end
```

```
find_isomorphism(Vec([:c, :b, :a, :b]), Int(3)).vals
```

```
Dict{Any, Any} with 3 entries:
  :a => 3
  :b => 2
  :c => 1
```

Takeaways

- You can compose morphisms between finite sets
- Isomorphisms tell you which finite sets are equivalent from the point of view of morphisms
- You can tell which finite sets are isomorphic by looking at their cardinalities

The Pigeonhole Principle

The point of this first lecture is to introduce you to finite sets and pure math, not category theory. Therefore, we end with a discussion of a theorem from combinatorics (come back after lecture 2).

If X and Y are finite sets, and X has a larger cardinality than Y, then for any morphism $f\colon X \to Y$, there exist distinct elements $x_1, x_2 \in X$ such that $f(x_1) = f(x_2)$.

Proof. Suppose that no such x_1, x_2 existed. Then if $X = \{x_1, \dots, x_n\}$ with $x_i \neq x_j$ for all i, j, we have $f(x_1), \dots, f(x_n)$ all distinct. Therefore, there are at least n distinct elements of Y, which contradicts the statement that Y has a smaller cardinality than X. We are done.

□

Suppose that $x_1, \dots, x_5$ are points on a sphere. I.e., each x_i is a 3-dimensional vector with distance from the origin 1. Then there is a hemisphere that contains 4 of the points.

Proof. Pick any two points. These two points determine a great circle of the sphere. Then at least two of the other three points must fall on one of the hemispheres determined by that great circle.

□

We can implement the pigeonhole principle in Julia.

```
function pigeonhole(f::Mor)
  @assert length(unique!([f.dom...])) > length(unique!([f.codom...]))
  holes = Dict(y => Any[] for y in f.codom)
  for pigeon in f.dom
    push!(holes[f(pigeon)], pigeon)
  end
  for hole in holes
    if length(hole) > 1
      return hole
    end
  end
```

```
end
```

There isn't much else to this part, but hopefully this has served as a more concrete illustration of how finite sets work in Julia.

Takeaways

- You can do basic combinatorics in the framework we have developed in this lecture

Lecture 2: Category Theory

Welcome to the second lecture in this series. I'm not going to repeat the introduction from the first lecture; you should know what to expect now. Specifically, it's going to go slowly. We start with (not-necessarily finite) sets.

Sets

A **set** X is a function from `Any` to `Bool`, which may or either be written in Julia or defined mathematically. If $X(x) =$ **true**, we write $x \in X$, and if $X(x) =$ **false** we write $x \notin X$.

Note that when we write down Julia definitions involving sets, we are implicitly assuming that the functions are written in Julia. However, there are some sets that we will use whose functions *cannot* be written down in Julia, so take the Julia definitions with a grain of salt.

For any finite set `A::` , the function `inA(x) = x  A` is a set.

Any Julia type `T` defines a set, via the function `inT(x) = x isa T`.

Given two sets X and Y, their **intersection** $X \cap Y$ is defined by

$$(X \cap Y)(x) = X(x) \wedge Y(x)$$

Their **union** $X \cup Y$ is defined by

$$(X \cup Y)(x) = X(x) \vee Y(x)$$

```
intersect(X,Y) = x -> X(x) && Y(x)
union(X,Y) = x -> X(x) || Y(x)
```

Given two sets X and Y, their **product** $X \times Y$ is the set of tuples (x, y) where $x \in X$ and $y \in Y$. That is, $z \in X \times Y$ if and only if `z isa Tuple`, `length(z) = 2`, $z[1] \in X$ and $z[2] \in Y$.

```
product(X,Y) = z -> (z isa Tuple) &&
  length(z) == 2 && X(z[1]) && Y(z[2])
```

We leave it to the reader to give a mathematical definition for **sum** given the following definition in Julia

```
struct Left
  val::Any
end
```

```
struct Right
  val::Any
end

sum(X,Y) = x ->
  if x isa Left
    X(x.val)
  elseif x isa Right
    Y(x.val)
  else
    false
  end
```

Given two sets A and B, the set $A \to B$ consists of all Julia callables f such that $f(a) \in B$ for all $a \in A$.

Note that even if `A` and `B` are Julia functions, there is no way to check in Julia that a given `f` is an element of $A \to B$ because this would involve iterating through possibly infinitely many elements of A. Again, languages where this is not the case don't have good ODE solvers.

Before we move on to categories, we briefly discuss some things that are not sets. There are some "collections of stuff" that are not sets by the definition we gave above. For instance, there is no "set of all sets", because not all sets are expressed by computable functions. Nevertheless, we will sometimes make reference to these collections that are "too large" to be called sets, and we will call them "classes".

Categories

The moment we've all be waiting for.

A **small category** C consists of

- a set C_0 of **objects**
- for every $x, y \in C_0$, a set $\mathrm{Hom}_C(x, y)$ of **morphisms** from x to y
- for every $x, y, z \in C_0$, a **composition function** $\circ \colon \mathrm{Hom}_C(y, z) \times \mathrm{Hom}_(x, y) \to \mathrm{Hom}(x, z)$
- for every $x \in C_0$, an **identity morphism** $1_x \in \mathrm{Hom}_C(x, x)$

such that

- for all $x, y, z, w \in C_0$, $f \in \mathrm{Hom}_C(x, y)$, $g \in \mathrm{Hom}_C(y, z)$, $h \in \mathrm{Hom}_C(z, w)$,

$$h \circ (g \circ f) = (h \circ g) \circ f$$

- for all $x, y \in C_0$, $f \in \mathrm{Hom}_C(x, y)$,

$$1_y \circ f = f = f \circ 1_x$$

These two laws are called the **associativity law** and **unitality law** respectively.

There is a category where the objects are finite sets and the morphisms are morphisms of finite sets, as defined in the previous lecture.

References